Homework 2

ECE345 - Group 16

October 13th, 2023

Total pages: 21

Team Member	Student Number
Ardavan Alaei Fard	1007934620
Rowan Honeywell	1007972945
Isaac Muscat	1007897135

Contents

Question 1

Question 2

Question 3

(a)

(b)

(c)

(d)

Question 4

(a)

(b)

The answer to this problem will be shown through a progression of algorithms which output the number of deadlines handled by Xun given optimal choices, from the most time-complex to the least. For the initial approach to this problem, consider the pseudocode below:

HomeworkMinimax-Worst($d, i, j, D, Xsum, Worker$)

```
1  if j − i < 3D
2      if Worker = Xun
3          Xsum = Xsum + Sum(d, i, j)
4      return Xsum
5  if Worker = Xun
6      max-deadlines = −∞
7      for X = 1 to 3D
8          new-i = i + X + 1
9          new-D = max(X, D)
10         new-Xsum = Xsum + Sum(d, i, X)
11         deadlines = HomeworkMinimax-Worst(d, new-i, j, new-D, new-Xsum, Yuntao)
12         max-deadlines = max(max-deadlines, deadlines)
13     return max-deadlines
14 if Worker = Yuntao
15     min-deadlines = ∞
16     for X = 1 to 3D
17         new-i = i + X + 1
18         new-D = max(X, D)
19         deadlines = HomeworkMinimax-Worst(d, new-i, j, new-D, Xsum, Xun)
20         min-deadlines = min(min-deadlines, deadlines)
21     return min-deadlines
```

Explanation of HomeworkMinimax-Worst**:**

The pseudocode above uses the minimax principle, where one person is trying to maximize some score, and the other is trying to minimize it. In this case, the score in question is the total number of deadlines handled by Xun, where Xun is also the person who is trying to maximize the score. The base case in this recursive function is when the size of the passed range $(i - j)$ is less than or equal to $3D$. This is because both Xun and Yuntao choose optimally, so if either of them have the chance to take the remainder of the deadlines, then they will.

Initially, this function will be called as:

$$\text{HomeworkMinimax-Worst}(d, 1, N, 1, 0, Xun)$$

This passes in the full array of deadlines, sets $D = 1$, the number of Xun's deadlines ($Xsum$) to 0, and sets Xun as the person currently working. When Xun is working (lines 5 - 13), each value of X, $1 \leq X \leq 3D$ is tried by calling the function recursively, now with $X + 1$ as the new start of the deadline array, $D = \max(X, D)$, the sum of deadlines from i to X is added to $Xsum$, and finally the worker is now Yuntao. Xun's tests return the maximum number of deadlines possible for a certain starting point in the deadline list, given that Yuntao is also making optimal decisions.

When it is Yuntao's turn (lines 14 - 21), he is trying to minimize the number of deadlines handled by Xun. This part of the algorithm is fundamentally the same as Xun's, however instead of returning a recursive maximum, it returns a recursive minimum. Also, instead of updating $Xsum$, it simply is left as how it was passed. This is because Yuntao is **taking away** X deadlines from Xun, not adding them.

There are two main reasons why this algorithm is the "worst". First, each time that a sum is computed, the array of deadlines must be iterated through:

```
SUM(d, i, j)
1  sum = 0
2  for x = i to j
3      sum = sum + d[x]
4  return sum
```

Asymptotically, the SUM function is $\mathcal{O}(n)$ in time since the distance between i and j is only bounded by the size of d. This issue can be mitigated through the use of **dynamic programming**. Specifically, we can use a bottom up approach to store the sum of each possible sub-array for $\mathcal{O}(1)$ use in HOMEWORKMINIMAX.

```
GETSUBARRAYSUMS(d, N)
1  Let s by an N × N array of zeros
2  for i = 1 to N
3      s[i, i] = d[i]
4  for l = 2 to N
5      for i = 1 to N − l + 1
6          j = i + l − 1
7          m[i, j] = m[i, j − 1] + m[j, j]
8  return m
```

In the algorithm above, lines 2 and 3 first find the sums of the smallest possible sub-arrays, which are single indices. From there, the **for** loop at line 4 specifies the size of the next sub-arrays whose sums will be calculated. For the corresponding size, l, the nested **for** loop (lines 5 - 7) finds two values in s which will directly add up to the correct sum. For example, it will calculate the sum of a sub-array from indices i to j, by adding the sums of the sub-arrays from i to $j-1$ and j to j. Since the sub-arrays are calculated in increasing order of

length (line 4) there will have already been sums stored for the sub-arrays from i to $j-1$ and j to j. Since there are $\binom{N}{2}$ total sub-arrays, the space complexity of this algorithm is $\mathcal{O}(n^2)$. In terms of time, the bottom-up approach allows this algorithm to be $\mathcal{O}(n^2)$ as well, since each $m[i,j]$ takes $\mathcal{O}(1)$ to calculate.