3.2 Divide and Conquer

Divide and Conquer (abbreviated as D&C) is a problem solving paradigm where we try to make a problem *simpler* by 'dividing' it into smaller parts and 'conquering' them. The steps:

1. Divide the original problem into *sub*-problems – usually by half or nearly half,
2. Find (sub-)solutions for each of these sub-problems – which are now easier,
3. If needed, combine the sub-solutions to produce a complete solution for the main problem.

We have seen this D&C paradigm in previous chapters in this book: various sorting algorithms like Quick Sort, Merge Sort, Heap Sort, and Binary Search in Section 2.2.1 utilize this paradigm. The way data is organized in Binary Search Tree, Heap, and Segment Tree in Section 2.2.2 & 2.3.3, also has the spirit of Divide & Conquer.

3.2.1 Interesting Usages of Binary Search

In this section, we discuss the spirit of D&C paradigm around a well-known Binary Search algorithm. We still classify Binary Search as 'Divide' and Conquer paradigm although some references (e.g. [14]) suggest that it should be classified as 'Decrease (by-half)' and Conquer as it does not actually 'combine' the result. We highlight this algorithm because many contestants know it, but not many are aware that it can be used in other ways than its ordinary usage.

Binary Search: The Ordinary Usage

Recall: The *ordinary* usage of Binary Search is for searching an item in a *static sorted array*. We check the middle portion of the sorted array if it is what we are looking for. If it is or there is no more item to search, we stop. Otherwise, we decide whether the answer is on the left or right portion of the sorted array. As the size of search space is halved (binary) after each query, the complexity of this algorithm is $O(\log n)$. In Section 2.2.1, we have seen that this algorithm has library routines, e.g. C++ STL <`algorithm`>: `lower_bound`, Java `Collections.binarySearch`.

This is *not* the only way to use and apply binary search. The pre-requisite to run binary search algorithm – a *static sorted array (or vector)* – can also be found in other uncommon data structure, as in the root-to-leaf path on a structured tree below.

Binary Search on Uncommon Data Structure (Thailand ICPC National Contest 2009)

Problem in short: given a weighted (family) tree of N vertices up to $N \leq 80K$ with a special trait: *vertex values are increasing from root to leaves*. Find the ancestor vertex closest to root from a starting vertex v that has weight at least P. There are up to $Q \leq 20K$ such queries.

Naïve solution is to do this linear $O(N)$ scan per query: Start from a given vertex v, then move up the family tree until we hit the first ancestor with value $< P$. In overall, as there are Q queries, this approach runs in $O(QN)$ and will get TLE as $N \leq 80K$ and $Q \leq 20K$.

A better solution is to store all the $20K$ queries first. Then traverse the family tree *just once* from root using $O(N)$ Depth First Search (DFS) algorithm (Section 4.2). Search for some non-existent value so that DFS explores the *entire* tree, building a partial root-to-leaf *sorted* array as it goes – this is because the vertices in the root-to-leaf path have increasing weights. Then, for each

vertex asked in query, perform a $O(\log N)$ **binary search**, i.e. `lower_bound`, along the current path from root to that vertex to get ancestor closest to root with weight at least P. Finally, do an $O(Q)$ post-processing to output the results. The overall time complexity of this approach is $O(Q \log N)$, which is now manageable.

Bisection Method

What we have seen so far are the usage of binary search in finding items in a static sorted array. However, the binary search **principle** can also be used to find the root of a function that may be difficult to compute mathematically.

Sample problem: You want to buy a car using loan and want to pay in monthly installments of d dollars for m months. Suppose the value of the car is originally v dollars and the bank charges $i\%$ interest rate for every unpaid money at the end of each month. What is the amount of money d that you must pay per month (rounded to 2 digits after decimal point)?

Suppose $d = 576$, $m = 2$, $v = 1000$, and $i = 10\%$. After one month, your loan becomes $1000 \times (1.1) - 576 = 524$. After two months, your loan becomes $524 \times (1.1) - 576 \approx 0$.

But if we are only given $m = 2$, $v = 1000$, and $i = 10\%$, how to determine that $d = 576$? In another words, find the root d such that loan payment function $f(d, 2, 1000, 10) \approx 0$. The *easy* way is to run the bisection method[2]. We pick a reasonable range as the starting point. In this case, we want to find d within range $[a \dots b]$. $a = 1$ as we have to pay something (at least $d = 1$ dollar). $b = (1 + i) \times v$ as the earliest we can complete the payment is $m = 1$, if we pay exactly $(1 + i\%) \times v = (1 \times 10) \times 1000 = 1100$ dollars after one month. Then, we apply bisection method to obtain d as follows:

- If we pay $d = (a + b)/2 = (1 + 1100)/2 = 550.5$ dollars per month, then we undershoot by 53.95 dollars after two months, so we know that we must *increase* the monthly payment.
- If we pay $d = (550.5 + 1100)/2 = 825.25$ dollars per month, then we overshoot by 523.025 dollars after two months, so we know that we must *decrease* the payment.
- If we pay $d = (550.5 + 825.25)/2 = 687.875$ dollars per month, then we overshoot by 234.5375 dollars after two months, so we know that we must *decrease* the payment.
- ... **few** logarithmic iterations after, to be precise, after $O(\log_2((b - a)/\epsilon))$ iterations where ϵ is the amount of error that we can tolerate.
- Finally, if we pay $d = 576.190476...$ dollars per month, then we manage to finish the payment (the error is now less than ϵ) after two months, so we know that $d = 576$ is the answer.

For bisection method to work[3], we must ensure that the function values of the two extreme points in the initial Real range $[a \dots b]$, i.e. $f(a)$ and $f(b)$ have opposite signs (true in the problem above). Bisection method in this example only takes $\log_2 1099/\epsilon$ tries. Using a small ϵ = 1e-9, this is just $\approx$ 40 tries. Even if we use an even smaller ϵ = 1e-15, we still just need $\approx$ 60 tries[4]. Bisection method is more efficient compared to linearly trying each possible value of $d = [1..1100]/\epsilon$.

[2]We use the term 'binary search principle' as a divide and conquer technique that involve halving the range of possible answer. We use the term 'binary search algorithm' (finding index of certain item in sorted array) and 'bisection method' (finding root of a function) as instances of this principle.

[3]Note that the requirement of bisection method (which uses binary search principle) is slightly different from the more well-known binary search algorithm which needs a sorted array.

[4]Thus some competitive programmers choose to do 'loop 100 times' which guarantees termination instead of testing whether the error is now less than ϵ as some floating point errors may lead to endless loop.

Binary Search the Answer

Binary Search 'the Answer' is another problem solving strategy that can be quite powerful. This strategy is shown using UVa 714 - Copying Books below.

In this problem, you are given m books numbered $1, 2, \ldots, m$ that may have different number of pages $(p_1, p_2, \ldots, p_m)$. You want to make one copy of each of them. Your task is to divide these books among k scribes, $k \leq m$. Each book can be assigned to a single scriber only, and every scriber must get a *continuous sequence* of books. That means, there exists an increasing succession of numbers $0 = b_0 < b_1 < b_2, \ldots < b_{k-1} \leq b_k = m$ such that i-th scriber gets a sequence of books with numbers between $b_{i-1} + 1$ and b_i. The time needed to make a copy of all the books is determined by the scriber who was assigned the most work. The task is to minimize the maximum number of pages assigned to a single scriber.

There exist Dynamic Programming solution for this problem, but this problem can already be solved by guessing the answer in binary search fashion! Suppose $m = 9$, $k = 3$ and $p_1, p_2, \ldots, p_9$ are 100, 200, 300, 400, 500, 600, 700, 800, and 900, respectively.

If we guess $ans = 1000$, then the problem becomes 'simpler', i.e. if the scriber with the most work can only copy up to 1000 pages, can this problem be solved? The answer is 'no'. We can greedily assign the jobs as: {100, 200, 300, 400} for scribe 1, {500} for scribe 2, {600} for scribe 3, but we have 3 books {700, 800, 900} unassigned. The answer must be at least 1000.

If we guess $ans = 2000$, then we greedily assign the jobs as: {100, 200, 300, 400, 500} for scribe 1, {600, 700} for scribe 2, and {800, 900} for scribe 3. We still have some slacks, i.e. scribe 1, 2, and 3 still have {500, 700, 300} unused potential. The answer must be at most 2000.

This ans is binary-searchable between $lo = 1$ (1 page) and $hi = p_1 + p_2 + \ldots + p_m$ (all pages).

Programming Exercises for problems solvable using Divide and Conquer:

1. UVa 679 - Dropping Balls (like Binary Search)
2. UVa 714 - Copying Books (Binary Search + Greedy)
3. UVa 957 - Popes (Complete Search + Binary Search: `upper_bound`)
4. UVa 10077 - The Stern-Brocot Number System (Binary Search)
5. UVa 10341 - Solve It (Bisection Method)
6. UVa 10369 - Arctic Networks (solvable using Kruskal's for MST too – Section 4.4)
7. UVa 10474 - Where is the Marble?
8. UVa 10611 - Playboy Chimp (Binary Search)
9. UVa 11262 - Weird Fence (Binary Search + Bipartite Matching, see Section 4.9.3)
10. LA 2565 - Calling Extraterrestrial Intelligence Again (Kanazawa02) (BSearch + Math)
11. LA 2949 - Elevator Stopping Plan (Guangzhou03) (Binary Search + Greedy)
12. LA 3795 - Against Mammoths (Tehran06)
13. LA 4445 - A Careful Approach (Complete Search + Bisection Method + Greedy)
14. IOI 2006 - Joining Paths
15. IOI 2009 - Mecho (Binary Search + BFS)
16. My Ancestor - Thailand ICPC National Contest 2009 (Problem set by: Felix Halim)
17. See Section 7.5 for 'Divide & Conquer for Geometry Problems'

3.3 Greedy

An algorithm is said to be greedy if it makes locally optimal choice at each step with the hope of finding the optimal solution. For some cases, greedy works - the solution code becomes short and runs efficiently. But for *many* others, it does not. As discussed in [4], a problem must exhibit two things in order for a greedy algorithm to work for it:

1. It has optimal sub-structures.

   Optimal solution to the problem contains optimal solutions to the sub-problems.

2. It has a greedy property (remark: hard to prove its correctness!).

   If we make a choice that seems best at the moment and solve the remaining subproblems later, we still reach optimal solution. We never have to reconsider our previous choices.

3.3.1 Classical Example

Suppose we have a large number of coins with different denominations, i.e. 25, 10, 5, and 1 cents. We want to make change with the *least number of coins used*. The following greedy algorithm works for this set of denominations of this problem: Keep using the largest denomination of coin which is not greater than the remaining amount to be made. Example: if the denominations are {25, 10, 5, 1} cents and we want to make a change of 42 cents, we can do: 42-25 = 17 → 17-10 = 7 → 7-5 = 2 → 2-1 = 1 → 1-1 = 0, of total 5 coins. This is optimal.

The coin changing example above has the two ingredients for a successful greedy algorithm:

1. It has optimal sub-structures.
   We have seen that in the original problem to make 42 cents, we have to use 25+10+5+1+1. This is an optimal 5 coins solution to the original problem!
   Now, the optimal solutions to its sub-problems are contained in this 5 coins solution, i.e.
   a. To make 17 cents, we have to use 10+5+1+1 (4 coins),
   b. To make 7 cents, we have to use 5+1+1 (3 coins), etc

2. It has a greedy property.
   Given every amount V, we greedily subtract it with the largest denomination of coin which is not greater than this amount V. It can be proven (not shown here for brevity) that using other strategy than this will not lead to optimal solution.

However, this greedy algorithm does *not* work for *all* sets of coin denominations, e.g. {1, 3, 4} cents. To make 6 cents with that set, a greedy algorithm would choose 3 coins {4, 1, 1} instead of the optimal solution using 2 coins {3, 3}. This problem is revisited later in Section 3.4.2.

There are many other classical examples of greedy algorithms in algorithm textbooks, for example: Kruskal's for Minimum Spanning Tree (MST) problem – Section 4.4, Dijkstra's for Single-Source Shortest Paths (SSSP) problem – Section 4.5, Greedy Activity Selection Problem [4], Huffman Codes [4], etc.

3.3.2 Non Classical Example

Today's contest problems usually do not ask for solution of trivial and classical greedy problems. Instead, we are presented with novel ones that requires creativity, like the one shown below.

UVa 410 - Station Balance

Given $1 \leq C \leq 5$ chambers which can store 0, 1, or 2 specimens, $1 \leq S \leq 2C$ specimens, and M: a list of mass of the S specimens, determine in which chamber we should store each specimen in order to minimize IMBALANCE. See Figure 3.3 for visual explanation.

A = $(\sum_{j=1}^{S} M_j)/C$, i.e. A is the average of all mass over C chambers.

IMBALANCE = $\sum_{i=1}^{C} |X_i - A|$, i.e. sum of differences between the mass in each chamber w.r.t A.

where X_i is the total mass of specimens in chamber i.

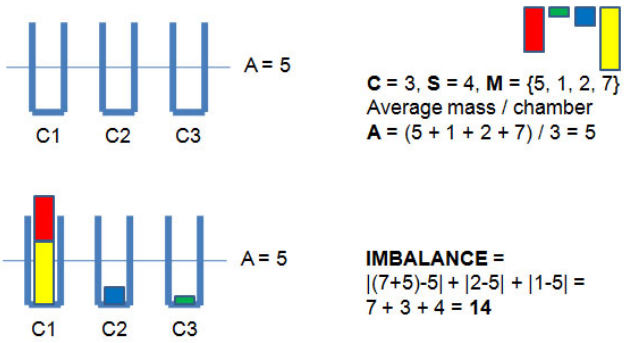

Figure 3.3: Visualization of UVa 410 - Station Balance

This problem can be solved using a greedy algorithm. But first, we have to make several observations. If there exists an empty chamber, at least one chamber with 2 specimens must be moved to this empty chamber! Otherwise the empty chambers contribute too much to IMBALANCE! See Figure 3.4.

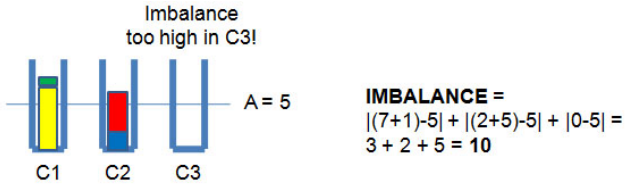

Figure 3.4: UVa 410 - Observation 1

Next observation: If $S > C$, then $S - C$ specimens must be paired with one other specimen already in some chambers. The Pigeonhole principle! See Figure 3.5.

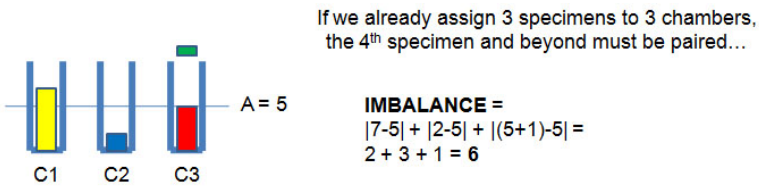

Figure 3.5: UVa 410 - Observation 2

Now, the key insight that can simplify the problem is this: If $S < 2C$, add dummy $2C - S$ specimens with mass 0. For example, $C = 3$, $S = 4$, $M = \{5, 1, 2, 7\} \rightarrow C = 3, S = 6, M = \{5, 1, 2, 7, 0, 0\}$. Then, sort these specimens based on their mass such that $M_1 \leq M_2 \leq \ldots \leq M_{2C-1} \leq M_{2C}$. In this example, $M = \{5, 1, 2, 7, 0, 0\} \rightarrow \{0, 0, 1, 2, 5, 7\}$.

By adding dummy specimens and then sorting them, a greedy strategy 'appears'. We can now:
Pair the specimens with masses $M_1 \& M_{2C}$ and put them in chamber 1, then
Pair the specimens with masses $M_2 \& M_{2C-1}$ and put them in chamber 2, and so on ...
This greedy algorithm – known as 'Load Balancing' – works! See Figure 3.6.

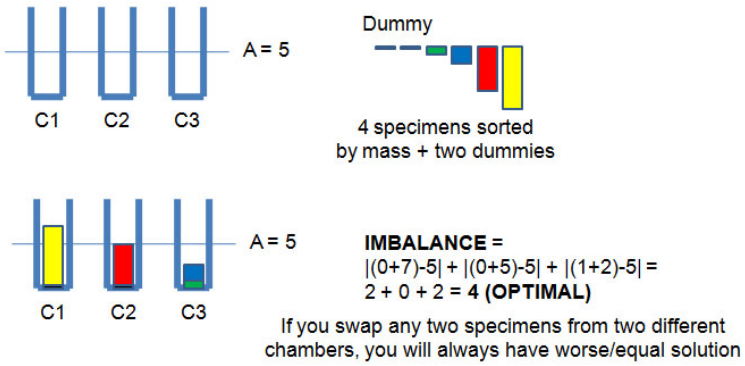

Figure 3.6: UVa 410 - Greedy Solution

To come up with this way of thinking is hard to teach but can be gained from experience! One tip from this example: If no obvious greedy strategy seen, try to sort the data first or introduce some tweaks and see if a greedy strategy emerges.

3.3.3 Remarks About Greedy Algorithm in Programming Contests

Using Greedy solutions in programming contests is usually risky. A greedy solution normally will not encounter TLE response, as it is lightweight, but tends to get WA response. Proving that a certain problem has optimal sub-structure and greedy property in contest time may be time consuming, so a competitive programmer usually do this:

He will look at the input size. If it is 'small enough' for the time complexity of either Complete Search or Dynamic Programming (see Section 3.4), he will use one of these approaches as both will ensure correct answer. He will *only* use Greedy solution if he knows for sure that the input size given in the problem is too large for his best Complete Search or DP solution.

Having said that, it is quite true that many problem setters nowadays set the input size of such can-use-greedy-algorithm-or-not-problems to be in some reasonable range so contestants *cannot* use the input size to quickly determine the required algorithm!

Programming Exercises solvable using Greedy (hints omitted):

1. UVa 410 - Station Balance (elaborated in this section)
2. UVa 10020 - Minimal Coverage
3. UVa 10340 - All in All
4. UVa 10440 - Ferry Loading II
5. UVa 10670 - Work Reduction
6. UVa 10763 - Foreign Exchange
7. UVa 11054 - Wine Trading in Gergovia
8. UVa 11292 - Dragon of Loowater
9. UVa 11369 - Shopaholic

In this section, we want to highlight another problem solving trick called: *Decomposition*!

While there are only 'few' basic algorithms used in contest problems (most of them are covered in this book), harder problems may require a *combination* of two (or more) algorithms for their solution. For such problems, try to decompose parts of the problems so that you can solve the different parts independently. We illustrate this decomposition technique using a recent top-level programming problems that combines *three* problem solving paradigms that we have just learned: Complete Search, Divide & Conquer, and Greedy!

ACM ICPC World Final 2009 - Problem A - A Careful Approach

You are given a scenario of airplane landings. There are $2 \leq n \leq 8$ airplanes in the scenario. Each airplane has a time window during which it can safely land. This time window is specified by two integers a_i, b_i, which give the beginning and end of a closed interval $[a_i, b_i]$ during which the i-th plane can land safely. The numbers a_i and b_i are specified in minutes and satisfy $0 \leq a_i \leq b_i \leq 1440$. In this problem, plane landing time is negligible. Then, your task is to:

1. Compute an **order for landing all airplanes** that respects these time windows.
HINT: order = permutation = Complete Search?
2. Furthermore, the airplane landings should be stretched out **as much as possible** so that the minimum achievable time gap between successive landings is as large as possible. For example, if three airplanes land at 10:00am, 10:05am, and 10:15am, then the smallest gap is five minutes, which occurs between the first two airplanes. Not all gaps have to be the same, but the smallest gap should be as large as possible!
HINT: Is this similar to 'greedy activity selection' problem [4]?
3. Print the answer split into minutes and seconds, rounded to the closest second.
See Figure 3.7 for illustration: line = the time window of a plane; star = its landing schedule.

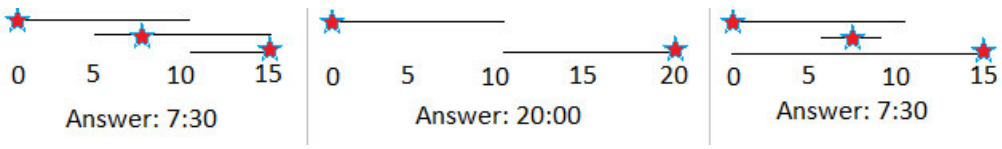

Figure 3.7: Illustration for ACM ICPC WF2009 - A - A Careful Approach

Solution:

Since the number of planes is at most 8, an optimal solution can be found by simply trying all 8! = 40320 possible orders for the planes to land. This is the **Complete Search** portion of the problem which can be easily solved using C++ STL `next_permutation`.

Now, for each specific landing order, we want to know the largest possible landing window. Suppose we use a certain window length L. We can greedily check whether this L is feasible by forcing the first plane to land as soon as possible and the subsequent planes to land in `max(a[that plane], previous landing time + L)`. This is a **Greedy Algorithm**.

A window length L that is too long/short will overshoot/undershoot `b[last plane]`, so we have to decrease/increase L. We can binary search the answer L – **Divide & Conquer**. As we only want the answer rounded to nearest integer, stopping binary search when error $\epsilon \leq$ 1e-3 is enough. For more details, please study our AC source code shown on the next page.

```
/* World Final 2009, A - A Careful Approach, LA 4445 (Accepted) */
#include <algorithm>
#include <cmath>
#include <stdio.h>
using namespace std;

int i, n, caseNo = 1, order[8];
double a[8], b[8], timeGap, maxTimeGap; // timeGap is the variable L mentioned in text

double greedyLanding() { // with certain landing order, and certain timeGap,
                         // try landing those planes and see what is the gap to b[order[n - 1]]
  double lastLanding = a[order[0]]; // greedy, first aircraft lands immediately
  for (i = 1; i < n; i++) { // for the other aircrafts
    double targetLandingTime = lastLanding + timeGap;
    if (targetLandingTime <= b[order[i]])
      // this aircraft can land, greedily choose max of a[order[i]] or targetLandingTime
      lastLanding = max(a[order[i]], targetLandingTime);
    else
      return 1; // returning positive value will force binary search to reduce timeGap
  }
  // returning negative value will force binary search to increase timeGap
  return lastLanding - b[order[n - 1]];
}

int main() {
  while (scanf("%d", &n), n) { // 2 <= n <= 8
    for (i = 0; i < n; i++) {
      scanf("%lf %lf", &a[i], &b[i]); // [ai, bi] is the interval where plane i can land safely
      a[i] *= 60; b[i] *= 60; // originally in minutes, convert to seconds
      order[i] = i;
    }

    maxTimeGap = -1; // variable to be searched for
    do { // permute plane landing order, 8!
      double lowVal = 0, highVal = 86400; // min 0s, max 1 day = 86400s
      timeGap = -1; // start with infeasible solution
      // This example code uses 'double' data type. This is actually not a good practice
      // Some other programmers avoid the test below and simply use 'loop 100 times (precise enough)'
      while (fabs(lowVal - highVal) >= 1e-3) { // binary search timeGap, ERROR = 1e-3 is OK here
        timeGap = (lowVal + highVal) / 2.0;    // we just want the answer to be rounded to nearest int
        double retVal = greedyLanding(); // round down first
        if (retVal <= 1e-2) // must increase timeGap
          lowVal = timeGap;
        else // if (retVal > 0) // infeasible, must decrease timeGap
          highVal = timeGap;
      }
      maxTimeGap = max(maxTimeGap, timeGap); // get the max over all permutations
    }
    while (next_permutation(order, order + n)); // keep trying all permutations
    // another way for rounding is to use printf format string: %.0lf:%0.2lf
    maxTimeGap = (int)(maxTimeGap + 0.5); // round to nearest second
    printf("Case %d: %d:%0.2d\n", caseNo++, (int)(maxTimeGap / 60), (int)maxTimeGap % 60);
} } // return 0;
// Challenge: rewrite this code to avoid double!
```