LEARNING dynamic-programming

#dynamic-programming

Table of Contents

About 1

Chapter 1: Getting started with dynamic-programming 2

Remarks 2

Examples 2

Introduction To Dynamic Programming 2

Understanding State in Dynamic Programming 4

Constructing a DP Solution 6

Chapter 2: Coin Changing Problem 11

Examples 11

Number of Ways to Get Total 11

Minimum Number of Coins to Get Total 12

Chapter 3: Dynamic Time Warping 16

Examples 16

Introduction To Dynamic Time Warping 16

Chapter 4: Knapsack Problem 20

Remarks 20

Examples 20

0-1 Knapsack Problem 20

Chapter 5: Matrix Chain Multiplication 25

Examples 25

Recursive Solution 25

Chapter 6: Rod Cutting 28

Examples 28

Cutting the Rod to get the maximum profit 28

Chapter 7: Solving Graph Problems Using Dynamic Programming 31

Examples 31

Floyd-Warshall Algorithm 31

Minimum Vertex Cover 33

Chapter 8: Subsequence Related Algorithms 38

Examples 38

Longest Common Subsequence 38

Longest Increasing Subsequence 42

Longest Palindromic Subsequence 46

Chapter 9: Weighted Activity Selection 49

Examples 49

Weighted Job Scheduling Algorithm 49

Credits 54

About

Chapter 1: Getting started with dynamic-programming

Remarks

This section provides an overview of what dynamic-programming is, and why a developer might want to use it.

It should also mention any large subjects within dynamic-programming, and link out to the related topics. Since the Documentation for dynamic-programming is new, you may need to create initial versions of those related topics.

Examples

Introduction To Dynamic Programming

Dynamic programming solves problems by combining the solutions to subproblems. It can be analogous to divide-and-conquer method, where problem is partitioned into disjoint subproblems, subproblems are recursively solved and then combined to find the solution of the original problem. In contrast, dynamic programming applies when the subproblems overlap - that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblems.

Let's look at an example. Italian Mathematician Leonardo Pisano Bigollo, whom we commonly know as Fibonacci, discovered a number series by considering the idealized growth of rabbit population. The series is:

```
1, 1, 2, 3, 5, 8, 13, 21, ......
```

We can notice that every number after the first two is the sum of the two preceding numbers. Now, let's formulate a function **F(n)** that will return us the nth Fibonacci number, that means,

```
F(n) = nth Fibonacci Number
```

So far, we've known that,

```
F(1) = 1
F(2) = 1
F(3) = F(2) + F(1) = 2
F(4) = F(3) + F(2) = 3
```

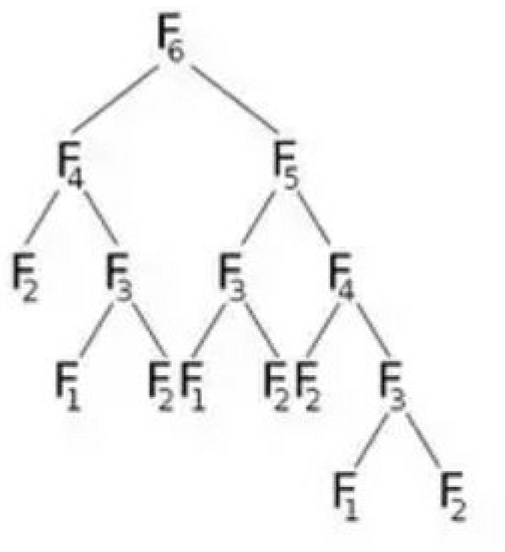
We can generalize it by:

```
F(1) = 1
F(2) = 1
F(n) = F(n-1) + F(n-2)
```

Now if we want to write it as a recursive function, we have `F(1)` and `F(2)` as our base case. So our Fibonacci Function would be:

```
Procedure F(n):                //A function to return nth Fibonacci Number
if n is equal to 1
    Return 1
else if n is equal to 2
    Return 1
end if
Return F(n-1) + F(n-2)
```

Now if we call `F(6)`, it will call `F(5)` and `F(4)`, which will call some more. Let's represent this graphically:



From the picture, we can see that `F(6)` will call `F(5)` and `F(4)`. Now `F(5)` will call `F(4)` and `F(3)`. After calculating `F(5)`, we can surely say that all the functions that were called by `F(5)` has been already calculated. That means, we have already calculated `F(4)`. But we are again calculating `F(4)` as `F(6)`'s right child indicates. Do we really need to recalculate? What we can do is, once we have calculated the value of `F(4)`, we'll store it in an array named **dp**, and will reuse it when needed. We'll initialize our **dp** array with **-1**(or any value that won't come in our calculation). Then we'll call **F(6)** where our modified **F(n)** will look like:

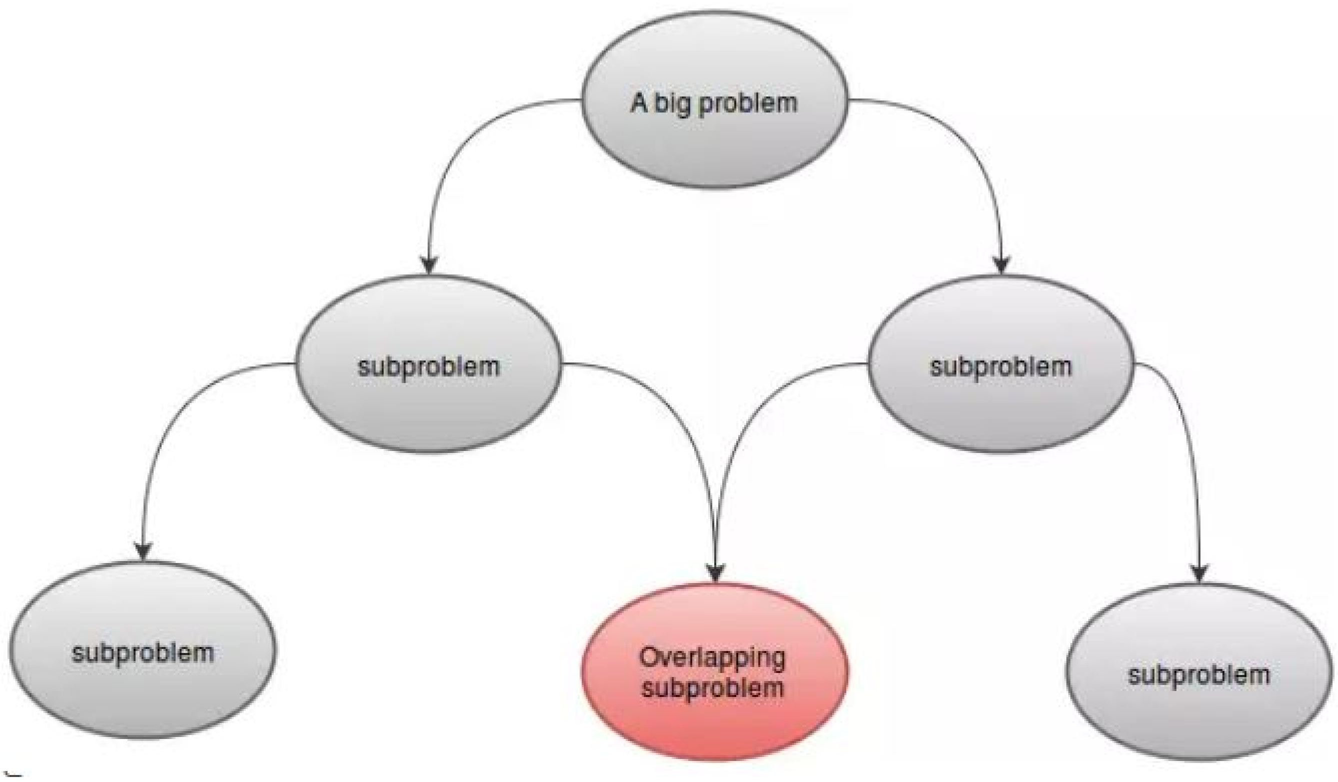
```
Procedure F(n):
if n is equal to 1
    Return 1
else if n is equal to 2
    Return 1
else if dp[n] is not equal to -1            //That means we have already calculated dp[n]
    Return dp[n]
else
    dp[n] = F(n-1) + F(n-2)
    Return dp[n]
end if
```

We've done the same task as before, but with a simple optimization. That is, we've used memoization technique. At first, all the values of **dp** array will be **-1**. When `F(4)` is called, we check if it is empty or not. If it stores **-1**, we will calculate its value and store it in **dp[4]**. If it stores anything but **-1**, that means we've already calculated its value. So we'll simply return the value.

This simple optimization using memoization is called **Dynamic Programming**.

A problem can be solved using Dynamic Programming if it has some characteristics. These are:

- **Subproblems:**
  A DP problem can be divided into one or more subproblems. For example: `F(4)` can be divided into smaller subproblems `F(3)` and `F(2)`. As the subproblems are similar to our main problem, these can be solved using same technique.
- **Overlapping Subproblems:**
  A DP problem must have overlapping subproblems. That means there must be some common part for which same function is called more than once. For example: `F(5)` and `F(6)` has `F(3)` and `F(4)` in common. This is the reason we stored the values in our array.



- **Optimal Substructure:**
  Let's say you are asked to minimize the function `g(x)`. You know that the value of `g(x)` depends on `g(y)` and `g(z)`. Now if we can minimize `g(x)` by minimizing both `g(y)` and `g(z)`, only then we can say that the problem has optimal substructure. If `g(x)` is minimized by only minimizing `g(y)` and if minimizing or maximizing `g(z)` doesn't have any effect on `g(x)`, then this problem doesn't have optimal substructure. In simple words, if optimal solution of a problem can be found from the optimal solution of its subproblem, then we can say the problem has optimal substructure property.

Understanding State in Dynamic Programming

Let's discuss with an example. From **n** items, in how many ways you can choose **r** items? You know it is denoted by ${}^{n}C_{r}$. Now think of a single item.

- If you don't select the item, after that you have to take **r** items from remaining **n-1** items, which is given by ${}^{n-1}C_{r}$.
- If you select the item, after that you have to take **r-1** items from remaining **n-1** items, which is given by ${}^{n-1}C_{r-1}$.

The summation of these two gives us the total number of ways. That is:
${}^{n}C_{r} = {}^{n-1}C_{r} + {}^{n-1}C_{r-1}$

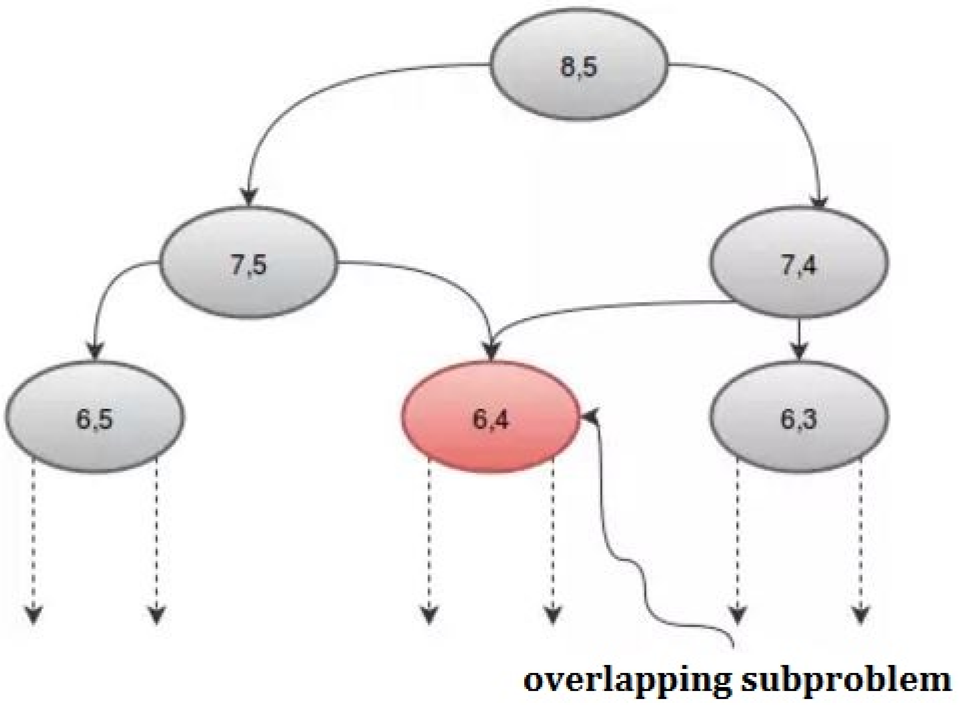
If we think `nCr(n,r)` as a function that takes `n` and `r` as parameter and determines ${}^{n}C_{r}$, we can write the relation mentioned above as:

```
nCr(n,r) = nCr(n-1,r) + nCr(n-1,r-1)
```

This is a recursive relation. To terminate it, we need to determine base case(s). We know that, ${}^{n}C_{1} = n$ and ${}^{n}C_{n} = 1$. Using these two as base cases, our algorithm to determine ${}^{n}C_{r}$ will be:

```
Procedure nCr(n, r):
if r is equal to 1
    Return n
else if n is equal to r
    Return 1
end if
Return nCr(n-1,r) + nCr(n-1,r-1)
```

If we look at the procedure graphically, we can see some recursions are called more than once. For example: if we take **n** = **8** and **r** = **5**, we get:



We can avoid this repeated call by using an array, **dp**. Since there are **2** parameters, we'll need a 2D array. We'll initialize the **dp** array with **-1**, where **-1** denotes the value hasn't been calculated yet. Our procedure will be:

```
Procedure nCr(n,r):
if r is equal to 1
    Return n
else if n is equal to r
    Return 1
else if dp[n][r] is not equal to -1        //The value has been calculated
    Return dp[n][r]
end if
dp[n][r] := nCr(n-1,r) + nCr(n-1,r-1)
Return dp[n][r]
```

To determine ${}^{n}C_{r}$, we needed two parameters **n** and **r**. These parameters are called *state*. We can simply deduce that the number of states determine the number of dimension of the **dp** array. The size of the array will change according to our need. Our dynamic programming algorithms will maintain the following general pattern:

```
Procedure DP-Function(state_1, state_2, ...., state_n)
Return if reached any base case
Check array and Return if the value is already calculated.
Calculate the value recursively for this state
Save the value in the table and Return
```

Determining *state* is one of the most crucial part of dynamic programming. It consists of the number of parameters that define our problem and optimizing their calculation, we can optimize the whole problem.

Constructing a DP Solution

No matter how many problems you solve using dynamic programming(DP), it can still surprise you. But as everything else in life, practice makes you better. Keeping these in mind, we'll look at the process of constructing a solution for DP problems. Other examples on this topic will help you understand what DP is and how it works. In this example, we'll try to understand how to come up with a DP solution from scratch.

Iterative VS Recursive:
There are two techniques of constructing DP solution. They are:

- Iterative (using for-cycles)
- Recursive (using recursion)

For example, algorithm for calculating the length of the Longest Common Subsequence of two strings **s1** and **s2** would look like:

```
Procedure LCSlength(s1, s2):
Table[0][0] = 0
for i from 1 to s1.length
    Table[0][i] = 0
```

```
endfor
for i from 1 to s2.length
    Table[i][0] = 0
endfor
for i from 1 to s2.length
    for j from 1 to s1.length
        if s2[i] equals to s1[j]
            Table[i][j] = Table[i-1][j-1] + 1
        else
            Table[i][j] = max(Table[i-1][j], Table[i][j-1])
        endif
    endfor
endfor
Return Table[s2.length][s1.length]
```

This is an iterative solution. There are a few reasons why it is coded in this way:

- Iteration is faster than recursion.
- Determining time and space complexity is easier.
- Source code is short and clean

Looking at the source code, you can easily understand how and why it works, but it is difficult to understand how to come up with this solution. However the two approaches mentioned above translates into two different pseudo-codes. One uses iteration(Bottom-Up) and another uses recursion(Top-Down) approach. The latter one is also known as memoization technique. However, the two solutions are more or less equivalent and one can be easily transformed into another. For this example, we'll show how to come up with a recursive solution for a problem.

Example Problem:
Let's say, you have **N** (**1, 2, 3, ..., N**) wines placed next to each other on a shelf. The price of **i**th wine is **p[i]**. The price of the wines increase every year. Suppose this is year **1**, on year **y** the price of the **i**th wine will be: year * price of the wine = **y***p[i]**. You want to sell the wines you have, but you have to sell exactly one wine per year, starting from this year. Again, on each year, you are allowed to sell only the leftmost or the rightmost wine and you can't rearrange the wines, that means they must stay in same order as they are in the beginning.

For example: let's say you have **4** wines in the shelf, and their prices are(from left to right):

```
+---+---+---+---+
| 1 | 4 | 2 | 3 |
+---+---+---+---+
```

The optimal solution would be to sell the wines in the order **1** -> **4** -> **3** -> **2**, which will give us a total profit of: $1 * 1 + 3 * 2 + 2 * 3 + 4 * 4 = 29$

Greedy Approach:
After brainstorming for a while, you might come up with the solution to sell the expensive wine as late as possible. So your *greedy* strategy will be:

```
Every year, sell the cheaper of the two (leftmost and rightmost) available wines.
```

Although the strategy doesn't mention what to do when the two wines cost the same, the strategy kinda feels right. But unfortunately, it isn't. If the prices of the wines are:

```
+---+---+---+---+---+
| 2 | 3 | 5 | 1 | 4 |
+---+---+---+---+---+
```

The greedy strategy would sell them in the order **1** -> **2** -> **5** -> **4** -> **3** for a total profit of: $2*1+3*2+4*3+1*4+5*5=49$

But we can do better if we sell the wines in the order **1** -> **5** -> **4** -> **2** -> **3** for a total profit of: $2*1+4*2+1*3+3*4+5*5=50$

This example should convince you that the problem is not so easy as it looks on the first sight. But it can be solved using Dynamic Programming.

Backtracking:
To come up with the memoization solution for a problem finding a backtrack solution comes handy. Backtrack solution evaluates all the valid answers for the problem and chooses the best one. For most of the problems it is easier to come up with such solution. There can be three strategies to follow in approaching a backtrack solution:

1. it should be a function that calculates the answer using recursion.
2. it should return the answer with *return* statement.
3. all the non-local variables should be used as read-only, i.e. the function can modify only local variables and its arguments.

For our example problem, we'll try to sell the leftmost or rightmost wine and recursively calculate the answer and return the better one. The backtrack solution would look like:

```
// year represents the current year
// [begin, end] represents the interval of the unsold wines on the shelf
Procedure profitDetermination(year, begin, end):
if begin > end                  //there are no more wines left on the shelf
    Return 0
Return max(profitDetermination(year+1, begin+1, end) + year * p[begin], //selling the leftmost
item
           profitDetermination(year+1, begin, end+1) + year * p[end])   //selling the
rightmost item
```

If we call the procedure using `profitDetermination(1, 0, n-1)`, where **n** is the total number of wines, it will return the answer.

This solution simply tries all the possible valid combinations of selling the wines. If there are **n** wines in the beginning, it will check 2^n possibilities. Even though we get the correct answer now, the time complexity of the algorithm grows exponentially.

The correctly written backtrack function should always represent an answer to a well-stated question. In our example, the procedure `profitDetermination` represents an answer to the question: *What is the best profit we can get from selling the wines with prices stored in the array p, when the current year is year and the interval of unsold wines spans through [begin, end], inclusive?* You should always try to create such a question for your backtrack function to see if you got it right and understand exactly what it does.

Determining State:
State is the number of parameters used in DP solution. In this step, we need to think about which of the arguments you pass to the function are redundant, i.e. we can construct them from the other arguments or we don't need them at all. If there are any such arguments, we don't need to pass them to the function, we'll calculate them inside the function.

In the example function `profitDetermination` shown above, the argument `year` is redundant. It is equivalent to the number of wines we have already sold plus one. It can be determined using the total number of wines from the beginning minus the number of wines we have not sold plus one. If we store the total number of wines **n** as a global variable, we can rewrite the function as:

```
Procedure profitDetermination(begin, end):
if begin > end
    Return 0
year := n - (end-begin+1) + 1        //as described above
Return max(profitDetermination(begin+1, end) + year * p[begin],
           profitDetermination(begin, end+1) + year * p[end])
```

We also need to think about the range of possible values of the parameters can get from a valid input. In our example, each of the parameters `begin` and `end` can contain values from **0** to **n-1**. In a valid input, we'll also expect `begin <= end + 1`. There can be `O(n²)` different arguments our function can be called with.

Memoization:
We are now almost done. To transform the backtrack solution with time complexity $O(2^n)$ into memoization solution with time complexity $O(n^2)$, we will use a little trick which doesn't require much effort.

As noted above, there are only $O(n^2)$ different parameters our function can be called with. In other words, there are only $O(n^2)$ different things we can actually compute. So where does $O(2^n)$ time complexity come from and what does it compute!!

The answer is: the exponential time complexity comes from the repeated recursion and because of that, it computes the same values again and again. If you run the code mentioned above for an arbitrary array of **n** = **20** wines and calculate how many times was the function called for arguments **begin** = **10** and **end** = **10**, you will get a number **92378**. That is a huge waste of time to compute the same answer that many times. What we could do to improve this is to store the values once we have computed them and every time the function asks for an already calculated value, we don't need to run the whole recursion again. We'll have an array **dp[N][N]**, initialize it with **-1** (or any value that will not come in our calculation), where **-1** denotes the value hasn't yet been calculated. The size of the array is determined by the maximum possible value of **begin** and **end** as we'll store the corresponding values of certain **begin** and **end** values in our array. Our procedure would look like:

```
Procedure profitDetermination(begin, end):
if begin > end
    Return 0
if dp[begin][end] is not equal to -1                //Already calculated
    Return dp[begin][end]
year := n - (end-begin+1) + 1
dp[begin][end] := max(profitDetermination(year+1, begin+1, end) + year * p[begin],
           profitDetermination(year+1, begin, end+1) + year * p[end])
Return dp[begin][end]
```

This is our required DP solution. With our very simple trick, the solution runs $O(n^2)$ time, because there are $O(n^2)$ different parameters our function can be called with and for each of them, the function runs only once with $O(1)$ complexity.

Summery:
If you can identify a problem that can be solved using DP, use the following steps to construct a DP solution:

- Create a backtrack function to provide the correct answer.
- Remove the redundant arguments.
- Estimate and minimize the maximum range of possible values of function parameters.
- Try to optimize the time complexity of one function call.
- Store the values so that you don't have to calculate it twice.

The complexity of a DP solution is: **range of possible values the function can be called with** * **time complexity of each call**.

Read Getting started with dynamic-programming online: https://riptutorial.com/dynamic-programming/topic/7946/getting-started-with-dynamic-programming

Chapter 2: Coin Changing Problem

Examples

Number of Ways to Get Total

Given coins of different denominations and a total, in how many ways can we combine these coins to get the total? Let's say we have `coins = {1, 2, 3}` and a `total = 5`, we can get the total in **5** ways:

- 1 1 1 1 1
- 1 1 1 2
- 1 1 3
- 1 2 2
- 2 3

The problem is closely related to knapsack problem. The only difference is we have unlimited supply of coins. We're going to use dynamic programming to solve this problem.

We'll use a 2D array **dp[n][total + 1]** where **n** is the number of different denominations of coins that we have. For our example, we'll need **dp[3][6]**. Here **dp[i][j]** will denote the number of ways we can get **j** if we had coins from **coins[0]** up to **coins[i]**. For example **dp[1][2]** will store if we had **coins[0]** and **coins[1]**, in how many ways we could make **2**. Let's begin:

For **dp[0][0]**, we are asking ourselves if we had only **1** denomination of coin, that is **coins[0]** = **1**, in how many ways can we get **0**? The answer is **1** way, which is if we don't take any coin at all. Moving on, **dp[0][1]** will represent if we had only **coins[0]**, in how many ways can we get **1**? The answer is again **1**. In the same way, **dp[0][2]**, **dp[0][3]**, **dp[0][4]**, **dp[0][5]** will be **1**. Our array will look like:

```
     +---+---+---+---+---+---+---+
(den)|   | 0 | 1 | 2 | 3 | 4 | 5 |
     +---+---+---+---+---+---+---+
 (1) | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
     +---+---+---+---+---+---+---+
 (2) | 1 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
 (3) | 2 |   |   |   |   |   |   |
     +---+---+---+---+---+---+---+
```

For **dp[1][0]**, we are asking ourselves if we had coins of **2** denominations, that is if we had **coins[0]** = **1** and **coins[1]** = **2**, in how many ways we could make **0**? The answer is **1**, which is by taking no coins at all. For **dp[1][1]**, since **coins[1]** = **2** is greater than our current total, **2** will not contribute to getting the total. So we'll exclude **2** and count number of ways we can get the total using **coins[0]**. But this value is already stored in **dp[0][1]**! So we'll take the value from the top. Our first formula:

```
if coins[i] > j
    dp[i][j] := dp[i-1][j]
end if
```

For **dp[1][2]**, in how many ways can we get **2**, if we had coins of denomination **1** and **2**? We can make **2** using coins of denomination of **1**, which is represented by **dp[0][2]**, again we can take **1** denomination of **2** which is stored in **dp[1][2-coins[i]]**, where **i** = **1**. Why? It'll be apparent if we look at the next example. For **dp[1][3]**, in how many ways can we get **3**, if we had coins of denomination **1** and **2**? We can make **3** using coins of denomination **1** in **1** way, which is stored in **dp[0][3]**. Now if we take **1** denomination of **2**, we'll need **3** - **2** = **1** to get the total. The number of ways to get **1** using the coins of denomination **1** and **2** is stored in **dp[1][1]**, which can be written as, **dp[i][j-coins[i]]**, where **i** = **1**. This is why we wrote the previous value in this way. Our second and final formula will be:

```
if coins[i] <= j
    dp[i][j] := dp[i-1][j] + dp[i][j-coins[i]]
end if
```

This is the two required formulae to fill up the whole **dp** array. After filling up the array will look like:

```
     +---+---+---+---+---+---+---+
(den)|   | 0 | 1 | 2 | 3 | 4 | 5 |
     +---+---+---+---+---+---+---+
 (1) | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
     +---+---+---+---+---+---+---+
 (2) | 1 | 1 | 1 | 2 | 2 | 3 | 3 |
     +---+---+---+---+---+---+---+
 (3) | 2 | 1 | 1 | 2 | 3 | 4 | 5 |
     +---+---+---+---+---+---+---+
```

dp[2][5] will contain our required answer. The algorithm:

```
Procedure CoinChange(coins, total):
n := coins.length
dp[n][total + 1]
for i from 0 to n
    dp[i][0] := 1
end for
for i from 0 to n
    for j from 1 to (total + 1)
        if coins[i] > j
            dp[i][j] := dp[i-1][j]
        else
            dp[i][j] := dp[i-1][j] + dp[i][j-coins[i]]
        end if
    end for
end for
Return dp[n-1][total]
```

The time complexity of this algorithm is `O(n * total)`, where **n** is the number of denominations of coins we have.

Minimum Number of Coins to Get Total

Given coins of different denominations and a total, how many coins do we need to combine to get the total if we use minimum number of coins? Let's say we have `coins = {1, 5, 6, 8}` and a `total = 11`, we can get the total using **2** coins which is `{5, 6}`. This is indeed the minimum number of coins required to get **11**. We'll also assume that there are unlimited supply of coins. We're going to use dynamic programming to solve this problem.

We'll use a 2D array **dp[n][total + 1]** where **n** is the number of different denominations of coins that we have. For our example, we'll need **dp[4][12]**. Here **dp[i][j]** will denote the minimum number of coins needed to get **j** if we had coins from **coins[0]** up to **coins[i]**. For example **dp[1][2]** will store if we had **coins[0]** and **coins[1]**, what is the minimum number of coins we can use to make **2**. Let's begin:

At first, for the **0**th column, can make **0** by not taking any coins at all. So all the values of **0**th column will be **0**. For **dp[0][1]**, we are asking ourselves if we had only **1** denomination of coin, that is **coins[0]** = **1**, what is the minimum number of coins needed to get **1**? The answer is **1**. For **dp[0][2]**, if we had only **1**, what is the minimum number of coins needed to get **2**. The answer is **2**. Similarly **dp[0][3]** = **3**, **dp[0][4]** = **4** and so on. One thing to mention here is that, if we didn't have a coin of denomination **1**, there might be some cases where the total can't be achieved using **1** coin only. For simplicity we take **1** in our example. After first iteration our **dp** array will look like:

```
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
 (denom)|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (1) | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (5) | 1 | 0 |   |   |   |   |   |   |   |   |   |   |   |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (6) | 2 | 0 |   |   |   |   |   |   |   |   |   |   |   |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (8) | 3 | 0 |   |   |   |   |   |   |   |   |   |   |   |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Moving on, for **dp[1][1]**, we are asking ourselves if we had **coins[0]** = **1** and **coins[1]** = **5**, what is the minimum number of coins needed to get **1**? Since **coins[1]** is greater than our current total, it will not affect our calculation. We'll need to exclude **coins[5]** and get **1** using **coins[0]** only. This value is stored in **dp[0][1]**. So we take the value from the top. Our first formula is:

```
if coins[i] > j
    dp[i][j] := dp[i-1][j]
```

This condition will be true until our total is **5** in **dp[1][5]**, for that case, we can make **5** in two ways:

- We take **5** denominations of **coins[0]**, which is stored on **dp[0][5]** (from the top).
- We take **1** denomination of **coins[1]** and (**5** - **5**) = **0** denominations of **coins[0]**.

We'll choose the minimum of these two. So **dp[1][5]** = min(**dp[0][5]**, **1** + **dp[1][0]**) = **1**. Why did we mention **0** denominations of **coins[0]** here will be apparent in our next position.

For **dp[1][6]**, we can make **6** in two ways:

- We take **6** denominations of **coins[0]**, which is stored on the top.
- We can take **1** denomination of **5**, we'll need **6 - 5** = **1** to get the total. The minimum number of ways to get **1** using the coins of denomination **1** and **5** is stored in **dp[1][1]**, which can be written as **dp[i][j-coins[i]]**, where **i** = **1**. This is why we wrote the previous value in that fashion.

We'll take the minimum of these two ways. So our second formula will be:

```
if coins[i] >= j
    dp[i][j] := min(dp[i-1][j], dp[i][j-coins[i]])
```

Using these two formulae we can fill up the whole table. Our final result will be:

```
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
  (denom)|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (1) | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (5) | 1 | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (6) | 2 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 4 | 2 | 2 |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (8) | 3 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 1 | 2 | 2 | 2 |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Our required result will be stored at **dp[3][11]**. The procedure will be:

```
Procedure coinChange(coins, total):
n := coins.length
dp[n][total + 1]
for i from 0 to n
    dp[i][0] := 0
end for
for i from 1 to (total + 1)
    dp[0][i] := i
end for
for i from 1 to n
    for j from 1 to (total + 1)
        if coins[i] > j
            dp[i][j] := dp[i-1][j]
        else
            dp[i][j] := min(dp[i-1][j], dp[i][j-coins[i]])
        end if
    end for
end for
Return dp[n-1][total]
```

The runtime complexity of this algorithm is: O(n*total) where **n** is the number of denominations of coins.

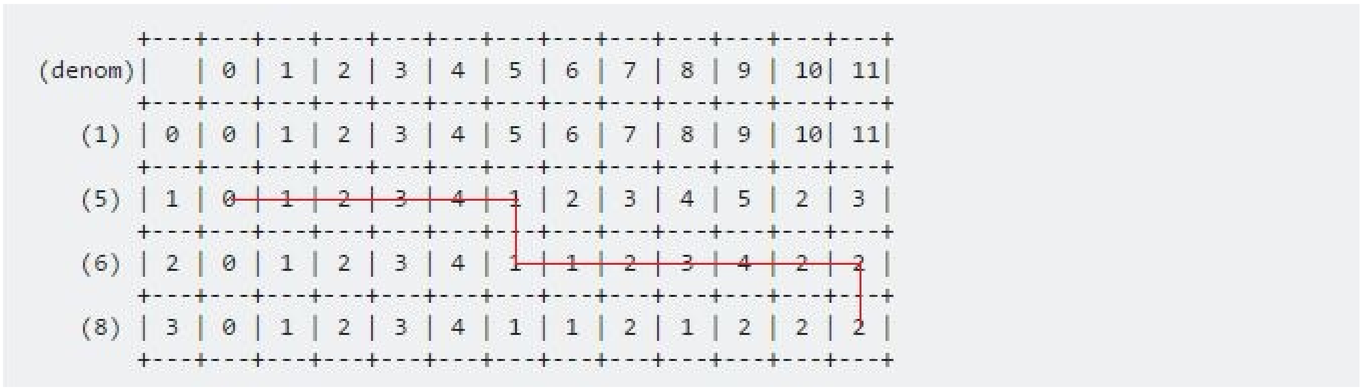
To print the coins needed, we need to check:

- if the value came from top, then the current coin is not included.
- if the value came from left, then the current coin is included.

The algorithm would be:

```
Procedure printChange(coins, dp, total):
i := coins.length - 1
j := total
min := dp[i][j]
while j is not equal to 0
    if dp[i-1][j] is equal to min
        i := i - 1
    else
        Print(coins[i])
        j := j - coins[i]
    end if
end while
```

For our example, the direction will be:



```
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
  (denom)|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (1) | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (5) | 1 | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (6) | 2 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 4 | 2 | 2 |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
     (8) | 3 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 1 | 2 | 2 | 2 |
         +---+---+---+---+---+---+---+---+---+---+---+---+---+
```

The values will be **6**, **5**.

Read Coin Changing Problem online: https://riptutorial.com/dynamic-programming/topic/8013/coin-changing-problem

Chapter 3: Dynamic Time Warping

Examples

Introduction To Dynamic Time Warping

Dynamic Time Warping(DTW) is an algorithm for measuring similarity between two temporal sequences which may vary in speed. For instance, similarities in walking could be detected using DTW, even if one person was walking faster than the other, or if there were accelerations and decelerations during the course of an observation. It can be used to match a sample voice command with others command, even if the person talks faster or slower than the prerecorded sample voice. DTW can be applied to temporal sequences of video, audio and graphics data- indeed, any data which can be turned into a linear sequence can be analyzed with DTW.

In general, DTW is a method that calculates an optimal match between two given sequences with certain restrictions. But let's stick to the simpler points here. Let's say, we have two voice sequences **Sample** and **Test**, and we want to check if these two sequences match or not. Here voice sequence refers to the converted digital signal of your voice. It might be the amplitude or frequency of your voice that denotes the words you say. Let's assume:

```
Sample = {1, 2, 3, 5, 5, 5, 6}
Test   = {1, 1, 2, 2, 3, 5}
```

We want to find out the optimal match between these two sequences.

At first, we define the distance between two points, *d(x, y)* where **x** and **y** represent the two points. Let,

```
d(x, y) = |x - y|     //absolute difference
```

Let's create a 2D matrix **Table** using these two sequences. We'll calculate the distances between each point of **Sample** with every points of **Test** and find the optimal match between them.

```
+------+------+------+------+------+------+------+------+
|      |  0   |  1   |  1   |  2   |  2   |  3   |  5   |
+------+------+------+------+------+------+------+------+
|   0  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   1  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   2  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   3  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   6  |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
```

Here, **Table[i][j]** represents the optimal distance between two sequences if we consider the sequence up to **Sample[i]** and **Test[j]**, considering all the optimal distances we observed before.

For the first row, if we take no values from **Sample**, the distance between this and **Test** will be *infinity*. So we put *infinity* on the first row. Same goes for the first column. If we take no values from **Test**, the distance between this one and **Sample** will also be infinity. And the distance between **0** and **0** will simply be **0**. We get,

```
+------+------+------+------+------+------+------+------+
|      |  0   |  1   |  1   |  2   |  2   |  3   |  5   |
+------+------+------+------+------+------+------+------+
|   0  |  0   | inf  | inf  | inf  | inf  | inf  | inf  |
+------+------+------+------+------+------+------+------+
|   1  | inf  |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   2  | inf  |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   3  | inf  |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  | inf  |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  | inf  |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   5  | inf  |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
|   6  | inf  |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
```

Now for each step, we'll consider the distance between each points in concern and add it with the minimum distance we found so far. This will give us the optimal distance of two sequences up to that position. Our formula will be,

```
Table[i][j] := d(i, j) + min(Table[i-1][j], Table[i-1][j-1], Table[i][j-1])
```

For the first one, **d(1, 1)** = **0**, **Table[0][0]** represents the minimum. So the value of **Table[1][1]** will be **0** + **0** = **0**. For the second one, **d(1, 2)** = **0**. **Table[1][1]** represents the minimum. The value will be: **Table[1][2]** = **0** + **0** = **0**. If we continue this way, after finishing, the table will look like:

```
+------+------+------+------+------+------+------+------+
|      |  0   |  1   |  1   |  2   |  2   |  3   |  5   |
+------+------+------+------+------+------+------+------+
|   0  |  0   | inf  | inf  | inf  | inf  | inf  | inf  |
+------+------+------+------+------+------+------+------+
|   1  | inf  |  0   |  0   |  1   |  2   |  4   |  8   |
+------+------+------+------+------+------+------+------+
|   2  | inf  |  1   |  1   |  0   |  0   |  1   |  4   |
+------+------+------+------+------+------+------+------+
|   3  | inf  |  3   |  3   |  1   |  1   |  0   |  2   |
+------+------+------+------+------+------+------+------+
|   5  | inf  |  7   |  7   |  4   |  4   |  2   |  0   |
+------+------+------+------+------+------+------+------+
|   5  |  inf |  11  |  11  |   7  |   7  |   4  |   0  |
+------+------+------+------+------+------+------+------+
|   5  |  inf |  15  |  15  |  10  |  10  |   6  |   0  |
+------+------+------+------+------+------+------+------+
|   6  |  inf |  20  |  20  |  14  |  14  |   9  |   1  |
+------+------+------+------+------+------+------+------+
```

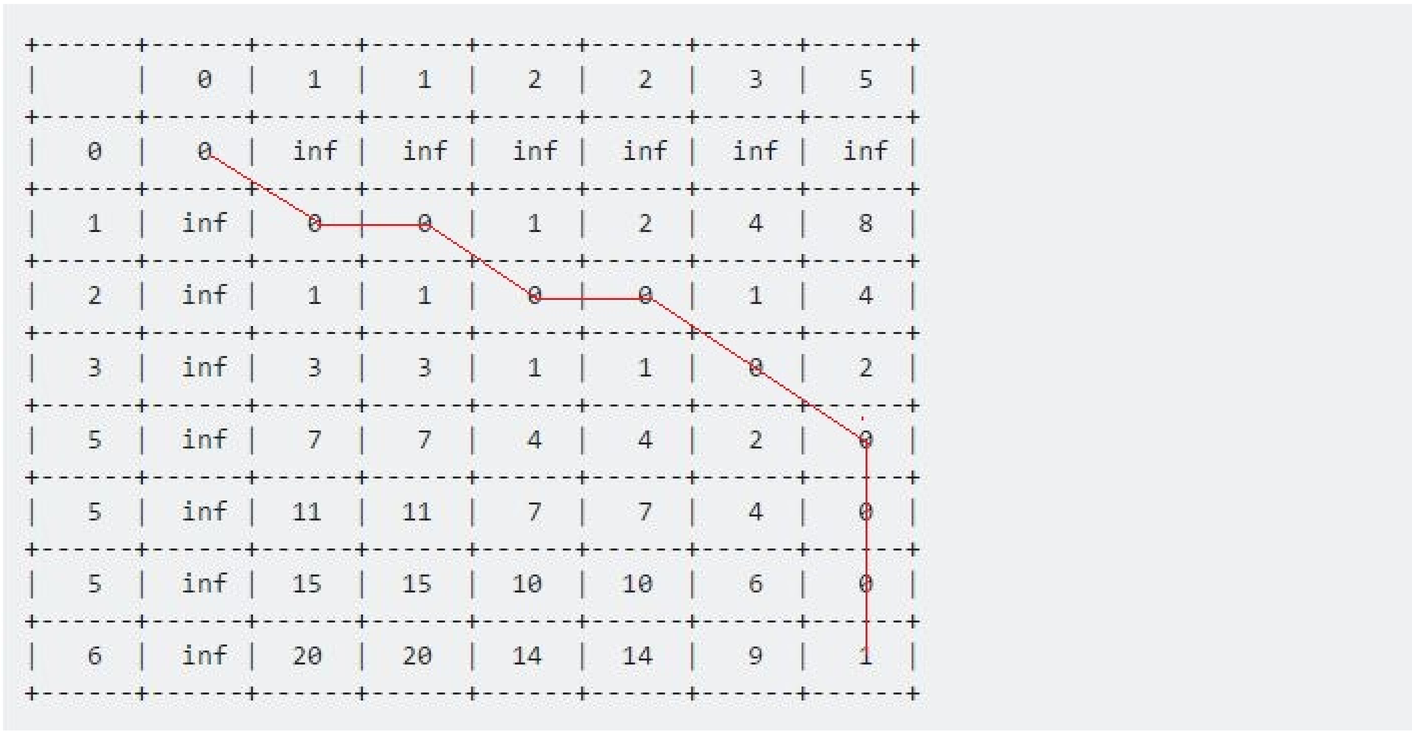
The value at **Table[7][6]** represents the maximum distance between these two given sequences. Here **1** represents the maximum distance between **Sample** and **Test** is **1**.

Now if we backtrack from the last point, all the way back towards the starting **(0, 0)** point, we get a long line that moves horizontally, vertically and diagonally. Our backtracking procedure will be:

```
if Table[i-1][j-1] <= Table[i-1][j] and Table[i-1][j-1] <= Table[i][j-1]
    i := i - 1
    j := j - 1
else if Table[i-1][j] <= Table[i-1][j-1] and Table[i-1][j] <= Table[i][j-1]
    i := i - 1
else
    j := j - 1
end if
```

We'll continue this till we reach **(0, 0)**. Each move has its own meaning:

- A horizontal move represents deletion. That means our **Test** sequence accelerated during this interval.
- A vertical move represents insertion. That means out **Test** sequence decelerated during this interval.
- A diagonal move represents match. During this period **Test** and **Sample** were same.



```
+------+------+------+------+------+------+------+------+
|      |   0  |   1  |   1  |   2  |   2  |   3  |   5  |
+------+------+------+------+------+------+------+------+
|   0  |   0  |  inf |  inf |  inf |  inf |  inf |  inf |
+------+------+------+------+------+------+------+------+
|   1  |  inf |   0  |   0  |   1  |   2  |   4  |   8  |
+------+------+------+------+------+------+------+------+
|   2  |  inf |   1  |   1  |   0  |   0  |   1  |   4  |
+------+------+------+------+------+------+------+------+
|   3  |  inf |   3  |   3  |   1  |   1  |   0  |   2  |
+------+------+------+------+------+------+------+------+
|   5  |  inf |   7  |   7  |   4  |   4  |   2  |   0  |
+------+------+------+------+------+------+------+------+
|   5  |  inf |  11  |  11  |   7  |   7  |   4  |   0  |
+------+------+------+------+------+------+------+------+
|   5  |  inf |  15  |  15  |  10  |  10  |   6  |   0  |
+------+------+------+------+------+------+------+------+
|   6  |  inf |  20  |  20  |  14  |  14  |   9  |   1  |
+------+------+------+------+------+------+------+------+
```

Our pseudo-code will be:

```
Procedure DTW(Sample, Test):
n := Sample.length
m := Test.length
Create Table[n + 1][m + 1]
for i from 1 to n
    Table[i][0] := infinity
end for
for i from 1 to m
    Table[0][i] := infinity
end for
Table[0][0] := 0
for i from 1 to n
    for j from 1 to m
        Table[i][j] := d(Sample[i], Test[j])
                       + minimum(Table[i-1][j-1],      //match
                                 Table[i][j-1],        //insertion
                                 Table[i-1][j])        //deletion
    end for
end for
Return Table[n + 1][m + 1]
```

We can also add a locality constraint. That is, we require that if `Sample[i]` is matched with `Test[j]`, then `|i - j|` is no larger than **w**, a window parameter.

Complexity:

The complexity of computing DTW is **O(m * n)** where **m** and **n** represent the length of each sequence. Faster techniques for computing DTW include PrunedDTW, SparseDTW and FastDTW.

Applications:

- Spoken word recognition
- Correlation Power Analysis

Read Dynamic Time Warping online: https://riptutorial.com/dynamic-programming/topic/7967/dynamic-time-warping

Chapter 4: Knapsack Problem

Remarks

The knapsack problem or rucksack problem is a problem in combinatorial optimization. Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by fixed-size knapsack and must fill it with the most valuable items.

The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatorics, computer science, complexity theory, cryptography, applied mathematics and daily fantasy sports.

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name "knapsack problem" dates back to the early works of mathematician Tobias Dantzig (1884-1956) and refers to the commonplace problem of packing your mosty valuable or useful items without overloading your luggage.

Examples

0-1 Knapsack Problem

Suppose you are asked, given the total weight you can carry on your knapsack and some items with their weight and values, how can you take those items in such a way that the sum of their values are maximum, but the sum of their weights don't exceed the total weight you can carry? The **0-1** indicates either you pick the item or you don't. Also we have one quantity of each item. It means that, you can't split the item. If it was not a **0-1 knapsack problem**, that means if you could have split the items, there's a greedy solution to it, which is called **fractional knapsack problem**.

Let's, for now, concentrate on our problem at hand. For example, let's say we have a knapsack capacity of **7**. We have **4** items. Their weights and values are:

```
+----------+---+---+---+---+
|   Item   | 1 | 2 | 3 | 4 |
+----------+---+---+---+---+
|  Weight  | 1 | 3 | 4 | 5 |
+----------+---+---+---+---+
|   Value  | 1 | 4 | 5 | 7 |
+----------+---+---+---+---+
```

One brute force method would be taking all possible combinations of items. Then we can calculate their total weights and exclude them which exceed our knapsack's capacity and find out the one that gives us maximum value. For **4** items, we'll need to check (**4! - 1**) = **23** possible combinations (excluding one with no items). This is quite cumbersome when the number of items increase. Here, a few aspects we can notice, that is:

- We can take lesser items and calculate the maximum value we can get using those items and combine them. So our problem can be divided into subproblems.
- If we calculate the combinations for item **{1,2}**, we can use it when we calculate **{1, 2, 3}**.
- If we minimize the weight and maximize the value, we can find out our optimal solution.

For these reasons, we'll use dynamic programming to solve our problem. Our strategy will be: whenever a new item comes, we'll check if we can pick the item or not and again we'll pick the items that give us maximum value. Now if we pick the item, our value will be the value of the item, plus whatever the value we can get by subtracting the value from our capacity and the maximum we can get for that remaining weight. If we don't pick the item, we'll pick the items that gives us maximum value without including the item. Let's try to understand it with our example:

We'll take a 2D array **table**, where the number of columns will be the maximum value we can get by taking the items + 1. And the number of rows will be same as the number of items. Our array will look like:

```
+-------+--------+---+---+---+---+---+---+---+---+
| Value | Weight | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+--------+---+---+---+---+---+---+---+---+
|   1   |    1   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
|   4   |    3   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
|   5   |    4   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
|   7   |    5   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
```

We've incorporate the weight and value of each item to the array for our convenience. Remember these are not part of the array, these are for calculation purpose only, you need not store these values in **table** array.

Our first column is filled with **0**. It means if our maximum capacity is **0**, no matter whatever item we have, since we can't pick any items, our maximum value will be **0**. Let's start from **Table[0][1]**. When we are filling **Table[1][1]**, we are asking ourselves if our maximum capacity was **1** and we had only the first item, what would be our maximum value? The best we can do is **1**, by picking the item. For **Table[0][2]** that means if our maximum capacity is **2** and we only have the first item, the maximum value we can get is **1**. This will be same for **Table[0][3]**, **Table[0][4]**, **Table[0][5]**, **Table[0][6]** and **Table[0][7]**. This is because we only have one item, which gives us value **1**. Since we have only **1** quantity of each item, no matter how we increase the capacity of our knapsack, from one item, **1** is the best value we can make. So our array will look like:

```
+-------+--------+---+---+---+---+---+---+---+---+
| Value | Weight | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+--------+---+---+---+---+---+---+---+---+
|   1   |    1   | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
+-------+--------+---+---+---+---+---+---+---+---+
|   4   |    3   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
|   5   |    4   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
|   7   |    5   | 0 |   |   |   |   |   |   |   |
+-------+--------+---+---+---+---+---+---+---+---+
```

Moving on, for **Table[1][1]**, we are asking ourselves, if we had item **1** and **2** and if the maximum capacity of our knapsack was **1**, what is the maximum value we can get? If we take both item **1** and **2**, the total weight will be **4**, which will exceed our current knapsack capacity. So item **2** can't be selected. Now what is the best we can do without taking item **2**? The value from the top, that is **Table[0][1]** which contains the maximum value we can get if we had maximum capacity **1** and we didn't pick the second item. For **Table[1][2]**, since **2** is less than **weight[2]**, that is the weight of the second item, we can't take the item. So we can establish that, if the weight of the current item is greater than our maximum capacity, we can't take that item. In this case, we'll simply take the value from top, which represents the maximum value we can take excluding the item.

```
if weight[i] > j
    Table[i][j] := Table[i-1][j]
end if
```

Now for **Table[1][3]** since our maximum capacity is equal to our current weight, we have two choices.

- We pick the item and add its value with the maximum value we can get from other remaining items after taking this item.
- We can exclude this item.

Among the two choices, we'll pick the one from which we can get maximum value. If we select the item, we get: value for this item + maximum value from rest of the items after taking this item = **4** + **0** = **4**. We get **4**(value of the item) from our **weight** array and the **0**(maximum value we can get from rest of the items after taking this item) comes by going **1** step above and **3** steps back. That is **Table[0][0]**. Why? If we take the item, our remaining capacity will be **3** - **3** = **0** and remaining item will be the first item. Well, if you recall **Table[0][0]** stores the maximum value we can get if our capacity was **0** and we only had the first item. Now if we don't select the item, the maximum value we can get is comes from **1** step above, that is **1**. Now we take the maximum of these two values(**4**, **1**) and set **Table[1][2]** = **4**. For **Table[1][4]**, since **4**, the maximum knapsack capacity is greater than **3**, the weight of our current item, we again have two options. We take max(**Weight[2]** + **Table[0][4-Weight[2]]**, **Table[0][4]**) = max(**Weight[2] + Table[0][1]**, **Table[0][4]**) = max(**4** + **1**, **1**) = **5**.

```
if weight[i] <= j
    w := weight[i]
    Table[i][j] := max(w + Table[i][j-w], Table[i-1][j])
end if
```

Using these two formulae, we can populate the whole **Table** array. Our array will look like:

```
+-------+--------+---+---+---+---+---+---+---+---+
| Value | Weight | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+--------+---+---+---+---+---+---+---+---+
|   1   |    1   | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
+-------+--------+---+---+---+---+---+---+---+---+
|   4   |    3   | 0 | 1 | 1 | 4 | 5 | 5 | 5 | 5 |
+-------+--------+---+---+---+---+---+---+---+---+
|   5   |    4   | 0 | 1 | 1 | 4 | 5 | 6 | 6 | 9 |
+-------+--------+---+---+---+---+---+---+---+---+
|   7   |    5   | 0 | 1 | 1 | 4 | 5 | 7 | 8 | 9 |
+-------+--------+---+---+---+---+---+---+---+---+
```

Here, The last value that we inserted in our array, **Table[3][7]** contains our required maximum value. This is the maximum value we can get if we had **4** items and our maximum capacity of the knapsack was **7**.

Here one thing we must remember that, even for the first row, the weight can be greater than the capacity of the knapsack. We'll need to keep another constraint to check the value while filling the first row. Or we can simply take another row and set all the values of the first row to **0**. The pseudo-code would look like:

```
Procedure Knapsack(Weight, Value, maxCapacity):
n := Item.size - 1
Table[n+1][maxCapacity+1]
for i from 0 to n
    Table[i][0] := 0
end for
for j from 1 to maxCapacity
    if j >= Weight[0]
        Table[0][j] := Weight[0]
    else
        Table[0][j] := 0
    end if
end for
for i from 1 to n
    for j from 1 to maxCapacity
        if Weight[i] >= j                                           //can't pick the item
            Table[i][j] := Table[i-1][j]
        else                                                       //can pick the item
            w := Weight[i]
            Table[i][j] := max(w + Table[i-1][j-w], Table[i-1][j])
        end if
    end for
end for
Return Table[n][maxCapacity]
```

The time complexity of this algorithm is `O(n*maxCapacity)`, where **n** is the number of items and `maxCapacity` is the maximum capacity of our knapsack.

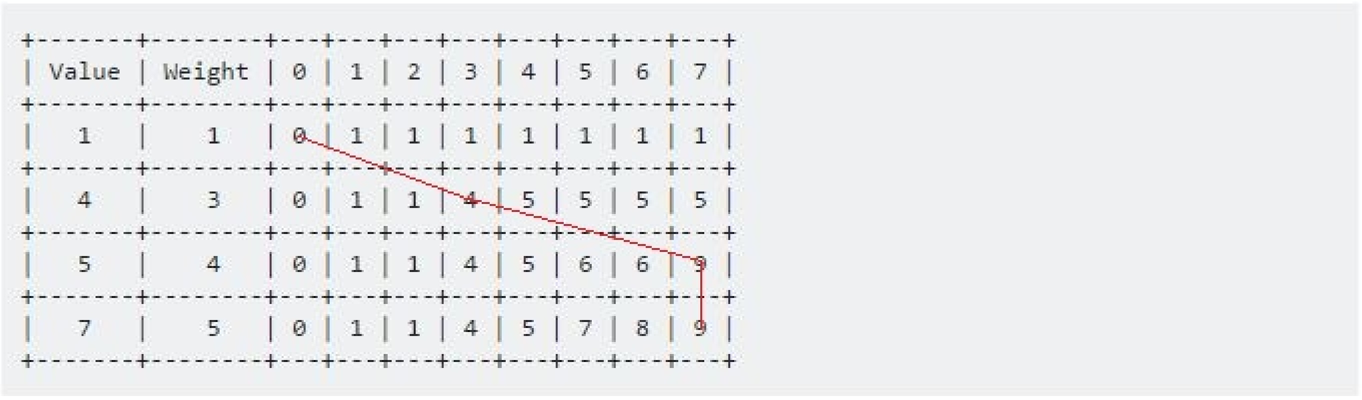
So far, we've found the maximum value we can get from our example. One question still remains. What are the actual items? We'll retrace the values in our **Table** array to find out the items we've taken. We'll follow two strategies:

- For any item, if the value is coming from the position above, we didn't take the current item. We go 1 step above.
- If the value is not coming from the position above, we took the item. So we go 1 step above and x steps back where x is the weight of the current item.

The pseudo-code will be:

```
Procedure printItems(Table, maxCapacity, Value):
i := Item.size - 1
j := maxCapacity
while j is not equal to 0
    if Table[i][j] is equal to Table[i-1][j]
        i := i - 1
    else
        Print: i
        j := j - weight[i]
        i := i - 1
    end if
end while
```

If we retrace our example, we'll get:



```
+-------+--------+---+---+---+---+---+---+---+---+
| Value | Weight | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+--------+---+---+---+---+---+---+---+---+
|   1   |    1   | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
+-------+--------+---+---+---+---+---+---+---+---+
|   4   |    3   | 0 | 1 | 1 | 4 | 5 | 5 | 5 | 5 |
+-------+--------+---+---+---+---+---+---+---+---+
|   5   |    4   | 0 | 1 | 1 | 4 | 5 | 6 | 6 | 9 |
+-------+--------+---+---+---+---+---+---+---+---+
|   7   |    5   | 0 | 1 | 1 | 4 | 5 | 7 | 8 | 9 |
+-------+--------+---+---+---+---+---+---+---+---+
```

From this, we can say that, we can take item **2** and **3** to get the maximum value.

Read Knapsack Problem online: https://riptutorial.com/dynamic-programming/topic/7972/knapsack-problem

Chapter 5: Matrix Chain Multiplication

Examples

Recursive Solution

Matrix chain multiplication is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, the goal is to find the most efficient way to multiply these matrices. The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved.

Let's say we have two matrices $\mathbf{A_1}$ and $\mathbf{A_2}$ of dimension **m * n** and **p * q**. From the rules of matrix multiplication, we know that,

1. We can multiply $\mathbf{A_1}$ and $\mathbf{A_2}$ if and only if **n** = **p**. That means the number of columns of $\mathbf{A_1}$ must be equal to the number of rows of $\mathbf{A_2}$.
2. If the first condition is satisfied and we do multiply $\mathbf{A_1}$ and $\mathbf{A_2}$, we'll get a new matrix, let's call it $\mathbf{A_3}$ of dimension **m** * **q**.
3. To multiply $\mathbf{A_1}$ and $\mathbf{A_2}$, we need to do some scaler multiplications. The total number of scaler multiplications, we need to perform is (**m** * **n** * **q**) or (**m** * **p** * **q**).
4. $\mathbf{A_1}$ * $\mathbf{A_2}$ is not equal to $\mathbf{A_2}$ * $\mathbf{A_1}$.

If we have three matrices $\mathbf{A_1}$, $\mathbf{A_2}$ and $\mathbf{A_3}$ having dimension **m** * **n**, **n** * **p** and **p** * **q** respectively, then $\mathbf{A_4}$ = $\mathbf{A_1}$ * $\mathbf{A_2}$ * $\mathbf{A_3}$ will have dimension of **m** * **q**. Now we can perform this matrix multiplication in two ways:

- First we multiply $\mathbf{A_1}$ and $\mathbf{A_2}$, then with the result we multiply $\mathbf{A_3}$. That is: ($\mathbf{A_1}$ * $\mathbf{A_2}$) * $\mathbf{A_3}$.
- First we multiply $\mathbf{A_2}$ and $\mathbf{A_3}$, then with the result we multiply $\mathbf{A_1}$. That is: $\mathbf{A_1}$ * ($\mathbf{A_2}$ * $\mathbf{A_3}$).

You can notice the order of the multiplication remains the same, i.e. we don't multiply ($\mathbf{A_1}$ * $\mathbf{A_3}$) * $\mathbf{A_2}$ because it might not be valid. We only change the *parenthesis* to multiply a set before multiplying it with the remaining. How we place these parenthesis are important. Why? Let's say, the dimension of 3 matrices $\mathbf{A_1}$, $\mathbf{A_2}$ and $\mathbf{A_3}$ are **10** * **100**, **100** * **5**, **5** * **50**. Then,

1. For ($\mathbf{A_1}$ * $\mathbf{A_2}$) * $\mathbf{A_3}$, the total number of scaler multiplications are: (**10** * **100** * **5**) + (**10** * **5** * **50**) = **7500** times.
2. For $\mathbf{A_1}$ * ($\mathbf{A_2}$ * $\mathbf{A_3}$), the total number of scaler multiplications are: (**100** * **5** * **50**) + (**10** * **100** * **50**) = **75000** times.

For the 2nd type the number of scaler multiplication is **10** times the number of 1st type! So if you can devise a way to find out the correct orientation of parenthesis needed to minimize the total scaler multiplication, it would reduce both time and memory needed for matrix multiplication. This is where matrix chain multiplication comes in handy. Here, we'll not be concerned with the actual multiplication of matrices, we'll only find out the correct parenthesis order so that the number of scaler multiplication is minimized. We'll have matrices $\mathbf{A_1}$, $\mathbf{A_2}$, $\mathbf{A_3}$ .... $\mathbf{A_n}$ and we'll find out the the minimum number of scaler multiplications needed to multiply these. We'll assume that the given dimensions are valid, i.e. it satisfies our first requirement for matrix multiplication.

We'll use divide-and-conquer method to solve this problem. Dynamic programming is needed because of common subproblems. For example: for **n** = **5**, we have **5** matrices $\mathbf{A_1}$, $\mathbf{A_2}$, $\mathbf{A_3}$, $\mathbf{A_4}$ and $\mathbf{A_5}$. We want to find out the minimum number of multiplication needed to perform this matrix multiplication $\mathbf{A_1}$ * $\mathbf{A_2}$ * $\mathbf{A_3}$ * $\mathbf{A_4}$ * $\mathbf{A_5}$. Of the many ways, let's concentrate on one: ($\mathbf{A_1}$ * $\mathbf{A_2}$) * ($\mathbf{A_3}$ * $\mathbf{A_4}$ * $\mathbf{A_5}$).

For this one, we'll find out $\mathbf{A_{left}}$ = $\mathbf{A_1}$ * $\mathbf{A_2}$. $\mathbf{A_{right}}$ = $\mathbf{A_3}$ * $\mathbf{A_4}$ * $\mathbf{A_5}$. Then we'll find out $\mathbf{A_{answer}}$ = $\mathbf{A_{left}}$ * $\mathbf{A_{right}}$. The total number of scaler multiplications needed to find out $\mathbf{A_{answer}}$: = The total number of scaler multiplications needed to determine $\mathbf{A_{left}}$ + The total number of scaler multiplications needed to determine $\mathbf{A_{right}}$ + The total number of scaler multiplications needed to determine $\mathbf{A_{left}}$ * $\mathbf{A_{right}}$.

The last term, The total number of scaler multiplications needed to determine $\mathbf{A_{left}}$ * $\mathbf{A_{right}}$ can be written as: The number of rows in $\mathbf{A_{left}}$ * the number of columns in $\mathbf{A_{left}}$ * the number of columns in $\mathbf{A_{right}}$. (According to the 2nd rule of matrix multiplication)

But we could set the parenthesis in other ways too. For example:

- $\mathbf{A_1}$ * ($\mathbf{A_2}$ * $\mathbf{A_3}$ * $\mathbf{A_4}$ * $\mathbf{A_5}$)
- ($\mathbf{A_1}$ * $\mathbf{A_2}$ * $\mathbf{A_3}$) * ($\mathbf{A_4}$ * $\mathbf{A_5}$)
- ($\mathbf{A_1}$ * $\mathbf{A_2}$ * $\mathbf{A_3}$ * $\mathbf{A_4}$) * $\mathbf{A_5}$ etc.

For each and every possible cases, we'll determine the number of scaler multiplication needed to find out $\mathbf{A_{left}}$ and $\mathbf{A_{right}}$, then for $\mathbf{A_{left}}$ * $\mathbf{A_{right}}$. If you have a general idea about recursion, you've already understood how to perform this task. Our algorithm will be:

```
- Set parenthesis in all possible ways.
- Recursively solve for the smaller parts.
- Find out the total number of scaler multiplication by merging left and right.
- Of all possible ways, choose the best one.
```

Why this is dynamic programming problem? To determine ($\mathbf{A_1}$ * $\mathbf{A_2}$ * $\mathbf{A_3}$), if you've already calculated ($\mathbf{A_1}$ * $\mathbf{A_2}$), it'll be helpful in this case.

To determine the state of this recursion, we can see that to solve for each case, we'll need to know the range of matrices we're working with. So we'll need a **begin** and **end**. As we're using divide and conquer, our base case will be having less than **2** matrices (**begin** >= **end**), where we don't need to multiply at all. We'll have **2** arrays: **row** and **column**. **row[i]** and **column[i]** will store the number of rows and columns for matrix $\mathbf{A_i}$. We'll have a **dp[n][n]** array to store the already calculated values and initialize it with **-1**, where **-1** represents the value has not been calculated yet. **dp[i][j]** represents the number of scaler multiplications needed to multiply $\mathbf{A_i}$, $\mathbf{A_{i+1}}$, ...,$\mathbf{A_j}$ inclusive. The pseudo-code will look like:

```
Procedure matrixChain(begin, end):
if begin >= end
    Return 0
else if dp[begin][end] is not equal to -1
    Return dp[begin][end]
```

```
end if
answer := infinity
for mid from begin to end
    operation_for_left := matrixChain(begin, mid)
    operation_for_right := matrixChain(mid+1, right)
    operation_for_left_and_right := row[begin] * column[mid] * column[end]
    total := operation_for_left + operation_for_right + operation_for_left_and_right
    answer := min(total, answer)
end for
dp[begin][end] := answer
Return dp[begin][end]
```

Complexity:

The value of **begin** and **end** can range from **1** to **n**. There are **n^2** different states. For each states, the loop inside will run **n** times. Total time complexity: `O(n³)` and memory complexity: `O(n²)`.

Read Matrix Chain Multiplication online: https://riptutorial.com/dynamic-programming/topic/7996/matrix-chain-multiplication

Chapter 6: Rod Cutting

Examples

Cutting the Rod to get the maximum profit

Given a rod of length **n** inches and an array of length **m** of prices that contains prices of all pieces of size smaller than n. We have to find the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is **8** and the values of different pieces are given as following, then the maximum obtainable value is **22**.

```
       +---+---+---+---+---+---+---+---+
(price)| 1 | 5 | 8 | 9 | 10| 17| 17| 20|
        +---+---+---+---+---+---+---+---+
```

We'll use a 2D array **dp[m][n + 1]** where n is the length of the rod and m is the length of the price array. For our example, we'll need **dp[8][9]**. Here **dp[i][j]** will denote the maximum price by selling the rod of length j.We can have the maximum value of length j as a whole or we could have broken the length to maximize the profit.

At first, for the 0th column, it will not contribute anything hence marking all the values as 0. So all the values of 0th column will be 0. For **dp[0][1]**, what is the maximum value we can get by selling rod of length 1.It will be 1.Similarly for rod of length 2 dp[0][2] we can have 2(1+1).This continues till **dp[0][8]**.So after the first iteration our dp[] array will look like.

```
      +---+---+---+---+---+---+---+---+---+
(price)| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
        +---+---+---+---+---+---+---+---+---+
 (1) 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
        +---+---+---+---+---+---+---+---+---+
 (5) 2 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
 (8) 3 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
 (9) 4 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
(10) 5 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
(17) 6 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
(17) 7 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
(20) 8 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
```

For **dp[2][2]** we hae to ask ourselves that what is the best I can get if I break the rod in two pieces(1,1) or taking the rod as a whole(length=2).We can see that if I break the rod in two pieces the maximum profit I can make is 2 and if if I have the rod as a whole I can sell it for 5.After second iteration the dp[] array will look like:

```
    +---+---+---+---+---+---+---+---+---+
 (price)| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
        +---+---+---+---+---+---+---+---+---+
  (1) 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
        +---+---+---+---+---+---+---+---+---+
  (5) 2 | 0 | 1 | 5 | 6 | 10| 11| 15| 16| 20|
        +---+---+---+---+---+---+---+---+---+
  (8) 3 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
  (9) 4 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
 (10) 5 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
 (17) 6 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
 (17) 7 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
 (20) 8 | 0 |   |   |   |   |   |   |   |   |
        +---+---+---+---+---+---+---+---+---+
```

So to calculate dp[i][j] our formula will look like:

```
if j>=i
    dp[i][j] = Max(dp[i-1][j], price[i]+arr[i][j-i]);
else
    dp[i][j] = dp[i-1][j];
```

After the last iteration our dp[] array will look like

```
      +---+---+---+---+---+---+---+---+---+
 (price)| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
        +---+---+---+---+---+---+---+---+---+
  (1) 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
        +---+---+---+---+---+---+---+---+---+
  (5) 2 | 0 | 1 | 5 | 6 | 10| 11| 15| 16| 20|
        +---+---+---+---+---+---+---+---+---+
  (8) 3 | 0 | 1 | 5 | 8 | 10| 13| 16| 18| 21|
        +---+---+---+---+---+---+---+---+---+
  (9) 4 | 0 | 1 | 5 | 8 | 10| 13| 16| 18| 21|
        +---+---+---+---+---+---+---+---+---+
 (10) 5 | 0 | 1 | 5 | 8 | 10| 13| 16| 18| 21|
        +---+---+---+---+---+---+---+---+---+
 (17) 6 | 0 | 1 | 5 | 8 | 10| 13| 17| 18| 22|
        +---+---+---+---+---+---+---+---+---+
 (17) 7 | 0 | 1 | 5 | 8 | 10| 13| 17| 18| 22|
        +---+---+---+---+---+---+---+---+---+
 (20) 8 | 0 | 1 | 5 | 8 | 10| 13| 17| 18| 22|
        +---+---+---+---+---+---+---+---+---+
```

We will have the result at **dp[n][m+1]**.

Implementation in Java

```
public int getMaximumPrice(int price[],int n){
        int arr[][] = new int[n][price.length+1];

        for(int i=0;i<n;i++){
            for(int j=0;j<price.length+1;j++){
                if(j==0 || i==0)
                    arr[i][j] = 0;
                else if(j>=i){
                    arr[i][j] = Math.max(arr[i-1][j], price[i-1]+arr[i][j-i]);
                }else{
                    arr[i][j] = arr[i-1][j];
                }
            }
        }
        return arr[n-1][price.length];
    }
```

Time Complexity

```
O(n^2)
```

Read Rod Cutting online: https://riptutorial.com/dynamic-programming/topic/10486/rod-cutting

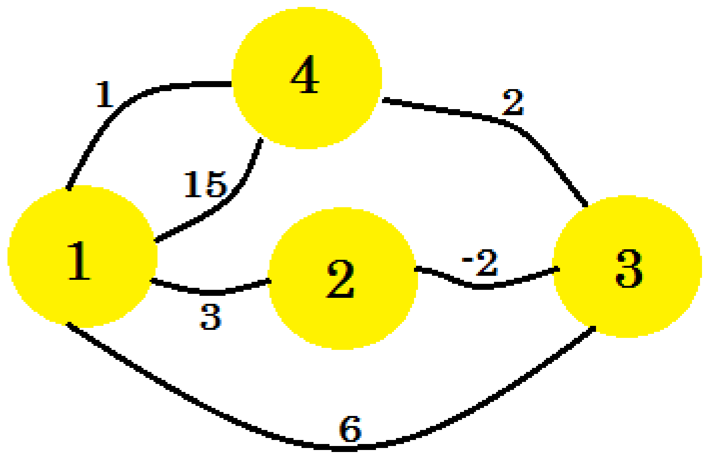
Chapter 7: Solving Graph Problems Using Dynamic Programming

Examples

Floyd-Warshall Algorithm

Floyd-Warshall's algorithm is for finding shortest paths in a weighted graph with positive or negative edge weights. A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pair of vertices. With a little variation, it can print the shortest path and can detect negative cycles in a graph. Floyd-Warshall is a Dynamic-Programming algorithm.

Let's look at an example. We're going to apply Floyd-Warshall's algorithm on this graph:



First thing we do is, we take two 2D matrices. These are adjacency matrices. The size of the matrices is going to be the total number of vertices. For our graph, we will take **4 * 4** matrices. The **Distance Matrix** is going to store the minimum distance found so far between two vertices. At first, for the edges, if there is an edge between **u-v** and the distance/weight is **w**, we'll store: `distance[u][v] = w`. For all the edges that doesn't exist, we're gonna put *infinity*. The **Path Matrix** is for regenerating minimum distance path between two vertices. So initially, if there is a path between **u** and **v**, we're going to put `path[u][v] = u`. This means the best way to come to **vertex-v** from **vertex-u** is to use the edge that connects **v** with **u**. If there is no path between two vertices, we're going to put **N** there indicating there is no path available now. The two tables for our graph will look like:

```
+-----+-----+-----+-----+-----+            +-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |            |     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+            +-----+-----+-----+-----+-----+
|  1  |  0  |  3  |  6  |  15 |            |  1  |  N  |  1  |  1  |  1  |
+-----+-----+-----+-----+-----+            +-----+-----+-----+-----+-----+
|  2  | inf |  0  | -2  | inf |            |  2  |  N  |  N  |  2  |  N  |
+-----+-----+-----+-----+-----+            +-----+-----+-----+-----+-----+
|  3  | inf | inf |  0  |  2  |            |  3  |  N  |  N  |  N  |  3  |
+-----+-----+-----+-----+-----+            +-----+-----+-----+-----+-----+
|  4  |  1  | inf | inf |  0  |           |  4  |  4  |  N  |  N  |  N  |
+-----+-----+-----+-----+-----+           +-----+-----+-----+-----+-----+
            distance                                     path
```

Since there is no loop, the diagonals are set **N**. And the distance from the vertex itself is **0**.

To apply Floyd-Warshall algorithm, we're going to select a middle vertex **k**. Then for each vertex **i**, we're going to check if we can go from **i** to **k** and then **k** to **j**, where **j** is another vertex and minimize the cost of going from **i** to **j**. If the current **distance[i][j]** is greater than **distance[i][k]** + **distance[k][j]**, we're going to put **distance[i][j]** equals to the summation of those two distances. And the **path[i][j]** will be set to **path[k][j]**, as it is better to go from **i** to **k**, and then **k** to **j**. All the vertices will be selected as **k**. We'll have 3 nested loops: for **k** going from 1 to 4, **i** going from 1 to 4 and **j** going from 1 to 4. We're going check:

```
if distance[i][j] > distance[i][k] + distance[k][j]
    distance[i][j] := distance[i][k] + distance[k][j]
    path[i][j] := path[k][j]
end if
```

So what we're basically checking is, *for every pair of vertices, do we get a shorter distance by going through another vertex?* The total number of operations for our graph will be **4 * 4 * 4** = **64**. That means we're going to do this check **64** times. Let's look at a few of them:

When **k** = **1**, **i** = **2** and **j** = **3**, **distance[i][j]** is **-2**, which is not greater than **distance[i][k]** + **distance[k][j]** = **-2** + **0** = **-2**. So it will remain unchanged. Again, when **k** = **1**, **i** = **4** and **j** = **2**, **distance[i][j]** = **infinity**, which is greater than **distance[i][k]** + **distance[k][j]** = **1** + **3** = **4**. So we put **distance[i][j]** = **4**, and we put **path[i][j]** = **path[k][j]** = **1**. What this means is, to go from **vertex-4** to **vertex-2**, the path **4->1->2** is shorter than the existing path. This is how we populate both matrices. The calculation for each step is shown here. After making necessary changes, our matrices will look like:

```
+-----+-----+-----+-----+-----+           +-----+-----+-----+-----+-----+
|     |  1  |  2  |  3  |  4  |           |     |  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+-----+           +-----+-----+-----+-----+-----+
|  1  |  0  |  3  |  1  |  3  |           |  1  |  N  |  1  |  2  |  3  |
+-----+-----+-----+-----+-----+           +-----+-----+-----+-----+-----+
|  2  |  1  |  0  | -2  |  0  |           |  2  |  4  |  N  |  2  |  3  |
+-----+-----+-----+-----+-----+           +-----+-----+-----+-----+-----+
|  3  |  3  |  6  |  0  |  2  |           |  3  |  4  |  1  |  N  |  3  |
+-----+-----+-----+-----+-----+           +-----+-----+-----+-----+-----+
|  4  |  1  |  4  |  2  |  0  |           |  4  |  4  |  1  |  2  |  N  |
+-----+-----+-----+-----+-----+           +-----+-----+-----+-----+-----+
            distance                                     path
```

This is our shortest distance matrix. For example, the shortest distance from **1** to **4** is **3** and the shortest distance between **4** to **3** is **2**. Our pseudo-code will be:

```
Procedure Floyd-Warshall(Graph):
for k from 1 to V     // V denotes the number of vertex
    for i from 1 to V
       for j from 1 to V
           if distance[i][j] > distance[i][k] + distance[k][j]
```

```
                distance[i][j] := distance[i][k] + distance[k][j]
                path[i][j] := path[k][j]
            end if
        end for
    end for
end for
```

Printing the path:

To print the path, we'll check the **Path** matrix. To print the path from **u** to **v**, we'll start from **path[u][v]**. We'll set keep changing **v** = **path[u][v]** until we find **path[u][v]** = **u** and push every values of **path[u][v]** in a stack. After finding **u**, we'll print **u** and start popping items from the stack and print them. This works because the **path** matrix stores the value of the vertex which shares the shortest path to **v** from any other node. The pseudo-code will be:

```
Procedure PrintPath(source, destination):
s = Stack()
S.push(destination)
while Path[source][destination] is not equal to source
    S.push(Path[source][destination])
    destination := Path[source][destination]
end while
print -> source
while S is not empty
    print -> S.pop
end while
```

Finding Negative Edge Cycle:

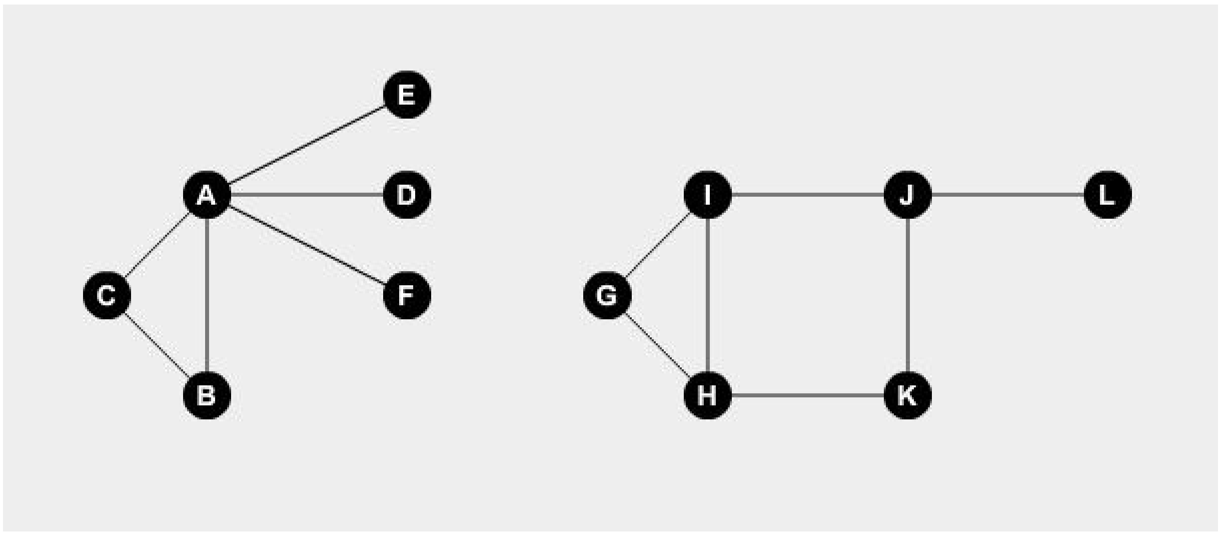
To find out if there is a negative edge cycle, we'll need to check the main diagonal of **distance** matrix. If any value on the diagonal is negative, that means there is a negative cycle in the graph.

Complexity:

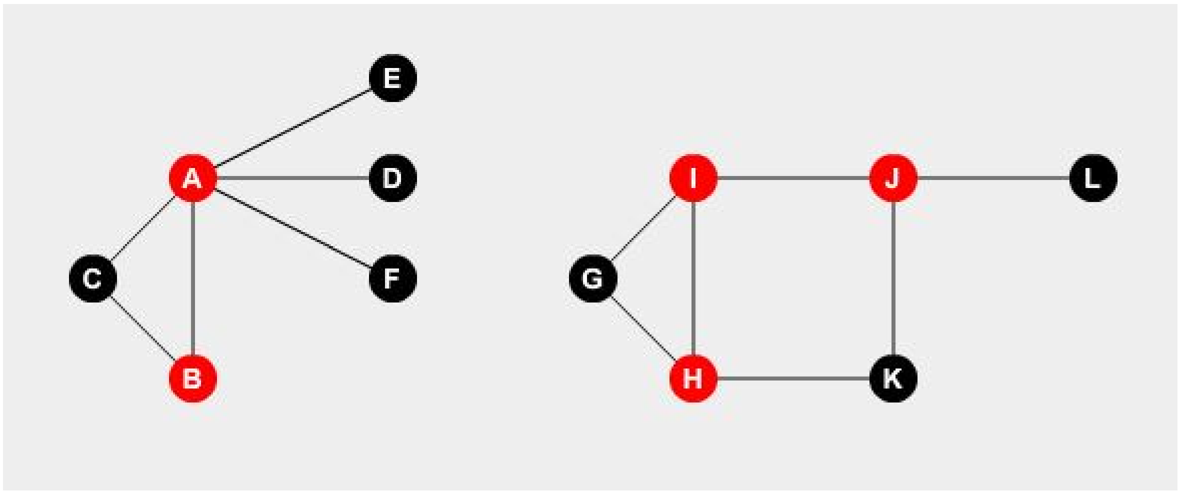
The complexity of Floyd-Warshall algorithm is **O(V^3)** and the space complexity is: **O(V^2)**.

Minimum Vertex Cover

Minimum Vertex Cover is a classic graph problem. Let's say, in a city we have a few roads connecting a few points. Let's represent the roads using edges and the points using nodes. Let's take two example graphs:

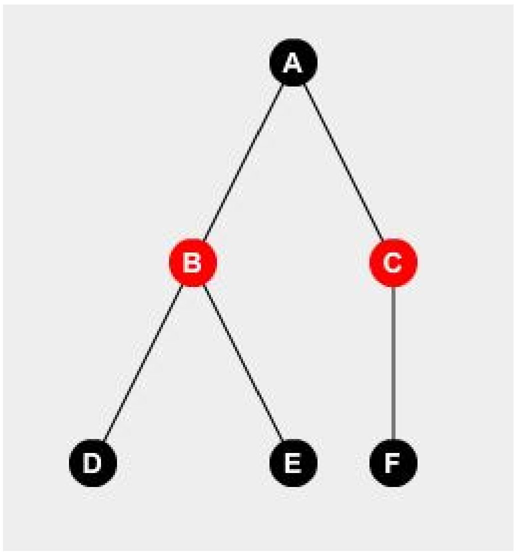


We want to set watchmen on some points. A watchman can guard all the roads connected to the point. The problem is, what is the minimum number of watchmen needed to cover all the roads? If we set watchmen at node **A**, **B**, **H**, **I** and **J**, we can cover all the roads.



This is our optimal solution. We need at least **5** watchmen to guard the whole city. How to determine this?

At first, we need to understand this is an *NP-hard* problem, i.e. the problem has no polynomial time solution. But if the graph was a **Tree**, that means if it had **(n-1)** nodes where **n** is the number of edges and there are no cycle in the graph, we can solve it using dynamic programming.



To construct a DP solution, we need to follow two strategies:

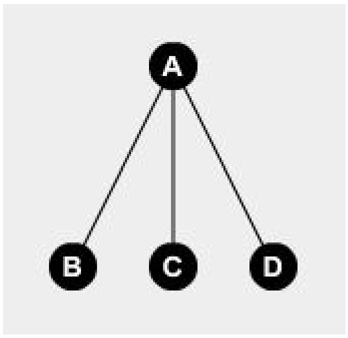
1. If there is no watchman in a node, all the nodes connected to it must have a watchman, or all the roads won't be covered. If **u** and **v** are connected and **u** doesn't have any watchman, then **v** must have a watchman.
2. If there is a watchman in a node, a different node connected to it may or may not have a watchman. That means it is not necessary to have a watchman, but it can be beneficial. If **u** and **v** are connected and **u** has a watchman, we'll check and find which on is beneficial for us by:
   - Setting watchman in **v**.
   - Not setting watchman in **v**.

Let's define a recursive function with state being the current node we're in and whether it has a watchman or not. Here:

```
F(u,1) = Currently we're in 'u' node and there is a watchman in this node.
F(u,0) = Currently we're in 'u' node and there is no watchman in this node.
```

The function will return the number of watchman in remaining nodes.

Let's take an example tree:



We can easily say that if we don't put watchman on node-**A**, we'll have to put watchmen on node-**B**, **C** and **D**. We can deduce:

```
F(A,0) = F(B,1) + F(C,1) + F(D,1) + 0
```

It returns us the number of watchmen needed if we don't put watchman in node-**A**. We've added **0** at the end because we didn't set any watchman in our current node.

Now `F(A,1)` means, we set watchman in node-**A**. For that, we can either set watchmen in all the connected nodes or we don't. We'll take the one that provides us with minimum number of watchmen.

```
F(A,1) = min(F(B,0), F(B,1) + min(F(C,0), F(C,1)) + min(F(D,0), F(D,1)) + 1
```

We check by setting and not setting watchman on each node and taking the optimal value.

One thing we must be careful that is, once we go to the child node, we'll never look back to the parent node. From the example above, we went to **B** from **A**, so **parent[B]** = **A**. So we'll not go back to **A** from **B**.

To determine base case, we can notice that, if from a node, we can't go to any other new node, we'll return **1** if there is a watchman in our current node, **0** if there is no watchman in our current node.

It is better to have a adjacency list for our tree. Let the list be denoted by **edge**. We'll have an array **dp[n][2]**, where **n** denotes the number of nodes to store the calculated values and initialize it with **-1**. We'll also have a **parent[n]** array to denote the parent and child relation between nodes. Our pseudo-code will look like:

```
Procedure f(u, isGuarded):
if edge[u].size is equal to 0                    //node doesn't have any new edge
    Return isGuarded
else if dp[u][isGuarded] is not equal to -1      //already calculated
    Return dp[u][isGuarded]
end if
sum := 0
for i from i to edge[u].size
    v := edge[u][i]
    if v is not equal to parent[u]               //not a parent
        parent[v] := u
        if isGuarded equals to 0                 //not guarded, must set a watchman
            sum := sum + f(v,1)
        else                                     //guarded, check both
            sum := sum + min(f(v,1), f(v,0)
        end if
    end if
end for
dp[u][isGuarded] := sum + isGuarded
Return dp[u][isGuarded]
```

If we denote node-**A** as root, we'll call the function by: `min(f(A,1), f(A,0))`. That means we'll also check if it is optimal to set watchman in the root node or not. This is our DP solution. This problem can also be solved using maximum matching algorithm or max-flow.

Read Solving Graph Problems Using Dynamic Programming online:

https://riptutorial.com/dynamic-programming/topic/7979/solving-graph-problems-using-dynamic-programming

Chapter 8: Subsequence Related Algorithms

Examples

Longest Common Subsequence

One of the most important implementations of Dynamic Programming is finding out the Longest Common Subsequence. Let's define some of the basic terminologies first.

Subsequence:

A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. Let's say we have a string **ABC**. If we erase zero or one or more than one character from this string we get the subsequence of this string. So the subsequences of string **ABC** will be {**"A"**, **"B"**, **"C"**, **"AB"**, **"AC"**, **"BC"**, **"ABC"**, **" "** }. Even if we remove all the characters, the empty string will also be a subsequence. To find out the subsequence, for each characters in a string, we have two options - either we take the character, or we don't. So if the length of the string is **n**, there are **2^n** subsequences of that string.

Longest Common Subsequence:

As the name suggest, of all the common subsequencesbetween two strings, the longest common subsequence(LCS) is the one with the maximum length. For example: The common subsequences between **"HELLOM"** and **"HMLD"** are **"H"**, **"HL"**, **"HM"** etc. Here **"HLL"** is the longest common subsequence which has length 3.

Brute-Force Method:

We can generate all the subsequences of two strings using *backtracking*. Then we can compare them to find out the common subsequences. After we'll need to find out the one with the maximum length. We have already seen that, there are **2^n** subsequences of a string of length **n**. It would take years to solve the problem if our **n** crosses **20-25**.

Dynamic Programming Method:

Let's approach our method with an example. Assume that, we have two strings **abcdaf** and **acbcf**. Let's denote these with **s1** and **s2**. So the longest common subsequence of these two strings will be **"abcf"**, which has length 4. Again I remind you, subsequences need not be continuous in the string. To construct **"abcf"**, we ignored **"da"** in **s1** and **"c"** in **s2**. How do we find this out using Dynamic Programming?

We'll start with a table (a 2D array) having all the characters of **s1** in a row and all the characters of **s2** in column. Here the table is 0-indexed and we put the characters from 1 to onwards. We'll traverse the table from left to right for each row. Our table will look like:

```
              0     1     2     3     4     5     6
     +-----+-----+-----+-----+-----+-----+-----+-----+
     | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  0  |     |     |     |     |     |     |     |     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  1  |  a  |     |     |     |     |     |     |     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  2  |  c  |     |     |     |     |     |     |     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  3  |  b  |     |     |     |     |     |     |     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  4  |  c  |     |     |     |     |     |     |     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  5  |  f  |     |     |     |     |     |     |     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
```

Here each row and column represent the length of the longest common subsequence between two strings if we take the characters of that row and column and add to the prefix before it. For example: **Table[2][3]** represents the length of the longest common subsequence between **"ac"** and **"abc"**.

The 0-th column represents the empty subsequence of **s1**. Similarly the 0-th row represents the empty subsequence of **s2**. If we take an empty subsequence of a string and try to match it with another string, no matter how long the length of the second substring is, the common subsequence will have 0 length. So we can fill-up the 0-th rows and 0-th columns with 0's. We get:

```
              0     1     2     3     4     5     6
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 1  |  a  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 2  |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 3  |  b  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 4  |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 5  |  f  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
```

Let's begin. When we're filling **Table[1][1]**, we're asking ourselves, if we had a string **a** and another string **a** and nothing else, what will be the longest common subsequence here? The length of the LCS here will be 1. Now let's look at **Table[1][2]**. We have string **ab** and string **a**. The length of the LCS will be 1. As you can see, the rest of the values will be also 1 for the first row as it considers only string **a** with **abcd**, **abcda**, **abcdaf**. So our table will look like:

```
              0     1     2     3     4     5     6
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 2  |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 3  |  b  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 4  |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 5  |  f  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
```

For row 2, which will now include **c**. For **Table[2][1]** we have **ac** on one side and **a** on the other side. So the length of the LCS is 1. Where did we get this 1 from? From the top, which denotes the LCS **a** between two substrings. So what we are saying is, if **s1[2]** and **s2[1]** are not same, then the length of the LCS will be the maximum of the length of LCS at the **top**, or at the **left**. Taking the length of the LCS at the top denotes that, we don't take the current character from **s2**. Similarly, Taking the length of the LCS at the left denotes that, we don't take the current character from **s1** to create the LCS. We get:

```
             0     1     2     3     4     5     6
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 2  |  c  |  0  |  1  |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 3  |  b  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 4  |  c  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
 5  |  f  |  0  |     |     |     |     |     |     |
    +-----+-----+-----+-----+-----+-----+-----+-----+
```

So our first formula will be:

```
if s2[i] is not equal to s1[j]
    Table[i][j] = max(Table[i-1][j], Table[i][j-1]
endif
```

Moving on, for **Table[2][2]** we have string **ab** and **ac**. Since **c** and **b** are not same, we put the maximum of the top or left here. In this case, it's again 1. After that, for **Table[2][3]** we have string **abc** and **ac**. This time current values of both row and column are same. Now the length of the LCS will be equal to the maximum length of LCS so far + 1. How do we get the maximum length of LCS so far? We check the diagonal value, which represents the best match between **ab** and **a**. From this state, for the current values, we added one more character to **s1** and **s2** which happened to be the same. So the length of LCS will of course increase. We'll put **1 + 1 = 2** in **Table[2][3]**. We get,

```
             0     1     2     3     4     5     6
    +-----+-----+-----+-----+-----+-----+-----+-----+
    | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  2  |  c  |  0  |  1  |  1  |  2  |     |     |     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  3  |  b  |  0  |     |     |     |     |     |     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  4  |  c  |  0  |     |     |     |     |     |     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  5  |  f  |  0  |     |     |     |     |     |     |
     +-----+-----+-----+-----+-----+-----+-----+-----+
```

So our second formula will be:

```
if s2[i] equals to s1[j]
    Table[i][j] = Table[i-1][j-1] + 1
endif
```

We have defined both the cases. Using these two formulas, we can populate the whole table. After filling up the table, it will look like this:

```
              0     1     2     3     4     5     6
     +-----+-----+-----+-----+-----+-----+-----+-----+
     | ch  |     |  a  |  b  |  c  |  d  |  a  |  f  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  0  |     |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  1  |  a  |  0  |  1  |  1  |  1  |  1  |  1  |  1  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  2  |  c  |  0  |  1  |  1  |  2  |  2  |  2  |  2  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  3  |  b  |  0  |  1  |  2  |  2  |  2  |  2  |  2  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  4  |  c  |  0  |  1  |  2  |  3  |  3  |  3  |  3  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
  5  |  f  |  0  |  1  |  2  |  3  |  3  |  3  |  4  |
     +-----+-----+-----+-----+-----+-----+-----+-----+
```

The length of the LCS between **s1** and **s2** will be **Table[5][6] = 4**. Here, 5 and 6 are the length of **s2** and **s1** respectively. Our pseudo-code will be:

```
Procedure LCSlength(s1, s2):
Table[0][0] = 0
for i from 1 to s1.length
    Table[0][i] = 0
endfor
for i from 1 to s2.length
    Table[i][0] = 0
endfor
for i from 1 to s2.length
    for j from 1 to s1.length
        if s2[i] equals to s1[j]
            Table[i][j] = Table[i-1][j-1] + 1
        else
            Table[i][j] = max(Table[i-1][j], Table[i][j-1])
        endif
    endfor
endfor
Return Table[s2.length][s1.length]
```

The time complexity for this algorithm is: **O(mn)** where **m** and **n** denotes the length of each strings.

How do we find out the longest common subsequence? We'll start from the bottom-right corner. We will check from where the value is coming. If the value is coming from the diagonal, that is if **Table[i-1][j-1]** is equal to **Table[i][j] - 1**, we push either **s2[i]** or **s1[j]** (both are the same) and move diagonally. If the value is coming from top, that means, if **Table[i-1][j]** is equal to **Table[i][j]**, we move to the top. If the value is coming from left, that means, if **Table[i][j-1]** is equal to **Table[i][j]**, we move to the left. When we reach the leftmost or topmost column, our search ends. Then we pop the values from the stack and print them. The pseudo-code:

```
Procedure PrintLCS(LCSlength, s1, s2)
temp := LCSlength
S = stack()
i := s2.length
j := s1.length
while i is not equal to 0 and j is not equal to 0
     if Table[i-1][j-1] == Table[i][j] - 1 and s1[j]==s2[i]
        S.push(s1[j])   //or S.push(s2[i])
        i := i - 1
        j := j - 1
    else if Table[i-1][j] == Table[i][j]
        i := i-1
    else
        j := j-1
    endif
endwhile
while S is not empty
    print(S.pop)
endwhile
```

Point to be noted: if both **Table[i-1][j]** and **Table[i][j-1]** is equal to **Table[i][j]** and **Table[i-1][j-1]** is not equal to **Table[i][j] - 1**, there can be two LCS for that moment. This pseudo-code doesn't consider this situation. You'll have to solve this recursively to find multiple LCSs.

The time complexity for this algorithm is: **O(max(m, n))**.

Longest Increasing Subsequence

The task is to find the length of the longest subsequence in a given array of integers such that all elements of the subsequence are sorted in ascending order. For example, the length of the longest increasing subsequence(LIS) for **{15, 27, 14, 38, 26, 55, 46, 65, 85}** is **6** and the longest increasing subsequence is **{15, 27, 38, 55, 65, 85}**. Again for **{3, 4, -1, 0, 6, 2, 3}** length of LIS is **4** and the subsequence is **{-1, 0, 2, 3}**. We'll use dynamic programming to solve this problem.

Let's take the second example: `Item = {3, 4, -1, 0, 6, 2, 3}`. We'll start by taking the an array **dp** of the same size of our sequence. **dp[i]** represents the length of the LIS if we include the **i**th item of our original sequence. At the very beginning we know that for each and every item at least the longest increasing subsequence is of length **1**. That is by considering the single element itself. So we'll initialize the **dp** array with **1**. We'll have two variables **i** and **j**. Initially **i** will be **1** and **j** will be **0**. Our array will look like:

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  1  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
           j     i
```

The number above the array represents the corresponding element of our sequence. Our strategy will be:

```
if Item[i] > Item[j]
    dp[i] := dp[j] + 1
```

That means if element at **i** is greater than element at **j**, the length of the LIS that contains element at **j**, will increase by length **1** if we include element at **i** with it. In our example, for **i** = **1** and **j** = **0**, **Item[i]** is greater than **Item[j]**. So **dp[i]** = **dp[j]** + **1**. Our array will look like:

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
           j     i
```

At each step, we'll increase **j** up to **i** and then reset **j** to **0** and increment **i**. For now, **j** has reached **i** , so we increment **i** to **2**.

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
           j           i
```

For **i** = **2**, **j** = **0**, **Item[i]** is not greater than **Item[j]**, so we do nothing and increment **j**.

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                 j     i
```

For **i** = **2**, **j** = **1**, **Item[i]** is not greater than **Item[j]**, so we do nothing and since **j** has reached its limit, we increment **i** and reset **j** to **0**.

```
          3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
           j              i
```

For **i** = **3**, **j** = **0**, **Item[i]** is not greater than **Item[j]**, so we do nothing and increment **j**.

```
          3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                 j          i
```

For **i** = **3**, **j** = **1**, **Item[i]** is not greater than **Item[j]**, so we do nothing and increment **j**.

```
          3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                       j     i
```

For **i** = **3**, **j** = **2**, **Item[i]** is greater than **Item[j]**, so **dp[i]** = **dp[j]** + **1**. After that since **j** has reached its limit, again we reset **j** to **0** and increment **i**.

```
          3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  2  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
           j                    i
```

For **i** = **4** and **j** = **0**, **Item[i]** is greater than **Item[j]**, so **dp[i]** = **dp[j]** + **1**. After that, we increment **j**.

```
          3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  2  |  2  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                 j              i
```

For **i** = **4** and **j** = **1**, **Item[i]** is greater than **Item[j]**. We can also notice that **dp[i]** = **dp[j]** + **1** will provide us **3**, which means if we take the LIS for **Item[j]** and add **Item[i]** with it, we'll get a better

LIS{3,4,6} than before {3,6}. So we set **dp[i]** = **dp[j]** + **1**. Then we increment **j**.

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  2  |  3  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                       j           i
```

For **i** = **4** and **j** = **2**, **Item[i]** is greater than **Item[j]**. But for this case, if we set **dp[i]** = **dp[j]** + **1**, we'll get **2**, which is{-1,6} not the best{3,4,6} we can do using **Item[i]**. So we discard this one. We'll add a condition to our strategy, that is:

```
if Item[i]>Item[j] and dp[i]<dp[j] + 1
    dp[i] := dp[j] + 1
```

We increment **j** by **1**. Following this strategy, if we complete our **dp** array, it will look like:

```
           3     4    -1     0     6     2     3
+-------+-----+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
+-------+-----+-----+-----+-----+-----+-----+-----+
| Value |  1  |  2  |  1  |  2  |  3  |  3  |  4  |
+-------+-----+-----+-----+-----+-----+-----+-----+
                                         j     i
```

Now we'll iterate through this array and find out the maximum value, which will be the length of the LIS. Our pseudo-code will be:

```
Procedure LISLength(Item):
n := Item.length
dp[] := new Array(n)
for i from 0  to n
    dp[i] := 1
end for
for i from 1 to n
    for j from 0 to i
        if Item[i]>Item[j] and dp[i]<dp[j] + 1
            dp[i] := dp[j] + 1
        end if
    end for
end for
l := -1
for i from 0 to n
    l := max(l, dp[i])
end for
Return l
```

The time complexity of this algorithm is `O(n²)` where **n** is the length of the sequence.

To find out the original sequence, we need to iterate backwards and match it with our length. The procedure is:

```
Procedure LIS(Item, dp, maxLength):
i := Item.length
while dp[i] is not equal to maxLength
    i := i - 1
end while
s = new Stack()
s.push(Item[i])
maxLength := maxLength - 1
current := Item[i]
while maxLength is not equal to 0
    i := i-1
    if dp[i] := maxLength and Item[i] < current
        current := Item[i]
        s.push(current)
        maxLength := maxLength - 1
    end if
end while
while s is not empty
    x := s.pop
    Print(s)
end while
```

The time complexity of this algorithm is: `O(n)`.

Longest Palindromic Subsequence

Given a string what is the longest palindromic subsequence(LPS) of it? Let's take a string **agbdba**. The LPS of this string is **abdba** of length **5**. Remember, since we're looking for *subsequence*, the characters need not to be continuous in the original string. The longest palindromic *substring* of the sequence would be **bdb** of length **3**. But we'll concentrate on the *subsequence* here. We're going to use dynamic programming to solve this problem.

At first, we'll take a 2D array of the same dimension of our original sequence. For our example: `S = "agbdba"`, we'll take **dp[6][6]** array. Here, **dp[i][j]** represents the length of the LPS we can make if we consider the characters from **s[i]** to **s[j]**. For example. if our string was **aa**, **dp[0][1]** would store **2**. Now we'll consider different lengths of our string and find out the longest possible length we can make out of it.

Length = 1:
Here, we are considering only **1** character at a time. So if we had a string of length **1**, what is the LPS we can have? Of course the answer is **1**. How to store it? **dp[i][j]** where **i** is equal to **j** represents a string of length **1**. So we'll set **dp[0][0]**, **dp[1][1]**, **dp[2][2]**, **dp[3][3]**, **dp[4][4]**, **dp[5][5]** to **1**. Our array will look like:

```
+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
| 0 | 1 |   |   |   |   |   |
+---+---+---+---+---+---+---+
| 1 |   | 1 |   |   |   |   |
+---+---+---+---+---+---+---+
| 2 |   |   | 1 |   |   |   |
+---+---+---+---+---+---+---+
| 3 |   |   |   | 1 |   |   |
+---+---+---+---+---+---+---+
| 4 |   |   |   |   | 1 |   |
+---+---+---+---+---+---+---+
| 5 |   |   |   |   |   | 1 |
+---+---+---+---+---+---+---+
```

Length = 2:
This time we'll consider strings of length **2**. Now considering strings of length **2**, the maximum length of LPS can be **2** if and only if the two characters of the string is same. So our strategy will be:

```
j := i + Length - 1
if s[i] is equal to s[j]
    dp[i][j] := 2
else
    dp[i][j] := 1
```

If we fill our array following the strategy for **Length** = **2**, we'll get:

```
+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
| 0 | 1 | 1 |   |   |   |   |
+---+---+---+---+---+---+---+
| 1 |   | 1 | 1 |   |   |   |
+---+---+---+---+---+---+---+
| 2 |   |   | 1 | 1 |   |   |
+---+---+---+---+---+---+---+
| 3 |   |   |   | 1 | 1 |   |
+---+---+---+---+---+---+---+
| 4 |   |   |   |   | 1 | 1 |
+---+---+---+---+---+---+---+
| 5 |   |   |   |   |   | 1 |
+---+---+---+---+---+---+---+
```

Length = 3:
Now we're looking at **3** characters at a time for our original string. From now on the LPS we can make from our string will be determined by:

- If the first and last characters match we'll have at least **2** items from which we can make the LPS + if we exclude the first and last character, what ever the best we can make from the remaining string.
- If the first and last characters do not match, the LPS we can make will come from either excluding the first character or the last character, which we've already calculated.

To summerize,

```
j := i + Length - 1
if s[i] is equal to s[j]
    dp[i][j] := 2 + dp[i+1][j-1]
else
    dp[i][j] := max(dp[i+1][j], dp[i][j-1])
end if
```

If we fill the **dp** array for **Length** = **3** to **Length** = **6**, we'll get:

```
+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
| 0 | 1 | 1 | 1 | 1 | 3 | 5 |
+---+---+---+---+---+---+---+
| 1 |   | 1 | 1 | 1 | 3 | 3 |
+---+---+---+---+---+---+---+
| 2 |   |   | 1 | 1 | 3 | 3 |
+---+---+---+---+---+---+---+
| 3 |   |   |   | 1 | 1 | 1 |
+---+---+---+---+---+---+---+
| 4 |   |   |   |   | 1 | 1 |
+---+---+---+---+---+---+---+
| 5 |   |   |   |   |   | 1 |
+---+---+---+---+---+---+---+
```

This is our required **dp** array and **dp[0][5]** will contain the length of the LPS. Our procedure will look like:

```
Procedure LPSLength(S):
n := S.length
dp[n][n]
for i from 0 to n
    dp[i][i] := 1
end for
for i from 0 to (n-2)
    if S[i] := S[i+1]
        dp[i][i+1] := 2
    else
        dp[i][i+1] := 1
    end if
end for
Length := 3
while Length <= n
    for i from 0 to (n - Length)
        j := i + Length - 1
        if S[i] is equal to s[j]
            dp[i][j] := 2 + dp[i+1][j-1]
        else
            dp[i][j] := max(dp[i+1][j], dp[i][j-1])
        end if
    end for
Length := Length + 1
end while
Return dp[0][n-1]
```

The time complexity of this algorithm is `O(n²)`, where **n** is the length of our given string. Longest Palindromic Subsequence problem is closely related to Longest Common Subsequence. If we take the second string as the reverse of the first string and calculate the length and print the result, that will be the longest palindromic subsequence of the given string. The complexity of that algorithm is also `O(n²)`.

Read Subsequence Related Algorithms online: https://riptutorial.com/dynamic-programming/topic/7969/subsequence-related-algorithms

Chapter 9: Weighted Activity Selection

Examples

Weighted Job Scheduling Algorithm

Weighted Job Scheduling Algorithm can also be denoted as Weighted Activity Selection Algorithm.

The problem is, given certain jobs with their start time and end time, and a profit you make when you finish the job, what is the maximum profit you can make given no two jobs can be executed in parallel?

This one looks like Activity Selection using Greedy Algorithm, but there's an added twist. That is, instead of maximizing the number of jobs finished, we focus on making the maximum profit. The number of jobs performed doesn't matter here.

Let's look at an example:

```
+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    A    |    B    |    C    |    D    |    E    |    F    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (2,5)  |  (6,7)  |  (7,9)  |  (1,3)  |  (5,8)  |  (4,6)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    6    |    4    |    2    |    5    |    11   |    5    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

The jobs are denoted with a name, their start and finishing time and profit. After a few iterations, we can find out if we perform **Job-A** and **Job-E**, we can get the maximum profit of 17. Now how to find this out using an algorithm?

The first thing we do is sort the jobs by their finishing time in non-decreasing order. Why do we do this? It's because if we select a job that takes less time to finish, then we leave the most amount of time for choosing other jobs. We have:

```
+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

We'll have an additional temporary array **Acc_Prof** of size **n** (Here, **n** denotes the total number of jobs). This will contain the maximum accumulated profit of performing the jobs. Don't get it? Wait and watch. We'll initialize the values of the array with the profit of each jobs. That means, **Acc_Prof[i]** will at first hold the profit of performing **i-th** job.

```
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Now let's denote **position 2** with **i**, and **position 1** will be denoted with **j**. Our strategy will be to iterate **j** from **1** to **i-1** and after each iteration, we will increment **i** by 1, until **i** becomes **n+1**.

```
                               j        i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

We check if **Job[i]** and **Job[j]** overlap, that is, if the **finish time** of **Job[j]** is greater than **Job[i]**'s start time, then these two jobs can't be done together. However, if they don't overlap, we'll check if **Acc_Prof[j] + Profit[i] > Acc_Prof[i].** If this is the case, we will update **Acc_Prof[i] = Acc_Prof[j] + Profit[i]**. That is:

```
if Job[j].finish_time <= Job[i].start_time
    if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
        Acc_Prof[i] = Acc_Prof[j] + Profit[i]
    endif
endif
```

Here **Acc_Prof[j] + Profit[i]** represents the accumulated profit of doing these two jobs toegther. Let's check it for our example:

Here **Job[j]** overlaps with **Job[i]**. So these to can't be done together. Since our **j** is equal to **i-1**, we increment the value of **i** to **i+1** that is **3**. And we make **j = 1**.

```
                               j                  i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |    5    |    4    |    11   |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Now **Job[j]** and **Job[i]** don't overlap. The total amount of profit we can make by picking these two jobs is: **Acc_Prof[j] + Profit[i] = 5 + 5 = 10** which is greater than **Acc_Prof[i]**. So we update **Acc_Prof[i] = 10**. We also increment **j** by 1. We get,

```
                                         j        i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |   11    |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |   10    |    4    |   11    |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Here, **Job[j]** overlaps with **Job[i]** and **j** is also equal to **i-1**. So we increment **i** by 1, and make **j = 1** . We get,

```
                               j                             i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |   11    |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |   10    |    4    |   11    |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Now, **Job[j]** and **Job[i]** don't overlap, we get the accumulated profit **5 + 4 = 9**, which is greater than **Acc_Prof[i]**. We update **Acc_Prof[i] = 9** and increment **j** by 1.

```
                                         j                   i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |   11    |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |   10    |    9    |   11    |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

Again **Job[j]** and **Job[i]** don't overlap. The accumulated profit is: **6 + 4 = 10**, which is greater than **Acc_Prof[i]**. We again update **Acc_Prof[i] = 10**. We increment **j** by 1. We get:

```
                                                   j         i

+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |   11    |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |   10    |   10    |   11    |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

If we continue this process, after iterating through the whole table using **i**, our table will finally look like:

```
+-------------------------+---------+---------+---------+---------+---------+---------+
|          Name           |    D    |    A    |    F    |    B    |    E    |    C    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|(Start Time, Finish Time)|  (1,3)  |  (2,5)  |  (4,6)  |  (6,7)  |  (5,8)  |  (7,9)  |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Profit          |    5    |    6    |    5    |    4    |   11    |    2    |
+-------------------------+---------+---------+---------+---------+---------+---------+
|         Acc_Prof        |    5    |    6    |   10    |   14    |   17    |    8    |
+-------------------------+---------+---------+---------+---------+---------+---------+
```

* A few steps have been skipped to make the document shorter.

If we iterate through the array **Acc_Prof**, we can find out the maximum profit to be **17**! The pseudo-code:

```
Procedure WeightedJobScheduling(Job)
sort Job according to finish time in non-decreasing order
for i -> 2 to n
    for j -> 1 to i-1
        if Job[j].finish_time <= Job[i].start_time
            if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
                Acc_Prof[i] = Acc_Prof[j] + Profit[i]
            endif
        endif
    endfor
endfor

maxProfit = 0
for i -> 1 to n
    if maxProfit < Acc_Prof[i]
        maxProfit = Acc_Prof[i]
return maxProfit
```

The complexity of populating the **Acc_Prof** array is **O(n^2).** The array traversal takes **O(n)**. So the total complexity of this algorithm is **O(n^2).**

Now, If we want to find out which jobs were performed to get the maximum profit, we need to traverse the array in reverse order and if the **Acc_Prof** matches the **maxProfit**, we will push the **name** of the job in a **stack** and subtract **Profit** of that job from **maxProfit**. We will do this until our **maxProfit > 0** or we reach the beginning point of the **Acc_Prof** array. The pseudo-code will look like:

```
Procedure FindingPerformedJobs(Job, Acc_Prof, maxProfit):
S = stack()
for i -> n down to 0 and maxProfit > 0
    if maxProfit is equal to Acc_Prof[i]
        S.push(Job[i].name
        maxProfit = maxProfit - Job[i].profit
    endif
endfor
```

The complexity of this procedure is: **O(n)**.

One thing to remember, if there are multiple job schedules that can give us maximum profit, we can only find one job schedule via this procedure.

Read Weighted Activity Selection online: https://riptutorial.com/dynamic-programming/topic/7999/weighted-activity-selection

Credits

S. No	Chapters	Contributors
1	Getting started with dynamic-programming	Bakhtiar Hasan, Community
2	Coin Changing Problem	Bakhtiar Hasan
3	Dynamic Time Warping	Bakhtiar Hasan
4	Knapsack Problem	Bakhtiar Hasan
5	Matrix Chain Multiplication	Bakhtiar Hasan
6	Rod Cutting	metahost, Vishwas
7	Solving Graph Problems Using Dynamic Programming	Bakhtiar Hasan
8	Subsequence Related Algorithms	Bakhtiar Hasan
9	Weighted Activity Selection	Bakhtiar Hasan