

Special Online Edition for UVA Online Judge Users

Art of Programming Contest

C Programming | Data Structures | Algorithms

2nd Edition

AHMED SHAMSUL AREFIN

FOREWORDED BY

MIGUEL A. REVILLA
UNIVERSITY OF VALLADOLID, SPAIN

REVIEWED BY

STEVEN HALIM
DR. M. LUTFAR RAHMAN

ACMSOLVER Training Series



An Essential Book for Programmers

ART OF PROGRAMMING CONTEST

C Programming Tutorials | Data Structures | Algorithms

Compiled by

Ahmed Shamsul Arefin

Graduate Student,
Institute of Information and Communication Technology
Bangladesh University of Engineering and Technology (BUET)
BSc. in Computer Science and Engineering, CUET

Reviewed By

Steven Halim

School of Computing, National University of Singapore
Singapore.

Dr. M. Lutfar Rahman

Professor, Department of Computer Science and Engineering
University of Dhaka.

Foreworded By

Professor Miguel A. Revilla

ACM-ICPC International Steering Committee Member and Problem Archivist
University of Valladolid,
Spain.

<http://acmicpc-live-archive.uva.es>

<http://online-judge.uva.es>



Gyankosh Prokashoni, Bangladesh

ISBN 984-32-3382-4

DEDICATED TO

Shahriar Manzoor

Judge ACM/ICPC World Finals 2003-2006

(Whose mails, posts and problems are invaluable to all programmers)

And

My loving parents and colleagues

ACKNOWLEDGEMENTS

I would like to thank following people for supporting me and helping me for the significant improvement of my humble works. Infact, this list is still incomplete.

<i>Professor Miguel A. Revilla</i>	<i>University of Valladolid, Spain.</i>
<i>Dr. M Kaykobad</i>	<i>North South University, Bangladesh</i>
<i>Dr. M. Zafar Iqbal</i>	<i>Shahjalal University of Science and Technology, Bangladesh</i>
<i>Dr. M. Lutfar Rahman</i>	<i>University of Dhaka, Bangladesh</i>
<i>Dr. Abu Taher</i>	<i>Daffodil International University</i>
<i>Howard Cheng</i>	<i>University of Lethbridge, Canada</i>
<i>Steven Halim</i>	<i>National University of Singapore, Singapore</i>
<i>Shahriar Manzoor</i>	<i>South East University, Bangladesh</i>
<i>Carlos Marcelino Casas Cuadrado</i>	<i>University of Valladolid, Spain</i>
<i>Mahbub Murshed Suman</i>	<i>Arizona State University, USA</i>
<i>Salahuddin Mohammad Masum</i>	<i>Daffodil International University</i>
<i>Samiran Mahmud</i>	<i>Dhaka University of Engineering and Technology</i>
<i>M H Rasel</i>	<i>Chittagong University of Engineering and Technology</i>
<i>Sadiq M. Alam</i>	<i>National University of Singapore, Singapore</i>
<i>Mehedi Bakht</i>	<i>Bangladesh University of Engineering and Technology</i>
<i>Ahsan Raja Chowdhury</i>	<i>University of Dhaka</i>
<i>Mohammad Rubaiyat Ferdous Jewel</i>	<i>University of Toronto, Canada</i>
<i>KM Hasan</i>	<i>North South University</i>
<i>Monirul Islam Sharif</i>	<i>Georgia Institute of Technology, USA</i>
<i>Gahangir Hossain</i>	<i>Chittagong University of Engineering and Technology</i>
<i>S.M Saif Shams</i>	<i>Shahjalal University of Science and Technology</i>
<i>Shah Md. Shamsul Alam</i>	<i>Daffodil International University</i>



Author's Biography: Ahmed Shamsul Arefin is completing his Masters from **Bangladesh University of Engineering & Technology (BUET)** and has completed BSc. in Computer Science and Engineering from **CUET**. In Computer Science and Engineering. He participated in the 2001 ACM Regional Contest in Dhaka, and his team was ranked 10th. He became contest organizer at **Valladolid online judge** by arranging "Rockford Programming Contest 2001" and local Contest at several universities. His Programming Contest Training Website "**ACMSolver.org**" has been linked with ACM Uva, USU and Polish Online Judge – Sphere.

His research interests are **Contests, Algorithms, Graph Theory** and **Web-based applications**. His Contact E-mail : asarefin@yahoo.com Web: <http://www.daffodilvarsity.edu.bd/acmsolver/asarefin/>

Preface to 2nd Edition

I am happy to be able to introduce the 2nd Edition of this book to the readers. The objective of this edition is not only to assist the contestants during the contest hours but also describing the core subjects of **Computer Science** such as **C Programming**, **Data Structures** and **Algorithms**. This edition is an improvement to the previous edition. Few more programming techniques like **STL (Standard Template Library)**, manipulating strings and handling mathematical functions are introduced here.

It is hoped that the new edition will be welcomed by all those for whom it is meant and this will become an **essential book for Computer Science students**.

Preface to 1st Edition

Why do programmers love Programming Contest? Because young computer programmers like to battle for fame, money, and they love algorithms. The first ACM-ICPC (International Collegiate Programming Contest) Asia Regional Contest Bangladesh was held at North South University in the year 1997. Except the year 2000, our country hosted this contest each year and our invaluable programmers have participated the world final every year from 1997.

Our performance in ACM/ICPC is boosting up day by day. The attention and time we are spending on solving moderate and difficult problems is noticeable. BUET, University of Dhaka, NSU and AIUB has produced many programmers who fought for World Finals. Institutions looking for boosting the performance of their teams in the programming contests may consider them as prospective coaches/trainers. Some universities have recently adopted another strategy. They are offering 1-credit courses for students interested in improving their problem-solving and programming skills.

I am very much grateful to our mentors, **Dr. M Kaykobad** who was honored with the “Best Coach” award in the World Finals in Honolulu. Under his dynamic presence our country teams became champion several times in the ACM/ICPC Asia Regional. **Dr. M. Zafar Iqbal**, Chief Judge of our ACM/ICPC Regional Contests. **Dr. Abul L Haque**, who first contacted **Dr. C.J. Hwang** (Asia Contests Director and Professor at Texas State University, San Marcos, USA) and wanted to have a n ACM/ICPC regional site at Dhaka back in 1997. Also a big thank should go to **Mr. Shahriar Manzoor**, our renown Problem Setter, Judging Director for ACM/ICPC Regional (Dhaka Site) and World Final Judge and Problem Setter. I would like to thank him personally because, he showed me the right way several times when I was setting problems for Valladolid Online Judge in “Rockford Programming Contest 2001” and while developing my Programming Contest Training Site “ACMSolver.org”.

Thanks to **Professor Miguel A. Revilla**, University of Valladolid, Spain for linking my ACMSolver (<http://www.acmsolver.org>) site with his world famous Valladolid Online Judge (<http://acm.uva.es/p>) and making me ACM Valladolid Online Judge Algorithmic Team Member for helping them to add some problems at live archive.

And also invaluable thanks to **Steven Halim**, a PhD Student of NUS, Singapore for the permission of using his website (<http://www.comp.nus.edu.sg/~stevenha/>) contents. A major part of this book is compiled from his renowned website. Of course, it is mentionable that his website is based upon USACO Training page located at (<http://ace.delos.com/>)

I am grateful to **Daffodil International University**, especially to honorable Vice-Chancellor **Professor Aminul Islam** and Dean, Faculty of Science and Informaion Technology **Dr. M. Lutfar Rahman** and all my colleagues at **Department of Computer Science and Engineering** here, for providing me the golden opportunity of doing something on ACM Programming Contest and other researches.

Furthermore, since this project is a collection of tutorials from several sources so all the authors of tutorials are acknowledged in the **Reference** section of this book. Tracking down the original authors of some of these tutorials is much difficult. I have tried to identify case by case and in each case asked permission. I apologize in advance if there are any oversights. If so, please let me know so that I can mention the name in future edition.

Finally I would like to add a line at the end of this preface, for last few years while making and maintaining my site on ACM Programming Contest, I have got few experiences. I felt that there should be some guideline for beginners to enter into the world of programming. So, I started collecting tutorials and compiling them to my site. Furthermore, this is another attempt to make Programming Contest in our country, as I have tried to put all my collections in a printed form. Your suggestions will be cordially accepted.

Best regards,

Ahmed Shamsul Arefin.



**UNIVERSIDAD DE
VALLADOLID**

Foreword Note

As the main responsible of the University of Valladolid Online Judge I has the feeling that this book is not only a recollection of tutorials as the author says in the preface, but also will be an essential part of the help sections of the UVa site, as it put together a lot of scattered information of the Online Judge, that may help to many programmers around the world, mainly to the newcomers, what is very important for us. The author proves a special interest in guiding the reader, and his tips must be considered almost as orders, as they are a result of a great experience as solver of problems as well as a problemsetter. Of course, the book is much more than an Online Judge user manual and contains very important information missing in our web, as the very interesting classification of a lot of problems by categories, that analyze in detail and with examples. I think it is a book all our users should be allowed to access to, as is a perfect complement to our Online Judge.

Miguel A. Revilla

**ACM-ICPC International Steering Committee Member and Problem Archivist
University of Valladolid, Spain.**

<http://acmicpc-live-archive.uva.es>

<http://online-judge.uva.es>

University Of Dhaka



Review Note

A Computer programming contest is a pleasurable event for the budding programmers, but only a few books are available as a training manual for programming competitions.

This book is designed to serve as a textbook for an algorithm course focusing on programming as well as a programming course focusing on algorithms. The book is specially designed to train students to participate in competitions such as the ACM International Collegiate Programming Contest.

The book covers several important topics related to the development of programming skills such as, fundamental concepts of contest, game plan for a contest, essential data structures for contest, Input/output techniques, brute force method, mathematics, sorting, searching, greedy algorithms, dynamic programming, graphs, computational geometry, Valladolid Online Judge problem category, selected ACM programming problems, common codes/routines for programming, Standard Template Library (STL), PC² contest administration and team guide. The book also lists some important websites/books for ACM/ICPC Programmers.

I believe that the book will be of immense use for young programmers interested in taking part in programming competitions.

A handwritten signature in black ink, appearing to read 'M. Lutfar Rahman'.

Dr. M. Lutfar Rahman
Professor, Department of Computer Science and Engineering (CSE)
University of Dhaka.
Bangladesh.



Notes from Steven Halim

When I created my own website World of Seven few years back (<http://www.comp.nus.edu.sg/~stevenha>), my aim was to promote understanding of data structures and algorithms especially in the context of programming contest and to motivate more programmers to be more competitive by giving a lot of hints for many University of Valladolid (UVa) Online Judge problems. However, due to my busyness, I never managed to set aside a time to properly publicize the content of my website in a book format. Thus, I am glad that Ahmed compiled this book and he got my permission to do so. Hopefully, this book will be beneficial for the programmers in general, but especially to the Bangladeshi programmers where this book will be sold.

A handwritten signature in black ink, appearing to be "SH", with a horizontal line extending to the right from the end of the signature.

Steven Halim
National University of Singapore (NUS)
Singapore.

Contents

Chapter 1	Fundamental Concepts	14
Chapter 2	Game Plan For a Contest	19
Chapter 3	Programming In C: a Tutorial	27
Chapter 4	Essential Data Structures for Contest	72
Chapter 5	Input/Output Techniques	81
Chapter 6	Brute Force Method	85
Chapter 7	Mathematics	91
Chapter 8	Sorting	106
Chapter 9	Searching	113
Chapter 10	Greedy Algorithms	117
Chapter 11	Dynamic Programming	121
Chapter 12	Graphs	134
Chapter 13	Computational Geometry	172
Chapter 14	Valladolid OJ Problem Category	174
Appendix A	ACM Programming Problems	176
Appendix B	Common Codes/Routines For Programming	188
Appendix C	Standard Template Library (STL)	230
Appendix D	PC ² Contest Administration And Team Guide	235
Appendix E	Important Websites/Books for ACM Programmers	242



acm International Collegiate
Programming Contest

IBM | event
sponsor

What is the ACM Programming Contest?

The Association for Computing Machinery (ACM) sponsors a yearly programming contest, recently with the sponsorship of IBM. The contest is both well-known and highly regarded: last year 2400 teams competed from more than 100 nations competed at the regional levels. Sixty of these went on to the international finals. This contest is known as **ACM International Collegiate Programming Contest (ICPC)**.

The regional contest itself is typically held in November, with the finals in March. Teams of three students use C, C++, or Java to solve six to eight problems within five hours. One machine is provided to each team, leaving one or two team members free to work out an approach. Often, deciding which problems to attack first is the most important skill in the contest. The problems test the identification of underlying algorithms as much as programming savvy and speed.

CHAPTER 1 FUNDAMENTAL CONCEPTS

Programming Contest is a delightful playground for the exploration of intelligence of programmers. To start solving problems in contests, first of all, you have to fix your aim. Some contestants want to increase the number of problems solved by them and the other contestants want to solve less problems but with more efficiency. Choose any of the two categories and then start. A contestant without any aim can never prosper in 24 hours online judge contests. So, think about your aim.^[1]

If you are a beginner, first try to find the easier problems. Try to solve them within short time. At first, you may need more and more time to solve even simple problems. But do not be pessimistic. It is for your lack of practice. Try to solve easier problems as they increase your programming ability. Many beginners spend a lot of time for coding the program in a particular language but to be a great programmer you should not spend more times for coding, rather you should spend more time for debugging and thinking about the algorithm for the particular problem. A good programmer spends 10% time for coding and 45% time for thinking and the rest of the time for debugging. So to decrease the time for coding you should practice to solve easier problems first.

Do not try to use input file for input and even any output file for output when sending the program to online judges. All input and output parts should be done using standard input and outputs. If you are a C or C++ programmer try this, while coding and debugging for errors add the lines at the first line of the main procedure i.e.

```
#include <stdio.h>
main ()
{
freopen("FILE_NAME_FOR_INPUT", "r", stdin);
freopen("FILE_NAME_FOR OUTPUT", "w", stdout);
Rest of the codes...
return 0;}
```

But while sending to online judges remove the two lines with `freopen` to avoid restricted function errors. If you use the first `freopen` above, it will cause your program to take input from the file "FILE_NAME_FOR_INPUT". Write down the inputs in the file to avoid entering input several times for debugging. It saves a lot of time. But as the function opens input file which can be a cause of hacking the websites of online judges they don't allow using the function and if you use it they will give compilation error (Restricted Function). The second `freopen` is for generating the output of your program in a specified file named "FILE_NAME_FOR_OUTPUT" on the machine. It is very helpful when the output can't be justified just viewing the output window (Especially for String Manipulation Problem where even a single space character can be a cause of

Wrong answer). To learn about the function more check Microsoft Developer Network (MSDN Collection) and C programming helps.

Programming languages and dirty debugging

Most of the time a beginner faces this problem of deciding which programming language to be used to solve the problems. So, sometimes he uses such a programming language which he doesn't know deeply. That is why; he debugs for finding the faults for hour after hour and at last can understand that his problem is not in the algorithm, rather it is in the code written in that particular language. To avoid this, try to learn only one programming language very deeply and then to explore other flexible programming languages. The most commonly used languages are C, C++, PASCAL and JAVA. Java is the least used programming language among the other languages. Avoid dirty debugging.

Avoid Compilation Errors

The most common reply to the beginner from 24 hours online judge is COMPILATION ERROR (CE). The advices are,

- 1) When you use a function check the help and see whether it is available in Standard Form of the language. For example, do not use `strev` function of `string.h` header file of C and C++ as it is not ANSI C, C++ standard. You should make the function manually if you need it. Code manually or avoid those functions that are available in your particular compiler but not in Standard Form of the languages.
- 2) Don't use input and output file for your program. Take all the inputs for standard input and write all the outputs on standard output (normally on the console window). Check my previous topics.
- 3) Do not use `conio.h` header file in C or C++ as it is not available in Standard C and C++. Usually don't use any functions available in `<conio.h>` header file. It is the great cause of Compilation Error for the programmers that use Turbo C++ type compiler.
- 4) built-in functions and packages are not allowed for using in online judge.
- 5) Don't mail your program i.e. don't use yahoo, hotmail etc. for sending your program to judge as it is a complex method—write judge id, problems number etc. Rather use submit-system of the online judge for example, Submit page of Valladolid. Using the former will give you CE most of the time as they include there advertisements at the beginning and end of your program. So the judge can't recognize the extra characters concatenated in your sent code and gives you CE. About 90% CE we ever got is for this

reason. The mail system also breaks your programs into several lines causing Wrong Answer or Other Errors though your program was correct indeed.

There are many famous online judges that can judge your solution codes 24 hours. Some of them are:

- ▲ Valladolid OJ (<http://acm.uva.es/p>)
- ▲ Ural OJ (<http://acm.timus.ru>)
- ▲ Saratov OJ (<http://acm.sgu.ru>)
- ▲ ZJU OJ (<http://acm.zju.edu.cn>)
- ▲ Official ACM Live Archive (<http://cii-judge.baylor.edu/>)
- ▲ Peking University Online Judge (<http://acm.pku.edu.cn/JudgeOnline/>)
- ▲ Programming Challenges (<http://www.programming-challenges.com>)

Forget Efficiency and start solving easier problems

Sometimes, you may notice that many programmers solved many problems but they made very few submissions (they are geniuses!). At first, you may think that I should try to solve the problems as less try as possible. So, after solving a problem, you will not want to try it again with other algorithm (may be far far better than the previous algorithm you used to solve that problem) to update your rank in the rank lists. But my opinion is that if you think so you are in a wrong track. You should try other ways as in that and only that way you can know that which of the algorithms is better. Again in that way you will be able to know about various errors than can occur. If you don't submit, you can't know it. Perhaps a problem that you solved may be solved with less time in other way. So, my opinion is to try all the ways you know. In a word, if you are a beginner forget about efficiency.

Find the easier problems. Those problems are called ADHOC problems. You can find the list of those problems available in 24 OJ in S. Halim's, acmbeginner's, acmsolver's websites. Try to solve these problems and in that way you can increase your programming capability.

Learn algorithms

Most of the problems of Online Judges are dependent on various algorithms. An algorithm is a definite way to solve a particular problem. If you are now skilled in coding and solving easier problems, read the books of algorithms next. Of course, you should have a very good mathematical skill to understand various algorithms. Otherwise, there is no other way but just to skip the topics of the books. If you have skill in math, read the algorithms one by one, try to understand. After understanding the algorithms, try to write it in the programming language you have learnt (This is because, most of the

algorithms are described in Pseudocode). If you can write it without any errors, try to find the problems related to the algorithm, try to solve them. There are many famous books of algorithms. Try to make modified algorithm from the given algorithms in the book to solve the problems.

Use simple algorithms, that are guaranteed to solve the problem in question, even if they are not the optimum or the most elegant solution. Use them even if they are the most stupid solution, provided they work and they are not exponential. You are not competing for algorithm elegance or efficiency. You just need a correct algorithm, and you need it now. The simplest the algorithm, the more the chances are that you will code it correctly with your first shot at it.

This is the most important tip to follow in a programming contest. You don't have the time to design complex algorithms once you have an algorithm that will do your job. Judging on the size of your input you can implement the stupidest of algorithms and have a working solution in no time. Don't underestimate today's CPUs. A for loop of 10 million repetitions will execute in no time. And even if it takes 2 or 3 seconds you needn't bother. You just want a program that will finish in a couple of seconds. Usually the timeout for solutions is set to 30 seconds or more. Experience shows that if your algorithm takes more than 10 seconds to finish then it is probably exponential and you should do something better.

Obviously this tip should not be followed when writing critical code that needs to be as optimized as possible. However in my few years of experience we have only come to meet such critical code in device drivers. We are talking about routines that will execute thousands of times per second. In such a case every instruction counts. Otherwise it is not worth the while spending 5 hours for a 50% improvement on a routine that takes 10 milliseconds to complete and is called whenever the user presses a button. Nobody will ever notice the difference. Only you will know.

Simple Coding

1. Avoid the usage of the ++ or -- operators inside expressions or function calls. Always use them in a separate instruction. If you do this there is no chance that you introduce an error due to post-increment or pre-increment. Remember it makes no difference to the output code produced.
2. Avoid expressions of the form *p++.
3. Avoid pointer arithmetic. Instead of (p+5) use p[5].
4. Never code like :

```
return (x*y)+Func(t)/(1-s);
```

but like :

```
temp = func(t);  
RetVal = (x*y) + temp/(1-s);  
return RetVal;
```

This way you can check with your debugger what was the return value of Func(t) and what will be the return code of your function.

5. Avoid using the = operator. Instead of :

```
return ((x*8-111)%7)>5 ? y : 8-x;
```

Rather use :

```
Temp = ((x*8-111)%7);    if (5<Temp) return y; else return 8-x;
```

If you follow those rules then you eliminate all chances for trivial errors, and if you need to debug the code it will be much easier to do so.

- NAMING 1 : Don't use small and similar names for your variables. If you have three pointer variables don't name them p1, p2 and p3. Use descriptive names. Remember that your task is to write code that when you read it it says what it does. Using names like Index, RightMost, and Retries is much better than i, rm and rt. The time you waste by the extra typing is nothing compared to the gain of having a code that speaks for itself.
- NAMING 2 : Use hungarian naming, but to a certain extent. Even if you oppose it (which, whether you like it or not, is a sign of immaturity) it is of immense help.
- NAMING 3 : Don't use names like {i,j,k} for loop control variables. Use {I,K,M}. It is very easy to mistake a j for an i when you read code or "copy, paste & change" code, but there is no chance that you mistake I for K or M.

Last words

Practice makes a man perfect. So, try to solve more and more problems. A genius can't be built in a day. It is you who may be one of the first ten of the rank lists after someday. So, get a pc, install a programming language and start solving problem at once.^[1]

CHAPTER 2 GAME PLAN FOR A CONTEST

During a real time contest, teams consisting of three students and one computer are to solve as many of the given problems as possible within 5 hours. The team with the most problems solved wins, where "solved" means producing the right outputs for a set of (secret) test inputs. Though the individual skills of the team members are important, in order to be a top team it is necessary to make use of synergy within the team.^[2]

However, to make full use of a strategy, it is also important that your individual skills are as honed as possible. You do not have to be a genius as practicing can take you quite far. In our philosophy, there are three factors crucial for being a good programming team:

- ▲ Knowledge of standard algorithms and the ability to find an appropriate algorithm for every problem in the set;
- ▲ Ability to code an algorithm into a working program; and
- ▲ Having a strategy of cooperation with your teammates.

What is an Algorithm?

"A good algorithm is like a sharp knife - it does exactly what it is supposed to do with a minimum amount of applied effort. Using the wrong algorithm to solve a problem is trying to cut a steak with a screwdriver: you may eventually get a digestible result, but you will expend considerable more effort than necessary, and the result is unlikely to be aesthetically pleasing."

Algorithm is a step-by-step sequence of instructions for the computer to follow.

To be able to do something competitive in programming contests, you need to know a lot of well-known algorithms and ability to identify which algorithms is suitable for a particular problem (if the problem is straightforward), or which combinations or variants of algorithms (if the problem is a bit more complex).

A good and correct algorithm according to the judges in programming contests:

1. Must terminate
[otherwise: Time Limit/Memory Limit/Output Limit Exceeded will be given]
2. When it terminate, it must produce a correct output
[otherwise: the famous Wrong Answer reply will be given]
3. It should be as efficient as possible
[otherwise: Time Limit Exceeded will be given]

Ability to quickly identify problem types

In all programming contests, there are only three types of problems:

1. I haven't see this one before
2. I have seen this type before, but haven't or can't solve it
3. I have solve this type before

In programming contests, you will be dealing with a set of problems, not only one problem. The ability to quickly identify problems into the above mentioned contest-classifications (haven't see, have seen, have solved) will be one of key factor to do well in programming contests.

Mathematics	Prime Number
	Big Integer
	Permutation
	Number Theory
	Factorial
	Fibonacci
	Sequences
	Modulus
Dynmic Programming	Longest Common Subsequence
	Longest Increasing Subsequence
	Edit Distance
	0/1 Knapsack
	Coin Change
	Matrix Chain Multiplication
	Max Interval Sum
Graph	Traversal
	Flood Fill
	Floyed Warshal
	MST
	Max Bipertite Matching
	Network Flow
	Aritculation Point
Sorting	Bubble Sort
	Quick Sort
	Merge Sort (DAndC)
	Selection Sort
	Radix Sort
Searching	Bucket Sort
	Complete Search, Brute Force
	Binary Search (DAndC)
	BST
Simulation	Josephus
String Processing	String Matching
	Pattern Matching
Computational Geometry	Convex Hull
AdHoc	Trivial Problems

Sometimes, the algorithm may be 'nested' inside a loop of another algorithm. Such as binary search inside a DP algorithm, making the problem type identification not so trivial.

If you want to be able to compete well in programming contests, you must be able to know all that we listed above, with some precautions to ad-hoc problems.

'Ad Hoc' problems are those whose algorithms do not fall into standard categories with well-studied solutions. Each Ad Hoc problem is different... No specific or general techniques exist to solve them. This makes the problems the 'fun' ones (and sometimes frustrating), since each one presents a new challenge.

The solutions might require a novel data structure or an unusual set of loops or conditionals. Sometimes they require special combinations that are rare or at least rarely encountered. It usually requires careful problem description reading and usually yield to an attack that revolves around carefully sequencing the instructions given in the problem. Ad Hoc problems can still require reasonable optimizations and at least a degree of analysis that enables one to avoid loops nested five deep, for example.

Ability to analyze your algorithm

You have identified your problem. You think you know how to solve it. The question that you must ask now is simple: Given the maximum input bound (usually given in problem description), can my algorithm, with the complexity that I can compute, pass the time limit given in the programming contest.

Usually, there are more than one way to solve a problem. However, some of them may be incorrect and some of them is not fast enough. However, the rule of thumb is: **Brainstorm many possible algorithms - then pick the stupidest that works!**

Things to learn in algorithm

1. Proof of algorithm correctness (especially for Greedy algorithms)
2. Time/Space complexity analysis for non recursive algorithms.
3. For recursive algorithms, the knowledge of computing recurrence relations and analyze them: iterative method, substitution method, recursion tree method and finally, Master Theorem
4. Analysis of randomized algorithm which involves probabilistic knowledge, e.g: Random variable, Expectation, etc.
5. Amortized analysis.
6. Output-sensitive analysis, to analyze algorithm which depends on output size, example: $O(n \log k)$ LIS algorithm, which depends on k , which is output size not input size.

Table 1: Time comparison of different order of growth

We assume our computer can compute 1000 elements in 1 seconds (1000 ms)

Order of Growth	n	Time (ms)	Comment
$O(1)$	1000	1	Excellent, almost impossible for most cases
$O(\log n)$	1000	9.96	Very good, example: Binary Search
$O(n)$	1000	1000	Normal, Linear time algorithm
$O(n \log n)$	1000	9960	Average, this is usually found in sorting algorithm such as Quick sort
$O(n^2)$	1000	1000000	Slow
$O(n^3)$	1000	10^9	Slow, btw, All Pairs Shortest Paths algorithm: Floyd Warshall, is $O(N^3)$
$O(2^n)$	1000	2^{1000}	Poor, exponential growth... try to avoid this. Use Dynamic Programming if you can.
$O(n!)$	1000	uncountable	Typical NP-Complete problems.

Table 2: Limit of maximum input size under 60 seconds time limit (with assumptions)

Order of Growth	Time (ms)	Max Possible n	Comment
$O(1)$	60.000 ms	Virtually infinite	Best
$O(\log n)$	60.000 ms	$6^{18.000}$	A very very large number
$O(n)$	60.000 ms	60.000	Still a very big number
$O(n \log n)$	60.000 ms	~ 5.000	Moderate, average real life size
$O(n^2)$	60.000 ms	244	small
$O(n^3)$	60.000 ms	39	very small
$O(2^n)$	60.000 ms	16	avoid if you can
$O(n!)$	60.000 ms	8	extremely too small.

It may be useful to memorize the following ordering for quickly determine which algorithm perform better asymptotically: **constant** < **log n** < **n** < **n log n** < **n^2** < **n^3** < **2^n** < **n!**

Some rules of thumb

1. Biggest built in data structure "long long" is $2^{63}-1$: $9 \cdot 10^{18}$ (up to 18 digits)
2. If you have k nested loops running about n iterations each, the program has $O(n^k)$ complexity
3. If your program is recursive with b recursive calls per level and has l levels, the program $O(b^l)$ complexity
4. Bear in mind that there are n! permutations and 2^n subsets or combinations of n elements when dealing with those kinds of algorithms

5. The best times for sorting n elements are $O(n \log n)$
6. DP algorithms which involves filling in a matrix usually in $O(n^3)$
7. In contest, most of the time $O(n \log n)$ algorithms will be sufficient.

The art of testing your code

You've done it. Identifying the problem, designing the best possible algorithm, you have calculate using time/space complexity, that it will be within time and memory limit given., and you have code it so well. But, is it 100% correct?

Depends on the programming contest's type, you may or may not get credit by solving the problem partially. In ACM ICPC, you will only get credit if your team's code solve all the test cases, that's it, you'll get either Accepted or Not Accepted (Wrong Answer, Time Limit Exceeded, etc). In IOI, there exist partial credit system, in which you will get score: number of correct/total number of test cases for each code that you submit.

In either case, you will need to be able to design a good, educated, tricky test cases. Sample input-output given in problem description is by default too trivial and therefore not a good way to measure your code's correctness for all input instances.

Rather than wasting your submission (and gaining time or points penalty) by getting wrong answer, you may want to design some tricky test cases first, test it in your own machine, and ensure your code is able to solve it correctly (otherwise, there is no point submitting your solution right?).

Some team coaches sometime ask their students to compete with each other by designing test cases. If student A's test cases can break other student's code, then A will get bonus point. You may want to try this in your school team training too.

Here is some guidelines in designing good test cases:

1. Must include sample input, the most trivial one, you even have the answer given.
2. Must include boundary cases, what is the maximum n, x, y , or other input variables, try varying their values to test for out of bound errors.
3. For multiple input test case, try using two identical test case consecutively. Both must output the same result. This is to check whether you forgot to initialize some variables, which will be easily identified if the first instance produce correct output but the second one doesn't.
4. Increase the size of input. Sometimes your program works for small input size, but behave wrongly when input size increases.
5. Tricky test cases, analyze the problem description and identify parts that are tricky, test them to your code.
6. Don't assume input will always nicely formatted if the problem description didn't say

so. Try inserting white spaces (space, tabs) in your input, check whether your code is able to read in the values correctly

7. Finally, do random test cases, try random input and check your code's correctness.

Producing Winning Solution

A good way to get a competitive edge is to write down a game plan for what you're going to do in a contest round. This will help you script out your actions, in terms of what to do both when things go right and when things go wrong. This way you can spend your thinking time in the round figuring out programming problems and not trying to figure out what the heck you should do next... it's sort of like Pre-Computing your reactions to most situations.

Read through all the problems first, don't directly attempt one problem since you may missed easier problem.

1. Order the problems: shortest job first, in terms of your effort (shortest to longest: done it before, easy, unfamiliar, hard).
2. Sketch the algorithms, complexity, the numbers, data structures, tricky details.
3. Brainstorm other possible algorithms (if any) - then pick the stupidest that works!
4. Do the Math! (space & time complexity & plug-in actual expected & worst case numbers).
5. Code it of course, as fast as possible, and it must be correct.
6. Try to break the algorithm - use special (degenerate?) test cases.

Coding a problem

1. Only coding after you finalize your algorithm.
2. Create test data for tricky cases.
3. Code the input routine and test it (write extra output routines to show data).
4. Code the output routine and test it.
5. Write data structures needed.
6. Stepwise refinement: write comments outlining the program logic.
7. Fill in code and debug one section at a time.
8. Get it working & verify correctness (use trivial test cases).
9. Try to break the code - use special cases for code correctness.

Time management strategy and "damage control" scenarios

Have a plan for what to do when various (foreseeable!) things go wrong; imagine problems you might have and figure out how you want to react. The central question is:

"When do you spend more time debugging a program, and when do you cut your losses and move on?". Consider these issues:

1. How long have you spent debugging it already?
2. What type of bug do you seem to have?
3. Is your algorithm wrong?
4. Do your data structures need to be changed?
5. Do you have any clue about what's going wrong?
6. A short amount (20 mins) of debugging is better than switching to anything else; but you might be able to solve another from scratch in 45 mins.
7. When do you go back to a problem you've abandoned previously?
8. When do you spend more time optimizing a program, and when do you switch?
9. Consider from here out - forget prior effort, focus on the future: how can you get the most points in the next hour with what you have?

Tips & tricks for contests

1. Brute force when you can, Brute force algorithm tends to be the easiest to implement.
2. KISS: Simple is smart! (Keep It Simple, Stupid !!! / Keep It Short & Simple).
3. Hint: focus on limits (specified in problem statement).
4. Waste memory when it makes your life easier (trade memory space for speed).
5. Don't delete your extra debugging output, comment it out.
6. Optimize progressively, and only as much as needed.
7. Keep all working versions!
8. Code to debug:
 - a. white space is good,
 - b. use meaningful variable names,
 - c. don't reuse variables, (we are not doing software engineering here)
 - d. stepwise refinement,
 - e. Comment before code.
9. Avoid pointers if you can.
10. Avoid dynamic memory like the plague: statically allocate everything. (yeah yeah)
11. Try not to use floating point; if you have to, put tolerances in everywhere (never test equality)
12. Comments on comments:
 - a. Not long prose, just brief notes.
 - b. Explain high-level functionality: `++i; /* increase the value of i by */` is worse than useless.
 - c. Explain code trickery.
 - d. Delimit & document functional sections.

Keep a log of your performance in each contest: successes, mistakes, and what you could have done better; use this to rewrite and improve your game plan!

The Judges Are Evil and Out to Get You

Judges don't want to put easy problems on the contest, because they have thought up too many difficult problems. So what we do is hide the easy problems, hoping that you will be tricked into working on the harder ones. If we want you to add two non-negative numbers together, there will be pages of text on the addition of '0' to the number system and 3D-pictures to explain the process of addition as it was imagined on some island that nobody has ever heard of.

Once we've scared you away from the easy problems, we make the hard ones look easy. 'Given two polygons, find the area of their intersection.' Easy, right?

It isn't always obvious that a problem is easy, so teams ignore the problems or start on overly complex approaches to them. Remember, there are dozens of other teams working on the same problems, and they will help you find the easy problems. **If everyone is solving problem G, maybe you should take another look at it.**^[6]

CHAPTER 3 PROGRAMMING IN C: A TUTORIAL^[11]

C was created by **Dennis Ritchie** at the **Bell Telephone Laboratories** in 1972. Because C is such a powerful and flexible language, its use quickly spread beyond Bell Labs. Programmers everywhere began using it to write all sorts of programs. Soon, however, different organizations began utilizing their own versions of C, and subtle differences between implementations started to cause programmers headaches. In response to this problem, the **American National Standards Institute (ANSI)** formed a committee in 1983 to establish a standard definition of C, which became known as ANSI Standard C. With few exceptions, every modern C compiler has the ability to adhere this standard [11].

A Simple C Program

A C program consists of one or more functions, which are similar to the functions and subroutines of a Fortran program or the procedures of PL/I, and perhaps some external data definitions. `main` is such a function, and in fact all C programs must have a `main`. Execution of the program begins at the first statement of `main`. `main` will usually invoke other functions to perform its job, some coming from the same program, and others from libraries.

```
main( ) {  
    printf("hello, world");  
}
```

`printf` is a library function which will format and print output on the terminal (unless some other destination is specified). In this case it prints

```
hello, world
```

A Working C Program; Variables; Types and Type Declarations

Here's a bigger program that adds three integers and prints their sum.

```
main( ) {  
    int a, b, c, sum;  
    a = 1; b = 2; c = 3;  
    sum = a + b + c;  
    printf("sum is %d", sum);  
}
```

Arithmetic and the assignment statements are much the same as in Fortran (except for the semicolons) or PL/I. The format of C programs is quite free. We can put several

statements on a line if we want, or we can split a statement among several lines if it seems desirable. The split may be between any of the operators or variables, but *not* in the middle of a name or operator. As a matter of style, spaces, tabs, and newlines should be used freely to enhance readability.

C has fundamental types of variables:

Variable Type	Keyword	Bytes Required	Range
Character	char	1	-128 to 127
Integer	int	2	-32768 to 32767
Short integer	short	2	-32768 to 32767
Long integer	long	4	-2,147,483,648 to 2,147,438,647
Unsigned character	unsigned char	1	0 to 255
Unsigned integer	unsigned int	2	0 to 65535
Unsigned short integer	unsigned short	2	0 to 65535
Unsigned long integer	unsigned long	4	0 to 4,294,967,295
Single-precision	float	4	1.2E-38 to
floating-point			3.4E38
Double-precision	double	8	2.2E-308 to
floating-point			1.8E308 ²

There are also arrays and structures of these basic types, pointers to them and functions that return them, all of which we will meet shortly.

All variables in a C program must be declared, although this can sometimes be done implicitly by context. Declarations must precede executable statements. The declaration

```
int a, b, c, sum;
```

Variable names have one to eight characters, chosen from A-Z, a-z, 0-9, and `_`, and start with a non-digit.

Constants

We have already seen decimal integer constants in the previous example-- 1, 2, and 3. Since C is often used for system programming and bit-manipulation, octal numbers are an important part of the language. In C, any number that begins with 0 (zero!) is an octal integer (and hence can't have any 8's or 9's in it). Thus `0777` is an octal constant, with decimal value 511.

A "character" is one byte (an inherently machine-dependent concept). Most often this is expressed as a character constant, which is one character enclosed in single quotes. However, it may be any quantity that fits in a byte, as in flags below:

```
char quest, newline, flags;
quest = '?';
newline = '\n';
flags = 077;
```

The sequence '\n' is C notation for "newline character", which, when printed, skips the terminal to the beginning of the next line. Notice that '\n' represents only a single character. There are several other "escapes" like '\n' for representing hard-to-get or invisible characters, such as '\t' for tab, '\b' for backspace, '\0' for end of file, and '\\' for the backslash itself.

Simple I/O – `getchar()`, `putchar()`, `printf ()`

`getchar` and `putchar` are the basic I/O library functions in C. `getchar` fetches one character from the standard input (usually the terminal) each time it is called, and returns that character as the value of the function. When it reaches the end of whatever file it is reading, thereafter it returns the character represented by '\0' (ascii NUL, which has value zero). We will see how to use this very shortly.

```
main( ) {
    char c;
    c = getchar( );
    putchar(c);
}
```

`putchar` puts one character out on the standard output (usually the terminal) each time it is called. So the program above reads one character and writes it back out. By itself, this isn't very interesting, but observe that if we put a loop around this, and add a test for end of file, we have a complete program for copying one file to another.

`printf` is a more complicated function for producing formatted output.

```
printf ("hello, world\n");
```

is the simplest use. The string "hello, world\n" is printed out.

More complicated, if `sum` is 6,

```
printf ("sum is %d\n", sum);
```

prints

```
sum is 6
```

Within the first argument of `printf`, the characters ```%d"` signify that the next argument in the argument list is to be printed as a base 10 number.

Other useful formatting commands are ```%c"` to print out a single character, ```%s"` to print out an entire string, and ```%o"` to print a number as octal instead of decimal (no leading zero). For example,

```
n = 511;
printf ("What is the value of %d in octal?", n);
printf ("%s! %d decimal is %o octal\n", "Right", n, n);
```

prints

```
What is the value of 511 in octal? Right! 511 decimal
is 777 octal
```

If - relational operators and compound statements

The basic conditional-testing statement in C is the `if` statement:

```
c = getchar( );
if( c == '?' )
    printf("why did you type a question mark?\n");
```

The simplest form of `if` is

```
if (expression) statement
```

The condition to be tested is any expression enclosed in parentheses. It is followed by a statement. The expression is evaluated, and if its value is non-zero, the statement is executed. There's an optional `else` clause, to be described soon.

The character sequence ```=='` is one of the relational operators in C; here is the complete set:

```
==      equal to (.EQ. to Fortraners)
!=      not equal to
>       greater than
```

```

<      less than
>=     greater than or equal to
<=     less than or equal to

```

The value of "`expression relation expression`" is 1 if the relation is true, and 0 if false. Don't forget that the equality test is `==`; a single `=` causes an assignment, not a test, and invariably leads to disaster.

Tests can be combined with the operators `&&` (AND), `||` (OR), and `!` (NOT). For example, we can test whether a character is blank or tab or newline with

```
if( c==' ' || c=='\t' || c=='\n' ) ...
```

C guarantees that `&&` and `||` are evaluated left to right -- we shall soon see cases where this matters.

As a simple example, suppose we want to ensure that `a` is bigger than `b`, as part of a sort routine. The interchange of `a` and `b` takes three statements in C, grouped together by `{}`:

```

if ( a < b ) {
    t = a;
    a = b;
    b = t;
}

```

As a general rule in C, anywhere you can use a simple statement, you can use any compound statement, which is just a number of simple or compound ones enclosed in `{}`. There is no semicolon after the `}` of a compound statement, but there *is* a semicolon after the last non-compound statement inside the `{}`.

While Statement; Assignment within an Expression; Null Statement

The basic looping mechanism in C is the while statement. Here's a program that copies its input to its output a character at a time. Remember that `\0` marks the end of file.

```

main( ) {
    char c;
    while( (c=getchar( )) != '\0' )
        putchar(c);
}

```

The while statement is a loop, whose general form is

```
while (expression) statement
```

Its meaning is

- (a) evaluate the expression
- (b) if its value is true (i.e., not zero) do the statement, and go back to (a)

Because the expression is tested before the statement is executed, the statement part can be executed zero times, which is often desirable. As in the if statement, the expression and the statement can both be arbitrarily complicated, although we haven't seen that yet. Our example gets the character, assigns it to *c*, and then tests if it's a `'\0'`. If it is not a `'\0'`, the statement part of the while is executed, printing the character. The while then repeats. When the input character is finally a `'\0'`, the while terminates, and so does main.

Notice that we used an assignment statement

```
c = getchar( )
```

within an expression. This is a handy notational shortcut which often produces clearer code. (In fact it is often the only way to write the code cleanly. As an exercise, rewrite the file-copy without using an assignment inside an expression.) It works because an assignment statement has a value, just as any other expression does. Its value is the value of the right hand side. This also implies that we can use multiple assignments like

```
x = y = z = 0;
```

Evaluation goes from right to left.

By the way, the extra parentheses in the assignment statement within the conditional were really necessary: if we had said

```
c = getchar( ) != '\0'
```

c would be set to 0 or 1 depending on whether the character fetched was an end of file or not. This is because in the absence of parentheses the assignment operator `'='` is evaluated after the relational operator `'!='`. When in doubt, or even if not, parenthesize.

```
main( ) {  
    while( putchar(getchar( )) != '\0' ) ;  
}
```

What statement is being repeated? None, or technically, the null statement, because all the work is really done within the test part of the while. This version is slightly different from the previous one, because the final `\0` is copied to the output before we decide to stop.

Arithmetic

The arithmetic operators are the usual `+`, `-`, `*`, and `/` (truncating integer division if the operands are both int), and the remainder or mod operator `%`:

```
x = a%b;
```

sets `x` to the remainder after `a` is divided by `b` (i.e., `a mod b`). The results are machine dependent unless `a` and `b` are both positive.

In arithmetic, char variables can usually be treated like int variables. Arithmetic on characters is quite legal, and often makes sense:

```
c = c + 'A' - 'a';
```

converts a single lower case ascii character stored in `c` to upper case, making use of the fact that corresponding ascii letters are a fixed distance apart. The rule governing this arithmetic is that all chars are converted to int before the arithmetic is done. Beware that conversion may involve sign-extension if the leftmost bit of a character is 1, the resulting integer might be negative. (This doesn't happen with genuine characters on any current machine.)

So to convert a file into lower case:

```
main( ) {
    char c;
    while( (c=getchar( )) != '\0' )
        if( 'A'<=c && c<='Z' )
            putchar(c+'a'-'A');
        else
            putchar(c); }
```

Else Clause; Conditional Expressions

We just used an `else` after an `if`. The most general form of `if` is

```
if (expression) statement1 else statement2
```

the `else` part is optional, but often useful. The canonical example sets `x` to the minimum of `a` and `b`:

```
if (a < b)
    x = a;
else
    x = b;
```

C provides an alternate form of conditional which is often more concise. It is called the "conditional expression" because it is a conditional which actually has a value and can be used anywhere an expression can. The value of

```
a < b ? a : b;
```

is `a` if `a` is less than `b`; it is `b` otherwise. In general, the form

```
expr1 ? expr2 : expr3
```

To set `x` to the minimum of `a` and `b`, then:

```
x = (a < b ? a : b);
```

The parentheses aren't necessary because `'?:'` is evaluated before `'='`, but safety first.

Going a step further, we could write the loop in the lower-case program as

```
while( (c=getchar( )) != '\0' )
    putchar( ('A'<=c && c<='Z') ? c-'A'+'a' : c );
```

If's and else's can be used to construct logic that branches one of several ways and then rejoins, a common programming structure, in this way:

```
if(...)
    {...}
else if(...)
    {...}
else if(...)
    {...}
else
    {...}
```

The conditions are tested in order, and exactly one block is executed; either the first one whose if is satisfied, or the one for the last `else`. When this block is finished, the next statement executed is the one after the last `else`. If no action is to be taken for the "default" case, omit the last `else`.

For example, to count letters, digits and others in a file, we could write

```
main( ) {
    int let, dig, other, c;
    let = dig = other = 0;
    while( (c=getchar( )) != '\0' )
        if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )
            ++let;
        else if( '0'<=c && c<='9' ) ++dig;
        else ++other;
    printf("%d letters, %d digits, %d others\n", let, dig, other);
}
```

This code letters, digits and others in a file.

Increment and Decrement Operators

In addition to the usual '-', C also has two other interesting unary operators, '++' (increment) and '--' (decrement). Suppose we want to count the lines in a file.

```
main( ) {
    int c,n;
    n = 0;
    while( (c=getchar( )) != '\0' )
        if( c == '\n' )
            ++n;
    printf("%d lines\n", n);
}
```

++n is equivalent to n=n+1 but clearer, particularly when n is a complicated expression. '++' and '--' can be applied only to int's and char's (and pointers which we haven't got to yet).

The unusual feature of '++' and '--' is that they can be used either before or after a variable. The value of ++k is the value of k *after* it has been incremented. The value of k++ is k *before* it is incremented. Suppose k is 5. Then

```
x = ++k;
```

increments k to 6 and then sets x to the resulting value, i.e., to 6. But

```
x = k++;
```

first sets *x* to 5, and *then* increments *k* to 6. The incrementing effect of *++k* and *k++* is the same, but their values are respectively 5 and 6. We shall soon see examples where both of these uses are important.

Arrays

In C, as in Fortran or PL/I, it is possible to make arrays whose elements are basic types. Thus we can make an array of 10 integers with the declaration

```
int x[10];
```

The square brackets mean subscripting; parentheses are used only for function references. Array indexes begin at zero, so the elements of *x* are

```
x[0], x[1], x[2], ..., x[9]
```

If an array has *n* elements, the largest subscript is *n*-1.

Multiple-dimension arrays are provided, though not much used above two dimensions. The declaration and use look like

```
int name[10] [20];
n = name[i+j] [1] + name[k] [2];
```

Subscripts can be arbitrary integer expressions. Multi-dimension arrays are stored by row (opposite to Fortran), so the rightmost subscript varies fastest; *name* has 10 rows and 20 columns.

Here is a program which reads a line, stores it in a buffer, and prints its length (excluding the newline at the end).

```
main( ) {
    int n, c;
    char line[100];
    n = 0;
    while( (c=getchar( )) != '\n' ) {
        if( n < 100 )
            line[n] = c;
        n++;
    }
    printf("length = %d\n", n);
}
```

As a more complicated problem, suppose we want to print the count for each line in the input, still storing the first 100 characters of each line. Try it as an exercise before looking at the solution:

```
main( ) {
    int n, c; char line[100];
    n = 0;
    while( (c=getchar( )) != '\0' )
        if( c == '\n' ) {
            printf("%d, n);
            n = 0;
        }
        else {
            if( n < 100 ) line[n] = c;
            n++;
        }
}
```

Above code stores first 100 characters of each line!

Character Arrays; Strings

Text is usually kept as an array of characters, as we did with `line[]` in the example above. By convention in C, the last character in a character array should be a `'\0'` because most programs that manipulate character arrays expect it. For example, `printf` uses the `'\0'` to detect the end of a character array when printing it out with a `'%s'`.

We can copy a character array `s` into another `t` like this:

```
i = 0;
while( (t[i]=s[i]) != '\0' )
    i++;
```

Most of the time we have to put in our own `'\0'` at the end of a string; if we want to print the line with `printf`, it's necessary. This code prints the character count before the line:

```
main( ) {
    int n;
    char line[100];
    n = 0;
    while( (line[n++]=getchar( )) != '\n' );
    line[n] = '\0';
    printf("%d:\t%s", n, line);
}
```

Here we increment *n* in the subscript itself, but only after the previous value has been used. The character is read, placed in `line[n]`, and only then *n* is incremented.

There is one place and one place only where C puts in the `'\0'` at the end of a character array for you, and that is in the construction

```
"stuff between double quotes"
```

The compiler puts a `'\0'` at the end automatically. Text enclosed in double quotes is called a *string*; its properties are precisely those of an (initialized) array of characters.

for Statement

The for statement is a somewhat generalized while that lets us put the initialization and increment parts of a loop into a single statement along with the test. The general form of the for is

```
for( initialization; expression; increment )
    statement
```

The meaning is exactly

```
initialization;
while( expression ) {
    statement
    increment;
}
```

This slightly more ornate example adds up the elements of an array:

```
sum = 0;
for( i=0; i<n; i++)sum = sum + array[i];
```

In the for statement, the initialization can be left out if you want, but the semicolon has to be there. The increment is also optional. It is *not* followed by a semicolon. The second clause, the test, works the same way as in the `while`: if the expression is true (not zero) do another loop, otherwise get on with the next statement. As with the `while`, the for loop may be done zero times. If the expression is left out, it is taken to be always true, so

```
for( ; ; ) ...
while( 1 ) ...
```

are both infinite loops.

You might ask why we use a `for` since it's so much like a `while`. (You might also ask why we use a `while` because...) The `for` is usually preferable because it keeps the code where it's used and sometimes eliminates the need for compound statements, as in this code that zeros a two-dimensional array:

```
for( i=0; i<n; i++ )
    for( j=0; j<m; j++ )
        array[i][j] = 0;
```

Functions; Comments

Suppose we want, as part of a larger program, to count the occurrences of the ascii characters in some input text. Let us also map illegal characters (those with value >127 or <0) into one pile. Since this is presumably an isolated part of the program, good practice dictates making it a separate function. Here is one way:

```
main( ) {
    int hist[129];      /* 128 legal chars + 1 illegal group*/
    ...
    count(hist, 128);  /* count the letters into hist */
    printf( ... );    /* comments look like this; use them */
    ...
    /* anywhere blanks, tabs or newlines could appear */
}

count(buf, size)
    int size, buf[ ]; {
    int i, c;
    for( i=0; i<=size; i++ )
        buf[i] = 0;          /* set buf to zero */
    while( (c=getchar( )) != '\0' ) { /* read til eof */
        if( c > size || c < 0 )
            c = size;        /* fix illegal input */
        buf[c]++;
    }
    return;
}
```

We have already seen many examples of calling a function, so let us concentrate on how to define one. Since `count` has two arguments, we need to declare them, as shown, giving their types, and in the case of `buf`, the fact that it is an array. The declarations of arguments go between the argument list and the opening `{`. There is no need to specify the size of the array `buf`, for it is defined outside of `count`.

The return statement simply says to go back to the calling routine. In fact, we could have omitted it, since a return is implied at the end of a function.

What if we wanted count to return a value, say the number of characters read? The return statement allows for this too:

```
int i, c, nchar;
nchar = 0;
...
while( (c=getchar( )) != '\0' ) {
    if( c > size || c < 0 )
        c = size;
    buf[c]++;
    nchar++;
}
return(nchar);
```

Any expression can appear within the parentheses. Here is a function to compute the minimum of two integers:

```
min(a, b)
int a, b; {
    return( a < b ? a : b );
}
```

To copy a character array, we could write the function

```
strcpy(s1, s2) /* copies s1 to s2 */
char s1[ ], s2[ ]; {
    int i;
    for( i = 0; (s2[i] = s1[i]) != '\0'; i++ ); }
```

As is often the case, all the work is done by the assignment statement embedded in the test part of the for. Again, the declarations of the arguments s1 and s2 omit the sizes, because they don't matter to strcpy. (In the section on pointers, we will see a more efficient way to do a string copy.)

There is a subtlety in function usage which can trap the unsuspecting Fortran programmer. Simple variables (not arrays) are passed in C by "call by value", which means that the called function is given a copy of its arguments, and doesn't know their addresses. This makes it impossible to change the value of one of the actual input arguments.

There are two ways out of this dilemma. One is to make special arrangements to pass to the function the address of a variable instead of its value. The other is to make the variable

a global or external variable, which is known to each function by its name. We will discuss both possibilities in the next few sections.

Local and External Variables

If we say

```
f( ) {
    int x;
    ...
}
g( ) {
    int x;
    ...
}
```

each *x* is *local* to its own routine -- the *x* in *f* is unrelated to the *x* in *g*. (Local variables are also called "automatic".) Furthermore each local variable in a routine appears only when the function is called, and *disappears* when the function is exited. Local variables have no memory from one call to the next and must be explicitly initialized upon each entry. (There is a static storage class for making local variables with memory; we won't discuss it.)

As opposed to local variables, external variables are defined external to all functions, and are (potentially) available to all functions. External storage always remains in existence. To make variables external we have to define them external to all functions, and, wherever we want to use them, make a declaration.

```
main( ) {
    extern int nchar, hist[ ];
    ...
    count( );
    ...
}
count( ) {
    extern int nchar, hist[ ];
    int i, c;
    ...
}

int    hist[129];    /* space for histogram */
int    nchar;       /* character count */
```

Roughly speaking, any function that wishes to access an external variable must contain an `extern` declaration for it. The declaration is the same as others, except for the added keyword `extern`. Furthermore, there must somewhere be a definition of the external variables external to all functions.

External variables can be initialized; they are set to zero if not explicitly initialized. In its simplest form, initialization is done by putting the value (which must be a constant) after the definition:

Pointers

A pointer in C is the address of something. It is a rare case indeed when we care what the specific address itself is, but pointers are a quite common way to get at the contents of something. The unary operator `&` is used to produce the address of an object, if it has one. Thus

```
int a, b;  
b = &a;
```

puts the address of `a` into `b`. We can't do much with it except print it or pass it to some other routine, because we haven't given `b` the right kind of declaration. But if we declare that `b` is indeed a pointer to an integer, we're in good shape:

```
int a, *b, c;  
b = &a;  
c = *b;
```

`b` contains the address of `a` and `*b` means to use the value in `b` as an address, i.e., as a pointer. The effect is that we get back the contents of `a`, albeit rather indirectly. (It's always the case that `*&x` is the same as `x` if `x` has an address.)

The most frequent use of pointers in C is for walking efficiently along arrays. In fact, in the implementation of an array, the array name represents the address of the zeroth element of the array, so you can't use it on the left side of an expression. (You can't change the address of something by assigning to it.) If we say

```
char *y; char x[100];
```

`y` is of type pointer to character (although it doesn't yet point anywhere). We can make `y` point to an element of `x` by either of

```
y = &x[0];  
y = x;
```

Since `x` is the address of `x[0]` this is legal and consistent.

Now `*y` gives `x[0]`. More importantly,

```
*(y+1) gives x[1]  
*(y+i) gives x[i]
```


and the sequence

```
y = &x[0];
y++;
```

leaves *y* pointing at *x*[1].

Let's use pointers in a function `length` that computes how long a character array is. Remember that by convention all character arrays are terminated with a `'\0'`. (And if they aren't, this program will blow up inevitably.) The old way:

```
length(s)
char s[ ]; {
    int n;
    for( n=0; s[n] != '\0'; )
        n++;
    return(n);
}
```

Rewriting with pointers gives

```
length(s)
char *s; {
    int n;
    for( n=0; *s != '\0'; s++ )
        n++;
    return(n); }
```

You can now see why we have to say what kind of thing *s* points to -- if we're to increment it with `s++` we have to increment it by the right amount.

The pointer version is more efficient (this is almost always true) but even more compact is

```
for( n=0; *s++ != '\0'; n++ );
```

The `'*s'` returns a character; the `'++'` increments the pointer so we'll get the next character next time around. As you can see, as we make things more efficient, we also make them less clear. But `'*s++'` is an idiom so common that you have to know it.

Going a step further, here's our function `strcpy` that copies a character array *s* to another *t*.

```
strcpy(s,t)
char *s, *t; {
    while(*t++ = *s++);
}
```

We have omitted the test against `\0`, because `\0` is identically zero; you will often see the code this way.

For arguments to a function, and there only, the declarations

```
char s[ ];  
char *s;
```

are equivalent -- a pointer to a type, or an array of unspecified size of that type, are the same thing.

Function Arguments

Look back at the [function `strcpy` in the previous section](#). We passed it two string names as arguments, then proceeded to clobber both of them by incrementation. So how come we don't lose the original strings in the function that called `strcpy`?

As we said before, C is a "call by value" language: when you make a function call like `f(x)`, the *value* of `x` is passed, not its address. So there's no way to *alter* `x` from inside `f`. If `x` is an array (`char x[10]`) this isn't a problem, because `x` is an address anyway, and you're not trying to change it, just what it addresses. This is why `strcpy` works as it does. And it's convenient not to have to worry about making temporary copies of the input arguments.

But what if `x` is a scalar and you do want to change it? In that case, you have to pass the *address* of `x` to `f`, and then use it as a pointer. Thus for example, to interchange two integers, we must write

```
flip(x, y)  
int *x, *y; {  
    int temp;  
    temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

and to call `flip`, we have to pass the addresses of the variables:

```
flip (&a, &b);
```

Which interchange two integers.

The Switch Statement ; Break ; Continue

The switch statement can be used to replace the multi-way test we used in the last example. When the tests are like this:

```
if( c == 'a' ) ...
else if( c == 'b' ) ...
else if( c == 'c' ) ...
else ...
```

testing a value against a series of constants, the switch statement is often clearer and usually gives better code. Use it like this:

```
switch( c ) {

case 'a':
    aflag++;
    break;
case 'b':
    bflag++;
    break;
case 'c':
    cflag++;
    break;
default:
    printf("%c?\n", c);
    break;
}
```

The case statements label the various actions we want; `default` gets done if none of the other cases are satisfied. (A `default` is optional; if it isn't there, and none of the cases match, you just fall out the bottom.)

The `break` statement in this example is new. It is there because the cases are just labels, and after you do one of them, you fall through to the next unless you take some explicit action to escape. This is a mixed blessing. On the positive side, you can have multiple cases on a single statement; we might want to allow both upper and lower

But what if we just want to get out after doing case `'a'`? We could get out of a `case` of the `switch` with a label and a `goto`, but this is really ugly. The `break` statement lets us exit without either `goto` or label.

The `break` statement also works in `for` and `while` statements; it causes an immediate exit from the loop.

The `continue` statement works *only* inside `for`'s and `while`'s; it causes the next iteration of the loop to be started. This means it goes to the increment part of the `for` and the test part of the `while`.

Structures

The main use of structures is to lump together collections of disparate variable types, so they can conveniently be treated as a unit. For example, if we were writing a compiler or assembler, we might need for each identifier information like its name (a character array), its source line number (an integer), some type information (a character, perhaps), and probably a usage count (another integer).

```
char   id[10];
int    line;
char   type;
int    usage;
```

We can make a structure out of this quite easily. We first tell C what the structure will look like, that is, what kinds of things it contains; after that we can actually reserve storage for it, either in the same statement or separately. The simplest thing is to define it and allocate storage all at once:

```
struct {
    char   id[10];
    int    line;
    char   type;
    int    usage;      } sym;
```

This defines `sym` to be a structure with the specified shape; `id`, `line`, `type` and `usage` are members of the structure. The way we refer to any particular member of the structure is

```
structure-name . member
```

as in

```
sym.type = 077;
if( sym.usage == 0 ) ...
while( sym.id[j++] ) ...
    etc.
```

Although the names of structure members never stand alone, they still have to be unique; there can't be another `id` or `usage` in some other structure.

So far we haven't gained much. The advantages of structures start to come when we have arrays of structures, or when we want to pass complicated data layouts between functions. Suppose we wanted to make a symbol table for up to 100 identifiers. We could extend our definitions like

```
char    id[100][10];
int     line[100];
char    type[100];
int     usage[100];
```

but a structure lets us rearrange this spread-out information so all the data about a single identifier is collected into one lump:

```
struct {
    char    id[10];
    int     line;
    char    type;
    int     usage;
} sym[100];
```

This makes `sym` an array of structures; each array element has the specified shape. Now we can refer to members as

```
sym[i].usage++; /* increment usage of i-th identifier */
for( j=0; sym[i].id[j++] != '\0'; ) ...
etc.
```

Thus to print a list of all identifiers that haven't been used, together with their line number,

```
for( i=0; i<nsym; i++ )
    if( sym[i].usage == 0 )
        printf("%d\t%s\n", sym[i].line, sym[i].id);
```

Suppose we now want to write a function `lookup(name)` which will tell us if `name` already exists in `sym`, by giving its index, or that it doesn't, by returning a `-1`. We can't pass a structure to a function directly; we have to either define it externally, or pass a pointer to it. Let's try the first way first.

```
int     nsym    0;      /* current length of symbol table */

struct {
    char    id[10];
    int     line;
    char    type;
    int     usage;
} sym[100];           /* symbol table */
```

```

main( ) {
    ...
    if( (index = lookup(newname)) >= 0 )
        sym[index].usage++;          /* already there ... */
    else
        install(newname, newline, newtype);
    ...
}

lookup(s)
char *s; {
    int i;
    extern struct {
        char    id[10];
        int     line;
        char    type;
        int     usage;
    } sym[ ];

    for( i=0; i<nsym; i++ )
        if( compar(s, sym[i].id) > 0 )
            return(i);

    return(-1);
}

compar(s1,s2)          /* return 1 if s1==s2, 0 otherwise */
char *s1, *s2; {
    while( *s1++ == *s2 )
        if( *s2++ == '\0' )
            return(1);

    return(0);
}

```

The declaration of the structure in `lookup` isn't needed if the external definition precedes its use in the same source file, as we shall see in a moment.

Now what if we want to use pointers?

```

struct symtag {
    char    id[10];
    int     line;
    char    type;
    int     usage;
} sym[100], *psym;

psym = &sym[0]; /* or p = sym; */

```

This makes `psym` a pointer to our kind of structure (the symbol table), then initializes it to point to the first element of `sym`.

Notice that we added something after the word `struct`: a ```tag"` called `syntag`. This puts a name on our structure definition so we can refer to it later without repeating the definition. It's not necessary but useful. In fact we could have said

```
struct syntag {
    ... structure definition
};
```

which wouldn't have assigned any storage at all, and then said

```
struct syntag sym[100];
struct syntag *psym;
```

which would define the array and the pointer. This could be condensed further, to

```
struct syntag sym[100], *psym;
```

The way we actually refer to an member of a structure by a pointer is like this:

```
ptr -> structure-member
```

The symbol ``->` means we're pointing at a member of a structure; ``->` is only used in that context. `ptr` is a pointer to the (base of) a structure that contains the structure member. The expression `ptr->structure-member` refers to the indicated member of the pointed-to structure. Thus we have constructions like:

```
psym->type = 1;
psym->id[0] = 'a';
```

For more complicated pointer expressions, it's wise to use parentheses to make it clear who goes with what. For example,

```
struct { int x, *y; } *p;
p->x++ increments x
++p->x so does this!
(++p)->x increments p before getting x
*p->y++ uses y as a pointer, then increments it
*(p->y)++ so does this
*(p++)->y uses y as a pointer, then increments p
```

The way to remember these is that `->`, `.` (dot), `()` and `[]` bind very tightly. An expression involving one of these is treated as a unit. `p->x`, `a[i]`, `y.x` and `f(b)` are names exactly as `abc` is.

If `p` is a pointer to a structure, any arithmetic on `p` takes into account the actual size of the structure. For instance, `p++` increments `p` by the correct amount to get the next element of the array of structures. But don't assume that the size of a structure is the sum of the sizes

of its members -- because of alignments of different sized objects, there may be "holes" in a structure.

Enough theory. Here is the lookup example, this time with pointers.

```

struct symtag {
    char    id[10];
    int     line;
    char    type;
    int     usage;
} sym[100];

main( ) {
    struct symtag *lookup( );
    struct symtag *psym;
    ...
    if( (psym = lookup(newname)) ) /* non-zero pointer */
        psym -> usage++;          /* means already there */
    else
        install(newname, newline, newtype);
    ...
}

struct symtag *lookup(s)
    char *s; {
    struct symtag *p;
    for( p=sym; p < &sym[nsym]; p++ )
        if( compar(s, p->id) > 0)
            return(p);
    return(0);
}

```

The function `compar` doesn't change: ``p->id'` refers to a string.

In `main` we test the pointer returned by `lookup` against zero, relying on the fact that a pointer is by definition never zero when it really points at something. The other pointer manipulations are trivial.

The only complexity is the set of lines like

```
struct symtag *lookup( );
```

This brings us to an area that we will treat only hurriedly; the question of function types. So far, all of our functions have returned integers (or characters, which are much the same). What do we do when the function returns something else, like a pointer to a structure? The rule is that any function that doesn't return an `int` has to say explicitly what it does return. The type information goes before the function name (which can make the name hard to see).

Examples:

```
char f(a)
    int a; {
        ...
    }
int *g( ) { ... }

struct symtag *lookup(s) char *s; { ... }
```

The function `f` returns a character, `g` returns a pointer to an integer, and `lookup` returns a pointer to a structure that looks like `symtag`. And if we're going to use one of these functions, we have to make a declaration where we use it, as we did in `main` above.

Notice the parallelism between the declarations

```
struct symtag *lookup( );
struct symtag *psym;
```

In effect, this says that `lookup()` and `psym` are both used the same way - as a pointer to a structure -- even though one is a variable and the other is a function.

Initialization of Variables

An external variable may be initialized at compile time by following its name with an initializing value when it is defined. The initializing value has to be something whose value is known at compile time, like a constant.

```
int    x =    0;    /* "0" could be any constant */
int    *p    &y[1]; /* p now points to y[1] */
```

An external array can be initialized by following its name with a list of initializations enclosed in braces:

```
int    x[4] =    {0,1,2,3}; /* makes x[i] = i */
int    y[ ] =    {0,1,2,3}; /* makes y big enough for 4 values */
char    *msg =    "syntax error\n"; /* braces unnecessary here */
char    *keyword[ ]={
        "if",
        "else",
        "for",
        "while",
        "break",
        "continue",
        0
    };
```

This last one is very useful -- it makes `keyword` an array of pointers to character strings, with a zero at the end so we can identify the last element easily. A simple lookup routine could scan this until it either finds a match or encounters a zero keyword pointer:

```
lookup(str)          /* search for str in keyword[ ] */
char *str; {
    int i,j,r;
    for( i=0; keyword[i] != 0; i++) {
        for( j=0; (r=keyword[i][j]) == str[j] && r != '\0'; j++)
);
        if( r == str[j] )
            return(i);
    }
    return(-1);
}
```

Scope Rules

A complete C program need not be compiled all at once; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. How do we arrange that data gets passed from one routine to another? We have already seen how to use function arguments and values, so let us talk about external data. Warning: the words declaration and definition are used precisely in this section; don't treat them as the same thing.

A major shortcut exists for making extern declarations. If the definition of a variable appears *before* its use in some function, no extern declaration is needed within the function. Thus, if a file contains

```
f1( ) { ... }
int foo;
f2( ) { ... foo = 1; ... }
f3( ) { ... if ( foo ) ... }
```

no declaration of `foo` is needed in either `f2` or `f3`, because the external definition of `foo` appears before them. But if `f1` wants to use `foo`, it has to contain the declaration

```
f1( ) {
    extern int foo;
    ...
}
```

This is true also of any function that exists on another file; if it wants `foo` it has to use an extern declaration for it. (If somewhere there is an extern declaration for something, there must also eventually be an external definition of it, or you'll get an "undefined symbol" message.)

There are some hidden pitfalls in external declarations and definitions if you use multiple source files. To avoid them, first, define and initialize each external variable only once in the entire set of files:

```
int    foo = 0;
```

You can get away with multiple external definitions on UNIX, but not on GCOS, so don't ask for trouble. Multiple initializations are illegal everywhere. Second, at the beginning of any file that contains functions needing a variable whose definition is in some other file, put in an extern declaration, outside of any function:

```
extern int    foo;

fl( ) { ... }          etc.
```

#define, #include

C provides a very limited macro facility. You can say

```
#define name          something
```

and thereafter anywhere ``name" appears as a token, ``something" will be substituted. This is particularly useful in parametering the sizes of arrays:

```
#define ARRAYSIZE    100
int    arr[ARRAYSIZE];
...
while( i++ < ARRAYSIZE )...
```

(now we can alter the entire program by changing only the define) or in setting up mysterious constants:

```
#define SET          01
#define INTERRUPT    02    /* interrupt bit */
#define ENABLED     04
...
if( x & (SET | INTERRUPT | ENABLED) ) ...
```

Now we have meaningful words instead of mysterious constants. (The mysterious operators `&' (AND) and `|' (OR) will be covered in the [next section](#).) It's an excellent practice to write programs without any literal constants except in #define statements.

There are several warnings about #define. First, there's no semicolon at the end of a #define; all the text from the name to the end of the line (except for comments) is taken to be the ``something". When it's put into the text, blanks are placed around it. The other

control word known to C is `#include`. To include one file in your source at compilation time, say

```
#include "filename"
```

Bit Operators

C has several operators for logical bit-operations. For example,

```
x = x & 0177;
```

forms the bit-wise AND of `x` and `0177`, effectively retaining only the last seven bits of `x`. Other operators are

```
|      inclusive OR
^      (circumflex) exclusive OR
~      (tilde) 1's complement
!      logical NOT
<<     left shift (as in x<<2)
>>     right shift (arithmetic on PDP-11; logical on H6070,
IBM360)
```

Assignment Operators

An unusual feature of C is that the normal binary operators like `+`, `-`, etc. can be combined with the assignment operator `=` to form new assignment operators. For example,

```
x -= 10;
```

uses the assignment operator `-=` to decrement `x` by 10, and

```
x =& 0177
```

forms the AND of `x` and `0177`. This convention is a useful notational shortcut, particularly if `x` is a complicated expression. The classic example is summing an array:

```
for( sum=i=0; i<n; i++ )
    sum += array[i];
```

But the spaces around the operator are critical! For

```
x = -10;
```

also decreases `x` by 10. This is quite contrary to the experience of most programmers. In particular, watch out for things like

```
c=*s++;
y=&x[0];
```

both of which are almost certainly not what you wanted. Newer versions of various compilers are courteous enough to warn you about the ambiguity.

Because all other operators in an expression are evaluated before the assignment operator, the order of evaluation should be watched carefully:

```
x = x<<y | z;
```

means "shift x left y places, then OR with z, and store in x."

Floating Point

C has single and double precision numbers. For example,

```
double sum;
float avg, y[10];
sum = 0.0;
for( i=0; i<n; i++ )
    sum += y[i]; avg = sum/n;
```

All floating arithmetic is done in double precision. Mixed mode arithmetic is legal; if an arithmetic operator in an expression has both operands `int` or `char`, the arithmetic done is integer, but if one operand is `int` or `char` and the other is `float` or `double`, both operands are converted to `double`. Thus if `i` and `j` are `int` and `x` is `float`,

<code>(x+i)/j</code>	<i>converts i and j to float</i>
<code>x + i/j</code>	<i>does i/j integer, then converts</i>

Type conversion may be made by assignment; for instance,

```
int m, n;
float x, y;
m = x;
y = n;
```

converts `x` to integer (truncating toward zero), and `n` to floating point.

Floating constants are just like those in Fortran or PL/I, except that the exponent letter is 'e' instead of 'E'. Thus:

```
pi = 3.14159;
large = 1.23456789e10;
```

`printf` will format floating point numbers: ```%w.df''` in the format string will print the corresponding variable in a field `w` digits wide, with `d` decimal places. An `e` instead of an `f` will produce exponential notation.

goto and labels

C has a `goto` statement and labels, so you can branch about the way you used to. But most of the time `goto`'s aren't needed. (How many have we used up to this point?) The code can almost always be more clearly expressed by `for/while`, `if/else`, and compound statements.

One use of `goto`'s with some legitimacy is in a program which contains a long loop, where a `while(1)` would be too extended. Then you might write

```
mainloop:
    ...
    goto mainloop;
```

Another use is to implement a `break` out of more than one level of `for` or `while`. `goto`'s can only branch to labels within the same function.

Manipulating Strings

A string is a sequence of characters, with its beginning indicated by a pointer and its end marked by the null character `\0`. At times, you need to know the length of a string (the number of characters between the start and the end of the string) [5]. This length is obtained with the library function `strlen()`. Its prototype, in `STRING.H`, is

```
size_t strlen(char *str);
```

The strcpy() Function

The library function `strcpy()` copies an entire string to another memory location. Its prototype is as follows:

```
char *strcpy( char *destination, char *source );
```

Before using `strcpy()`, you must allocate storage space for the destination string.

```
/* Demonstrates strcpy(). */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char source[] = "The source string.";
```

```

main()
{
    char dest1[80];
    char *dest2, *dest3;

    printf("\nsource: %s", source );

    /* Copy to dest1 is okay because dest1 points to */
    /* 80 bytes of allocated space. */

    strcpy(dest1, source);
    printf("\ndest1: %s", dest1);

    /* To copy to dest2 you must allocate space. */
    dest2 = (char *)malloc(strlen(source) +1);
    strcpy(dest2, source);
    printf("\ndest2: %s\n", dest2);

    return(0);
}
source: The source string.
dest1:  The source string.
dest2:  The source string.

```

The strncpy() Function

The `strncpy()` function is similar to `strcpy()`, except that `strncpy()` lets you specify how many characters to copy. Its prototype is

```
char *strncpy(char *destination, char *source, size_t n);
```

```

/* Using the strncpy() function. */

#include <stdio.h>
#include <string.h>

char dest[] = ".....";
char source[] = "abcdefghijklmnopqrstuvwxyz";

main()
{
    size_t n;

    while (1)
    {
        puts("Enter the number of characters to copy (1-26)");
        scanf("%d", &n);

        if (n > 0 && n < 27)
            break;
    }
}

```

```

printf("\nBefore strncpy destination = %s", dest);

strncpy(dest, source, n);

printf("\nAfter strncpy destination = %s\n", dest);
return(0); }

Enter the number of characters to copy (1-26)
15
Before strncpy destination = .....
After strncpy destination = abcdefghijklmno.....

```

The strdup() Function

The library function `strdup()` is similar to `strcpy()`, except that `strdup()` performs its own memory allocation for the destination string with a call to `malloc()`. The prototype for `strdup()` is

```
char *strdup( char *source );
```

Using `strdup()` to copy a string with automatic memory allocation.

```

/* The strdup() function. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char source[] = "The source string.";

main()
{
    char *dest;

    if ( (dest = strdup(source)) == NULL)
    {
        fprintf(stderr, "Error allocating memory.");
        exit(1);
    }

    printf("The destination = %s\n", dest);
    return(0);
}
The destination = The source string.

```


The strcat() Function

The prototype of `strcat()` is

```
char *strcat(char *str1, char *str2);
```

The function appends a copy of `str2` onto the end of `str1`, moving the terminating null character to the end of the new string. You must allocate enough space for `str1` to hold the resulting string. The return value of `strcat()` is a pointer to `str1`. Following listing demonstrates `strcat()`.

```
/* The strcat() function. */  
  
#include <stdio.h>  
#include <string.h>  
  
char str1[27] = "a";  
char str2[2];  
  
main()  
{  
    int n;  
  
    /* Put a null character at the end of str2[]. */  
  
    str2[1] = '\0';  
  
    for (n = 98; n < 123; n++)  
    {  
        str2[0] = n;  
        strcat(str1, str2);  
        puts(str1);  
    }  
    return(0);  
}
```

```
ab  
abc  
abcd  
abcde  
abcdef  
abcdefg  
abcdefgh  
abcdefghi  
abcdefghij  
abcdefghijk  
abcdefghijkl  
abcdefghijklm  
abcdefghijklmn  
abcdefghijklmno
```

```
abcdefghijklmnop  
abcdefghijklmnopq  
abcdefghijklmnopqr  
abcdefghijklmnopqrs  
abcdefghijklmnopqrst  
abcdefghijklmnopqrstu  
abcdefghijklmnopqrstuv  
abcdefghijklmnopqrstuvw  
abcdefghijklmnopqrstuvwx  
abcdefghijklmnopqrstuvwxy  
abcdefghijklmnopqrstuvwxyz
```

Comparing Strings

Strings are compared to determine whether they are equal or unequal. If they are unequal, one string is "greater than" or "less than" the other. Determinations of "greater" and "less" are made with the ASCII codes of the characters. In the case of letters, this is equivalent to alphabetical order, with the one seemingly strange exception that all uppercase letters are "less than" the lowercase letters. This is true because the uppercase letters have ASCII codes 65 through 90 for A through Z, while lowercase a through z are represented by 97 through 122. Thus, "ZEBRA" would be considered to be less than "apple" by these C functions.

The ANSI C library contains functions for two types of string comparisons: comparing two entire strings, and comparing a certain number of characters in two strings.

Comparing Two Entire Strings

The function `strcmp()` compares two strings character by character. Its prototype is

```
int strcmp(char *str1, char *str2);
```

The arguments `str1` and `str2` are pointers to the strings being compared. The function's return values are given in Table. Following Listing demonstrates `strcmp()`.

The values returned by `strcmp()`.

Return Value	Meaning
< 0	str1 is less than str2.
0	str1 is equal to str2.
> 0	str1 is greater than str2.

Using strcmp() to compare strings.

```
/* The strcmp() function. */

#include <stdio.h>
#include <string.h>

main()
{
    char str1[80], str2[80];
    int x;

    while (1)
    {

        /* Input two strings. */
        printf("\n\nInput the first string, a blank to exit: ");
        gets(str1);

        if ( strlen(str1) == 0 )
            break;

        printf("\nInput the second string: ");
        gets(str2);

        /* Compare them and display the result. */

        x = strcmp(str1, str2);

        printf("\nstrcmp(%s,%s) returns %d", str1, str2, x);
    }
    return(0);
}
```

```
Input the first string, a blank to exit: First string
Input the second string: Second string
strcmp(First string,Second string) returns -1
Input the first string, a blank to exit: test string
Input the second string: test string
strcmp(test string,test string) returns 0
Input the first string, a blank to exit: zebra
Input the second string: aardvark
strcmp(zebra,aardvark) returns 1
Input the first string, a blank to exit:
```

Comparing Partial Strings

The library function strncmp() compares a specified number of characters of one string to another string. Its prototype is

```
int strncmp(char *str1, char *str2, size_t n);
```

The function `strncmp()` compares `n` characters of `str2` to `str1`. The comparison proceeds until `n` characters have been compared or the end of `str1` has been reached. The method of comparison and return values are the same as for `strcmp()`. The comparison is case-sensitive.

Comparing parts of strings with `strncmp()`.

```
/* The strncmp() function. */
#include <stdio.h>
#include [Sigma]>tring.h>

char str1[] = "The first string.";
char str2[] = "The second string.";
main()
{
    size_t n, x;

    puts(str1);
    puts(str2);

    while (1)
    {
        puts("\n\nEnter number of characters to compare, 0 to exit.");
        scanf("%d", &n);

        if (n <= 0)
            break;

        x = strncmp(str1, str2, n);

        printf("\nComparing %d characters, strncmp() returns %d.", n, x);
    }
    return(0);
}
```

```
The first string.
The second string.
Enter number of characters to compare, 0 to exit.
3
Comparing 3 characters, strncmp() returns .@]
Enter number of characters to compare, 0 to exit.
6
Comparing 6 characters, strncmp() returns -1.
Enter number of characters to compare, 0 to exit.
0
```

The strchr() Function

The strchr() function finds the first occurrence of a specified character in a string. The prototype is

```
char *strchr(char *str, int ch);
```

The function strchr() searches str from left to right until the character ch is found or the terminating null character is found. If ch is found, a pointer to it is returned. If not, NULL is returned.

When strchr() finds the character, it returns a pointer to that character. Knowing that str is a pointer to the first character in the string, you can obtain the position of the found character by subtracting str from the pointer value returned by strchr(). Following Listing illustrates this. Remember that the first character in a string is at position 0. Like many of C's string functions, strchr() is case-sensitive. For example, it would report that the character F isn't found in the string raffle.

Using strchr() to search a string for a single character.

```
/* Searching for a single character with strchr(). */

#include <stdio.h>
#include <string.h>

main()
{
    char *loc, buf[80];
    int ch;

    /* Input the string and the character. */

    printf("Enter the string to be searched: ");
    gets(buf);
    printf("Enter the character to search for: ");
    ch = getchar();

    /* Perform the search. */

    loc = strchr(buf, ch);

    if ( loc == NULL )
        printf("The character %c was not found.", ch);
    else
        printf("The character %c was found at position %d.\n",
            ch, loc-buf);
    return(0); }
```

```
Enter the string to be searched: How now Brown Cow?
Enter the character to search for: C
The character C was found at position 14.
```

The strcspn() Function

The library function `strcspn()` searches one string for the first occurrence of any of the characters in a second string. Its prototype is

```
size_t strcspn(char *str1, char *str2);
```

The function `strcspn()` starts searching at the first character of `str1`, looking for any of the individual characters contained in `str2`. This is important to remember. The function doesn't look for the string `str2`, but only the characters it contains. If the function finds a match, it returns the offset from the beginning of `str1`, where the matching character is located. If it finds no match, `strcspn()` returns the value of `strlen(str1)`. This indicates that the first match was the null character terminating the string

Searching for a set of characters with `strcspn()`.

```
/* Searching with strcspn(). */

#include <stdio.h>
#include <string.h>

main()
{
    char buf1[80], buf2[80];
    size_t loc;

    /* Input the strings. */

    printf("Enter the string to be searched: ");
    gets(buf1);
    printf("Enter the string containing target characters: ");
    gets(buf2);

    /* Perform the search. */

    loc = strcspn(buf1, buf2);

    if ( loc == strlen(buf1) )
        printf("No match was found.");
    else
        printf("The first match was found at position %d.\n", loc);
    return(0);
}
```

```
Enter the string to be searched: How now Brown Cow?
Enter the string containing target characters: Cat
The first match was found at position 14.
```

The strpbrk() Function

The library function `strpbrk()` is similar to `strcspn()`, searching one string for the first occurrence of any character contained in another string. It differs in that it doesn't include the terminating null characters in the search. The function prototype is

```
char *strpbrk(char *str1, char *str2);
```

The function `strpbrk()` returns a pointer to the first character in `str1` that matches any of the characters in `str2`. If it doesn't find a match, the function returns `NULL`. As previously explained for the function `strchr()`, you can obtain the offset of the first match in `str1` by subtracting the pointer `str1` from the pointer returned by `strpbrk()` (if it isn't `NULL`, of course).

The strstr() Function

The final, and perhaps most useful, C string-searching function is `strstr()`. This function searches for the first occurrence of one string within another, and it searches for the entire string, not for individual characters within the string. Its prototype is

```
char *strstr(char *str1, char *str2);
```

The function `strstr()` returns a pointer to the first occurrence of `str2` within `str1`. If it finds no match, the function returns `NULL`. If the length of `str2` is 0, the function returns `str1`. When `strstr()` finds a match, you can obtain the offset of `str2` within `str1` by pointer subtraction, as explained earlier for `strchr()`. The matching procedure that `strstr()` uses is case-sensitive.

Using strstr() to search for one string within another.

```
/* Searching with strstr(). */

#include <stdio.h>
#include <string.h>

main()
{
    char *loc, buf1[80], buf2[80];

    /* Input the strings. */

    printf("Enter the string to be searched: ");
```

```
gets(buf1);
printf("Enter the target string: ");
gets(buf2);

/* Perform the search. */

loc = strstr(buf1, buf2);

if ( loc == NULL )
    printf("No match was found.\n");
else
    printf("%s was found at position %d.\n", buf2, loc-buf1);
return(0);}
Enter the string to be searched: How now brown cow?
Enter the target string: cow
Cow was found at position 14.
```

The strrev() Function

The function `strrev()` reverses the order of all the characters in a string (Not ANSI Standard). Its prototype is

```
char *strrev(char *str);
```

The order of all characters in `str` is reversed, with the terminating null character remaining at the end.

String-to-Number Conversions

Sometimes you will need to convert the string representation of a number to an actual numeric variable. For example, the string "123" can be converted to a type `int` variable with the value 123. Three functions can be used to convert a string to a number. They are explained in the following sections; their prototypes are in `STDLIB.H`.

The atoi() Function

The library function `atoi()` converts a string to an integer. The prototype is

```
int atoi(char *ptr);
```

The function `atoi()` converts the string pointed to by `ptr` to an integer. Besides digits, the string can contain leading white space and a `+` or `--` sign. Conversion starts at the beginning

of the string and proceeds until an unconvertible character (for example, a letter or punctuation mark) is encountered. The resulting integer is returned to the calling program. If it finds no convertible characters, `atoi()` returns 0. Table lists some examples.

String-to-number conversions with `atoi()`.

String	Value Returned by <code>atoi()</code>
"157"	157
"-1.6"	-1
"+50x"	50
"twelve"	0
"x506"	0

The `atol()` Function

The library function `atol()` works exactly like `atoi()`, except that it returns a type `long`. The function prototype is

```
long atol(char *ptr);
```

The `atof()` Function

The function `atof()` converts a string to a type `double`. The prototype is

```
double atof(char *str);
```

The argument `str` points to the string to be converted. This string can contain leading white space and a `+` or `--` character. The number can contain the digits 0 through 9, the decimal point, and the exponent indicator `E` or `e`. If there are no convertible characters, `atof()` returns 0. Table 17.3 lists some examples of using `atof()`.

String-to-number conversions with `atof()`.

String	Value Returned by <code>atof()</code>
"12"	12.000000
"-0.123"	-0.123000
"123E+3"	123000.000000
"123.1e-5"	0.001231

Character Test Functions

The header file CTYPE.H contains the prototypes for a number of functions that test characters, returning TRUE or FALSE depending on whether the character meets a certain condition. For example, is it a letter or is it a numeral? The `isxxxx()` functions are actually macros, defined in CTYPE.H.

The `isxxxx()` macros all have the same prototype:

```
int isxxxx(int ch);
```

In the preceding line, `ch` is the character being tested. The return value is TRUE (nonzero) if the condition is met or FALSE (zero) if it isn't. Table lists the complete set of `isxxxx()` macros.

The `isxxxx()` macros.

Macro	Action
<code>isalnum()</code>	Returns TRUE if <code>ch</code> is a letter or a digit.
<code>isalpha()</code>	Returns TRUE if <code>ch</code> is a letter.
<code>isascii()</code>	Returns TRUE if <code>ch</code> is a standard ASCII character (between 0 and 127).
<code>isctrl()</code>	Returns TRUE if <code>ch</code> is a control character.
<code>isdigit()</code>	Returns TRUE if <code>ch</code> is a digit.
<code>isgraph()</code>	Returns TRUE if <code>ch</code> is a printing character (other than a space).
<code>islower()</code>	Returns TRUE if <code>ch</code> is a lowercase letter.
<code>isprint()</code>	Returns TRUE if <code>ch</code> is a printing character (including a space).
<code>ispunct()</code>	Returns TRUE if <code>ch</code> is a punctuation character.
<code>isspace()</code>	Returns TRUE if <code>ch</code> is a whitespace character (space, tab, vertical tab, line feed, form feed, or carriage return).
<code>isupper()</code>	Returns TRUE if <code>ch</code> is an uppercase letter.
<code>isxdigit()</code>	Returns TRUE if <code>ch</code> is a hexadecimal digit (0 through 9, a through f, A through F).

You can do many interesting things with the character-test macros. One example is the function `get_int()`, shown in Listing. This function inputs an integer from `stdin` and returns it as a type `int` variable. The function skips over leading white space and returns 0 if the first nonspace character isn't a numeric character.

Using the `isxxxx()` macros to implement a function that inputs an integer.

```
/* Using character test macros to create an integer */
/* input function. */
#include <stdio.h>
```

```
#include <ctype.h>

int get_int(void);

main()
{
    int x;
    x = get_int();
    printf("You entered %d.\n", x);
}

int get_int(void)
{
    int ch, i, sign = 1;

    while ( isspace(ch = getchar()) );

    if (ch != '-' && ch != '+' && !isdigit(ch) && ch != EOF)
    {
        ungetc(ch, stdin);
        return 0;
    }

    /* If the first character is a minus sign, set */
    /* sign accordingly. */

    if (ch == '-')
        sign = -1;

    /* If the first character was a plus or minus sign, */
    /* get the next character. */

    if (ch == '+' || ch == '-')
        ch = getchar();

    /* Read characters until a nondigit is input. Assign */
    /* values, multiplied by proper power of 10, to i. */

    for (i = 0; isdigit(ch); ch = getchar() )
        i = 10 * i + (ch - '0');

    /* Make result negative if sign is negative. */

    i *= sign;

    /* If EOF was not encountered, a nondigit character */
    /* must have been read in, so unget it. */
    if (ch != EOF)
        ungetc(ch, stdin);

    /* Return the input value. */

    return i;
}
```

```

-100
You entered -100.
abc3.145
You entered 0.
9 9 9
You entered 9.
2.5
You entered 2.

```

Mathematical Functions

The C standard library contains a variety of functions that perform mathematical operations. Prototypes for the mathematical functions are in the header file MATH.H. The math functions all return a type double. For the trigonometric functions, angles are expressed in radians. Remember, one radian equals 57.296 degrees, and a full circle (360 degrees) contains 2π radians.

Trigonometric Functions

Function	Prototype	Description
acos()	double acos(double x)	Returns the arccosine of its argument. The argument must be in the range $-1 \leq x \leq 1$, and the return value is in the range $0 \leq \text{acos} \leq \pi$.
asin()	double asin(double x)	Returns the arcsine of its argument. The argument must be in the range $-1 \leq x \leq 1$, and the return value is in the range $-\pi/2 \leq \text{asin} \leq \pi/2$.
atan()	double atan(double x)	Returns the arctangent of its argument. The return value is in the range $-\pi/2 \leq \text{atan} \leq \pi/2$.
atan2()	double atan2(double x, double y)	Returns the arctangent of x/y . The value returned is in the range $-\pi \leq \text{atan2} \leq \pi$.
cos()	double cos(double x)	Returns the cosine of its argument.
sin()	double sin(double x)	Returns the sine of its argument.
tan()	double tan(double x)	Returns the tangent of its argument.

Exponential and Logarithmic Functions

Function	Prototype	Description
exp()	double exp(double x)	Returns the natural exponent of its argument, that is, e^x where e equals 2.7182818284590452354.
log()	double log(double x)	Returns the natural logarithm of its argument. The argument must be greater than 0.
log10()	double log10	Returns the base-10 logarithm of its argument. The argument must be greater than 0.

Hyperbolic Functions

Function	Prototype	Description
cosh()	double cosh(double x)	Returns the hyperbolic cosine of its argument.
sinh()	double sinh(double x)	Returns the hyperbolic sine of its argument.
tanh()	double tanh(double x)	Returns the hyperbolic tangent of its argument.

Other Mathematical Functions

Function	Prototype	Description
sqrt()	double sqrt(double x)	Returns the square root of its argument. The argument must be zero or greater.
ceil()	double ceil(double x)	Returns the smallest integer not less than its argument. For example, ceil(4.5) returns 5.0, and ceil(-4.5) returns -4.0. Although ceil() returns an integer value, it is returned as a type double.
abs()	int abs(int x)	Returns the absolute
labs()	long labs(long x)	value of their arguments.
floor()	double floor(double x)	Returns the largest integer not greater than its argument. For example, floor(4.5) returns 4.0, and floor(-4.5) returns -5.0.
modf()	double modf(double x, double *y)	Splits x into integral and fractional parts, each with the same sign as x. The fractional part is returned by the function, and the integral part is assigned to *y.
pow()	double pow(double x, double y)	Returns x^y . An error occurs if $x == 0$ and $y <= 0$, or if $x < 0$ and y is not an integer.

CHAPTER 4 ESSENTIAL DATA STRUCTURES

In every algorithm, there is a need to store data. Ranging from storing a single value in a single variable, to more complex data structures. In programming contests, there are several aspect of data structures to consider when selecting the proper way to represent the data for a problem. This chapter will give you some guidelines and list some basic data structures to start with.^[2]

Will it work?

If the data structures won't work, it's not helpful at all. Ask yourself what questions the algorithm will need to be able to ask the data structure, and make sure the data structure can handle it. If not, then either more data must be added to the structure, or you need to find a different representation.

Can I code it?

If you don't know or can't remember how to code a given data structure, pick a different one. Make sure that you have a good idea how each of the operations will affect the structure of the data.

Another consideration here is memory. Will the data structure fit in the available memory? If not, compact it or pick a new one. Otherwise, it is already clear from the beginning that it won't work.

Can I code it in time? or has my programming language support it yet?

As this is a timed contest, you have three to five programs to write in, say, five hours. If it'll take you an hour and a half to code just the data structure for the first problem, then you're almost certainly looking at the wrong structure.

Another very important consideration is, whether your programming language has actually provide you the required data structure. For C++ programmers, STL has a wide options of a very good built-in data structure, what you need to do is to master them, rather than trying to built your own data structure.

In contest time, this will be one of the winning strategy. Consider this scenario. All teams are given a set of problems. Some of them required the usage of 'stack'. The best programmer from team A directly implement the best known stack implementation, he need 10 minutes to do so. Surprisingly for them, other teams type in only two lines:

"#include <stack>" and "std::stack<int> s;", can you see who have 10 minutes advantage now?

Can I debug it?

It is easy to forget this particular aspect of data structure selection. Remember that a program is useless unless it works. Don't forget that debugging time is a large portion of the contest time, so include its consideration in calculating coding time.

What makes a data structure easy to debug? That is basically determined by the following two properties.

1. State is easy to examine. The smaller, more compact the representation, in general, the easier it is to examine. Also, statically allocated arrays are **much** easier to examine than linked lists or even dynamically allocated arrays.

2. State can be displayed easily. For the more complex data structures, the easiest way to examine them is to write a small routine to output the data. Unfortunately, given time constraints, you'll probably want to limit yourself to text output. This means that structures like trees and graphs are going to be difficult to examine.

Is it fast?

The usage of more sophisticated data structure will reduce the amount of overall algorithm complexity. It will be better if you know some advanced data structures.

Things to Avoid: Dynamic Memory

In general, you should avoid dynamic memory, because:

1. It is too easy to make mistakes using dynamic memory. Overwriting past allocated memory, not freeing memory, and not allocating memory are only some of the mistakes that are introduced when dynamic memory is used. In addition, the failure modes for these errors are such that it's hard to tell where the error occurred, as it's likely to be at a (potentially much later) memory operation.

2. It is too hard to examine the data structure's contents. The interactive development environments available don't handle dynamic memory well, especially for C. Consider parallel arrays as an alternative to dynamic memory. One way to do a linked list, where instead of keeping a next point, you keep a second array, which has the index of the next

element. Sometimes you may have to dynamically allocate these, but as it should only be done once, it's much easier to get right than allocating and freeing the memory for each insert and delete.

All of this notwithstanding, sometimes dynamic memory is the way to go, especially for large data structures where the size of the structure is not known until you have the input.

Things to Avoid: Coolness Factor

Try not to fall into the 'coolness' trap. You may have just seen the neatest data structure, but remember:

1. Cool ideas that don't work aren't.
2. Cool ideas that'll take forever to code aren't, either

It's much more important that your data structure and program work than how impressive your data structure is.

Basic Data Structures

There are five basic data structures: arrays, linked lists, stacks, queues, and deque (pronounced deck, a double ended queue). It will not be discussed in this section, go for their respective section.

Arrays

Array is the most useful data structures, in fact, this data structure will almost always used in all contest problems. Lets look at the good side first: if index is known, searching an element in an array is very fast, $O(1)$, this good for looping/iteration. Array can also be used to implement other sophisticated data structures such as stacks, queues, hash tables. However, being the easiest data structure doesn't mean that array is efficient. In some cases array can be very inefficient. Moreover, in standard array, the size is fixed. If you don't know the input size beforehand, it may be wiser to use vector (a resizable array). Array also suffer a very slow insertion in ordered array, another slow searching in unordered array, and unavoidable slow deletion because you have to shift all elements.

Search	$O(n/2)$ comparisons	$O(n)$ comparisons
Insertion	No comparison, $O(1)$	No comparisons, $O(1)$
Deletion	$O(n/2)$ comparisons, $O(n/2)$ moves	$O(n)$ comparisons more than $O(n/2)$ moves

We commonly use one-dimensional array for standard use and two-dimensional array to represent matrices, board, or anything that two-dimensional. Three-dimensional is rarely used to model 3D situations.

Row major, cache-friendly, use this method to access all items in array sequentially.

```
for (i=0; i<numRow; i++)    // ROW FIRST = EFFECTIVE
    for (j=0; j<numCol; j++)
        array[i][j] = 0;
```

Column major, NOT cache-friendly, **DO NOT** use this method or you'll get very poor performance. Why? you will learn this in Computer Architecture course. For now, just take note that computer access array data in row major.

```
for (j=0; j<numCol; j++)    // COLUMN FIRST = INEFFECTIVE
    for (i=0; i<numRow; i++)
        array[i][j] = 0;
```

Vector Trick - A resizable array

Static array has a drawback when the size of input is not known beforehand. If that is the case, you may want to consider vector, a resizable array.

There are other data structures which offer much more efficient resizing capability such as Linked List, etc.

TIPS: UTILIZING C++ VECTOR STL

C++ STL has vector template for you.

```
#include <stdio.h>
#include <vector>
#include <algorithm>

using namespace std;

void main() {
    // just do this, write vector<the type you want,
    // in this case, integer> and the vector name
    vector<int> v;

    // try inserting 7 different integers, not ordered
    v.push_back(3); v.push_back(1); v.push_back(2);
    v.push_back(7); v.push_back(6); v.push_back(5);
    v.push_back(4);
```

```
// to access the element, you need an iterator...
vector<int>::iterator i;

printf("Unsorted version\n");
// start with 'begin', end with 'end', advance with i++
for (i = v.begin(); i!= v.end(); i++)
    printf("%d ",*i); // iterator's pointer hold the value
printf("\n");

sort(v.begin(),v.end()); // default sort, ascending

printf("Sorted version\n");
for (i = v.begin(); i!= v.end(); i++)
    printf("%d ",*i); // iterator's pointer hold the value
printf("\n");
}
```

Linked List

Motivation for using linked list: Array is static and even though it has $O(1)$ access time if index is known, Array must shift its elements if an item is going to be inserted or deleted and this is absolutely inefficient. Array cannot be resized if it is full (see resizable array - vector for the trick but slow resizable array).

Linked list can have a very fast insertion and deletion. The physical location of data in linked list can be anywhere in the memory but each node must know which part in memory is the next item after them.

Linked list can be any big as you wish as long there is sufficient memory. The side effect of Linked list is there will be wasted memory when a node is "deleted" (only flagged as deleted). This wasted memory will only be freed up when garbage collector doing its action (depends on compiler / Operating System used).

Linked list is a data structure that is commonly used because of it's dynamic feature. Linked list is not used for fun, it's very complicated and have a tendency to create run time memory access error). Some programming languages such as Java and C++ actually support Linked list implementation through API (Application Programming Interface) and STL (Standard Template Library).

Linked list is composed of a data (and sometimes pointer to the data) and a pointer to next item. In Linked list, you can only find an item through complete search from head until it found the item or until tail (not found). This is the bad side for Linked list, especially for a very long list. And for insertion in Ordered Linked List, we have to search for appropriate place using Complete Search method, and this is slow too. (There are some tricks to improve searching in Linked List, such as remembering references to specific nodes, etc).

Variations of Linked List

With tail pointer

Instead of standard head pointer, we use another pointer to keep track of the last item. This is useful for queue-like structures since in Queue, we enter Queue from rear (tail) and delete item from the front (head).

With dummy node (sentinels)

This variation is to simplify our code (I prefer this way), It can simplify empty list code and inserting to the front of the list code.

Doubly Linked List

Efficient if we need to traverse the list in both directions (forward and backward). Each node now has 2 pointers, one points to the next item, the other one points to the previous item. We need dummy head & dummy tail for this type of linked list.

TIPS: UTILIZING C++ LIST STL

A demo on the usage of STL list. The underlying data structure is a doubly linked list.

```
#include <stdio.h>

// this is where list implementation resides
#include <list>

// use this to avoid specifying "std::" everywhere
using namespace std;

// just do this, write list<the type you want,
// in this case, integer> and the list name
list<int> l;
list<int>::iterator i;

void print() {
    for (i = l.begin(); i != l.end(); i++)
        printf("%d ", *i); // remember... use pointer!!!
    printf("\n");
}

void main() {
    // try inserting 8 different integers, has duplicates
    l.push_back(3); l.push_back(1); l.push_back(2);
    l.push_back(7); l.push_back(6); l.push_back(5);
    l.push_back(4); l.push_back(7);
    print();
}
```

```
l.sort(); // sort the list, wow sorting linked list...
print();

l.remove(3); // remove element '3' from the list
print();

l.unique(); // remove duplicates in SORTED list!!!
print();

i = l.begin(); // set iterator to head of the list
i++; // 2nd node of the list
l.insert(i,1,10); // insert 1 copy of '10' here
print();
}
```

Stack

A data structures which only allow insertion (**push**) and deletion (**pop**) from the top only. This behavior is called **Last In First Out (LIFO)**, similar to normal stack in the real world.

Important stack operations

1. Push (C++ STL: push())
Adds new item at the top of the stack.
2. Pop (C++ STL: pop())
Retrieves and removes the top of a stack.
3. Peek (C++ STL: top())
Retrieves the top of a stack without deleting it.
4. IsEmpty (C++ STL: empty())
Determines whether a stack is empty.

Some stack applications

1. To model "real stack" in computer world: Recursion, Procedure Calling, etc.
2. Checking palindrome (although checking palindrome using Queue & Stack is 'stupid').
3. To read an input from keyboard in text editing with backspace key.
4. To reverse input data, (another stupid idea to use stack for reversing data).
5. Checking balanced parentheses.
6. Postfix calculation.
7. Converting mathematical expressions. Prefix, Infix, or Postfix.

Some stack implementations

1. Linked List with head pointer only (Best)
2. Array
3. Resizable Array

TIPS: UTILIZING C++ STACK STL

Stack is not difficult to implement. Stack STL's implementation is very efficient, even though it will be slightly slower than your custom made stack.

```
#include <stdio.h>
#include <stack>

using namespace std;

void main() {
    // just do this, write stack<the type you want,
    // in this case, integer> and the stack name
    stack<int> s;

    // try inserting 7 different integers, not ordered
    s.push(3); s.push(1); s.push(2);
    s.push(7); s.push(6); s.push(5);
    s.push(4);

    // the item that is inserted first will come out last
    // Last In First Out (LIFO) order...
    while (!s.empty()) {
        printf("%d ", s.top());
        s.pop();
    }
    printf("\n");}
```

Queue

A data structures which only allow insertion from the back (rear), and only allow deletion from the head (front). This behavior is called First In First Out (FIFO), similar to normal queue in the real world.

Important queue operations:

1. Enqueue (C++ STL: push())
Adds new item at the back (rear) of a queue.
2. Dequeue (C++ STL: pop())

Retrieves and removes the front of a queue at the back (rear) of a queue.

3. Peek (C++ STL: top())

Retrieves the front of a queue without deleting it.

4. IsEmpty (C++ STL: empty())

Determines whether a queue is empty.

TIPS: UTILIZING C++ QUEUE STL

Standard queue is also not difficult to implement. Again, why trouble yourself, just use C++ queue STL.

```
#include <stdio.h>
#include <queue>

// use this to avoid specifying "std::" everywhere
using namespace std;

void main() {
    // just do this, write queue<the type you want,
    // in this case, integer> and the queue name
    queue<int> q;

    // try inserting 7 different integers, not ordered
    q.push(3); q.push(1); q.push(2);
    q.push(7); q.push(6); q.push(5);
    q.push(4);

    // the item that is inserted first will come out first
    // First In First Out (FIFO) order...
    while (!q.empty()) {
        // notice that this is not "top()" !!!
        printf("%d ", q.front());
        q.pop();
    }
    printf("\n");}
```

CHAPTER 5 INPUT/OUTPUT TECHNIQUES

In all programming contest, or more specifically, in all useful program, you need to read in input and process it. However, the input data can be as nasty as possible, and this can be very troublesome to parse.^[2]

If you spent too much time in coding how to parse the input efficiently and you are using C/C++ as your programming language, then this tip is for you. Let's see an example:

How to read the following input:

```
1 2 2 3 1 2 3 1 2
```

The fastest way to do it is:

```
#include <stdio.h>
int N;
void main() {
while (scanf("%d", &N)==1){
    process N.. }
}
```

There are N lines, each lines always start with character '0' followed by '.', then unknown number of digits x, finally the line always terminated by three dots "...".

```
N
0.xxxx...
```

The fastest way to do it is:

```
#include <stdio.h>
char digits[100];

void main() {
scanf("%d", &N);
for (i=0; i<N; i++) {
    scanf("0.%[0-9]...", &digits); // surprised?
    printf("the digits are 0.%s\n", digits);
}
}
```

This is the trick that many C/C++ programmers doesn't aware of. Why we said C++ while scanf/printf is a standard C I/O routines? This is because many C++ programmers

"forcing" themselves to use cin/cout all the time, without realizing that scanf/printf can still be used inside all C++ programs.

Mastery of programming language I/O routines will help you a lot in programming contests.

Advanced use of printf() and scanf()

Those who have forgotten the advanced use of printf() and scanf(), recall the following examples:

```
scanf("%[ABCDEFGHijklmnopqrstuvwxyz]", &line); //line is a string
```

This scanf() function takes only uppercase letters as input to line and any other characters other than A..Z terminates the string. Similarly the following scanf() will behave like gets():

```
scanf("%[^\n]", line); //line is a string
```

Learn the default terminating characters for scanf(). Try to read all the advanced features of scanf() and printf(). This will help you in the long run.

Using new line with scanf()

If the content of a file (input.txt) is

```
abc  
def
```

And the following program is executed to take input from the file:

```
char input[100], ch;  
void main(void)  
{  
    freopen("input.txt", "rb", stdin);  
    scanf("%s", &input);  
    scanf("%c", &ch);  
}
```


The following is a slight modification to the code:

```
char input[100],ch;
void main(void)
{
    freopen("input.txt","rb",stdin);
    scanf("%s\n",&input);
    scanf("%c",&ch);
}
```

What will be their value now? The value of ch will be 'n' for the first code and 'd' for the second code.

Be careful about using gets() and scanf() together !

You should also be careful about using gets() and scanf() in the same program. Test it with the following scenario. The code is:

```
scanf("%s\n",&dummy);
gets(name);
```

And the input file is:

```
ABCDEF
bbbbbbXXX
```

What do you get as the value of name? "XXX" or "bbbbbbXXX" (Here, "b" means blank or space)

Multiple input programs

"Multiple input programs" are an invention of the online judge. The online judge often uses the problems and data that were first presented in live contests. Many solutions to problems presented in live contests take a single set of data, give the output for it, and terminate. This does not imply that the judges will give only a single set of data. The judges actually give multiple files as input one after another and compare the corresponding output files with the judge output. However, the Valladolid online judge gives only one file as input. It inserts all the judge inputs into a single file and at the top of that file, it writes how many sets of inputs there are. This number is the same as the number of input files the contest judges used. A blank line now separates each set of data. So the structure of the input file for multiple input program becomes:

```
Integer N //denoting the number of sets of input
--blank line---
input set 1 //As described in the problem statement
--blank line---

input set 2 //As described in the problem statement
--blank line---
input set 3 //As described in the problem statement
--blank line---
.
.
.
--blank line---
input set n //As described in the problem statement
--end of file--
```

Note that there should be no blank after the last set of data. The structure of the output file for a multiple input program becomes:

```
Output for set 1 //As described in the problem statement
--Blank line---
Output for set 2 //As described in the problem statement
--Blank line---
Output for set 3 //As described in the problem statement
--Blank line---
.
.
.
--blank line---
Output for set n //As described in the problem statement
--end of file--
```

The USU online judge does not have multiple input programs like Valladolid. It prefers to give multiple files as input and sets a time limit for each set of input.

Problems of multiple input programs

There are some issues that you should consider differently for multiple input programs. Even if the input specification says that the input terminates with the end of file (EOF), each set of input is actually terminated by a blank line, except for the last one, which is terminated by the end of file. Also, be careful about the initialization of variables. If they are not properly initialized, your program may work for a single set of data but give correct output for multiple sets of data. All global variables are initialized to their corresponding zeros.^[6]

CHAPTER 6 BRUTE FORCE METHOD

This is the most basic problem solving technique. Utilizing the fact that computer is actually very fast.

Complete search exploits the brute force, straight-forward, try-them-all method of finding the answer. This method should almost always be the first algorithm/solution you consider. If this works within time and space constraints, then do it: it's easy to code and usually easy to debug. This means you'll have more time to work on all the hard problems, where brute force doesn't work quickly enough.

Party Lamps

You are given N lamps and four switches. The first switch toggles all lamps, the second the even lamps, the third the odd lamps, and last switch toggles lamps 1,4,7,10,...

Given the number of lamps, N , the number of button presses made (up to 10,000), and the state of some of the lamps (e.g., lamp 7 is off), output all the possible states the lamps could be in.

Naively, for each button press, you have to try 4 possibilities, for a total of 4^{10000} (about 10^{6020}), which means there's no way you could do complete search (this particular algorithm would exploit recursion).

Noticing that the order of the button presses does not matter gets this number down to about 10000^4 (about 10^{16}), still too big to completely search (but certainly closer by a factor of over 10^{6000}).

However, pressing a button twice is the same as pressing the button no times, so all you really have to check is pressing each button either 0 or 1 times. That's only $2^4 = 16$ possibilities, surely a number of iterations solvable within the time limit.

The Clocks

A group of nine clocks inhabits a 3×3 grid; each is set to 12:00, 3:00, 6:00, or 9:00. Your goal is to manipulate them all to read 12:00. Unfortunately, the only way you can manipulate the clocks is by one of nine different types of move, each one of which rotates a certain subset of the clocks 90 degrees clockwise. Find the shortest sequence of moves which returns all the clocks to 12:00.

The 'obvious' thing to do is a recursive solution, which checks to see if there is a solution of 1 move, 2 moves, etc. until it finds a solution. This would take 9^k time, where k is the number of moves. Since k might be fairly large, this is not going to run with reasonable time constraints.

Note that the order of the moves does not matter. This reduces the time down to k^9 , which isn't enough of an improvement.

However, since doing each move 4 times is the same as doing it no times, you know that no move will be done more than 3 times. Thus, there are only 49 possibilities, which is only 262,072, which, given the rule of thumb for run-time of more than 10,000,000 operations in a second, should work in time. The brute-force solution, given this insight, is perfectly adequate.

It is beautiful, isn't it? If you have this kind of reasoning ability. Many seemingly hard problems is eventually solvable using brute force.

Recursion

Recursion is cool. Many algorithms need to be implemented recursively. A popular combinatorial brute force algorithm, backtracking, usually implemented recursively. After highlighting it's importance, lets study it.

Recursion is a function/procedure/method that calls itself again with a smaller range of arguments (break a problem into simpler problems) or with different arguments (it is useless if you use recursion with the same arguments). It keeps calling itself until something that is so simple / simpler than before (which we called base case), that can be solved easily or until an exit condition occurs.

A recursion must stop (actually, all program must terminate). In order to do so, a valid base case or end-condition must be reached. Improper coding will leads to stack overflow error.

You can determine whether a function recursive or not by looking at the source code. If in a function or procedure, it calls itself again, it's recursive type.

When a method/function/procedure is called:

- caller is suspended,
- "state" of caller saved,
- new space allocated for variables of new method.

Recursion is a powerful and elegant technique that can be used to solve a problem by solving a smaller problem of the same type. Many problems in Computer Science involve recursion and some of them are naturally recursive.

If one problem can be solved in both way (recursive or iterative), then choosing iterative version is a good idea since it is faster and doesn't consume a lot of memory. Examples: Factorial, Fibonacci, etc.

However, there are also problems that are can only be solved in recursive way or more efficient in recursive type or when it's iterative solutions are difficult to conceptualize. Examples: Tower of Hanoi, Searching (DFS, BFS), etc.

Types of recursion

There are 2 types of recursion, Linear recursion and multiple branch (Tree) recursion.

1. Linear recursion is a recursion whose order of growth is linear (not branched). Example of Linear recursion is Factorial, defined by $fac(n) = n * fac(n-1)$.

2. Tree recursion will branch to more than one node each step, growing up very quickly. It provides us a flexible ways to solve some logically difficult problem. It can be used to perform a Complete Search (To make the computer to do the trial and error). This recursion type has a quadratic or cubic or more order of growth and therefore, it is not suitable for solving "big" problems. It's limited to small. Examples: solving Tower of Hanoi, Searching (DFS, BFS), Fibonacci number, etc.

Some compilers can make some type of recursion to be iterative. One example is tail-recursion elimination. Tail-recursive is a type of recursive which the recursive call is the last command in that function/procedure. This

Tips on Recursion

1. Recursion is very similar to mathematical induction.
2. You first see how you can solve the base case, for $n=0$ or for $n=1$.
3. Then you assume that you know how to solve the problem of size $n-1$, and you look for a way of obtaining the solution for the problem size n from the solution of size $n-1$.

When constructing a recursive solution, keep the following questions in mind:

1. How can you define the problem in term of a smaller problem of the same type?
2. How does each recursive call diminish the size of the problem?

3. What instance of the problem can serve as the base case?
4. As the problem size diminishes, will you reach this base case?

Sample of a very standard recursion, Factorial (in Java):

```
static int factorial(int n) {
    if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

Divide and Conquer

This technique use analogy "smaller is simpler". Divide and conquer algorithms try to make problems simpler by dividing it to sub problems that can be solved easier than the main problem and then later it combine the results to produce a complete solution for the main problem.

Divide and Conquer algorithms: Quick Sort, Merge Sort, Binary Search.

Divide & Conquer has 3 main steps:

1. Divide data into parts
2. Find sub-solutions for each of the parts recursively (usually)
3. Construct the final answer for the original problem from the sub-solutions

```
DC(P) {
    if small(P) then
        return Solve(P);
    else {
        divide P into smaller instances P1,...,Pk, k>1;
        Apply DC to each of these subproblems;
        return combine (DC (P1) , ... ,DC (Pk) );
    }
}
```

Binary Search, is not actually follow the pattern of the full version of Divide & Conquer, since it never combine the sub problems solution anyway. However, lets use this example to illustrate the capability of this technique.

Binary Search will help you find an item in an array. The naive version is to scan through the array one by one (sequential search). Stopping when we found the item (success) or when we reach the end of array (failure). This is the simplest and the most inefficient way

to find an element in an array. However, you will usually end up using this method because not every array is ordered, you need sorting algorithm to sort them first.

Binary search algorithm is using Divide and Conquer approach to reduce search space and stop when it found the item or when search space is empty.

Pre-requisite to use binary search is the array must be an ordered array, the most commonly used example of ordered array for binary search is searching an item in a dictionary.

Efficiency of binary search is $O(\log n)$. This algorithm was named "binary" because it's behavior is to divide search space into 2 (binary) smaller parts).

Optimizing your source code

Your code must be fast. If it cannot be "fast", then it must at least "fast enough". If time limit is 1 minute and your currently program run in 1 minute 10 seconds, you may want to tweak here and there rather than overhaul the whole code again.

Generating vs Filtering

Programs that generate lots of possible answers and then choose the ones that are correct (imagine an 8-queen solver) are filters. Those that hone in exactly on the correct answer without any false starts are generators. Generally, filters are easier (faster) to code and run slower. Do the math to see if a filter is good enough or if you need to try and create a generator.

Pre-Computation / Pre-Calculation

Sometimes it is helpful to generate tables or other data structures that enable the fastest possible lookup of a result. This is called Pre-Computation (in which one trades space for time). One might either compile Pre-Computed data into a program, calculate it when the program starts, or just remember results as you compute them. A program that must translate letters from upper to lower case when they are in upper case can do a very fast table lookup that requires no conditionals, for example. Contest problems often use prime numbers - many times it is practical to generate a long list of primes for use elsewhere in a program.

Decomposition

While there are fewer than 20 basic algorithms used in contest problems, the challenge of combination problems that require a combination of two algorithms for solution is

daunting. Try to separate the cues from different parts of the problem so that you can combine one algorithm with a loop or with another algorithm to solve different parts of the problem independently. Note that sometimes you can use the same algorithm twice on different (independent!) parts of your data to significantly improve your running time.

Symmetries

Many problems have symmetries (e.g., distance between a pair of points is often the same either way you traverse the points). Symmetries can be 2-way, 4-way, 8-way, and more. Try to exploit symmetries to reduce execution time.

For instance, with 4-way symmetry, you solve only one fourth of the problem and then write down the four solutions that share symmetry with the single answer (look out for self-symmetric solutions which should only be output once or twice, of course).

Solving forward vs backward

Surprisingly, many contest problems work far better when solved backwards than when using a frontal attack. Be on the lookout for processing data in reverse order or building an attack that looks at the data in some order or fashion other than the obvious.

CHAPTER 7 MATHEMATICS

Base Number Conversion

Decimal is our most familiar base number, whereas computers are very familiar with binary numbers (octal and hexa too). The ability to convert between bases is important. Refer to mathematics book for this. ^[2]

TEST YOUR BASE CONVERSION KNOWLEDGE

Solve UVa problems related with base conversion:

[343 - What Base Is This?](#)

[353 - The Bases Are Loaded](#)

[389 - Basically Speaking](#)

Big Mod

Modulo (remainder) is important arithmetic operation and almost every programming language provide this basic operation. However, basic modulo operation is not sufficient if we are interested in finding a modulo of a very big integer, which will be difficult and has tendency for overflow. Fortunately, we have a good formula here:

$$(A*B*C) \bmod N == ((A \bmod N) * (B \bmod N) * (C \bmod N)) \bmod N.$$

To convince you that this works, see the following example:

`(7*11*23) mod 5 is 1; let's see the calculations step by step:`

```
((7 mod 5) * (11 mod 5) * (23 mod 5)) Mod 5 =
((2 * 1 * 3) Mod 5) =
(6 Mod 5) = 1
```

This formula is used in several algorithm such as in Fermat Little Test for primality testing:

$R = B^P \bmod M$ (B^P is a **very very big** number). By using the formula, we can get the correct answer without being afraid of overflow.

This is how to calculate **R** quickly (using Divide & Conquer approach, see exponentiation below for more details):

```

long bigmod(long b,long p,long m) {
    if (p == 0)
        return 1;
    else if (p%2 == 0)
        return square(bigmod(b,p/2,m)) % m; // square(x) = x * x
    else
        return ((b % m) * bigmod(b,p-1,m)) % m;
}

```

TEST YOUR BIG MOD KNOWLEDGE

Solve UVa problems related with Big Mod:

[374 - Big Mod](#)

[10229 - Modular Fibonacci](#) - plus Fibonacci

Big Integer

Long time ago, 16-bit integer is sufficient: $[-2^{15} \text{ to } 2^{15-1}] \sim [-32,768 \text{ to } 32,767]$.

When 32-bit machines are getting popular, 32-bit integers become necessary. This data type can hold range from $[-2^{31} \text{ to } 2^{31-1}] \sim [2,147,483,648 \text{ to } 2,147,483,647]$. Any integer with 9 or less digits can be safely computed using this data type. If you somehow must use the 10th digit, be careful of overflow.

But man is very greedy, 64-bit integers are created, referred as programmers as long long data type or int64 data type. It has an awesome range up that can covers 18 digits integers fully, plus the 19th digit partially $[-2^{63}-1 \text{ to } 2^{63-1}] \sim [9,223,372,036,854,775,808 \text{ to } 9,223,372,036,854,775,807]$. Practically this data type is safe to compute most of standard arithmetic problems, except some problems.

Note: for those who don't know, in C, `long long n` is read using: `scanf("%lld",&n)`; and `unsigned long long n` is read using: `scanf("%llu",&n)`; **For 64 bit data:** `typedef unsigned long long int64; int64 test_data=6400000000LL`

Now, RSA cryptography need 256 bits of number or more. This is necessary, human always want more security right? However, some problem setters in programming contests also follow this trend. Simple problem such as finding n'th factorial will become very very hard once the values exceed 64-bit integer data type. Yes, long double can store very high numbers, but it actually only stores first few digits and an exponent, long double is not precise.

Implement your own arbitrary precision arithmetic algorithm. Standard pen and pencil algorithm taught in elementary school will be your tool to implement basic arithmetic operations: addition, subtraction, multiplication and division. More sophisticated

algorithm are available to enhance the speed of multiplication and division. Things are getting very complicated now.

TEST YOUR BIG INTEGER KNOWLEDGE

Solve UVa problems related with Big Integer:

485 - Pascal Triangle of Death
495 - Fibonacci Freeze - plus Fibonacci
10007 - Count the Trees - plus Catalan formula
10183 - How Many Fibs - plus Fibonacci
10219 - Find the Ways ! - plus Catalan formula
10220 - I Love Big Numbers ! - plus Catalan formula
10303 - How Many Trees? - plus Catalan formula
10334 - Ray Through Glasses - plus Fibonacci
10519 - !!Really Strange!!
10579 - Fibonacci Numbers - plus Fibonacci

Carmichael Number

Carmichael number is a number which is not prime but has ≥ 3 prime factors. You can compute them using prime number generator and prime factoring algorithm.

The first 15 Carmichael numbers are:

561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, 46657, 52633, 62745, 63973

TEST YOUR CARMICHAEL NUMBER KNOWLEDGE

Solve UVa problems related with Carmichael Number:

10006 - Carmichael Number

Catalan Formula

The number of distinct binary trees can be generalized as Catalan formula, denoted as $C(n)$.

$$C(n) = \frac{2n}{n+1} C_{n-1}$$

If you are asked values of several $C(n)$, it may be a good choice to compute the values bottom up.

Since $C(n+1)$ can be expressed in $C(n)$, as follows:

$$\begin{aligned}
 \text{Catalan}(n) &= \\
 &\frac{2n!}{n! * n! * (n+1)} \\
 \text{Catalan}(n+1) &= \\
 &\frac{2 * (n+1)}{(n+1)! * (n+1)! * ((n+1)+1)} = \\
 &\frac{(2n+2) * (2n+1) * 2n!}{(n+1) * n! * (n+1) * n! * (n+2)} = \\
 &\frac{(2n+2) * (2n+1) * 2n!}{(n+1) * (n+2) * n! * n! * (n+1)} = \\
 &\frac{(2n+2) * (2n+1)}{(n+1) * (n+2)} * \text{Catalan}(n)
 \end{aligned}$$

TEST YOUR CATALAN FORMULA KNOWLEDGE

Solve UVa problems related with Catalan Formula:

[10007 - Count the Trees](#) - * n! -> number of trees + its permutations

[10303 - How Many Trees?](#)

Counting Combinations - $C(N,K)$

$C(N,K)$ means how many ways that N things can be taken K at a time. This can be a great challenge when N and/or K become very large.

Combination of (N,K) is defined as:

$$\frac{N!}{(N-K)! * K!}$$

For your information, the exact value of $100!$ is:

```
93,326,215,443,944,152,681,699,238,856,266,700,490,715,968,264,381,621,46
8,592,963,895,217,599,993,229,915,608,941,463,976,156,518,286,253,697,920
,827,223,758,251,185,210,916,864,000,000,000,000,000,000,000,000
```

So, how to compute the values of $C(N,K)$ when N and/or K is big but the result is guaranteed to fit in 32-bit integer?

Divide by GCD before multiply (sample code in C)

```
long gcd(long a,long b) {
    if (a%b==0) return b; else return gcd(b,a%b);
}
void Divbygcd(long& a,long& b) {
    long g=gcd(a,b);
    a/=g;
    b/=g;
}
long C(int n,int k){
    long numerator=1,denominator=1,toMul,toDiv,i;
    if (k>n/2) k=n-k; /* use smaller k */
    for (i=k;i;i--) {
        toMul=n-k+i;
        toDiv=i;
        Divbygcd(toMul,toDiv); /* always divide before multiply */
        Divbygcd(numerator,toDiv);
    }
    Divbygcd(toMul,denominator);
    numerator*=toMul;
    denominator*=toDiv;
}
return numerator/denominator;
}
```

TEST YOUR $C(N,K)$ KNOWLEDGE

Solve UVa problems related with Combinations:

[369 - Combinations](#)

[530 - Binomial Showdown](#)

Divisors and Divisibility

If d is a divisor of n , then so is n/d , but d & n/d cannot both be greater than \sqrt{n} .

```
2 is a divisor of 6, so 6/2 = 3 is also a divisor of 6
```

Let $N = 25$, Therefore no divisor will be greater than $\sqrt{25} = 5$.
(Divisors of 25 is 1 & 5 only)

If you keep that rule in your mind, you can design an algorithm to find a divisor better, that's it, no divisor of a number can be greater than the square root of that particular number.

If a number $N = a^i * b^j * \dots * c^k$ then N has $(i+1)*(j+1)*\dots*(k+1)$ divisors.

TEST YOUR DIVISIBILITY KNOWLEDGE

Solve UVa problems related with Divisibility:

[294 - Divisors](#)

Exponentiation

(Assume `pow()` function in `<math.h>` doesn't exist...). Sometimes we want to do exponentiation or take a number to the power n . There are many ways to do that. The standard method is standard multiplication.

Standard multiplication

```
long exponent(long base, long power) {
    long i, result = 1;
    for (i=0; i<power; i++) result *= base;
    return result;
}
```

This is 'slow', especially when power is big - $O(n)$ time. It's better to use divide & conquer.

Divide & Conquer exponentiation

```
long square(long n) { return n*n; }
long fastexp(long base, long power) {
    if (power == 0)
        return 1;
    else if (power%2 == 0)
        return square(fastexp(base, power/2));
    else
        return base * (fastexp(base, power-1));
}
```

Using built in formula, $a^n = \exp(\log(a)*n)$ or $\text{pow}(a,n)$

```
#include <stdio.h>
#include <math.h>

void main() {
    printf("%lf\n", exp(log(8.0)*1/3.0));
    printf("%lf\n", pow(8.0,1/3.0));
}
```

TEST YOUR EXPONENTIATION KNOWLEDGE

Solve UVa problems related with Exponentiation:

[113 - Power of Cryptography](#)

Factorial

Factorial is naturally defined as $\text{Fac}(n) = n * \text{Fac}(n-1)$.

Example:

```
Fac(3) = 3 * Fac(2)
Fac(3) = 3 * 2 * Fac(1)
Fac(3) = 3 * 2 * 1 * Fac(0)
Fac(3) = 3 * 2 * 1 * 1
Fac(3) = 6
```

Iterative version of factorial

```
long FacIter(int n) {
    int i,result = 1;
    for (i=0; i<n; i++) result *= i;
    return result;
}
```

This is the best algorithm to compute factorial. $O(n)$.

Fibonacci

Fibonacci numbers was invented by Leonardo of Pisa (Fibonacci). He defined fibonacci numbers as a growing population of immortal rabbits. Series of Fibonacci numbers are: 1,1,2,3,5,8,13,21,...

Recurrence relation for Fib(n):

```
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2)
```

Iterative version of Fibonacci (using Dynamic Programming)

```
int fib(int n) {
    int a=1,b=1,i,c;
    for (i=3; i<=n; i++) {
        c = a+b;
        a = b;
        b = c;
    }
    return a;
}
```

This algorithm runs in linear time $O(n)$.

Quick method to quickly compute Fibonacci, using Matrix property.

```
Divide_Conquer_Fib(n) {
    i = h = 1;
    j = k = 0;
    while (n > 0) {
        if (n%2 == 1) { // if n is odd
            t = j*h;
            j = i*h + j*k + t;
            i = i*k + t;
        }
        t = h*h;
        h = 2*k*h + t;
        k = k*k + t;
        n = (int) n/2;
    } return j;}

```

This runs in $O(\log n)$ time.

TEST YOUR FIBONACCI KNOWLEDGE

Solve UVa problems related with Fibonacci:

[495 - Fibonacci Freeze](#)

[10183 - How Many Fibs](#)

[10229 - Modular Fibonacci](#)

[10334 - Ray Through Glasses](#)

[10450 - World Cup Noise](#)

[10579 - Fibonacci Numbers](#)

Greatest Common Divisor (GCD)

As its name suggests, Greatest Common Divisor (GCD) algorithm finds the greatest common divisor between two numbers a and b . GCD is a very useful technique, for example to reduce rational numbers into its smallest version ($3/6 = 1/2$). The best GCD algorithm so far is Euclid's Algorithm. Euclid found this interesting mathematical equality: $\text{GCD}(a,b) = \text{GCD}(b, (a \bmod b))$. Here is the fastest implementation of GCD algorithm

```
int GCD(int a,int b) {
    while (b > 0) {
        a = a % b;
        a ^= b;    b ^= a;    a ^= b;    }    return a;
}
```

Lowest Common Multiple (LCM)

Lowest Common Multiple and Greatest Common Divisor (GCD) are two important number properties, and they are interrelated. Even though GCD is more often used than LCM, it is useful to learn LCM too.

```
LCM (m,n) = (m * n) / GCD (m,n)

LCM (3,2) = (3 * 2) / GCD (3,2)
LCM (3,2) = 6 / 1
LCM (3,2) = 6
```

Application of LCM -> to find a synchronization time between two traffic lights, if traffic light A displays green color every 3 minutes and traffic light B displays green color every 2 minutes, then every 6 minutes, both traffic lights will display green color at the same time.

Mathematical Expressions

There are three types of mathematical/algebraic expressions. They are Infix, Prefix, & Postfix expressions.

Infix expression grammar:

```
<infix> = <identifier> | <infix><operator><infix>
<operator> = + | - | * | / | <identifier> = a | b | .... | z
```

Infix expression example: $(1 + 2) = 3$, the operator is in the middle of two operands. Infix expression is the normal way humans compute numbers.

Prefix expression grammar:

<prefix> = <identifier> | <operator><prefix><prefix>
 <operator> = + | - | * | / <identifier> = a | b | | z

Prefix expression example: $(+ 1 2) \Rightarrow (1 + 2) = 3$, the operator is the first item.

One programming language that use this expression is Scheme language.

The benefit of prefix expression is it allows you write: $(1 + 2 + 3 + 4)$

like this $(+ 1 2 3 4)$, this is simpler.

Postfix expression grammar:

<postfix> = <identifier> | <postfix><postfix><operator>
 <operator> = + | - | * | / <identifier> = a | b | | z

Postfix expression example: $(1 2 +) \Rightarrow (1 + 2) = 3$, the operator is the last item.

Postfix Calculator

Computer do postfix calculation better than infix calculation. Therefore when you compile your program, the compiler will convert your infix expressions (most programming language use infix) into postfix, the computer-friendly version.

Why do computer like Postfix better than Infix?

It's because computer use stack data structure, and postfix calculation can be done easily using stack.

Infix to Postfix conversion

To use this very efficient Postfix calculation, we need to convert our Infix expression into Postfix. This is how we do it:

Example:

Infix statement: $((4 - (1 + 2 * (6 / 3) - 5)))$. This is what the algorithm will do, look carefully at the steps.

Expression	Stack (bottom to top)	Postfix expression
$((4-(1+2*(6/3)-5)))$	(
$((4-(1+2*(6/3)-5)))$	((
$((4-(1+2*(6/3)-5)))$	((4
$((4-(1+2*(6/3)-5)))$	((-	4

((4-(1+2*(6/3)-5)))	(((-	4
((4-(1+2*(6/3)-5)))	(((-	41
((4-(1+2*(6/3)-5)))	(((-+	41
((4-(1+2*(6/3)-5)))	(((-+	412
((4-(1+2*(6/3)-5)))	(((-+*	412
((4-(1+2*(6/3)-5)))	(((-+*(412
((4-(1+2*(6/3)-5)))	(((-+*(4126
((4-(1+2*(6/3)-5)))	(((-+*(/	4126
((4-(1+2*(6/3)-5)))	(((-+*(/	41263
((4-(1+2*(6/3)-5)))	(((-+*	41263/
((4-(1+2*(6/3)-5)))	(((-	41263/*+
((4-(1+2*(6/3)-5)))	(((-	41263/*+5
((4-(1+2*(6/3)-5)))	(((-	41263/*+5-
((4-(1+2*(6/3)-5)))	(41263/*+5--
((4-(1+2*(6/3)-5)))		41263/*+5--

TEST YOUR INFIX->POSTFIX KNOWLEDGE

Solve UVa problems related with postfix conversion:

727 - Equation

Prime Factors

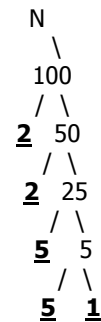
All integers can be expressed as a product of primes and these primes are called **prime factors** of that number.

Exceptions:

For negative integers, multiply by -1 to make it positive again.
For -1,0, and 1, no prime factor. (by definition...)

Standard way

Generate a prime list again, and then check how many of those primes can divide n. This is very slow. Maybe you can write a very efficient code using this algorithm, but there is another algorithm that is much more effective than this.



Creative way, using Number Theory

1. Don't generate prime, or even checking for primality.
2. Always use **stop-at-sqrt** technique
3. Remember to check repetitive prime factors, example= $20 \rightarrow 2 * 2 * 5$, don't count 2 twice. We want distinct primes (however, several problems actually require you to count these multiples)
4. Make your program use constant memory space, no array needed.
5. Use the definition of prime factors wisely, this is the key idea. A number can always be divided into a prime factor and another prime factor or another number. This is called the factorization tree.

From this factorization tree, we can determine these following properties:

1. If we take out a prime factor (F) from any number N, then the N will become smaller and smaller until it become 1, we stop here.
2. This smaller N can be a prime factor or it can be another smaller number, we don't care, all we have to do is to repeat this process until $N=1$.

TEST YOUR PRIME FACTORS KNOWLEDGE

Solve UVa problems related with prime factors:

[583 - Prime Factors](#)

Prime Numbers

Prime numbers are important in Computer Science (For example: Cryptography) and finding prime numbers in a given interval is a "tedious" task. Usually, we are looking for big (very big) prime numbers therefore searching for a better prime numbers algorithms will never stop.

1. Standard prime testing

```
int is_prime(int n) {
    for (int i=2; i<=(int) sqrt(n); i++) if (n%i == 0) return 0;
    return 1;
}void main() {
    int i, count=0;
    for (i=1; i<10000; i++) count += is_prime(i);
    printf("Total of %d primes\n", count);
}
```

2. Pull out the sqrt call

The first optimization was to pull the sqrt call out of the limit test, just in case the compiler wasn't optimizing that correctly, this one is faster:

```
int is_prime(int n) {
    long lim = (int) sqrt(n);
    for (int i=2; i<=lim; i++) if (n%i == 0) return 0; return 1;}

```

3. Restatement of sqrt.

```
int is_prime(int n) {
    for (int i=2; i*i<=n; i++) if (n%i == 0) return 0;
    return 1;
}

```

4. We don't need to check even numbers

```
int is_prime(int n) {
    if (n == 1) return 0;           // 1 is NOT a prime
    if (n == 2) return 1;         // 2 is a prime
    if (n%2 == 0) return 0;       // NO prime is EVEN, except 2
    for (int i=3; i*i<=n; i+=2)    // start from 3, jump 2 numbers
        if (n%i == 0)             // no need to check even numbers
            return 0;
    return 1;
}

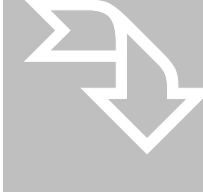
```

5. Other prime properties

A (very) little bit of thought should tell you that no prime can end in 0,2,4,5,6, or 8, leaving only 1,3,7, and 9. It's fast & easy. Memorize this technique. It'll be very helpful for your programming assignments dealing with relatively small prime numbers (16-bit integer 1-32767). This divisibility check (step 1 to 5) will not be suitable for bigger numbers. First prime and the only even prime: 2. Largest prime in 32-bit integer range: $2^{31} - 1 = 2,147,483,647$

6. Divisibility check using smaller primes below sqrt(N):

Actually, we can improve divisibility check for bigger numbers. Further investigation concludes that a number N is a prime if and only if no primes below sqrt(N) can divide N.



How to do this ?

1. Create a large array. How large?
2. Suppose max primes generated will not greater than $2^{31}-1$ (2,147,483,647), maximum 32-bit integer.
3. Since you need smaller primes below \sqrt{N} , you only need to store primes from 1 to $\sqrt{2^{31}}$
4. Quick calculation will show that of $\sqrt{2^{31}} = 46340.95$.
5. After some calculation, you'll find out that there will be at most 4792 primes in the range 1 to 46340.95. So you only need about array of size (roughly) 4800 elements.
6. Generate that prime numbers from 1 to 46340.955. This will take time, but when you already have those 4792 primes in hand, you'll be able to use those values to determine whether a bigger number is a prime or not.
7. Now you have 4792 first primes in hand. All you have to do next is to check whether a big number N a prime or not by dividing it with small primes up to \sqrt{N} . If you can find at least one small primes can divide N , then N is not prime, otherwise N is prime.

Fermat Little Test:

This is a probabilistic algorithm so you cannot guarantee the possibility of getting correct answer. In a range as big as 1-1000000, Fermat Little Test can be fooled by (only) 255 Carmichael numbers (numbers that can fool Fermat Little Test, see Carmichael numbers above). However, you can do multiple random checks to increase this probability.

Fermat Algorithm

If $2^N \text{ modulo } N = 2$ then N has a high probability to be a prime number.

Example:

let $N=3$ (we know that 3 is a prime).

$2^3 \text{ mod } 3 = 8 \text{ mod } 3 = 2$, then N has a high probability to be a prime number... and in fact, it is really prime.

Another example:

let $N=11$ (we know that 11 is a prime).

$2^{11} \text{ mod } 11 = 2048 \text{ mod } 11 = 2$, then N has a high probability to be a prime number... again, this is also really prime.

Sieve of Eratosthenes:

Sieve is the best prime generator algorithm. It will generate a list of primes very quickly, but it will need a very big memory. You can use Boolean flags to do this (Boolean is only 1 byte).

Algorithm for Sieve of Eratosthenes to find the prime numbers within a range L,U (inclusive), where must be $L \leq U$.

```
void sieve(int L,int U) {
    int i,j,d;
    d=U-L+1; /* from range L to U, we have d=U-L+1 numbers. */
    /* use flag[i] to mark whether (L+i) is a prime number or not. */

    bool *flag=new bool[d];
    for (i=0;i<d;i++) flag[i]=true; /* default: mark all to be true */

    for (i=(L%2!=0);i<d;i+=2) flag[i]=false;

    /* sieve by prime factors staring from 3 till sqrt(U) */
    for (i=3;i<=sqrt(U);i+=2) {
        if (i>L && !flag[i-L]) continue;

        /* choose the first number to be sieved -- >=L,
           divisible by i, and not i itself! */
        j=L/i*i;    if (j<L) j+=i;
        if (j==i) j+=i; /* if j is a prime number, have to start form next
one */

        j-=L; /* change j to the index representing j */
        for (;j<d;j+=i) flag[j]=false;
    }

    if (L<=1) flag[1-L]=false;
    if (L<=2) flag[2-L]=true;

    for (i=0;i<d;i++) if (flag[i]) cout << (L+i) << " ";
    cout << endl;
}
```

CHAPTER 8 SORTING

Definition of a sorting problem

Input: A sequence of N numbers (a_1, a_2, \dots, a_N)

Output: A permutation $(a_{1'}, a_{2'}, \dots, a_{N'})$ of the input sequence such that $a_{1'} \leq a_{2'} \leq \dots \leq a_{N'}$.

Things to be considered in Sorting

These are the difficulties in sorting that can also happen in real life:

- A. Size of the list to be ordered is the main concern. Sometimes, the computer memory is not sufficient to store all data. You may only be able to hold part of the data inside the computer at any time, the rest will probably have to stay on disc or tape. This is known as the problem of external sorting. However, rest assured, that almost all programming contests problem size will never be extremely big such that you need to access disc or tape to perform external sorting... (such hardware access is usually forbidden during contests).
- B. Another problem is the stability of the sorting method. Example: suppose you are an airline. You have a list of the passengers for the day's flights. Associated to each passenger is the number of his/her flight. You will probably want to sort the list into alphabetical order. No problem... Then, you want to re-sort the list by flight number so as to get lists of passengers for each flight. Again, "no problem"... - except that it would be very nice if, for each flight list, the names were still in alphabetical order. This is the problem of stable sorting.
- C. To be a bit more mathematical about it, suppose we have a list of items $\{x_i\}$ with x_a equal to x_b as far as the sorting comparison is concerned and with x_a before x_b in the list. The sorting method is stable if x_a is sure to come before x_b in the sorted list.

Finally, we have the problem of key sorting. The individual items to be sorted might be very large objects (e.g. complicated record cards). All sorting methods naturally involve a lot of moving around of the things being sorted. If the things are very large this might take up a lot of computing time -- much more than that taken just to switch two integers in an array.

Comparison-based sorting algorithms

Comparison-based sorting algorithms involves comparison between two object a and b to determine one of the three possible relationship between them: less than, equal, or greater

than. These sorting algorithms are dealing with how to use this comparison effectively, so that we minimize the amount of such comparison. Lets start from the most naive version to the most sophisticated comparison-based sorting algorithms.

Bubble Sort

Speed: $O(n^2)$, extremely slow

Space: The size of initial array

Coding Complexity: Simple

This is the simplest and (unfortunately) the worst sorting algorithm. This sort will do double pass on the array and swap 2 values when necessary.

```
BubbleSort(A)
  for i <- length[A]-1 down to 1
    for j <- 0 to i-1
      if (A[j] > A[j+1]) // change ">" to "<" to do a descending sort
        temp <- A[j]
        A[j] <- A[j+1]
        A[j+1] <- temp
```

Slow motion run of Bubble Sort (**Bold** == sorted region):

```
5 2 3 1 4
2 3 1 4 5
2 1 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 >> done
```

TEST YOUR BUBBLE SORT KNOWLEDGE

Solve UVa problems related with Bubble sort:

[299 - Train Swapping](#)

[612 - DNA Sorting](#)

[10327 - Flip Sort](#)

Quick Sort

Speed: $O(n \log n)$, one of the best sorting algorithm.

Space: The size of initial array

Coding Complexity: Complex, using Divide & Conquer approach

One of the best sorting algorithm known. Quick sort use Divide & Conquer approach and partition each subset. Partitioning a set is to divide a set into a collection of mutually disjoint sets. This sort is much quicker compared to "stupid but simple" bubble sort. Quick sort was invented by C.A.R Hoare.

Quick Sort - basic idea

Partition the array in $O(n)$

Recursively sort left array in $O(\log_2 n)$ best/average case

Recursively sort right array in $O(\log_2 n)$ best/average case

Quick sort pseudo code:

```
QuickSort(A,p,r)
  if p < r
    q <- Partition(A,p,r)
    QuickSort(A,p,q)
    QuickSort(A,q+1,r)
```

Quick Sort for C/C++ User

C/C++ standard library <stdlib.h> contains qsort function.

This is not the best quick sort implementation in the world but it fast enough and VERY EASY to be used... therefore if you are using C/C++ and need to sort something, you can simply call this built in function:

```
qsort(<arrayname>,<size>,sizeof(<elementsize>),compare_function);
```

The only thing that you need to implement is the compare_function, which takes in two arguments of type "const void", which can be cast to appropriate data structure, and then return one of these three values:

- ▲ negative, if a should be before b
- ▲ 0, if a equal to b
- ▲ positive, if a should be after b

1. Comparing a list of integers

simply cast a and b to integers

if $x < y$, $x - y$ is negative, $x == y$, $x - y = 0$, $x > y$, $x - y$ is positive

$x - y$ is a shortcut way to do it :)

reverse $*x - *y$ to $*y - *x$ for sorting in decreasing order

```
int compare_function(const void *a, const void *b) {
    int *x = (int *) a;
    int *y = (int *) b;
    return *x - *y;
}
```

2. Comparing a list of strings

For comparing string, you need strcmp function inside string.h lib. strcmp will by default return -ve, 0, ve appropriately... to sort in reverse order, just reverse the sign returned by strcmp

```
#include <string.h>

int compare_function(const void *a, const void *b) {
    return (strcmp((char *)a, (char *)b));
}
```

3. Comparing floating point numbers

```
int compare_function(const void *a, const void *b) {
    double *x = (double *) a;
    double *y = (double *) b;
    // return *x - *y; // this is WRONG...
    if (*x < *y) return -1;
    else if (*x > *y) return 1; return 0;
}
```

4. Comparing records based on a key

Sometimes you need to sort a more complex stuffs, such as record. Here is the simplest way to do it using qsort library

```
typedef struct {
    int key;
    double value;
} the_record;
```

```
int compare_function(const void *a, const void *b) {
    the_record *x = (the_record *) a;
    the_record *y = (the_record *) b;
    return x->key - y->key;
}
```

Multi field sorting, advanced sorting technique

Sometimes sorting is not based on one key only.

For example sorting birthday list. First you sort by month, then if the month ties, sort by date (obviously), then finally by year.

For example I have an unsorted birthday list like this:

```
24 - 05 - 1982 - Sunny
24 - 05 - 1980 - Cecilia
31 - 12 - 1999 - End of 20th century
01 - 01 - 0001 - Start of modern calendar
```

I will have a sorted list like this:

```
01 - 01 - 0001 - Start of modern calendar
24 - 05 - 1980 - Cecilia
24 - 05 - 1982 - Sunny
31 - 12 - 1999 - End of 20th century
```

To do multi field sorting like this, traditionally one will choose multiple sort using sorting algorithm which has "stable-sort" property.

The better way to do multi field sorting is to modify the `compare_function` in such a way that you break ties accordingly... I'll give you an example using birthday list again.

```
typedef struct {
    int day, month, year;
    char *name;
} birthday;

int compare_function(const void *a, const void *b) {
    birthday *x = (birthday *) a;
    birthday *y = (birthday *) b;

    if (x->month != y->month) // months different
        return x->month - y->month; // sort by month
```

```
else { // months equal..., try the 2nd field... day
    if (x->day != y->day) // days different

        return x->day - y->day; // sort by day
    else // days equal, try the 3rd field... year
        return x->year - y->year; // sort by year
}
```

TEST YOUR MULTI FIELD SORTING KNOWLEDGE

10194 - Football (aka Soccer)

Linear-time Sorting

a. Lower bound of comparison-based sort is $O(n \log n)$

The sorting algorithms that we see above are comparison-based sort, they use comparison function such as $<$, \leq , $=$, $>$, \geq , etc to compare 2 elements. We can model this comparison sort using decision tree model, and we can proof that the shortest height of this tree is $O(n \log n)$.

b. Counting Sort

For Counting Sort, we assume that the numbers are in the range $[0..k]$, where k is at most $O(n)$. We set up a counter array which counts how many duplicates inside the input, and the reorder the output accordingly, without any comparison at all. Complexity is $O(n+k)$.

c. Radix Sort

For Radix Sort, we assume that the input are n d -digits number, where d is reasonably limited.

Radix Sort will then sorts these number digit by digit, starting with the least significant digit to the most significant digit. It usually use a stable sort algorithm to sort the digits, such as Counting Sort above.

Example:

input:

321

257

113
622

sort by third (last) digit:

321
622
113
257

after this phase, the third (last) digit is sorted.

sort by second digit:

113
321
622
257

after this phase, the second and third (last) digit are sorted.

sort by second digit:

113
257
321
622

after this phase, all digits are sorted.

For a set of n d -digits numbers, we will do d pass of counting sort which have complexity $O(n+k)$, therefore, the complexity of Radix Sort is $O(d(n+k))$.

CHAPTER 9 SEARCHING

Searching is very important in computer science. It is very closely related to sorting. We usually search after sorting the data. Remember Binary Search? This is the best example of an algorithm which utilizes both Sorting and Searching.^[2]

Search algorithm depends on the data structure used. If we want to search a graph, then we have a set of well known algorithms such as DFS, BFS, etc

Binary Search

The most common application of binary search is to find a specific value in a sorted list. The search begins by examining the value in the center of the list; because the values are sorted, it then knows whether the value occurs before or after the center value, and searches through the correct half in the same way. Here is simple pseudocode which determines the index of a given value in a sorted list a between indices left and right:

```
function binarySearch(a, value, left, right)
  if right < left
    return not found
  mid := floor((left+right)/2)
  if a[mid] = value
    return mid
  if value < a[mid]
    binarySearch(a, value, left, mid-1)
  else
    binarySearch(a, value, mid+1, right)
```

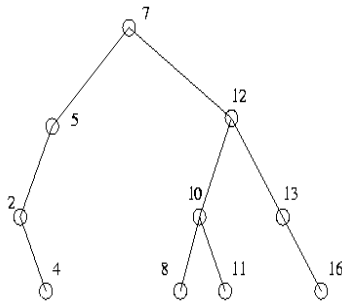
In both cases, the algorithm terminates because on each recursive call or iteration, the range of indexes right minus left always gets smaller, and so must eventually become negative.

Binary search is a logarithmic algorithm and executes in $O(\log n)$ time. Specifically, $1 + \log_2 N$ iterations are needed to return an answer. It is considerably faster than a linear search. It can be implemented using recursion or iteration, as shown above, although in many languages it is more elegantly expressed recursively.

Binary Search Tree

Binary Search Tree (BST) enable you to search a collection of objects (each with a real or integer value) quickly to determine if a given value exists in the collection.

Basically, a binary search tree is a node-weighted, rooted binary ordered tree. That collection of adjectives means that each node in the tree might have no child, one left child, one right child, or both left and right child. In addition, each node has an object associated with it, and the weight of the node is the value of the object.



The binary search tree also has the **property** that each node's left child and descendants of its left child have a value less than that of the node, and each node's right child and its descendants have a value greater or equal to it.

Binary Search Tree

The nodes are generally represented as a structure with four fields, a pointer to the node's left child, a pointer to the node's right child, the weight of the object stored at this node, and a pointer to the object itself. Sometimes, for easier access, people add pointer to the parent too.

Why are Binary Search Tree useful?

Given a collection of n objects, a binary search tree takes only $O(\text{height})$ time to find an objects, assuming that the tree is not really poor (unbalanced), $O(\text{height})$ is $O(\log n)$. In addition, unlike just keeping a sorted array, inserting and deleting objects only takes $O(\log n)$ time as well. You also can get all the keys in a Binary Search Tree in a sorted order by traversing it using $O(n)$ inorder traversal.

Variations on Binary Trees

There are several variants that ensure that the trees are never poor. Splay trees, Red-black trees, B-trees, and AVL trees are some of the more common examples. They are all much more complicated to code, and random trees are generally good, so it's generally not worth it.

Tips: If you're concerned that the tree you created might be bad (it's being created by inserting elements from an input file, for example), then randomly order the elements before insertion.

Dictionary

A dictionary, or hash table, stores data with a very quick way to do lookups. Let's say there is a collection of objects and a data structure must quickly answer the question: 'Is

this object in the data structure?' (e.g., is this word in the dictionary?). A hash table does this in less time than it takes to do binary search.

The idea is this: find a function that maps the elements of the collection to an integer between 1 and x (where x , in this explanation, is larger than the number of elements in your collection). Keep an array indexed from 1 to x , and store each element at the position that the function evaluates the element as. Then, to determine if something is in your collection, just plug it into the function and see whether or not that position is empty. If it is not check the element there to see if it is the same as the something you're holding,

For example, presume the function is defined over 3-character words, and is $(\text{first letter} + (\text{second letter} * 3) + (\text{third letter} * 7)) \bmod 11$ ($A=1, B=2$, etc.), and the words are 'CAT', 'CAR', and 'COB'. When using ASCII, this function takes 'CAT' and maps it to 3, maps 'CAR' to 0, and maps 'COB' to 7, so the hash table would look like this:

```
0: CAR
1
2
3: CAT
4
5
6
7: COB
8
9
10
```

Now, to see if 'BAT' is in there, plug it into the hash function to get 2. This position in the hash table is empty, so it is not in the collection. 'ACT', on the other hand, returns the value 7, so the program must check to see if that entry, 'COB', is the same as 'ACT' (no, so 'ACT' is not in the dictionary either). In the other hand, if the search input is 'CAR', 'CAT', 'COB', the dictionary will return true.

Collision Handling

This glossed over a slight problem that arises. What can be done if two entries map to the same value (e.g., we wanted to add 'ACT' and 'COB')? This is called a collision. There are couple ways to correct collisions, but this document will focus on one method, called chaining.

Instead of having one entry at each position, maintain a linked list of entries with the same hash value. Thus, whenever an element is added, find its position and add it to the

beginning (or tail) of that list. Thus, to have both 'ACT' and 'COB' in the table, it would look something like this:

```
0: CAR
1
2
3: CAT
4
5
6
7: COB -> ACT
8
9
10
```

Now, to check an entry, all elements in the linked list must be examined to find out the element is not in the collection. This, of course, decreases the efficiency of using the hash table, but it's often quite handy.

Hash Table variations

It is often quite useful to store more information than just the value. One example is when searching a small subset of a large subset, and using the hash table to store locations visited, you may want the value for searching a location in the hash table with it.

CHAPTER 10 GREEDY ALGORITHMS

Greedy algorithms are algorithms which follow the problem solving meta-heuristic of making the locally optimum choice at each stage with the hope of finding the global optimum. For instance, applying the greedy strategy to the traveling salesman problem yields the following algorithm: "At each stage visit the nearest unvisited city to the current city".[Wiki Encyclopedia]

Greedy algorithms do not consistently find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. For example, all known greedy algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice. Examples of such greedy algorithms are Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees and the algorithm for finding optimum Huffman trees. The theory of matroids, as well as the even more general theory of greedoids, provide whole classes of such algorithms.

In general, greedy algorithms have five pillars:

- ◆ A candidate set, from which a solution is created
- ◆ A selection function, which chooses the best candidate to be added to the solution
- ◆ A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
- ◆ An objective function, which assigns a value to a solution, or a partial solution, and
- ◆ A solution function, which will indicate when we have discovered a complete solution

Barn Repair

There is a long list of stalls, some of which need to be covered with boards. You can use up to N ($1 \leq N \leq 50$) boards, each of which may cover any number of consecutive stalls. Cover all the necessary stalls, while covering as few total stalls as possible. How will you solve it?

The basic idea behind greedy algorithms is to build large solutions up from smaller ones. Unlike other approaches, however, greedy algorithms keep only the best solution they find as they go along. Thus, for the sample problem, to build the answer for $N = 5$, they find the best solution for $N = 4$, and then alter it to get a solution for $N = 5$. No other solution for $N = 4$ is ever considered.

Greedy algorithms are fast, generally linear to quadratic and require little extra memory. Unfortunately, they usually aren't correct. But when they do work, they are often easy to implement and fast enough to execute.

Problems with Greedy algorithms

There are two basic problems to greedy algorithms.

1. How to Build

How does one create larger solutions from smaller ones? In general, this is a function of the problem. For the sample problem, the most obvious way to go from four boards to five boards is to pick a board and remove a section, thus creating two boards from one. You should choose to remove the largest section from any board which covers only stalls which don't need covering (so as to minimize the total number of stalls covered).

To remove a section of covered stalls, take the board which spans those stalls, and make into two boards: one of which covers the stalls before the section, one of which covers the stalls after the second.

2. Does it work?

The real challenge for the programmer lies in the fact that greedy solutions don't always work. Even if they seem to work for the sample input, random input, and all the cases you can think of, if there's a case where it won't work, at least one (if not more!) of the judges' test cases will be of that form.

For the sample problem, to see that the greedy algorithm described above works, consider the following:

Assume that the answer doesn't contain the large gap which the algorithm removed, but does contain a gap which is smaller. By combining the two boards at the end of the smaller gap and splitting the board across the larger gap, an answer is obtained which uses as many boards as the original solution but which covers fewer stalls. This new answer is better, so therefore the assumption is wrong and we should always choose to remove the largest gap.

If the answer doesn't contain this particular gap but does contain another gap which is just as large, doing the same transformation yields an answer which uses as many boards and covers as many stalls as the other answer. This new answer is just as good as the original solution but no better, so we may choose either.

Thus, there exists an optimal answer which contains the large gap, so at each step, there is always an optimal answer which is a superset of the current state. Thus, the final answer is optimal.

If a greedy solution exists, use it. They are easy to code, easy to debug, run quickly, and use little memory, basically defining a good algorithm in contest terms. The only missing element from that list is correctness. If the greedy algorithm finds the correct answer, go for it, but don't get suckered into thinking the greedy solution will work for all problems.

Sorting a three-valued sequence

You are given a three-valued (1, 2, or 3) sequence of length up to 1000. Find a minimum set of exchanges to put the sequence in sorted order.

The sequence has three parts: the part which will be 1 when in sorted order, 2 when in sorted order, and 3 when in sorted order. The greedy algorithm swaps as many as possible of the 1's in the 2 part with 2's in the 1 part, as many as possible 1's in the 3 part with 3's in the 1 part, and 2's in the 3 part with 3's in the 2 part. Once none of these types remains, the remaining elements out of place need to be rotated one way or the other in sets of 3. You can optimally sort these by swapping all the 1's into place and then all the 2's into place.

Analysis: Obviously, a swap can put at most two elements in place, so all the swaps of the first type are optimal. Also, it is clear that they use different types of elements, so there is no "interference" between those types. This means the order does not matter. Once those swaps have been performed, the best you can do is two swaps for every three elements not in the correct location, which is what the second part will achieve (for example, all the 1's are put in place but no others; then all that remains are 2's in the 3's place and vice-versa, and which can be swapped).

Topological Sort

Given a collection of objects, along with some ordering constraints, such as "A must be before B," find an order of the objects such that all the ordering constraints hold.

Algorithm: Create a directed graph over the objects, where there is an arc from A to B if "A must be before B." Make a pass through the objects in arbitrary order. Each time you find an object with in-degree of 0, greedily place it on the end of the current ordering, delete all of its out-arcs, and recurse on its (former) children, performing the same check. If this algorithm gets through all the objects without putting every object in the ordering, there is no ordering which satisfies the constraints.

TEST YOUR GREEDY KNOWLEDGE

Solve UVa problems related which utilizes Greedy algorithms:

[10020 - Minimal Coverage](#)

[10340 - All in All](#)

[10440 - Ferry Loading \(II\)](#)

CHAPTER 11 DYNAMIC PROGRAMMING

Dynamic Programming (shortened as DP) is a programming technique that can dramatically reduce the runtime of some algorithms (but not all problems have DP characteristics) from exponential to polynomial. Many (and still increasing) real world problems are only solvable within reasonable time using DP.

To be able to use DP, the original problem must have:

1. **Optimal sub-structure** property:
Optimal solution to the problem contains within it optimal solutions to sub-problems
2. **Overlapping sub-problems** property
We accidentally recalculate the same problem twice or more.

There are 2 types of DP: We can either build up solutions of sub-problems from small to large (bottom up) or we can save results of solutions of sub-problems in a table (top down + memoization).

Let's start with a sample of Dynamic Programming (DP) technique. We will examine the simplest form of overlapping sub-problems. Remember Fibonacci? A popular problem which creates a lot of redundancy if you use standard recursion $f_n = f_{n-1} + f_{n-2}$.

Top-down Fibonacci DP solution will record each Fibonacci calculation in a table so it won't have to re-compute the value again when you need it, a simple table-lookup is enough (memoization), whereas Bottom-up DP solution will build the solution from smaller numbers.

Now let's see the comparison between Non-DP solution versus DP solution (both bottom-up and top-down), given in the C source code below, along with the appropriate comments

```
#include <stdio.h>

#define MAX 20 // to test with bigger number, adjust this value

int memo[MAX]; // array to store the previous calculations

// the slowest, unnecessary computation is repeated
int Non_DP(int n) {
    if (n==1 || n==2)
        return 1;
```

```
else
    return Non_DP(n-1) + Non_DP(n-2);
}

// top down DP
int DP_Top_Down(int n) {
    // base case
    if (n == 1 || n == 2)
        return 1;

    // immediately return the previously computed result
    if (memo[n] != 0)
        return memo[n];

    // otherwise, do the same as Non_DP
    memo[n] = DP_Top_Down(n-1) + DP_Top_Down(n-2);
    return memo[n];
}

// fastest DP, bottom up, store the previous results in array
int DP_Bottom_Up(int n) {
    memo[1] = memo[2] = 1; // default values for DP algorithm

    // from 3 to n (we already know that fib(1) and fib(2) = 1
    for (int i=3; i<=n; i++)
        memo[i] = memo[i-1] + memo[i-2];

    return memo[n];
}

void main() {
    int z;

    // this will be the slowest
    for (z=1; z<MAX; z++) printf("%d-", Non_DP(z));
    printf("\n\n");

    // this will be much faster than the first
    for (z=0; z<MAX; z++) memo[z] = 0;
    for (z=1; z<MAX; z++) printf("%d-", DP_Top_Down(z));
    printf("\n\n");

    /* this normally will be the fastest */
    for (z=0; z<MAX; z++) memo[z] = 0;
    for (z=1; z<MAX; z++) printf("%d-", DP_Bottom_Up(z));
    printf("\n\n");
}
```


Matrix Chain Multiplication (MCM)

Let's start by analyzing the cost of multiplying 2 matrices:

```

Matrix-Multiply(A,B):
  if columns[A] != columns[B] then
    error "incompatible dimensions"
  else
    for i = 1 to rows[A] do
      for j = 1 to columns[B] do
        C[i,j]=0
        for k = 1 to columns[A] do
          C[i,j] = C[i,j] + A[i,k] * B[k,j]
        return C
  
```

Time complexity = $O(pqr)$ where $|A|=p \times q$ and $|B|=q \times r$

$|A|=2 \times 3$, $|B|=3 \times 1$, therefore to multiply these 2 matrices, we need $O(2 \times 3 \times 1)=O(6)$
scalar multiplication. The result is matrix C with $|C|=2 \times 1$

TEST YOUR MATRIX MULTIPLICATION KNOWLEDGE

Solve UVa problems related with Matrix Multiplication:

[442 - Matrix Chain Multiplication](#) - Straightforward problem

Matrix Chain Multiplication Problem

Input: Matrices A_1, A_2, \dots, A_n , each A_i of size $P_{i-1} \times P_i$

Output: Fully parenthesized product $A_1 A_2 \dots A_n$ that minimizes the number of scalar multiplications

A product of matrices is fully parenthesized if it is either

1. a single matrix
2. the product of 2 fully parenthesized matrix products surrounded by parentheses

Example of MCM problem:

We have 3 matrices and the size of each matrix:

A_1 (10 x 100), A_2 (100 x 5), A_3 (5 x 50)

We can fully parenthesize them in two ways:

1. $(A_1 (A_2 A_3)) = 100 \times 5 \times 50 + 10 * 100 * 50 = 75000$
2. $((A_1 A_2) A_3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ (10 times better)

See how the cost of multiplying these 3 matrices differ significantly. The cost truly depend on the choice of the fully parenthesization of the matrices. However, exhaustively checking all possible parenthesizations take exponential time.

Now let's see how MCM problem can be solved using DP.

Step 1: characterize the optimal sub-structure of this problem.

Let $A_{i,j}$ ($i < j$) denote the result of multiplying $A_i A_{i+1} \dots A_j$. $A_{i,j}$ can be obtained by splitting it into $A_{i,k}$ and $A_{k+1,j}$ and then multiplying the sub-products. There are $j-i$ possible splits (i.e. $k=i, \dots, j-1$)

Within the optimal parenthesization of $A_{i,j}$:

- (a) the parenthesization of $A_{i,k}$ must be optimal
- (b) the parenthesization of $A_{k+1,j}$ must be optimal

Because if they are not optimal, then there exist other split which is better, and we should choose that split and not this split.

Step 2: Recursive formulation

Need to find $A_{1,n}$

Let $m[i,j]$ = minimum number of scalar multiplications needed to compute $A_{i,j}$

Since $A_{i,j}$ can be obtained by breaking it into $A_{i,k} A_{k+1,j}$, we have

$$m[i,j] = 0, \text{ if } i=j$$

$$= \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \}, \text{ if } i < j$$

let $s[i,j]$ be the value k where the optimal split occurs.

Step 3 Computing the Optimal Costs

Matrix-Chain-Order (p)
 $n = \text{length}[p] - 1$

```

for i = 1 to n do
  m[i,i] = 0
for l = 2 to n do
  for i = 1 to n-l+1 do
    j = i+l-1
    m[i,j] = infinity
    for k = i to j-1 do
      q = m[i,k] + m[k+1,j] + pi-1*pk*pj
      if q < m[i,j] then
        m[i,j] = q
        s[i,j] = k
return m and s

```

Step 4: Constructing an Optimal Solution

```

Print-MCM(s,i,j)
  if i=j then
    print Ai
  else
    print "(" + Print-MCM(s,l,s[i,j]) + "*" + Print-MCM(s,s[i,j]+1,j) +
    ")"

```



Note: As any other DP solution, MCM also can be solved using Top Down recursive algorithm using memoization. Sometimes, if you cannot visualize the Bottom Up, approach, just modify your original Top Down recursive solution by including memoization. You'll save a lot of time by avoiding repetitive calculation of sub-problems.

TEST YOUR MATRIX CHAIN MULTIPLICATION KNOWLEDGE

Solve UVa problems related with Matrix Chain Multiplication:

348 - Optimal Array Multiplication Sequence - Use algorithm above

Longest Common Subsequence (LCS)

Input: Two sequence

Output: A longest common subsequence of those two sequences, see details below.

A sequence Z is a **subsequence** of $X \langle x_1, x_2, \dots, x_m \rangle$, if there exists a strictly increasing sequence $\langle i_1, i_2, \dots, i_k \rangle$ of indices of X such that for all $j=1, 2, \dots, k$, we have $x_{i_j} = z_j$. example: $X = \langle B, C, A, D \rangle$ and $Z = \langle C, A \rangle$.

A sequence Z is called **common subsequence** of sequence X and Y if Z is subsequence of both X and Y.

longest common subsequence (LCS) is just the longest "common subsequence" of two sequences.

A brute force approach of finding LCS such as enumerating all subsequences and finding the longest common one takes too much time. However, Computer Scientist has found a Dynamic Programming solution for LCS problem, we will only write the final code here, written in C, ready to use. Note that this code is slightly modified and we use global variables (yes this is not Object Oriented).

```
#include <stdio.h>
#include <string.h>
#define MAX 100

char X[MAX], Y[MAX];
int i, j, m, n, c[MAX][MAX], b[MAX][MAX];

int LCSlength() {

    m=strlen(X);
    n=strlen(Y);

    for (i=1; i<=m; i++) c[i][0]=0;
    for (j=0; j<=n; j++) c[0][j]=0;

    for (i=1; i<=m; i++)
        for (j=1; j<=n; j++) {
            if (X[i-1]==Y[j-1]) {
                c[i][j]=c[i-1][j-1]+1;
                b[i][j]=1; /* from north west */
            }

            else if (c[i-1][j]>=c[i][j-1]) {
                c[i][j]=c[i-1][j];

                b[i][j]=2; /* from north */
            }

            else {
                c[i][j]=c[i][j-1];
                b[i][j]=3; /* from west */
            }
        }
    return c[m][n];
}
```

```
void printLCS(int i,int j) {
    if (i==0 || j==0) return;

    if (b[i][j]==1) {
        printLCS(i-1,j-1);
        printf("%c",X[i-1]);}
    else if (b[i][j]==2)
        printLCS(i-1,j);
    else
        printLCS(i,j-1);
}

void main() {
    while (1) {
        gets(X);
        if (feof(stdin)) break; /* press ctrl+z to terminate */
        gets(Y);
        printf("LCS length -> %d\n",LCSlength()); /* count length */
        printLCS(m,n); /* reconstruct LCS */
        printf("\n");
    }
}
```

TEST YOUR LONGEST COMMON SUBSEQUENCE KNOWLEDGE

Solve UVA problems related with LCS:

[531 - Compromise](#)

[10066 - The Twin Towers](#)

[10100 - Longest Match](#)

[10192 - Vacation](#)

[10405 - Longest Common Subsequence](#)

Edit Distance

Input: Given two string, Cost for deletion, insertion, and replace

Output: Give the minimum actions needed to transform first string into the second one.

Edit Distance problem is a bit similar to LCS. DP Solution for this problem is very useful in Computational Biology such as for comparing DNA.

Let $d(\text{string1}, \text{string2})$ be the distance between these 2 strings.

A two-dimensional matrix, $m[0..|s1|,0..|s2|]$ is used to hold the edit distance values, such that $m[i,j] = d(s1[1..i], s2[1..j])$.

```

m[0][0] = 0;
for (i=1; i<length(s1); i++) m[i][0] = i;
for (j=1; j<length(s2); j++) m[0][j] = j;

for (i=0; i<length(s1); i++)
  for (j=0; j<length(s2); j++) {
    val = (s1[i] == s2[j]) ? 0 : 1;
    m[i][j] = min( m[i-1][j-1] + val,
                  min(m[i-1][j]+1 , m[i][j-1]+1));
  }

```

To output the trace, use another array to store our action along the way. Trace back these values later.

TEST YOUR EDIT DISTANCE KNOWLEDGE

[164 - String Computer](#)
[526 - String Distance and Edit Process](#)

Longest Inc/Dec-reasing Subsequence (LIS/LDS)

Input: Given a sequence

Output: The longest subsequence of the given sequence such that all values in this longest subsequence is strictly increasing/decreasing.

$O(N^2)$ DP solution for LIS problem (this code check for increasing values):

```

for i = 1 to total-1
  for j = i+1 to total
    if height[j] > height[i] then
      if length[i] + 1 > length[j] then
        length[j] = length[i] + 1
        predecessor[j] = i

```

Example of LIS

height sequence: 1,6,2,3,5

length initially: [1,1,1,1,1] - because max length is at least 1 rite...

predecessor initially: [nil,nil,nil,nil,nil] - assume no predecessor so far

```

After first loop of j:
  length: [1,2,2,2,2], because 6,2,3,5 are all > 1
  predecessor: [nil,1,1,1,1]
After second loop of j: (No change)
  length: [1,2,2,2,2], because 2,3,5 are all < 6
  predecessor: [nil,1,1,1,1]
After third loop:
  length: [1,2,2,3,3], because 3,5 are all > 2
  predecessor: [nil,1,1,3,3]

After fourth loop:
  length: [1,2,2,3,4], because 5 > 3
  predecessor: [nil,1,1,3,4]

```

We can reconstruct the solution using recursion and predecessor array.

Is $O(n^2)$ is the best algorithm to solve LIS/LDS ?

Fortunately, the answer is “No”.

There exist an $O(n \log k)$ algorithm to compute LIS (for LDS, this is just a reversed-LIS), where k is the size of the actual LIS.

This algorithm use some invariant, where for each longest subsequence with length l , it will terminate with value $A[l]$. (Notice that by maintaining this invariant, array A will be naturally sorted.) Subsequent insertion (you will only do n insertions, one number at one time) will use binary search to find the appropriate position in this sorted array A

```

0  1  2  3  4  5  6  7  8
a  -7,10, 9, 2, 3, 8, 8, 1

A -i  i, i, i, i, i, i, i, i, i (iteration number, i = infinity)
A -i -7, i, i, i, i, i, i, i, i (1)
A -i -7,10, i, i, i, i, i, i, i (2)
A -i -7, 9, i, i, i, i, i, i, i (3)
A -i -7, 2, i, i, i, i, i, i, i (4)
A -i -7, 2, 3, i, i, i, i, i, i (5)
A -i -7, 2, 3, 8, i, i, i, i, i (6)
A -i -7, 2, 3, 8, i, i, i, i, i (7)
A -i -7, 1, 3, 8, i, i, i, i, i (8)

```

You can see that the length of LIS is 4, which is correct. To reconstruct the LIS, at each step, store the predecessor array as in standard LIS + this time remember the actual values, since array A only store the last element in the subsequence, not the actual values.

TEST YOUR LONGEST INC/DEC-INCREASING SUBSEQUENCE KNOWLEDGE

[111 - History Grading](#)
[231 - Testing the CATCHER](#)
[481 - What Goes Up - need \$O\(n \log k\)\$ LIS](#)
[497 - Strategic Defense Initiative](#)
[10051 - Tower of Cubes](#)
[10131 - Is Bigger Smarter](#)

Zero-One Knapsack

Input: N items, each with various V_i (Value) and W_i (Weight) and max Knapsack size MW .

Output: Maximum value of items that one can carry, if he can either take or not-take a particular item.

Let $C[i][w]$ be the maximum value if the available items are $\{X_1, X_2, \dots, X_i\}$ and the knapsack size is w .

- ▲ if $i == 0$ or $w == 0$ (if no item or knapsack full), we can't take anything $C[i][w] = 0$
- ▲ if $W_i > w$ (this item too heavy for our knapsack), skip this item $C[i][w] = C[i-1][w]$;
- ▲ if $W_i \leq w$, take the maximum of "not-take" or "take" $C[i][w] = \max(C[i-1][w], C[i-1][w-W_i]+V_i)$;
- ▲ The solution can be found in $C[N][W]$;

```

for (i=0; i<=N ; i++) C[i][0] = 0;
for (w=0; w<=MW; w++) C[0][w] = 0;

for (i=1; i<=N; i++)
  for (w=1; w<=MW; w++) {
    if (W[i] > w)
      C[i][w] = C[i-1][w];
    else
      C[i][w] = max(C[i-1][w] , C[i-1][w-W[i]]+V[i]);
  }
output (C[N][MW]);

```

TEST YOUR 0-1 KNAPSACK KNOWLEDGE

Solve UVa problems related with 0-1 Knapsack:

[10130 - SuperSale](#)

Counting Change

Input: A list of denominations and a value N to be changed with these denominations

Output: Number of ways to change N

Suppose you have coins of 1 cent, 5 cents and 10 cents. You are asked to pay 16 cents, therefore you have to give 1 one cent, 1 five cents, and 1 ten cents. Counting Change algorithm can be used to determine how many ways you can use to pay an amount of money.

The number of ways to change amount A using N kinds of coins equals to:

1. The number of ways to change amount A using all but the first kind of coins, +
2. The number of ways to change amount A-D using all N kinds of coins, where D is the denomination of the first kind of coin.

The tree recursive process will gradually reduce the value of A, then using this rule, we can determine how many ways to change coins.

1. If A is exactly 0, we should count that as 1 way to make change.
2. If A is less than 0, we should count that as 0 ways to make change.
3. If N kinds of coins is 0, we should count that as 0 ways to make change.

```
#include <stdio.h>
#define MAXTOTAL 10000
long long nway[MAXTOTAL+1];

int coin[5] = { 50,25,10,5,1 };

void main()
{
    int i,j,n,v,c;
    scanf("%d",&n);
    v = 5;
    nway[0] = 1;
    for (i=0; i<v; i++) {
        c = coin[i];
        for (j=c; j<=n; j++)
            nway[j] += nway[j-c];
    }
    printf("%lld\n",nway[n]);
}
```

TEST YOUR COUNTING CHANGE KNOWLEDGE

[147 - Dollars](#)
[357 - Let Me Count The Ways - Must use Big Integer](#) [674 - Coin Change](#)

Maximum Interval Sum

Input: A sequence of integers

Output: A sum of an interval starting from index i to index j (consecutive), this sum must be maximum among all possible sums.

Numbers : -1 6

Sum : -1 6

^

max sum

Numbers : 4 -5 4 -3 4 4 -4 4 -5

Sum : 4 -1 4 1 5 9 5 9 4

^

^

stop

max sum

Numbers : -2 -3 -4

Sum : -2 -3 -4

^

max sum, but negative... (this is the maximum anyway)

So, just do a linear sweep from left to right, accumulate the sum one element by one element, start new interval whenever you encounter partial sum < 0 (and record current best maximum interval encountered so far)...

At the end, output the value of the maximum intervals.

TEST YOUR MAXIMUM INTERVAL SUM KNOWLEDGE

Solve UVa problems related with Maximum Interval Sum:

[507 - Jill Rides Again](#)

Other Dynamic Programming Algorithms

Problems that can be solved using Floyd Warshall and its variant, which belong to the category all-pairs shortest path algorithm, can be categorized as Dynamic Programming solution. Explanation regarding Floyd Warshall can be found in Graph section.

Other than that, there are a lot of ad hoc problems that can utilize DP, just remember that when the problem that you encountered exploits optimal sub-structure and repeating sub-problems, apply DP techniques, it may be helpful.

TEST YOUR DP KNOWLEDGE

Solve UVa problems related with Ad-Hoc DP:

[108 - Maximum Sum](#)

[836 - Largest Submatrix](#)

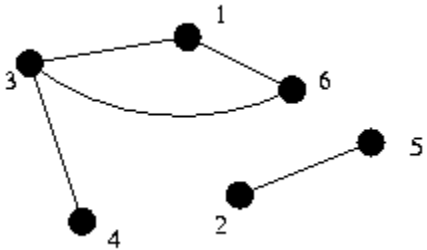
[10003 - Cutting Sticks](#)

[10465 - Homer Simpson](#)

CHAPTER 12 GRAPHS

A *graph* is a collection of *vertices* V and a collection of *edges* E consisting of pairs of vertices. Think of vertices as "locations". The set of vertices is the set of all the possible locations. In this analogy, edges represent paths between pairs of those locations. The set E contains all the paths between the locations.^[2]

Vertices and Edges



The graph is normally represented using that analogy. Vertices are points or circles, edges are lines between them.

In this example graph:

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(1,3), (1,6), (2,5), (3,4), (3,6)\}.$$

Each *vertex* is a member of the set V . A vertex is sometimes called a *node*.

Each *edge* is a member of the set E . Note that some vertices might not be the end point of any edge. Such vertices are termed "isolated".

Sometimes, numerical values are associated with edges, specifying lengths or costs; such graphs are called *edge-weighted* graphs (or *weighted* graphs). The value associated with an edge is called the *weight* of the edge. A similar definition holds for node-weighted graphs.

Telecommunication

Given a set of computers and a set of wires running between pairs of computers, what is the minimum number of machines whose crash causes two given machines to be unable to communicate? (The two given machines will not crash.)

Graph: The vertices of the graph are the computers. The edges are the wires between the computers. Graph problem: minimum dominating sub-graph.

Riding The Fences

Farmer John owns a large number of fences, which he must periodically check for integrity. He keeps track of his fences by maintaining a list of points at which fences intersect. He records the name of the point and the one or two fence names that touch that point. Every fence has two end points, each at some intersection point, although the intersection point may be the end point of only one fence.

Given a fence layout, calculate if there is a way for Farmer John to ride his horse to all of his fences without riding along a fence more than once. Farmer John can start and finish anywhere, but cannot cut across his fields (i.e., the only way he can travel between intersection points is along a fence). If there is a way, find one way.

Graph: Farmer John starts at intersection points and travels between the points along fences. Thus, the vertices of the underlying graph are the intersection points, and the fences represent edges. Graph problem: Traveling Salesman Problem.

Knight moves

Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.

Graph: The graph here is harder to see. Each location on the chessboard represents a vertex. There is an edge between two positions if it is a legal knight move. Graph Problem: Single Source Shortest Path.

Overfencing

Farmer John created a huge maze of fences in a field. He omitted two fence segments on the edges, thus creating two "exits" for the maze. The maze is a "perfect" maze; you can find a way out of the maze from any point inside it.

Given the layout of the maze, calculate the number of steps required to exit the maze from the "worst" point in the maze (the point that is "farther" from either exit when walking optimally to the closest exit).

Here's what one particular $W=5$, $H=3$ maze looks like:

```

+-+--+--+
|      |
+-+ +-+ + +
|  |||
+ +-+ + +
||  |
+-+ +-+--+

```

Graph: The vertices of the graph are positions in the grid. There is an edge between two vertices if they represent adjacent positions that are not separated by a wall.

Terminology

An edge is a *self-loop* if it is of the form (u,u) . The sample graph contains no self-loops.

A graph is *simple* if it neither contains self-loops nor contains an edge that is repeated in E . A graph is called a *multigraph* if it contains a given edge more than once or contains self-loops. For our discussions, graphs are assumed to be simple. The example graph is a simple graph.

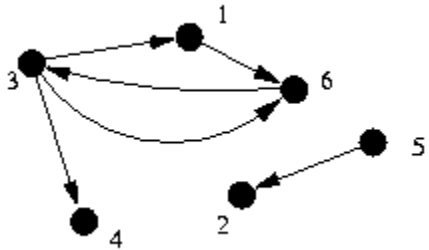
An edge (u,v) is *incident* to both vertex u and vertex v . For example, the edge $(1,3)$ is incident to vertex 3.

The *degree* of a vertex is the number of edges which are incident to it. For example, vertex 3 has degree 3, while vertex 4 has degree 1.

Vertex u is *adjacent* to vertex v if there is some edge to which both are incident (that is, there is an edge between them). For example, vertex 2 is adjacent to vertex 5.

A graph is said to be *sparse* if the total number of edges is small compared to the total number possible $((N \times (N-1))/2)$ and *dense* otherwise. For a given graph, whether it is dense or sparse is not well-defined.

Directed Graph



Graphs described thus far are called *undirected*, as the edges go 'both ways'. So far, the graphs have connoted that if one can travel from vertex 1 to vertex 3, one can also travel from vertex 1 to vertex 3. In other words, $(1,3)$ being in the edge set implies $(3,1)$ is in the edge set.

Sometimes, however, a graph is *directed*, in which case the edges have a direction. In this

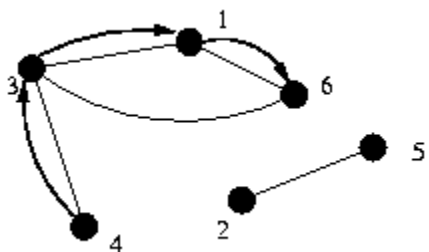
case, the edges are called *arcs*.

Directed graphs are drawn with arrows to show direction.

The *out-degree* of a vertex is the number of arcs which begin at that vertex. The *in-degree* of a vertex is the number of arcs which end at that vertex. For example, vertex 6 has in-degree 2 and out-degree 1.

A graph is assumed to be undirected unless specifically called a directed graph.

Paths



A *path* from vertex u to vertex x is a sequence of vertices (v_0, v_1, \dots, v_k) such that $v_0 = u$ and $v_k = x$ and (v_0, v_1) is an edge in the graph, as is (v_1, v_2) , (v_2, v_3) , etc. The length of such a path is k .

For example, in the undirected graph above, $(4, 3, 1, 6)$ is a path.

This path is said to *contain* the vertices v_0, v_1 , etc., as well as the edges (v_0, v_1) , (v_1, v_2) , etc.

Vertex x is said to be *reachable* from vertex u if a path exists from u to x .

A path is *simple* if it contains no vertex more than once.

A path is a *cycle* if it is a path from some vertex to that same vertex. A cycle is *simple* if it contains no vertex more than once, except the start (and end) vertex, which only appears as the first and last vertex in the path.

These definitions extend similarly to directed graphs (e.g., (v_0, v_1) , (v_1, v_2) , etc. must be arcs).

Graph Representation

The choice of representation of a graph is important, as different representations have very different time and space costs.

The vertices are generally tracked by numbering them, so that one can index them just by their number. Thus, the representations focus on how to store the edges.

Edge List

The most obvious way to keep track of the edges is to keep a list of the pairs of vertices representing the edges in the graph.

This representation is easy to code, fairly easy to debug, and fairly space efficient. However, determining the edges incident to a given vertex is expensive, as is determining if two vertices are adjacent. Adding an edge is quick, but deleting one is difficult if its location in the list is not known.

For weighted graphs, this representation also keeps one more number for each edge, the edge weight. Extending this data structure to handle directed graphs is straightforward. Representing multigraphs is also trivial.

Example

	V1	V2
e1	4	3
e2	1	3
e3	2	5
e4	6	1
e5	3	6

Adjacency Matrix

A second way to represent a graph utilized an *adjacency matrix*. This is a N by N array (N is the number of vertices). The i,j entry contains a 1 if the edge (i,j) is in the graph; otherwise it contains a 0. For an undirected graph, this matrix is symmetric.

This representation is easy to code. It's much less space efficient, especially for large, sparse graphs. Debugging is harder, as the matrix is large. Finding all the edges incident to a given vertex is fairly expensive (linear in the number of vertices), but checking if two vertices are adjacent is very quick. Adding and removing edges are also very inexpensive operations.

For weighted graphs, the value of the (i,j) entry is used to store the weight of the edge. For an unweighted multigraph, the (i,j) entry can maintain the number of edges between the vertices. For a weighted multigraph, it's harder to extend this.

Example

The sample undirected graph would be represented by the following adjacency matrix:

	V1	V2	V3	V4	V5	V6
V1	0	0	1	0	0	1
V2	0	0	0	0	1	0
V3	1	0	0	1	0	1
V4	0	0	1	0	0	0
V5	0	1	0	0	0	0
V6	1	0	1	0	0	0

It is sometimes helpful to use the fact that the (i,j) entry of the adjacency matrix raised to the k -th power gives the number of paths from vertex i to vertex j consisting of exactly k edges.

Adjacency List

The third representation of a matrix is to keep track of all the edges incident to a given vertex. This can be done by using an array of length N , where N is the number of vertices. The i -th entry in this array is a list of the edges incident to i -th vertex (edges are represented by the index of the other vertex incident to that edge).

This representation is much more difficult to code, especially if the number of edges incident to each vertex is not bounded, so the lists must be linked lists (or dynamically allocated). Debugging this is difficult, as following linked lists is more difficult. However, this representation uses about as much memory as the edge list. Finding the vertices adjacent to each node is very cheap in this structure, but checking if two vertices are adjacent requires checking all the edges adjacent to one of the vertices. Adding an edge is easy, but deleting an edge is difficult, if the locations of the edge in the appropriate lists are not known.

Extend this representation to handle weighted graphs by maintaining both the weight and the other incident vertex for each edge instead of just the other incident vertex. Multigraphs are already representable. Directed graphs are also easily handled by this representation, in one of several ways: store only the edges in one direction, keep a separate list of incoming and outgoing arcs, or denote the direction of each arc in the list.

Example

The adjacency list representation of the example undirected graph is as follows:

Vertex	Adjacent Vertices
1	3, 6
2	5
3	6, 4, 1
4	3
5	2
6	3, 1

Implicit Representation

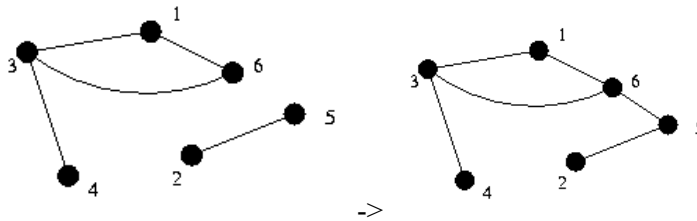
For some graphs, the graph itself does not have to be stored at all. For example, for the Knight moves and Overfencing problems, it is easy to calculate the neighbors of a vertex, check adjacency, and determine all the edges without actually storing that information, thus, there is no reason to actually store that information; the graph is implicit in the data itself.

If it is possible to store the graph in this format, it is generally the correct thing to do, as it saves a lot on storage and reduces the complexity of your code, making it easy to both write and debug.

If N is the number of vertices, M the number of edges, and d_{\max} the maximum degree of a node, the following table summarizes the differences between the representations:

Efficiency	Edge List	Adj Matrix	Adj List
Space	$2 \cdot M$	N^2	$2 \cdot M$
Adjacency Check	M	1	d_{\max}
List of Adjacent Vertices	M	N	d_{\max}
Add Edge	1	1	1
Delete Edge	M	2	$2 \cdot d_{\max}$

Connectedness



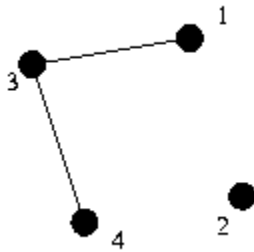
An undirected graph is said to be *connected* if there is a path from every vertex to every other vertex. The example graph is not connected, as there is no path from vertex 2 to vertex 4. However, if you add an edge between vertex 5 and vertex 6, then the graph becomes connected.

A *component* of a graph is a maximal subset of the vertices such that every vertex is reachable from each other vertex in the component. The original example graph has two components: $\{1, 3, 4, 6\}$ and $\{2, 5\}$. Note that $\{1, 3, 4\}$ is not a component, as it is not maximal.

A directed graph is said to be *strongly connected* if there is a path from every vertex to every other vertex.

A *strongly connected component* of a directed graph is a vertex u and the collection of all vertices v such that there is a path from u to v and a path from v to u .

Subgraphs

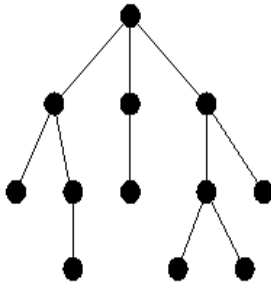


Graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if V' is a subset of V and E' is a subset of E .

The subgraph of G *induced* by V' is the graph (V', E') , where E' consists of all the edges of E that are between members of V' .

For example, for $V' = \{1, 3, 4, 2\}$, the subgraph is like the one shown ->

Tree

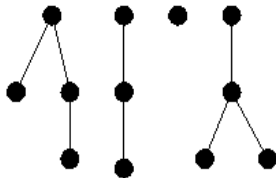


An undirected graph is said to be a *tree* if it contains no cycles and is connected.

A rooted tree

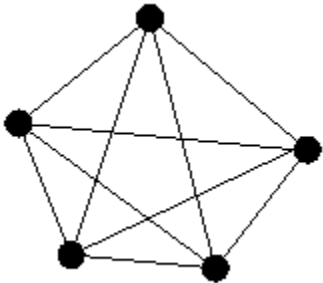
Many trees are what is called *rooted*, where there is a notion of the "top" node, which is called the root. Thus, each node has one *parent*, which is the adjacent node which is closer to the root, and may have any number of *children*, which are the rest of the nodes adjacent to it. The tree above was drawn as a rooted tree.

Forest



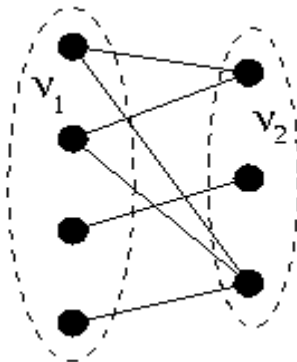
An undirected graph which contains no cycles is called a *forest*. A directed acyclic graph is often referred to as a *dag*.

Complete Graph



A graph is said to be *complete* if there is an edge between every pair of vertices.

Bipartite Graph



A graph is said to be *bipartite* if the vertices can be split into two sets V_1 and V_2 such there are no edges between two vertices of V_1 or two vertices of V_2 .

Uninformed Search

Searching is a process of considering possible sequences of actions, first you have to formulate a goal and then use the goal to formulate a problem.

A **problem** consists of four parts: the **initial state**, a set of **operators**, a **goal test** function, and a **path cost** function. The environment of the problem is represented by a

state space. A **path** through the state space from the initial state to a goal state is a **solution**.

In real life most problems are ill-defined, but with some analysis, many problems can fit into the state space model. A single general search algorithm can be used to solve any problem; specific variants of the algorithm embody different strategies. Search algorithms are judged on the basis of **completeness**, **optimality**, **time complexity**, and **space complexity**. Complexity depends on b , the **branching factor** in the state space, and d , the **depth** of the shallowest solution.

This 6 search type below (there are more, but we only show 6 here) classified as uninformed search, this means that the search have no information about the number of steps or the path cost from the current state to the goal - all they can do is distinguish a goal state from a non-goal state. Uninformed search is also sometimes called blind search.

Breadth First Search (BFS)

Breadth-first search expands the shallowest node in the search tree first. It is complete, optimal for unit-cost operators, and has time and space complexity of $O(b^d)$. The space complexity makes it impractical in most cases.

Using BFS strategy, the root node is expanded first, then all the nodes generated by the root node are expanded next, and their successors, and so on. In general, all the nodes at depth d in the search tree are expanded before the nodes at depth $d+1$.

Algorithmically:

```
BFS(G, s) {
  initialize vertices;
  Q = {s};
  while (Q not empty) {
    u = Dequeue(Q);
    for each v adjacent to u do {
      if (color[v] == WHITE) {
        color[v] = GRAY;
        d[v] = d[u]+1; // compute d[]
        p[v] = u; // build BFS tree
        Enqueue(Q, v);
      }
    }
    color[u] = BLACK;
  }
}
```

BFS runs in $O(V+E)$

Note: BFS can compute $d[v]$ = shortest-path distance from s to v , in terms of minimum number of edges from s to v (un-weighted graph). Its breadth-first tree can be used to represent the shortest-path.

BFS Solution to Popular JAR Problem

```
#include<stdio.h>
#include<conio.h>
#include<values.h>

#define N 105
#define MAX MAXINT

int act[N][N], Q[N*20][3], cost[N][N];
int a, p, b, m, n, fin, na, nb, front, rear;

void init()
{
    front = -1, rear = -1;
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            cost[i][j] = MAX;
    cost[0][0] = 0;
}

void nQ(int r, int c, int p)
{
    Q[++rear][0] = r, Q[rear][1] = c, Q[rear][2] = p;
}

void dQ(int *r, int *c, int *p)
{
    *r = Q[++front][0], *c = Q[front][1], *p = front;
}

void op(int i)
{
    int currCapA, currCapB;
    if(i==0)
        na = 0, nb = b;
    else if(i==1)
        nb = 0, na = a;
    else if(i==2)
        na = m, nb = b;
    else if(i==3)
        nb = n, na = a;
    else if(i==4)
    {
        if(!a && !b)
```

```

        return;
        currCapB = n - b;
        if(currCapB <= 0)
            return;
        if(a >= currCapB)
            nb = n, na = a, na -= currCapB;
        else
            nb = b, nb += a, na = 0;
    }
    else
    {
        if(!a && !b)
            return;
        currCapA = m - a;
        if(currCapA <= 0)
            return;
        if(b >= currCapA)
            na = m, nb = b, nb -= currCapA;
        else
            nb = 0, na = a, na += b;
    }
}

void bfs()
{
    nQ(0, 0, -1);
    do{

        dQ(&a, &b, &p);
        if(a==fin)
            break;
        for(int i=0; i<6; i++)
        {
            op(i); /* na, nb will b changed for this func
                    according to values of a, b
                    */
            if(cost[na][nb]>cost[a][b]+1)
            {
                cost[na][nb]=cost[a][b]+1;
                act[na][nb] = i;
                nQ(na, nb, p);
            }
        }
    } while (rear!=front);
}

void dfs(int p)
{
    int i = act[na][nb];
    if(p==-1)
        return;
    na = Q[p][0], nb = Q[p][1];
    dfs(Q[p][2]);
    if(i==0)

```



```
        printf("Empty A\n");
    else if(i==1)
        printf("Empty B\n");
    else if(i==2)
        printf("Fill A\n");
    else if(i==3)
        printf("Fill B\n");
    else if(i==4)
        printf("Pour A to B\n");
    else
        printf("Pout B to A\n");
}

void main()
{
    clrscr();
    while(scanf("%d%d%d", &m, &n, &fin)!=EOF)
    {
        printf("\n");
        init();
        bfs();
        dfs(Q[p][2]);
        printf("\n");
    }
}
```

Uniform Cost Search (UCS)

Uniform-cost search expands the least-cost leaf node first. It is complete, and unlike breadth-first search is optimal even when operators have differing costs. Its space and time complexity are the same as for BFS.

BFS finds the shallowest goal state, but this may not always be the least-cost solution for a general path cost function. UCS modifies BFS by always expanding the lowest-cost node on the fringe.

Depth First Search (DFS)

Depth-first search expands the deepest node in the search tree first. It is neither complete nor optimal, and has time complexity of $O(b^m)$ and space complexity of $O(bm)$, where m is the maximum depth. In search trees of large or infinite depth, the time complexity makes this impractical.

DFS always expands one of the nodes at the deepest level of the tree. Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower levels.

Algorithmically:

```

DFS(G) {
  for each vertex u in V
    color[u] = WHITE;
  time = 0; // global variable
  for each vertex u in V

    if (color [u] == WHITE)
      DFS_Visit(u);
}

DFS_Visit(u) {
  color[u] = GRAY;
  time = time + 1; // global variable
  d[u] = time; // compute discovery time d[]
  for each v adjacent to u
    if (color[v] == WHITE) {
      p[v] = u; // build DFS-tree
      DFS_Visit(u);
    }
  color[u] = BLACK;
  time = time + 1; // global variable
  f[u] = time; // compute finishing time f[]
}

```

DFS runs in $O(V+E)$

DFS can be used to classify edges of G :

1. Tree edges: edges in the depth-first forest
2. Back edges: edges (u,v) connecting a vertex u to an ancestor v in a depth-first tree
3. Forward edges: non-tree edges (u,v) connecting a vertex u to a descendant v in a depth-first tree
4. Cross edges: all other edges

An undirected graph is acyclic iff a DFS yields no back edges.

DFS algorithm Implementation

Form a one-element queue consisting of the root node.

Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node. If the first element is the goal node, do nothing. If the first element is not the goal node, remove the first element from the queue and add the first element's children, if any, to the **front** of the queue.

If the goal node has been found, announce success, otherwise announce failure.

Note: This implementation differs with BFS in insertion of first element's children, DFS from **FRONT** while BFS from **BACK**. The worst case for DFS is the best case for BFS and vice versa. However, avoid using DFS when the search trees are very large or with infinite maximum depths.

N Queens Problem

Place n queens on an $n \times n$ chess board so that no queen is attacked by another queen.

The most obvious solution to code is to add queens recursively to the board one by one, trying all possible queen placements. It is easy to exploit the fact that there must be exactly one queen in each column: at each step in the recursion, just choose where in the current column to put the queen.

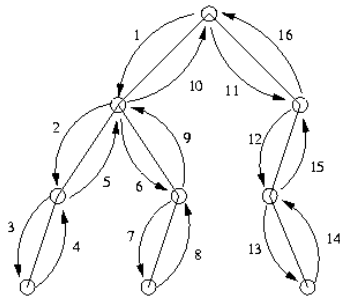
DFS Solution

```
1 search(col)
2   if filled all columns
3     print solution and exit
4   for each row
5     if board(row, col) is not attacked
6       place queen at (row, col)
7       search(col+1)
8       remove queen at (row, col)
```

Calling `search(0)` begins the search. This runs quickly, since there are relatively few choices at each step: once a few queens are on the board, the number of non-attacked squares goes down dramatically.

This is an example of depth first search, because the algorithm iterates down to the bottom of the search tree as quickly as possible: once k queens are placed on the board, boards with even more queens are examined before examining other possible boards with only k queens. This is okay but sometimes it is desirable to find the simplest solutions before trying more complex ones.

Depth first search checks each node in a search tree for some property. The search tree might look like this:



The algorithm searches the tree by going down as far as possible and then backtracking when necessary, making a sort of outline of the tree as the nodes are visited. Pictorially, the tree is traversed in the following manner:

Suppose there are d decisions that must be made. (In this case $d=n$, the number of columns we must fill.) Suppose further that there are C choices for each decision. (In this case $c=n$ also, since any of the rows could potentially be chosen.) Then the entire search will take time proportional to c^d , i.e., an exponential amount of time. This scheme requires little space, though: since it only keeps track of as many decisions as there are to make, it requires only $O(d)$ space.

Knight Cover

Place as few knights as possible on an $n \times n$ chess board so that every square is attacked. A knight is not considered to attack the square on which it sits.

Depth First with Iterative Deepening (DF-ID)

An alternative to breadth first search is iterative deepening. Instead of a single breadth first search, run D depth first searches in succession, each search allowed to go one row deeper than the previous one. That is, the first search is allowed only to explore to row 1, the second to row 2, and so on. This ``simulates" a breadth first search at a cost in time but a savings in space.

```

1 truncated_dfsearch(hnextpos, depth)
2   if board is covered
3     print solution and exit
4   if depth == 0
5     return
6   for i from nextpos to n*n
7     put knight at i
8     search(i+1, depth-1)
9     remove knight at i
10  dfid_search
11  for depth = 0 to max_depth
12    truncated_dfsearch(0, depth)

```

The space complexity of iterative deepening is just the space complexity of depth first search: $O(n)$. The time complexity, on the other hand, is more complex. Each truncated depth first search stopped at depth k takes c^k time. Then if d is the maximum number of decisions, depth first iterative deepening takes $c^0 + c^1 + c^2 + \dots + c^d$ time.

Which to Use?

Once you've identified a problem as a search problem, it's important to choose the right type of search. Here are some things to think about.

Search	Time	Space	When to use
DFS	$O(c^k)$	$O(k)$	Must search tree anyway, know the level the answers are on, or you aren't looking for the shallowest number.
BFS	$O(c^d)$	$O(c^d)$	Know answers are very near top of tree, or want shallowest answer.
DFS+ID	$O(c^d)$	$O(d)$	Want to do BFS, don't have enough space, and can spare the time.

d is the depth of the answer, k is the depth searched, $d \leq k$.

Remember the ordering properties of each search. If the program needs to produce a list sorted shortest solution first (in terms of distance from the root node), use breadth first search or iterative deepening. For other orders, depth first search is the right strategy.

If there isn't enough time to search the entire tree, use the algorithm that is more likely to find the answer. If the answer is expected to be in one of the rows of nodes closest to the root, use breadth first search or iterative deepening. Conversely, if the answer is expected to be in the leaves, use the simpler depth first search.

Be sure to keep space constraints in mind. If memory is insufficient to maintain the queue for breadth first search but time is available, use iterative deepening.

Depth Limited Search

Depth-limited search places a limit on how deep a depth-first search can go. If the limit happens to be equal to the depth of shallowest goal state, then time and space complexity are minimized.

DLS stops to go any further when the depth of search is longer than what we have defined.

Iterative Depending Search

Iterative deepening search calls depth-limited search with increasing limits until a goal is found. It is complete and optimal, and has time complexity of $O(b^d)$

IDS is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then depth 1, then depth 2, and so on. In effect, IDS combines the benefits of DFS and BFS.

Bidirectional Search

Bidirectional search can enormously reduce time complexity, but is not always applicable. Its memory requirements may be impractical.

BDS simultaneously search both forward from the initial state and backward from the goal, and stop when the two searches meet in the middle, however search like this is not always possible.

Superprime Rib

A number is called superprime if it is prime and every number obtained by chopping some number of digits from the right side of the decimal expansion is prime. For example, 233 is a superprime, because 233, 23, and 2 are all prime. Print a list of all the superprime numbers of length n , for $n \leq 9$. The number 1 is not a prime.

For this problem, use depth first search, since all the answers are going to be at the n th level (the bottom level) of the search.

Betsy's Tour

A square township has been partitioned into n^2 square plots. The Farm is located in the upper left plot and the Market is located in the lower left plot. Betsy takes a tour of the township going from Farm to Market by walking through every plot exactly once. Write a program that will count how many unique tours Betsy can take in going from Farm to Market for any value of $n \leq 6$.

Since the number of solutions is required, the entire tree must be searched, even if one solution is found quickly. So it doesn't matter from a time perspective whether DFS or BFS is used. Since DFS takes less space, it is the search of choice for this problem.

Udder Travel

The Udder Travel cow transport company is based at farm A and owns one cow truck which it uses to pick up and deliver cows between seven farms A, B, C, D, E, F, and G. The (commutative) distances between farms are given by an array. Every morning, Udder Travel has to decide, given a set of cow moving orders, the order in which to pick up and deliver cows to minimize the total distance traveled. Here are the rules:

1. The truck always starts from the headquarters at farm A and must return there when the day's deliveries are done.
2. The truck can only carry one cow at time.
3. The orders are given as pairs of letters denoting where a cow is to be picked up followed by where the cow is to be delivered.

Your job is to write a program that, given any set of orders, determines the shortest route that takes care of all the deliveries, while starting and ending at farm A.

Since all possibilities must be tried in order to ensure the best one is found, the entire tree must be searched, which takes the same amount of time whether using DFS or BFS. Since DFS uses much less space and is conceptually easier to implement, use that.

Desert Crossing

A group of desert nomads is working together to try to get one of their group across the desert. Each nomad can carry a certain number of quarts of water, and each nomad drinks a certain amount of water per day, but the nomads can carry differing amounts of water, and require different amounts of water. Given the carrying capacity and drinking requirements of each nomad, find the minimum number of nomads required to get at least one nomad across the desert.

All the nomads must survive, so every nomad that starts out must either turn back at some point, carrying enough water to get back to the start or must reach the other side of the desert. However, if a nomad has surplus water when it is time to turn back, the water can be given to their friends, if their friends can carry it.

This problem actually is two recursive problems: one recursing on the set of nomads to use, the other on when the nomads turn back. Depth-first search with iterative deepening works well here to determine the nomads required, trying first if any one can make it across by themselves, then seeing if two work together to get across, etc.

Addition Chains

An addition chain is a sequence of integers such that the first number is 1, and every subsequent number is the sum of some two (not necessarily unique) numbers that appear in the list before it. For example, 1 2 3 5 is such a chain, as 2 is $1+1$, 3 is $2+1$, and 5 is $2+3$. Find the minimum length chain that ends with a given number.

Depth-first search with iterative deepening works well here, as DFS has a tendency to first try 1 2 3 4 5 ... n, which is really bad and the queue grows too large very quickly for BFS.

Informed Search

Unlike Uninformed Search, Informed Search knows some information that can be used to improve the path selection. Examples of Informed Search: Best First Search, Heuristic Search such as A*.

Best First Search

we define a function $f(n) = g(n)$ where $g(n)$ is the estimated value from node 'n' to goal. This search is "informed" because we do a calculation to estimate $g(n)$

A* Search

$f(n) = h(n) + g(n)$, similar to Best First Search, it uses $g(n)$, but also uses $h(n)$, the total cost incurred so far. The best search to consider if you know how to compute $g(n)$.

Components	
	<p>Given: a undirected graph (see picture on the right)</p> <p>The <i>component</i> of a graph is a maximal-sized (though not necessarily maximum) subgraph which is connected.</p> <p>Calculate the component of the graph.</p> <p>This graph has three components: $\{1,4,8\}$, $\{2,5,6,7,9\}$, & $\{3\}$.</p>

Flood Fill Algorithm

Flood fill can be performed three basic ways: depth-first, breadth-first, and breadth-first scanning. The basic idea is to find some node which has not been assigned to a component and to calculate the component which contains. The question is how to calculate the component.

In the depth-first formulation, the algorithm looks at each step through all of the neighbors of the current node, and, for those that have not been assigned to a component yet, assigns them to this component and recurses on them.

In the breadth-first formulation, instead of recursing on the newly assigned nodes, they are added to a queue.

In the breadth-first scanning formulation, every node has two values: component and visited. When calculating the component, the algorithm goes through all of the nodes that have been assigned to that component but not visited yet, and assigns their neighbors to the current component.

The depth-first formulation is the easiest to code and debug, but can require a stack as big as the original graph. For explicit graphs, this is not so bad, but for implicit graphs, such as the problem presented has, the numbers of nodes can be very large.

The breadth-formulation does a little better, as the queue is much more efficient than the run-time stack is, but can still run into the same problem. Both the depth-first and breadth-first formulations run in $N + M$ time, where N is the number of vertices and M is the number of edges.

The breadth-first scanning formulation, however, requires very little extra space. In fact, being a little tricky, it requires no extra space. However, it is slower, requiring up to $N^2 + M$ time, where N is the number of vertices in the graph.

Breadth-First Scanning

```
# component(i) denotes the component that node i is in

1 function flood_fill(new_component)
2 do
3   num_visited = 0
4   for all nodes i
5     if component(i) = -2
6       num_visited = num_visited + 1
7       component(i) = new_component
8       for all neighbors j of node i
9         if component(j) = nil
10          component(j) = -2
11 until num_visited = 0
12 function find_components
13   num_components = 0
14   for all nodes i
15     component(i) = nil
16   for all nodes i
17     if component(i) is nil
18       num_components = num_components + 1
19       component(i) = -2
20       flood_fill(component(num_components))
```

Running time of this algorithm is $O(N^2)$, where N is the numbers of nodes. Every edge is traversed twice (once for each end-point), and each node is only marked once.

Company Ownership

Given: A weighted directed graph, with weights between 0 and 100.

Some vertex A "owns" another vertex B if:

1. $A = B$
2. There is an arc from A to B with weight more than 50.
3. There exists some set of vertices C_1 through C_k such that A owns C_1 through C_k , and each vertex has an arc of weight x_1 through x_k to vertex B, and $x_1 + x_2 + \dots + x_k > 50$.

Find all (a,b) pairs such that a owns b.

This can be solved via an adaptation of the calculating the vertices reachable from a vertex in a directed graph. To calculate which vertices vertex A owns, keep track of the "ownership percentage" for each node. Initialize them all to zero. Now, at each recursive step, mark the node as owned by vertex A and add the weight of all outgoing arcs to the "ownership percentages." For all percentages that go above 50, recurse into those vertices.

Street Race

Given: a directed graph, and a start point and an end point.

Find all points p that any path from the start point to the end must travel through p.

The easiest algorithm is to remove each point in turn, and check to see if the end point is reachable from the start point. This runs in $O(N(M + N))$ time. Since the original problem stated that $M \leq 100$, and $N \leq 50$, this will run in time easily.

Cow Tours

The diameter of a connected graph is defined as the maximum distance between any two nodes of the graph, where the distance between two nodes is defined as the length of the shortest path.

Given a set of points in the plane, and the connections between those points, find the two points which are currently not in the same component, such that the diameter of the resulting component is minimized.

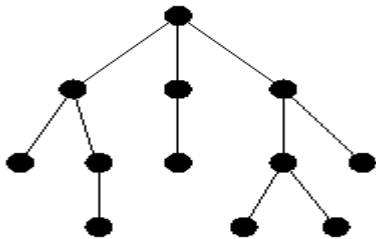
Find the components of the original graph, using the method described above. Then, for each pair of points not in the same component, try placing a connection between them. Find the pair that minimizes the diameter.

Connected Fields

Farmer John contracted out the building of a new barn. Unfortunately, the builder mixed up the plans of Farmer John's barn with another set of plans. Farmer John's plans called for a barn that only had one room, but the building he got might have many rooms. Given a grid of the layout of the barn, tell Farmer John how many rooms it has.

Analysis: The graph here is on the non-wall grid locations, with edge between adjacent non-wall locations, although the graph should be stored as the grid, and not transformed into some other form, as the grid is so compact and easy to work with.

Tree



Tree is one of the most efficient data structure used in a computer program. There are many types of tree. Binary tree is a tree that always have two branches, Red-Black-Trees, 2-3-4 Trees, AVL Trees, etc. A well balanced tree can be used to design a good searching algorithm. A Tree is an undirected graph that contains no cycles and is connected. Many trees are what is called *rooted*, where there is a notion of the "top"

node, which is called the root. Thus, each node has one *parent*, which is the adjacent node which is closer to the root, and may have any number of *children*, which are the rest of the nodes adjacent to it. The tree above was drawn as a rooted tree.

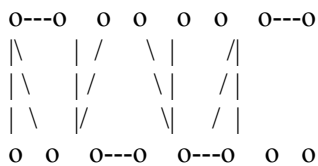
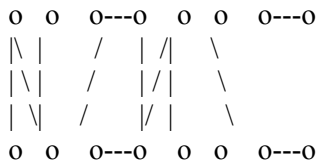
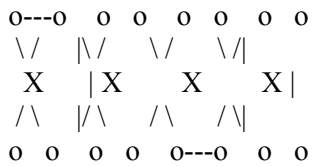
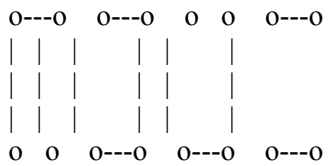
Minimum Spanning Trees

Spanning trees

A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree. A graph may have many spanning trees; for instance the complete graph on four vertices



has sixteen spanning trees:



Minimum spanning trees

Now suppose the edges of the graph have weights or lengths. The weight of a tree is just the sum of weights of its edges. Obviously, different trees have different lengths. The problem: how to find the minimum length spanning tree?

Why minimum spanning trees?

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

Note that if you have a path visiting all points exactly once, it's a special kind of tree. For instance in the example above, twelve of sixteen spanning trees are actually paths. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree. So in general the MST weight is less than the TSP weight, because it's a minimization over a strictly larger set.

On the other hand, if you draw a path tracing around the minimum spanning tree, you trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

How to find minimum spanning tree?

The stupid method is to list all spanning trees, and find minimum of list. We already know how to find minima... But there are far too many trees for this to be efficient. It's also not really an algorithm, because you'd still need to know how to list all the trees.

A better idea is to find some key property of the MST that lets us be sure that some edge is part of it, and use this property to build up the MST one edge at a time.

For simplicity, we assume that there is a unique minimum spanning tree. You can get ideas like this to work without this assumption but it becomes harder to state your theorems or write your algorithms precisely.

Lemma: Let X be any subset of the vertices of G , and let edge e be the smallest edge connecting X to $G-X$. Then e is part of the minimum spanning tree.

Proof: Suppose you have a tree T not containing e ; then we want to show that T is not the MST. Let $e=(u,v)$, with u in X and v not in X . Then because T is a spanning tree it contains a unique path from u to v , which together with e forms a cycle in G . This path has to include another edge f connecting X to $G-X$. $T+e-f$ is another spanning tree (it has the same number of edges, and remains connected since you can replace any path containing f by one going the other way around the cycle). It has smaller weight than T since e has smaller weight than f . So T was not minimum, which is what we wanted to prove.

Kruskal's algorithm

We'll start with Kruskal's algorithm, which is easiest to understand and probably the best one for solving problems by hand.

```
Kruskal's algorithm:
sort the edges of G in increasing order by length
keep a subgraph S of G, initially empty
for each edge e in sorted order
    if the endpoints of e are disconnected in S
        add e to S
return S
```

Note that, whenever you add an edge (u,v) , it's always the smallest connecting the part of S reachable from u with the rest of G , so by the lemma it must be part of the MST.

This algorithm is known as a *greedy algorithm*, because it chooses at each step the cheapest edge to add to S . You should be very careful when trying to use greedy algorithms to solve other problems, since it usually doesn't work. E.g. if you want to find a shortest path from a to b , it might be a bad idea to keep taking the shortest edges. The greedy idea only works in Kruskal's algorithm because of the key property we proved.

Analysis: The line testing whether two endpoints are disconnected looks like it should be slow (linear time per iteration, or $O(mn)$ total). The slowest part turns out to be the sorting step, which takes $O(m \log n)$ time.

Prim's algorithm

Rather than build a subgraph one edge at a time, Prim's algorithm builds a tree one vertex at a time.

```

Prim's algorithm:
let T be a single vertex x
while (T has fewer than n vertices) {
    find the smallest edge connecting T to G-T
    add it to T
}

```

Example of Prim's algorithm in C language:

```

/* usedp=>how many points already used
p->array of structures, consisting x,y,& used/not used
this problem is to get the MST of graph with n vertices
which weight of an edge is the distance between 2 points */

usedp=p[0].used=1; /* select arbitrary point as starting point */
while (usedp<n) {
    small=-1.0;

    for (i=0;i<n;i++) if (p[i].used)
        for (j=0;j<n;j++) if (!p[j].used) {
            length=sqrt(pow(p[i].x-p[j].x,2) + pow(p[i].y-p[j].y,2));

            if (small===-1.0 || length<small) {
                small=length;
                smallp=j;
            }
        }
    minLength+=small;

    p[smallp].used=1;
    usedp++ ;
}

```

Finding Shortest Paths using BFS

This only applicable to graph with unweighted edges, simply do BFS from start node to end node, and stop the search when it encounters the first occurrence of end node.

The **relaxation** process updates the costs of all the vertices, v , connected to a vertex, u , if we could improve the best estimate of the shortest path to v by including (u,v) in the path to v . The relaxation procedure proceeds as follows:


```

initialize_single_source(Graph G, Node s)
  for each vertex v in Vertices(G)
    G.d[v] := infinity
    G.pi[v] := nil
  G.d[s] := 0;

```

This sets up the graph so that each node has no predecessor ($\mathbf{pi[v] = nil}$) and the estimates of the cost (distance) of each node from the source ($\mathbf{d[v]}$) are infinite, except for the source node itself ($\mathbf{d[s] = 0}$).

Note that we have also introduced a further way to store a graph (or part of a graph - as this structure can only store a spanning tree), the **predecessor sub-graph** - the list of predecessors of each node, $\mathbf{pi[j]}$, $1 \leq j \leq |V|$. The edges in the predecessor sub-graph are $\mathbf{(pi[v],v)}$.

The relaxation procedure checks whether the current best estimate of the shortest distance to \mathbf{v} ($\mathbf{d[v]}$) can be improved by going through \mathbf{u} (*i.e.* by making \mathbf{u} the predecessor of \mathbf{v}):

```

relax(Node u, Node v, double w[][] )
  if d[v] > d[u] + w[u,v] then
    d[v] := d[u] + w[u,v]
    pi[v] := u

```

Dijkstra Algorithm

Dijkstra's algorithm (invented by Edsger W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the *source*) to a destination. It turns out that one can find the shortest paths from a given source to *all* points in a graph in the same time, hence this problem is called the **Single-source shortest paths** problem.

There will also be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex. For a graph, $\mathbf{G=(V,E)}$ where \mathbf{V} is a set of vertices and \mathbf{E} is a set of edges.

Dijkstra's algorithm keeps two sets of vertices: \mathbf{S} (the set of vertices whose shortest paths from the source have already been determined) *and* $\mathbf{V-S}$ (the remaining vertices). The other data structures needed are: \mathbf{d} (array of best estimates of shortest path to each vertex) & \mathbf{pi} (an array of predecessors for each vertex)

The basic mode of operation is:

1. Initialise \mathbf{d} and \mathbf{pi} ,
2. Set \mathbf{S} to empty,

3. While there are still vertices in V-S,
4. Sort the vertices in V-S according to the current best estimate of their distance from source,
5. Add u, the closest vertex in V-S, to S,
6. Relax all the vertices still in V-S connected to u

```

DIJKSTRA(Graph G,Node s)
  initialize_single_source(G,s)
  S:={ 0 } /* Make S empty */
  Q:=Vertices(G) /* Put the vertices in a PQ */
  while not Empty(Q)
    u:=ExtractMin(Q);
    AddNode(S,u); /* Add u to S */
    for each vertex v which is Adjacent with u
      relax(u,v,w)

```

Bellman-Ford Algorithm

A more generalized single-source shortest paths algorithm which can find the shortest path in a graph with negative weighted edges. If there is no negative cycle in the graph, this algorithm will update each $d[v]$ with the shortest path from s to v , fill up the predecessor list "pi", and return TRUE. However, if there is a negative cycle in the given graph, this algorithm will return FALSE.

```

BELLMAN_FORD(Graph G,double w[][[]],Node s)
  initialize_single_source(G,s)
  for i=1 to |V[G]|-1
    for each edge (u,v) in E[G]
      relax(u,v,w)

  for each edge (u,v) in E[G]
    if d[v] > d[u] + w(u, v) then
      return FALSE
  return TRUE

```

TEST YOUR BELLMAN FORD KNOWLEDGE

Solve Valladolid Online Judge Problems related with Bellman Ford:

[558 - Wormholes](#), simply check the negative cycle existence.

Single-source shortest paths in Directed Acyclic Graph (DAG)

There exist a more efficient algorithm for solving Single-source shortest path problem for a Directed Acyclic Graph (DAG). So if you know for sure that your graph is a DAG, you may want to consider this algorithm instead of using Dijkstra.

```
DAG_SHORTEST_PATHS(Graph G, double w[][[]], Node s)
  topologically sort the vertices of G // O(V+E)
  initialize_single_source(G, s)

  for each vertex u taken in topologically sorted order
    for each vertex v which is Adjacent with u
      relax(u, v, w)
```

A sample application of this DAG_SHORTEST_PATHS algorithm (as given in CLR book) is to solve critical path problem, i.e. finding the longest path through a DAG, for example: calculating the fastest time to complete a complex task consisting of smaller tasks when you know the time needed to complete each small task and the precedence order of tasks.

Floyd Warshall

Given a directed graph, the Floyd-Warshall All Pairs Shortest Paths algorithm computes the shortest paths between each pair of nodes in $O(n^3)$. In this page, we list down the Floyd Warshall and its variant plus the source codes.

Given:

w : edge weights

d : distance matrix

p : predecessor matrix

$w[i][j]$ = length of direct edge between i and j

$d[i][j]$ = length of shortest path between i and j

$p[i][j]$ = on a shortest path from i to j , $p[i][j]$ is the last node before j .

Initialization

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
    d[i][j] = w[i][j];
    p[i][j] = i;
  }
for (i=0; i<n; i++) d[i][i] = 0;
```

The Algorithm

```

for (k=0;k<n;k++) /* k -> is the intermediate point */
  for (i=0;i<n;i++) /* start from i */
    for (j=0;j<n;j++) /* reaching j */
      /* if i-->k + k-->j is smaller than the original i-->j */
      if (d[i][k] + d[k][j] < d[i][j]) {

          /* then reduce i-->j distance to the smaller one i->k->j */
          graph[i][j] = graph[i][k]+graph[k][j];
          /* and update the predecessor matrix */
          p[i][j] = p[k][j];
      }

```

In the k -th iteration of the outer loop, we try to improve the currently known shortest paths by considering k as an intermediate node. Therefore, after the k -th iteration we know those shortest paths that only contain intermediate nodes from the set $\{0, 1, 2, \dots, k\}$. After all n iterations we know the real shortest paths.

Constructing a Shortest Path

```

print_path (int i, int j) {
  if (i!=j)   print_path(i,p[i][j]);
  print(j);
}

```

TEST YOUR FLOYD WARSHALL KNOWLEDGE

Solve Valladolid Online Judge Problems related with Floyd Warshall:

[104 - Arbitrage](#) - modify the Floyd Warshall parameter correctly

[423 - MPI Maelstrom](#)

[436 - Arbitrage \(II\)](#) - modify the Floyd Warshall parameter correctly

[567 - Risk](#) - even though you can solve this using brute force

Transitive Hull

Given a directed graph, the Floyd-Warshall algorithm can compute the Transitive Hull in $O(n^3)$. Transitive means, if i can reach k and k can reach j then i can reach j . Transitive Hull means, for all vertices, compute its reachability.

w : adjacency matrix

d : transitive hull

$w[i][j]$ = edge between i and j (0=no edge, 1=edge)
 $d[i][j]$ = 1 if and only if j is reachable from i

Initialization

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    d[i][j] = w[i][j];

for (i=0; i<n; i++)
  d[i][i] = 1;
```

The Algorithm

```
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      /* d[i][j] is true if d[i][j] already true
         or if we can use k as intermediate vertex to reach j from i,
         otherwise, d[i][j] is false */
      d[i][j] = d[i][j] || (d[i][k] && d[k][j]);
```

TEST YOUR TRANSITIVE HULL FLOYD WARSHALL KNOWLEDGE

Solve Valladolid Online Judge Problems related with Transitive Hull:
 334 - [Identifying Concurrent Events](#) - internal part of this problem needs transitive hull, even though this problem is more complex than that.

MiniMax Distance

Given a directed graph with edge lengths, the Floyd-Warshall algorithm can compute the minimax distance between each pair of nodes in $O(n^3)$. For example of a minimax problem, refer to the Valladolid OJ problem below.

w : edge weights
d : minimax distance matrix
p : predecessor matrix

$w[i][j]$ = length of direct edge between i and j
 $d[i][j]$ = length of minimax path between i and j

Initialization

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    d[i][j] = w[i][j];

for (i=0; i<n; i++)
  d[i][i] = 0;
```

The Algorithm

```
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      d[i][j] = min(d[i][j], max(d[i][k], d[k][j]));
```

TEST YOUR MINIMAX FLOYD WARSHALL KNOWLEDGE

Solve Valladolid Online Judge Problems related with MiniMax:

[534 - Frogger](#) - select the minimum of longest jumps

[10048 - Audiophobia](#) - select the minimum of maximum decibel along the path

MaxiMin Distance

You can also compute the maximin distance with the Floyd-Warshall algorithm. Maximin is the reverse of minimax. Again, look at Valladolid OJ problem given below to understand maximin.

w : edge weights

d : maximin distance matrix

p : predecessor matrix

$w[i][j]$ = length of direct edge between i and j

$d[i][j]$ = length of maximin path between i and j

Initialization

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    d[i][j] = w[i][j];

for (i=0; i<n; i++)
  d[i][i] = 0;
```

The Algorithm

```
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      d[i][j] = max(d[i][j], min(d[i][k], d[k][j]));
```

TEST YOUR MAXIMIN FLOYD WARSHALL KNOWLEDGE

Solve Valladolid Online Judge Problems related with MaxiMin:

[544 - Heavy Cargo](#) - select the maximum of minimal weight allowed along the path.
[10099 - The Tourist Guide](#) - select the maximum of minimum passenger along the path, then divide total passenger with this value to determine how many trips needed.

Safest Path

Given a directed graph where the edges are labeled with survival probabilities, you can compute the safest path between two nodes (i.e. the path that maximizes the product of probabilities along the path) with Floyd Warshall.

w : edge weights

p : probability matrix

$w[i][j]$ = survival probability of edge between i and j

$p[i][j]$ = survival probability of safest path between i and j

Initialization

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    p[i][j] = w[i][j];

for (i=0; i<n; i++)
  p[i][i] = 1;
```

The Algorithm

```

for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      p[i][j] = max(p[i][j], p[i][k] * p[k][j]);

```

Graph Transpose

Input: directed graph $G = (V, E)$

Output: graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \text{ in } V \times V : (u, v) \text{ in } E\}$.
i.e. G^T is G with all its edges reversed.

Describe efficient algorithms for computing G^T from G , for both the adjacency-list and adjacency-matrix representations of G . Analyze the running times of your algorithms.

Using Adjacency List representation, array B is the new array of Adjacency List G^T

```

for (i=1; i<=p; i++)
  B[i] = nil;

for (i=1; i<=p; i++)
  repeat {
    append i to the end of linked list B[A[i]];
    get next A[i];
  } until A[i] = nil;

```

Eulerian Cycle & Eulerian Path

Euler Cycle

Input: Connected, directed graph $G = (V, E)$

Output: A cycle that traverses every edge of G exactly once, although it may visit a vertex more than once.

Theorem: A directed graph possesses an Eulerian cycle iff

- 1) It is connected
- 2) For all $\{v\}$ in $\{V\}$ $\text{indegree}(v) = \text{outdegree}(v)$

Euler Path

Input: Connected, directed graph $G = (V, E)$

Output: A path from v_1 to v_2 , that traverses every edge of G exactly once, although it may visit a vertex more than once.

Theorem: A directed graph possesses an Eulerian path iff

- 1) It is connected
- 2) For all $\{v\}$ in $\{V\}$ $\text{indegree}(v) = \text{outdegree}(v)$ with the possible exception of two vertices v_1, v_2 in which case,
 - a) $\text{indegree}(v_1) = \text{outdegree}(v_2) + 1$
 - b) $\text{indegree}(v_2) = \text{outdegree}(v_1) - 1$

Topological Sort

Input: A directed acyclic graph (DAG) $G = (V, E)$

Output: A linear ordering of all vertices in V such that if G contains an edge (u, v) , then u appears before v in the ordering.

If drawn on a diagram, Topological Sort can be seen as a vertices along horizontal line, where all directed edges go from left to right. A directed graph G is acyclic if and only if a DFS of G yields no back edge.

Topological-Sort(G)

1. call DFS(G) to compute finishing times $f[v]$ for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

Topological-Sort runs in $O(V+E)$ due to DFS.

Strongly Connected Components

Input: A directed graph $G = (V, E)$

Output: All strongly connected components of G , where in strongly connected component, all pair of vertices u and v in that component, we have $u \rightsquigarrow v$ and $v \rightsquigarrow u$, i.e. u and v are reachable from each other.

Strongly-Connected-Components(G)

1. call DFS(G) to compute finishing times $f[u]$ for each vertex u
2. compute GT , inverting all edges in G using adjacency list
3. call DFS(GT), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ as computed in step 1
4. output the vertices of each tree in the depth-first forest of step 3 as a separate strongly connected component.

Strongly-Connected-Components runs in $O(V+E)$

CHAPTER 13 COMPUTATIONAL GEOMETRY

Computational Geometry is an important subject. Mastering this subject can actually help you in programming contests since every contest usually include 1-2 geometrical problems.^[2]

Geometrical objects and its properties

Earth coordinate system

People use *latitudes* (horizontal lines) and *longitudes* (vertical lines) in Earth coordinate system.

Longitude spans from 0 degrees (Greenwich) to +180* East and -180* West. Latitude spans from 0 degrees (Equator) to +90* (North pole) and -90* (South pole).

The most interesting question is what is the spherical / geographical distance between two cities p and q on earth with radius r, denoted by (p_lat,p_long) to (q_lat,q_long). All coordinates are in radians. (i.e. convert [-180..180] range of longitude and [-90..90] range of latitudes to [-pi..pi] respectively).

After deriving the mathematical equations. The answer is as follow:

```
spherical_distance(p_lat,p_long,q_lat,q_long) =
acos( sin(p_lat) * sin(q_lat) + cos(p_lat) * cos(q_lat) * cos(p_long -
q_long) ) * r
```

since $\cos(a-b) = \cos(a)\cos(b) + \sin(a)\sin(b)$, we can simplify the above formula to:

```
spherical_distance(p_lat,p_long,q_lat,q_long) =
acos( sin(p_lat) * sin(q_lat) +
      cos(p_lat) * cos(q_lat) * cos(p_long) * cos(q_long) +
      cos(p_lat) * cos(q_lat) * sin(p_long) * sin(q_long)
    ) * r
```

TEST YOUR EARTH COORDINATE SYSTEM KNOWLEDGE

Solve UVa problems related with Earth Coordinate System:

[535 - Globetrotter](#)

[10075 - Airlines](#) - combined with all-pairs shortest path

Convex Hull

Basically, Convex Hull is the most basic and most popular computational geometry problem. Many algorithms are available to solve this efficiently, with the best lower bound $O(n \log n)$. This lower bound is already proven.

Convex Hull problem (2-D version):

Input: A set of points in Euclidian plane

Output: Find the minimum set of points that enclosed all other points.

Convex Hull algorithms:

- a. Jarvis March / Gift Wrapping
- b. Graham Scan
- c. Quick Hull
- d. Divide and Conquer

CHAPTER 14 VALLADOLID OJ PROBLEM CATEGORY^[2]

Math	
(General)	113,202,256,275,276,294,326,332,347,350,356,374,377,382,386,412,465,471,474,485,498,550,557,568,594,725,727,846,10006,10014,10019,10042,10060,10071,10093,10104,10106,10107,10110,10125,10127,10162,10190,10193,10195,10469
Prime Numbers	406,516,543,583,686,10140,10200,10490
Geometry	190,191,378,438,476,477,478,10112,10221,10242,10245,10301,10432,10451
Big Numbers	324,424,495,623,713,748,10013,10035,10106,10220,10334
Base Numbers	343,355,389,446,575,10183,10551
Combinations / Permutations	369,530
Theories / Formulas	106,264,486,580
Factorial	160,324,10323,10338
Fibonacci	495,10183,10334,10450
Sequences	138,10408
Modulo	10176,10551
Dynamic Programming	
General	108,116,136,348,495,507,585,640,836,10003,10036,10074,10130,10404
Longest Inc/Decreasing Subsequence	111,231,497,10051,10131
Longest Common Subsequence	531,10066,10100,10192,10405
Counting Change	147,357,674
Edit Distance	164,526
Graph	
Floyd Warshall All-Pairs Shortest Path	1043,436,534,544,567,10048,10099,10171,112,117,122,193,336,352,383,429,469,532,536,590,614,615,657,677,679,762,785,10000,10004,10009,10010,10116,10543
Network Flow	820,10092,10249
Max Bipartite Matching	670,753,10080
Flood Fill	352,572
Articulation Point	315,796
MST	10034,10147,10397
Union Find	459,793,10507
Chess	167,278,439,750
Mixed Problems	
Anagram	153,156,195,454,630
Sorting	120,10152,10194,10258
Encryption	458,554,740,10008,10062
Greedy Algorithm	10020,10249,10340
Card Game	162,462,555
BNF Parser	464,533
Simulation	130,133,144,151,305,327,339,362,379,402,440,556,637,758,10033,10500
Output-related	312,320,330,337,381,391,392,400,403,445,488,706,10082

Ad Hoc	101,102,103, 105,118, 119,121,128, 142,145,146,154, 155,187,195220,227,232, 271,272,291,297,299,300,311,325,333,335,340,344,349,353,380,384,394,401,408,409,413,414,417,4 34,441,442,444,447,455,457,460,468,482,483,484,489,492,494,496,499537,541,542,551,562,573,574 ,576,579,586,587,591602,612,616,617,620,621,642,654,656,661,668,671,673729,755,837,10015,100 17,10018,10019,10025,10038, 10041,10045,10050,10055,10070,10079,10098,10102, 10126, 10161,10182,10189, 10281,10293, 10487
Array Manipulation	466,10324,10360,10443
Binary Search	10282,10295,10474
Backtracking	216,291422,524,529, 539, 571, 572, 574,10067,10276,10285,10301,10344,10400,10422,10452
3n+1 Problem	100,371,694

This problems are available at (<http://acm.uva.es/p>). New problems are added after each online contest at Valladolid Online Judge as well as after each ACM Regional Programming Contest, problems are added to live ACM archive (<http://cii-judge.baylor.edu/>).

APPENDIX A

ACM PROGRAMMING PROBLEMS



This part of this book contains some interesting problems from ACM/ICPC. Problems are collected from Valladolid Online Judge. You can see the reference section for finding more problem sources.

Find the ways !

The Problem

An American, a Frenchman and an Englishwoman had been to Dhaka, the capital of Bangladesh. They went sight-seeing in a taxi. The three tourists were talking about the sites in the city. The American was very proud of tall buildings in New York. He boasted to his friends, "Do you know that the Empire State Building was built in three months?"

"Really?" replied the Frenchman. "The Eiffel Tower in Paris was built in only one month! (However, The truth is, the construction of the Tower began in January 1887. Forty Engineers and designers under Eiffel's direction worked for two years. The tower was completed in March 1889.)

"How interesting!" said the Englishwoman. "Buckingham Palace in London was built in only two weeks!!"

At that moment the taxi passed a big slum (However, in Bangladesh we call it "Bostii"). "What was that? When it was built ?" The Englishwomen asked the driver who was a Bangladeshi.

"I don't know!" , answered the driver. "It wasn't there yesterday!"

However in Bangladesh, illegal establishment of slums is a big time problem. Government is trying to destroy these slums and remove the peoples living there to a far place, formally in a planned village outside the city. But they can't find any ways, how to destroy all these slums!

Now, can you imagine yourself as a slum destroyer? In how many ways you can destroy k slums out of n slums ! Suppose there are 10 slums and you are given the permission of destroying 5 slums, surly you can do it in 252 ways, which is only a 3 digit number, Your task is to find out the digits in ways you can destroy the slums !

The Input

The input file will contain one or more test cases. Each test case consists of one line containing two integers n ($n \geq 1$) and k ($1 \leq k \leq n$).

Sample Input

```
20 5
100 10
200 15
```

Sample Output

```
5
14
23
```

Problem Setter : Ahmed Shamsul Arefin

I Love Big Numbers !

The Problem

A Japanese young girl went to a Science Fair at Tokyo. There she met with a Robot named Mico-12, which had AI (You must know about AI-Artificial Intelligence). The Japanese girl thought, she can do some fun with that Robot. She asked her, "Do you have any idea about maths ?". "Yes! I love mathematics", The Robot replied.

"Okey ! Then I am giving you a number, you have to find out the Factorial of that number. Then find the sum of the digits of your result!. Suppose the number is 5.You first calculate $5!=120$, then find sum of the digits $1+2+0=3$.Can you do it?"

"Yes. I can do!"Robot replied."Suppose the number is 100, what will be the result ?".At this point the Robot started thinking and calculating. After a few minutes the Robot head burned out and it cried out loudly "Time Limit Exceeds".The girl laughed at the Robot and said "The sum is definitely 648". "How can you tell that ?" Robot asked the girl. "Because I am an ACM World Finalist and I can solve the Big Number problems easily." Saying this, the girl closed her laptop computer and went away. Now, your task is to help the Robot with the similar problem.

The Input

The input file will contain one or more test cases. Each test case consists of one line containing an integers n ($n \leq 1000$).

The Output

For each test case, print one line containing the required number. This number will always fit into an integer, i.e. it will be less than $2^{31}-1$.

Sample Input

5
60
100

Sample Output

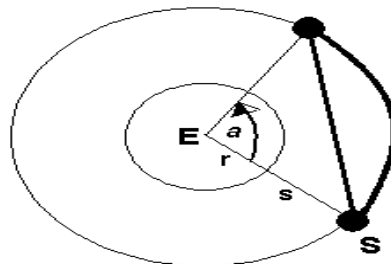
3
288
648

Problem Setter : Ahmed Shamsul Arefin

Satellites

The Problem

The radius of earth is 6440 Kilometer. There are many Satellites and Asteroids moving around the earth. If two Satellites create an angle with the center of earth, can you find out the *distance* between them? By *distance* we mean both the **arc** and **chord** distances. Both satellites are on the same orbit. (However, please consider that they are revolving on a circular path rather than an elliptical path.)



E = Earth S = Satellite

The Input

The input file will contain one or more test cases.

Each test case consists of one line containing two-integer s and a and a string "**min**" or "**deg**". Here s is the distance of the satellite from the surface of the earth and a is the angle that the satellites make with the center of earth. It may be in minutes (') or in degrees (^o). Remember that the same line will never contain minute and degree at a time.

The Output

For each test case, print one line containing the required distances i.e. both *arc distance* and *chord distance* respectively between two satellites in Kilometer. The distance will be a floating-point value with six digits after decimal point.

Sample Input

```
500 30 deg
700 60 min
200 45 deg
```

Sample Output

```
3633.775503 3592.408346
124.616509 124.614927
5215.043805 5082.035982
```

Problem Setter : Ahmed Shamsul Arefin

Decode the Mad man

The Problem

Once in BUET, an old professor had gone completely mad. He started talking with some peculiar words. Nobody could realize his speech and lectures. Finally the BUET authority fall in great trouble. There was no way left to keep that man working in university. Suddenly a student (definitely he was a registered author at UVA ACM Chapter and hold a good rank on 24 hour-Online Judge) created a program that was able to decode that

professor's speech. After his invention, everyone got comfort again and that old teacher started his everyday works as before.

So, if you ever visit BUET and see a teacher talking with a microphone, which is connected to a IBM computer equipped with a voice recognition software and students are taking their lecture from the computer screen, don't get thundered! Because now your job is to write the same program which can decode that mad teacher's speech!

The Input

The input file will contain only one test case i.e. the encoded message.

The test case consists of one or more words.

The Output

For the given test case, print a line containing the decoded words. However, it is not so hard task to replace each letter or punctuation symbol by the two immediately to its left alphabet on your standard keyboard.

Sample Input

```
k[r dyt I[o
```

Sample Output

```
how are you
```

Problem Setter: Ahmed Shamsul Arefin

How many nodes ?

The Problem

One of the most popular topic of Data Structures is Rooted Binary Tree. If you are given some nodes you can definitely able to make the maximum number of trees with them. But if you are given the maximum number of trees built upon a few nodes, Can you find out how many nodes built those trees?

The Input

The input file will contain one or more test cases. Each test case consists of an integer n ($n \leq 4,294,967,295$). Here n is the maximum number of trees.

The Output

For each test case, print one line containing the actual number of nodes.

Sample Input

```
5
14
42
```

Sample Output

```
3
4
5
```

Problem Setter: Ahmed Shamsul Arefin

Power of Cryptography

Background

Current work in cryptography involves (among other things) large prime numbers and computing powers of numbers modulo functions of these primes. Work in this area has resulted in the practical use of results from number theory and other branches of mathematics once considered to be of only theoretical interest.

This problem involves the efficient computation of integer roots of numbers.

The Problem

Given an integer $n \geq 1$ and an integer $p \geq 1$ you are to write a program that determines $\sqrt[n]{p}$, the positive n^{th} root of p . In this problem, given such integers n and p , p will always be of the form k^n for an integer k (this integer is what your program must find).

The Input

The input consists of a sequence of integer pairs n and p with each integer on a line by itself. For all such pairs $1 \leq n \leq 200$, $1 \leq p < 10^{101}$, and there exists an integer k , $1 \leq k \leq 10^9$, such that $k^n = p$.

The Output

For each integer pair n and p the value $\sqrt[n]{p}$ should be printed, i.e., the number k such that $k^n = p$.

Sample Input

```
2
16
3
27
7
4357186184021382204544
```

Sample Output

```
4
3
1234
```

Roman Roulette

The historian Flavius Josephus relates how, in the Romano-Jewish conflict of 67 A.D., the Romans took the town of Jotapata which he was commanding. Escaping, Josephus found himself trapped in a cave with 40 companions. The Romans discovered his whereabouts and invited him to surrender, but his companions refused to allow him to do so. He therefore suggested that they kill each other, one by one, the order to be decided by lot. Tradition has it that the means for effecting the lot was to stand in a circle, and, beginning at some point, count round, every third person being killed in turn. The sole survivor of this process was Josephus, who then surrendered to the Romans. Which begs the question: had Josephus previously practised quietly with 41 stones in a dark corner, or had he calculated mathematically that he should adopt the 31st position in order to survive?

Having read an account of this gruesome event you become obsessed with the fear that you will find yourself in a similar situation at some time in the future. In order to prepare yourself for such an eventuality you decide to write a program to run on your hand-held PC which will determine the position that the counting process should start in order to ensure that you will be the sole survivor.

In particular, your program should be able to handle the following variation of the processes described by Josephus. $n > 0$ people are initially arranged in a circle, facing inwards, and numbered from 1 to n . The numbering from 1 to n proceeds consecutively in a clockwise direction. Your allocated number is 1. Starting with person number i , counting starts in a clockwise direction, until we get to person number k ($k > 0$), who is promptly killed. We then proceed to count a further k people in a clockwise direction, starting with the person immediately to the left of the victim. The person number k so selected has the job of burying the victim, and then returning to the position in the circle that the victim had previously occupied. Counting then proceeds from the person to his immediate left, with the k th person being killed, and so on, until only one person remains.

For example, when $n = 5$, and $k = 2$, and $i = 1$, the order of execution is 2, 5, 3, and 1. The survivor is 4.

Input and Output

Your program must read input lines containing values for n and k (in that order), and for each input line output the number of the person with which the counting should begin in order to ensure that you are the sole survivor. For example, in the above case the safe starting position is 3. Input will be terminated by a line containing values of 0 for n and k .

Your program may assume a maximum of 100 people taking part in this event.

Sample Input

```
1 1
1 5
0 0
```

Sample Output

```
1
1
```

Joseph's Cousin

The Problem

The Joseph's problem is notoriously known. For those who are not familiar with the problem, among n people numbered $1, 2, \dots, n$, standing in circle every m th is going to be executed and only the life of the last remaining person will be saved. Joseph was smart enough to choose the position of the last remaining person, thus saving his life to give the message about the incident.

Although many good programmers have been saved since Joseph spread out this information, Joseph's Cousin introduced a new variant of the malignant game. This insane character is known for its barbarian ideas and wishes to clean up the world from silly programmers. We had to infiltrate some the agents of the ACM in order to know the process in this new mortal game. In order to save yourself from this evil practice, you must develop a tool capable of predicting which person will be saved.

The Destructive Process

The persons are eliminated in a very peculiar order; m is a dynamical variable, which each time takes a different value corresponding to the prime numbers' succession (2,3,5,7...). So in order to kill the i th person, Joseph's cousin counts up to the i th prime.

The Input

It consists of separate lines containing n [$1..3501$], and finishes with a 0.

The Output

The output will consist in separate lines containing the position of the person which life will be saved.

Sample Input

```
6  
0
```

Sample Output

```
4
```

Integer Inquiry

One of the first users of BIT's new supercomputer was Chip Diller. He extended his exploration of powers of 3 to go from 0 to 333 and he explored taking various sums of those numbers.

``This supercomputer is great," remarked Chip. ``I only wish Timothy were here to see these results." (Chip moved to a new apartment, once one became available on the third floor of the Lemon Sky apartments on Third Street.)

Input

The input will consist of at most 100 lines of text, each of which contains a single VeryLongInteger. Each VeryLongInteger will be 100 or fewer characters in length, and will only contain digits (no VeryLongInteger will be negative).

The final input line will contain a single zero on a line by itself.

Output

Your program should output the sum of the VeryLongIntegers given in the input.

Sample Input

```
123456789012345678901234567890
123456789012345678901234567890
123456789012345678901234567890
0
```

Sample Output

```
370370367037037036703703703670
```


The Decoder

Write a complete program that will correctly decode a set of characters into a valid message. Your program should read a given file of a simple coded set of characters and print the exact message that the characters contain. The code key for this simple coding is a one for one character substitution based upon a *single arithmetic manipulation* of the printable portion of the ASCII character set.

Input and Output

For example: with the input file that contains:

```
1JKJ'pz' {ol' {yhklthyr'vm' {ol'Jvu {yvs'Kh {h'Jvywvyh {pvu5
1PIT'pz'h' {yhklthyr'vm' {ol'Pu {lyuh {pvuhs'l|zpu|zz'Thjopul'Jvywvyh {pvu5
1KLJ'pz' {ol' {yhklthyr'vm' {ol'Kpnp {hs'Lx|pw|tlu {'Jvywvyh {pvu5
```

your program should print the message:

- *CDC is the trademark of the Control Data Corporation.
- *IBM is a trademark of the International Business Machine Corporation.
- *DEC is the trademark of the Digital Equipment Corporation.

Your program should accept all sets of characters that use the same encoding scheme and should print the actual message of each set of characters.

Sample Input

```
1JKJ'pz' {ol' {yhklthyr'vm' {ol'Jvu {yvs'Kh {h'Jvywvyh {pvu5
1PIT'pz'h' {yhklthyr'vm' {ol'Pu {lyuh {pvuhs'l|zpu|zz'Thjopul'Jvywvyh {pvu5
1KLJ'pz' {ol' {yhklthyr'vm' {ol'Kpnp {hs'Lx|pw|tlu {'Jvywvyh {pvu5
```

Sample Output

- *CDC is the trademark of the Control Data Corporation.
- *IBM is a trademark of the International Business Machine Corporation.
- *DEC is the trademark of the Digital Equipment Corporation.

APPENDIX B

COMMON CODES/ROUTINES FOR PROGRAMMING



This part of this book contains some COMMON codes/routines/pseudocodes for programmers that one can EASILY use during the contest hours.^[10]

Finding GCD,LCM and Prime Factor

```
#include<iostream>
#include<vector>
#include<fstream>
using namespace std;

int gcd(int a, int b)
{
    int r=1;
    if(a>b)
    {
        while ( r!=0)
        {
            r=a%b;
            a=b;
            b=r;
        }
    }
    return a;
}
else if ( b>a)
{
    while (r!=0)
    {
        r=b%a;
        b=a;
        a=r;
    }
}
return b;
}
else if ( a==b)
    return a;
}

int lcm(int a, int b)
{
    return a*b/gcd(a,b);
}

bool isprime(int a)
{
    {
        if(a==2) return true;
        else
        {
            for(int i=2;i<a/2+1;++i)
                if(a%i==0) return false;
            if(i==a/2) return true;}
        }
}

vector<int> primefactors(int a)
{
    vector<int> primefactors;
    for(int i=1;i<=a;++i)
        if(a%i==0&&isprime(i))
        {
            primefactors.push_back(i);
            a=a/i;i=1;
        }
    return primefactors;
}

int main()
{
    int a=5,b=2500;
    out<<"GCD = "<<gcd(a,b)<<endl;
    out<<"LCM = "<<lcm(a,b)<<endl;
    if(!isprime(b)) out<<"not
    prime"<<endl;
    vector <int> p =
    primefactors(b);
    for(int i=0;i<p.size();++i)
        out<<p[i]<<endl;
    return 0;
}
```

Recursively find the GCD by Euclid's Algorithm

```

//*****      GCD using recursive way      *****/
#include<stdio.h>
unsigned int Euclid_g_c_d(unsigned int a, unsigned int b){
if(b==0)
    return a;
    else
        return Euclid_g_c_d(b,a%b);
}
int main(){
    unsigned int a,b,gcd;
    while( scanf("%d%d",&a,&b)==2){
        gcd=Euclid_g_c_d(a,b);
        printf("%d",gcd);
    }
    return 0;
}

```

The another application of GCD is Extended Euclid's Algorithm..We can solve the formula $d = ax + by \Rightarrow \text{gcd}(a,b) = ax + by$. This algorithm take the inputs a and b..and out put the d(gcd of a,b) and the value of the root x and y.

```

//*****      Extended Euclid's Algorithm      *****/
#include<stdio.h>
unsigned long d,x,y;
void Extended_Euclid(unsigned long a, unsigned long b){
    unsigned long x1;
    if(b>a){
        x1=a;    //if b>a so I used this if condition
        a=b;    // result is ok but x and y swaped
        b=x1;
    }
    if(b==0){
        d=a;
        x=1;
        y=0;
        return;
    }
    Extended_Euclid(b,a%b);
    d = d;
    x1 = x-(a/b) * y;
    x = y;
    y = x1;
}
int main(){
    unsigned long a,b;    // d = gcd(a,b) = ax+by
    while(scanf("%lu %lu", &a, &b)==2){
        Extended_Euclid(a,b);
        printf("%lu %lu %lu\n",d,x,y);}return 0;}

```

GCD of N numbers:

This program takes a number that is for how many numbers you want to find the GCD and show the GCD of them. This takes input until end of file.

```

//*****          GCD for n numbers          *****//
#include<stdio.h>
int main(){
    long gcd,a,b,n,i;
    while (scanf("%ld",&n)==1){
        scanf("%ld%ld",&a,&b);
        i=2;
        while(i<n){
            gcd=g_c_d(a,b);
            a=gcd;

i++;
            scanf("%ld",&b);
        }
        gcd=g_c_d(a,b);
        printf("%ld\n",gcd);
    }
    return 0;
}

```

LCM (Lowest Common Multiple)

It is also a important topic in programming. Suppose different clocks are set in 12 point, and each and every clock has their own time to complete the cycle. When every clock again set in the 12 point.? This type of problem can be solved by LCM. LCM was implemented by the help of GCD.

LCM (m, n) = [m / GCD(m, n)] * n;
 Or, D1=m / GCD(m, n);
 D2=n / GCD(m, n);
 LCM(m, n) = D1 * D2 * GCD(m, n);

Here is the code of some int numbers LCM.

```

long GCD(long a, long b){
    if(b==0)
        return a;
    else
        return GCD(b,a%b);}
long LCM(long a,long b){

```

```

        return (a/GCD(a,b)*b);
    }
int main(){
    long a,b;
    int n,i;
    while(scanf("%d",&n)==1){
        if(n==0) break;
        scanf("%ld",&a);
        for(i=1;i<n;i++){
            scanf("%ld",&b);
            a=LCM(a,b);
        }
        printf("%ld\n",a);
    }
    return 0;}

```

Factorials

$n! = n * (n-1) * (n-2) * (n-3) \dots \text{upto } 1.$
 $5! = 5 * (5-1) * (5-2) * (5-3) * 2 * 1 = 120.$
 This is a recursion process.

```

/** Factorial by recursion *****//
#include<stdio.h>
long fac(long x){
    if(x==0)
        return 1;
    return x * fac(x-1);
}
/*****//
int main(){
    long n,res;
    while(scanf("%ld",&n)==1){
        res=fac(n);
        printf("%ld\n",res);
    }
    return 0;}

```

This is very easy, but one thing we have to know that factorials increases multiply. The 12th factorial is 479001600 which is very big so can you imagine the factorial of 1000!....? The above program can compute upto 12th factorials so for less then 12th we can use it. But for time consideration in ACM we always try to avoids the recursion process.

Iterative Factorial

```

//***** Factorial by normal multiplication *****//
#include<stdio.h>
long fac(long x){
    long i,res=1;
    for(i=1;i<=x;i++){
        res*=i;
    }
    return res;
}
int main(){
    long n,res;
    while(scanf("%ld",&n)==1){
        res=fac(n);
        printf("%ld\n",res);
    }
    return 0;
}

```

So, how can we compute the factorial of big numbers? Here is a code that can show the factorials up to 1000. And this a ACM problem (623-500!). This code is very efficient too!. We pre-calculate all the 1000th factorials in a two dimensional array. In the main function just takes input and then just print.

Factorial of Big Number!

```

#include<stdio.h>
#include<string.h>
char f[10000];
char factorial[1010][10000];
void multiply(int k){
    int cin,sum,i;
    int len = strlen(f);
    cin=0;
    i=0;
    while(i<len){
        sum=cin+(f[i] - '0') * k;
        f[i] = (sum % 10) + '0';
        i++;
        cin = sum/10;
    }
    while(cin>0){
        f[i++] = (cin%10) + '0';
        cin/=10;
    }
    f[i]='\0';
    for(int j=0;j<i;j++){
        factorial[k][j]=f[j];
    }
    factorial[k][i]='\0';
}

```

```

void fac(){
    int k;
    strcpy(f,"1");
    for(k=2;k<=1000;k++)
        multiply(k);
}
void print(int n){
    int i;
    int len = strlen(factorial[n]);
    printf("%d!\n",n);
    for(i=len-1;i>=0;i--){
        printf("%c",factorial[n][i]);
    }
    printf("\n");
}
int main(){
    int n;
    factorial[0][0]='1';
    factorial[1][0]='1';
    fac();
    while(scanf("%d",&n)==1){
        print(n);
    }
    return 0;
}

```

Factorial Frequencies

Now how many digits in the factorial of N numbers..? Yes we can count them by the above process. But we can do it simply!

```

/*****      How many digits in N factorials      *****/
#include<stdio.h>
#include<math.h>
/*****/
double count_digit(long a){
    double sum;
    long i;
    if(a==0) return 1;
    else{
        sum=0;
        for(i=1;i<=a;i++){
            sum+=log10(i);
        }
        return floor(sum)+1;
    }
}
/*****/
int main(){
    double sum;
    long n;

```



```

while (scanf ("%ld", &n) == 1) {
    sum = count_digit (n);
    printf ("%0.0lf\n", sum);
}
return 0;
}

```

How many trailing zeros in Factorials ?

We know that factorials of any number has so many zeros (0's) at last...example $17! = 355687428096000$. It has 3 last zero digits. So we now see how many trailing zeros have in N factorials.

```

//*****          How many trailing zeros in N!
*****/*****//
#include<stdio.h>
long zero(long number, long factor) {
    long total, deno;
    if (number == 5) return 1;
    total = 0;
    deno = factor;
    while (deno < number) {
        total += number / deno;
        deno *= factor;
    }
    return total;
}
/*****/
int main() {
    long N, c2, c1;
    while (scanf ("%ld", &N) == 1) {
        c1 = zero (N, 2);
        c2 = zero (N, 5);
        if (c1 < c2) printf ("%ld\n", c1);
        else printf ("%ld\n", c2);
    }
    return 0;
}

```

Big Mod

We are not always interested in the full answers, however. Sometimes the remainder suffices for our purposes. In that case we use Modular Arithmetic. The key to efficient modular arithmetic is understanding how the basic operation of addition, subtraction, and multiplication work over a given modulus.

- Addition:- $(x + y) \bmod n \dots? = ((x \bmod n) + (y \bmod n)) \bmod n$.
- Subtraction:- $(x - y) \bmod n \dots? = ((x \bmod n) - (y \bmod n)) \bmod n$.

We can convert a negative number mod n to positive number by adding a multiple of n to it.

➤ Multiplication:- $xy \bmod n = (x \bmod n)(y \bmod n) \bmod n$.

What is $x^y \bmod n$? Exponentiation is just repeated multiplication, $(x \bmod n)^y \bmod n$. So if the x and y are very big equal to 32 bit. Then we can also compute it without overflow.

```

/* Big mod */
#include<stdio.h>
long square(long s){
    return s*s;
}
long bigmod(long b, long p, long m){
    if (p == 0)
        return 1;
    else if (p%2 == 0)
        return square( bigmod( b,p/2,m)) % m; // square(x) = x * x
    else
        return ((b % m) * bigmod( b,p-1,m)) % m;
}
int main(){
    long b,p,m,sum;
    while(scanf("%ld%ld%ld",& b,& p,& m)==3){
        sum = bigmod( b, p, m);
        printf("%ld\n", sum);
    }
    return 0;}

```

Number Conversions - Integer ↔ Binary

Convert an integer number into binary number

```

int inttobin(unsigned int a,int
*bin)
{
    int one=0;
    unsigned int c=1;
    int i ;
    for(i=31;i>=0;i--)
    {
        one = one + (a&c);
        bin[i] = (a&c)?1:0;
        c<<=1;
    }
    return one; }

```

Convert a binary number into binary number

```

unsigned int creatnum(int *num)
{
    int i;
    unsigned int a=0;
    for(i=0;i<32;i++)
    {
        a= a|num[i];
        if(i!=31)
            a <<= 1;
    }
    return a;
}

```

Integer ⇔ hex ⇔ Integer

```
char hex[100];
int num = 4095;
sprintf(hex,"%X",num); // convert the num in upper case hex decimal in hex
sprintf(hex,"%x",num); // convert the num in lower case hex decimal in hex
sscanf(hex,"%x",&num); // convert the hex number hex in integer num
```

Integer to any base

Following code help you to convert any integer number on any base from 2 to 36.

```
#include<string.h>
#include<stdio.h>
#define MAXDGT 200          /* maximum number of digit */

/* ***** */
/*      A general swap function that swap two object.      */
/*      Input   : Two object.                                */
/*      Return  : None                                       */
/* ***** */
template <class T>

void swap(T &x, T &y)
{   T tmp=x;   x=y;   y=tmp;   };

/* ***** */
/*      A general string reverse function that reverse the  */
/*      passed string and store the string int to the parameter.  */
/*      Input   : A string                                     */
/*      Return  : None                                       */
/* ***** */
void revstr(char *str)
{   int i=0,l=strlen(str);
    while(i<l)
        swap(str[i++],str[--l]);
}

/* ***** */
/*      A general base conversion function                    */
/*      Input   : A number n and a base b                    */
/*      Return  : A character array which contain the number n in base b */
/* ***** */
char *itob(long int n,int b=10)
{   char num[MAXDGT];
    int j,sign;
    register int i=0;
```

```

        if( (sign=n) <0 )
            n= -n;
do    {    j=n%b;
        num[i++]= (j<10) ? (j+'0') : ('A'+j-10);
        }while((n/=b) !=0);

    if(sign < 0)        num[i++]='-';

    num[i]='\0';
    revstr(num);
    return num;
}

/* Sample main */
main(void)
{    printf(itob(71),36);    }

```

Decimal To Roman

Roman number is very useful number system. Often we need to convert a decimal number into Roman number. In Roman number the following symbol are used.

i, v, x, l, c, m for 1, 5, 10, 50, 100 and 500 respectively.

```
#include<stdio.h>
```

```

/* ***** */
/* Convert number 1 to 10 */
/* Input : A integer number */
/* Return : None */
/* ***** */

void unit(int n)
{
    switch(n){
        case 3 : printf("i");
        case 2 : printf("i");
        case 1 : printf("i"); break;
        case 4 : printf("i");
        case 5 : printf("v"); break;
        case 6 : printf("vi"); break;
        case 7 : printf("vii"); break;
        case 8 : printf("viii"); break;
        case 9 : printf("ix"); break;
    }
}

/* ***** */
/* Convert number 10 to 100 */

```

```
/*      Input   : A integer number          */
/*      *****                               */
void ten(int n)
{
    switch(n){
        case 3 : printf("x");
        case 2 : printf("x");
        case 1 : printf("x"); break;
        case 4 : printf("x");
        case 5 : printf("l"); break;
        case 6 : printf("lx"); break;
        case 7 : printf("lxx"); break;
        case 8 : printf("lxxx"); break;
        case 9 : printf("xc"); break;
    }
}

/*      *****                               */
/*      Convert number 100 to 500           */
/*      Input   : A integer number          */
/*      *****                               */

void hnd(int n)
{
    switch(n){
        case 3 : printf("c");
        case 2 : printf("c");
        case 1 : printf("c"); break;
        case 4 : printf("c");
        case 5 : printf("M"); break;
    }
}

/*      *****                               */
/*      Convert an integer number into roman system */
/*      Input   : A integer number          */
/*      *****                               */

void roman(int n)
{
    int a,i;
    if(n>=500)
        {
            a=n/500 ;
            for(i=1;i<=a;i++)
                printf("M");
        }
    n=n%500;
    hnd(n/100);
    n=n%100;
    ten(n/10);
    unit(n%10);
}

main(void)
{
    roman(390);
    return 0;
}
```

Josephus Problem

Flavius Josephus was a famous historian of the first century. Legend has it that Josephus wouldn't have lived to become famous without his mathematical talents. During the Jewish-Roman war, he was among a band of 41 Jewish rebels trapped in cave by the Romans. Preferring suicide to capture, the rebels decided to form a circle and, proceeding around it, to kill every third remaining person until no one was left. But Josephus, along with an un-indicted co-conspirator, wanted none of this suicide nonsense; so he quickly calculated where he and his friend stand in the vicious circle.

```
#include <stdio.h>
#define MAX 150

/** Implementation of circular queue **/

int queue[MAX];
int e,f,r;

/** _f is front; _r is rear & _e is element number **/

/* ***** */
/* A general push algorithm act over queue */
/* ***** */

void push(int i)
{
    r++;
    e++;
    if(r>=MAX)
        r=1;
    queue[r] = i;
}

/* ***** */
/* A general pop algorithm act over queue */
/* ***** */

int pop(void)
{
    e--;
    int i=queue[f];
    f++;
    if(f>=MAX)
        f = 1;
    return i;
}
```

```
/** End of circular queue **/  
/* ***** */  
/* A general joshuf function. */  
/* This function start removing */  
/* element from first element */  
/* and return the sirviver */  
/* ***** */  
  
int joshups(int n,int v)  
{  
    register int i;  
    e=n;  
    for(i=1;i<=n;i++)  
        queue[i] = i;  
    f = 1; // 0 for serviving first element.  
    r = n; // n+1 for serviving first element.  
    i = 0;  
    if(n > 1)  
        pop();  
    while(e!=1)  
    {  
        if(i!=v)  
        {  
            i++;  
            push(pop());  
        }  
        else  
        {  
            pop();  
            i = 0;  
        }  
    }  
    return queue[f];  
}  
  
/*sample main function*/  
  
main(void)  
{  
    int i,m;  
    scanf("%d",&i);  
    while(i)  
    {  
        m=1;  
        while((joshups(i,m++)) != 13 );  
        printf("%d",m);  
        putchar('\n');  
        scanf("%d",&i);  
    }  
  
    return 0; }
```

Combinations

```

/* ***** */
/* Calculate nCm */
/* Input : Two integer number n m */
/* Return : The nCm */
/* ***** */

double nCr(int n,int m)
{
    int k;
    register int i,j;
    double c,d;

    c=d=1;
    k=(m>(n-m)) ?m:(n-m);
    for(j=1,i=k+1;(i<=n);i++,j++)
    {
        c*=i;
        d*=j;
        if( !fmod(c,d) && (d!=1) )
        { c/=d;
          d=1;
        }
    }

    return c;
}

/* A sample main function */
main(void)
{
    int n,m;

    while( scanf("%d%d",&n,&m) !=EOF)
        printf("%.0lf\n",nCr(n,m));
    return 0;
}

```

Another way to calculate the nC_m by using the Pascal's triangle.

```

#include<stdio.h>
#define MAXTRI 50

unsigned long int pas[MAXTRI][MAXTRI];

void pascals(int n)
{

```



```

register int i,j;
pas[0][0]=1;
pas[1][0]=pas[1][1]=1;
for(i=2;i<=n;i++)
{
pas[i][0]=1;
for(j=1;j<i;j++)
{
pas[i][j]= pas[i-1][j-1]+pas[i-1][j];
}
pas[i][j]=1;
}
}

main(void)
{
pascals(10);
int n,m;
while (scanf("%d%d",&n,&m) !=EOF)
{
printf("%lu",pas[n][m]);
}
return 0;
}

```

Combination with repeated objects :

If in a word of s length character, a character is repeated l times, then the number of arrangement is:

$$\frac{s!}{l!}$$

If there is more than one repeated character then we can write,

$$\frac{s!}{l_1! * l_2! * \dots * l_n!}$$

where,

l_1 , is the repetition times of first repeated character.

l_2 , is for second repeated character

and, so on...

but remember calculating both $n!$ and $l_1! * l_2! * l_3! * \dots * l_n!$ may occur overflow. So I use

$$\frac{s!}{1! * 2! * \dots * n!}$$

$$= \frac{1 * 2 * 3 * \dots * s}{1 * 2 * \dots * 1 * 1 * 2 * 3 * \dots * 2 * \dots * 1 * 2 * \dots * n}$$

$$= \frac{1}{1} * \frac{2}{2} * \dots * \frac{m}{1} * \frac{m+1}{1} * \frac{m+2}{2} * \dots * \frac{m2}{i2} * \dots * \frac{s}{ln}$$

The code of this type of combination is as follows:

```
#include<stdio.h>
#include<math.h>
#include<string.h>

#define MAX 30

/* ***** */
/* A sample function that calculate how many ways that you can */
/* rearrange a word with its letter */
/* ***** */
double test(char *str)
{
    int de[MAX]={0};
    int ss[300] = {0};
    int l = strlen(str);
    int i,j=0;
    double c=1,d=1;
    for(i=0;i<l;i++)
    {
        ss[str[i]]++;
        if(ss[str[i]] > 1)
            de[j++] = ss[str[i]];
    }
    c = 1;
    for(i=2;i<=l;i++)
    {
        c*=i;

        if(j>0)
            d*= de[--j];
        if((d!=1) && !(fmod(c,d)))
        {
            c /= d;
            d=1;
        }
    }
    return c;
}

/* A sample main function */
```

```

main(void)
{
    char word[MAX];
    int n;
    int j=0;
    scanf("%d",&n);
    for(;n>0;n--)
    {
        scanf("%s",word);
        printf("Data set %d: %.0f",++j,test(word));
        putchar('\n');
    }
    return 0;
}

```

longest common subsequence (LCS)

Given two sequence X and Y, we say that a sequence z is a **common subsequence** of C and Y if Z is a subsequence of both X and Y.

longest common subsequence (LCS) is just the longest "common subsequence" of two sequences.

LCS LENGTH(X,Y)

Input two sequence X<x₁,x₂,x₃,...,x_m> and Y<y₁,y₂,y₃.....y_n>. It stores the C[i,j] values in the table C[0...m,0...n] whose entries are computed in row major order. It also maintain the table b[1...m,1...n] to simplify construction of an optimal solution.

```

1. m <- length[X]
2. n <- length[Y]
3. for i <- 1 to m
4.     do c[i,0] <- 0
5. for j <- 0 to n
6.     do c[0,j] <- 0
7. for i <- 1 to m
8.     for j <- 1 to n
9.         do if xi = yj
10.            then c[i,j] <- c[i-1,j-1]+1
11.                b[i,j] <- 1
12.            else if c[i-1,j] >= c[i,j-1]
13.                then c[i,j] <- c[i-1,j]
14.                    b[i,j] <- 2
15.            else c[i,j] <- c[i,j-1]
16. return c and b

```

PRINT LCS(b,X,i,j)

```

1. if i = 0 or j = 0
2.     then return
3. if b[i,j] = 1
4.     then PRINT_LCS(b,X,i-1,j-1)
5. else if b[i,j] = 2
6.     then PRINT_LCS(b,X,i-1,j)
7. else PRINT_LCS(b,X,i,j-1)

```

Code for LCS

```

#include<stdio.h>
#include<string.h>

#define MAX 100 // size of each sequence

char str1[MAX],str2[MAX]; // the sequences

/* ***** */
/* This function print the LCS to the screen */
/* Input : A table generated by the LCS_LNT */
/* and the length of the sequences */
/* Output: None */
/* ***** */

void p_lcs(int b[MAX][MAX],int i,int j)
{
    if ( (i == 0) || (j == 0) ) return ;
    if( b[i][j] == 1 )
    { p_lcs(b,i-1,j-1);
      printf("%3c",str1[i-1]);
    }
    else if( b[i][j] == 2) p_lcs(b,i-1,j);
    else p_lcs(b,i,j-1);
}

/* ***** */
/* This function calculate the LCS length */
/* Input : Tow Sequence and an bool I. If */
/* I is FALSE(0) then the function */
/* do not print the LCS and if */
/* TRUE(1) then print using the */
/* above p_lcs function */
/* Output: None */
/* ***** */

void LCS_LNT(bool I)
{
    int c[MAX][MAX]={0},b[MAX][MAX]={0},l1,l2;
    l1 = strlen(str1)+1;
    l2 = strlen(str2)+1;
}

```

```

register int i,j;
for(i=1;i<l1;i++)
{ for(j=1;j<l2;j++)
  { if( str1[i-1] == str2[j-1] )
    { c[i][j] = c[i-1][j-1] + 1;
      b[i][j] = 1;
    }
    else if(c[i-1][j] >= c[i][j-1])
    { c[i][j] = c[i-1][j];
      b[i][j] = 2;
    }
    else c[i][j] = c[i][j-1];
  }
}
printf("%d\n",c[l1-1][l2-1]);
if(I) p_lcs(b,l1-1,l2-1);
}

/* a sample main function */
main(void)
{
while(1)
{
if(!(gets(str1))) return 0;
if(!(gets(str2))) return 0;
LCS_LNT(1);
}
}

```

Another code for finding LCS

```

#include<stdio.h>
#include<string.h>
#define MAX 105

char str1[MAX][50],str2[MAX][50],lcs[MAX][50];
int lcswl;

/* ***** */
/* This function print the LCS to the screen */
/* Input : A table generated by the LCS_LNT */
/* and the length of the sequences */
/* Output: None */
/* ***** */
void p_lcs(int b[MAX][MAX],int i,int j)
{
if ( (i == 0) || (j == 0) ) return ;
if( b[i][j] == 1 )
{
p_lcs(b,i-1,j-1);
strcpy(lcs[lcswl++],str1[i-1]);
}
else if( b[i][j] == 2) p_lcs(b,i-1,j);
}

```

```

    else
        p_lcs(b,i,j-1);
}

/* ***** */
/* This function calculate the LCS length */
/* Input : Tow Sequence and an bool I. If */
/* I is FALSE(0) then the function */
/* do not print the LCS and if */
/* TRUE(1) then print using the */
/* above p_lcs function */
/* Output: None */
/* ***** */
void LCS_LNT(int l1, int l2,bool I)
{
    int c[MAX][MAX]={0},b[MAX][MAX]={0};
    register int i,j;
    for(i=1;i<l1;i++)
    {
        for(j=1;j<l2;j++)
        {
            if( !(strcmp(str1[i-1],str2[j-1])))
            {
                c[i][j] = c[i-1][j-1] + 1;
                b[i][j] = 1;
            }
            else if(c[i-1][j] >= c[i][j-1])
            {
                c[i][j] = c[i-1][j];
                b[i][j] = 2;
            }
            else
                c[i][j] = c[i][j-1];
        }
    }
}

if(I)
{
    lcswl = 0;
    p_lcs(b,l1-1,l2-1);
    j = c[l1-1][l2-1];
    printf("%s",lcs[0]);
    for(i = 1; i < j ;i++)
        printf(" %s",lcs[i]);
    putchar('\n');
}

/* Sample main function */
main(void)
{
    char word[50];
    int i=0,j=0,l1,l2,ln;
    while(scanf("%s",word) != EOF)
    {
        ln = strlen(word);
        if(ln==1)
            if(word[0] == '#')
            {
                if(i==0)
                {
                    i = 1;

```

```

        l1 =j;
        j = 0;
        continue;
    }
    else
    {
        l2 = j;
        j = i = 0;
        test(l1+1,l2+1,1);
        continue;
    }
}
if(i==0) strcpy(str1[j++],word);
else strcpy(str2[j++],word);
}
return 0;
}

```

Matrix Chain Multiplication (MCM)

Background

If two matrix A and B whose dimension is (m,n) and (n,p) respectively then the multiplication of A and B needs $m*n*p$ scalar multiplication.

Suppose you are given a sequence of n matrix A_1, A_2, \dots, A_n . Matrix A_i has dimension (P_{i-1}, P_i) . Your task is to parenthesize the product A_1, A_2, \dots, A_n such a way that minimize the number of scalar product.

As matrix multiplication is associative so all the parenthesizations yield the same product.

MATRIX CHAIN ORDER(P)

The input is a sequence $P = \langle P_0, P_1, P_2, \dots, P_n \rangle$ where $\text{length}[P] = n+1$. The procedure uses an auxiliary table $m[1 \dots n, 1 \dots n]$ for storing the $m[i, j]$ costs and auxiliary table $s[1 \dots n, 1 \dots n]$ that records which index of k achieve the optimal cost in computing $m[i, j]$.

```

1. n <- length[P]-1
2. for i <- 1 to n
3.     do m[i, j] <- 0
4. for l <- 2 to n
5.     do for i <- 1 to n - l + 1
6.         do j <- i + l - 1

```

```

7.          m[i,j] <- INF      // INF is infnity
8.          for k <- i to j -1
9.            do q <- m[i,k]+m[k+1,j]+ Pi-1PkPj
10.         if q < m[i,j]
11.         then m[i,j] <- q
12.         s[i,j] <- k
13. return m,s

```

PRINT(s,i,j)

The following recursive procedure prints an optimal parentthesization of $(A_i, A_{i+1}, \dots, A_j)$ given the s table computed by

MATRIX CHAIN ORDER and indices i and j. The initial call PRINT(s,1,n) prints optimal parentthesization of (A_1, A_2, \dots, A_n) .

```

1.  if i == j
2.    then print "A"i
3.    else print "("
4.         PRINT(s,i,s[i,j])
5.    PRINT(s,s[i,j]+1,j)
6.    print ")"

```

Code :

```

#include<stdio.h>
#define MAX 15          /* number of matrix */
#define INF 4294967295 /* a maximum number of multiplication */

int num;
/* ***** */
/* Print out the sequence */
/* Input : A two dimentional array(created */
/* by the matrix chan order function) */
/* with there starting point */
/* Return : None */
/* ***** */
void print(unsigned long s[][MAX],int i,int j)
{
  if(i==j)
    printf("A%d",num++);
  else
  {
    printf("(");
    print(s,i,s[i][j]);
    printf(" x ");
    print(s,s[i][j]+1,j);
    printf(")");
  }
}

```



```

/* ***** */
/* Find out the order */
/* Input : A sequence of dimension and the */
/*          number of matrix */
/* Return : none */
/* ***** */

void matrix_chan_order(int *p,int n)
{
    unsigned long m[MAX][MAX] = {0};
    unsigned long s[MAX][MAX] = {0};
    unsigned int q;
    int l,j,i,k;
    for(l = 2; l <= n ;l++)
    {
        for(i = 1 ; i <= n - l +1 ; i++)
        {
            j = i + l -1;
            m[i][j] = INF;          for(k=i ; k < j ; k++)          {
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if(q < m[i][j])
                {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
        num =l;
        print(s,l,n);
    }

/* A sample main function */

main(void)
{
    int n;
    int p[MAX]={0};
    int i;

    while (scanf("%d",&n),n)
    {
        for(i=1;i<=n;i++)
            scanf("%d %d",&p[i-1],&p[i]);
        matrix_chan_order(p,n);
        putchar('\n');

    }
    return 0;
}

```

Big Number Addition

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#define MAX 1000
void reverse(char *from, char *to ){
    int len=strlen(from);
    int l;
    for(l=0;l<len;l++){
        to[l]=from[len-l-1];
    }
    to[len]='\0';
}
void call_sum(char *first, char *sec, char *result){
    char F[MAX], S[MAX], Res[MAX];
    int f,s,sum,extra,now;
    f=strlen(first);
    s=strlen(sec);
    reverse(first,F);
    reverse(sec,S);
    for(now=0,extra=0;(now<f && now<s);now++){
        sum=(F[now]-'0') + (S[now]-'0') + extra;
        Res[now]=sum%10 +'0';
        extra= sum/10;
    }
    for(;now<f;now++){
        sum=F[now] + extra-'0';
        Res[now]=sum%10 +'0';
        extra=sum/10;
    }
    for(;now<s;now++){
        sum=S[now] + extra-'0';
        Res[now]=sum%10 +'0';
        extra=sum/10;
    }
    if(extra!=0) Res[now++]=extra+'0';
    Res[now]='\0';
    if(strlen(Res)==0) strcpy(Res,"0");
    reverse(Res,result);}

int main(){
    char fir[MAX],sec[MAX],res[MAX];
    while(scanf("%s%s",&fir,&sec)==2){
        call_sum(fir,sec,res);
        int len=strlen(res);
        for(int i=0;i<len;i++) printf("%c",res[i]);
        printf("\n");
    }
    return 0;
}
```

Big Number Subtraction

```

/*****          Big Number Subtraction          *****/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#define MAX 1000
/*****/
void reverse(char *from, char *to ){
    int len=strlen(from);
    int l;
    for(l=0;l<len;l++){
        to[l]=from[len-l-1];
    }
    to[len]='\0';
}
int call_minus(char *large, char *small, char *result){
    char L[MAX], S[MAX];
    int l,s,now,hold,diff;
    l=strlen(large);
    s=strlen(small);
    bool sign = 0;
    if(l<s){
        strcpy(result,large);
        strcpy(large,small);
        strcpy(small,result);
        now=l; l=s; s=now;
        sign = 1;
    }
    //return 0;
    if(l==s){
        if(strcmp(large, small)<0){
            strcpy(result,large);
            strcpy(large,small);
            strcpy(small,result);
            now=l; l=s; s=now;
            sign =1;
        }
        //return 0;
    }
    reverse(large,L);
    reverse(small,S);
    for(;s<l;s++)
        S[s]='0';
    S[s]='\0';
    for(now=0,hold=0;now<l;now++){
        diff=L[now]-(S[now]+hold);
        if(diff<0){
            hold=1;
            result[now]=10+diff+'0';
        }
        else{
            result[now]=diff+'0';
            hold=0;
        }
    }
}

```

```

    }
    for(now=l-1;now>0;now--){
        if(result[now]!='0')
            break;
    }
    result[now+1]='\0';
    reverse(result,L);
    strcpy(result,L);
    //return 1;
    return sign;
}
int main(){
    char fir[MAX],sec[MAX],res[MAX];
    while(scanf("%s%s",&fir,&sec)==2){
        if(call_minus(fir,sec,res)==1)
            printf("-");
        int len = strlen(res);
        for(int i=0;i<len;i++)
            printf("%c",res[i]);
        printf("\n");

    }
    return 0;
}

```

Big Number Multiplication

```

/*****      Big Number Multiplication      *****/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#define MAX 1000
/*****/
void reverse(char *from, char *to ){
    int len=strlen(from);
    int l;
    for(l=0;l<len;l++){
        to[l]=from[len-l-1];
    }
    to[len]='\0';
}
/*****/
void call_mult(char *first,char *sec,char *result){
    char F[MAX],S[MAX],temp[MAX];
    int f_len,s_len,f,s,r,t_len,hold,res;
    f_len=strlen(first);
    s_len=strlen(sec);
    reverse(first,F);
    reverse(sec,S);
    t_len=f_len+s_len;
    r=-1;

```

```

for(f=0;f<=t_len;f++)
    temp[f]='0';
temp[f]='\0';
for(s=0;s<s_len;s++){
    hold=0;
    for(f=0;f<f_len;f++){
        res=(F[f]-'0')*(S[s]-'0') + hold+(temp[f+s]-'0');
        temp[f+s]=res%10+'0';
        hold=res/10;
        if(f+s>r) r=f+s;
    }
    while(hold!=0){
        res=hold+temp[f+s]-'0';
        hold=res/10;
        temp[f+s]=res%10+'0';
        if(r<f+s) r=f+s;
    }
    f++;
}
for(;r>0 && temp[r]=='0';r--);
temp[r+1]='\0';
reverse(temp,result);
}
/*****
int main(){
    char fir[MAX],sec[MAX],res[MAX];
    while(scanf("%s%s",&fir,&sec)==2){
        call_mult(fir,sec,res);
        int len=strlen(res);
        for(int i=0;i<len;i++) printf("%c",res[i]);
        printf("\n");
    }
    return 0;
}

```

Big Number Division and Remainder

```

/*****          Big Number division          *****/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#define MAX 1000
/*****
int call_div(char *number,long div,char *result){
    int len=strlen(number);
    int now;
    long extra;
    char Res[MAX];
    for(now=0,extra=0;now<len;now++){
        extra=extra*10 + (number[now]-'0');

```

```

        Res[now]=extra / div +'0';
        extra%=div;
    }
    Res[now]='\0';
    for(now=0;Res[now]!='\0';now++);
    strcpy(result, &Res[now]);
    if(strlen(result)==0)
        strcpy(result, "0");
    return extra;
}
/*****
int main(){
    char fir[MAX],res[MAX];
    long sec,remainder;
    while(scanf("%s%ld",&fir,&sec)==2){
        if(sec==0) printf("Divide by 0 error\n");
        else{
            remainder=call_div(fir,sec,res);
            int len=strlen(res);
            for(int i=0;i<len;i++) printf("%c",res[i]);
            printf("\t%ld",remainder);
            printf("\n");
        }
    }
    return 0;
}

```

Big Number Square Root

```

/*****      Big Number Sqrt      *****/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#define MAX 1000
/*****
int call_minus(char *large, char *small, char *result){
    char L[MAX], S[MAX];
    int l,s,now,hold,diff;
    l=strlen(large);
    s=strlen(small);
    if(l<s)
        return 0;
    if(l==s){
        if(strcmp(large, small)<0)
            return 0;
    }
    reverse(large,L);
    reverse(small,S);
    for(;s<l;s++)
        S[s]='\0';

```

```

S[s]='\0';
for(now=0,hold=0;now<l;now++){
    diff=L[now]-(S[now]+hold);
    if(diff<0){
        hold=1;
        result[now]=10+diff+'0';
    }
    else{
        result[now]=diff+'0';
        hold=0;
    }
}
for(now=l-1;now>0;now--){
    if(result[now]!='0')
        break;
}
result[now+1]='\0';
reverse(result,L);
strcpy(result,L);
return 1;
}
/*****/
void call_sqrt(char *number,char *result,char *extra){
    int num,start,e,mul,l,r=0,len;
    char left[MAX],after[MAX];
    char who[5],temp[MAX],two[5];
    len=strlen(number);
    if(len%2==0){
        num=10*(number[0]-'0') + number[1]-'0';
        start=2;
    }
    else{
        num=number[0]-'0';
        start=1;
    }
    mul=(int) sqrt(num);
    result[0]=mul+'0';
    result[1]='\0';
    if(num-mul*mul ==0)
        extra[0]='\0';
    else
        sprintf(extra,"%d",num-mul*mul);
    for(;start<len;start+=2){
        e=strlen(extra);
        extra[e]=number[start];
        extra[e+1]=number[start+1];
        extra[e+2]='\0';
        two[0]='2';
        two[1]='\0';
        call_mult(result,two,left);
        l=strlen(left);
        for(mul=9;mul>=0;mul--){
            who[0]=mul+'0';
            who[1]='\0';

```

```

        strcat(left,who);
        call_mult(left,who,after);
        if(call_minus(extra,after,temp)==1){
            result[++r]=mul+'0';
            result[r+1]='\0';
            strcpy(extra,temp);
            break;
        }
else
        left[l]='\0';
    }
    result[++r]='\0';
}
/*****
int main(){
    char fir[MAX],ex[MAX],res[MAX];
    while(scanf("%s",&fir)==1){
        call_sqrt(fir,res,ex);
        int len=strlen(res);
        for(int i=0;i<len;i++) printf("%c",res[i]);
        printf("\n");
    }
    return 0;
}

```

Fibonacci Numbers

```

/** This run O(log n) time//
#include<stdio.h>
long conquer_fibonacci(long n){
    long i,h,j,k,t;
    i=h=1;
    j=k=0;
    while(n>0){
        if(n%2==1){
            t=j*h;
            j=i*h + j*k +t;
            i=i*k + t;
        }
        t=h*h;
        h=2*k*h + t;
        k=k*k + t;
        n=(long) n/2;}
    return j;}
int main(){
    long n,res;
    while(scanf("%ld",&n)==1){
        res=conquer_fibonacci(n);
        printf("%ld\n",res);}
    return 0;
}

```


Fibnacci Number by Big Integer

```
#include<iostream.h>
#include<string.h>
int main()
{
    char *fibonacci[5001]={0};
    fibonacci[0]="0";
    fibonacci[1]="1";
    int l1=strlen(fibonacci[0]);
    int l2=strlen(fibonacci[1]);
    int i;
    for(long i=2;i<=5000;i++)
    {
        char str[10000];
        if(l1>=l2)l1=l1;
        else l1=l2;
        int ca=0;
        long j,k,m,p;
        for(j=l1-1,k=l2-1,m=0,p=0;p<l1;j--,k--,m++,p++)
        {
            int s1;
            if(j<0) fibonacci[i-2][j]='0';
            s1=fibonacci[i-2][j]-48;
            int s2;
            if(k<0) fibonacci[i-1][k]='0';
            s2=fibonacci[i-1][k]-48;
            int ans=0;
            ans+=s1+s2+ca;
            if(ans>9)
            {
                str[m]=(ans-10)+48;
                ca=1;
            }
            else
            {
                str[m]=ans+48;
                ca=0;
            }
        }
        if(ca>0){str[m]=ca+48; m++;}
        str[m]='\0';
        fibonacci[i]=new char[m+1];
        long y=0;
        for(long x=m-1;x>=0;x--,y++) fibonacci[i][y]=str[x];
        fibonacci[i][y]='\0';
        l1=strlen(fibonacci[i-1]);
        l2=strlen(fibonacci[i]);
    }
}
```

```

int n;

while(cin>>n)
{
    cout<<"The Fibonacci number for "<<n<<" is "<<fibonacci[n]<<"\n";
}
return 0;
}

```

BFS/DFS (Maze Problem)

Input :

```

8 8
#.#.#.#.
.....
#.#.....
#.#.#.#.
..#..##.
#..##...
...#...#
#.s#d.#.

```

```

#include<stdio.h>
#include<conio.h>
#include<values.h>

#define N 100
#define MAX MAXINT

int mat[N][N], Q[N][3], cost[N][N], front = -1, rear = -1;
int m, n, sc, sr, p, nr, nc, r, c, leaf, i, j, res[10][10];
int R[4] = {0, -1, 0, 1};
int C[4] = {-1, 0, 1, 0};

void nQ(int r, int c, int p)
{
    Q[++rear][0] = r, Q[rear][1] = c, Q[rear][2] = p;}

void dQ(int *r, int *c, int *p)
{
    *r = Q[++front][0], *c = Q[front][1], *p = front;
}

void bfs(int sr, int sc, int p)
{
    nQ(sr, sc, p);
    do{
        dQ(&r, &c, &p);
        for(int i=0; i<4; i++)
        {

```

```
        nr = r + R[i], nc = c + C[i];
        if(mat[nr][nc]==1)
        {
            if(cost[nr][nc]>cost[r][c]+1)
                cost[nr][nc]=cost[r][c]+1, nQ(nr, nc, p);
        }
    }
    } while (rear!=front);
    leaf = p;
}
void show()
{
    for(int i=0; i<=m; i++)
    {
        for(int j=0; j<=n; j++)
        {
            if(res[i][j])
                printf("X");
            else
                printf(" ");
        }
        printf("\n");
    }
}
void dfs(int leaf)
{
    if(Q[leaf][2]==-1)
    {
        res[Q[leaf][0]][Q[leaf][1]] = 1;
        return;
    }
    dfs(Q[leaf][2]);
    res[Q[leaf][0]][Q[leaf][1]] = 1;
}
void main()
{
    clrscr();
    char ch;
    freopen("maze.txt", "r", stdin);
    scanf("%d%d", &m, &n);
    getchar();
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
        {
            cost[i][j] = MAX;
            scanf("%c", &ch);
            if(ch=='#')
                mat[i][j] = 0;
            else if(ch=='.')
                mat[i][j] = 1;
            else if(ch=='s')
```

```

                mat[i][j] = 2, sc = j, sr = i;
            else
                mat[i][j] = 3, res[i][j] = 1;
        }
        getchar();
    }
    bfs(sr, sc, -1);
    dfs(leaf);    show();
}

```

MinSum (DFS/QUEUE)

Input :

```

5 6
3 4 1 2 8 6
6 1 8 2 7 4
5 9 3 9 9 5
8 4 1 3 2 6
3 7 2 1 2 3

```

Code

```

#include<stdio.h>
#include<conio.h>
#include<values.h>

#define N 100
#define MAX MAXINT

int mat[N][N], M[N][N], Q[N][4], front = -1, rear = -1;
int m, n, nc, nr, p, s, finalSum=MAX, leaf, r, c, i;

void init()
{
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            M[i][j] = MAX, mat[i][j] = 0;
}

void nQ(int r, int c, int p, int s)
{
    Q[++rear][0] = r;    Q[rear][1] = c;    Q[rear][2] = p;
    Q[rear][3] = s;
}

void dQ(int *r, int *c, int *p, int *s)
{
    *r = Q[++front][0];
    *c = Q[front][1];
    *p = front;
    *s = Q[front][3];
}

```

```
void bfs()
{
    for(r=0, c=0; r<m; r++)
        nQ(r, c, -1, mat[r][c]);
    do
    {
        dQ(&r, &c, &p, &s);
        if(c<n-1)
            for(i=-1; i<2; i++)
            {
                nr=(m+r+i)%m, nc=c+1;
                if(M[nr][nc] > s+mat[nr][nc])
                    nQ(nr, nc, p, s+mat[nr][nc]), M[nr][nc] = s+mat[nr][nc];
            }
        else if(s<finalSum)
            finalSum = s, leaf = p;
    } while(rear!=front);
}

void dfs(int leaf)
{
    if(Q[leaf][2]==-1)
    {
        printf(" <%d, %d>", Q[leaf][0]+1, Q[leaf][1]+1);
        return;
    }
    dfs(Q[leaf][2]);
    printf(" <%d, %d>", Q[leaf][0]+1, Q[leaf][1]+1);
}

void main()
{
    clrscr();
    int i, j, t;
    init();
    freopen("in.txt", "r", stdin);
    scanf("%d%d", &m, &n);
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
        {
            scanf("%d", &t);
            mat[i][j] = t;
        }
    bfs();

    printf("Final sum: %d\nPath:", finalSum);

    dfs(leaf);
}
```

Floyed Warshal**Input**

```
5 7
1 2 4
1 3 1
1 5 6
2 5 3
2 4 1
3 2 1
4 5 1
0 0
```

Code :

```
#include<stdio.h>
#include<values.h>

#define N 100
#define INF MAXINT

int mat[N][N], path[N][N], n, e;

void initMat()
{
    for(int i=1; i<=n; i++)
        for(int j=1; j<=n; j++)
            mat[i][j] = INF;
}

void initPath()
{
    for(int i=1; i<=n; i++)
        for(int j=1; j<=n; j++)
            if(mat[i][j]!=INF)
                path[i][j] = j;
            else
                path[i][j] = 0;
}

void floyd_warshall()
{
    for(int k=1; k<=n; k++)
        for(int i=1; i<=n; i++)
            for(int j=1; j<=n; j++)
                if(mat[i][k]!=INF && mat[k][j]!=INF)
                    if(mat[i][k]+mat[k][j] < mat[i][j])
                        mat[i][j] = mat[i][k] + mat[k][j],
                        path[i][j] = path[i][k];
}
```

```
}  
void showPath(int i, int j)  
{  
    if(i==j)  
    {  
        printf("->%d", i);  
        return;  
    }  
    printf("->%d", i);  
    showPath(path[i][j], j);  
}  
void main()  
{  
    while(scanf("%d%d", &n, &e) && n && e)  
    {  
        initMat();  
        for(int i, j, c, k=0; k<e; k++)  
        {  
            scanf("%d%d%d", &i, &j, &c);  
            mat[i][j] = c;  
        }  
        initPath();  
        floyd_warshall();  
        for(i=1; i<=n; i++)  
        {  
            for(j=1; j<=n; j++)  
                if(path[i][j])  
                {  
                    printf("%d", i);  
                    showPath(path[i][j], j);  
                    printf("\n");  
                }  
            printf("\n");  
        }  
    }  
}
```

Graph Coloring

```
#include<stdio.h>  
int a[20][20],x[20],n,m;  
void next(int k)  
{  
    int j;  
    while(1)
```

```

    {
        x[k]=(x[k]+1)%(m+1);
        if(x[k]==0)
            return;
        for(j=1;j<=n;j++)
            if(a[k][j]!=0&&x[k]==x[j])
                break;
        if(j==n+1)
            return;
    }
}
void mcolor(int k)
{
    int j;
    while(1)
    {
        next(k);
        if(x[k]==0)
            return;
        if(k==n)
        {
            printf(" ");
            for(j=1;j<=n;j++)
                printf("%2d",x[j]);
        }
        else
            mcolor(k+1);
    }
}
void main()
{
    int i,u,v;
    printf("\n\n Enter how many colors : ");
    scanf("%d",&m);
    printf("\n\n Enter how many nodes(0<n<20) :");
    scanf("%d",&n);
    printf("\n\n Enter your edges(ex- u sp v) (press 'e' for end) : \n");
    for(i=1;i<=(n*n)/2;i++)
    {
        if(getchar()=='e')
            break;
        scanf("%d%d",&u,&v);
        a[u][v]=a[v][u]=1;
    }mcolor(1); printf("\n\n");
}

```

Cycle Detection (Hamiltonian)

```

#include<stdio.h>
int a[20][20],x[20],n;
void next(int k)
{
    int j;

```



```
while(1)
{
    x[k]=(x[k]+1)%(n+1);
    if(x[k]==0)
        return;
    if((a[x[k-1]][x[k]]!=0)
    {
        for(j=1;j<=k-1;j++)
            if(x[k]==x[j])
                break;
        if(j==k)
            if((k<n)|| (k==n&& a[x[n]][x[1]]!=0))
                return;
    }
}
}
void hamilt(int k)
{
    int j;
    while(1)
    {
        next(k);
        if(x[k]==0)
            return;
        if(k==n)
        {
            printf(" ");
            for(j=1;j<=n;j++)
                printf("%2d",x[j]);
        }
        else
            hamilt(k+1);
    }
}
void main()
{
    int i,u,v;
    x[1]=1;
    printf("\n\n Enter how many nodes(0<n<20) :");
    scanf("%d",&n);
    printf("\n\n Enter your edges(ex- u sp v) (press 'e' for end) : \n");
    for(i=1;i<=(n*n)/2;i++)
    {
        if(getchar()=='e')
            break;
        scanf("%d%d",&u,&v);
        a[u][v]=a[v][u]=1;
    }
    hamilt(2);
    printf("\n\n");
}
```

Finding Articulation Point

```

#include<stdio.h>
int d[11],num=1,b[11][11],l[11],at[11],s=1;
void art(int u,int v)
{
    int i,j=1,w,f=0;
    d[u]=num;
    l[u]=num;
    num++;
    for(i=1;b[u][i]!=0;i++)
    {
        w=b[u][i];
        if(d[w]==0)
        {
            art(w,u);
            l[u]=(l[u]<l[w])?l[u]:l[w];
        }
        else if(w!=v)
            l[u]=(l[u]<d[w])?l[u]:d[w];
        if(d[u]<=l[w])
            f=1;
    }
    if(f)
        at[s++]=u;
}
void main()
{
    int i,j,a[11][11],n,u,v,k,f=0;
    for(i=1;i<11;i++)
        for(j=1;j<11;j++)
            a[i][j]=0;
    printf("\n\n Enter how many nodes (0<n<11) :");
    scanf("%d",&n);
    printf("\n\n Enter your edges(ex- u sp v) (press 'e' for end) : ");
    for(i=1;i<=(n*n)/2;i++)
    {
        if(getchar()=='e')
            break;
        scanf("%d%d",&u,&v);
        a[u][v]=a[v][u]=1;
    }
    for(i=1;i<=n;i++)
    {
        k=1;
        for(j=1;j<=n;j++)
            if(a[i][j])
            {
                b[i][k]=j;
                k++;
            }
    }
}

```

```
        b[i][k]=0;
    }
    for(j=1,i=1;b[1][j]!=0;j++)
    {
        k=b[1][j];
        if(b[k][2])
            i++;
    }
    if(j==i)
        f=1;
    art(1,1);
    at[s]=-9;
    printf("\n\n Articulation points are : ");
    if(!f)
        for(i=1;at[i]!=-9;i++)
            printf("%3d",at[i]);
    if(f)
        for(i=1;at[i]!=1;i++)
            printf("%3d",at[i]);
    printf("\n\n");
}
```

APPENDIX C

STANDARD TEMPLATE LIBRARY (STL)



The programming languages for the contest will be C, C++, Pascal, and Java and the `scanf()/printf()` family of functions. The C++ string library and Standard Template Library (STL) is also available. This part of this book contains an introductory note about STL^[9].

Standard Template Library (STL)

The Standard Template Library, or *STL*, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a *generic* library, meaning that its components are heavily parameterized: almost every component in the STL is a template. You should make sure that you understand how templates work in C++ before you use the STL.

Containers and algorithms

Like many class libraries, the STL includes *container* classes: classes whose purpose is to contain other objects. The STL includes the classes `vector`, `list`, `deque`, `set`, `multiset`, `map`, `multimap`, `hash_set`, `hash_multiset`, `hash_map`, and `hash_multimap`. Each of these classes is a template, and can be instantiated to contain any type of object. You can, for example, use a `vector<int>` in much the same way as you would use an ordinary C array, except that `vector` eliminates the chore of managing dynamic memory allocation by hand.

```
vector<int> v(3);           // Declare a vector of 3 elements.
  v[0] = 7;
  v[1] = v[0] + 3;
  v[2] = v[0] + v[1];      // v[0] == 7, v[1] == 10, v[2] == 17
```

The STL also includes a large collection of *algorithms* that manipulate the data stored in containers. You can reverse the order of elements in a vector, for example, by using the `reverse` algorithm.

```
reverse(v.begin(), v.end()); // v[0] == 17, v[1] == 10, v[2] == 7
```

There are two important points to notice about this call to `reverse`. First, it is a global function, not a member function. Second, it takes two arguments rather than one: it operates on a *range* of elements, rather than on a container. In this particular case the range happens to be the entire container `v`.

The reason for both of these facts is the same: `reverse`, like other STL algorithms, is decoupled from the STL container classes.

```
double A[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
  reverse(A, A + 6);
  for (int i = 0; i < 6; ++i)
    cout << "A[" << i << "] = " << A[i];
```

This example uses a *range*, just like the example of reversing a vector: the first argument to `reverse` is a pointer to the beginning of the range, and the second argument points one

element past the end of the range. This range is denoted $[A, A + 6)$; the asymmetrical notation is a reminder that the two endpoints are different, that the first is the beginning of the range and the second is *one past* the end of the range.

Website for downloading STL

<http://www.sgi.com/tech/stl/download.htm>

Individual files in STL

algo.h	hash_map.h	numeric	stdexcept	stl_heap.h	stl_slist.h
algorithmbase.h	hash_set	pair.h	stl_algo.h	stl_iterator.h	stl_stack.h
algorithm	hash_set.h	pthread_alloc	stl_algorithmbase.h	stl_iterator_base.h	stl_string_fwd.h
alloc.h	hashtable.h	pthread_alloc.h	stl_alloc.h	stl_list.h	stl_tempbuf.h
bitset	heap.h	queue	stl_bvector.h	stl_map.h	stl_threads.h
bvector.h	iterator	rope	stl_config.h	stl_multimap.h	stl_tree.h
char_traits.h	iterator.h	rope.h	stl_construct.h	stl_multiset.h	stl_uninitialized.h
concept_checks.h	limits	ropeimpl.h	stl_traits_fns.h	stl_numeric.h	stl_vector.h
container_concepts.h	list	sequence_concepts.h	stl_deque.h	stl_pair.h	string
defalloc.h	list.h	set	stl_exception.h	stl_queue.h	tempbuf.h
deque	map	set.h	stl_function.h	stl_range_errors.h	tree.h
deque.h	map.h	slist	stl_hash_fun.h	stl_raw_storage_iter.h	type_traits.h
function.h	memory	slist.h	stl_hash_map.h	stl_relops.h	utility
functional	multimap.h	stack	stl_hash_set.h	stl_rope.h	valarray
hash_map	multiset.h	stack.h	stl_hashtable.h	stl_set.h	vector

Which compilers are supported?

The STL has been tested on these compilers: SGI 7.1 and later, or 7.0 with the `-n32` or `-64` flag; gcc 2.8 or egcs 1.x; Microsoft 5.0 and later. (But see below.) Boris Fomitchev distributes a port for some other compilers.

If you succeed in using the SGI STL with some other compiler, please let us know, and please tell us what modifications (if any) you had to make. We expect that most of the changes will be restricted to the `<stl_config.h>` header.

STL EXAMPLES**Search**

```
const char S1[] = "Hello, world!";
const char S2[] = "world";
const int N1 = sizeof(S1) - 1;
const int N2 = sizeof(S2) - 1;

const char* p = search(S1, S1 + N1, S2, S2 + N2);
printf("Found subsequence \"%s\" at character %d of sequence
\"%s\".\n",
      S2, p - S1, S1);
```

Queue

```
int main() {
    queue<int> Q;
    Q.push(8);
    Q.push(7);
    Q.push(6);
    Q.push(2);

    assert(Q.size() == 4);
    assert(Q.back() == 2);

    assert(Q.front() == 8);
    Q.pop();

    assert(Q.front() == 7);
    Q.pop();

    assert(Q.front() == 6);
    Q.pop();
    assert(Q.front() == 2);
    Q.pop();
    assert(Q.empty());}
```

Doubly linked list

```
list<int> L;
L.push_back(0);
L.push_front(1);
L.insert(++L.begin(), 2);
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
```

Sort

```
int A[] = {1, 4, 2, 8, 5, 7};
const int N = sizeof(A) / sizeof(int);
sort(A, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
// The output is " 1 2 4 5 7 8".
```

Complexity

$O(N \log(N))$ comparisons (both average and worst-case), where N is last - first.

Binary Search

```
int main()
{
    int A[] = { 1, 2, 3, 3, 3, 5, 8 };
    const int N = sizeof(A) / sizeof(int);

    for (int i = 1; i <= 10; ++i) {
        cout << "Searching for " << i << ": "
              << (binary_search(A, A + N, i) ? "present" : "not present") <<
endl;
    }
}
```

The output

```
Searching for 1: present
Searching for 2: present
Searching for 3: present
Searching for 4: not present
Searching for 5: present
Searching for 6: not present
Searching for 7: not present
Searching for 8: present
Searching for 9: not present
Searching for 10: not present
```


APPENDIX D

PC² CONTEST ADMINISTRATION AND TEAM GUIDE



PC² is a dynamic, distributed real-time system designed to manage and control Programming Contests. It includes support for multi-site contests, heterogeneous platform operations including mixed Windows and Unix in a single contest, and dynamic real-time updates of contest status and standings to all sites. Here we describe the steps required to install, configure, and run a contest using PC². Further information on PC², including how to obtain a copy of the system, can be found at <http://www.ecs.csus.edu/pc2>.

Programming Contest Judge Software - PC²

PC² operates using a client-server architecture. Each site in a contest runs a single PC² *server*, and also runs multiple PC² *clients* which communicate with the site server. Logging into a client using one of several different types of PC² accounts (Administrator, Team, Judge, or Scoreboard) enables that client to perform common contest operations associated with the account type, such as contest configuration and control (Administrator), submitting contestant programs (Team), judging submissions (Judge), and maintaining the current contest standings (Scoreboard).

PC² clients communicate only with the server at their site, regardless of the number of sites in the contest. In a multi-site contest, site servers communicate not only with their own clients but also with other site servers, in order to keep track of global contest state. The following communication requirements must therefore be met in order to run a contest using PC²: (1) a machine running a PC² server must be able to communicate via TCP/IP with every machine running a PC² client at its site; and (2) in a multi-site contest, every machine running a PC² server must be able to communicate via TCP/IP with the machines running PC² servers at every other site^[1]. In particular, there must not be any firewalls which prohibit these communication paths; the system will not operate if this communication is blocked. It is not necessary for client machines to be able to contact machines at other sites.

Each PC² module (server or client) reads one or more .ini initialization files when it starts; these files are used to configure the module at startup. The client module also tailors its configuration when a user (Team, Judge, etc.) logs in. In a typical PC² contest configuration, each Team, Judge, etc. uses a separate physical machine, and each of these machines runs exactly one client module. It is possible to have multiple clients running on the same physical machine, for example by having different users logging in to different accounts on a shared machine. In this case, each user (Team, Judge, etc.) will be executing their own Java Virtual Machine (JVM), and must have their own separate directory structure including their own separate copy of the PC² initialization files in their account.

PC² Setup for Administrator

For those people who hate to read manuals and would rather take a chance with a shortcut list, here is a *very terse* summary of the steps necessary to install PC² and get it running. Please note that this is provided as a convenience for geniuses (or gluttons for punishment). The remainder of the manual was written to help everyone else. If you have

* For further information about PC² please check : <http://www.ecs.csus.edu/pc2>

problems after following this list, *please read the rest of the manual* before sending us a request for help.

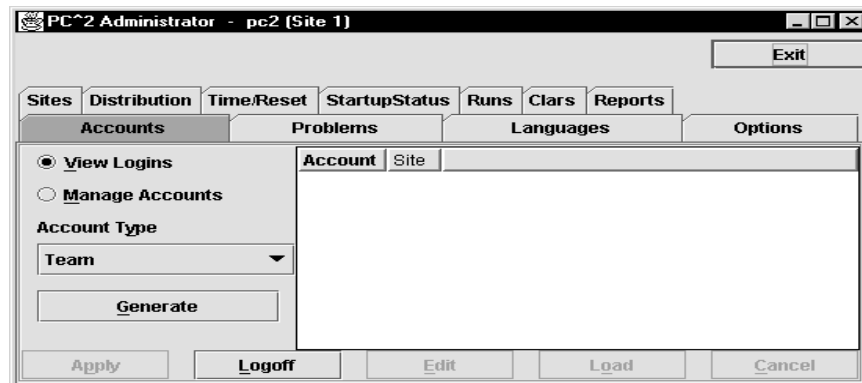
- ❑ Install Java (version 1.3.1 or greater) ;
- ❑ Install PC² by unzipping the PC² distribution to the PC² installation directory;
- ❑ Add the Java *bin* directory and the PC² installation directory to the PATH;
- ❑ Add java/lib, and the PC² installation directory to the CLASSPATH;
- ❑ Modify the *sitelist.ini* file as necessary to specify each site server name.
- ❑ Edit the *pc2v8.ini* file to point servers and clients to the server IP:port and to specify the appropriate site server name; put the modified .ini file on every server and client machine;

```
# sample pc2v8.ini file for site named Site

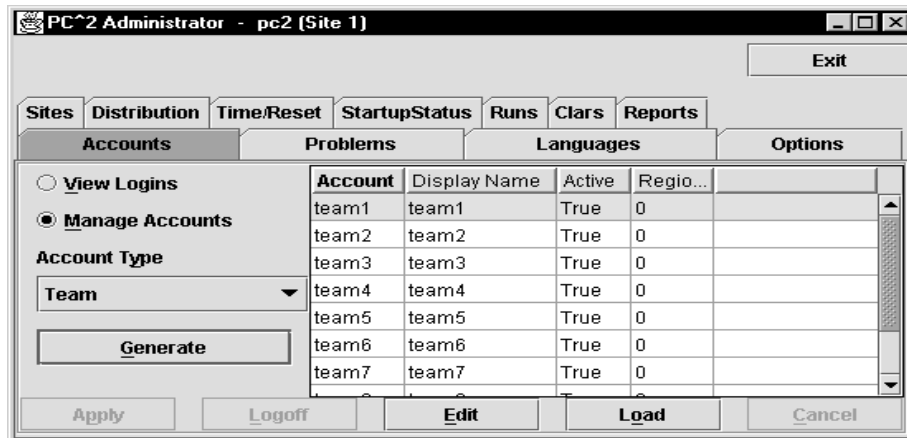
[client]
# tell the client what site it belongs to
# and where to find its server (IP and port)
site=Site1
server=192.168.1.117:50002

[server]
# tell the server which site it is serving
site=Site1
```

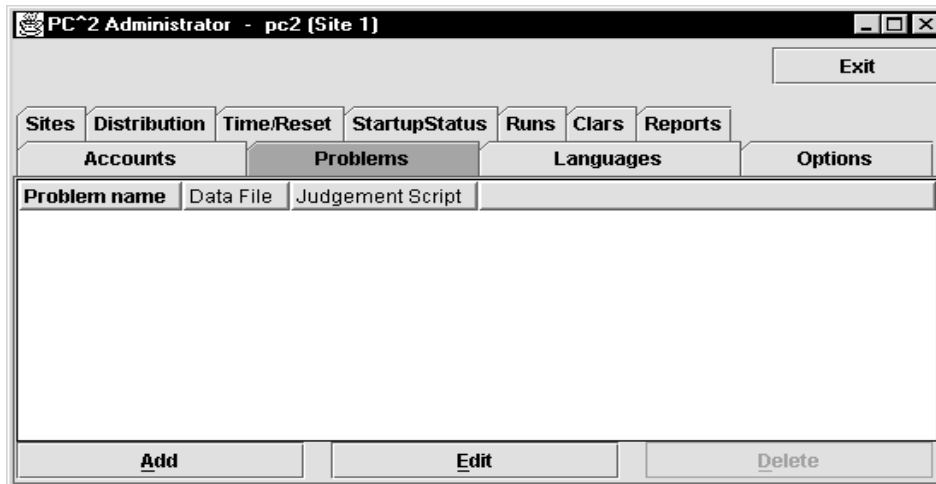
- ❑ Start a PC² server using the command `pc2server` and answer the prompted question.
- ❑ Start a PC² Admin client using the command `pc2admin` and login using the name `root` and password `root`.



- ❑ Configure at least the following contest items via the Admin:
- ❑ Accounts (generate the necessary accounts);

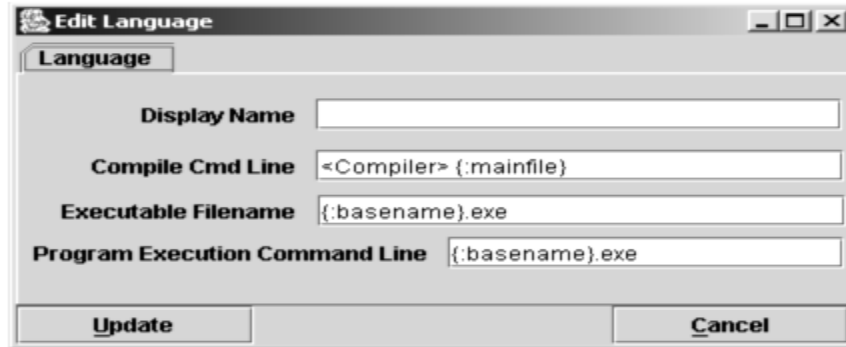


- ❑ Problems (create one or more contest problems, specifying the problem input data file if there is one);



- ❑ Languages (create contest languages, specifying the language name, compile command line, executable filename, and execution command line).

❑ Language Parameter for Turbo C/C++ : `tcc -Ic:\tc3\bin -Lc:\tc3\lib { :mainfile}`

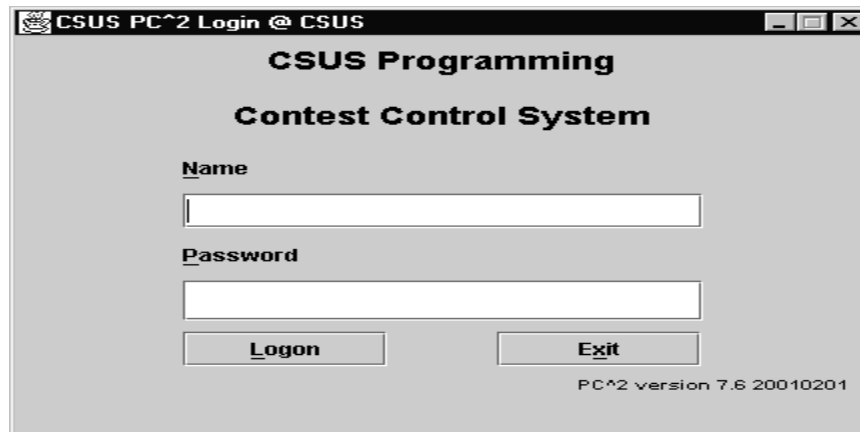


- Press the Start Contest button on the Admin Time/Reset tab;
- Start a PC² client on each Team and Judge machine and log in using the Admin-created accounts and passwords.
- Start a PC² client on the Scoreboard machine and log in using the board1 Scoreboard account/password; arrange for the scoreboard-generated HTML files to be accessible to users browsers.



PC² Team Guide (For Contestants)

This guide is intended to familiarize you with the process of submitting programs to Contest Judges using the PC² (“P-C-Squared”) **P**rogramming **C**ontest **C**ontrol system. Starting PC² will bring up the PC² **login screen**, shown below:



To login to PC², click once on the **Name** box on the login screen, enter your assigned team ID, press the TAB key or click on the **Password** box, then enter your assigned password. Your team ID will be of the form **teamxx**, where xx is your assigned team number (for example, “**team3**” or “**team12**”). After entering your team name and password, click on the **Logon** button.

Submitting a Program to the Judges

Once the system accepts your login, you will be at the PC² **Main Menu** screen, shown below. Note that the team ID (“team1” in this case) and the team’s site location (“CSUS” in this case) are displayed in the title bar.



Clicking on the **SUBMIT** tab near the top of the screen displays the **Submit Run** screen, which is shown above.

Clicking in the **Problem** field will display a list of the contest problems; choose the problem for which you wish to submit a program (“run”) to the Judges (in the example, a problem named “Bowling” has been chosen).

Clicking in the **Language** field will display a list of the programming languages allowed in the contest; choose the language used by the program that you wish to submit to the Judges (in the example, “Java” has been chosen).

To submit a program to the Judges, you must specify the name of the file containing your **main program**. Click on the **Select** button to invoke the “File Dialog” which lets you locate and select your main file. The Dialog lets you automatically navigate to the correct path and file location (in the example, the main program file “C:\work\bowling.java” has been selected).

If your program consists of more than one file, you must enter the additional file names in the **Additional Files** box. Click the **Add** button to invoke the dialog which lets you locate and select your additional files; select a file in the box and click the **Remove** button to remove a previously-selected file.

Important: you *must* specify the name of your *main program* file in the **Main File** field, *not* in the **Additional Files** box! Select only *source code* files for submission to the Judges. Do not submit data files or executable files.

PC² Uninstall

To uninstall PC² it is only necessary to undo the above installation steps; that is, remove the \$PC2HOME directory and its contents and restore the system environment variables to their former values. PC² itself does not make any changes to any machine locations outside those listed above either during installation or execution. In particular, for example, it makes no entries in the registry in a Windows environment, nor does it copy any files to locations outside the installation directories in any environment.

APPENDIX E

IMPORTANT WEBSITES/ FOR ACM/ICPC PROGRAMMERS



Now a days the world is open for learners and the Internet makes the world inside our room. There are many websites now on the Internet for the beginners as well as programmers. Some sites are for contest and tutorial, some for books, some sites are for programming language. Mathematics is one of the essential for programming and there are a lot of mathematical site in the net. The pioneers also publish their ideas and advice on the Internet for us. In this page you will find the link of those pages.^[10]

Online Judges And Tutorial Sites

<http://online-judge.uva.es/> and <http://cii-judge.baylor.edu/>

University of Valladolid 24-hour Online Judge and Official ACM/ ICPC Live archive contains all problems of ACM Regionals.

<http://acm.timus.ru/>

24-hour Online Judge hosted by Ural State University of Russia. This site contain the problems from many Russian contest. This can be very useful site for students to practice. From this site we can understand how these Russian Problems differ from non-Russian problems.

<http://plg.uwaterloo.ca/~acm00/>

This is the contest page of University of Waterloo. This university currently leads the contest arena of the World. They arrange programming contest almost in every alternate term and problems of these contest set standard for many other universities.

<http://oldweb.uwp.edu/academic/mathematics/usaco/> and <http://ace.delos.com/>

Online training site for contests which guides beginners with a step by step problem solution.

<http://www.acmsolver.org/>

Another training site for 24-hour online programming contest developed by "Ahmed Shamsul Arefin" mainly for novice programmers now linked with world's major online judges.

<http://www.acm.inf.ethz.ch/ProblemSetArchive.html>

This site is the best resource for the past ACM Regionals and World Finals problem set. It has also solutions and judge data for many contests, which will help us for practice.

<http://acm.uva.es/problemset/replies.html>

This page contains the information about the meaning of different judge replies. This is also a link page of the site acm.uva.es.

<http://contest.uvarov.ru/>

This is the POTM master's home page. POTM Contest means Programmer of the Month Contest. The problem set of this contest is different then the conventional programming contest. The organizer gives more stress on Optimization than number of solution.

<http://www.acmbeginner.tk>

Is it very difficult to disclose oneself as a good programmer? To solve the problems, the difficulty that you face in the 24-hour online judge and the help and guideline that found to overcome them is integrated in this site "ACMBEGINNER" by M H Rasel.

<http://groups.yahoo.com/group/icpc-l>
http://groups.yahoo.com/group/acm_solver

You talk about algorithms, data structures and mathematical concepts. The group is made for teams, ex-teams and everybody interested in on-line programming. Coaches and teachers are very welcome.

Site of Pioneers !

Newcomers advice, by Shahriar Manzoor

One of the greatest programmer/ problemsetter of acm.uva.es as well as a renowned progeammer of Bangladesh, advices the newcomers. He is now the judge of UVa OJ site and WF. He is also the elite panel problem setter of the site acm.uva.es. you can visit his publication by the using the URL <http://www.acm.org/crossroads/xrds7-5/>

World of Steven, by Steven Halim

Steven Halim published his site for the problem solvers including the problemset tricks tutorials with the difficulty level. In his site he has also categorized problems. This site contains almost all hints of the problems that he has solved. You can visit the site using the link <http://www.comp.nus.edu.sg/~stevenha/programming/acmoj.html>

Reuber Guerra Duarte

Reuber Guerra Duarte developed this site with some solutions (the real source code) for some problems that already solved by him. Please don't directly send his solutions to judge. <http://www.dcc.ufmg.br/~reuber/solutions/index.html>

Ed's Programming Contest Problem Archive by Ed-Karrels

Ed-karrel one of the greatest problem setter publish this site with the solutions to many problems. visit the site with the URL http://www.karrels.org/Ed/ACM/prob_index.html

Mark Dettinger

A living legend of programming contest. He was a main contest architect of the University of ULM, Germany. Now this responsibility lies on another person Water Guttman. If you visit the site you will find many useful link as well as many interesting incidents of his life and onmces again you will realize again that the boy gets eleven in math out of hundred can also be genius. URL <http://www.informatik.uni-ulm.de/pm/mitarbeiter/mark/>

Books**Introduction to Algorithm (CLR)**

Introduction to Algorithms (MIT Electrical Engineering and Computer Science Series) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. Now they already publish the 2nd Edition + Stein as the 4th author. This book is a-must-have to be a good programmer.

Algorithm Design Manual

The Algorithm Design Manual by Steven S. Skiena (+ book website). Also a-must-have for every good programmer.

The Art of Computer Programming

The Art of Computer Programming, Volumes 1-3 Boxed Set by Donald E. Knuth. Hard to understand for beginner but worth to read

Concrete Mathematics

Concrete Mathematics, Second Edition + by Ronald L. Graham, Donald E. Knuth, Oren Patashnik. This book contain mathematical problem and solutions, but This is very hard to understand. Though it is hard, it is very useful for programming contest. Several formulation techniques discussed here.

Number Theory

Number Theory by S G Telang is a book, consists of lot of knowledge about the number theory. It is better idea to read this book before reading the book Concrete Mathematics. This is easy to understand.

Discrete Mathematics And Its Application

Discrete Mathematics And Its Application by Kenneth H. Rosen consists of a lot of practical help on integer matrices, mathematical reasoning, counting, graph tree problems.

Data Structure Using C and C++

Data structure using C and C++ by ,Yedidyah Langsam, Moshe J. Augenstein Aaron M. Tenenbaum. Second Edition. This book contains basic problem related to Data Structures with C coding.

Programming Challenges

Programming Challenges, by Steven Skiena Miguel Revilla. We dont want to say anything about this two legend. All the programmers are familiar with this two ultimate Programming Contest Legends. Use this book to learn what to do for the programming contest.

Art of Programming Contest

Well, Art of Programming Contest is this one, by Ahmed Shamsul Arefin.

Programming Language Guides

<http://www.sgi.com/tech/stl/>

In this site you will find the STL documentation. The STL is very important for programming. It is efficient and make code simple and sort. To use this the C++ template knowledge is necessary. The beginner can overlook the STL at beginning but is is needed in future.

www.cplusplus.com / www.cprogramming.com

Great sites for the C/C++ programmers. These sites contains documents, References, sources codes with electronic forum where you can consult with great programmers.

Mathematics Site

mathworld.wolfram.com

MathWorld is a comprehensive and interactive mathematics encyclopedia intended for students, educators, math enthusiasts, and researchers. Like the vibrant and constantly evolving discipline of mathematics, this site is continuously updated to include new material and incorporate new discoveries.

<http://mathforum.org/>

The Math Forum is a leading center for mathematics and mathematics education on the Internet. Its mission is to provide resources, materials, activities, person-to-person interactions, and educational products and services that enrich and support teaching and learning in an increasingly technological world.

<http://www.math.com/>

Math.com is dedicated to providing revolutionary ways for students, parents, teachers, and everyone to learn math. Combining educationally sound principles with proprietary technology, Math.com offers a unique experience that quickly guides the user to the solutions they need and the products they want.

<http://www.math-net.org/>

Math-Net intends to coordinate the electronic information and communication activities of the global mathematical community with the aim to enhance the free-flow of information within the community.

REFERENCES

- [1] “HOW TO DO BETTER IN 24 HOURS ONLINE JUDGES”, Anupam Bhattacharjee, , Published by ACMSolver.org, 2003.
- [2] “World of Seven”, METHODS TO SOLVE VALLADOLID ONLINE JUDGE PROBLEMS, Steven Halim, National University of Singapore, <http://www.comp.nus.edu.sg/~stevenha/>
- [3] ACMSolver.org, ACM/ICPC Programming Contest Tutorial Website for Valladolid OJ by Ahmed Shamsul Arefin <http://www.acmsolver.org>
- [4] “Theoretical Computer Science Cheat Sheet”, Steve Seiden, <http://www.csc.lsu.edu/~seiden>
- [5] “Teach Yourself C in 21 Days”, Peter Aitken, Bradley L. Jones, website: <http://www.mcp.com/info/0-672/0-672-31069-4/>
- [6] “Common Mistakes in Online and Real-time Contests”, Shahriar Manzoor, ACMCross Roads Student Magazine, www.acm.org/crossroads/xrds7-5/contests.html
- [7] Verhoeff, T. Guidelines for Producing a Programming-Contest Problem Set: <http://www.wpa.win.tue.nl/wstomv/publications/guidelines.html>
- [8] Ernst, F., J. Moelands, and S. Pieterse. Teamwork in Programming Contests: 3 * 1 = 4, Crossroads, 3.2. <http://www.acm.org/crossroads/xrds3-2/progcon.html>
- [9] STL (Standard Template Library), Silicon Graphics, Inc, <http://www.sgi.com/tech/stl/>
- [10] ACMBeginner.tk, ACM Valladolid Online Judge (OJ) Tools, Tips and Tutorial by M H Rasel, <http://www.acmbeginner.tk/>
- [11] Brian W. Kernighan- “Programming in C: A Tutorial” <http://www.lysator.liu.se/c/bwk-tutor.html>
- [12] Rob Kolstad , USACO Training Gateway, <http://ace.delos.com/>

INDEX

- ADHOC problems, 16
Adjacency List, 139
Adjacency Matrix, 139
Algorithm, 19
Array, 74
Base Number Conversion, 91
Bellman-Ford Algorithm, 164
Big Integer, 92
Big Mod, 91, 195
Binary Search, 113
Binary Search Tree, 113
Breadth-first search, 144
BRUTE FORCE METHOD, 85
Bubble Sort, 107
Carmichael Number, 93
Collision Handling, 115
Combinations, 202
COMPILATION ERROR (CE), 15
Connected Fields, 158
Connectedness, 141
Convex Hull, 173
Counting Change, 131
Counting Combinations, 94
Counting Sort, 111
DATA STRUCTURES, 72
debugging, 15
Decimal To Roman, 198
Decomposition, 89
Depth First with Iterative Deepening, 150
Depth-first search, 147
Dictionary, 114
Dijkstra Algorithm, 163
Directed Acyclic Graph, 165
Directed Graph, 137
Divide and Conquer, 88
Divisors and Divisibility, 95
DYNAMIC PROGRAMMING, 121
Earth coordinate system, 172
Edge List, 138
Edit Distance, 127
Euclid's Algorithm, 190
Euler Cycle, 170
Euler Path, 170
Exponentiation, 96
Factorial, 97
Fermat Algorithm, 104
Fermat Little Test:, 104
Fibonacci, 97
Floyd Warshall, 165
GCD, 191
Graph Transpose, 170
GRAPHS, 134
Greatest Common Divisor (GCD), 99
GREEDY ALGORITHMS, 117
Hash Table variations, 116
Infix to Postfix conversion, 100
Informed Search, 154
Integer to any base, 197
Josephus Problem, 200
Judges, 26
Kruskal's algorithm, 161
LCM, 191
LCS, 205
Linear-time Sorting, 111
Linked List, 76
Longest Common Subsequence, 125
Longest Inc/Dec-reasing Subsequence (LIS/LDS), 128
Lowest Common Multiple (LCM), 99
Matrix Chain Multiplication, 123
Matrix Chain Multiplication Problem, 123
Maximum Interval Sum, 132
MCM, 209
MiniMax Distance, 167
Minimum spanning trees, 160
Minimum Spanning Trees, 159
Multiple input programs, 83
Number Conversions, 196
optimal sub-structure, 124
Optimizing, 89
order of growth, 22
Other Dynamic Programming Algorithms, 133
PC², 235
Postfix Calculator, 100
Pre-Computation, 89
Prime Factors, 101
Prime Numbers, 102
prime testing, 102
Prim's algorithm, 162
Queue, 79
Quick Sort, 108
Radix Sort, 111
Recursion, 86
SEARCHING, 113
Sieve of Eratosthenes, 105
SORTING, 106
sorting algorithms, 106
Stack, 78
STL, 231
Strongly Connected Components, 171
Subgraphs, 142
Symmetries, 90
Topological Sort, 119, 171
Transitive Hull, 166
Uninformed Search, 143
Valladolid OJ, 16
Variations on Binary Trees, 114
Zero-One Knapsack, 130