Sum Yi Li
(UID: 505146702)

Discussion 1A
10:00am – 11:50am
Goldstein Or Paz

input array

① - First, we have an array P that have n points in it.

- We want to find the smallest distance between two points in a given array.
- We sorted the input array according to the x coordinates
- Then we find the middle point of the sorted array, then we can take the middle point as $P[n/2]$
- We divide the input array P into halves. The first subarray will contain the points from $P[0]$ to $P[n/2]$.
  The second subarray will contain the points from $P[n/2+1]$ to $P[n-1]$
- After that, we will recursively find the smallest distance in both the first and second subarray.
  Let $d\ell$ be one of the distance from the left subarray
  Let dr be one of the distance from the right subarray
- The minimum distance between left & right subarray is minimum distance $d = \min(d\ell, dr)$
- d will become our upper bound of the minimum distance. Then, we need to consider the pair such that one of the point in the pair is from the left half and the other point in the pair is from the right half.

- Let's consider there is vertical line going through the the middle point of the array which is $P[n/2]$
- We need to find all the points whose x coordinate is closer than the minimum distance d to the middle vertical line.
- We will store all of those points from above in another array called temp[ ]
- We then assume to presort all points from temp[ ] according to y coordinates.
- Let assume the sorted array based from y coordinate to be P_y[ ].
- When we make recursive calls of finding the minimum distance, we need to divide points of P_y[ ] according to the vertical line. It is done by doing the follows.
- We can process every point & compare its own x coordinate with the middle line's x coordinate.
- Therefore, for every point in the array temp[ ], we need to check at most 7 pair of points to find the smallest distance in temp[ ]
- Finally, we need to return the minimum distance d & the distance that is calculated from above.

① cont.

Prove of Correctness

- Let assume the set of points are denoted by $P = \{P_1, P_2, \dots, P_n\}$ where P_i has coordinates (x_i, y_i)
- For 2 points $P_i, P_j \in P$, we will use $d(P_i, P_j)$ to denote the distance between them
- Let L denote the vertical line separate into Q & R side.
- If there is exists $q \in Q$ and $r \in R$ such that $d(q, r) < \delta$, then each of q & r lies within a distance δ of L.
- Let S be the set that contains points that are within δ of L.
- We will show the correctness by induction in size of P
- When $|P| \leq 3$, it is clear to find out the closest pair
- For given P, the closest pair is computed correctly by induction.
- By the following 2 characteristics, the algorithm determine whether any pair of points in S is at distance $< \delta$, if yes it will return that closest pair.
    1) If we have s, s' in S & $d(s, s') < \delta$, then s & s' are within 15 positions from each other in the sorted list S_y based from the y coordinates.
    2) There is a $q \in Q$ & $r \in R$ such that $d(q, r) < \delta$ if there exsists s, s' in S such that $d(s, s') < \delta$
- This means that the closest pair of point in P is either has both the points entirely in Q or entirely in R, or one element from Q & one element from R

- In the first case, the closest pair is discovered by the recursive call.
- In the second case, the closest pair is at distance $< \delta$, it is also correctly discovered by the remaining algorithm.

① cont.

Time Analysis

- Let $T(n)$ be the time complexity of the algorithm.
- Assume we use a $O(n\log n)$ sorting algorithm to sort the array.
- The algorithm divide all the points in two sets & recursively calls the 2 sets.
- After dividing, we find the strip in $O(n)$ time.
- It takes $O(n)$ to divide the $P_y[\,]$ array (the array sorted according to the y coordinates) around the middle vertical line.
- At last, we find the closest point in temp[] array in $O(n)$ time.
- So, $T(n)$ can be expressed as follows:

$$T(n) = 2T(n/2) + O(n) + O(n) + O(n)$$

$$T(n) = 2T(n/2) + O(n)$$

$$\boxed{T(n) = T(n\log n)}$$

② Exercise 3, page 314

a) - Let assume we have a graph with 5 nodes which are n_1, n_2, n_3, n_4, n_5

- We will have the following edges as listed:

(n_1, n_2)

(n_1, n_3)

(n_2, n_5)

(n_3, n_4)

(n_4, n_5)

- The algorithm will try to return the longest path is only 2 which is $(n_1, n_2) \longrightarrow (n_2, n_5)$

- But, the real answer of the longest path is 3 which is $(n_1, n_3) \rightarrow (n_3, n_4) \longrightarrow (n_4, n_5)$

(2)

b) - We will apply the idea of dynamic programming algorithm
- We will apply the optimal subproblems opt[i] in the case of finding the length of the longest path from node V_1 to node V_n.
- We need to be aware that among all the nodes V_i, there is no guarantee that there is a path from node V_1 to node V_i.
- We will let opt[i] $= -\infty$
- At the beginning, we will use opt(1) $= 0$ to indicate the longest path from node V_1 to itself since the node doesn't have any edges.
- The algorithm is follows:
- Let's create an array M that has index 1 to n which is M[1....n]
- Then we set M[1] $= 0$ which means the longest path from V_1 to V_1 is 0 edges.
- For $i = 2, \ldots, n$. Then, we are looping from node V_2 to node V_n
- We set M $= -\infty$
    - For all the edges indicated by (j, i)
        - if M[j] is not equal to $-\infty$
            - if M is less than M[j] + 1
                - update M to be M[j] + 1

- end of inner if else statement
- end of outer if else statement
- end of inner for loop
- We then set M[i] to be M
- end of outer for loop.
- At last, we return M[n] as our longest path length.

Time Analysis

- Assume that all the edges that go into a node i could be listed in $O(n)$.
- As we loop from node V_2 to node V_n, it takes $O(n)$
- For each node, we go through all the edges (j, i), it takes $O(n)$
- Therefore, it takes $\boxed{O(n^2)}$ in total as running time

② Cont.

Prove of correctness

- For each vertex v, we will define the $d[v]$ to be the longest path from vertex s to vertex v.
- For each incoming edge u, v for a given vertex v, we will treat it as the last edge in the longest path from s to v.
- If we assume the edge u, v is the last edge, the longest path has weight $d[u] + w(u,v)$.
- The recursion become $d[v] = \max_{(u,v) \in \text{edge}} (d[u] + w(u,v))$
- We consider each vertex iteratively in a sorted order
- For each step, we are looking for any possibilities of adding one more edge to the path
- Therefore, the algorithm will output the longest path that begins at V_1 & ends at V_n.

③ Exercise 5, page 316

- We will apply the concept of dynamic programming concept.
- Let assume we have $y_1, y_2, \ldots, y_n$ is the optimal segmentation for string y. Therefore, $y_1, y_2, \ldots, y_{n-1}$ is also an optimal segmentation that exclude y_n in terms of the prefix of y.
- This means we can get a better solution by substituting the optimal segmentation for the prefix in the original problem.
- Let $opt(a)$ be the ratings in terms of score points of the best segmentation for the prefix which contain the first a characters of y.
- Then, we can say that $opt(a) = \min_{b \leq a} \{opt(b-1) + score(b \ldots n)$ will output the best optimal segmentation.
- We want to compute $opt(n)$
- $Score(x \ldots y)$ is referring to the score of the word which is formed by letters from index x to index y

---

Prove of correctness by induction

- The base case is a word with only one letter in it
- We have define a function to find the optimal solution for index less than a
- For the inductive steps, we want to show that $opt(a)$ will output the optimal cost of any segmentation of prefix y & up to a-th position of letter

- We will look at the last word of the optimal segmentation of the prefix & assume the word starts at index b which is less than or equal to the index a.
- Based from the previous assumption, there is an optimal solution when the prefix is formed by the first b-1 letters.
- However, from the inductive steps, opt(b) will output the previous optimal segmentation solution.
- As a result, opt(a) = opt(b) + last word segmentation cost.
- The inductive step indeed calculate the optimal cost for every possible choice of last word. Therefore, the inductive step would output the optimal segmentation cost.

Running Time Analysis :

- We have n as the number of letters for the input word.
- If we start looping from index 1 to n for this

equation : $opt(a) = \min_{b \leq a} \{opt(b-1) + score(b \dots n)\}$

- The algorithm is quadratic algorithm.

(4) Exercise 10, page 321

a) Let's consider the following 2 examples.

First example

Machine	Minute 1	Minute 2
A	4	20
B	2	40

The optimal solution will choose machine B for both minute 1 & 2. If we compared with a greedy solution which will end up choosing machine A for both minute 1 & 2.

Second example

Machine	Minute 1	Minute 2	Minute 3	Minute 4
A	4	2	2	400
B	2	2	40	200

The optimal solution will choose machine A & B for minute 1, 2, 3, 4. The correct solution should be choosing machine A at minute 1, then move at minute 2, then choose machine B at minute 3 & 4.

b) - Let's define the maximum value of an optimal plan that stay on machine A from minute 1 to minute X to be opt (x, A)

- Then define the maximum value of an optimal plan that stay on machine B from minute 1 to minute X to be opt (x, B)

- Let assume that we are currently on machine A when we are at minute X, we need to find out which machine we have chosen at minute X − 1

- There are two possible scenario. First, we have already stayed at machine A at the first place. Second, we are moving from machine B to machine A.

- For the first scenario, we have the following expression $opt(x, A) = opt(x-1, A) + a_x$, where a_x refers to we have picked machiine A at minute X

- For the second scenario, we have the following expression $opt(x, A) = opt(x-2, B) + a_x$ since we are at machine B at minute X − 2 & in the process of moving to machine A at minute X − 1

- By combining both equation, the optimal plan equation on machine A :

$$opt(x, A) = \max\{opt(x-1, A), opt(x-2, B)\} + a_x$$

- We will have the same situation hold for machine B
- For the first scenario, we will have the following expression.

  $opt(x, B) = opt(x-1, B) + b_x$, where b_x refers to we have picked machine B at minute x
- For the second scenario, we have the following expression

  $opt(x, B) = opt(x-2, A) + b_x$, since we are at machine A at minute $x-2$ & in the process of moving to machine B at minute $x-1$
- By combining both equation, the optimal plan equation on machine B:

  $$opt(x, B) = \max\{opt(x-1, B), opt(x-2, A)\} + b_x$$
- For the algorithm, when $x = 1$, we have

  $$opt(1, A) = \max\{opt(1-1, A), opt(1-2, B)\} + a_1$$
  $$opt(1, A) = \max\{opt(0, A), opt(-1, B)\} + a_1$$
  $$opt(1, A) - a_1 = \max\{opt(0, A), opt(-1, B)\}$$

  $$opt(1, B) = \max\{opt(1-1, B), opt(1-2, A)\} + b_1$$
  $$opt(1, B) = \max\{opt(0, B), opt(-1, A)\} + b_1$$
  $$opt(1, B) - b_1 = \max\{opt(0, B), opt(-1, A)\}$$
- The algorithm computes $opt(x, A)$ & $opt(x, B)$ from $x-2, 3, \dots, n$

Time Analysis

- For each of n-1 iterations, it takes constant time to accomplish that, so the total running time is $O(n)$

4b) cont.

Prove of correctness

- Throughout the entire schedule. There are a total of 4 possible scenarios, 2 for each machines. which are A & B.
- For each machine schedule, we are getting the maximum value between whether we start & end with the same machine or we start & end with a different machine at each minute x.
- For machine A, assume we have the optimal choice for all previous minutes, whether to choose to stay at machine A or switch to machine B for the next minute, we picks the maximum among the two. This is done inductively.
- The same process repeats for machine B
- As a result, we have a optimal plan from machine A & B to get the final optimal schedule.
- The algorithm output the value of an optimal plan

5 - For input, we will have n as the rod length. We also have a array called price[] to store the prices of different pieces.

- We first create a array called val[] with the size of n+1
- Then, initialize val[0] to be zero.
- We will have two counter i & j.
- First, we are looping from $i = 1$ to $i \leq n$
  - create a variable max_value and initialize to to some integer minimum value
  - Loop from $j=1$ to $j \leq i$
    - set the max_value to be the maximum between max_value and the value of (price[j]+val[i-j-1]), which is max_value=max{max_value, price[j]+val[i-j-1]}
  - set val[i] to be max_value
- end of the outer for loop
- return val[n] as the maximum value obtainable by cutting up the rod & selling the pieces.

Time Analysis

- Let $T(n) = T(n-1) + T(n-2) + \ldots + T(1) + 1$ be the equation to comput the time it takes to compute the maximum value obtainable by cutting up rod into n pieces.

- $T(1) = 1$

  $T(2) = T(1) + 1 = 2$

  $T(3) = T(2) + T(1) + 1 = 2 + 1 + 1$

  $\vdots$

  $T(n) = n + n-1 + n-2 + \ldots$ in arithmetic progression.

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

⑤ cont.

Prove of correctness :

- First we will make all the cuts, then we will sell the final pieces.
- At the end of the cutting process, all the lengths of the pieces add up to n. We can always achieve any combination of piece sizes through some kind of sequence cuts
- There is at least one piece inside whichever the combination is the optimal. We can get that piece and denote the length of that piece to be i & sell it.
- For the remaining pieces, they have total length $n-i$ & they are gonna cut in the optimal way for a rod of length $n-i$.
- If the remaining pieces were not cut in an optimal way for a rod of length $n-i$, the original combination could not have been optimal.
- Since the remaining pieces of total length of $n-i$ portion of it could be replaced with the optimal pieces to improve the solution.

(6) - We first create a 2D array table with n as the number of row and the number of column of the table.

- The output of the algorithm will be from diagonal elements up to table[0][n-1]

- We will have 3 counter variables i, j and g

- First, we start to loop from g=0 to g<n

- n is the number of coins in a row & it is a even number

  - Initialize i=0, start loop from j=g to j<n, each iteration, i & j will increment one at a time.

  - initialize 3 variables, x, y, z to be zero

  - if (i+2) is less than equal to j, then update x to be table[i+2][j]

  - if (i+1) is less than equal to j-1, then update y to be table[i+1][j-1]

  - if i is less than equal to j-2, then update z to be table[i][j-2]

  - Assign table[i][j] to be the maximum between coin_array[i] + the minimum value between x & y and coin-array[j] + the minimum value between y & z.

  - end of inner for loop

- end of outer for loop

- return the value from table[0][n-1] as the maximum possible amount of money we can win if we move first.

Time Analysis

- When we are looping from $g=0$ to $g<n$, there are $O(n)$ iterations in the algorithm. Each iteration runs in $O(n)$ time. for computing $table_{i,j}$ values
- The total time for the algorithm takes $O(n^2)$

(6) Prove of Correctness

- Each time, the users have 2 choices to get the maximum possible amount of money the user can win if the user move first.
- The first choice is the user chooses the ith coin with value V_i. The opponent intends to choose the coin that leaves the user with minimum value. The opponent either chooses (i+1)th coin or the jth coin.
- The maximum possible amount of money that user can collect is $V_i + \min(F(i+2, j), F(i+1, j-1))$
- The second choice is the user chooses the jth coin with value V_j. The opponent either chooses ith coin or the (j-1)th coin. The oppoent still intend to choose the coin that leaves the user with the minimum value.
- The maximum possible amount of money that user can collect is $V_j + \min(F(i+1, j-1), F(i, j-2))$
- Therefore, each iteration, $F(i,j)$ denote as the maximum value the user can collect from ith coin to the jth coin.

$$F(i,j) = \max\{V_i + \min(F(i+2, j), F(i+1, j-1)), \\ V_j + \min(F(i+1, j-1), F(i, j-2))\}$$

- Each time, we are taking the maximum of the two choices that user could possibly face when it is the turn to choose coin that maximize the possible amount of money.

- Assume we have the optimal choice for all previous picks, any new picks, we are following the scheme of picking the max of the 2 choices.
- Inductively, our algorithm will output the maximum possible amount of money if we move first.

Add back Base Cases

- If j equal to i, $F(i,j) = V_i$
- If j equal to $i+1$, $F(i,j) = \max(V_i, V_j)$