211A-COMSCI180-1 Final

MENGAN WANG

TOTAL POINTS

102 / 100

QUESTION 1

1 Problem 1 **37 / 35**

- ✓ **- 0 pts** Correct
- **- 5 pts** there is no selection of projects
- **- 20 pts** wrong algorithm
- ✓ **+ 2 pts** Name
- **- 10 pts** The algorithm may produce the non-optimal solution

I think it's more common to say that the complexity is O(mn) since m is in a parameter in the setting.

QUESTION 2

2 Problem 2 **35 / 35**

- ✓ **- 0 pts** Correct
- **- 3 pts** Complexity is incorrect.
- **- 20 pts** The algorithm does not give the schedule.
- **- 20 pts** Wrong algorithm.
- **- 15 pts** Wrong greedy strategy.
- **- 10 pts** No proof of correctness.

QUESTION 3

3 Problem 3 **30 / 30**

- ✓ **- 0 pts** Correct
- **- 5 pts** Mistakes in the algorithm.
- **- 10 pts** The algorithm does not take competition of v1 and vn into account.
- **- 15 pts** Wrong algorithm.
- **- 15 pts** Missing significant details in the algorithm.
- **- 15 pts** Algorithm is not efficient.
- **- 5 pts** Algorithm is not clear: only some descriptions.
- **- 5 pts** No explanation of correctness.
- **- 28 pts** No reasonable algorithm.
- **- 29 pts** No algorithm.

CS180 Final

Mengan Wang

1 Problem 1

1.1 Explanation/Proof + Algorithm:

Let us assume that n is an integer, greater than 0, and the number of total projects to choose from. We also assume m is an even integer, greater than 0, and is the maximum number of months that the projects length can add up to.

We are going to find the maximum profit that the company can make by calculating the profit that would be made with and without the 10 extra programmers. We will also have a method which will call the algorithm with 10 extra programmers and the algorithm without and compare the two values for which one is greater. The two algorithms work in this way:

Def: Opt(R_i,m) = optimal profit with items 1,...i with max time limit m.

Goal: Get the maximum profit for Opt(R_n,m).

Case 1: Opt(R_i) doesn't select software system R_i, meaning that Opt(R_i,w) selects the optimal profit from 1,2,...i-1 with maximum time limit m.

Case 2: Opt(R_i) selects software system R_i. Thus we need to collect profit p_i, set time limit = m - m_i, and Opt(R_i,w) selects the optimal profit from 1,2,...i-1 with the new time limit.

Bellman Equation:

$$Opt(R_{\text{i}}, w) = \begin{cases} 0 & \text{if } i = 0 \\ Opt(R_{\text{i}}, w) & \text{if m}_i > \text{m} \\ max(Opt(R_{\text{i-1}}, m), p_i + \text{Opt(R}_{\text{i-1}}\text{,m-m}_i\text{)} & \text{otherwise} \end{cases}$$

For the second method when we have 10 extra programmers, we use the same logic but at the very end subtract the cost \$10*a from the total profit which is what it cost to hire them. Also we will divide the time it takes to complete each project m_i by 2.

```
maxProfitWithoutMoreProgrammers(n, m, a, [$p_1$, $p_2$, ... $p_n$], [$m_1$, $m_2$, ... $m_n$]):
  Array M[0...n, 0...m]
  M[0,i] = 0 for each i = 0,1,...,w
  for i = 1,2,...,n
    for j = 0, ..., m
      if m < $m_i$, then M[i][j] = M[i-1][j]
      else M[i][j] = max(M[i-1][j], $p_i$+M[i-1][m-$m_i$])
      end if
    end for
  end for
  return M
```

============

```
maxProfitWithMoreProgrammers(n, m, [$p_1$, $p_2$, ... $p_n$], [$m_1$, $m_2$, ... $m_n$]):
  m[i] = (m[i]/2) for i = 0,1,...n
  Array M[0...n, 0...m]
  M[0,i] = 0 for each i = 0,1,...,m
  for i = 1,2,...,n
    for j = 0, ..., m
      if m < $m_i$, then M[i][j] = M[i-1][j]
      else M[i][j] = max(M[i-1][j], $p_i$+M[i-1][m-$m_i$])
      end if
    end for
  end for
  return M
```

```
============

chooseProjects(i, j, Subset, M):

  if i <= 0 or j <= 0

    then we terminate this function call instance

  end if

  if M[i,j] > M[i-1,j], then add project i into Subset, and recurse chooseProjects(i-1,m-$m_i$,Subset,M)

  else recurse chooseProjects(i-1,j,Subset,M)

  end if

============

getMaxProfit(n, m, [$R_1$, $R_2$, ... $R_n$], [$p_1$, $p_2$, ... $p_n$], [$m_1$, $m_2$, ... $m_n$]):

  Array profit1 = maxProfitWithoutMoreProgrammers(n,m,[$p_1$, $p_2$, ... $p_n$], [$m_1$, $m_2$, ... $m_n$])

  Array profit2 = maxProfitWithMoreProgrammers(n,m,a,[$p_1$, $p_2$,...$p_n$],[$m_1$, $m_2$,...$m_n$])-10*a

  Array Subset[0...n]

  if (profit1[n][m] > profit2[n][m])

    chooseProjects(n,m,Subset,profit1)

  else chooseProjects(n,m,Subset,profit2)

  end if

  return Subset

end getMaxProfit
```

1.2 Time Complexity:

The time complexity of our algorithm is **O(n)**.

This is because we have two methods where there is a for loop that iterates n times and the inner loop each time loops m times (which is a constant). It takes O(1) time per table entry, multiplied by n*m, which would be O(n*m). Since we have the array store values we have computed, there are no repeat computations.

Thus, we have a time complexity of O(n*m) for each method, and we multiply that by 2 because it runs 2 times giving us O(2m*n).

However, m which we are given is a constant. Thus when taken asymptotically O(2m*n) is O(n).

Our function chooseProjects also runs in O(n) time, because it's a recursive function that takes in n as a parameter, and keeps calling itself with n-1 until the value passed by n-1 is less than 0. The function runs a total of n times.

Thus, O(n) + O(n) = O(2n) = O(n).

1 Problem 1 37 / 35

✓ **- 0 pts** Correct

- 5 pts there is no selection of projects

- 20 pts wrong algorithm

✓ **+ 2 pts** Name

- 10 pts The algorithm may produce the non-optimal solution

I think it's more common to say that the complexity is O(mn) since m is in a parameter in the setting.

2 Problem 2

2.1 Algorithm:

The problem states that for each job J_i, we need a supercomputer which works on it for time r_i, a desktop for time q_i, and a specialized computer for time t_i in that specific order.

There is only 1 supercomputer, but more than n desktops and n specialized computers.

Our goal is to schedule all the jobs and minimize the time spent on all of them.

Conditions:

$n > 0$ and is an integer.

The following algorithm is creates an array S of size n, which will contain the order in which to do the jobs such that the first job is S[0], then S[1]...S[n-1].

```
============

minimizeTime(n, [$J_1$, $J_2$, ... $J_n$], [$r_1$, $r_2$, ... $r_n$], [$q_1$, $q_2$, ... $q_n$], [$t_1$, $t_2$, ... $t_n$]):

  Array S[0...n-1]

  sort Jobs in descending order by time consumed by ($r_i$ + $q_i$) so M[0] >= M[1] >= ... M[n]

  for i = 0 to n-1

    S[i] = J[i]

  end for

  return S

============
```

2.2 Proof/Explanation:

Because the supercomputer has to process all the jobs one time, it does not matter which order the jobs are processed for the supercomputer. Thus what is more important is the order the supercomputer does the jobs to minimize the time spent processing for the desktop and specialized computers. We want the jobs that take longest for the desktop and specialized computers to start working as fast as they can.

The individual time required for the desktop or specialized computer to run for the job does not matter. This is because there are more than n desktops and more than n specialized computers. There will always be a desktop or specialized computer that is open to run the job on. What matters is the overall time that (r_i + q_i) takes, so that we can prioritize running that job first on the supercomputer. Thus, we have minimized the total completion time and have an optimal solution.

2.3 Time Complexity:

Our program starts off by sorting Jobs in descending order based off of the time consumed by (r_i + q_i). We sort using a known O(nlogn) algorithm such as mergesort or quicksort (the specific kind of sort does not matter, it just has to be an efficient sort). Then, we store the result into an array with a for loop, which loops n times. Each storing is O(1), which is run n times so O(n).

When taken asymptotically, O(nlogn + n) becomes O(nlogn).

Hence, our program is **O(nlogn)**.

2 Problem 2 **35 / 35**

- ✓ **- 0 pts** Correct
- **- 3 pts** Complexity is incorrect.
- **- 20 pts** The algorithm does not give the schedule.
- **- 20 pts** Wrong algorithm.
- **- 15 pts** Wrong greedy strategy.
- **- 10 pts** No proof of correctness.

3 Problem 3

3.1 Algorithm + Proof/Explanation:

We are given a set G of n companies which can be represented as a graph. G is cyclic and there are no nodes not connected to 2 other nodes. Each node also has information about its brand value. A company is a competitor to another if it is directly connected to another in the graph. Assume all brand values are non-negative.

Our goal is to choose nodes that are not competitors (not directly connected) to each other with a maximum brand value (MPP).

Here is how we'll implement the algorithm:

Let's label the nodes in the graph starting with the root node as i = 1, and choose a direction to traverse the graph until we reach the root once again. The node we traverse to first after the root is labeled as i = 2, and so on until the nth node is i = n which comes back to root node i = 1.

Def: Opt(i) = max brand value of a perfect subset for the subproblem containing companies 1,2,...i from set G

Goal: Get maximum brand value for Opt(n) of a perfect subset of set G.

Case 1: Opt(i) selects the current node and moves to neighbor i+2, meaning that we collect the brand value of company i as V_i. The current choice is part of the optimum solution which consists of the remaining companies i+3,i+4,...n

Case 2: Opt(i) skips current node and chooses next neighbor i+1

Bellman Equation:

$$Opt(i) = \begin{cases} 0 & \text{if i = 0} \\ V_i & \text{if i = 1} \\ max(Opt(i-1), V_i + \text{Opt(i-2)}) & \text{if i > 1} \end{cases}$$

For the algorithm here is the implementation:

One thing to note is that if you select the first item, then you cannot select the last time and vice versa. Thus, we can split the problem into to subproblems. One is to find the max brand value with non-competitive brands from brands i = 1 to i = n-1, and also subproblem i = 2 to i = n.

```
findMax()
    put graph G into an array form V
    Array path1[0...n]
    Array path2[0...n]
    valueSubproblem1 = maximizeBrandValue(n, subarray of V from index 0 to n-2, path1)
    valueSubproblem2 = maximizeBrandValue(n, subarray of V from index 1 to n-1, path2)
    Array subset[0...n]
    if valueSubproblem1 > valueSubproblem2, then return findSubset(n, subset, path1)
    else return findSubset(n, subset, path2)
    endif
end findMax
============
maximizeBrandValue(n, V, dp, path):
    Array dp[0...n]
    dp[0] = 0
    dp[1] = V[1]
    for i=2...n
        dp[i] = max(dp[i-1], V[i] + dp[i-2])
        if V[i] + dp[i-2] > dp[i-1]
            path[i] = i-2 // (note that path is mutable and will be updated after exiting method)
            else path[i] = i-1
        end if
    end for
    return dp[n]
end maximizeBrandValue
```

============

```
findSubset(n, subset, path)
  int i = n
  while (i > 0)
    if path[i] == (i - 1)
      then decrement i
    else append i to subset and set i = path[i]
    end if
  end while
  return subset
end findSubset
```

3.2 Time Complexity:

Our total time complexity for this problem is O(n).

This is because in our method maximizeBrandValue, we have a for loop that loops through n times. Each loop-through we perform an O(1) operation. Since our array stores intermediate results, there are no repeated calculations. This makes the overall run time of this function to be O(n).

Our method findSubset to find the subset is also O(n). This is because we have a while loop that iterates n times.

Thus, O(n) + O(n) = O(2n) = O(n) when taken asymptotically.

3 Problem 3 **30 / 30**

- ✓ **- 0 pts** Correct
- **- 5 pts** Mistakes in the algorithm.
- **- 10 pts** The algorithm does not take competition of v1 and vn into account.
- **- 15 pts** Wrong algorithm.
- **- 15 pts** Missing significant details in the algorithm.
- **- 15 pts** Algorithm is not efficient.
- **- 5 pts** Algorithm is not clear: only some descriptions.
- **- 5 pts** No explanation of correctness.
- **- 28 pts** No reasonable algorithm.
- **- 29 pts** No algorithm.