BUSRA CALISKAN 504211507 – ALGORITHM ANALYSIS-HOMEWORK 3

A-)

1.for availability_intervals.txt , create places and their dates and keep as map O(n log m) there are n lines in the input file.m is the number of elements in the map.

2.for daily_schedule.txt , create salons and their times and keep as map and put belonging places already created O(n * (log m + log k)) there are n lines in the input file.m is the number of elements in the map. k is the number of elements in the salons map

3.for capacity.txt, put capacity of salons already created O(n * log m) there are n lines in the input file.m is the number of elements in the map

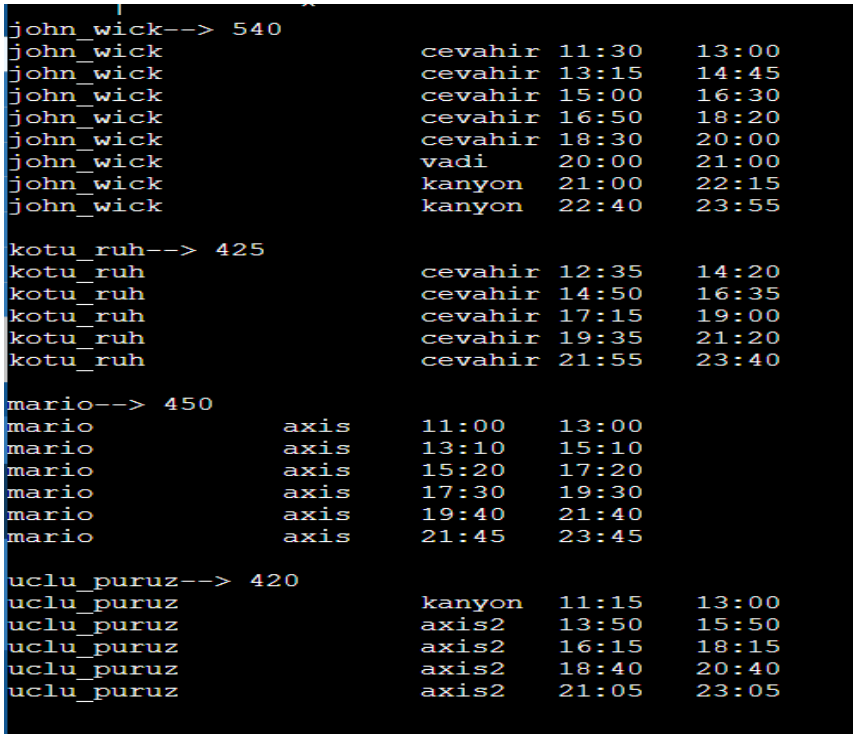
4.or assets.txt, create vector assest. O(n)

//It prints correct answer but spot of place name may be changed because used map str.

```
Cevahir_Salon--> 210
Cevahir_Salon           Salon_2 10:00   13:00
Cevahir_Salon           Salon_1 14:00   17:00
Cevahir_Salon           Salon_1 17:00   20:00

Torium_Sahne--> 235
Torium_Sahne            Salon_2 10:00   11:00
Torium_Sahne            Salon_1 11:00   12:00
Torium_Sahne            Salon_2 12:00   13:00
Torium_Sahne            Salon_3 13:00   14:00
Torium_Sahne            Salon_3 14:00   15:00
Torium_Sahne            Salon_1 15:00   16:00
Torium_Sahne            Salon_2 16:00   17:00
Torium_Sahne            Salon_3 17:00   18:00
Torium_Sahne            Salon_3 18:00   19:00

Trump_Sahne--> 300
Trump_Sahne             Salon_1 8:00    10:00
Trump_Sahne             Salon_2 10:00   12:00
Trump_Sahne             Salon_2 13:00   15:00
Trump_Sahne             Salon_2 16:00   18:00
```



```
john_wick--> 540
john_wick                  cevahir 11:30   13:00
john_wick                  cevahir 13:15   14:45
john_wick                  cevahir 15:00   16:30
john_wick                  cevahir 16:50   18:20
john_wick                  cevahir 18:30   20:00
john_wick                  vadi    20:00   21:00
john_wick                  kanyon  21:00   22:15
john_wick                  kanyon  22:40   23:55

kotu_ruh--> 425
kotu_ruh                   cevahir 12:35   14:20
kotu_ruh                   cevahir 14:50   16:35
kotu_ruh                   cevahir 17:15   19:00
kotu_ruh                   cevahir 19:35   21:20
kotu_ruh                   cevahir 21:55   23:40

mario--> 450
mario             axis     11:00   13:00
mario             axis     13:10   15:10
mario             axis     15:20   17:20
mario             axis     17:30   19:30
mario             axis     19:40   21:40
mario             axis     21:45   23:45

uclu_puruz--> 420
uclu_puruz                 kanyon  11:15   13:00
uclu_puruz                 axis2   13:50   15:50
uclu_puruz                 axis2   16:15   18:15
uclu_puruz                 axis2   18:40   20:40
uclu_puruz                 axis2   21:05   23:05
```

5.WIS1(map<string, Salon>& sal, vector<Weighted>& result, int& t): O(M log M) explained follow.

jobs = empty vector of Weighted

// Create jobs vector

for each salon in sal:

  for each time in salon.times:

    create a new Weighted item job

    add job to jobs vector

sort jobs vector in non-decreasing order of end times using myfunction1 as the comparator

n = size of jobs

tasks = array of vectors of integers with size n, each initialized as an empty vector

maxProfit = array of integers with size n, initialized to 0

for i = 0 to n-1:

  maxProfit[i] = 0

  for j = 0 to i-1:

    update i.salon if the j. salon is non-conflicting and leading to max profit

  tasks[i].push_back(i)

```
    maxProfit[i] += jobs[i].capacito
find an index with the maximum profit
 for each i in tasks[index]:
    t = t + jobs[i].capacito
    create a new Weighted item n
    add n to result vector
// The result vector now contains the selected jobs with maximum profit.
```

WIS2 complexity is O(M log M).

Creating the jobs vector: The nested loops iterate over all the salons and their times, resulting in a complexity of O(M), where M is the total number of salon times.

Sorting the jobs vector: Sorting the vector of size M takes O(M log M) time complexity, assuming an efficient sorting algorithm like quicksort or mergesort.

Initializing tasks and maxProfit arrays: Both arrays have a size of N, where N is the number of jobs. Therefore, the initialization step takes O(N) time.

Nested loops for finding the maximum profit: The outer loop iterates N times, and the inner loop iterates up to i-1 times. On average, the inner loop runs N/2 times, resulting in an overall complexity of O(N^2).

Finding the index with the maximum profit: This loop iterates N times, resulting in a time complexity of O(N).

Constructing the result vector: The loop iterates over the tasks[index] vector, which has at most N elements. Therefore, the construction of the result vector takes O(N) time.

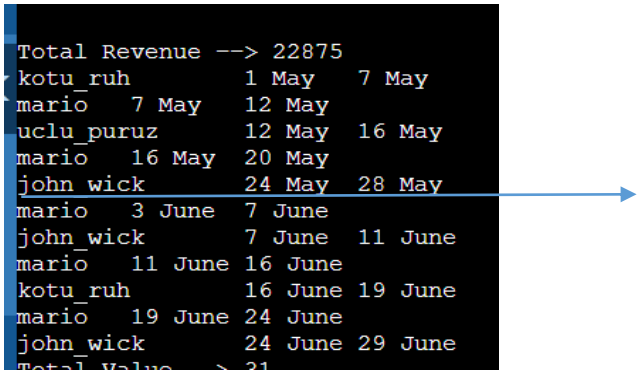
Overall, the time complexity of the code is dominated by the sorting step, resulting in a complexity of O(M log M). The additional operations have complexities of O(N^2) and O(N), but they are relatively smaller compared to the sorting step. Thus, the overall complexity can be approximated as O(M log M).

BUSRA CALISKAN 504211507 – ALGORITHM ANALYSIS-HOMEWORK 3

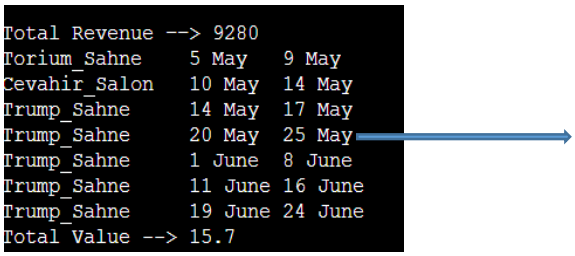
6.

//This part is also true except 1 or 2 line and missing total revenue

Missing for second case: mario May 29 June 1



Missing for first case: Cevahir_Salon May 17 May 22 and Torium_Sahne May 24 May 30 except Trump Sahne 20 May 25 May



int nonover2(int j,vector<Weighted>&b){

For i=j-1 to 0

Compares the end date of the b[i] element with the start date of the b[j] element using the dateToDays function.

If the end date of b[i] is less than or equal to the start date of b[j],

it means they are non-overlapping, and the function returns index i. If no non-overlapping element is found, the function returns -1.

}

The complexity of the nonover2 function is O(n), where n is the value of j.Since the loop iterates j-1 times in the worst case, the time complexity of the function is O(j) or O(n), where n is the value of j.

int Compute_Opt2(int j,vector<Weighted>&b){ O(n)

if(j<0)

return 0;

else

return max of b[j].capacity+Compute_Opt2 non-overlapping of j and Compute_Opt2 for j-1 O(n)

}

int Find_Solution2(int j,vector<Weighted>&b,vector<Weighted>&results){ O(n)

if (j = 0)

return 0

else if b[j].capacity+Compute_Opt2 for non-overlapping of j is bigger than Compute_Opt2 for j-1

put b[j] in result vector

return Find-Solution for non-overlapping of j-1

else Find-Solution for j-1

}

pair<int, vector<Weighted>> WIS2(map<string, Place>& sal){ O(n*m+n *logn)

jobs = empty vector of Weighted

// Create jobs vector

for each salon in sal: O(n)

for each time in salon.dates: O(m)

create a new Weighted item job

add job to jobs vector

sort jobs vector in non-decreasing order of end times using myfunction1 as the comparator O(n log n)

n = size of jobs

int n = jobs.size();

vector<int> z(n);

vector<Weighted> result

Compute_Opt2(n- 1, jobs)O(n)

Find_Solution2(i,jobs,result[i]) O(n)

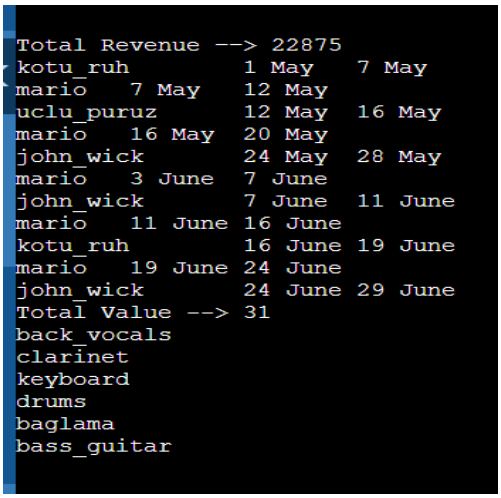
Reverse result vector O(n log n)

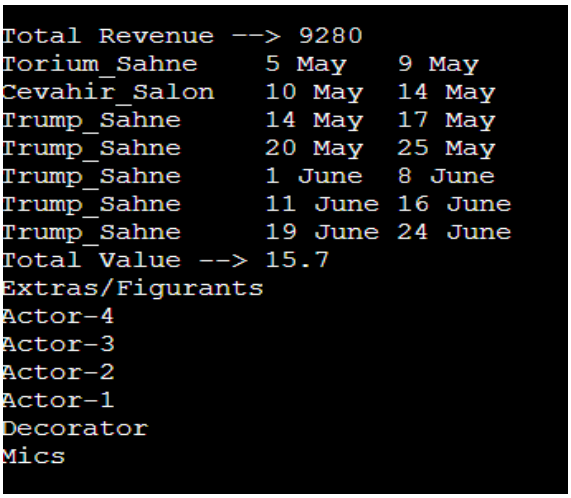
return { z[n - 1], result;

}

7.

//This part is partly true because of wrong total revenue from second part.But if you give true total revenue as named optValue,As you see follow outputs;





pair<float, vector<string>> knapsack(vector<Asset>& assets, int capacity) O(n * totalValue) explained in follow

n = length(items) // number of items

// Create a 2D vector to store the maximum values for each subproblem

dp = create 2D vector with dimensions (n + 1) x (capacity + 1) has 0.

// Initialize the first row and column with zeros

// Fill the dynamic programming table

for i = 1 to n:

for w = 1 to capacity:

// Check if the current item can be included

if price[i - 1] <= w:

// Choose the maximum value between including and excluding the current item

```
            dp[i][w] = max(values[i - 1] + dp[i - 1][w - price[i - 1]], dp[i - 1][w])
        else:
            // If the current item's weight exceeds the current capacity, exclude it
            dp[i][w] = dp[i - 1][w]

    // Trace back the selected items
    selected_items = []
    i = n
    w = capacity
    while i > 0 and w > 0:
        // If the value comes from including the current item, add it to the selected items
        if dp[i][w] != dp[i - 1][w]:
            selected_items.append(items[i - 1])
            w = w - price[i - 1]
        i = i - 1
    // Return the maximum value and the selected items
    return dp[n][capacity], selected_items
```

Let n be the number of assets in the assets vector.

Let totalValue be the total value constraint for the knapsack.

Initializing the dp table takes O(n * totalValue) time and space.

The nested loops iterating over i and j have a time complexity of O(n * totalValue).

Inside the loops, the if-else statement has constant time complexity O(1).

Constructing the selectedAssets vector takes O(n) time.

The second loop iterating over i and j in reverse has a time complexity of O(n + totalValue), as it depends on the number of selected assets and the totalValue.

The complexity of the knapsack function can be analyzed as follows: O(n * totalValue)

Creating the dp matrix: This step involves initializing a 2D vector dp of size (n+1) x (totalValue+1) and setting all elements to 0. This operation takes O(n * totalValue) time and space.

Nested loops: The function utilizes nested loops to fill in the dp matrix. The outer loop iterates from 1 to n, and the inner loop iterates from 0 to totalValue. Therefore, the time complexity of these loops is O(n * totalValue).

Updating dp values: Inside the nested loops, there is a conditional statement that checks if the price of the current asset is less than or equal to the current value j. If true, it updates the dp value based on the maximum of two options. This step has a constant time complexity.

Constructing the selected assets: After filling in the dp matrix, the function constructs the vector selectedAssets by tracing back the items that contribute to the maximum value. This step involves iterating from n to 1 and performs operations that have constant time complexity.

In summary, the overall time complexity of the knapsack function is O(n * totalValue), where n is the number of assets and totalValue is the target value. The space complexity is also O(n * totalValue) due to the dp matrix.

B-)

By discarding these constraints, the nature of the problem undergoes a significant change. The resulting schedule would consist of a combination of intervals from different places, allowing for greater flexibility in terms of where and when you can be present. However, this removal of constraints would also introduce additional complexity to the problem. The absence of these constraints makes the search space larger and finding an optimal solution more challenging.

Discarding these constraints would provide more flexibility in scheduling, enabling visits to multiple places within overlapping time intervals. However, it would also introduce greater complexity to the problem, as finding the best solution becomes more difficult due to the larger search space and increased possibilities for scheduling conflicts.

If you discard the second restriction in Case 1 of weighted interval scheduling, which states that you cannot break the determined available time intervals, it means you are allowed to split your time within a given place into multiple intervals, even if they overlap with other intervals.

By breaking the second restriction, you can have overlapping intervals for different places. For example, you can be at place A from 15:00-17:00 and then be at place B from 16:30-18:30. This allows for more flexibility in scheduling and allows for simultaneous presence in different places.

Now, if you also discard the first restriction, which requires you to be in one place for the whole day, it means you can visit multiple places throughout the day without any restrictions. You can have overlapping intervals for different places, and you are not limited to staying in one place for the entire day.

By discarding both restrictions, the resulting schedule in Case 1 would involve multiple intervals from different places, potentially with overlapping time periods. The schedule would be more flexible and may allow for better utilization of time and resources. However, it is important to

note that discarding these constraints would make the problem more complex and may require additional considerations and optimization techniques to find the best schedule.

Overall, by discarding these constraints, the resulting schedule in Case 1 would allow for more flexibility and simultaneous presence in different places, potentially optimizing the use of time and resources.