CS 180 Homework 5

Name: Wang, Zheng
UID: 404855295

CS 180 Dis 1E
Time: F 10:00-11:50
TA: Orpaz Goldstein

Question 1 (closest pair of points):

Explanation:

Suppose that we have set $S = \{p_1: (x_1, y_1), p_2: (x_2, y_2), \dots, p_n: (x_n, y_n)\}$ be the set of all points we are investigating. p_i represent the ith point and (x_i, y_i) represent the coordinate of ith point.

We will first discuss the way to do division.

Assume that we have two list: S_x containing the points p_1 to p_n that are sorted increasingly according to their x coordinates. S_y containing the points p_1 to p_n that are sorted increasingly according to their y coordinates. Also assume each point p_i in both lists, it has a record of its position in S_x and S_y.

Then, suppose we have divided the points in S in to two, call them S_1 and S_2. We divide S according to the following steps:

1. Take the first $\lfloor n/2 \rfloor$ points in S_x as S_1. Take the rest as S_2. In this split, we will maintain S_{1x} for S_1 and S_{2x} for S_2. S_{1x} contains the points in S_1 sorted increasingly by their x coordinate. S_{2x} is defined similarly.
2. Maintain the S_{1y} for S_1 and S_{2y} for S_2. Where S_{1y} contains points in S_1 sorted increasingly by their y coordinate. Similar definition applies of S_{2y} as well.
3. Let each point in S_{1x} and S_{1x} record its position in both lists. Do this for every point in S_{2x} and S_{2y} too.

Notice that we can keep divide until each set is of size less or equal to 3. Then we can find the closes pairs in each of these subsets very easily—just try all possible pairs.

Then, we discuss the way of merging

Now, assume we know that the closest pairs of both S_1 and S_2. For S_1, the closest points are (p_{s1}, p'_{s1}), and the distance between the two point is δ_1. For S_2, the closest points are (p_{s2}, p'_{s2}), and the distance between the two point is δ_2. It is now important to find a linear time algorithm to merge the two set and find the closest pair in $S_1 \cup S_2$.

We first observe that the closest pair in $S_1 \cup S_2$ has three cases:

1. Both come from S_1
2. Both come from S_2
3. One come from S_1, the other come form S_2.

For the first two cases, there is no need for discussion. If we define $\delta = \min\{\delta_1, \delta_2\}$, we simply return the point pair that gives δ.

For the third case, there is some trick we need to use.

First, we define p_x^* being the x coordinate value of the rightmost point in S_{1x}.

Then we define a vertical line L with equation $x = p_x^*$. Thus, we know that L will be separating S_1 and S_2.

We claim that if there exist some point (p_{s1}^*, p_{s2}^*) such that $d(p_{s1}^*, p_{s2}^*) < \delta$ (δ defined in a previous paragraph), then p_{s1}^* and p_{s2}^* both lies within δ distance from L.

Proof.

Suppose towards contradiction that this is not true. Let (x_1, y_1) be the coordinate of p_{s1}^* and (x_2, y_2) be the coordinate of p_{s2}^*. We have $|x_1 - x_2| > 2\delta$. But since $d(p_{s1}^*, p_{s2}^*) \geq |x_1 - x_2| > 2\delta > \delta$, we encountered a contradiction.

Let S_δ be the set of all point all points within δ distance from L. We then define $S_{\delta y}$ being the all points in S_δ sorted according to their y coordinate in increasing order. We claim that if such (p_{s1}^*, p_{s2}^*) exist, in the list $S_{\delta y}$, p_{s1}^* and p_{s2}^* are of maximum 15 points apart.

Proof.

Divide the region within δ distance from L into boxes of side $\frac{\delta}{2} * \frac{\delta}{2}$. Then each box must contain in maximum one point. Otherwise, there exist two points that have distance $\frac{\sqrt{2}}{2}\delta$, contradicting that δ is the minimum distance of points pair from S_1 or S_2.

Thus, without loss of generality, suppose p_{s1}^* has y coordinate larger than p_{s2}^*. If p_{s2}^* is more than 15 points after p_{s1}^* in $S_{\delta y}$, then since every box has at most one point and there 4 boxes in a row, p_{s2} is more than 3 rows above p_{s1}. So $d(p_{s1}, p_{s2}) \geq |y_1 - y_2| \geq \frac{3}{2}\delta$, a contradiction.

By the above claim, for each point in the region within δ distance from L, we only need to check constant number of points. Thus, each merge will take $O(n)$ time.

Algorithm:

```
➢ Define function Get_closest_pair($S_x, S_y$)
    o If $|S_x| \leq 3$
        ▪ Go through all pairs to find the minimum pair in $S_x$ and return that pair
    o Make $S_{1x}$, $S_{2x}$, $S_{1y}, S_{2y}$
    o Set closest_pair1 = Get_closest_pair($S_{1x}, S_{1y}$)
    o Set closest_pair2 = Get_closest_pair($S_{2x}, S_{2y}$)
    o Find $\delta = \min\{ d(\text{closest\_pair2}), d(\text{closest\_pair1}) \}$
    o Find $p_x^*$ = $S_{1x}[\ |S_{1x}|\ ]$
    o Construct $S_\delta$ using $P_x^*, S_{1x}$, and $S_{2x}$.
    o Construct $S_{\delta y}$ using $S_\delta$, $S_{1y}$, and $S_{2y}$
    o For each $p \in S_{\delta y}$
```

```
        ▪ Check 15 points before $p$ and 15 points after $p$ to find the closest pair
          involving $p$, denote $(p, x_{min})$
        ▪ Among all $(p, x_{min})$, find the minimum, denote $(p_{min}, x_{min})$
    o End for
    o If $d(p_{min}, x_{min}) \leq \delta$
        ▪ Return $(p_{min}, x_{min})$
    o Else if $d(\text{closest\_pair2}) < d(\text{closest\_pair1})$
        ▪ Return closest_pair2
    o Else
        ▪ Return closest_pair1
```

- Sort the points to give S_x and S_y
- Call the recursive function: **Get_closest_pair**(S_x, S_y)

Complexity:

1. In each recursive call:

   A. Construct S_{1x}, S_{2x}, S_{1y}, S_{2y} will take $O(n)$ time.

      Proof.
      Constructing S_{1x} and S_{2x}, we simply split the list S_x into two; constructing S_{1y} and S_{2y} will need us to go over the list S_y once. Since each point in S_y has a record of its position in S_x, overall, we check if each element is in S_{1x} or S_{2x} in $O(n)$ time.

   B. Constructing $S_{\delta y}$ will take $O(n)$ time.

      Proof.
      Same reason as above, we check every element in S_{1x} and S_{2x} to identify the points in S_δ. Then, as each point in S_{1x} and S_{2x} knows its position in S_{1y} and S_{2y}, for each point p in S_δ, we can locate it in S_{1y} and S_{2y}. Finally, we merge p's in S_{1y} and S_{2y} as we did in merge sort to get $S_{\delta y}$.

   C. Check every point in the $S_{\delta y}$ to find (p_{min}, x_{min}) will take $O(n)$ time (proof is done in the **explanation** part)

   D. So overall, it will take $O(n)$ time to run each recursive call.

2. Since each recursive call divide the points into two subsets, and there are at most $O(\log n)$ such divide. The overall time complexity of the recursive part will be $O(n \log n)$.
3. Sorting the points into S_x ande S_y will take $O(n \log n)$ time to run.

Thus, the overall time complexity will be $O(n \log n)$.

Question 2 (Exercise 4, page 315):

(a) An example could be $M = 10$ and cost summarized as following

MONTH	1	2	3
NY	**100**	**100**	**95**
SF	**1**	**1**	**100**

In this case using the provided greedy algorithm, it will choose $SF \to SF \to NY$, with total cost $1 + 1 + 10 + 95 = 107$. But the optimal solution is: $SF \to SF \to SF$, with total cost $1 + 1 + 100 = 102$. So, the algorithm provided will fail.

(b) An example could be $M = 10$ and the cost summarized as following:

MONTH	1	2	3	4
NY	**200**	**5**	**200**	**5**
SF	**5**	**200**	**5**	**200**

The optimal solution in this case has cost $5 \times 4 + 10 \times 3 = 50$. It will take 3 moves.

If we choose to move less than 3 times, then it is necessary that we will have to bear a cost of 200, and the total cost will be over 200 as well, so it will not be optimal.

(c) Algorithm (assume we have already had the two lists $\{NY_1, NY_2, \dots, NY_n\}$ and $\{SF_1, SF_2, \dots, SF_n\}$ containing the cost at each city at each month:

```
➢ Initialize two list $SF\_opt$ and $NY\_opt$, each with $n + 1$ elements (index from $0$ to $n$)
➢ Initialize $SF\_opt[0] = 0$ and $NY\_opt[0] = 0$
➢ For $i$ from $1$ to $n$:
    o $SF\_opt[i] = SF_i + \min\{NY\_opt[i-1] + M, SF\_opt[i-1]\}$
    o $NY\_opt[i] = NY_i + \min\{NY\_opt[i-1], SF\_opt[i-1] + M\}$
➢ End for
➢ Return the smaller one of $NY\_opt[n]$ and $SF\_opt[n]$
```

Explanation:

We observe that for each month, there are two cases:

1. We work at SF
2. We work at NY

For each of the cases discussed above, we are either moving from a different location or we are continue working in the same location. Thus, without loss of generality, if we are forced to work in SF this month, we can calculate the optimal we can achieve by taking the minimum of the following:

1. Optimal cost of previous month working in NY + M + the cost we need to work in SF this month (moving from a different location)
2. Optimal cost of previous month working in SF + the cost we need to work in SF this month. (stay the same)

We can continue doing this until the last month. We are guaranteed to have an optimal cost given we work in NY and an optimal cost given we work in

SF. Thus, we will know the overall optimal solution by taking the minimum of the two conditional optimal mentioned above.

Complexity:

1. Since in each iteration of the for loop, we are only doing constant number of things, we will need $O(1)$ to run each iteration of the for loop
2. Since we are running the for loop n times. Thus, it will take $O(n)$ time to complete the loop.

Thus, the overall complexity will be $O(n)$.

Question 3 (Exercise 6, page 317):
The idea of solving this question come from the "Segmented Least Square" Problem. We are basically trying to figure out a way of dividing the wordlist into several lines.

Algorithm:

```
➢ Initialize an array $opt$ with length $n+1$
➢ Set $opt[0] = 0$
➢ Initialize a $n * n$ 2d array $Slack_{sq}$
➢ For $j$ from 1 to $n$:
    o For $1 \leq i \leq j$
        ▪ Calculate $Slack_{sq}[i][j]$ being the **square** of slack of putting word $i$ to $j$ (both inclusive) in one line.
        ▪ If word $i$ to $j$ does not form a valid line, then assign the value to be infinity.
    o End for
➢ End for
➢ For $k = 1$ to $n$:
    o $opt[k] = \min_{1 \leq i \leq k} \{Slack_{sq}[i][k] + opt[i-1]\}$
    o When above minimum is achieved at $i^*$, let $opt[k]$ points to $opt[i^* - 1]$.
➢ End for
➢ Initialize a list $Par$
➢ Backtrack from $opt[n]$. Whenever we encounter $opt[before]$, store $before + 1$ in $Par$.
➢ Output the reverse of $Par$
➢ Return $opt[n]$
```

Explanation:

We notice that for this question, we are simply dividing the word list into lines. Of course, we will have a last line. Let's say that the last line contains words k to n which has slack square $Slack_{sq}[k][n]$. Then, the optimal we can achieve given k is

$$Slack_{sq}[k][n] + opt[k-1]$$

Where $opt[k-1]$ is optimized way of distributing word 1 to word k. But there are many k's we can choose. So, the global optimal should be, as we are supposed to check all possible k:

$$opt[n] = \min_{1 \le k \le n} \{Slack_{sq}[k][n] + opt[k-1]\}$$

Notice that the above equation is a recursive form, and it is true for all $n \ge 1$. The base case will be the $opt[0]$, which is the optimal way of putting 0 word. Of course, since there is no word, we don't need any line and the slack should be 0. Using the idea of dynamic programming, we can start from $opt[0]$, find $opt[1], opt[2], \dots, opt[n]$. To output the actual divide, simply backtrack from $opt[n]$. Whenever the pointer has form $\cdots \leftarrow opt[before] \leftarrow opt[after] \leftarrow \cdots$, we know that we have choose to output word $before + 1$ to word $after$ in a line, so the divide the at $before + 1$.

Complexity:

1. Fill the $Slack_{sq}$ takes $O(n^2)$ time to run.
2. Find $opt[k]$ takes $O(n)$ to run for each k and there are n k's we are investigating. So, the for loop will take $O(n^2)$ time to run.
3. When we backtrack and output the partition point, there can be at most n elements (as there are at most n words). So this step will take $O(n)$ time to run.

Thus, the overall time complexity of this algorithm will be $O(n^2)$.

Question 4 (Exercise 9, page 320):

(a)

An example is summarized in the following table:

Day	1	2	3	4	5
x	10	10	10	10	10
s	5	1	1	1	1

The optimal solution will be to boot the system twice, then maximum we will be able to process 5+5+5 = 15 terabytes of data.

If we are rebooting only once, then we can process in maximum 5+5+1+1 = 13 terabyte of data.

If we are not rebooting the system overall the entire period, then we can process in maximum 5+1+1+1+1 = 9 terabyte of data.

Thus, the best solution will be rebooting the system on day 2 and day 4.

(b)

Explanation.

Directly thinking how much data we can process from days 1 to i based on the optimal solution of days 1 to $i-1$ is hard. But if we add another constrain, it will be relatively easy.

The idea is that if we denote the optimal amount of data we can process from days

1 to i, given that the latest reboot was j days ago, to be $opt[i][j]$, the following will hold:

$$opt[i][j] = opt[i-1][j-1] + \min\{ s_i, x_i \} \quad \text{if } j \neq 0$$
$$opt[i][0] = \max_{0 \le j \le i-1} \{ opt[i-1][j] \}$$

The first equation holds as if we do not reboot on day i, then obviously, the best we can do in terms of processing data is the sum of data we process today and the best we have done from day 1 to day $i-1$, given that that last reboot was $j-1$ days before day $i-1$.
The second equation holds as if we are rebooting on day i, then we are not doing anything on day i. Thus, the best we can do in terms of data processing is the maximum amount of data we can do from day 1 to day $i-1$, which is described by $\max_{0 \le j \le i-1} \{ opt[i-1][j] \}$.

Algorithm:

```
➢ Initialize a $(n+1) * (n+1)$ 2d array $opt$
➢ For $i = 0,1, \dots, n$:
    o $opt[0][i] = 0$
➢ End for
➢ For $i = 1,2,\dots,n$:
    o $opt[i][0] = \max_{0 \le j \le i-1} \{ opt[i-1][j] \}$
    o For $j = 1,2,\dots,n$:
        ▪ $opt[i][j] = opt[i-1][j-1] + \min\{ s_i, x_i \}$
    o End for
➢ End for
➢ Return $\max_{0 \le j \le n} \{ opt[n][j] \}$
```

Complexity:

1. The for loop that initialize the $opt[0][i]$ will take $O(n)$ time to run.
2. In the second for loop will run $O(n)$ times:
   a) The initialize $opt[i][0]$ will taks $O(n)$ time to run
   b) Since each iteration of the nested for loop takes constant number of step; thus the overall complexity of the nested loop is $O(n)$

   Thus, the overall time complexity of the for loop will be $O(n^2)$
3. Find the maximum from $opt[n][j] \quad \forall j$ will take $O(n)$ time to run.

Thus, the overall time complexity of this algorithm will be $O(n^2)$.

Question 5 (cutting rod):
Explanation:

The solution to this problem is very similar to that of knapsack problem. In this case, we will regard the total length of the rod as the capacity of knapsack, and regard the segment's length as the "volume" or the space the segment is going to occupy in the knapsack.
Suppose we have list $\{L_1, L_2, \dots, L_n\}$ containing all length segment we can have. Also, suppose that we have list $\{p_1, p_2, \dots, p_n\}$ where p_i is price of L_i.

Given the above, we are able to summarize how the algorithm work in the following opt table (i.e. a 2d array):
NOTE: in the column ***Segment***, assume L_k means allow segments $L_1, L_2, \dots, L_k$.

Total length / ***Segments***	*0*	*1*	…	*i*	…	*n*
L_0	0	0	…	0	…	0
L_1	0	p_1	…	…	…	…
L_2	0	p_1	…	…	…	…
…	…	…	…	…	…	…
L_j	0	p_1	…	**??**		
…						
L_n						

Let's assume that the anything with coordinate $(\text{Segments} = L_k, \text{total length} = S)$ shaded in grey has already record the maximum value we can obtained by cutting a rod of length S into possible segments $L_1, L_2, \dots, L_k$.
Then, when we are approaching '**??**' position in the table, we have two choice:

1. That we will chose to cut a segment of length L_j
2. We are not doing 1, i.e. the way we are cutting the rod does not result in a segment with length L_j.

For case 1, since we are required to have a segment of length L_j, so, we will cut the L_j out from the rod and we will be resulting with a rod of length $i - L_j$. Then we are looking for optimal way of cutting this $i - L_j$ rod. However, by the assumption we made, the optimal solution of how to cut rod $i - L_j$ has been done for us, and it is at $(\text{Segment} = L_j, \text{total length} = i - L_j)$ position of the table.

For case 2, since we must not include segments with length L_j. Thus, it is equivalent to we do not allow L_j appear in the segments. Again, this optimal solution, by our assumption, can be find the grey area of the table and it is at $(\text{Segment} = L_{j-1}, \text{total length} = i)$ position of the table.

Then, by taking the maximum of the two cases, we obtained our optimal solution at position $(\text{total length} = i, \text{Segment} = L_j)$ of the table. Since this is true for all i and L_j. Thus, we can finally get the desired solution at $(\text{Segment} = L_n, \text{total length} = n)$

Algorithm:
For simplicity, an element in the table is denoted by $opt[*][**]$, $*$ position is the ***Segment***; also, when we say $opt[i][**]$, we are referring to the L_i row of the table. $**$ position is the ***Total length***.

```
➢ Initialize a $(n+1)*(n+1)$ 2d array $opt$
➢ For $i = 0,1,\dots,n$
    o Set $opt[0][i] = 0$
➢ End for
➢ For $i = 1,2,\dots,n$
    o For $k = 0,1,2,\dots,n$
        ▪ If $k - L_i < 0$
            ▪ Case_1 = $0$
        ▪ Else
            ▪ Case_1 = $opt[i][k - L_i] + p_i$
        ▪ $opt[L_i][k] = \max\{opt[i-1][k],$ Case_1$\}$
    o End for
➢ End for
➢ Return $opt[n][n]$
```

Complexity:
1. The first for loop that initialize $opt[0][i]$ $\forall i$ will take $O(n)$ time to run.
2. The second nested for loop has constant number of operations in the inner most loop. Thus, this nested loop will take $O(n^2)$ time to run.

Thus, the overall time complexity of the algorithm will be $O(n^2)$.

Question 5 (coin game):
Explanation:
Let's figure out a way of representing the recursive relationship happen in this question.
Suppose we are at the stage where we have coins $v_i, v_{i+1}, \dots, v_j$ left on the table. Then there are two cases:
1. We take the coin v_i
2. We take the coin v_j

Now, although we current don't know which coin to take, let's assume that the optimal value we can achieve from the coin $v_i, v_{i+1}, \dots, v_j$ is somehow done by a function $opt(i, j)$.
Then, we can define the value we can get for **case 1** is $v_i + \min\{opt(i+2, j), opt(i+1, j-1)\}$. Why we are getting this? Since we are assuming that the opponent will always pick the coin that minimize the money we can win, after the opponent has taken his /her coin, we have two cases:
1. We have coin $v_{i+2}, v_{i+3}, \dots, v_j$ if opponent takes v_{i+1}.
2. We have coin $v_{i+1}, v_{i+2}, \dots, v_{j-1}$, if the opponent takes v_j.

Since we assume that the opponent is very smart, we can face the case that win us less money. Thus, we can in maximum win $v_i + \min\{opt(i+2,j), opt(i+1,j-1)\}$ amount.
With a similar argument, we can see that for **case 2**, the maximum amount of money we can win is $v_j + \min\{opt(i+1,j-1), opt(i,j-2)\}$.
Thus, the optimal we can obtain is the maximum of the two amount we mentioned above.

$$opt(i,j) = \max\{\, v_i + \min\{opt(i+2,j), opt(i+1,j-1)\},\\ v_j + \min\{opt(i+1,j-1), opt(i,j-2)\}\,\}$$

Now, let's consider the base cases of $opt(i,j)$.
Notice that if $i = j$, then we are left with one coin, then we take it, so $opt(i,i) = v_i$.
Also, if $j = i + 1$, then we are left with two coin, then we take the larger coin, so $opt(i, 1+i) = \max\{v_i, v_{i+1}\}$.

Algorithm:
Assume the minimum index is 1; assume the list $v_1, v_2, \dots, v_n$ has been given:

```
➢ Initialize a 2d array $opt_w$ with size $n * n$.
➢ For $i = 1,2,\dots,n$
    o Set $opt_w[i][i] = v_i$
➢ End for
➢ For $i = 1,2,\dots,n-1$
    o Set $opt_w[i][i+1] = \max\{v_i, v_{i+1}\}$
➢ End for
➢ For $(diag = 3;\ diag \le n;\ diag{+}{+})$
    o For $(i = 1, j = diag;\ j \le n;\ i{+}{+},\ j{+}{+})$
        ▪ $opt_w[i][j] = \max\{\, v_i + \min\{opt_w[i+1][j], opt_w[i+1][j-1]\},$
                  $v_j + \min\{opt_w[i+1],[j-1], opt_w[i][j-2]\}\,\}$
    o End for
➢ End for
➢ Return $opt_w[1][n]$
```

NOTICE that we are not going to fill the whose 2d array and only filling the lower triangular part. But this is ok, since

1. we have assumed that $i \le j$
2. we are interested in $opt_w[1][n]$, which at the lower left corner of the 2d array.

Complexity:
It is not very easy to see the complexity of the nested loop as we are introducing some new variable $diag$ (which is used to help us to fill opt_w in diagonal fashion). However, we can analyze the overall complexity with a small trick.

1. We notice that for each $opt_w[i][i]$, we take $O(1)$ time to fill them for all i.
2. Also, we take $O(1)$ time to fill $opt[i][i+1]$, for all possible i. (use one comparison to get the max)
3. For the rest of $opt_w[i][j]$ that haven't been filled, we see that they are filled

by the nested loop. But surely, each $opt_w[i][j]$ will be filled by constant number of operations. Thus, each $opt_w[i][j]$ will take $O(1)$ time to fill.

Now, since in total we are filling about n^2 elements, thus, the overall time complexity is $O(n^2)$.