CPSC 320 2024W1: Assignment 4

This assignment is due **Friday, November 15 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.

- Hansel Poe - 82673492

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):
3. One or more of us consulted with course staff during office hours.
   ○ Yes ● No
4. One or more of us collaborated with other CPSC 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 CPSC 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

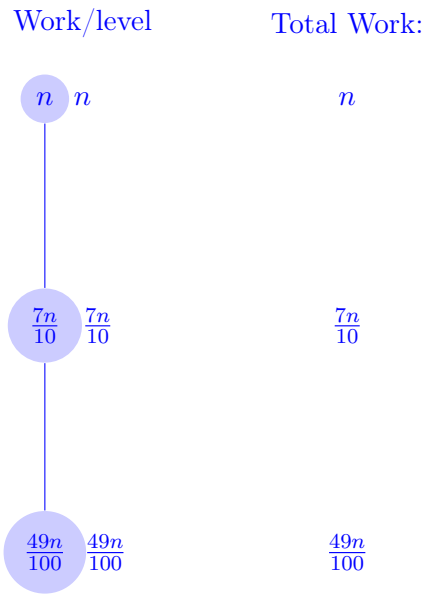
The following variant of the QuickSelect algorithm carefully chooses a pivot, so as to guarantee that the sizes of Lesser and Greater are at most $7n/10$, and may equal $7n/10$ in the worst case. It does this by choosing $\lceil n/5 \rceil$ elements in $\Theta(n)$ time (line 7 below), and setting the 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.

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

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.



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

   For a level i, the work done is $(\frac{7}{10})^i n$. Summing up the work gives us $\sum_{i=0}^{l}(\frac{7}{10})^i n$, where l is the number of levels. Note that if we let $l \to \infty$, $(\frac{7}{10})^i$ converges to $\frac{10}{3}$, and our summation would tend to $(\frac{10}{3})n$. This gives us an upper bound of $O(n)$. Because the first level performs n work outside of recursive calls, we must have the lower bound $\Omega(n)$ for our algorithm. Our algorithm is thus $\Theta(n)$.

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.

   The worst case for the original QuickSelect is when we always pick either the largest or smallest element in each recursive call, giving us the recurrence:

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

   Which is $\Theta(n^2)$. This is worse than our MysteryQuickSelect.

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 $\mathrm{Sub}(5)$.

   $[1], [2], [-3], [2].[1]$

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.

   procedure $\mathrm{SS}(A[1 \ldots n], l, r, a, b, c)$
   $sum \leftarrow 0$;
   for i from l to r **do**
   $sum \leftarrow sum + A[i]$
   end for
   return $a(sum)^2 + b(sum) + c$
   end procedure

   procedure GENERATEALLSS$(A[1 \ldots n], l, r, a, b, c)$
   create the array $SS[1 \ldots n][1 \ldots n]$ to store the results
   for k from 1 to n **do**
   for l from 1 to $n - k + 1$ **do**
   $\mathrm{SS}[l][l + k - 1] \leftarrow \mathrm{SS}(A[1 \ldots n], l, l + k - 1, a, b, c)$
   end for
   end for
   end procedure

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 Sub() applied to a smaller problem instance.

   $\mathrm{Sub}(n) = \mathrm{SS}(i + 1, n) + \mathrm{Sub}(i)$

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

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

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).

```
procedure SUB(n)
    create Soln[1...n]
    for i from 1 to n do
        Soln[i] ← −1
    end for
     return SubHelper(n)
end procedure

procedure SUBHELPER(n)
    if n == 1 then return SS(1, 1)
    else if n == 0 then return 0;
    else
        if Soln[n] == −1 then
            Soln[n] ← minimum of SubHelper(i) + SS(i + 1, n) for all 0 ≤ i ≤ n − 1
        end if
    end if
     return Soln[n]
end procedure
```

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

   In the recurrence tree there are $n+1$ Internal nodes (from 0 to n),we can assume that when computing minimum of SubHelper(i) + SS($i+1, n$), we do the recursive calls in increasing order of i, this will ensure that in subsequent calls, the value for SubHelper(i) will have been computed. Each node i will then have the children from 0 to $i-1$, giving us the total nodes to be $\sum_{i=0}^{n} i$, which 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:

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

   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.

5 Legend of Zelda

[Thanks to Denis Lalaj for contributing this problem.]

See the dynamic programming tutorial for motivation for 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 $1 \leq i \leq m$ and $1 \leq j \leq m$, 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.

In the tutorial, you studied the following recurrence for $HP[i, j]$:

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

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.

```
procedure FINDSTARTINGHP(G[1...m][1...n], i, j)
    return findHP(G[1...m][1...n], 1, 1);
end procedure

procedure FINDHP(G[1...m][1...n], i, j)
    create HP[1...m][1...n]
    HP[m][n] ← max(1, 1 − G[m][n])
    for i from m to 1 do
        for j from n to 1 do
            if i == m and j == n then
                continue
            elseHP[i, j] = max(1, min(HP(i + 1, j), HP(i, j + 1)) − G[i][j])
            end if
        end for
    end for
     return HP[i][j]
end procedure
```

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

   The algorithm traverses the 2D array HP from bottom to top, right to left, so that with each i recursive call, we will have already computed (HP$(i + 1, j)$ and HP$(i, j + 1)$. Because there are $m \times n$ entries, the algorithm is then $\Theta(nm)$.

6 Subset Sums mod m

In this example of dynamic programming, the recurrence describes a *set* of values, rather than a single value as in earlier examples. An instance I of the problem is a set $\{x_1, x_2, \ldots, x_n\}$ of nonnegative integers and a positive integer m. We say that a value v, with $0 \leq v \leq m-1$, is *feasible* with respect to instance I if for some non-empty subset R of $\{x_1, x_2, \ldots, x_n\}$,

$$\sum_{x \in R} x \equiv v \pmod{m}.$$

Also, for each $i, 0 \leq i \leq n$, let $V(i)$ be the set of feasible values with respect to instance $(\{x_1, x_2, \ldots, x_i\}, m)$.

1. [3 points] Explain why the following recurrence holds for $V(i)$.

$$V(i) = \begin{cases} \emptyset, & \text{if } i = 0 \\ V(i-1) \quad \cup \quad \cup_{v \in V(i-1)}\{(v + x_i) \pmod{m}\} \quad \cup \quad \{x_i \pmod{m}\}, & \text{if } 1 \leq i \leq n. \end{cases}$$

   The term $V(i-1)$ gives us the set of feasible values for the instance $(\{x_1, x_2, \ldots, x_{i-1}\}, m)$.After adding x_i into our set,the next two terms gives us any new feasible values.

   The final term corresponds to feasible value for the subset $\{x_i\}$ (because $(x_i mod m) \equiv x_i (mod m)$).

   Meanwhile, for the second term we use the fact that if $v \in V(i-1)$ then there exists subset R such that $\sum_{x \in R} x \equiv v (mod m)$, Now we know that $x_i \equiv x_i mod m$, using the properties of congruence modulo, we have $\sum_{x \in R} x + x_i \equiv v + x_i mod m$. And so this gives us new feasible values.

2. [5 points] Design a dynamic programming algorithm that, given an instance $I = (\{x_1, x_2, \ldots, x_n\}, m)$ of Subset Sums mod m, determines whether 0 is feasible with respect to I. The algorithm outputs "Yes" or "No". Your algorithm should run in $O(nm)$ time. Your pseudocode should create a solution array, Soln[0..n], and should store the set of values $V(i)$ in Soln[i]. The pseudocode can use set operations such as $\cup$ as in the recurrence above.

```
procedure IS0FEASIBLE({x_1, ..., x_n})
    return isFeasible({x_1, ..., x_n}, 0)
end procedure

procedure ISFEASIBLE({x_1, ..., x_n},u)
    create Soln[0...n]
    Soln[0] ← ∅
    for i from 1 to n do
        r ← x_i mod m
        for v in Soln[i − 1] do
            a ← (v + x_i) mod m
        end for
        Soln[i] = Soln[i − 1] ∪ a ∪ r
    end for
    return u ∈ Soln(n)
end procedure
```

3. [2 points] Explain why your algorithm of part 2 runs in time $O(nm)$. The outer for loop iterates for all values of n, and the inner for loop iterates for all previous feasible values which is at most m (because $0 \leq v \leq m$). So in total, our algorithm is $O(nm)$

4. [3 points] Suppose that indeed there is a subset R of S such that $\sum_{x \in R} x \equiv 0 \pmod{m}$. Fill in the missing parts (indicated by ????) of the following algorithm, that finds such a subset R by making a

call to FIND-$R(n, 0)$. You can assume that the array Soln$[0..n]$ has already been pre-computed, where Soln$[i]$ stores the set $V(i)$; the algorithm uses the array Soln. Your algorithm should run in time $O(n)$.

```
procedure FIND-R(i, v)
    ▷ return non-empty R ⊆ {x_1, ..., x_i} for which Σ_{x∈R} x ≡ v (mod m), given that R exists
    ▷ the algorithm has access to the array Soln[0..n]
    if x_i (mod m) = v then
        return {x_i}
    else if v ∈ Soln[i − 1] then
        return Find-R(i − 1, v)
    else
        v' ← v − x_i (mod m)
        return {x_i} ∪ Find-R(i − 1, v')
    end if
end procedure
```

5. [2 points] Explain why your algorithm of part 4 runs in time $O(n)$.

   The work done outside of recursive calls is in constant time (because we already compute Soln$[0 \ldots n]$). Now each recursive call decrements i by one and so there can be at most n recursive calls, giving us $O(n)$.