CS 180 Homework 5

Name: Wang, Zheng CS 180 Dis 1E

UID: 404855295 Time: F 10:00-11:50

TA: Orpaz Goldstein

**Question 1 (closest pair of points):**

**Explanation**:

Suppose that we have set be the set of all points we are investigating. represent the th point and represent the coordinate of th point.

**We will first discuss the way to do division.**

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

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

1. Take the first points in as . Take the rest as . In this split, we will maintain for and for . contains the points in sorted increasingly by their coordinate. is defined similarly.
2. Maintain the for and for . Where contains points in sorted increasingly by their coordinate. Similar definition applies of as well.
3. Let each point in and record its position in both lists. Do this for every point in and 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 and ­. For , the closest points are , and the distance between the two point is . For , the closest points are , and the distance between the two point is . It is now important to find a linear time algorithm to merge the two set and find the closest pair in .

We first observe that the closest pair in has three cases:

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

For the first two cases, there is no need for discussion. If we define , we simply return the point pair that gives .

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

First, we define being the coordinate value of the rightmost point in . Then we define a vertical line with equation . Thus, we know that will be separating and .

We claim that if there exist some point such that ( defined in a previous paragraph), then and both lies within distance from .

*Proof.*

Suppose towards contradiction that this is not true. Let be the coordinate of ­ and be the coordinate of . We have . But since , we encountered a contradiction.

Let be the set of all point all points within distance from . We then define being the all points in sorted according to their coordinate in increasing order. We claim that if such exist, in the list , and are of maximum 15 points apart.

*Proof.*

Divide the region within distance from into boxes of side . Then each box must contain in maximum one point. Otherwise, there exist two points that have distance , contradicting that is the minimum distance of points pair from or .

Thus, without loss of generality, suppose has coordinate larger than . If is more than 15 points after in , then since every box has at most one point and there 4 boxes in a row, is more than 3 rows above . So , a contradiction.

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

**Algorithm**:

* Define function **Get\_closest\_pair**()
  + If
    - Go through all pairs to find the minimum pair in and return that pair
  + Make , ,
  + Set closest\_pair1 = **Get\_closest\_pair**()
  + Set closest\_pair2 = **Get\_closest\_pair**()
  + Find
  + Find =
  + Construct using and .
  + Construct using , , and
  + For each
    - Check 15 points before and 15 points after to find the closest pair involving , denote
    - Among all , find the minimum, denote
  + End for
  + If
    - Return
  + Else if
    - Return
  + Else
    - Return
* Sort the points to give and
* Call the recursive function: **Get\_closest\_pair**()

**Complexity**:

1. In each recursive call:
   1. Construct , , will take time.

*Proof.*

Constructing and , we simply split the list into two; constructing and will need us to go over the list once. Since each point in has a record of its position in , overall, we check if each element is in or in time.

* 1. Constructing will take time.

*Proof.*

Same reason as above, we check every element in and to identify the points in . Then, as each point in and knows its position in and , for each point in , we can locate it in and Finally, we merge ’s in and as we did in merge sort to get .

* 1. Check every point in the to find will take time (proof is done in the **explanation** part)
  2. So overall, it will take time to run each recursive call.

1. Since each recursive call divide the points into two subsets, and there are at most such divide. The overall time complexity of the recursive part will be .
2. Sorting the points into ande will take time to run.

Thus, the overall time complexity will be

**Question 2 (Exercise 4, page 315):**

1. An example could be 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 , with total cost . But the optimal solution is: , with total cost . So, the algorithm provided will fail.

1. An example could be 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 . 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 , and the total cost will be over as well, so it will not be optimal.

1. **Algorithm (assume we have already had the two lists**  **and**  **containing the cost at each city at each month:**

* Initialize two list and , each with elements (index from to )
* Initialize and
* For from to :
* End for
* Return the smaller one of and

**Explanation**:

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

1. We work at
2. We work at

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 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 the cost we need to work in this month (moving from a different location)
2. Optimal cost of previous month working in the cost we need to work in 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 and an optimal cost given we work in . 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 to run each iteration of the for loop
2. Since we are running the for loop times. Thus, it will take time to complete the loop.

Thus, the overall complexity will be .

**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 with length
* Set
* Initialize a 2d array
* For from 1 to :
  + For
    - Calculate being the **square** of slack of putting word to (both inclusive) in one line.
    - If word to does not form a valid line, then assign the value to be infinity.
  + End for
* End for
* For to :
  + When above minimum is achieved at , let points to .
* End for
* Initialize a list
* Backtrack from . Whenever we encounter , store in .
* Output the reverse of
* Return

**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 to which has slack square . Then, the optimal we can achieve given is

Where is optimized way of distributing word to word . But there are many ’s we can choose. So, the global optimal should be, as we are supposed to check all possible :

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

**Complexity**:

1. Fill the takes time to run.
2. Find takes to run for each and there are s we are investigating. So, the for loop will take time to run.
3. When we backtrack and output the partition point, there can be at most elements (as there are at most words). So this step will take time to run.

Thus, the overall time complexity of this algorithm will be .

**Question 4 (Exercise 9, page 320):**

**(a)**

An example is summarized in the following table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Day | 1 | 2 | 3 | 4 | 5 |
|  | 10 | 10 | 10 | 10 | 10 |
|  | 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 to based on the optimal solution of days to 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 to , given that the latest reboot was days ago, to be , the following will hold:

The first equation holds as if we do not reboot on day , 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 to day , given that that last reboot was days before day .

The second equation holds as if we are rebooting on day , then we are not doing anything on day . Thus, the best we can do in terms of data processing is the maximum amount of data we can do from day to day , which is described by .

**Algorithm**:

* Initialize a 2d array
* For :
* End for
* For :
  + For :
  + End for
* End for
* Return

**Complexity**:

1. The for loop that initialize the will take time to run.
2. In the second for loop will run times:
   1. The initialize will taks time to run
   2. Since each iteration of the nested for loop takes constant number of step; thus the overall complexity of the nested loop is

Thus, the overall time complexity of the for loop will be

1. Find the maximum from will take time to run.

Thus, the overall time complexity of this algorithm will be .

**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 containing all length segment we can have. Also, suppose that we have list where is price of .

Given the above, we are able to summarize how the algorithm work in the following table (i.e. a 2d array):

NOTE: in the column ***Segment***, assume means allow segments .

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Total length** | 0 | 1 | … | i | … | n |
| **Segments** |
|  | 0 | 0 | … | 0 | … | 0 |
|  | 0 |  | … | … | … | … |
|  | 0 |  | … | … | … | … |
| … | … | … | … | … | … | … |
|  | 0 |  | … | **??** |  |  |
| … |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Let’s assume that the anything with coordinate shaded in grey has already record the maximum value we can obtained by cutting a rod of length into possible segments .

Then, when we are approaching ‘**??**’ position in the table, we have two choice:

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

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

For case 2, since we must not include segments with length . Thus, it is equivalent to we do not allow appear in the segments. Again, this optimal solution, by our assumption, can be find the grey area of the table and it is at position of the table.

Then, by taking the maximum of the two cases, we obtained our optimal solution at position of the table. Since this is true for all and . Thus, we can finally get the desired solution at

**Algorithm**:

For simplicity, an element in the table is denoted by , position is the ***Segment***; also, when we say , we are referring to the row of the table. position is the ***Total length***.

* Initialize a 2d array
* For
  + Set
* End for
* For
  + For
    - If
      * **Case\_1 =**
    - Else
      * **Case\_1 =**  +
  + End for
* End for
* Return

**Complexity**:

1. The first for loop that initialize will take 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 time to run.

Thus, the overall time complexity of the algorithm will be .

**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 left on the table. Then there are two cases:

1. We take the coin
2. We take the coin

Now, although we current don’t know which coin to take, let’s assume that the optimal value we can achieve from the coin is somehow done by a function .

Then, we can define the value we can get for **case 1** is . 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 if opponent takes .
2. We have coin , if the opponent takes .

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 amount.

With a similar argument, we can see that for **case 2**, the maximum amount of money we can win is .

Thus, the optimal we can obtain is the maximum of the two amount we mentioned above.

Now, let’s consider the base cases of .

Notice that if , then we are left with one coin, then we take it, so .

Also, if , then we are left with two coin, then we take the larger coin, so .

**Algorithm:**

**Assume the minimum index is 1; assume the list has been given**:

* Initialize a 2d array with size .
* For
  + Set
* End for
* For
  + Set
* End for
* For ++
  + For ++, ++
  + End for
* End for
* Return

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
2. we are interested in , 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 (which is used to help us to fill in diagonal fashion). However, we can analyze the overall complexity with a small trick.

1. We notice that for each , we take time to fill them for all .
2. Also, we take time to fill , for all possible . (use one comparison to get the max)
3. For the rest of that haven’t been filled, we see that they are filled by the nested loop. But surely, each will be filled by constant number of operations. Thus, each will take time to fill.

Now, since in total we are filling about elements, thus, the overall time complexity is .