CPSC 320 2024W1: Assignment 4 Solutions

3 Mystery QuickSelect

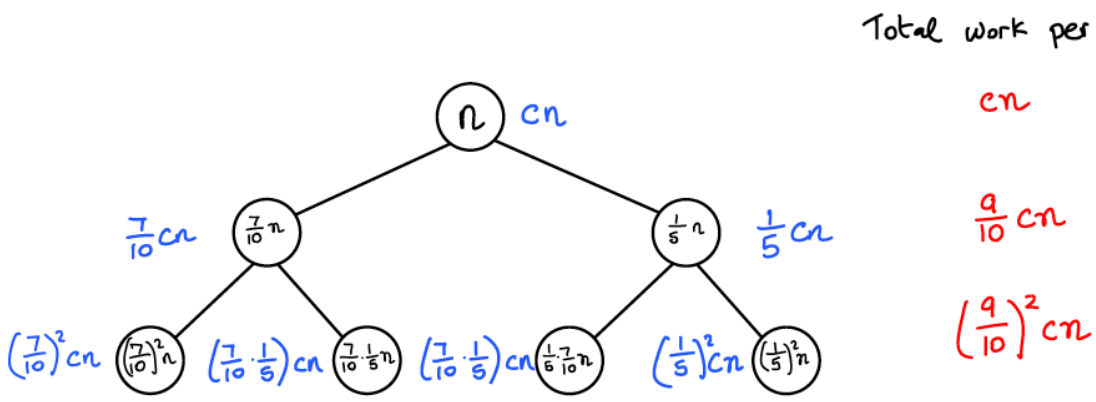
The following variant of the QuickSelect algorithm carefully chooses a pivot, so as to guarantee that the sizes of Lesser and Greater are at most $7n/10$, and may equal $7n/10$ in the worst case. It does this by choosing $\lceil n/5 \rceil$ elements in $\Theta(n)$ time (line 7 below), and setting the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

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

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.



Work per node is shown in blue, and total work per level is shown in red.

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

   The worst-case runtime is $\Theta(n)$. We can get an upper bound by summing the work per level:

   $$cn(1 + 9/10 + (9/10)^2 + \ldots) = cn/(1 - 9/10) = 10cn = (n),$$

   where we use the formula for geometric sum, with $x = 9/10 < 1$ to get the closed form expression for the sum. We get a lower bound of $\Omega(n)$ simply by accounting for the work at the root of the recursion tree. The matching upper and lower bounds give a $\Theta(n)$ worst-case runtime.

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 runtime is $\Theta(n^2)$. This happens, for example, when the pivot is always the largest element in the array. Then the recurrence when $n > 1$ is

   $$T(n) = T(n - 1) + cn.$$

   A recurrence tree for this has depth n, with the work at level i being $n - i$. Summing up over all the levels gives the result.

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 $\mathrm{Sub}(5)$.

   The partition $[1], [2], [-3], [2], [1]$ into singleton sets is optimal, and has value 47.

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 SS(A[1..n])
       for l from 1 to n do
           Sum[l, l] ← A[l]
           SS[l, l] ← a(Sum[l, l])^2 + b(Sum[l, l]) + c
           for r from l + 1 to n do
               Sum[l, r] ← Sum[l, r − 1] + A[r]
               SS[l, r] ← a(Sum[l, r])^2 + b(Sum[l, r]) + 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.

   In this case,

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

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

$$\text{Sub}(n) = \begin{cases} 0, & \text{if } n = 0, \\ \max_{0 \le i \le n-1} \text{Sub}(i) + \text{SS}(i+1, n)\}, & \text{if } n \ge 1. \end{cases}$$

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).

Algorithm 1 Memoized approach for computing Sub(n)

```
procedure MEMO-SUBWAY(A[1...n], a, b, c)        ▷ n ≥ 1
    ▷ initialization of a solution array goes here
    create an array SUB([0..n])
    initialize all of SUB to −∞
    SUB[0] ← 0
    ▷ a call to the recursive helper function goes here
    return MEMO-SUBWAY-HELPER (A[1...n], n)
end procedure

procedure MEMO-SUBWAY-HELPER(A[1...n])        ▷ j ≥ 1
    ▷ put pseudocode for the helper function here
    if SUB[j] is −∞ then
        for i from j − 1 to 0 do
            SUB[j] ← max{SUB[j], MEMO-SUBWAY-HELPER(A[1...n], i) + SS(i + 1, j)}
        end for
    end if
    return SUB[j]
end procedure
```

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

   The runtime is $\Theta(n^2)$. There is $O(n)$ initial work in the main procedure, before the helper function is called. When the helper function is called with parameter j, where also SUB[j] is ∞, the work done not counting recursive calls costs $\Theta(j)$, because of the **for** loop. For each j in the range from 1 to n, there is one call to the helper function with parameter j, for which SUB[j] is $infty$. Summing up over these calls, the total work is $\sum_{j=1}^{n} \Theta(j) = \Theta(n^2)$.

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.

   The key observation is that every partition will get the same score when summing up the quantities bx. For example, suppose the partition of the array results in the sums $x_1, x_2, ..x_k$. The overall sum would be $bx_1 + bx_2..bx_k$, which is equal to $b(x_1 + x_2..x_k)$ which is equivalent to $b \cdot \sum_{i=1}^{n} A[i]$. Thus every partition will result with the same overall sum of the components bx, and this sum can be computed in

$O(n)$ time by summing up all elements in the array. Lastly we consider the impact of c. If c is positive, then we would like to include c as many times as possible, and so the partition of the array into n sub-arrays, each of size 1, is optimal. Otherwise, we only want c to be included once by choosing the partition to have just one sub-array, which equals the full array $A[1..n]$. This decision can be made in $O(1)$ time. So the total time to compute the optimal score is $O(n)$.

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.

```
function DPSaveZelda(G[1...m][1...n])
    HP ← Create array of size m × n
    Initialize all entries to −∞
    HP[m, n] ← max(1, 1 − G[m, n])          ▷ Fill in the base case, at least 1 HP to survive the last cell
    ▷ Tabulate last row and column - only one direction here
    for i from m − 1 downto 1 do
        HP[i, n] ← max(1, HP[i + 1, n] − G[i, n])
    end for
    for j from n − 1 downto 1 do
        HP[m, j] ← max(1, HP[m, j + 1] − G[m, j])
    end for
    ▷ Fill in entire DP array
    for i from m − 1 downto 1 do
        for j from n − 1 downto 1 do
            HP[i, j] ← max(1, min(HP[i + 1, j], HP[G, i, j + 1]) − G[i, j])
        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.

   $O(mn)$ because we have to fill all entries on the table of size $m \times n$ but work per cell is constant time.

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 $V(i)$.

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

   Each of the three sets on the right hand side, namely $V(i-1)$, $\cup_{v \in V(i-1)}\{(v + x_i) \pmod{m}\}$, and $\{x_i \pmod{m}\}$, are clearly subsets of $V(i)$, since each contains values that are sums of elements from the set $\{x_1, \ldots, x_i\} \pmod{m}$. So, the right hand side is a subset of the left hand side. To see that the left hand side, namely $V(i)$, is a subset of the right hand side, note that the first set on the right hand side, $V(i-1)$, accounts for all feasible values that can be obtained from any non-empty subset of $\{x_1, \ldots, x_{i-1}\}$, the second set accounts for values that can be obtained from a set containing x_i *plus* some other non-empty subset of $\{x_1, \ldots, x_{i-1}\}$ (whose values sum to $v \pmod{m}$), and the last set accounts for the subset of $V(i)$ that contains just x_i. So, all subsets R of $\{x_1, \ldots, x_{i-1}\}$ are accounted for on the right hand side, and we can conclude that it contains $V(i)$.

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 SUBSET-SUM-MOD-m({x_1, x_2, ..., x_n}, m)
    ▷ returns "Yes" if for some non-empty subset R of {x_1, x_2, ..., x_n}, Σ_{x∈R} x ≡ 0 (mod m)
    ▷ and returns "No" otherwise

    create an array Soln[0..n]                          ▷ entries of this array will be sets
    Soln[0] ← ∅

    for i from 1 to n do
        Soln[i] ← Soln[i − 1] + ∪_{v∈Soln[i−1]}{v + x_i (mod m)} ∪ {x_i (mod m)}
    end for
    if 0 is in the set Soln[n] then
        return "Yes"
    else
        return "No"
    end if
end procedure
```

3. [2 points] Explain why your algorithm of part 2 runs in time $O(nm)$.

   For each $i, 1 \leq i \leq n$, the size of the set $V(i)$, and therefore the set Soln[i], is at most m, since it is a subset of $\{0, 1, \ldots, m-1\}$. Therefore, the time to compute Soln[i] at the ith iteration of the **for** loop

is $O(m)$, since it takes the union of at most $m+1$ sets, one of which (namely Soln$[i-1]$) has size at most m and the remaining of which have size 1. Summing up the time for the **for** loop over all i gives a total runtime of $O(nm)$.

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 time for a call to FIND-$R(i, v)$, ignoring recursive calls, is $O(1)$. To obtain this time, we could represent each set Soln$[i]$ as an array with m entries, so that Soln$[i][v]$ is if v is in the set Soln$[i]$ and is 0 otherwise. There is one recursive call to FIND-$R(i-1, v')$. So the run time can be described by the recurrence

$$T(i) = \begin{cases} c, & \text{if } i = 0 \\ T(i-1) + c, & \text{if } i > 0 \end{cases}$$

The recursion tree for this recurrence has at most n levels, with one node per level (since there is just one recursive call), and the work per level is c. Summing up over all the levels, the total work is $O(n)$.