COSC 320: Assignment 4

This assignment is due **Friday, March 14 at 7 PM**. Late submissions will not be accepted. All the submission and formatting rules for Assignment 1 apply to this assignment as well.

1 List of names of group members (as listed on Canvas)

Provide the list here. This is worth 1 mark. Include student numbers as a secondary failsafe if you wish.

1. Rin Meng, 51940633
2. Mika Panagsagan 29679552
3. Kevin Zhang 10811057

2 Statement on collaboration and use of resources

To develop good practices in doing homeworks, citing resources and acknowledging input from others, please complete the following. This question is worth 2 marks.

1. All group members have read and followed the guidelines for groupwork on assignments given in the Syllabus).

   ● Yes ○ No

2. We used the following resources (list books, online sources, etc. that you consulted):

   (a) DeepSeek R1, ChatGPT to generate ideas, explain concepts, and check if we are going in the right direction.

   (b) Google Search, Google's AI Overview

   (c) COSC 320 Lecture Slides

3. One or more of us consulted with course staff during office hours.

   ○ Yes ● No

4. One or more of us collaborated with other COSC 320 students; none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   ○ Yes ● No

   If yes, please list their name(s) here:

5. One or more of us collaborated with or consulted others outside of COSC 320; none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   ○ Yes ● No

   If yes, please list their name(s) here:

3 Mystery QuickSelect

The following variant of the QuickSelect algorithm carefully chooses a pivot, so as to guarantee that the sizes of Lesser and Greater are at most $7n/10$, and may equal $7n/10$ in the worst case. It does this by choosing $\lceil n/5 \rceil$ elements in $\Theta(n)$ time (line 7 below), and setting the pivot to be the median of these elements via a recursive call (line 8). Exactly how the $\lceil n/5 \rceil$ elements are chosen is not important. The rest of the algorithm is identical to QuickSelect.

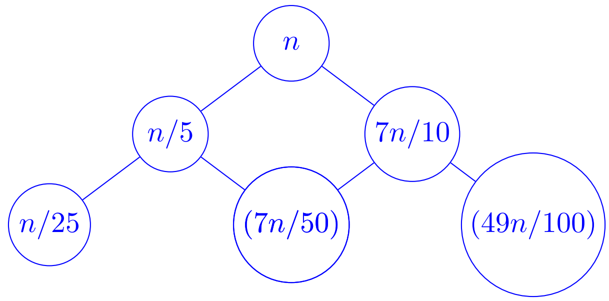
```
1:  function MysteryQuickSelect(A[1..n], k)
2:      ▷ return the kth smallest element of A, where 1 ≤ k ≤ n and all elements of A are distinct
3:      if n == 1 then
4:          return A[1]
5:      else
6:          ▷ the next two lines select the pivot element p
7:          create array A′ of ⌈n/5⌉ "carefully choosen" elements of A          ▷ this takes Θ(n) time
8:          p ← MysteryQuickSelect(A′, ⌈|A′|/2⌉)                                ▷ A′ has ⌈n/5⌉ elements
9:          Lesser ← all elements from A less than p           ▷ Lesser has 7n/10 elements in the worst case
10:         Greater ← all elements from A greater than p      ▷ Greater has 7n/10 elements in the worst case
11:         if |Lessor| ≥ k then
12:             return MysteryQuickSelect(Lesser, k)
13:         else if |Lesser| = k − 1 then
14:             return p
15:         else    ▷ |Lesser| < k − 1
16:             return MysteryQuickSelect(Greater, k − |Lesser| − 1)
17:         end if
18:     end if
19: end function
```

1. [3 points] Let $T'(n)$ be the *worst-case* run time of MysteryQuickSelect. Complete the following recurrence for $T'(n)$. by replacing the parts with ???s. You can ignore floors and ceilings. No justification needed.

$$T'(n) = \begin{cases} c, & \text{if ???} \\ ??? + cn, & \text{if } n > 1 \end{cases}$$

$$T'(n) = \begin{cases} c, & \text{if } n = 1 \\ T'(\frac{n}{5}) + T'(\frac{7n}{10}) + cn, & \text{if } n > 1 \end{cases}$$

2. [3 points] Draw the first three levels (0,1, and 2) of the recursion tree for your recurrence. Label each node with the size of the subproblem it represents, as well as the work done at that node not counting recursive calls. Finally, put the total work per level in a column on the right hand side of your tree. You can provide a hand-drawn figure as long as it is clear and legible.



Level	Total Work Per Level
0	$O(n)$
1	$O(n/5) + O(7n/10) = O(n)$
2	$O(n/25) + O(7n/50) + O(7n/50) + O(49n/100) = O(n)$

3. [2 points] What is the worst-case runtime of MysteryQuickSelect? Provide a short justification of your answer.

   The worst-case runtime of MysteryQuickSelect is $O(n)$. This is because the recurrence $T'(n) = T'(n/5) + T'(7n/10) + cn$ gets solved to $O(n)$, as shown in the previous question.

4. [2 points] For comparison, what is the worst case runtime of the *original* QuickSelect algorithm from the worksheet, in which lines 7 and 8 of the pseudocode above are replaced by setting the pivot p to $A[1]$? Provide a short justification of your answer.

   Line 8 of the pseudocode above is replaced by setting the pivot p to $A[1]$, which results in the worst-case runtime of $O(n^2)$.

   - In the worst case, every recursive call only removes one element from the list, so the recurrence is $T(n) = T(n-1) + cn$.
   - This results to a linear depth recursion tree, and the total work done is $O(n^2)$.
   - This is similar to the worst case of QuickSort.

4 Subway Sub-array

[Thanks to Peter Gu for contributing this problem.]

Peter has just purchased a sandwich from the Life Building Subway. The sandwich can be represented as an array A of n real-valued quantities. Peter is peculiar, he enjoys partitioning his sandwich into contiguous sub-arrays such that every element from the original array belongs to a sub-array. He then scores each sub-array $A[l, r], 1 \leq l \leq r \leq n$ with the following formula:

$$\mathrm{SS}(l, r) = ax^2 + bx + c,$$

where $a, b, c \in \mathbb{Z}$ and x is the sum of all elements in the sub-array $A[l..r]$. He would like you to develop an algorithm to find the score of a partition that maximizes the sum of the scores of its subarrays.

More formally, an instance of the problem consists of the array $A[1..n]$ plus the quantities a, b, and c, and the goal is to determine $\mathrm{Sub}(n)$, defined as follows when $n \geq 1$:

$$\mathrm{Sub}(n) = \max_{1 \leq k \leq n} \max_{0=p_0<p_1<p_2<\ldots<p_k=n} \mathrm{SS}(p_0+1, p_1) + \mathrm{SS}(p_1+1, p_2) + \mathrm{SS}(p_2+1, p_3) + \ldots \mathrm{SS}(p_{k-1}+1, p_k).$$

(Here, k is the number of subarrays in the partition, and for $1 \leq i \leq k$, the ith subarray ranges from $p_{i-1}+1$ to p_i.) We'll also define $\mathrm{Sub}(0)$ to be 0.

Example: Suppose that the array is $[1, 2, -3, 2, 1]$ and $a = 1, b = 1, c = 5$. One possible partition into subarrays is $[1, 2], [-3], [2, 1]$. The respective scores for this partition are $[3^2+3+5], [3^2-3+5], [3^2+3+5]$ and the total sum is 45.

Note: No justification is needed for the first five parts of this problem.

1. [1 point] For the example array above, give an optimal partition and write down its value $\mathrm{Sub}(5)$.

   One possible optimal partition is $[1, 2], [-3], [2, 1]$, with a total score of 45 which was given in the example.

2. [3 points] Provide pseudocode to calculate all of the quantities $\mathrm{SS}(l, r)$, for $1 \leq l \leq r \leq n$. Your pseudocode should run in $O(n^2)$ time.

```
function CalculateAllQuanitites(A[1..n], a, b, c)
    n ← length(A)
    Q[1..n][1..n] ← 0
    for l = 1 to n do
        for r = l to n do
            x ← 0
            for i = l to r do
                x ← x + A[i]
            end for
            Q[l][r] ← a · x^2 + b · x + c
        end for
    end for
    return Q
end function
```

3. [3 points] Let $n \geq 1$. Suppose that in an optimal partition, the rightmost subarray is $A[i+1..n]$, where $0 \leq i \leq n-1$. Write down an expression for the value of $\mathrm{Sub}(n)$ in terms of the quantity $\mathrm{SS}(i+1, n)$ and the function $\mathrm{Sub}()$ applied to a smaller problem instance.

   For $n \geq 1$, $A[i+1..n]$, $0 \leq i \leq n-1$, $SS(i+1, n)$

$$\mathrm{Sub}(n) = \max_{0 \leq i \leq n-1} (\mathrm{SS}(i+1, n) + \mathrm{Sub}(i))$$

4. [2 points] Provide a recurrence for Sub(n). Your answer to the previous part should be useful.

   Base case $n = 0$: $\mathrm{Sub}(0) = 0$. For $n \geq 1$, $\mathrm{Sub}(n) = \max_{0 \leq i \leq n-1} (\mathrm{SS}(i+1, n) + \mathrm{Sub}(i))$. Therefore the recurrence is:

$$\mathrm{Sub}(n) = \begin{cases} 0, & \text{if } n = 0 \\ \max_{0 \leq i \leq n-1} (\mathrm{SS}(i+1, n) + \mathrm{Sub}(i)), & \text{if } n > 0 \end{cases}$$

5. [5 points] Using the recurrence, develop a memoized algorithm to compute Sub(n). Remember that a memoized algorithm is always recursive. Your algorithm can use the quantities SS(l, r) (since these values can be pre-computed by your algorithm from part 1 above).

```
function MemoizedSub(A[1..n], a, b, c)
    n ← length(A)
    Q ← CalculateAllQuantities(A, a, b, c)
    M[0..n] ← None
    return MemoizedSubHelper(A, a, b, c, Q, n, M)
end function

function MemoizedSubHelper(A[1..n], a, b, c, Q, n, M)
    if n == 0 then
        return 0
    end if
    if M[n] ≠ None then
        return M[n]
    end if
    M[n] ← −∞                                        ▷ Set to smallest possible value
    for i = 0 to n − 1 do
        M[n] ← max(M[n], Q[i + 1][n] + MemoizedSubHelper(A, a, b, c, Q, i, M))
    end for
    return M[n]
end function
```

6. [2 points] What is the runtime of your memoized algorithm? Provide a short justification.

   The runtime of the memoized algorithm is $O(n^2)$.

   - The algorithm computes all the quantities SS(l, r) in $O(n^2)$ time.
   - The memoized algorithm computes Sub(n) using the recurrence in $O(n)$ time.
   - The total runtime is $O(n^2)$.

7. [3 points] Bonus: Suppose that instead of using the SS() function to score a subarray, now use a linear variant:

$$\mathrm{SS}'(l, r) = bx + c,$$

   where $b, c \in \mathbb{Z}$ and x is the sum of all the elements in the sub-array $A[l..r]$. For this variant, we want to find the score of a partition that maximizes the sum of all subarrays. Explain how we can achieve this with an algorithm whose runtime is faster than that for the original problem.

   The absence of the x^2 term in the scoring function $\mathrm{SS}'(l, r)$ means that the scoring function is linear in the sum of the elements in the subarray. This means that the score of a partition is maximized by maximizing the sum of the elements in the subarrays. Therefore, the optimal partition is to have all the elements in one subarray. This can be done in $O(n)$ time by computing the sum of all the elements in the array.

5 Legend of Zelda

[Thanks to Denis Lalaj for contributing this problem.]

An instance of the problem is $G[1..m][1..n]$—a 2D array of size $m \times n$ where each entry $G[i, j]$ is an integer (either positive or negative), representing the points that Link can accumulate in the room at coordinates (i, j).

Link starts his quest at room $(1, 1)$ and needs to get to the (m, n) corner, via a path where each move is either to the right or downwards. Link starts with some initial number of points, and upon entering each room (including the first), Link's points are adjusted up or down by adding the points in that room. Link must ensure that he always has at least one point.

For $1 \le i \le m$ and $1 \le j \le m$, let $HP[i, j]$ denote the minimum number of points that Link needs to have, when starting by entering the room with coordinates (i, j), in order to reach (m, n) while always having at least one point. The problem is to compute $HP[1, 1]$—the minimum number of points needed initially in the full game.

We have the following recurrence for $HP[i, j]$:

$$HP[i,j] = \begin{cases} \max(1, 1 - G[m, n]), & \text{if } i = m \text{ and } j = n, \\ \max(1, \min(HP[i+1, j], HP[i, j+1]) - G[i, j]), & \text{if } 1 \le i \le m-1 \text{ or } 1 \le j \le n-1, \\ \infty, & \text{if } i = m+1 \text{ or } j = n+1. \end{cases}$$

1. [5 points] Using this recurrence, develop a dynamic programming algorithm to compute $HP(1, 1)$. Remember that a dynamic programming algorithm is iterative (involving loops), not recursive.

```
function HP(G[1..m][1..n])
    m ← nums rows of G
    n ← nums columns of G
    HP[m+1][n+1] ← ∞
    HP[m][n] ← max(1, 1 − G[m, n])
    for i = m down to 1 do
        for j = n down to 1 do
            if i == m and j == n then
                min HP ← min(HP[i+1, j], HP[i, j+1])
                HP[i, j] ← max(HP[i+1, j], HP[i, j+1]) − G[i, j]
            end if
        end for
    end for
    return HP[1, 1]
end function
```

2. [2 points] What is the runtime of each of your algorithm? Provide a short justification.

   The runtime of the dynamic programming algorithm is $O(mn)$.

   - It's a double loop that iterates over all the rooms in the grid, so the runtime is $O(mn)$.
   - It is a 2D array of size $m \times n$.

3. [Optional: 0 points] Use the recurrence above to write a memoized algorithm to compute $HP(1, 1)$.