Isaac Delgado
4189182267

1. Suppose we define a new kind of directed graph in which positive weights are assigned to the vertices but not to the edges. If the length of a path is defined by the total weight of all nodes on the path, describe an algorithm that finds the shortest path between two given points A and B within this graph.

Since we must traverse through a connecting edge to get to a vertex we can modify our graph by simply assigning our vertex weights to the preceding connected edge thus removing the vertex weights. We must exclude our initial starting point, in this case point A since there is no preceding edge. We then can run Dijkstra's algorithm from our starting point A and find the shortest path to point B. Lastly we can add the weight of our initial point A to the path cost to get the exact path cost. Although the question only asks for the shortest path and not the exact path adding the initial vertex weight will not make a difference since it all wights are positive.

2. A new startup FastRoute wants to route information along a path in a communication network, represented as a graph. Each vertex represents a router and each edge a wire between routers. The wires are weighted by the maximum bandwidth they can support. FastRoute comes to you and asks you to develop an algorithm to find the path with maximum bandwidth from any source s to any destination t. As you would expect, the bandwidth of a path is the minimum of the bandwidths of the edges on that path; the minimum edge is the bottleneck. Explain how to modify Dijkstra's algorithm to do this.

Since the bandwidth of a path in our network is the minimum of bandwidth edge along our path then we want to maximize our minimum bandwidth edge. We can take this logic into our modified Dijkstra's Algorithm as the following;

We're are only focused on minimums so in our relaxation step we will take the minimum between our total bandwidth at our current node and the bandwidth of the edge to the adjacent node. After this our greedy choice will be to take the largest our minimum bandwidth adjacent nodes.

1.) Start with vertex s, add it to an empty tree T. This will be the root of T.
2.) Expand T by adding a vertex from $V \setminus T$ having the MAXIMUM path length from vertex s.
Thus we will be using a max-heap in this case.
3.) Update distances from vertex s too adjacent vertices in BY THEIR MINIMUM PATH.
Thus we will initialize our nodes to 0 instead of infinity except for our starting point

More concisely putted, we will put $\min(d(u), w(u, g))$ into our max-heap, where $d(u)$ is the bandwidth from our source s and $w(u, g)$ is the bandwidth between nodes u and g

Isaac Delgado
4189182267

3. Solve the following recurrences by the master theorem

$T(n) = 3\ T(n\ 2) + n$ log

$c = \log_b(a) = \log_2(3)$
Leaves: $n^{\log_2(3)}$
Internal: $n\log_2 n$
Case 1: Since $n^{\log_2(3)}$ is a power function greater than 1 thus grows faster than $n\log_2 n$
Therefore: $\theta(n^{\log_2(3)})$

$T(n) = 8\ T(n\ 6) + \log n$

$c = \log_b(a) = \log_6(8) = x \qquad 1 < x < 2$
Leaves: $n^{\log_6(8)}$
Internal: $\log_2 n$
Case 1: Since $n^{\log_6(8)}$ is a power function greater than 1 thus grows faster than $n\log_2 n$
Therefore: $\theta(n^{\log_6(8)})$

$T(n) = \sqrt{7}\ T(n\ 2) + n\ \sqrt{3}$

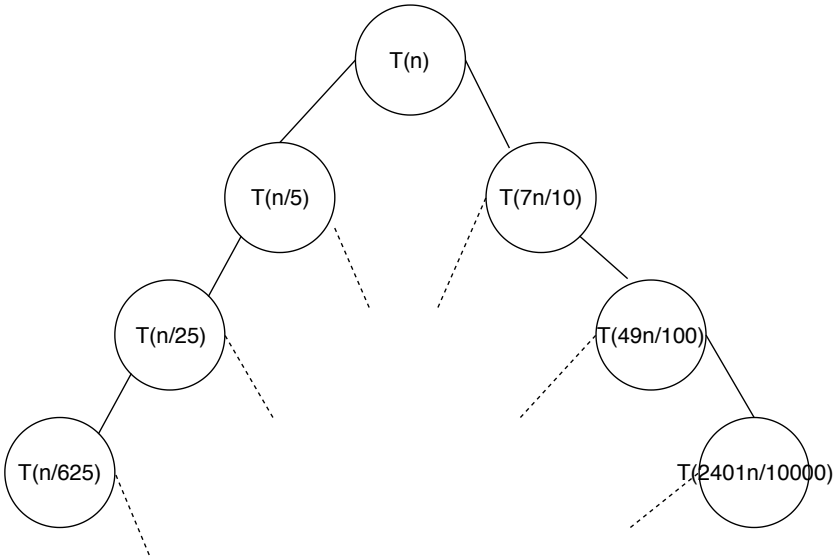
$c = \log_b(a) = \log_2(\sqrt{7}) = 1/2^* \log_2(7) = 1.40$
Leaves: $n^{\log_2(\sqrt{7})}$
Internal: $n^{\sqrt{3}}$
Case 3: Since $\sqrt{3} = 1.732$, which is bigger than $\log_2(\sqrt{7})$
Therefore: $\theta(n^{\sqrt{3}})$

$T(n) = 10\ T(n\ 2) + 2$

$c = \log_b(a) = \log_2(10)$
Leaves: $n^{\log_2(10)}$
Internal: 2^n
Case 3: Since internal is exponential
Therefore: $\theta(2^n)$

Isaac Delgado
4189182267

4. Solve the following recurrence by the recursive tree method



Height of recurrence tree is $\log_b(n)$

Left side of tree height: $\log_5(n)$
Right side of tree height: $\log_{10/7}(n)$

Work at each node is n, so work on left and right side are $n\log_5(n)$ and $n\log_{10/7}(n)$ respectively.
Right side of tree is bigger.

Therefore: $\theta(n\log_{10/7}(n))$

Isaac Delgado
4189182267

5. We know that binary search on a sorted array of size n takes O(log n) time. Design a similar divide-and-conquer algorithm for searching in a sorted singly linked list of size n.

   A. Describe the steps of your algorithm in plain English

      We will first search through our linked-list for our middle element. We can use a two pointer method which will take O(n) to find our element. Then we compare it with our search item which takes O(1). We adjust our starting pointer or last pointer according to the comparison, consequently deleting/ignoring half of the linked list. This pointer adjustment will take O(1). Continue this till we have found or exhausted our choices in the linked-list.

   B. Write a recurrence equation for the runtime complexity

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

   C. Solve the equation by the master theorem

      $c = \log_b(a) = \log_2(1)$

      Leaves: $n^{\log_2(1)}$
      Internal: n

      Case 3; Therefore: $\theta(n)$

6. Given a sorted array of n integers that has been rotated an unknown number of times, give an O(log n) algorithm that finds an element in the resulting array. Note, after a single rotation, the array is not sorted anymore, so we cannot use the binary search. An example of rotations: given a sorted array A = [1, 3, 5, 7, 11], after first rotation A= [3, 5, 7,11, 1], after second rotation A= [5, 7, 11, 1, 3].

Since the array will originally be sorted, the first and the last element will always be next to each other (If these two elements are not together then the item can be searched using normal binary search which will take O(log n)). We must first find the location of where the original first element and the original last element are. This is done by using binary search on the original smallest and original biggest integers. From there we can split our rotted array into two subarrays and consequently we can use a regular binary search on these two subarrays to find our item.

Binary Search to find where the original first element and the original last element meet: O(logn)
Binary Search in subarrays: O(log n)

Therefore: O(logn)