Question 1)

Algorithm Idea: We are given values i and j in the form of an adjacency matrix where both are value of n. We will assume that we are given an adjacency matrix with no empty spots in them. We will also assume that within the adjacency matrix, anytime there is no edge between (i,j), there will be a ∞. Given G in the form of a adjacency matrix, where i in the matrix will represent rows and j will represent columns. We will also have another adjacency matrix T with size of i x j which is also n x n, which will be used to store the final graph’s path. We also make two int values, a and b where a will be the start S value and b will be used later. We now make a for loop which starts at 1 and and keeps going until it reaches the end of i. We make another for loop within that for loop with the same thing expect instead of the end of i, it keeps going until it hits the end of j. We first check if we are dealing with the first column during each row. If so, we can set a to be our c(i,1) since this will be our first stop in the row. Now, we check if a is infinity. If so, we ignore it since that means the edge between c(i,1) doesn’t exists. If it’s not, then we place a into T[x;1]. If we aren’t dealing with the first column, we continue on. We set b to be the current c(i,j) value. We check if b is infinity. If so, we ignore it since that means the edge between c(i,j) doesn’t exists. If it’s not, we check if a == 0 since that means you are on the same spot and you want to move away from that spot. Set a to be b and remove remove T[x,y-1] and palce a into T[x,y]. We do this because we want to keep the smaller value as our path, so we want to remove the greater value and add in the new value, since the new value is the smaller value. If a isn’t 0, we continue on. Now we check if a is greater than or equal to b. If so, then we set a to be b and do the whole removing and placing process. However, if a is less than b, then we keep a. We don’t need to remove and add anything since the spot of a is already there and there was no change to a. After all that, we return the matrix.

Algorithm Detail:

Given i and j are values of n

Given an adjacency matrix G of size i x j where it’s not empty and no edges between i,j will be filled with ∞

Matrix T of size i x j

Int a

Int b

For x = 1; i <= i; x++

For y = 1; y <= j; y++

If y = 1

a = current c(i,1) value

if a == ∞

ignore a since edge between c(i,1) doesn’t exist

else

place a into T[x;1]

else

b = current c(i,j) value

if b == ∞

ignore b since edge between c(i,j) doesn’t exist

else

if a == 0

a = b

remove T[x,y-1]

input a into T[x,y]

else (a != 0)

if a => b

a = b

remove T[x,y-1]

input a into T[x,y] else if a < b

a = a

Print T

Proof Idea: My algorithm follows Prim’s algorithm of choosing the smallest value between the known edges. It also deals with an adjacency matrix and produces an adjacency matrix of the smallest value path. I will also sum up each part of my algorithm to show that it runs in O(n2).

Runtime Analysis: I will prove that my algorithm runs in O(n2). Creating an empty matrix of n x n and two int values runs in O(1). I have O(1) + O(1) + O(1) = O(3) = O(1). Since I have a nested for loop, that nested for loop becomes O(n2). Within the nested for loop, I have small if statements that run in O(1) and removing and adding stuff into the matrix on a certain location that runs in O(1) as well. So within the for loop, I have O(1) + O(1) + O(1) = O(3) = O(1). So in total, the nested for loop becomes O(n2) \* O(1) = O(n2). When I sum everything up, I get O(1) + O(n2) = O(n2)

Sources: Piazza post, lecture notes 26 and 27.

Question 2)

Part a)

Algorithm Detail:

Given integers a and n where both are non-negative

int x = 1 where x is the final value

if n == 0

x = x

else (n>0)

for i = 1; i <= n; i++

x = x \* a

return x

Part b)

Algorithm Idea: The problem gives us that int a and n are non-negative integers. We first create a function which will deal with our recursion. I will call the function power with two parameters a and n, where both of them are given, as stated in the problem. We create an int x where x will be the final value and another int t, which will be our temp value for each recursion. We now check if n is equal to 0. If so, then x is equal to 1 since anything to the zero power is 1. Now, on to the recursion. We first set t to be power(a,(n/2)This causes the recursion to happen. Now, we check if n/2 is an even integer. If so, then we have t = power(a,n/2) and then x will be t \* t, since both sides are exactly the same. If n/2 is an odd integer, x = t \* t and then x = a \* x. The reason we do a \* x is because if n/2 is an odd number, one side will be greater than the other by a size of one and the division algorithm only calculates the number before the decimal. For example, 23/2 is 12.5 but the division algorithm only shows 12. So to fix this, we just multiply another set of a after each odd n/s recursion. After this, we return x.

Algorithm Detail:

Given integers a and n where both are non-negative

int x where x is the final value

int t where t is the temp value for each recursion

if n == 0

x = 1

t = power (a, n/2)

if n%2 == 0 (check if n is even)

x = t \* t

else (n is odd)

x = t \* t

x = x \* a

return x

Proof Idea: My algorithm follows everything that the question is asking for. It is an algorithm which takes in two values, a and n and produces the produce of a^n. It also uses recursive divide and conquer algorithm to solve this. I will sum up each part of my algorithm to show that it runs in O(logn)

Runtime Analysis: I will prove that my algorithm runs in O(logn). Creating the two int instances is only O(1), and since we have two, it becomes O(1) + O(1) = O(2) = O(1). Anything in constant value will be O(1) time. Next, since my algorithm is running a divide and conquer, I divide everything in half, and don’t worry about one of the halves since it’s exactly the same as the other. I can drop one of the halves and just work on the one half that I still have. I keep dividing and dropping until I get the base case. From there, since I know each halves are exactly the same as each other, I use a temp value to store the value of that half and just multiple that temp value by itself over and over again to get my desired value. This method of divide and conquer runs in O(logn). The reason my divide and conquer algorithm is in O(logn) and not higher is because we get to assume that all multiplication statements between two numbers runs in O(1) and the if statements that I am using are just checking for odd or even. That runs in O(1) as well. So that becomes O(1) \* O(1) \* O(logn) = O(logn).

Source: Lecture notes 28 and 29, textbook page 218.