Chapter 14: Dynamic Programming

Dynamic programming is a widely used concept and its often used for optimization. It refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner usually a bottom-up approach. There are two key attributes that a problem must have in order for dynamic programming to be applicable "Optimal substructure" and "Overlapping sub-problems". To achieve its optimization, dynamic programming uses a concept called memoization

Section 14.1: Edit Distance

The problem statement is like if we are given two string str1 and str2 then how many minimum number of operations can be performed on the str1 that it gets converted to str2.

Implementation in Java

```
public class EditDistance {

public static void main(String[] args) {
    // TODO Auto-generated method stub
    String str1 = "march";
    String str2 = "cart";

    EditDistance ed = new EditDistance();
    System.out.println(ed.getMinConversions(str1, str2));
}

public int getMinConversions(String str1, String str2){
    int dp[][] = new int[str1.length()+1][str2.length()+1];
    for(int i=0;i<=str1.length();i++){
        for(int j=0;j<=str2.length();j++){
            if(i==0)
                dp[i][j] = j;
            else if(j==0)
                dp[i][j] = i;
            else if(str1.charAt(i-1) == str2.charAt(j-1))
                dp[i][j] = dp[i-1][j-1];
            else{
                dp[i][j] = 1 + Math.min(dp[i-1][j], Math.min(dp[i][j-1], dp[i-1][j-1]));
            }
        }
    }
    return dp[str1.length()][str2.length()];
}
```

}

Output

3

Section 14.2: 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(n2).** The array traversal takes **O(n)**. So the total complexity of this algorithm is **O(n2).**

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.

Section 14.3: Longest Common Subsequence

If we are given with the two strings we have to find the longest common sub-sequence present in both of them.

Example

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

Implementation in Java

```
public class LCS {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String str1 = "AGGTAB";
        String str2 = "GXTXAYB";
        LCS obj = new LCS();
        System.out.println(obj.lcs(str1, str2, str1.length(), str2.length()));
        System.out.println(obj.lcs2(str1, str2));
    }

    //Recursive function
    public int lcs(String str1, String str2, int m, int n){
```

```
        if(m==0 || n==0)
            return 0;
        if(str1.charAt(m-1) == str2.charAt(n-1))
            return 1 + lcs(str1, str2, m-1, n-1);
        else
            return Math.max(lcs(str1, str2, m-1, n), lcs(str1, str2, m, n-1));
    }

    //Iterative function
    public int lcs2(String str1, String str2){
        int lcs[][] = new int[str1.length()+1][str2.length()+1];

        for(int i=0;i<=str1.length();i++){
            for(int j=0;j<=str2.length();j++){
                if(i==0 || j== 0){
                    lcs[i][j] = 0;
                }
                else if(str1.charAt(i-1) == str2.charAt(j-1)){
                    lcs[i][j] = 1 + lcs[i-1][j-1];
                }else{
                    lcs[i][j] = Math.max(lcs[i-1][j], lcs[i][j-1]);
                }
            }
        }

        return lcs[str1.length()][str2.length()];
    }

}
```

Output

```
4
```

Section 14.4: Fibonacci Number

Bottom up approach for printing the nth Fibonacci number using Dynamic Programming.

Recursive Tree

```
                        fib(5)
                    /             \
               fib(4)               fib(3)
             /      \               /     \
         fib(3)      fib(2)        fib(2)    fib(1)
        /     \        /    \       /    \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

Overlapping Sub-problems

Here fib(0),fib(1) and fib(3) are the overlapping sub-problems.fib(0) is getting repeated 3 times, fib(1) is getting repeated 5 times and fib(3) is getting repeated 2 times.

Implementation

```
public int fib(int n){
        int f[] = new int[n+1];
```

```
        f[0]=0;f[1]=1;
        for(int i=2;i<=n;i++){
            f[i]=f[i-1]+f[i-2];
        }
        return f[n];
    }
```

Time Complexity

O(n)

Section 14.5: Longest Common Substring

Given 2 string str1 and str2 we have to find the length of the longest common substring between them.

Examples

Input : X = "abcdxyz", y = "xyzabcd" Output : 4

The longest common substring is "abcd" and is of length 4.

Input : X = "zxabcdezy", y = "yzabcdezx" Output : 6

The longest common substring is "abcdez" and is of length 6.

Implementation in Java

```
public int getLongestCommonSubstring(String str1,String str2){
        int arr[][] = new int[str2.length()+1][str1.length()+1];
        int max = Integer.MIN_VALUE;
        for(int i=1;i<=str2.length();i++){
            for(int j=1;j<=str1.length();j++){
                if(str1.charAt(j-1) == str2.charAt(i-1)){
                    arr[i][j] = arr[i-1][j-1]+1;
                    if(arr[i][j]>max)
                        max = arr[i][j];
                }
                else
                    arr[i][j] = 0;
            }
        }
        return max;
    }
```

Time Complexity

O(m*n)