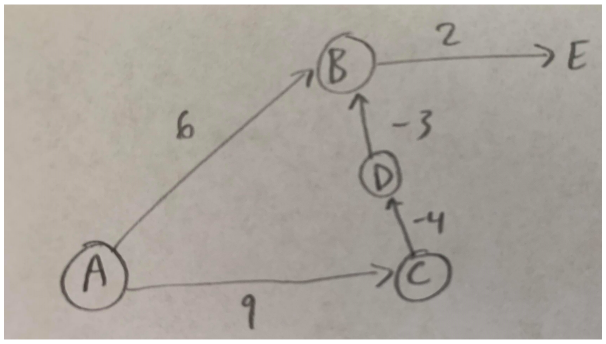
Ethan Wong
COM SCI 180
Discussion 1B
UID 305319001

Homework 3

1. Dijkstra's algorithm does not do well when there are negative-weight edges within the graph that it is inspecting. This is because when the algorithm marks a vertex as closed, that vertex will no longer appear in the open set of vertices to be analyzed. An example of a graph with negative edges (but no negative cycle) that would cause Dijkstra's algorithm to fail is as follows:



The runthrough of Dijkstra's algorithm will be like this (start node is A, end node is E):

- Start at A with current path length 0. We can reach B with distance 6, or C with distance 9. SInce 6 < 9, we choose to traverse to B.
- We are now at B with path length 6. The only possible option is to traverse to E. The path from B to E is weight 2. Therefore the length of the path is now 8.
- The algorithm will return the shortest path as [A-B-E] with length of 8.

This runthrough of the algorithm is flawed as it doesn't think about the negative weight paths. Because the algorithm chose to take the path from A to B, it never investigates what happens after traversing to C. This is because once the path from A-B is found, A will be closed, and the algorithm will not check A again. This means the algorithm never investigates the path A-C. Traversing to C would have a runthrough like this:

- Start at A with current path length 0. We traverse to C with distance 9.
- We are now at C with path length 9. The only possible option is to traverse to D. The path from C to D is weight -4. Therefore the length of the path is now 5.
- We are now at D with path length 5. The only possible option is to traverse to B. The path from D to B is weight -3. Therefore the length of the path is now 2.
- We are now at B with path length 2. The only possible option is to traverse to E. The path from B to E is weight 2. Therefore the length of the path is now 4.
- The shortest path is [A-C-D-B-E] with length of 4.

As seen from this example, Dijkstra’s algorithm will return a path with length 8 for the given graph, while there can be a shorter path of length 4 if the negative edges are traversed. Therefore, Dijkstra’s algorithm will fail in this example where there are edges of negative weight even if there is no negative cycle.

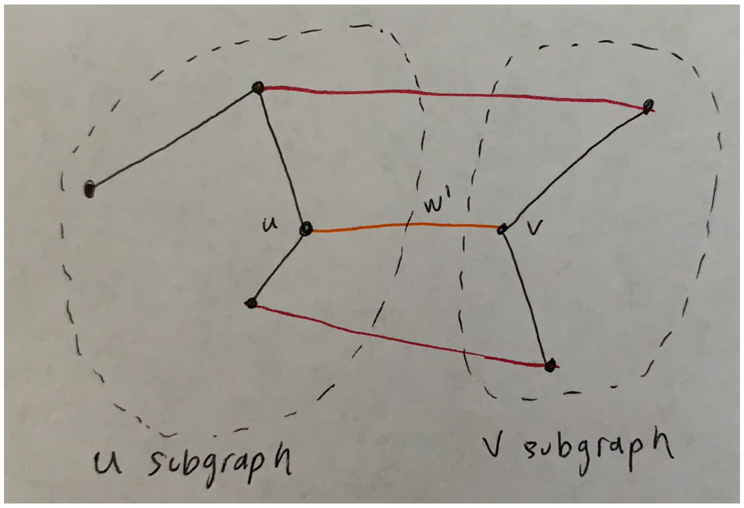
2. The pseudocode for the algorithm is as follows:

```
verifyMST(G, T, E, w'):
  G = undirected weighted graph (given as input to algorithm)
  T = minimum spanning tree of G (given as input to algorithm)
  E = edge between node u and node v (given as input to algorithm)
  w' = new weight for E (given as input to algorithm)

  Establish a cut C = (u,v) in T
  T should now be split into two distinct subgraphs; there is a cut between u and v
  uNodes = nodes from the u subgraph
  vNodes = nodes from the v subgraph
  CS = [ ]
  if len(uNodes) < len(vNodes):
    for u in uNodes:
      if u is connected to the v subgraph
      v = node in v subgraph that u is connected to
      CS.append((u, v))  // establish the cut-set (edges connecting u and v subgraphs)
  else:
    for v in vNodes:
      if v is connected to the u subgraph
      u = node in u subgraph that v is connected to
      CS.append((v, u))  // establish the cut-set (edges connecting v and u subgraphs)
  minimum = min(CS)     //find smallest edge of the cut set
  if minimum == w':
    return true
  return false
```

The purpose of this algorithm is to check whether changing the weight of an edge E in a MST from w to w' will invalidate the original MST. If w' <= w, then there is no need to worry — it will either be a MST of less weight or remain as the same MST. The main case to worry about is when w' > w.

The algorithm takes advantage of the Cut Property of Graphs: For any cut C of a graph, if the weight of an edge e in the cut-set of CS is strictly smaller than the weights of all other edges of the cut-set of CS, then this edge belongs to all MSTs of the graph. First, the algorithm establishes a cut between u and v so that there are two subgraphs. According to the Cut Property, the cut-set must be found. Once the cut-set has been established, the algorithm finds the minimum edge from the cut-set. If this minimum edge is w', then that means that w' must be a part of the MST and therefore the algorithm returns true. Otherwise, the algorithm will return false. Here is a visualization of what the subgraphs and the cut set would look like:



The colored edges represent the cut-set of the two subgraphs. The dotted lines represent the u subgraph and the v subgraph respectively. Assuming that the edges are drawn to scale, then in this case the algorithm would return true because w' is the minimum edge of the cut set.

The time complexity of this algorithm is O(m) because the for loop will choose to iterate on the subgraph with the fewer number of nodes. This ensures that the algorithm will run in linear time and be O(m). The process of finding the minimum cut from the cutset is also linear as it is an iteration through a list, so it is also O(m). Overall the algorithm is O(m).

3. The solutions for both parts are as follows:
    - We can prove that the weight of the optimal tour is at least the weight of the MST using proof by contradiction.
        - Say that the weight of the optimal tour is less than the weight of the MST.
        - The traveling salesman is trying to find the lowest weight tour of the graph such that every node is visited at least once (edges and vertices can be repeated).
        - The minimum spanning tree is the subset of edges that connects all the vertices of a graph together with the minimum possible weight and without any cycles.
        - Given these definitions, it is impossible for the salesman to visit every node with a path that has a lesser weight than the MST. We assumed in the beginning that the salesman's optimal tour was less than the weight of the MST. This is a direct contradiction to the definition of the MST, because then the salesman's tour that visits all the vertices would then be the real MST. Therefore, the salesman's optimal tour would have an equivalent or greater weight than that of the MST.

An algorithm for the Traveling Salesman Problem is as follows:

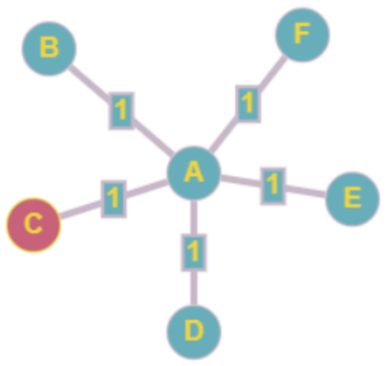
```
TSP(G, S):
  G = MST of the graph that the salesman wants to traverse (given as input to algorithm)
  S = current node that the salesman is at (given as input to the algorithm)
  P = ordered stack that holds all the nodes that the salesman traverses
  Tbv = stack of edges to nodes to be visited by the salesman
  V = list of already visited nodes
  Push (S, S) onto Tbv
  Add S to V

  while (Tbv is not empty):
    (Start, End) = top(Tbv)
    if not currently at End:
      traverse to End
      add End to V
      add End to P
    push all edges directly connected to S that are not in V onto Tbv
    If there are no such edges:
      traverse back to Start
      pop(Tbv)   //pop the top of the stack off
      add Start to P
  return P as the path the salesman travels
```

This algorithm is not guaranteed to find the optimal tour for the traveling salesman, but it is guaranteed to find a tour that visits every node with the cost at most twice the cost of the MST. This is due to the fact that the salesman starts off with the minimum spanning tree of the graph he wants to traverse. This MST is found using Prim's algorithm and given as input to the TSP algorithm. Now that the salesman is traversing the MST instead of the entire graph, it ensures that he can visit all the nodes with a lower cost than aimlessly traversing the original graph.

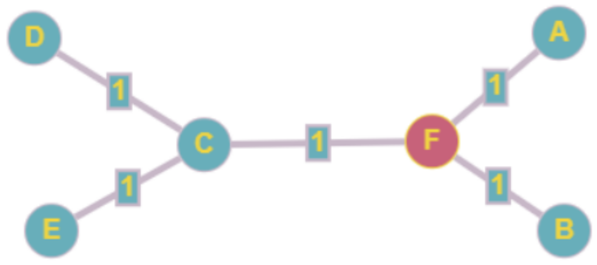
The idea behind the algorithm is to have the salesman start at a certain point of the MST. All edges adjacent to this starting point are added to a stack, and the salesman will follow each path one at a time to reach new nodes. If the new node has adjacent edges that contain unvisited nodes, these edges get added to the stack and the salesman will traverse them as well to reach these unvisited nodes. If the new node has no adjacent nodes that contain unvisited nodes, the salesman will simply backtrack to the node that he came from and investigate one of the other edges by looking at the edge at the top of the stack. In a way, the traversal is similar to DFS. All the while, the salesman will keep track of when he visits an unvisited node for the first time with the list V, and keep track of the path he traverses with the list P. When there are no more unvisited nodes, the stack will be empty and the algorithm will finish, causing the salesman's path P to get returned.

This algorithm is guaranteed to visit every node with the cost at most twice the cost of the MST because in the worst case scenario, the salesman will just travel back and forth from the starting point to visit every node. A simple example of this is as follows:



In the above example, it does not matter where the salesman starts at, but the total cost of the path that the TSP algorithm outputs will be 10. Say the salesman starts at C, then the path P would be something like [C, A, D, A, E, A, F, A, B, A, C] for a path of cost 10. This is twice the cost of the MST, which falls within the constraints of the problem.

A more complex example, of the algorithm runthrough is as follows:

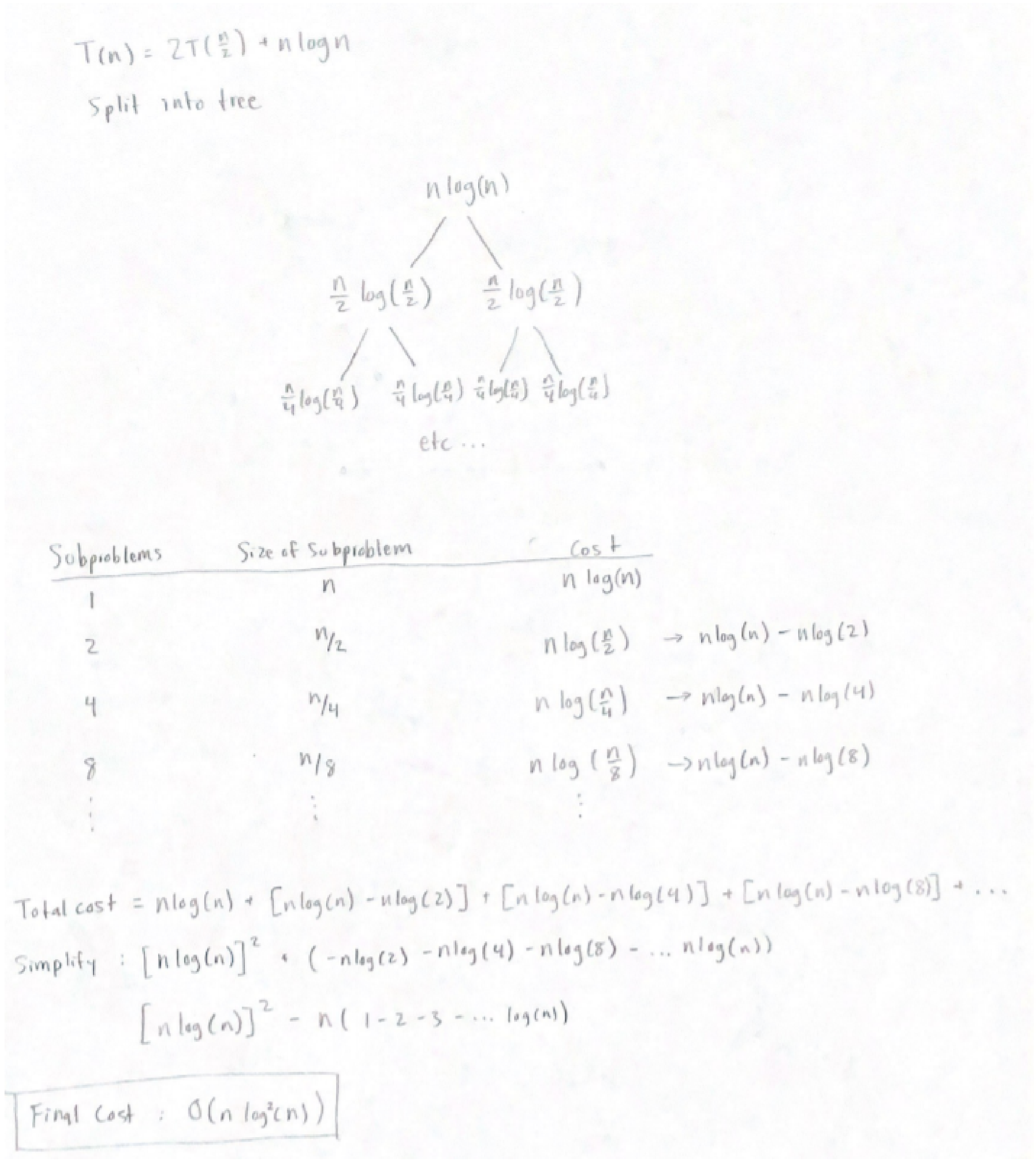


This traversal is a bit more complicated than the previous example due to the fact that both F and C have multiple nodes they are adjacent to, while the previous example only had one node with multiple adjacent nodes. Say the salesman starts at F, then the path P would be something like [F, A, F, B, F, C, D, C, E, C, F] for a path of cost 10.

An example of a scenario where the path is less than double the cost of the MST would be as follows. Say the salesman starts at A, then the path P would be something like [A, F, B, F, C, D, C, E, C, F] for a path of cost 9. Both of these examples fall within the constraints of the problem.

There is a more logical proof of the fact that the path will never exceed twice the cost of the MST. Say we define a subtree as a portion of the MST that contains a node that has multiple adjacent nodes. For example, in the above diagram there would be a subtree consisting of [F, A, B] and another consisting of [C, D, E]. The maximum cost of traversing the subtree would be if the salesman started at the root node. For instance, if the salesman started at F he would have to traverse a path like [F, A, F, B, F] for a cost of 4, as he has to reach the root node again to reach the other subtree. If the salesman started at A, he would have to traverse a path like [A, F, B, F] for a cost of 3. Therefore, the maximum weight of traversing a subtree is double the cost of all its edges. The edge(s) that connect these subtrees (in this case, the edge [F, C]) would have to be traversed a maximum of two times. This means that for any given MST, the salesman would have to traverse each of the subtrees and each of the edges connecting the subtrees a maximum of two times, for an overall maximum of twice the cost of the MST.

4. The order of this relation can be derived by either the general method or the Master Theorem. This first image details the process of deriving the order by using the general method covered in class. The final order is $O(nlog^2(n))$.



$T(n) = 2T(\frac{n}{2}) + n\log n$

Split into tree

$$\begin{tikzcd}[column sep=tiny]
 & & & n\log(n) \arrow[dll, no head] \arrow[drr, no head] & & & \\
 & \frac{n}{2}\log(\frac{n}{2}) \arrow[dl, no head] \arrow[dr, no head] & & & & \frac{n}{2}\log(\frac{n}{2}) \arrow[dl, no head] \arrow[dr, no head] & \\
\frac{n}{4}\log(\frac{n}{4}) & & \frac{n}{4}\log(\frac{n}{4}) & & \frac{n}{4}\log(\frac{n}{4}) & & \frac{n}{4}\log(\frac{n}{4})
\end{tikzcd}$$

etc ...

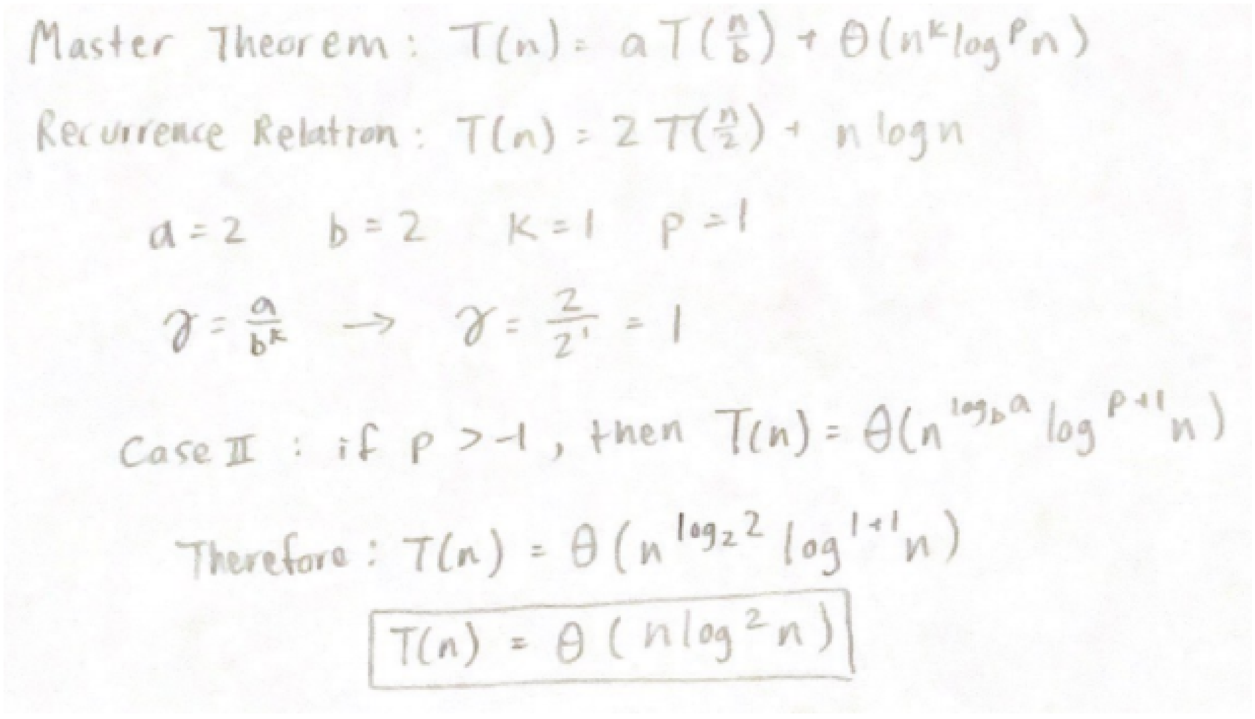
Subproblems	Size of Subproblem	Cost	
1	n	$n\log(n)$	
2	$n/2$	$n\log(\frac{n}{2})$	$\rightarrow n\log(n) - n\log(2)$
4	$n/4$	$n\log(\frac{n}{4})$	$\rightarrow n\log(n) - n\log(4)$
8	$n/8$	$n\log(\frac{n}{8})$	$\rightarrow n\log(n) - n\log(8)$
$\vdots$	$\vdots$	$\vdots$	

Total cost $= n\log(n) + [n\log(n) - n\log(2)] + [n\log(n) - n\log(4)] + [n\log(n) - n\log(8)] + \dots$

Simplify : $[n\log(n)]^2 + (-n\log(2) - n\log(4) - n\log(8) - \dots n\log(n))$

$[n\log(n)]^2 - n(1 - 2 - 3 - \dots \log(n))$

$\boxed{\text{Final Cost} : O(n\log^2(n))}$



Master Theorem: $T(n) = aT(\frac{n}{b}) + \theta(n^k \log^p n)$

Recurrence Relation: $T(n) = 2T(\frac{n}{2}) + n \log n$

$a = 2 \quad b = 2 \quad k = 1 \quad p = 1$

$\gamma = \frac{a}{b^k} \rightarrow \gamma = \frac{2}{2^1} = 1$

Case II : if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

Therefore : $T(n) = \theta(n^{\log_2 2} \log^{1+1} n)$

$$\boxed{T(n) = \theta(n \log^2 n)}$$

This method shows how to solve the order of the recurrence relation via the Master Theorem. The Master Theorem was covered in lecture, but I had to use a more complex version of it in order to solve this problem. The result is the same as the general method, but the Master Theorem is significantly simpler to use. (I realize that I used the θsymbol in the Master Theorem version — please just treat it as an O).

Note: Sorry this is handwritten, it was just easier to write it out this way for formatting and typesetting reasons.

5. A straightforward way to approach this problem would be to use the divide-and-conquer method. The pseudocode for the algorithm is as follows:
(Note: for simplicity of the algorithm, I am going to have the arrays be indexed starting at 1, instead of starting at 0 — this makes the arithmetic simpler)

This algorithm should be called as majority(A, A.length(), 1) in order to work as expected.

```
majority(A, n, s):
  A = array of objects (given as input to algorithm)
  n = number of objects in A (given as input to algorithm)
  s = starting index for array to begin investigation (given as input to algorithm)
  if (n == 1):          //there's only one element in the array
    return A[1]
  mid  = (s + n) / 2          //if mid ends in .5, round down
  maj1 = (A, mid, s)
  maj2 = (A, n, mid + 1)
  if (maj1 == maj2):          //majority element in both halves of array are the same
    return maj1
  else if (maj1 != null):
    temp = number of times maj1 appears in A
    if (temp > [(n + 1 - s) / 2]:        //if right side of inequality ends in .5, round up
      return maj1
    else:
      temp = number of times maj2 appears in A
      if (temp > [(n + 1 - s) / 2]:      //if right side of inequality ends in .5, round up
        return maj2
      else:
        return null
  else:  //else if (maj1 == null)
    temp = number of times maj2 appears in the subarray A[s:n]
      if (temp > [(n + 1 - s) / 2]:      //if right side of inequality ends in .5, round up
        return maj2
      else:
        return null
```

The idea behind this algorithm is to split the array in half and find the majority element of each half using recursion. Once the majority element of each half has been found, they must be compared. If both of the majority elements are the same, then that majority element can be returned as the majority element of the entire array A. If a majority element (maj1) was found for the first half of the array, then the algorithm checks if the total number of times that maj1 appears in the array is greater than the size of half of the array. If this is the case, maj1 is returned as the majority element of A. If this is not the case, then the same process is repeated for the majority element of the second half of the array (maj2). Finally, if maj1 was found to be null, then the algorithm checks if the total number of times that maj2 appears in the array is greater than the size of half of the array. If this is the case, then maj2 is returned as the majority element of A. If this was not the case, then null is returned as there was no majority element found for A. Divide and conquer paired with recursion works very well here because it can keep splitting the problem into smaller ones until a solution is found.

Overall the algorithm runs in O(nlogn) time. This is because the process of finding the total number of times an element appears in an array is an O(n) operation. There are also two subproblems as the algorithm divides the array in half. Each of these half-arrays is a subproblem of size n/2. Therefore the recurrence relation is $T(n) = 2T(n/2) + O(n)$, which simplifies to O(nlogn).