Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 5 REPORT

YUNUS ÇEVİK
141044080

Course Assistant: Mehmet Burak KOCA

Part – 1

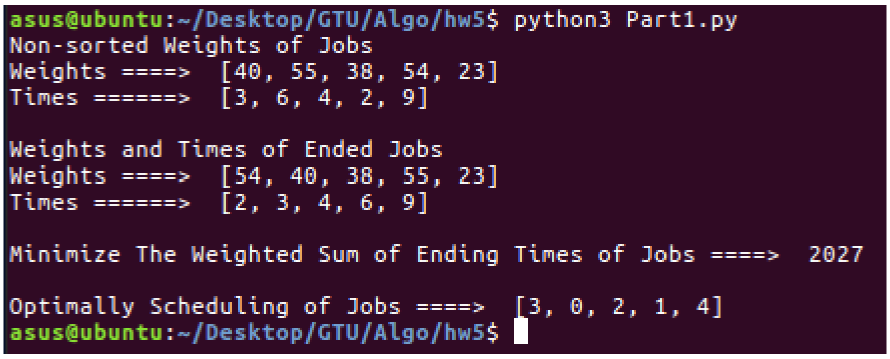
A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time. In part 1 question, A small business provides a photocopying service with a single large machine. This business is getting a big job from its customers and considering the satisfaction of its customers. It wants to sort jobs in a structure that minimizes the weighted total of the completion times. If all jobs have equal weight, this is exactly sorted in a non-decreasing order indefinitely, so it works faster. If they do not have equal time, their order of priority is given. It is first sorted by weights and time dependent $\frac{w_i}{t_i} < \frac{w_{i+1}}{t_{i+1}}$. Then the sum of these costs in a structure that minimizes this weighted sum occurs.

Complexity

When we look at the time complexity of the jobsScheduling function. First of all, we add values from 0 to the size of the array of weights into an empty array to determine the optimum times. This process takes T (n) = O (n) Linear Time.
The part that interests us is that we divide the weights into time and perform a sorting process based on the values. The sorting algorithm that can do this is "Bubble Sort".
With "Bubble Sort", the optimal timelines are set. Because this process is done with two For loops, and because it looks at all the array elements,
the Worst - Case: T (n) = O (n ^ 2) Quadratic Time works as complexity. However, if there is an ordered structure, Best - Case: T (n) = O (n) Linear Time works as complexity. Finally, the sorted weights and time sequences are collected by multiplying. However, the situation that should not be forgotten here should be collected from time to time until the second work is done. T (n) = O (n) Linear Time takes place in this function.

As a result, the complexity analysis of this function is T (n) = O (n) + O (n ^ 2) + O (n) ==> T (n) = O (n ^ 2) Quadratic Time works as complexity.

Result



```
asus@ubuntu:~/Desktop/GTU/Algo/hw5$ python3 Part1.py
Non-sorted Weights of Jobs
Weights ====>  [40, 55, 38, 54, 23]
Times ======>  [3, 6, 4, 2, 9]

Weights and Times of Ended Jobs
Weights ====>  [54, 40, 38, 55, 23]
Times ======>  [2, 3, 4, 6, 9]

Minimize The Weighted Sum of Ending Times of Jobs ====>  2027

Optimally Scheduling of Jobs ====>  [3, 0, 2, 1, 4]
asus@ubuntu:~/Desktop/GTU/Algo/hw5$
```

Part – 2

Part – 2.A

#	Month 1	Month 2	Month 3	Month 4
NY	1	3	20	30
SF	50	20	2	4

Suppose that M = 10,
If the pseudocode specified below is run with the values in the above table, it is for the person to go there which gives less cost in the city.
Then the minimum cost of the plan would be:

[NY, NY, SF, SF]

with a total cost of 1 + 3 + 2 + 4 + 10 = 20, where the final term of 10 arises because person changes locations once.

```
for i= 1 to n
    if Ni < Si then
        Output "NY in Month i"
    else
        Output "SF in Month i"
end
```

#	Month 1	Month 2	Month 3
NY	1	10	1
SF	50	2	50

However, if the person thinks of staying in the fixed city
1 + 10 + 1 = 12 ==> ['NY', 'NY', 'NY']
50 + 2 + 50 = 102 ==> ['SF', 'SF', 'SF']
If we look at the cost as NY <SF, it is optimal for the person to remain in NY. However, according to the above greedy algorithm we need to change the city as ['NY', 'SF', 'NY'].
1 + 10 + 2 + 10 + 1 = 24

optimal fixed city cost < optimally changing city cost

12 < 24

The above pseudocode will have worked incorrectly because of the given values.

Part – 2.B
Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to recomupute them when needed later. This situation makes easier some process. In part 2.B question, For dynamic programming, we need to keep the optimal arrays in which the cities of NY and SF are kept at each iteration cost. Displacement costs should be taken into account when adding elements to these arrays. Also, whichever of the two cities costs less, it should be added to the arrays.
The following pseudocode does this.

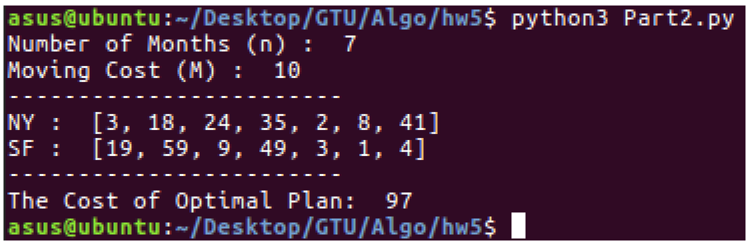
```
optN(0) = NY[0]
optS(0) = SF[0]
For i = 1, ….. , n
        optN(i) = N[i] + min(optN(i – 1), (M + optS(i – 1)))
        optS(i) = S[i] + min(optS(i – 1), (M + optN(i – 1)))
End
Return min(optN(n), optS(n))
```

As a result, the above pseudocode gives the minimum cost as a dynamic programming algorithm.

Complexity

T(n) = O(n) Linear Time works as coplexity until a single For loop will perform as much iterations as the number of elements of arrays.

Result



```
asus@ubuntu:~/Desktop/GTU/Algo/hw5$ python3 Part2.py
Number of Months (n) :  7
Moving Cost (M) :  10
------------------------
NY :  [3, 18, 24, 35, 2, 8, 41]
SF :  [19, 59, 9, 49, 3, 1, 4]
------------------------
The Cost of Optimal Plan:  97
asus@ubuntu:~/Desktop/GTU/Algo/hw5$
```