Faith Twardzik Orpaz Goldstein, 1F, 2-3:50 pm

HW5

Question 1)

Managing points in the x and y-dimensions:

- The first thing we do is sort all of the points on our graph into two lists: a list P(x) of points in order of increasing x-coordinate, and a list P(y) of points in order of increasing y-coordinate, where each coordinate in P(x) and P(y) also records the (x, y) position of its corresponding point
- Next, we set up each level of recursion, where the first level's set-up is as follows:
    - Divide the graph into two halves:
        - We do this by consulting the list of all x-values P(x), and selecting the middle element (we can call this mid(x)) to divide the graph. All elements with a lower x-value than mid(x) will correspond to the left half of the graph (we will call this L), and all x-values greater than or equal to mid(x) will correspond to the right half of the graph (R)
        - Now we have two sets of points, one in L and one in R
    - Generate four more lists from the sets L and R and be sure to record the (x, y) position of each corresponding point:
        - Sort the x-coordinates in L in increasing order to generate list L(x)
        - Sort the x-coordinates in R in increasing order to generate list R(x)
        - Sort the y-coordinates in L in increasing order to generate list L(y)
        - Sort the y-coordinates in R in increasing order to generate list R(y)
    - Use these lists to calculate the closest-distance pair in L (closest(L)) and the closest-distance pair in R (closest(R))
    - Compare closest(L) and closest(R) and choose the smaller of the two distances. This will be the delta value.
    - Now, we compare between sets L and R to see if there is a distance between the sets that is smaller than delta
    - Choose the point with the highest x-value in L(x) (call this separate(x)) and we construct the following line between sets L and R using this x-value: x = separate(x)
    - We know that if there is a pair of points which has a distance less than delta, then the points MUST lie within a region that is a distance of less than delta from the line separate(x), so we generate a list B (this stands for Between) that contains all of the points less than or equal to a distance of delta from line separate(x)
    - Sort the y-coordinates in B in increasing order to generate list B(y)
    - For each point corresponding to each y-coordinate in B,
        - Calculate the distance to each of the other 15 points in B which are within a delta distance from separate(x), and record the smallest distance (and its corresponding pair) if it is less than delta
    - If there are distances smaller than delta, compare them and choose the smallest one (and its corresponding pair), otherwise return either closest(L) or closest(R), whichever corresponded to a distance of delta

Time Complexity:

When we construct our original sorted lists P(x) and P(y) of the entire graph, we know that this will take O(nlogn) time. The rest of the algorithm runs with a time complexity of O(n), because we are using the basic mergesort pattern of dividing the sets in linear time and merging them in linear time. For each level of recursion, we do linear time work: just like for mergesort, we have $2^{level\#}$ subproblems that take a maximum of $cn/2^{level\#}$ time, with the level *level#* taking a max time, therefore, of *cn.* So, each level contributes a linear runtime. However, the entire problem is continuously halved *logn* times, so we multiply this by the *cn* time after each level, to get an overarching time complexity of O(nlogn).

Question 2)

Exercise 4, pg. 315

a) The following set of costs, followed by the plan of minimum cost generated by the given algorithm shows that it does *not* produce an optimal solution:

Month 1: NY = 1, SF = 2

Month 2: NY = 2, SF = 1

Month 3: NY = 1, SF = 2

Month 4: NY = 2, SF = 1

$n = 4$, $M = 10$

For this set-up, the algorithm will give the following plan:

[NY, SF, NY, SF], where the cost of this plan is: 1 + 10 + 1 + 10 + 1 + 10 + 1 = 34

However, this is not the optimal plan. The optimal plan is:

[NY, NY, NY, NY], where the cost of this plan is: 1 + 2 + 1 + 2 = 6

b)

Month 1: NY = 4, SF = 10

Month 2: NY = 10, SF = 4

Month 3: NY = 4, SF = 10

Month 4: NY = 10, SF = 4

$n = 4$, $M = 1$

The optimal plan with this set-up is:

[NY, SF, NY, SF], where the cost of the plan is: 4 + 1 + 4 + 1 + 4 + 1 + 4 = 19

This is an example where every optimal plan must move three times. After each month there is a move, because the cost of moving $M = 1$ is less than the cost of staying in NY (6) after the first month. Then the cost of moving (1) is less than the cost of staying in SF (6) after the second month, and so on.

c)

This is going to be a case of dynamic programming in a bottom-up sequence. Essentially, we are recursing to the last level (the last month), and working backwards. We must end either in NY or SF (we don't know which, yet), and we will work backwards by calculating the cost of either spending the previous month in the same location as the ending location, or spending the previous month in the other location + the moving cost. We choose the lesser between the two, and then do the same for the month before that, until we have built a "route" of minimum-cost months. The algorithm is as follows:

- For every month *m*,
    - The optimal plan cost ending in NY = the cost of NY in month *m,* plus the minimum between [the optimal plan cost ending in NY up to month *m-1*] and [the optimal plan cost ending in SF up to month *m-1* + the cost of moving *M*]
    - The optimal plan cost ending in SF = the cost of SF in month *m,* plus the minimum between [the optimal plan cost ending in SF up to month *m-1*] and [the optimal plan cost ending in NY up to month *m-1* + the cost of moving *M*]
- Compare the two, and choose the one with the lower cost

Time Complexity:

We have *m* months that we are iterating over, and for each we are doing an O(1) recursive comparison of whether the previous month's optimal plan cost for the same location vs. the previous month's optimal plan cost for a different location + the moving cost is a minimum, so we end up with a total time complexity of O(m).

Question 3)

Exercise 6, pg. 317

We are attempting to find a partition of *w* words into lines so that the squares of the slacks of the lines are minimized. We use this to develop a backwards-forwards recurrence relation, which will basically have the following intuition: the solution at any line *m* that minimizes the squares of slacks will be the minimum of the slack of the lines for some number *s* up to *m*, plus the optimal solution's minimum cost up to line *s.* Since we are recursively building subproblems, starting with the slack of the last line, which ends with a word *w* (though we do not know *where* it ends yet), and then we use this and the recurrence relation to backwards-solve the our subproblems, which in this case are the set of lines before the last line, then the set of lines before the second-to-last line, and so on. So, we have the following algorithm:

- For each of the *m* lines in the text in ascending order,
    - The optimal solution at line *m* (minimized cost of squares of slacks) is the minimum of the squares of slacks from some line *s* up to *m*, plus the optimal solution at the level before *s* (which is *s – 1*), where we will try this for all values of *s* to produce a minimum
- We return the optimal solution at line *m* after the for loop ends

As we can see, the recurrence relation here is: *OPT[m] = [min of the squares of the slacks from s to m where s produces a minimum] + OPT[s – 1]*

Time Complexity:

Traversing the array backward to reproduce the optimal solution will take a max O(n) time after the for loop. For every value of *m*, we calculate the squares of slacks for every sequence of *s* to *m*, where *s* needs to be minimized and so has the possibility of a max of *m* values. In this way, for every value of *m* we are calculating the squares of slacks a max of *m* times, so this the for loop will run with $O(n^2)$ time complexity, which is the total runtime of the algorithm.

Question 4)

Exercise 9, pg. 320

a)

Day 1: x = 10, s = 10

Day 2: x = 10, s = 4

Day 3: x = 10, s = 3

Day 4: x = 10, s = 2

Day 5: x = 10, s = 1

This is an example of an instance for which there is a surplus of data and the optimal solution reboots the system exactly twice. The optimal solution will be to process 10 terabytes on day 1, reboot on day 2, process 10 terabytes on day 3, reboot on day 4, and process 10 terabytes on day 5, for a total of 30 terabytes processed. This is greater than terabytes processed with only one reboot (27) or no reboots (20).

b)

We are seeking to return an optimal solution that maximizes the number of terabytes processed by the system, given values for *x* and *s*. We can do this by setting up a system of recurrences for different scenarios. As it stands, there are three separate scenarios that can occur on a given day: terabytes are processed, there is a reboot, or it is the first/last day in the sequence and a reboot would not make sense (on the first day, we assume that a reboot just occurred one day prior, because the problem does not specify, and on the last day, since a reboot only maximizes terabyte processing starting the *next* day, and there is no next day on the last day). We break up the recurrences for these three scenarios into the following:

- A normal day of processing. The optimal solution on day *m* will be either *x(m)* or *s(k)*, whichever is smaller (the constraining factor), plus the optimal solution terabytes for the next day *m+1*. We also introduce k, the number of days since a reboot occurred, to keep track of the reboots and the corresponding *s(k)* values.
    - *OPT(m, k) = [the minimum between x(m) and s(k)] + the optimal solution OPT(m+1, k+1)*
- A reboot day. No terabytes are processed, so the optimal solution for a reboot day is just the optimal solution at day *m+1*, whose *s(k)* value will be 1 (there has only been one day since a reboot occurred):
    - *OPT(m, k) = OPT(m+1, 1)*

- The first and last days, in which a reboot does not make any sense. For the last day, we will not add on the optimal solution for the next day (there is no next day), so we just have:
    - *OPT(m, k) = the minimum between x(m) and s(k)*
    - For the first day, we have the same recurrence as for a normal day of processing.

Now that we have our recurrences, we can build the following algorithm to make use of them. We will utilize a double for-loop because as we traverse the days (the *x(m)* values) in decreasing order from the ending day, we traverse the *s(k)* values in increasing order. The algorithm is such:

- Calculate the *OPT* value for the last day, with every possible *s(k)* value (because we do not know on which day after the reboot the sequence will be ending)
- For all *m,* starting from the end and traversing backwards
- For all *k* up until the current value of *m*
    - The optimal solution *OPT* is the maximum of the recurrences for choosing either a normal day of processing or a reboot day, so *OPT(m, k) = the maximum between [[the minimum between x(m) and s(k)] + the optimal solution OPT(m+1, k+1)] and [OPT(m+1, 1)]*
- After the for loops end, we will return the optimal solution generated for the first day based on our backwards traversal

Time Complexity: For every value of *m,* we are recursing in two directions: either choosing a normal day of processing to continue the path, or a reboot day to continue the path, with each generating a different path. So, for every value of *m,* we end up with *m* values, plus the trivial O(1) comparison to find the max between the two calls, so that gives us a total time complexity of $O(m^2)$, or more generally, $O(n^2)$.

Question 5

Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. X is the summation of values on each face when all the dice are thrown. You need to use dynamic programming to solve this problem.

We will use a dynamic programming, top-to-bottom approach in order to solve for all of the different ways of getting the sum *X* with our *n* dice of *m* faces. The intuition for the approach is thus:

- For the base case in which we only have one die and assuming that *X* is less than or equal to *m* (that there is a single face that sums to *X*), we will return 1, as there is only one way to get the sum *X* (a single face with the value *X*)
- The recurrence relation for the case in which we have *n* number of dice is the following:
    - *OPT(n, m, X) = [the summation from i = 1 to the minimum between m and X] of OPT(n – 1, X – i)*
    - Basically, what we are doing here, is finding all of the possible summations given all of the different available values for any given die (*n – 1)*
    - If we had a problem in which *X = 5* and *n = 3* for example, the recursive calls would include *OPT(3, 5) = OPT(2, 4) + OPT(2, 3) + OPT(2, 2) + OPT(2, 1) + OPT(1, 4) …* and so on, with each subproblem including one less die, and one less for the summation *X*

Therefore, the algorithm is as such:

- Initialize a multidimensional array, *OPT,* with values of *n+1* and *x+1* as horizontal and vertical bounds, respectively
- For the case in which we only have one die and *X* is less than or equal to *m*,
    - There is only one way to get *X* – return 1
- With *n* dice, for *i=2* to *n*
    - For *j=1* up until *X*
        - For *k=1 to the minimum between m and j*
            - Calculate the summation of *OPT(n – 1, X – i)* by adding it into the *OPT* array during each iteration of this loop, where the indices will be: *OPT[i][j] = OPT[i][j] + OPT[i-1][j-k],* corresponding to the loop indices
- Return *OPT[i][j]* of the last iteration, where [i] = *n* and [j] = *X*

Time Complexity:

Because we are iterating not only over *n* dice, but also *X* values for the summation for each value of *n,* as well as *m* summations during the innermost loop, we end up with a total time complexity of O(m * n * X).

Question 6)

You are given a set of n types of rectangular 3-D boxes, where the i-th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

Here we will again be using dynamic programming. We notice a few intuitive pieces of information, including: if the dimensions of the lower box must be strictly larger than the higher box, then we can't have the same box type on top of one another (the dimensions will be equal), and given the possibility of rotating the box any of three ways, there are only 3*i boxes/orientation types that can be used (where i is the number of boxes). In addition, the base dimensions of the boxes will decrease as we continue to stack the boxes, so we will need to sort the boxes from largest base to smallest base.

As such, the intuition for our approach will be to maximize the sum of the heights of the boxes that are to be stacked on one another with the added stipulation that the base dimensions for higher boxes must be strictly less than the base dimensions of lower boxes. The algorithm, then, is as follows:

- *Sort the boxes in decreasing order of base dimension*
- *For all values of i, where i = 3*the number of boxes*
- *OPT[i] = 0 when i = 0, else*
- *OPT[i] = the maximum of [the height of box i plus the maximum summation of the height of the stack of boxes up to i – 1] IF box i satisfies the following:*
    - *The dimensions of the base of box i < the dimensions of the top box i-1 AND*
    - *i > i-1*
- *Return OPT[i], which gives us the maximum height of the stack of boxes*

The algorithm runs on the recurrence:

- *OPT(i) = height(i) + maximum summation OPT(i-1) IF dimensions(i) < dimensions(i-1)*

Time Complexity: The sorting of the boxes in decreasing order will require a runtime of O(nlogn). Since for each value of 3*i we are running the for loop 3*i times in order to compare the maximums of stack heights with each of the 3*i possibilities as the current top box, we will have a worst-case total runtime of $O(n^2)$, in the sense that because of the stipulation on strictly decreasing box base dimensions, it will likely be less than this.