CS 180 Homework 5

Prithvi Kannan

405110096

1. Consider the divide and conquer algorithm for finding the closest pair of points. Analyze the time complexity of the algorithm . Include and discuss a detailed discussion of how to manage points in the x-dimension and how to manage (and search) points in the y-dimension. (You should do this without consulting the book or your notes)

The divide and conquer algorithm goes as follows:

> As a pre-processing step, the input array is sorted according to x coordinates.
>
> 1) Find the middle point in the sorted array, we can take P[n/2] as middle point.
>
> 2) Divide the given array in two halves. The first subarray contains points from P[0] to P[n/2]. The second subarray contains points from P[n/2+1] to P[n-1].
>
> 3) Recursively find the smallest distances in both subarrays. Let the distances be dl and dr. Find the minimum of dl and dr. Let the minimum be d.
>
> 4) From the above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from the left half and the other is from the right half. Consider the vertical line passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip[] of all such points.
>
> 5) Sort the array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by recursively sorting and merging.
>
> 6) Find the smallest distance in strip[]. This is tricky. From the first look, it seems to be a O(n^2) step, but it is actually O(n). It can be proved geometrically that for every point in the strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate). See this for more analysis.

We can represent the runtime of the above algorithm using a recurrence relation $T(n)$. In each recursive call we call a $O(nlogn)$ sort, partition the points in $O(n)$, recursively call the algorithm on $n/2$ points, and finally find the closest points in the middle section in $O(n)$. Therefore, we have $T(n) = 2T\left(\frac{n}{2}\right) + O(n) + O(n) + O(nlogn)$ which is equal to $T(n) = 2T(n/2) + O(nlogn)$. Solving this recurrence relation gives $T(n) = O(nlognlogn)$.

This approach starts by considering values sorted by the x coordinate. Since distance is dependent on x and y, we choose arbitrarily to begin with x. As part of the recursion, we assume we already know dl and dr, so now we must only compute distances from l to r. Now, we sort by the y coordinates.

2. Exercise 3 on page 314

a. Consider a graph with the following adjacency list:

v1: v2, v3

v2: v5

v3: v4

v4: v5

The algorithm described would choose to go from v1 → v2 → v5 and return length 2 since at each step, the algorithm picks the vertex connected with lowest index. However, the longest path is length 3 and goes from v1 → v3 → v4 → v5.

b. This can be solved using dynamic programming instead. Let $opt(i)$ represent the maximum length path to a vertex i from vertex 1. An optimal substructure for the problem looks like the following: $opt(i) = \max(opt(j) + 1)$ for all j connected to i. This substructure proves to be correct since there only way to get to i is from a vertex j that is connected to it and taking the maximum path to the vertex j and adding 1 would create the maximum path to i. We have the base case of $opt(1) = 0$ since the path from vertex 1 to vertex 1 is length 0. The pseudocode looks like:

```
initialize an array of size n to hold the length of the longest path from to
each vertex

set opt[1] to 0 since the path from v1 to v1 can be considered 0

loop through the elements in the array

      for all vertices j connected to the current vertex, i

            if opt[i] < opt[j]+1, set opt[i] to opt[j]+1

return the last element in the array
```

The runtime for this algorithm is $O(n^2)$. We loop through all of the elements in the array in time $O(n)$. In the worst case, i is connected to n elements so the inner loop also runs $O(n)$ times. Checking the value at $opt[j]$ takes constant time, so we have $O(n * n) = O(n^2)$

3. Exercise 5 on page 316

The objective is to maximize the sum of the qualities of words. We start by writing the optimal solution recursively. Let $Opt(j)$ represent the maximum quality of words in j letters. $Opt(j) =$ max from $1 \leq k \leq j$ of ($Opt(k-1) + quality(yk \dots yj)$). This recurrence is proven to be correct since it considers all possible ways to break up j letters to maximize the value. We can stop the algorithm at the base case where $Opt(1)$ since we can simply return the $quality(y)$ as the only possible value. From this recursion, we can write an algorithm as follows:

```
for (j = 1 to n) {

   temp = -infinity

   for (k = 1 to j) {
```

```
            if temp < OPT[k-1] + quality(yk...yj)) then {
               temp = OPT[k-1] + quality(yk...yj))
            }
         OPT[j] = temp
      }
```

We use two nested for loops (the outer runs from 1 to length of the string and the inner runs from 1 to the outer index) and all the operations inside the loops take constant time. Note that for each value of j in the outer loop, the inner look is executed j times. Hence, the runtime of the algorithm is $O(n^2)$.

4. Exercise 10 on Page 321

a. The greedy algorithm proposed does not work for the case where minute 1 is 5 for A and 1 for B and minute 2 is 10 for A and 100 for B. The algorithm would choose A for both steps which would sum to 15, but the optimal solution would choose B for both and sum to 100.

b. Consider the optimal solution that ends at A at time t. $Opt(t, A)$ can either have come from A at time $t-1$ or come from B at $t-2$ (since it had to skip $t-1$ to switch over), so $Opt(t, A) = ai + max(Opt(t-1, A), Opt(t-2, B))$. By symmetry, B follows the same equation and $Opt(t, B) = bi + max(Opt(t-1, B), Opt(t-2, A))$.

The base cases for the recurrence relations are $Opt(1, A) = a1$ and $Opt(1, B) = b1$. Now the algorithm computes $Opt(i, A)\ and\ Opt(i, B)$ for all i from 1 through n. Each step of computing opt at an index takes constant time since it only involves looking at two previous stored values in the opt array, therefore the overall runtime of the algorithm is $O(n)$. At the end, we compare $Opt(n, A)$ and $Opt(n, B)$ in constant time and return the larger one.

5. Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces.

We start by constructing the optimal substructure for the problem. The best price for a rod size n, $best(n) = max\ (price[i] + best(n-i-1))\ for\ 0 < i < n-1$. This comes from the fact that we must explore cutting the rod at any size and then exploring any further cuts we can make to the rod to create more value.

The algorithm looks like the following:

```
loop through i values up to n
      loop through j values up to i
            maxTotal = Math.max(maxTotal, price[j] + best[i-j-1]))
return maxTotal
```

The runtime of this algorithm is $O(n^2)$ since we must loop through all possible places to cut the rod and then consider all places to further cut the rod. The lookup of a smaller element in the array, $best[i - j - 1]$, takes $O(1)$ since that value has already been computed.

6. Consider a row of n coins of values $v1, \dots, vn$, where n is even. We play a game against an opponent by alternating turns (you can both see all coins at all times). In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can win if we move first.

At a turn we have two choices: pick the first coin or the last coin. Since none of the middle coins can be selected currently, we can consider those 'hidden'. If we remove the first coin, we 'uncover' the second coin for our opponent in the next turn. If we remove the last coin, we 'uncover' the second to last coin for our opponent in the next turn. Therefore, an optimal substructure must attempt to maximize the sum of the coin we pick up and minimize the potential coin that the opponent can pick up. This looks like the following where $Z(i, j)$ represents the best move from the i'th to j'th coin:

$$Z(i,j) \;=\; Max(Vi \;+\; min(Z\;(i+2,j), Z\;(i+1,j-1)\;), Vj \;+\; min(Z\;(i+1,j-1), Z\;(i,j-2)\;))$$

This algorithm will run in $O(n^2)$ time since we will make an $n\ x\ n$ table to represent each of the starting and ending indexes available to be chosen and we must explore each grid entry. It takes constant time to compute a particular entry, (i, j), since it only involves lookups to indexes that are previously known (larger values of i or smaller values of j)