CS 180, Fall 2019
Problem Set 5
Due November 20, 2019

Tian Ye
UID: 704931660

November 19, 2019

Problem 1

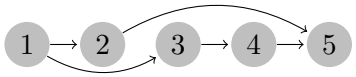
We will construct p_x and p_y, which are points sorted in increasing order of x and y, respectively. If we have less than four points, we can call the following algorithm and find the solution in constant time. If there are at least four points, we will divide the problem in half and recursively call it on both halves. Consider the case where there is a shortest distance pair within the left or right half; the recursive call will find it. The case is that the shortest distance is between the left and the right halves. In this case, the algorithm will search for a minimum pair that falls within a distance d, or the distance of either individual half.

```
1  Sort points in x and y dimension in ascending order
2  Store sorted points into p_x and p_y
3  Create function dst(a , b) that finds distance
4
5  Begin recursive function (p_x, p_y):
6          If p has less than four points
7                  Find and return closest pair via comparison
8  '       Create two havles of points increasing by x and y
9          q_x, q_y, r_x, r_y, divided by a vertical line l
10         (q, q') = Function call (q_x, q_y)
11         (r, r') = Function call (r_x, r_y)
12         Let dist = min(dst(q, q'), dst(r, r'))
13         Construct set S containing points within dist of L
14           ordered by increasing y value
15         For each point in S
16                 Find distance to next 15 points in S
17                 Save minimum pair as s and s'
18         If dst(s, s') < dist
19                 Return (s, s')
20         Return minimum pair between dst(q, q') and dst(r, r'))
```

Proof. The algorithm runs in $O(n \log n)$ time complexity as it first sorts the all the points. It then divides the problem into halves, and at each iteration it does at most $O(n)$ work. This yields a net time complexity of $O(n \log n)$. □

Exercise 3 Page 314

a.



The given algorithm will find L=3, whereas the correct solution is L=4.
b.

Each individual subproblem will consider the levels in the vertices, levels being similar to their meaning in a BFS, where each level will contain all V nodes in the graph. There will be at most $n-1$ levels, any more will imply a cycle. At each level, assuming we have the optimal solution to the previous level, we can use the previous level to solve for the current level's optimal solution. This can be expressed in the following recurrence relation: opt(L, V) = max(opt(L-1, V), max(opt(L-1, V')+1)). V' is some vertex that has an edge to V inside of L-1. To set this up, the base case will have the starting node's opt(0, v_1) = 0, while everything else is -1. This leaves two situation in the recurrence relation: either the longest path from the previous level is used, or it is not used when we are calculating the maximal path. In the former, opt(L, V) = opt(L-1, V')+1, as the previous path is taken and incremented by length of 1. In the latter, opt(L, V) = opt(L-1, V), as we are not including that edge from V' to V.

$\text{opt}(L, V) = \max(\text{opt}(L-1, V), \max(\text{opt}(L-1, V') + 1))$

```
1  Initialize array opt[n-1][n-1] = -1
2  opt[0][1] = 0
3  For (int L = 1; L < n; L++)
4          For (int i = 0; i < n; i++)
5                  For each (v_j that can traverse to v_i)
6                          If (opt[L-1][v_j] >= 0)
7                                  opt[L][v_i] =
8                                    max(opt[L-1][v_i],
9                                     opt[L-1][v_j]+1)
10 return opt[n-1]
```

Proof. The algorithm runs in $O(n^3)$ time complexity. The first loop iterates through $n-2$ elements, while the second and third loops iterate through $n-1$ elements. Furthermore, the third loop performs constant time operations onto the 2-D array. This results in a time complexity of $O(n^3)$. □

Exercise 5 Page 316

Defining opt(i) as the highest quality segmentation from the first to the ith character, we will use the following recurrence relation: opt(i) = $\max_{0 \leq j \leq i}$(quality(substr(j, i)) opt($j-1$)), where we will try all j from 0 to i. To set this up, the base case will have the opt(0) = quality(y_1), the first character. This leaves us with two situations in the recurrance relation: at position j, being some index from 0 to i, we can either choose to create a segment by cutting at j or we do not. In the former, opt(i) = quality(substr(j, i)) + opt(j - 1), as we will take the optimal for the substring that is up to the position of our cut, and add the quality of the new segment. In the latter case, we choose not to create a segment, and continue on to the next j. This case is trivial; hence we assume at every j we create a segmentation. To find the maximal, we consider the possible segmentation combinations by cutting at each j.

opt(i) = $\max_{0 \leq j \leq i}$(quality(substr(j, i)) opt($j-1$))

```
1 Create array opt[n] = 0
2 opt[0] = quality(y_1)
3 For (int i = 1; i < n; i++)
4         For (int j = 0; j<=i; j++)
5                 opt[i] = max(opt[i],
6                   quality(y_j y_j+1 ... y_i) + opt[j-1])
7 Return opt[n-1]
```

Proof. The algorithm runs within $O(n^2)$ time complexity, as the outer loop iterates through all n characters, and the inner loop iterates through i characters, being at worst case n. The comparison within the inner loop is constant. □

Exercise 10 Page 321

a.

The given algorithm finds an answer of 20, while the correct answer is 110.

Minute	1	2	3
A	10	2	5
B	0	100	10

b.

We will build the subproblem using the two following recurrence relations: opt(A, i) = max(a_i + opt(A, i+1), opt(B, i+1)) ; opt(B, i) = max(b_i + opt(B, i+1), opt(A, i+1)). The base case will start at the last minute, in which opt(A, n) = a_n and opt(B, n) = b_n. From here, assuming at time i we have built solutions for opt(A, i+1) and opt(B, i+1), we have two options: to keep performing the job on the current machine or to switch machines. In the former, the optimal solution will be to take the weight of the job on the given machine at minute i and add the optimal solution for the job at minute i+1, being a_i + opt(A, i+1) or b_i + opt(B, i+1) on the respective machines. In the latter, the job is moved to another machine. Since switching the job takes time, the optimal solution at minute i is the same as the optimal solution at minute i+1, being opt(B, i+1) and opt(A, i+1) for machines A and B, respectively. We then choose the maximum of the two possibilities at the very end to see if it is more optimal to start on machine A or B.

opt(A, i) = max(a_i + opt(A, i+1), opt(B, i+1))
opt(B, i) = max(b_i + opt(B, i+1), opt(A, i+1))

```
1  opt_A[n-1], opt_B[n-1]
2  opt_A[n-1] = a_n
3  opt_B[n-1] = b_n
4  For (int i = n-1; i > 0; i++)
5          opt_A[i] = max(a_i + opt_A[i+1], opt_B[i+1])
6          opt_B[i] = max(b_i + opt_B[i+1], opt_A[i+1])
7  Return max(opt_A[1], opt_B[1])
```

Proof. The algorithm performs two linear scans on an array of length n, resulting in a time complexity of $O(n)$. The other operations are constant time complexity, yielding an overall time complexity of $O(n)$. □

Exercise 5

We will use a recurrence relation that considers the solutions to all the subproblems from length 1 to i-1 to solve for the optimal subrod lengths at length i. We will define this as opt(i) = $\max_{1 \leq j \leq i}$(Price(j) + opt(i-j)) for all j from 1 to i. This yields a base case of opt(0) = 0 and opt(1) = Price(1), since a rod of length 0 will be worth 0 and there is only 1 choice for a rod of length 1. This yields the following possibilities: either we choose to cut the rod at length j or we choose not to. In the former, we are left with a rod of length j and the remainder being a length of i-j. Since we have subproblems that are solved from length 1 to i, this can be represented as opt(i) = Price(j) + opt(i - j). In the latter case, we choose not to cut the rod. This is a trivial case; we always cut the rod to consider all possibilities. This search is exhaustive and will cover all possibilities.

opt(i) = $\max_{1 \leq j \leq i}$(Price(j) + opt(i-j))

```
1 opt[n] = 0
2 opt[1] = Price(1)
3 For (int i = 2; i <= n; i++)
4         For (int j = 1; j <= n; j++)
5                 opt[i] = max(opt[i], Price(j) + opt[i-j])
6 Return opt[n]
```

Proof. The time complexity is $O(n^2)$, as the outer loop iterates through a maximum of n elements, and the inner loop iterates through another n elements, for a total of n^2 elements in the worst cast. The operations within each iteration are constant. □

Exercise 6

We will set v_i as the value of the coin at index i and v_j as the value of the coin at index j. We build the following recurrence relation: 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))). Our base case will be that opt(n, n)=v_n. To prove the correctness of this algorithm, suppose we are considering the optimal solution where we can choose the coin at index i and the coin at index j, where i $\leq$ j. We also assume we have created the optimal choices from 0 to j-1 and i+1 to n. At this point, there are two choices: the player chooses to take the front or the end of the row. In the former, the player chooses the front, denoted by i. We then assume the other player is playing optimally as well, and therefore we take the minimum of what is possible to get after the other player chooses. This is expressed as opt(i, j) = v_i + min(opt(i, j-2), opt(i+1, j-1)). In the latter, the situation is the converse of the former. This is expressed as opt(i, j) = v_j + min(opt(i, j-2), opt(i+1, j-1)). We then combine these two and find the maximum at every iteration to solve for the optimal solution.

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)))

```
1  opt[n+1][n+1] = 0
2  opt[n][n] = v_n
3  For (int i = n-1; i > 0; i++)
4          For (int j = i; j <= n; j++)
5                  If i == j
6                          opt[i][j] = v_i
7                  Else
8                          opt(i, j) = max(v_i + min(opt(i+2, j),
9                            opt(i+1, j-1)), v_j+min(opt(i+1, j-1),
10                           opt(i, j-2)))
11 Return opt[1][n]
```

Proof. The time complexity is $O(n^2)$, as the outer loop iterates through a maximum of n elements, and the inner loop iterates through another n elements, for a total of n^2 elements in the worst cast. The operations within each iteration are constant □