Problem 1

Here, there is a recursive call which adds d[i] to both a and b and finding the minimum value between them. Therefore, this question breaks down the question into 2 parts, adding d[i] to a and b. The comparison part (finding the minimum takes constant time, c). In addition, here, the recursive call reduces the problem size simply by 1 since we are considering all the values from d[1…n] one by one.

So, the runtime of this algorithm is…

T(n) = 2 \* T(n-1) + c, where c is the constant time.

= 4 \* T(n-2) + 2c

= 8 \* T(n-3) + 3c

=…

= 2^k \* T(n-k) + kc

This function terminates when n-k = 0, when there are no more elements to plug in from the array d.

When n-k = 0, k = n.

When we substitute k = n, we get

= 2^n \* T(0) + nc

At T(0), at the base case, we are simply getting the maximum between a and b, so it is also a constant time.

= 2^n + c1 + nc

The value 2^n grows faster than n \* c since exponential functions, as limit goes to infinity, grows faster than polynomial functions.

Therefore, the running time of this function is O(2^n) .

Problem 2

1. (f(n/2))^3 grows faster.

If f(n) = 2^n, f(n/2) = 2^(n/2) and f(n/3) = 2^(n/3).

If we compute (f(n/2))^3 and (f(n/3))^2, we get 2^(n/2) \* 2^(n/2) \* 2^(n/2) and 2^(n/3) \* 2^(n/3), respectively.

If we simplify them, we get 2^(3n/2) and 2^(2n/3). As n goes to infinity or n grows larger, the value of 3n/2 is greater than 2n/3. Therefore, 2^(3n/2) grows faster and (f(n/2))^3 grows faster.

1. N^3 grows faster.

Considering the term 5^2 + 5^3 + … + 5^log(n), it is simply a geometric series with the rate r =5 and the first term = 25 and n = log(n). When we put this into a geometric function, we get a(r^n-1)/(r-1) = 25(5^logn – 1)/(5-1). If we simplify the function above, we get 25\*(5^logn -1)/4 and we can see that this function varies on the value 5^log(n). In other words, we are comparing 5^log(n) with n^3. According to the logarithm theorem, 5^log(n) = n^log(5). Assuming that log5 is log base 2 of 5, if we calculate log5, we get 2.32, which is n^2.32. When we calculate n^3 and n^2.32, the value of n^3 is larger as n gets large.

1. n\* log(log(n)) is faster.

If we consider both functions, they are multiplied by the value n, so we can factor out n from both sides, allowing us to compare simply 2+log(n)/n^.5 with log(log(n)). Polynomial functions, such as n^0.5 grows faster than a logarithmic function, so log(n)/n^.5 as n grows to infinity is 0, giving 2 + log(n)/n^.5 = 2 as n grows larger. Now, as n grows larger in the function log(log(n)), log(log(n)) goes to infinity. If we compare infinity to 2, infinity is higher, so n \* log(log(n)) is faster.

1. 4^log(n) grows faster.

If we consider 4^log(n), we can rewrite it as 2^2log(n) since 4 = 2^2. Now, if we consider n^1.5, we can rewrite this function as 2^1.5log(n). Since the base of both functions is 2, we can compare 2log(n) with 1.5log(n). Since the value log(n) is the same, we only compare the coefficient and 2 is greater than 1.5. Therefore, 2^2log(n), which is 4^log(n) grows faster.

1. 1.2^(n-1) + n^.5 + log(n) grows faster.

If we consider the function 1.2^(n-1) + n^.5 + log(n), even though as n goes to infinity, all of the individual functions 1.2^(n-1), n^.5, log(n) go to infinity, 1.2^(n-1) grows the fastest. Now, on the other side, there is simply n^1.5. By definition of limits, it is true that exponential values with base greater than 1 always grows much faster than polynomial functions. Therefore, 1.2^(n-1) grows faster than n^1.5 as n grows larger, giving 1.2^(n-1) + n^.5 + log(n) a faster growth.

Problem 3

In this question, graph G has exactly two components that are made up of n nodes. We can now divide the n nodes into two different components: i and n-i, where i is greater than 0. If the value of i is 0 or less than 0, the question does not make sense because if i equals 0, then we have only one component and if i is negative, we have negative amount of nodes, which does not make sense. The two components have edges that cannot be removed. According to piazza, if we remove one more edge, then there are three components, which breaks the question. This means there are minimum number of edges that connect the nodes n-i and i. When there are x nodes, the minimum number of edges required to connect all the x nodes together is x-1 edges, where 2 of the nodes out of x nodes have only one path (end nodes or endpoints) and the other x-2 nodes have two paths that connect all the nodes together. Then, in the component n-i nodes, if we want to have minimum number of edges, the number of edges is simply n-i-1 and for i nodes, if we want to have minimum number of edges, the number of edges is i-1. If we add the number of edges together, we get n – i – 1 + i – 1, which is n-2 edges, once we simplify it. Therefore, the number of edges that are required in the structure of the question is n-2 edges, where n is the number of nodes.

Problem 4

1. We can turn this question into a directed unweighted graph G. This function will simply take the graph G and the course c in C that we are trying to find the pre-required courses. The nodes (v0,…,vn) represent each individual class and the edges represent the path to take the each proposed major class. There is an edge (vi,vj) if i < j and the class in node i is a pre-required course for the class in node j. Once you construct the correct directed unweighted graph, we can do topological sorting using depth-first search, which gives the topological sorting in reverse direction, according to the lecture slide 02\_04\_graphhs.pdf slide 25. After getting the topological order in an array of nodes, we can go through each node in the array and end the function when we find the course c in C, the desired major course. We can do repeat this until we find every course c in C.

Proof: If we connect the graph in the order so that pre-required classes point to major classes and return (or print) until the major class c is reached by using topological order, the topological order will get the pre-required classes first (nodes that have 0 indegree values – which are pre-required classes). In this case, we do not need to worry about cycles since we will always get a topological order. In order to get a cycle, course A has to require course B and course B has to require course A. In the course world, such thing is impossible. Even though irrelevant classes might also be included before the class c is returned in the topological order, those classes do not matter and does not affect the policy since you are qualified for the course due to taking the pre-required classes. For example, if you are a BU student trying to take CS 112, you have to take CS 111. It does not matter whether you took CAS MA 124, CAS PY 211, or etc. As long as you took CS 111, you can take CS 112. The similar approach is done here. It does not matter what courses you took, but as long as you took the required course for course c, you can take course c.

Runtime: Our algorithm does depth-first search, which is O(n+m). n is the number of nodes and m is the number of edges. In this question, every course c in C has pre-required courses B, giving c1B1 + c2B2 +…+cnBn, which are the number of nodes. If c1 requires B1 courses, then B1 courses will point towards c1, giving the number of edges as B1. Following the format, the number of edges is B1+..+Bn, giving the runtime O(n + m) = O(C+B)

1. In terms of C and B, if we change the runtime, the number of nodes is n = c1B1 + c2B2 +…+cnBn and number of edges is B1+…+Bn. Therefore, the runtime is O((c1B1 + c2B2 +…+cnBn+ B1+…+Bn), giving O(C+B).
2. This question is simply asking how many ranks or stages there are in the topological order. To calculate this, we can take the algorithm in part (a) and add a variable count so that every time we go through each loop in DFS, we add 1 to count and return the value count.

Proof: If we can take as many classes as possible in a semester, we can take all the courses that are the basic pre-requirements and go up that are the next-requirements, and follow this pattern. For example, if you need CS 111 and MA 124 to take CS 112 and need to take CS 101 to take CS 131 (for example) and need CS 112 and CS 131 to take CS237, you can take CS 111, MA 124, CS 101 in the first semester, which are the requirements for the next stage, CS 112 and CS 131. Then at the next semester, since you fulfilled the requirements, you can take CS 112 and CS 131. At the final semester, you can take CS 237. To fulfill the pre-requirements B to take class c in C and those requirements for the pre-requirement courses B are fulfilled, you can take class all the pre-requirement courses B together, which is done on doing one loop inside the DFS search. Therefore, the number of loops running inside DFS is the minimum number of semesters you need.

Runtime: This question is basically running the DFS algorithm, which is O(C+B).

Problem 5

1. True

Minimum spanning tree is a tree that consists of all nodes in the graph with the lowest weight possible. Since we are not adding 1 to some weights but are adding to all weights in the tree, the algorithm that calculates the minimum spanning tree will be the same for G and G’. The one with smallest edge in G will also be smallest in G’ since all weights are added by 1. Similarly, the weights selected in G will also be selected in G’ since they are the minimum weights of G’, despite the fact that 1 is added from G. Therefore, the answer is true.

1. False

I will provide a counter-example to prove that the statement here is false. Consider a graph with nodes a, b, and c where there exists edges between (a,b), (a,c), and (b,c). In other words, there is a cycle. In this case, let the weight of edge (a,b) = 10, weight of edge(a,c) = 25, and weight of edge(b,c) = 15. Now, if we want to find distance from node a to node c, we can calculate it by either going from node a to b to c or node a to c directly. If we are going from node a to b to c, we have to add the two weights (a,b) and (b,c), giving 10 + 15 = 25. If we want to calculate directly from a to c, we need (a,c) = 25. Since the two values are the same, the minimum cost from going a to c is 25 in G.

Now, if we add 1 to all edges in G and transform into G’, we get (a,b) = 11, (a,c) = 26, and (b,c) = 16. Here, if we want to go from node a to c again, there are two paths. If we follow node a to b to c, we get 11+16 = 27, but if we go from node a to c directly, we get 26. This happens because you add one additional 1 if the path consists of two edges. Therefore, the path from node a to b to c is not the shortest path anymore in G, giving this statement false.

Problem 6

(a)

In this question, the algorithm provided is a greedy algorithm and therefore, finds the optimal solution to the problem.

Here, even though it is not stated on the algorithm, it is best to sort the array in terms of the starting time of the jobs (according to Piazza we need to sort it).

We need to sort it according to starting time because our algorithm bases the number of jobs on the starting time of each jobs.

If we consider two jobs x and y, x is the ith job while y is the jth job, where j = i + 1.

As mentioned in the problem, there are three special cases for the jobs.

1. Job x and job y are disjoint
2. Job x completely contains job y
3. Job x and y are not disjoint and job x does not completely contain job y – this means that job x partially contains job y

If two jobs x and y are disjoint, it means that the starting time of job y is greater than or equal to the finish time of x (si <= fj). This means that all jobs (except the job x itself) have the starting time that is greater than the finishing time of job x. Therefore, job x should be included in the schedule set S. Here is the proof that the solution is optimal. Let’s say the set of jobs without job x, in this case, that can be completed is A. Since job x is not interfering with any jobs in the set A, job x should also be added to A, and therefore gives an optimal solution. After you complete it with job x, job y can be recursed to continue with the next set of jobs.

If job x completely contains job y, it means that the finishing time of job x is greater than the starting time and the finishing time of job y. In this case, we have to discard x in the optimal solution. This is because if job x completely contains job y, every other jobs that have interval problems with y will also have problems with x. In other words, there may be more than simply job y that is completely in the job x, so the optimal solution is to discard job x and recurse on y to continue.

Now, the one remaining factor is job x partially contains job y. In this case, it means that the finishing time of job x is greater than the starting time of job y but the finishing time of job x is less than the finishing time of job y. In other words, only some interval interferes between jobs x and y and job x ends faster than job y. There are two possible choices here: either to discard job x or discard job y since you cannot add both jobs to the set due to the overlapping. In this algorithm, we choose to add job x to the schedule set S because if jobs that come afterwards have partial interval problem with y, the problem can be far apart from finding an optimal solution, so we discard y and continue to recurse on x.

(b)

The running time of this algorithm is O(nlog(n)).

The sorting of the starting time requires nlog(n), according to the lecture slide that discussed the greedy algorithm.

The recursive part is here: the if-elif-else statements that are done take constant time c, which is irrelevant in finding the running time. The algorithm runs recursively and each step takes two jobs with one of the two jobs reappearing in the next recursive step. Therefore, there are total of n-1 recursive steps, assuming that n is the length of possible job list given. (if there are 4 jobs a,b,c,d- let’s say we do jobs a and b first, then jobs a and c, then jobs c and d – four jobs and three recursive steps in this example) Multiplying c with (n-1), we get c(n-1), which is still O(n).

Adding O(n) to O(nlog(n)) gives O(nlog(n)) since nlog(n) grows faster than n as n goes to infinity so we can disregard the n part

Runtime: O(nlog(n))

Problem 7

This question asks whether there is a subset of terminal points that have connectivity, where the size of the subset is greater than n/2 in less than n^2 time complexity.

We can do this with divide and conquer, using recursion, similar to the merge algorithm.

We first assume that we can rewrite as n = 2^r for some r, without loss of generality.

Then, we can break the set T that has length 2^r into two different subsets with equal length until there exists one element in the subsets (the base case).

As we merge those subsets together in the form where we merge the consecutive pairs (if there are n1, n2, n3, n4 subsets – do n1 and n2 to produce n12, do n3 and n4 to produce n34, and do n1234).

As we merge those subsets together, we can compare whether each of the two elements have connectivity. If the elements have connectivity, we can store it inside a set A. If you repeat this accordingly until the full size initial set T is reached, set A will have the terminal points that have connectivity within the set T. Finally, we can compute whether the length of set A and find that it is greater than n/2 in a simple one command comparison.

This is a divide and conquer problem.

Proof:

We divide the set T into two different subsets t1 and t2 with equal lengths (we assumed) until the base case of subset tn equals the size of 1. When this is reached, we now merge the data and compute whether they have connectivity and store it inside a different set variable A that gets carried on until the full set size n of T is reached. When T is reached, the set A will have all the connectivity of terminal points that were stored in each step of the climbing up stage to compare with n/2 to check whether it is greater or not. The running time of significantly less than n^2 will be shown below in the runtime analysis.

Runtime:

This algorithm breaks the set T that they receive into two equal (try to be equal) sizes until base case of each length of set T equal to 1 is reached. When these two sets merge, we do the comparison to find connectivity between terminal points, which takes O(n) time. At the end, we do a simple length comparison between the set A and n/2, which is constant time c, which does not affect the big-oh value. Therefore, this question can be written as

T(n) = 2T(n/2) + O(n) + c (only done once)

= 4T(n/4) + 2O(n)

= 8T(n/8) + 3O(n)

= …

= 2^kT(n/2^k) + kO(n) where it stops when n/2^k = 1 and T(1) = 1 (simply return the set in the base case)

n/2^k = 1 gives k = log(base 2) of n

If we substitute k = log(n),

= 2^log(n)\*T(1) + log(n) \* O(n)

= 2^log(n) + log(n) \* O(n)

= n + nlog(n)

In this case, the time complexity is n\*log(n) instead of n^2.

Problem 8

g(n) = 5g(n/2) grows faster than f(n) = n^2 + 4 \* f(n/2).

f(n) = n^2 + 4 \* f(n/2)

= 2\*n^2 + 16\*f(n/4)

= 3\*n^2 + 48\*f(n/8)

=…

=kn^2+4^k\*f(n/2^k) and this function stops when n/2^k= 1.

When n/2^k = 1, k = log(n) (the log here is log base 2).

When we substitute k = log(n), we get

= log(n) \* n^2 + 4^log(n) \* f(1) --- f(1) is 1 as given, so

= log(n) \* n^2 + 4^log(n)

= n^2\*log(n) + n^log(4)

= n^2\*log(n) + n^2

Since we have n^2\*log(n) + n^2, we get O(n^2\*log(n)) since n^2\*log(n) grows faster than n^2 due to the log(n) value being multiplied.

g(n) = 5 \* g(n/2)

= 25 \* g(n/4)

= 125 \* g(n/8)

= 5^k \* g(n/2^k) and this function stops when n/2^k= 1.

When n/2^k = 1, k = log(n) (the log here is log base 2).

When we substitute k = log(n), we get

= 5^log(n) \* g(1) --- g(1) is 1 as given, so

= 5^log(n)

= n^log(5) --- log (base 2) of 5 is 2.32 when calculated.

= n^2.32

Therefore, we get O(n^2.32).

Now, if we compare O(n^2.32) and O(n^2\*log(n)), we can see that n^2.32 grows faster. This is because when we compare them, we can delete the common terms, which is n^2, giving O(n^.32) and O(log(n)). By definition of limits, it is true that polynomial functions grow faster than logarithmic functions. Therefore, n^0.32 > log(n) gives O(n^2.32) a faster growth and function g(n) = 5g(n/2) a faster growth.