Assignment #1

Due: May 29, 2021, 11:59 pm (EDT)

SIWEI TANG(tangsiw3)
Yuewei Wang (wangyuew)

Q1:

a)
```
1:procedure LocationSet(X,T)
2:      sort X and T in increasing order
3:      # $x_1$, $x_2$, $x_3$, $x_4$...$x_k$
4:      # $t_1$, $t_2$, $t_3$, $t_4$...$t_j$
5:      L := {}
6:      for k = 0:K do
7:              if  L == {} or$x_k$ is not in range of any cell tower of L:
8:                      find the maximum j such that$t_j$ can cover $x_k$ ($|t_j$ -$x_k| \le$ R)
9:                      if $t_j$is not None:
10:                             L = L + {$t_j$}
11:     return  L
```

b) We will prove that part a) greedy algorithm would choose the minimum number of cell phone towers needed and the location set L would be the same as other greedy algorithms.
Define P(k): k is in range [0, K], the location set $L_k$would be the best solution set that minimizes the set size and covers all the customers as other optimal algorithms like G.

Base Case: k = 0; No customers, so set L would be empty. P(0) holds
Inductive Step: Assume P(k) holds i.e. $L_k$and G_k is the same.
want to prove P(k+1) holds i.e. $L_{k+1}$and G_k is the same.
Case 1: For the customer location X_{k+1}, it is already covered by L_k and G_k, so $L_{k+1} = L_k = G_k$, cover all the customer locations, so G_{k+1} should be the same as L_{k+1}

Case 2: For the customer location X_{k+1}that is not covered by L_k and G_k.
By algorithm, choose cell tower t_{new} that covers X_{k+1} and t_{new} is the largest possible value. L_{k+1} = L_k+ {t_{new}}, if G_{k+1} = G_k+ {t_{new}} = L_k+ {t_{new}}=L_{k+1} , then part a) greedy algorithm works the same as optimal algorithm G.

If G_{k+1}chooses a different cell tower different from t_{new}i.e. g_{new} that covers X_{k+1}but on the left side of t_{new} since t_{new} is the largest possible value. For all the customers covered by g_{new}, if the customers location smaller than X_{k+1}, cell towers in $L_k(G_k)$ would cover them, otherwise t_{new} would cover all the customers that larger than X_{k+1}.

$G_{k+1} = G_k + \{g_{new}\}$ is the optimal solution since G is an optimal algorithm.

so $G_{k+1} - \{g_{new}\} + \{t_{new}\} = G_k + \{t_{new}\} = L_k + \{t_{new}\} = L_{k+1}$ would cover all the customers that G_{k+1} covers and they have the same size so L_{k+1} is optimal.

c) For line 2, sort X and T in increasing order, since X is size k and T is size j, so it takes $O(Klg(K)) + O(Jlg(J))$, the for loop starts from line 6 to line 10 takes $O(JK)$, so in total $O(Klg(K)) + O(Jlg(J)) + O(JK)$.

Q2:

(a)

The solution takes all n requests into a set (R) as input in the algorithm. Sort all requests from R by finish time in non-decreasing order first. Schedule the first request into A1, then check each request along with the order. Compare the start time of the request with the latest finish time in two sets (F1 and F2) to check whether it can be scheduled in either one set. If not, skip that request. For n-1 rounds of checking, return the set pair (A1, A2). The pseudocode will be:

```
1: **procedure** INTERVALSCHEDULE2 ($R, n$):
2:     sort all requests in $R$ by the finish time
       # i.e. $R = [r_1, r_2, \dots, r_n]$ $where\ f_1, f_2, f_3, f_4 \dots f_n$ are in non-decreasing order
3:     $F1 := f_1$
4:     $F2 := -\infty$
5:     $A1 := \{1\}$     # the initial request ($r_1$) must be finished earliest
6:     $A2 := \{\}$
7:     **for** $i = 2:n$ **do**
8:         **if** $s_i >= F1$ **then**
9:             $A1 = A1 \cup \{i\}$
10:            $F1 = f_i$
11:        **else if** $s_i >= F2$ **then**
12:            $A2 = A2 \cup \{i\}$
13:            $F2 = f_i$
14:    **return** $(A1, A2)$
```

In INTERVALSCHEDULE2, sorting in line 2 takes $\Theta(nlg(n))$. Line 3 to 7 takes constant steps to assign all variables. The loop in line 8 takes $n - 1$ iterations, each iteration takes constant times in line 9 to 14, which is a total $\Theta(n)$. Therefore, the whole complexity of the code is $\Theta(n\ lg(n))$.

We will show the complexity is $o(n^2)$by using limits:

$$\lim_{n \to \infty} \frac{n\ lg(n)}{n^2} = \lim_{n \to \infty} \frac{lg(n)}{n} = \lim_{n \to \infty} \frac{\frac{1}{n\ ln(2)}}{1} = \lim_{n \to \infty} \frac{1}{n\ ln(2)} = \frac{1}{ln(2)} \times \lim_{n \to \infty} \frac{1}{n} = \frac{1}{ln(2)} \times 0 = 0$$

Therefore, the complexity of the code is $o(n^2)$.

(b)

Let k be the number of requests that have been checked for scheduling in the algorithm.
WTS: The algorithm provides an optimal schedules for A1 and A2 for $1 \leq k \leq n$

B.C. $k = 1$, since $n \geq 1$, at least 1 request is present in input. In line 5 we schedule the earliest request without entering the loop, A1 has been scheduled with the earliest finished request (i.e. $i = 1$). The schedule (A1, A2) is optimal.

I.H. Assume for all $1 \leq k < n$, $(A1_k, A2_k)$is the optimal schedule for k requests of checking.

I.S. WTS: $(A1_{k+1}, A2_{k+1})$is the optimal schedule for $k + 1$ requests of checking.

For the $(k + 1)^{th}$request, there are three cases:

<u>case 1</u>: the $(k + 1)^{th}$request can be scheduled into A1

Then $s_{k+1} \leq F1$must be true due to no overlapping, line 10 to 12 execute, $A1_{k+1} = A1_k \cup \{k + 1\}andA2_{k+1} = A2_k$.

Since $A1_k$and $A2_k$is the optimal solution for k requests of checking (I.H.), $A1_{k+1}$ includes the $(k + 1)^{th}$request, then $(A1_{k+1}, A2_{k+1})$ is the optimal solution for the $(k + 1)$ requests of checking.

<u>case 2</u>: the $(k + 1)^{th}$request can be scheduled into A2

Then $s_{k+1} \leq F2$ must be true, line 13 to 15 execute, $A2_{k+1} = A2_k \cup \{k + 1\}$ and$A1_{k+1} = A1_k$. Note that this case occurs when the $(k + 1)^{th}$request overlaps with the existed schedule in $A1_k$, thus $s_{k+1} > F1$is true as well.

Similar as case 1(I.H.), $(A1_{k+1}, A2_{k+1})$is the optimal solution for $(k + 1)$ requests of checking.

<u>case 3</u>: the $(k + 1)^{th}$request cannot be scheduled

Then$F2 < s_{k+1}$and $F1 < s_{k+1}$must be true, this request is not added into any sets.$A2_{k+1}$and$A1_{k+1}$will remains as $A1_k$and $A2_k$. The schedule is still optimal.

Q3:

(a)

A greedy algorithm could be: Sort all the coins by values in non-increasing order. Find the first coin that is smaller than or equal to$A$. Update the value difference in values (denoted asA'), continuously find the wanted value forA'as before, until sorted all the coins.

This solution fails since picking the best choice at each time cannot guarantee to find the solution. A counter example: Let coins set$= \{2, 2, 6, 7, 10\}$and $A = 8$. After sorting, the coins set =$\{10, 7, 6, 2, 2\}$. The algorithm will pick 7 initially and continuously search for 1. However, the solution should be $\{6, 2\}$.

(b)

Step 1. Optimal substructure / recursive structure:

Let O be an optimal solution for A such that $\{i_1, i_2, ...\ i_k\}$by value non-decreasing order,$\{i_1, i_2, ...\ i_k\} \subseteq \{1,\ 2,\ ...\ , m\}$. For the largest value coin(i_k), there are two cases:

1).$i_k < m$: then c_m is not included in$O$, and$\{i_1, i_2, ...\ i_k\} \subseteq \{1,\ 2,\ ...\ , m - 1\}$.

2).$i_k = m$: then c_m is in$O$. $A - c_m = A - c_{i_k}$, then$O' = \{i_1,\ i_2,\ ...\ ,\ i_{k-1}\}$is optimal solution for $(A - c_m)$and$\{i_1, i_2, ...\ i_{k-1}\} \subseteq \{1,\ 2,\ ...\ , m - 1\}$.

Step 2. Define memoization:

Let M be the 2-D memoization array where M[$k_{A'}$] is the minimum number of coins can be found from $\{c_1,\ c_2,\ ...\ ,\ c_k\}$and the sum is A', where $A' \in N,\ 0 \leq A' \leq A$.

Step 3. Recursive relation in terms of the array:

Combined step1, there are two cases for the recursive relation:

1). $c_k > A'$: then$c_k$cannot be the solution forA'. The range of input coins for A'became to$\{c_1,\ c_2,\ ...\ ,\ c_{k-1}\}$,which is M[$k_{A'}$] = M[$(k - 1)_{A'}$].

2). $c_k \leq A'$: then $c_k$is potential in the optimal solution for A'. The minimum number of coins have two possibles:

a). if $c_k$is in the optimal solution, then minimum number of coins for $A'$is M[m [$(k - 1)_{A'-c_k}$]+1

b). if $c_k$is not in the optimal solution, then similar to case 1.

The minimum number of coins is the smaller value between M[$(k - 1)_{A'-c_k}$]+1and M[$(k - 1)_{A'}$], which is M[$k_{A'}$] = min (M[$(k - 1)_{A'-c_k}$]+1, M[$(k - 1)_{A'}$]) .

Note that $0 < k \leq m$ and $0 < A' \leq A$ for the above two cases. If we reach to the base cases, such as:

1).$A' = 0$: which is M[k_0] , then M[k_0] = 0 since no coin can make value be 0.

2). $k = 0$: it is equivalent as M[$0_{A'}$] , then M[$0_{A'}$] = -1 since the coin range is 0. Thus, there is no solution for A' and marks as -1.

Step 4. Bottom-up iterative algorithm:

```
1: procedure M-BOTTOM-UP-COMPUTE ($c_1, c_2, \dots, c_m, A$):
2:     Let M[$i_j$] where $0 \le i \le m,\ 0 \le j \le A$ be initialized as $-\infty$:
3:     M[$0_0$] = 0
4:     for $j = 1{:}A$ do
5:         M[$0_j$] = -1
6:     for $i = 1{:}m$ do
7:         M[$i_0$] = 0
8:         for $j = 1{:}A$ do
9:             if $c_i > j$ then
10:                M[$i_j$] = M[$(i-1)_j$]
11:            else
12:                M[$i_j$] = min (M[$(i-1)_{j-c_i}$]+1, M[$(i-1)_j$])
13:    return M[$m_A$]
```

Complexity: $\Theta(mA)$(construct the 2-D array takes mA iterations.)

Step 5. Optimal solution using the array values:

Finally, use M to compute the optimal solution as:

```
1: procedure FIND-SOLUTION-WRAPPER ($c_1, c_2, \dots, c_m, A$, M):
2:     $S := []$
3:     return FIND-SOLUTION($c_1, c_2, \dots, c_m, A$, M, S)
```

```
1: procedure FIND-SOLUTION ($c_1, c_2, \dots, c_m, A$, M, S):
2:     if M[$m_A$] == -1 then
3:         return {}
4:     else
5:         $A' := A$
6:         for $i = m{:}1$ do
7:             if M[$i_{A'}$] != M[$(i-1)_{A'}$] then
8:                 S = S ∪ {$i$}
9:                 $A' = A' - c_i$
10:        return S
```

Complexity: $\Theta(m)$(the loop in FIND-SOLUTION line 6 takes m iterations, and FIND-SOLUTION-WRAPPER takes constant time.)

(c)

As step 4 in (b), M-BOTTOM-UP-COMPUTE construct array, the loop go iteratively in $m \times A$times, which takes $\Theta(mA)$. As step 5 in (b), FIND-SOLUTION go iteratively search the solution, the worst-case could occur when go over all $m$times, which takes $\Theta(m)$.

Therefore, the whole algorithm has the worst-case running time is $\Theta(mA)$.

Q4:

Step 1. Optimal substructure / recursive structure:

By the second requirement of output, the path goes by depth of layer, which increases 1 each depth. Let $j_1, j_2, \ldots, j_L$ be the optimal path. In such a case, $(1, j_1)$ is the starting block for our path and the hardness is d - H[$1, j_1$]. Each drill would go to next level (one just beneath) directly or diagonally, which means (2, j_2) is the next block where $j_2 \in \{j_1 - 1, j_1, j_1 + 1\}$. Then, starting from (2, j_2), the path $j_3, j_4, \ldots, j_L$ must be optimal.

Step 2. Define memoization:

Let M be the 3-D memoization array where M[H', (i, j)] represents the maximum amount of gold can be found from $(i, j)$and H is the initial hardness of the drill, where $1 \le i \le m$, $1 \le j \le n$, $0 \le H' \le d$.

Step 3. Recursive relation in terms of the array:

For each block, there are two cases for the recursive relation:

1). $H' < H[i, j]$: which means the current drill hardness is not enough for (i, j). Then, we mark M[H', (i, j)] = 0

2). $H' \ge H[i, j]$: which means the current drill hardness is enough for (i, j). Then, we include the gold content$G[i, j]$ into M[H', (i, j)]. Also, the maximum value among the beneath blocks is also included, which means M[H', (i, j)] = max{M[$H' - H[i, j]$, $(i + 1, j - 1)$], M[$H' - H[i, j]$, $(i + 1, j)$] , M[$H' - H[i, j]$, $(i + 1, j + 1)$] } + $G[i, j]$.

Note for $(i + 1)$, $(j - 1)$ *and* $(j + 1)$, out of the boundary could occur. Then we remove the option of that block for the max() evaluation, which requires a statement to check.

Step 4. Bottom-up iterative algorithm:

```
1: procedure M-BOTTOM-UP-ITERATIVE ($d, H, G$):
2:     for $H'$ = 1: $d$ do
3:         for $i = m$: 1 do
4:             for $j = 1: n$ do
5:                 if $H' < H[i, j]$ then            #drill is not enough
6:                     M[$H'$, $(i, j)$]: = 0
7:                 else
                       #construct a set $S$contains three possibles of next block:
8:                     $S$: ={M[$H' - H[i, j]$, $(i + 1, j - 1)$],
                              M[$H' - H[i, j]$, $(i + 1, j)$] ,
                              M[$H' - H[i, j]$, $(i + 1, j + 1)$]}
9:                     if $i + 1 > m$ then        # bottom layer, no next block
```

```
10:                 $S$={}
11:             else if $j + 1 > n$ then       # no right underneath block
12:                 $S = S$\ M[$H' - H[i,j]$, $(i + 1, j + 1)$]
13:             else if $j - 1 \leq 0$ then     # no left underneath block
14:                 $S = S$\ M[$H' - H[i,j]$, $(i + 1, j - 1)$]
15:             M[$H'$, $(i, j)$] := max ($S$)+ $G[i, j]$
```

Complexity: $\Theta(dmn)$because of the triple-nested for loop.

Step 5. Optimal solution using the array values:

Finally, use M to compute the optimal solution as:

```
1: procedure FIND-PATH-WRAPPER ($d$, $H$, $G$, M):
2:     $start := 1$
3:     $curr :=$M[$d$, $(1, start)$]
4:     for $j = 2 : n$ do         #find the maximum value as starting block
5:         if $curr >$M[$H'$, $(1, j)$] do
6:             $start = j$
7:             $curr =$M[$H'$, $(1, start)$]
8:     return FIND-PATH ($d$, $start$)
```

```
1: procedure FIND-PATH ($d$, $start$):
2:     $P := []$
3:     $H' := d$
4:     $l := 1$
5:     while  M[$H'$, $(l, start)$]> 0
6:         $P = P \cup \{start\}$
7:         $l = l + 1$
8:         $H' = H' - H[l, start]$
9:         $start =$ max{M[$H'$, $(l, start - 1)$], M[$H'$, $(l, start)$], M[
$H'$, $(l, start + 1)$]}
10:    return $P$
```

Complexity: $\Theta(n + m)$(the loop in line 4 of FIND-PATH-WRAPPER takes $\Theta(n)$. The while loop in FIND-PATH takes $\Theta(m)$since the number of layers will not be more than m. Thus, the whole complexity is $\Theta(n + m)$)