Homework 3

ECE345 - Group 16

October 30th, 2023

Total pages: 18

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

Contents

Question 1

The following will describe the advantages and disadvantages to each of the four options of data structures that could be used for the implementation of a dictionary that allows for insert, search, delete, and rank.

Linked List: A linked list will allow the insertion of an arbitrary key limited by the number of bits used as its representation. While inserting a key will take $\mathcal{O}(1)$ time, deleting or searching for a key in a linked list takes $\mathcal{O}(n)$ time. The rank operation will iterate through the linked list keeping a count of the number of nodes with a key less than k. This will also take $\mathcal{O}(n)$ time. One advantage to the linked list is that it will only take $\mathcal{O}(n)$ memory regardless of the operations performed on it and it can grow or shrink easily. One may also consider an ordered linked list, but this causes insert to run in $\mathcal{O}(n)$ while giving an asymptotically unimportant increase in speed to the rank operation - clearly not a viable modification.

Array: For an array with size m, inserting, deleting and searching for a key $k < m$ will take $\mathcal{O}(1)$ time while requiring a fixed amount of space (or in the case of an arraylist, a continuously increasing amount of space depending on the operations performed on it). The rank operation will take $\mathcal{O}(n)$ time because although the keys are ordered in the array, an iteration must be executed to determine if spots full or not. One major limitation of the array is that the key is limited by the size of the array. Therefore, if the size of the keys are much larger than the number of keys, an unneccessary amount of space is required to maintain the dictonary - clearly not an effective solution.

Hash Map: A hash map of size m with n keys supports insertion, deletion, and searching in $\mathcal{O}(1)$ time while supporting an arbitrary key size. The major limitation of the hashmap

is the rank operation. To determine the rank of key k, the entire hasmap must be iterated over in order to find every slot with a valid key which takes $\mathcal{O}(m)$ time if the hasmap is implemented with doubled hashing, $\mathcal{O}(n)$ time with a load factor < 1, and $\mathcal{O}(m)$ time with a load factor ≥ 1.

Self-Balancing Binary Search Tree (SBBST): An SBBST makes a tradeoff for the rank operation. Each operation takes $\mathcal{O}(\log n)$ time (since this is always a balanced BST) which means the rank operation runs asymptotically faster than all the other datastructures while the other operations run more slowly than the hashmap and the array. In order for the rank operation to run in $\mathcal{O}(\log n)$ time, each entry (or node depending on how the tree is implemented) must also contain a rank field which tracks how many nodes are smaller than itself. Then, every time the insert and delete operations are called, the parent of the node whose rank changed must be updated recursively which takes $\mathcal{O}(\log n)$ time. Once rank is called, the key must be found using search which takes $\mathcal{O}(\log n)$ and then the rank of the entry is returned which takes $\mathcal{O}(1)$ time.

Conclusion: Assuming each operation is called with the same frequency, the SBBST would be the datastructure of choice because the running time of the hashmap over all operations is at best $\mathcal{O}(n)$ while the worst case running time of the SBBST is $\mathcal{O}(\log n)$.

Question 2

Given the primary and secondary hash functions, the doubled hash function will be

$$h(k,i) = (k \mod 11 + i(3k \mod 4)) \mod 11$$

The array starts as

NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL.

7 is inserted for

$$h(7,0) = 7.$$

Array:

NIL NIL NIL NIL NIL NIL NIL 7 NIL NIL NIL.

9 is inserted for

$$h(9,0) = 9.$$

Array:

NIL NIL NIL NIL NIL NIL NIL 7 NIL 9 NIL.

88 is inserted for

$$h(88,0) = 0.$$

Array:

88 NIL NIL NIL NIL NIL NIL 7 NIL 9 NIL.

11 is inserted for

$$h(11,1) = 1$$

since

$$h(11, 0) = 0$$

which is a collision.

Array:

88 11 NIL NIL NIL NIL NIL 7 NIL 9 NIL.

25 is inserted for

$$h(25, 0) = 3.$$

Array:

88 11 NIL 25 NIL NIL NIL 7 NIL 9 NIL.

23 is inserted for

$$h(23, 1) = 2$$

since

$$h(23, 0) = 1$$

which is a collision.

Array:

88 11 23 25 NIL NIL NIL 7 NIL 9 NIL.

22 is inserted for

$$h(22, 2) = 4$$

since

$$h(22, 0) = 0$$

which is a collision, and

$$h(22, 1) = 2$$

is a collision.

Array:

88 11 23 25 22 NIL NIL 7 NIL 9 NIL.

28 is inserted for

$$h(28, 0) = 6.$$

Array:

88 11 23 25 22 NIL 28 7 NIL 9 NIL.

14 is inserted for

$$h(14, 1) = 5$$

since

$$h(14, 0) = 3$$

which is a collision.

Array:

88 11 23 25 22 14 28 7 NIL 9 NIL.

21 is inserted for

$$h(21, 0) = 10.$$

Array:

88 11 23 25 22 14 28 7 NIL 9 21.

Question 3

(a)

The loss after visiting n clients would be:

$$loss = \sum_{i=1}^{n} (t_i \times \sum_{m=1}^{i} o_m)$$

The greedy algorithm to minimize the loss would be to sort the clients by their ratio of impatience to order time (i.e. $\frac{t_i}{o_i}$) from highest to lowest. In other words, we would prefer to take order from someone who is impatient AND quick in ordering. In terms of time complexity, it is required to go over the list to calculate the ratio for each client and then sort the list based on that, which leads to time complexity of $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$

(b)

The subproblem for this question is to find the client with the highest impatience-to-order time ratio within the clients left. In other words, every time that we visit a client, the subproblem is to search for the next client with the highest $\frac{t_i}{o_i}$ from the pool of clients we haven't visited yet.

(c)

We use a direct proof for proving the greedy choice property:
assume that the following sequence of the clients is the optimal solution with $\frac{t_1}{o_1}$ being the maximum (c_i represents the i_{th} client)

$$c_1 \rightarrow c_2 \rightarrow c_3 \cdots \rightarrow c_n$$

Let's compare two consecutive clients: c_m and c_{m+1}. Based on our optimal solution, if we go to c_{m+1} before c_m, our loss would be greater or equal to the minimum loss, and since the loss before m and after $m+1$ would be unchanged we can conclude that:

$$loss(c_{m+1} \to c_m) \geq loss(c_m \to c_{m+1})$$

Now let's expand the loss function for both sides:

$$LHS : t_{m+1} \times (o_1 + o_2 + \cdots + o_{m-1} + o_{m+1}) + t_m \times (o_1 + o_2 + \cdots + o_{m-1} + o_{m+1} + o_m)$$

$$RHS : t_m \times (o_1 + o_2 + \cdots + o_{m-1} + o_m) + t_{m+1} \times (o_1 + o_2 + \cdots + o_{m-1} + o_m + o_{m+1})$$

By crossing out the duplicates of both sides, we will get:

$$t_m \times o_{m+1} \geq t_{m+1} \times o_m$$

$$\frac{t_m}{o_m} \geq \frac{t_{m+1}}{o_{m+1}}$$

By this, we can conclude that the t-to-o ratio of each client has to be greater or equal to the ratio of the next client in the sequence, meaning that there exists an optimal solution that agrees with the first greedy choice of our algorithm, which is the clinet with the highest t-to-o ratio. **Q.E.D**

(d)

The proof for the greedy choice property would be valid to prove the optimal substructure as well. In the previous section, it is proved that in order to reach the minimum loss, the t-to-o ratio of each client has to be higher than the ratio of the consecutive client, meaning

that the loss can be minimized when the clients are ordered based on their t-to-o ratio, from largest to lowest. We can also prove this using contradiction:

$$c_1 \rightarrow c_2 \rightarrow c_3 \cdots \rightarrow c_n$$

We have ordered the clients such that $\frac{t_1}{o_1} \geq \frac{t_2}{o_2} \geq \cdots \geq \frac{t_n}{o_n}$. From this, we can derive that:

$$t_i \times o_j \geq t_j \times o_i \mid 1 \leq i < j \leq n \tag{1}$$

Let's assume that this is not the optimal solution. This means that there is at least one swap in a sub-sequence of clients that we need to make in order to reach the optimal solution:

$$c_k \rightarrow c_{k+1} \rightarrow c_{k+2} \cdots \rightarrow c_{k+p} \quad \rightarrow \quad c_{k+p} \rightarrow c_{k+1} \rightarrow c_{k+2} \cdots \rightarrow c_k$$

Since this swap has made our solution more optimal, it means that:

$$loss(c_{k+p} \rightarrow c_{k+1} \rightarrow c_{k+2} \cdots \rightarrow c_k) \leq loss(c_k \rightarrow c_{k+1} \rightarrow c_{k+2} \cdots \rightarrow c_{k+p})$$

If we remove the duplicates on both sides, we will be left with:

$$LHS : t_{k+p-1} \times o_{k+p} + t_{k+p-2} \times o_{k+p} + \cdots + t_k \times o_{k+p} + t_k \times o_{k+p-1} + \cdots + t_k \times o_{k+1}$$

$$RHS : t_{k+1} \times o_k + t_{k+2} \times o_k + \cdots + t_{k+p} \times o_k + t_{k+p} \times o_{k+1} + \cdots + t_{k+p} \times o_{k+p-1}$$

However, refering to Eqn. 1, LHS cannot be less than or equal to RHS as for every $t_i \times o_j | i < j$ on the left hand side, there exists the expression $t_j \times o_i$ on the right hand side which is smaller in value, making a contradiction in the assumption, proving that the substructure gives the optimal solution. **Q.E.D**

Question 4

(a)

The recursive relationship for the number of deadlines that Xun handles is defined as:

$$S(i, D, W) = \begin{cases} \sum_{k=i}^{N} d[k] & N - i \leq 3D \text{ and } W = Xun \\ \min_{1 \leq X \leq 3D} \{S(i + X + 1, \max(X, D), Xun)\} & W = Yuntao \\ \max_{1 \leq X \leq 3D} \{S(i + X + 1, N, \max(X, D), Yuntao) + \sum_{k=i}^{i+X} d[k]\} & W = Xun \end{cases}$$

The left-hand side shows a call S being made for some starting day i, D, and worker W (Xun or Yuntao), where S represents the number of deadlines that Xun will handle from day i to the end. Firstly is the case when the number of days that remain is less than or equal to $3D$. Since both Xun and Yuntao choose optimally, they will always take the full remainder if possible. Therefore, in this case, Xun will simply handle all of the remaining deadlines.

Next is the case when it is Yuntao's turn. Since he is looking to minimize the number of deadlines that Xun handles, he will iterate through all possible values of X (from 1 to $3D$) and select the one which yields the smallest value. To produce a value, a recursive call is made where the current day, i, has been updated to $i + X + 1$, which would be the starting day for Xun's next turn.

Finally, we can examine the case of Xun's turn. This case is basically the converse of the one previously described. Instead of trying to find the minimum value, Xun is trying to find which choice of X (from 1 to $3D$) will maximize the number of deadlines that she handles. We must also make sure that, for Xun's choice of X, the corresponding sum of deadlines from i to $i + X$ will be added to her total.

(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:

Q4B-WORST($d, i, N, D, Xsum$, *Worker*)

```
1  if N − i < 3D
2      if Worker = Xun
3          Xsum = Xsum + SUM(d, i, N)
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 = Q4B-WORST(d, new-i, N, 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 = Q4B-WORST(d, new-i, N, new-D, Xsum, Xun)
20         min-deadlines = min(min-deadlines, deadlines)
21     return min-deadlines
```

Explanation of Q4B-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 remainder of days in the deadline array d 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:

$$\text{Q4B-WORST}(d, 1, N, 1, 0, Xun)$$

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 $i + X + 1$ as the new current day, D set to $\max(X, D)$, the sum of deadlines from i (current day) 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.

Analysis of Q4B-WORST**:**

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.

The second reason why Q4B-WORST is poor is because of the number of recursive calls that it makes. There is nothing stopping the algorithm from calculating a certain configuration of the worker, D, and current day that it already has in the past. This is, of course, a classic example of where dynamic programming can help optimize an algorithm.

The time complexity of this algorithm can be shown through the expression:

$$T(n) = \sum_{X=1}^{n-1}(T(n-X)+X)$$

Each value of X that can be chosen is iterated through (with an upper bound of n being placed on X), and a recursive call is made on what remains in the deadline array. The value of X is also added for each call, since this is the amount of time that the SUM function will take. We can rearrange the expression to achieve:

$$T(n) = \sum_{X=1}^{n-1} T(n-X) + \sum_{X=1}^{n-1} X = \sum_{X=0}^{n-1} T(X) + \mathcal{O}(n^2)$$

The following upper bound can be placed on this expression (CLRS, 3rd Edition, pg. 364):

$$T(n) = \mathcal{O}(n^2 2^n)$$

In terms of space, at any given point the number of recursive calls that will be stored will

be bounded by the depth of the recursive tree. The maximum depth of the recursive tree occurs when each call removes a minimum amount of the deadline array. Since the minimum amount of the array that can be removed is 1, then the maximum depth of this tree is bounded by $\mathcal{O}(n)$. Therefore, since each call uses a constant amount of memory, this algorithm is $\mathcal{O}(n)$ in space.

Introducing dynamic programming:

The aforementioned issues can be mitigated through the use of dynamic programming. To begin, we can use a bottom-up approach to store the sum of each possible sub-array for $\mathcal{O}(1)$ use in the Q4B algorithm.

```
BottomUpSums(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              s[i, j] = s[i, j − 1] + s[j, j]
8  return s
```

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 $s[i, j]$ takes $\mathcal{O}(1)$ to calculate.

Next, we can address the issue of how many recursive calls are made. The dynamic programming approach that will be used is top-down, or "memoization". Since Xun and Yuntao are choosing optimally, we can store the outcome (number of deadlines handled by Xun) of each sub-array that traverses from some day i to the end individually. Therefore, a structure that we can use to store such data is a hash table, where the key is a tuple of three values ($Worker$, D, i), and the value is the outcome given the respective configuration. This tuple contains i, the starting point of the remainder of the array after some set of deadlines are chosen, the corresponding D value, and the worker (Xun or Yuntao) whose turn is next.

This addition greatly reduces the number of recursive calls that are made. In short, there is one sub-array that spans some index i to the end of the full array, and a total of N possible values of i. Letting k represent the number of possible values D may take, the total number of recursive calls that needs to be made is $2\mathcal{O}(kn)$ (the coefficient represents two players). However, since the value of D is only bounded by n, we can express the new total number of recursive calls as $2\mathcal{O}(n^2) = \mathcal{O}(n^2)$.

Consider the pseudocode below:

```
InitMemo(N)
1  Let m be a hash table that takes keys of three-integer tuples
2  for D = 1 to N
3      for i = 1 to N
4          m[(Xun, D, i)] = −1
5          m[(Yuntao, D, i)] = −1
6  return m
```

The above pseudocode shows how the hash table used to memoize the tuples would be initialized, with each unseen configuration set to -1.

We must also consider the space requirements of this incorporation. Since we are storing a single integer for $2n^2$ possible pairs, the space complexity of this structure is $\mathcal{O}(n^2)$.

Creating an optimized algorithm:

Finally, let us imagine some algorithm Q4B-Better which uses the same recursive principles described in Q4B-Worst, but incorporates the bottom-up and top-down optimizations previously described. To reiterate, this algorithm will obtain sums of sub-arrays from a pre-calculated table in $\mathcal{O}(1)$ time and memoize sub-arrays based on the starting day, value of D, and starting worker (Xun or Yuntao). This second optimization will limit the number of recursive calls made to just $\mathcal{O}(n^2)$. To show this, we can examine each dimension of the memoization keys. The first is the starting day of the sub-array; since there are only N possible starting days, this dimension has a maximum size of N. Next is the value of D. Since D is proportional to whatever value of X is chosen, and the maximum possible value of X is N, D is also limited by N, and therefore its dimension also has a maximum size of N. Finally the last dimension is just the player, therefore it has a maximum size of 2. Therefore, the maximum number of recursive calls that can be made is bounded by $2 \times N \times N = \mathcal{O}(n^2)$.

Analysis of the optimized algorithm:

To analyze the full optimized approach to this problem, we must analyze the union of BottomUpSums, InitMemo, and Q4B-Better. As stated previously, the tabulation of the sums of all sub-arrays within the array of deadlines is calculated by the BottomUpSums algorithm in both $\mathcal{O}(n^2)$ time and space. Next, the initialization of the memoization table

is facilitated by the InitMemo function. As also stated previously, this function stores an integer value for each combination of indices of the deadline array (bounded by N) and possible values of D (also bounded by N). Therefore, this function is also bounded by $\mathcal{O}(n^2)$ in time and space. Finally, we come to our Q4b-Better algorithm. We now must analyze the time complexity of an individual recursive call. Each recursive call can find the sum of a given subarray in $\mathcal{O}(1)$ time, with this action being performed once per iteration through the possible choices of X. The number of choices of X is bounded by N, the size of the full deadline array, so each recursive call has an $\mathcal{O}(n)$ upper bound. Also, because the amount of data stored with each recursive call is independent of input size, each recursive call is individually $\mathcal{O}(1)$ in space. Therefore, since the use of memoization limits the number of recursive calls made to $\mathcal{O}(n^2)$, Q4b-Better is $\mathcal{O}(n^3)$ in time and $\mathcal{O}(n)$ in space for the same reason as described with the un-optimzed algorithm.

Finally, the total time complexity of the dynamic approach can be expressed as:

$$\mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n^3) = 2\mathcal{O}(n^2) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$$

And the space complexity is shown to be:

$$\mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n) = 2\mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$$

Getting a final answer:

Finally, since the Q4b-Better algorithm only gives us the number of deadlines that Xun handles, we can find who handles the most deadlines by the same principle used in the last sentence of the previous section. Therefore, consider the pseudocode below:

```
Q4B-FinalAnswer(d, N)
1  s = BottomUpSums(d, N)
2  m = InitMemo(N)
3  deadlines-Xun = Q4B-Better(d, i, N, D, 0, Xun, s, m)
4  if deadlines-Xun > (1/2)s[1, N]
5       return Xun
6  elseif deadlines-Xun < (1/2)s[1, N]
7       return Yuntao
8  else
9       return Tie
```

If the number of deadlines that Xun handles is greater than half of the total deadlines, then she handles the most. Otherwise if it is less, then Yuntao does. Finally, if the number of deadlines handled by each worker is equal, then it is a tie.

Summary:

In the solution to this question, we first examined the baseline minimax implementation of the Xun and Yuntao's strategies, Q4B-Worst. Then, we introduced dynamic programming to greatly decrease the original algorithm's time complexity. This was done in two steps:

- A bottom-up approach for summing sub-arrays
- A top-down (memoization) approach for storing the output of deadlines completed by Xun within a certain sub-array

The first optimization in Q4B-Better allowed for an $\mathcal{O}(1)$ retrieval of sums instead of $\mathcal{O}(n)$, and the second optimization limited the number of recursive calls that needed to be made to $\mathcal{O}(n^2)$. Finally, through an analysis of the full process, it was found that the time complexity was reduced from $\mathcal{O}(n^2 2^n)$ to $\mathcal{O}(n^3)$, with the space complexity increasing from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$.