Homework 1

CSOR W4231 - Analysis of Algorithms

Chandan Suri

UNI - CS4090

Collaborators: Arnavi Chedda (amc2476), Parth Jawale (pcj2105), Gursifath Bhasin (gb2760), Uday Theja (um2162)

Submitted on -

February 21, 2022

Problem 1

Problem Description:
Input: A sorted array A of n distinct integers.
Output: An index i such that A[i] = i, if one exists; -1, otherwise.

Solution:
Description of the Algorithm: I am using a variation of the Binary Search algorithm here because of the reason that the array given is sorted and has distinct elements. Due to this property of the input array, I can create some conditions, using which I can divide the array in two parts and search over only one of them at each point in the iteration.
Firstly, I go to the middle index of the array and see if the element at the middle of the array is equal to the index itself. If it is so, then I return that index there itself.
For the other cases, till I have not found such an element in the array or if I haven't exhausted my search space, I check if the element at the middle of the array is less than the middle index at that point in the iterative procedure. If it is less than the index, then it would mean that the element we are trying to find will not be present on the left hand side of the array due to the array being sorted and elements being distinct. Thus, I update the start index to $mid + 1$. Otherwise, (when element at the middle of the array is greater than the middle index) the element we are trying to find, will not be present on the right hand side of the middle index . Thus, I update the end index to $mid - 1$.
Using these conditions at each point in the iteration reduces my search space to half and thus, make the searching algorithm an efficient one. Also, if no such element can be found, I return -1 at the end of the algorithm.
Pseudocode:

Algorithm 1: Algorithm to find i such that A[i] = i

```
$start \leftarrow 0$;
$end \leftarrow length(A) - 1$;
while $start < end$ do
    $mid = start + \frac{end-start}{2}$;
    if $A[mid] == mid$ then
        return $mid$;
    else if $A[mid] < mid$ then
        $start = mid + 1$;
    else
        $end = mid - 1$;
end while
return $-1$;
```

Correctness of the Algorithm: For proving the correctness of the algorithm, we will look upon the modifications done in the original binary search algorithm which is basically a divide and conquer strategy.

Base Case: If the number of elements equals 1, and the value of the element present is 0, then my algorithm returns the index 0 as A[0] = 0. Otherwise, if the value of the element is anything other than zero, then my algorithm returns -1, signifying that an index i such that A[i] = i does not exist.

For cases when $n > 1$, we will look at multiple cases:

1. When A[mid] = mid, then we have reached the correct answer and we can return the index *mid*.
2. When $A[mid] < mid$, we move forward with only the right part (after the mid element) of the array to search for the element. We discard the left part of the array as it's certain that the required index cannot lie in the left part of the array since all the elements in the array are unique integers. Moreover, if the value A[mid] is lesser than the *mid* index, then value of all the elements of the array on the left hand side of the middle element A[mid], i.e., A[mid-1], A[mid-2],...., A[0] cannot equal the indices mid-1, mid-2,...., 0. These will always be lesser than the indices, since the array must have distinct elements and is also sorted.
3. For $A[mid] > mid$, we move forward with only the left part (before the mid element) of the array to search for the mid element. We discard the right part of the array as it's certain that the required index cannot lie in the right part of the array since all the elements in the array are unique integers. Moreover, if the value A[mid] is greater than the *mid* index, then value of all the elements of the array on the right hand side of the middle element A[mid], i.e., A[mid+1], A[mid+2],...., A[n-1] cannot equal the indices mid+1, mid+2,...., n-1. These will always be greater than the indices, since the array must have distinct elements and is also sorted.

If in case, none of these cases hold true and we come out of the loop, we simply return -1, indicating that such an element does not exist in the input array. Hence, we have proved the correctness of the algorithm.

Time Complexity Analysis: I will analyze the time complexity of the algorithm by noting down the time taken to execute each of the lines separately. Post which, I will combine the time complexities for each of the lines to compute the total time complexity of the complete algorithm.
Let the time taken by the algorithm on an array of input size n be equal to $T(n)$.
Complexities for different lines is shown below:
Lines 1-2 take constant time to execute as it is just initialization.
Furthermore, at each iteration of the loop, we discard one half of the array and thus, only

one half of the array is selected. Consequently, line 3 will be executed $T(\frac{n}{2})$ times.
Within each call of the while loop, lines 4-11 take constant O(1) time.
Line 12 (return statement) is a constant O(1) time operation.
Finally, the total time taken by the algorithm can be determined by adding up the individual times of all the operations discussed above:
$T(n) = T(\frac{n}{2}) + O(1)$
Next, using the master theorem, we know that:

$$\text{T(n)} = \begin{cases} \mathcal{O}(n^{\log_b a}) & \text{if } a > b^k \\ \mathcal{O}(n^k \log n) & \text{if } a = b^k \\ \mathcal{O}(n^k) & \text{if } a < b^k \end{cases}$$

In our case, a = 1, b = 2, k = 0. So, $a = b^k$ condition holds. Therefore, we will use the second case of the master theorem. Thus, we have $T(n) = \mathcal{O}(\log n)$.
So, the worst case time complexity of this algorithm is $\mathcal{O}(\log n)$.

Problem 2

(a)

Problem Description:
Input: An unsorted array A of n integers.
Output: Elements x, y $\in$ A such that $\mid x - y \mid$ is maximum.

Solution:
Description of the Algorithm: Assumption: There are at least two elements in the input array.
For this algorithm, to maximize the absolute difference of the elements in the unsorted array provided, we will have to find the global minimum and global maximum of the array. Thus, for doing so I loop over the array once maintaining two variables holding the global minimum (min_value) and global maximum (max_value).
At each point in the iteration, I check whether the current element of the array is less than the current global minimum till that point in the array. If it is such, I update the global minimum to that current element. Similarly, for finding the global maximum, I check whether the current element of the array is greater than the current global maximum till that point in the array. If it is such, I update the global maximum to that current element.
At the end of the loop, I return the global minimum and the global maximum of the array whose absolute difference is going to be the maximum.
Pseudocode:

Algorithm 2.1: Elements x, y ∈ A such that $\mid x - y \mid$ is maximum.

```
$arr\_len \leftarrow length(A)$;
$min\_value \leftarrow \infty$;
$max\_value \leftarrow -\infty$;
for $idx \leftarrow 0$ to $arr\_len - 1$ do
    if $A[idx] < min\_value$ then
        $min\_value = A[idx]$;
    if $A[idx] > max\_value$ then
        $max\_value = A[idx]$;
end for
return $(min\_value, max\_value)$;
```

Correctness of the Algorithm:
Claim: The maximum absolute difference in an array can be found by taking the absolute difference of the global minimum and global maximum of the array.
Base case: If there are only 2 elements in the array, then the absolute difference of those 2 elements will be the maximum absolute difference for the whole array.
Hypothesis: When there are k elements in the array, let's say that when we take the absolute difference of the global minimum value and global maximum value, that will give the maximum absolute difference of two elements in the array. Also, let's denote the global minimum element of the array as 'min_element' and global maximum element as 'max_element'.
Step: Now, let's say an element was added to the array making the size of the array as 'k + 1', Now, we need to analyze 3 cases here:

1. The new element added to the array is the global minimum of the array. So, if the global minimum has changed now, the absolute difference with the max_element and the new global minimum added will be greater than the previous absolute difference.

2. The new element added to the array is the global maximum of the array. So, if the global maximum has changed now, the absolute difference with the min_element and the new global maximum added will be greater than the previous absolute difference.

3. The new element added to the array is neither the global minimum nor the global maximum of the array. If I take the difference of this new number with either the min_element or the max_element, the absolute difference will always be less than the absolute difference of the global minimum and maximum value of the array.

Thus, for all the cases above, we can see that the absolute difference of the global minimum and global maximum element of the array will always give us the maximum absolute difference possible.
Hence, we have proved the correctness of the algorithm.

Time Complexity Analysis: I will analyze the time complexity of the algorithm by noting down the time taken to execute each of the lines separately. Post which, I will combine the time complexities for each of the lines to compute the total time complexity of the complete algorithm.
Let the time taken by the algorithm on an array of input size n be equal to $T(n)$.
Lines 1-3 take constant $\mathcal{O}(1)$ time (initialization steps).
Line 4 runs arr_len times, where arr_len is the size of the input array.
All the operations inside the for loop (lines 5-8) take O(1) time as these are just comparison and assignment operations.
Line 10 (return statement) takes constant time as well.
Finally, the total time taken by the algorithm can be determined by adding up the individual times of all the operations discussed above (which is at most n with some constant operations):
T(n) = $\mathcal{O}(n) + \mathcal{O}(1)$
So, the worst case time complexity of this algorithm is $\mathcal{O}(n)$.

(b)

Problem Description:
Input: A sorted array A of n integers.
Output: Elements x, y $\in$ A such that $\mid x - y \mid$ is maximum.

Solution:
Description of the Algorithm: Assumption: There are at least two elements in the input array.
For this algorithm, to maximize the absolute difference of the elements of the array, we will need to find the global minimum and the global maximum of the array. As the array has already been sorted, we can directly index or get these elements from the array.
For instance, if the array has been sorted in non-decreasing order, the first element of the array will the global minimum and the last element of the array will be the global maximum. As the absolute difference of these elements in the maximum for the array, we can directly get these elements from the array.
Conversely, if the array had been sorted in a non-increasing order, then also these two elements at the start and end of the array will hold but the first element of the array will be the global maximum while last element of the array will be the global minimum. For any of these cases, the absolute difference of these two elements will be the maximum absolute difference of the elements in the array.

Pseudocode:

Algorithm 2.2: Elements x, y $\in$ A such that $\mid x - y \mid$ is maximum.

```
1 $arr\_len \leftarrow length(A)$;
2 $min\_value \leftarrow A[0]$;
3 $max\_value \leftarrow A[arr\_len - 1]$;
4 **return** $(min\_value, max\_value)$;
```

Correctness of the Algorithm:
Claim: The maximum absolute difference in an array can be found by taking the absolute difference of the global minimum and global maximum of the array.
Base case: If there are only 2 elements in the array, then the absolute difference of those 2 elements will be the maximum absolute difference for the whole array.
Hypothesis: When there are k elements in the array, let's say that when we take the absolute difference of the global minimum value and global maximum value, that will give the maximum absolute difference of two elements in the array. Also, let's denote the global minimum element of the array as 'min_element' and global maximum element as 'max_element'. As the array has been sorted, the global minimum will be present at the first index of the array and the global maximum will be present at the last index of the array.
Step: Now, let's say an element was added to the array making the size of the array as 'k + 1', Now, we need to analyze 3 cases here:

1. The new element added to the array is the global minimum of the array (added at the start of the array since the input array should be sorted). So, if the global minimum has changed now, the absolute difference with the max_element and the new global minimum added will be greater than the previous absolute difference.

2. The new element added to the array is the global maximum of the array (added at the end of the array since the input array should be sorted). So, if the global maximum has changed now, the absolute difference with the min_element and the new global maximum added will be greater than the previous absolute difference.

3. The new element added to the array is neither the global minimum nor the global maximum of the array (added somewhere in the middle of the array since the input array should be sorted). If I take the difference of this new number with either the min_element or the max_element, the absolute difference will always be less than the absolute difference of the global minimum and maximum value of the array.

Thus, for all the cases above, we can see that the absolute difference of the global minimum and global maximum element of the array will always give us the maximum absolute difference possible. As the array has been sorted, we can get these global minimum and maximum values by directly indexing the first and the last element of the array respectively.
Hence, we have proved the correctness of the algorithm.

Time Complexity Analysis: I will analyze the time complexity of the algorithm by noting down the time taken to execute each of the lines separately. Post which, I will combine the time complexities for each of the lines to compute the total time complexity of the complete algorithm.
Let the time taken by the algorithm on an array of input size n be equal to $T(n)$.
All the lines 1-4 are constant time operations since they are just getting the length of the array and assignment or indexing operations.
So, the total time complexity of this algorithm is $\mathcal{O}(1)$.

(c)

Problem Description:
Input: An unsorted array A of n integers.
Output: Elements x, y $\in$ A such that $\mid x - y \mid$ is minimum.

Solution:
Description of the Algorithm: Assumption: There are at least two elements in the input array.
For this algorithm, we need to find the minimum absolute difference of any of the two elements in the array. As the array is unsorted initially, to reduce the complexity of the overall algorithm, we would need to sort the array. Here, I am using merge sort to do so which is an out-of-place algorithm which would use some extra memory.
After the array has been sorted, I iterate over the array starting at the second element of the array taking the absolute difference between the current element and the previous element of the array. As the array has been sorted, the minimum difference can only occur between two elements that are side by side in the array. Following this fact, I take the absolute difference between the current and the previous element of the array.
For finding the minimum absolute difference, I compare that to the minimum difference found till that point in the array. If it's less than the minimum difference till that point in the array, I update the minimum difference with the current minimum absolute difference. Also, I maintain the minimum and the maximum values at the same time which we need to return at the end of loop.

Pseudocode:

Algorithm 2.3: Elements x, y ∈ A such that $\mid x - y \mid$ is minimum.

```
$arr\_len \leftarrow length(A)$;
$sorted\_A = merge\_sort(A)$;
$min\_diff \leftarrow \infty$;
$min\_value \leftarrow \infty$;
$max\_value \leftarrow -\infty$;
for $idx \leftarrow 1$ to $arr\_len - 1$ do
    $curr\_diff = abs(sorted\_A[idx] - sorted\_A[idx - 1])$;
    if $curr\_diff < min\_diff$ then
        $min\_diff = curr\_diff$;
        $min\_value = sorted\_A[idx - 1]$;
        $max\_value = sorted\_A[idx]$;
end for
return $(min\_value, max\_value)$;
```

Correctness of the Algorithm:
For this algorithm, to efficiently solve the problem at hand, we will need to sort the array first. This step is needed so that we can efficiently find the minimum absolute difference of the elements, taking two adjacent elements at a time, in the array.
Let's take a sorted array with 3 elements with values as a, b, and c. As the array is sorted, the condition $a \leq b \leq c$ holds. Now, if we take the absolute difference of b and a and take the absolute difference of c and a along with the assumption that all these are distinct for now. Then, the absolute difference of b and a will always be less than the absolute difference of c and a as c is farther from a than b. Even so, if these are not distinct, then it doesn't matter but the same algorithm will work for that too. This fact tells us that we only need to take the differences of the adjacent pair of elements in the array to find the minimum absolute difference and the corresponding elements.
Now, if we take a big array with more elements, this same fact would work there too. Basically, we will go over each pair of elements finding the absolute difference of the adjacent elements. The pair that gives us the minimum absolute difference is the answer.
Hence, we have proved the correctness of the algorithm.

Time Complexity Analysis: I will analyze the time complexity of the algorithm by noting down the time taken to execute each of the lines separately. Post which, I will combine the time complexities for each of the lines to compute the total time complexity of the complete algorithm.
Let the time taken by the algorithm on an array of input size n be equal to $T(n)$.

In line 2 I am using a sorting technique called Merge Sort which runs in $\mathcal{O}(n \log n)$ time, which was covered in the class.
Lines 1 and 3-5 are simply initialization steps (line 1 is getting the length and I am taking

that to be constant time) which runs in constant time $\mathcal{O}(1)$.
Furthermore, the for loop in line 6 runs $arr\_len = n$ times. Thus, it takes $\mathcal{O}(n)$ time.
Also, Lines 7-11 are just constant time operations since they are just initialization, comparison and assignment operations. These operations are run $arr\_len = n$ times.
Line 13 (return statement) is also a constant time $\mathcal{O}(1)$ operation.
Finally, the total time taken by the algorithm can be determined by adding up the individual times of all the operations discussed above:
$T(n) = (\mathcal{O}(n \log n) + \mathcal{O}(n))$
As asymptotically, $n < n \log n$ and we always take the higher order term when finding the asymptotic time complexities, we get the following:
$T(n) = \mathcal{O}(n \log n)$

(d)

Problem Description:
Input: An sorted array A of n integers.
Output: Elements x, y $\in$ A such that $\mid x - y \mid$ is minimum.

Solution:
Description of the Algorithm: Assumption: There are at least two elements in the input array.
For this algorithm, we need to find the minimum absolute difference of any of the two elements in the array. As the array has been already sorted, I iterate over the array starting at the second element of the array taking the absolute difference between the current element and the previous element of the array. The minimum difference can only occur between two elements that are side by side in the array. Following this fact, I take the absolute difference between the current and the previous element of the array.
For finding the minimum absolute difference, I compare that to the minimum difference found till that point in the array. If it's less than the minimum difference till that point in the array, I update the minimum difference with the current minimum absolute difference. Also, I maintain the minimum and the maximum values at the same time which we need to return at the end of loop.

Pseudocode:

Algorithm 2.4: Elements x, y $\in$ A such that $| x - y |$ is minimum.

```
$arr\_len \leftarrow length(A)$;
$min\_diff \leftarrow \infty$;
$min\_value \leftarrow \infty$;
$max\_value \leftarrow -\infty$;
for $idx \leftarrow 1$ to $arr\_len - 1$ do
    $curr\_diff = abs(A[idx] - A[idx - 1])$;
    if $curr\_diff < min\_diff$ then
        $min\_diff = curr\_diff$;
        $min\_value = A[idx - 1]$;
        $max\_value = A[idx]$;
end for
return $(min\_value, max\_value)$;
```

Correctness of the Algorithm:
To prove the correctness of this algorithm, let's take a sorted array with 3 elements with values as a, b, and c. As the array is sorted, the condition $a \leq b \leq c$ holds. Now, if we take the absolute difference of b and a and take the absolute difference of c and a along with the assumption that all these are distinct for now. Then, the absolute difference of b and a will always be less than the absolute difference of c and a as c is farther from a than b. Even so, if these are not distinct, then it doesn't matter but the same algorithm will work for that too. This fact tells us that we only need to take the differences of the adjacent pair of elements in the array to find the minimum absolute difference and the corresponding elements.
Now, if we take a big array with more elements, this same fact would work there too. Basically, we will go over each pair of elements finding the absolute difference of the adjacent elements. The pair that gives us the minimum absolute difference is the answer.
Hence, we have proved the correctness of the algorithm.

Time Complexity Analysis: I will analyze the time complexity of the algorithm by noting down the time taken to execute each of the lines separately. Post which, I will combine the time complexities for each of the lines to compute the total time complexity of the complete algorithm.
Let the time taken by the algorithm on an array of input size n be equal to $T(n)$.
Lines 1-4 are simply initialization steps (line 1 is getting the length and I am taking that to be constant time) which runs in constant time $\mathcal{O}(1)$.
Furthermore, the for loop in line 5 runs $arr\_len = n$ times. Thus, it takes $\mathcal{O}(n)$ time.
Also, Lines 6-10 are just constant time operations since they are just initialization, comparison and assignment operations. These operations are run $arr\_len = n$ times.
Line 12 (return statement) is also a constant time $\mathcal{O}(1)$ operation.
Finally, the total time taken by the algorithm can be determined by adding up the individual times of all the operations discussed above (which is at most linear time as some of the lines

are constant time and others are run at most n times:
$T(n) = \mathcal{O}(n)$

Problem 3

Problem Description:
Input: An array A of n entries.
Output: The majority element (the same element present more than half the size of the array), if one exists. Otherwise, return NULL.

Solution:
Description of the Algorithm: For this algorithm, we use the technique of ”Divide and Conquer”. We divide the array in 2 parts (from the middle of the array) using recursion.
As we are dividing the array in two parts recursively at first, we would need to handle some cases after the call has been made to get the majority element from the the sub arrays at that point in the tree.
The conditions to check for will be as follows:

1. Check whether the left and right majority elements we get are the same. If that's the case, then we can return any of the majority elements back as it doesn't matter. Here, I have chosen to return the left majority element,i.e., the majority element from the left sub-array.

2. Check whether the left majority element (majority element from the left sub-array at that point in time) is NULL. If the left majority element was NULL, that would mean that no element from the left sub array is the majority element. Thus, we will try to find the majority element from the right sub-array, namely right majority element. For this case, we check whether the right majority element we have got can be the majority element for the whole array. This is achieved by calling a sub-routine where we count the occurrence of the right majority element found in the whole array. If it is the majority element, then we return the right majority element.

3. Similarly, check whether the right majority element (majority element from the right sub-array at that point in time) is NULL. If the right majority element was NULL, that would mean that no element from the right sub array is the majority element. Thus, we will try to find the majority element from the left sub-array, namely left majority element. For this case, we check whether the left majority element we have got can be the majority element for the whole array. This is achieved by calling a sub-routine where we count the occurrence of the left majority element found in the whole array. If it is the majority element, then we return the left majority element.

4. If none of the above cases are fulfilled, then we return NULL. As we shouldn't do this for the leaf nodes of the tree that are basically single element of the array, we will have to place another check at the top of the recursive function (above the recursive calls) to check whether it's a leaf node. If the current one would be a leaf node, then we return the element itself. The check for the leaf node is done by checking whether the left and right indices are the same. If the indices are the same, that would mean that we are at the leaf node level of the tree.

As we recursively pass on the majority element while backtracking (bottom-up in the tree formed). If a majority element exists, I would get the majority element from some point in the tree while backtracking and as I reach the top part of the recursion tree formed, I would have a majority element from both parts of the sub-arrays. If I don't get a majority element as I reach the top of the recursion tree while backtracking, I return NULL to suggest that no such majority element exists in the array.

Pseudocode:

Algorithm 3: Finding the majority element, if one exists, else -1.

```
Function findMajorityElement($A$, $left$, $right$):
    if $left == right$ then
        return $A[left]$;
    $mid = left + \frac{right-left}{2}$;
    $left\_majority\_element = findMajorityElement(A, left, mid)$;
    $right\_majority\_element = findMajorityElement(A, mid+1, right)$;
    if $left\_majority\_element == right\_majority\_element$ then
        return $left\_majority\_element$;
    else if $left\_majority\_element == NULL$ then
        if $isMajorityElement(A, right\_majority\_element)$ then
            return $right\_majority\_element$;
    else if $right\_majority\_element == NULL$ then
        if $isMajorityElement(A, left\_majority\_element)$ then
            return $left\_majority\_element$;
    else
        return $NULL$;
Function isMajorityElement($A$, $curr\_element$):
    $count \leftarrow 0$;
    $arr\_len \leftarrow length(A)$;
    for $idx \leftarrow 0$ to $arr\_len - 1$ do
        if $A[idx] == curr\_element$ then
            $count = count + 1$;
    end for
    return $count > \frac{arr\_len}{2}$;
```

Correctness of the Algorithm:
Claim: If an array has a majority element, then one of it's sub-arrays must have it as it's majority element.
We will prove the correctness of the algorithm by "contradiction". We will contradict the claim above effectively negating it and saying that if a majority element exists, it will not be returned by either of the two sub-arrays.
We are dividing the array into two equal halves at each point of the iteration. So, if the left sub-array won't return the majority element that would mean that the majority element of the whole array can't be present for more than $(n/4)$ times in the left sub-array. This comes from the fact that the majority element of the left sub-array should be present for more than half the size of the array, so, if it's not being returned, that would mean that it's present for less than equal to $((n/2)/2$ times in the left sub-array. If that's the case, that would leave the number of times the majority element occurrences in the right sub-array as follows (taking the assumption into consideration that majority element exists, thus, the occurrences should

more than half from the right sub-array):
size of the right sub-array $+1 - (n/4) = (n/2) + 1 - (n/4) = (n/4) + 1$
The computation above shows us that the right sub-array will have to return the majority element as it should be present for more than half the elements of that sub-array. Thus, the contradiction fails. This fact tells us that we would surely get the majority element from at least one of the sub-arrays and when the whole backtracking finishes from bottom-up in the recursion tree, we will have the majority element as the answer.
Hence, we have proved the correctness of the algorithm.

Time Complexity Analysis: I will analyze the time complexity of the algorithm by noting down the time taken to execute each of the lines separately. Post which, I will combine the time complexities for each of the lines to compute the total time complexity of the complete algorithm.
Let the time taken by the algorithm on an array of input size n be equal to $T(n)$.

Lines 2-4 are just constant time $\mathcal{O}(1)$ operations (comparisons, return statement and assignment).
Lines 5-6 takes $T(\frac{n}{2})$ time each as it is a recursive call and at each call, the array is divided into two equal halves and in the recursion tree, we will have two children from each call. Thus, the total time taken by both the lines is $2T(\frac{n}{2})$.
Lines 7-16 (except 10 and 13) take constant time $\mathcal{O}(1)$ as these are just comparisons and return operations.
For lines 10 and 13, I will have to analyze $isMajorityElement(A, curr\_element)$: Now, lines 18-19 takes constant $\mathcal{O}(1)$ time. And, line 20 runs n times thus taking $\mathcal{O}(n)$ time in traversing the full array. Also, lines 21-24 are just constant time $\mathcal{O}(1)$ operations. Thus, $isMajorityElement(A, curr\_element)$ takes $\mathcal{O}(n)$ time since it runs cn operations.
Finally, the total time taken by the algorithm can be determined by adding up the individual times of all the operations discussed above:
$T(n) = 2T(\frac{n}{2}) + cn$
So, using master theorem, we know that:

$$\text{T(n)} = \begin{cases} \mathcal{O}(n^{\log_b a}) & \text{if } a > b^k \\ \mathcal{O}(n^k \log n) & \text{if } a = b^k \\ \mathcal{O}(n^k) & \text{if } a < b^k \end{cases}$$

In our case, a = 2, b = 2, k = 1. Thus, we use the condition $a = b^k$ whereby we will use the second case of the master theorem.
Thus, we have $T(n) = \mathcal{O}(n \log n)$
Finally, the worst case time complexity of this algorithm is $\mathcal{O}(n \log n)$.
Now, for analyzing the lower bound of the algorithm, the best case input will be when all the elements in the array are equal (which will be majority). Following this, the first condition, i.e., $left\_majority\_element == right\_majority\_element$ will be satisfied at every call. However, we still have to divide the array into two equal halves and perform all the

checks including the check if an element returned as majority is the majority of the whole array. Because we haven't been able to cut down on the number of operations needed to run this algorithm for the best case input, the best case time complexity also stands to be $\Omega(n \log n)$. So, we can conclusively say that the average case time complexity of this solution is $\Theta(n \log n)$.

Problem 4

Problem Description:
Input: A undirected and complete binary tree T with $2^d - 1$ nodes and $d > 1$
Output: The Local Minimum of T.

Solution:
Description of the Algorithm: For this algorithm, As we need to find a local minimum of T which would be possible if there is a node v with a real number x_v, such that all x_v are distinct, that has value that is less than the values of all the nodes connected to it through an edge. Thus, we need to compare each node with three other nodes - its parent, left child and right child. Also, it's given that each probe takes constant time.
So, for doing the same, I have written a modified form of the tree traversal (top-down traversal). Firstly, I get the current value (root of the tree at the current point in the traversal), then I need to perform some checks and access the values of different nodes before traversing the tree.
So, I check whether both the left and right children are NULL (not present). If it is such, I return the current node value. Otherwise before accessing the value of the children, I check for both the children being NULL separately. Also, if the current value of the root node is less than both the children's values, then I return the current value itself as it will be the local minimum of T. This is so because when we are going down recursively checking for this, then at the current node in the tree, I would have already checked for it's parent node. Thus, we don't need to check for it again.
Also, if the left child is smaller than the right child, we traverse the left sub-tree by calling the function again with the left child node. Otherwise, I traverse the right sub-tree by calling the function again with the right child. This is done so because there will always be a local minimum in a sub-tree which contains distinct numbers one after the other. Based on this fact, we can traverse the tree however, we want as we only need to return any one of the local minimums of the tree T.

Pseudocode:

Algorithm 4: Find the Local Minimum in an undirected complete binary tree T.

```
Function findLocalMinima($T$):
    $curr\_val \leftarrow value[T]$;
    if $T.left == NULL$ and $T.right == NULL$ then
        return $curr\_val$;
    if $T.left \neq NULL$ then
        $left\_val \leftarrow value[T.left]$;
    if $T.right \neq NULL$ then
        $right\_val \leftarrow value[T.right]$;
    if $curr\_val < left\_val$ and $curr\_val < right\_val$ then
        return $curr\_val$;
    else if $left\_val < right\_val$ then
        return $findLocalMinima(T.left)$;
    else
        return $findLocalMinima(T.right)$;
```

Correctness of the Algorithm:
Claim: We will always find a local minimum as there is always a minimum number out of some set of numbers.
Base case: The tree provided will always have at least 3 nodes. So, our base case is a tree T with 3 nodes. As it's a complete binary tree, we will have 3 nodes as the root node, left child node, right child node. As one of them will be minimum so, we are bound to find a local minimum out of those 3 nodes.
Hypothesis: Assuming that we haven't found a local minimum till we have reached kth level of the tree, we know that the parent of the current node in the kth level must have a value greater than the current node value. In the step part of the tree traversal, we will try and see if we can find a local minimum at the (k+1)th level. Also, we know that as the current node is not the local minimum, thus, it's greater than at least one of it's children at the (k+1)th level.
Step: When we move to the (k+1)th level, as we already know that the parent would be greater than the current node value, we don't have to check for parent again. Now, we will only check for the children. If we find that the children nodes have values not greater than the current node, then we have successfully found the local minimum. If not, we will move further down the tree. The cases can be shown as follows:

1. If the current value of the node is smaller than both the children, then that would mean that the current node is the local minimum.

2. If the current value of the node is smaller than left child node's value but is greater than the right child node's value. Then we move in the direction of the right node. As

just in case the right node is a leaf node, then that would become our local minimum.

3. If the current value of the node is smaller than right child node's value but is greater than the left child node's value. Then we move in the direction of the left node. As just in case the left node is a leaf node, then that would become our local minimum.

4. If the current node itself is the leaf node, then we know by our hypothesis that parent was greater, thus, the current node would become our local minimum.

One additional thing to note here is that if the (k+1)th level is the level at which the leaf nodes are present and the current node is a leaf node, then, the only connection a leaf node has is the edge with the parent. As our hypothesis states that the parent node's value was greater, then we know that the leaf node will be the local minimum.

Time Complexity Analysis:
I will analyze the time complexity of the algorithm by noting down the time taken to execute each of the lines separately. Post which, I will combine the time complexities for each of the lines to compute the total time complexity of the complete algorithm.
Let the time taken by the algorithm on an array of input size n be equal to $T(n)$.

All the lines 2-14 are just constant time operations except lines 12 and 14. These are just constant time operations as they involve comparison, or assignment, and return statements. Lines 15 and 17 are basically recursive calls wherein one of the sub-trees (through left or right child) is taken into consideration for the next call. Because for each iteration, at most one of these recursive calls is made, the total time taken by both these lines is $T(\frac{n}{2})$.
Finally, the total time taken by the algorithm can be determined by adding up the individual times of all the operations discussed above:
$T(n) = T(\frac{n}{2}) + \mathcal{O}(1)$
So, using master theorem, we know that:

$$\text{T(n)} = \begin{cases} \mathcal{O}(n^{\log_b a}) & \text{if } a > b^k \\ \mathcal{O}(n^k \log n) & \text{if } a = b^k \\ \mathcal{O}(n^k) & \text{if } a < b^k \end{cases}$$

In our case, a = 1, b = 2, k = 0. So, we will use $a = b^k$, thus, we will use the second case of the master theorem. Thus, we have $T(n) = \mathcal{O}(\log n)$

Thus, the worst case time complexity of this algorithm is $\mathcal{O}(\log n)$.

Problem 5

Problem Description:
Input: A sorted array T holding the trace in the form of triplets with each triplet having

the node labels as integers that communicate with each other along with the time at which they communicated. Also, a virus query is inputted with 2 node labels and the time range (virus introduced at time t_1, virus spreads to the second node till time t_2) as two numbers. Lastly, the number of computers is also provided.
Output: True or False stating whether the virus can spread from the first computer in the query to the second computer till time t_2, if the virus is introduced at time t_1 at the first computer.

Solution:
Description of the Algorithm: For this algorithm:
Firstly, I am going over the traces provided for adding the nodes and the edges. While doing so, I add the tuple holding the computer node and the time of communication to the graph. For doing the same, I check if the time of communication of the computers lies in the virus query range provided. If it does, then I add a directed edge between the two computer nodes. As the communication between computers is bi-directional, I add the edges in both the directions. Although, I firstly check whether the communication had already been visited,i.e., the edge was already created. If it wasn't done, I add the same.
After the full graph is prepared having all the communications between the computers during the virus query range, I check whether the computer nodes provided are present in the graph using Breadth-First-Search (BFS).
Assumption: I am taking the assumption that BFS will return whether both the nodes are present or not. If any of them aren't present, False is returned that signifies that the virus cannot spread between the mentioned computer nodes.

Pseudocode:

Algorithm 5: Find if the virus can spread between the two computers in a particular time range specified.

```
Function canVirusSpread($T$, $n$, $C_1$, $C_2$, $t_1$, $t_2$):
    $graph = \{\}$;
    $times \leftarrow Array(size : n)$;
    for $trace \in T$ do
        $C_i, C_j, curr\_time \leftarrow trace$;
        if $< C_i, curr\_time >$ not in graph then
            $graph \leftarrow graph \cup < C_i, curr\_time >$;
        end if
        if $< C_j, curr\_time >$ not in graph then
            $graph \leftarrow graph \cup < C_j, curr\_time >$;
        end if
        Add an 2 directed edges (signifies bi-directional) between
        $< C_i, curr\_time >$ and $< C_j, curr\_time >$;
        if $times[C_i] == NULL$ then
            $times[C_i] = curr\_time$;
        end if
        if $times[C_j] == NULL$ then
            $times[C_j] = curr\_time$;
        end if
        if $times[C_i] \geq t_1$ and $times[C_i] \leq t_2$ then
            Add a directed edges between $< C_i, times[C_i] >$ and
            $< C_j, curr\_time >$;
        end if
        if $times[C_j] \geq t_1$ and $times[C_j] \leq t_2$ then
            Add a directed edges between $< C_j, times[C_j] >$ and
            $< C_i, curr\_time >$;
        end if
    end for
    return Check that $C_1$ and $C_2$ are present in graph using BFS (Breadth
      First Search);
```

Correctness of the Algorithm:
Let the virus query be (C_i, C_j, x, y),
where C_i and C_j are the two computers and x and y are the timestamps.

We need to show that when there exists a directed path between C_i and C_j, these computers can infect each other. Since we performed BFS on our graph, we have a directed path from (C_i, x) and (C_j, t) such that $t \leq y$.

The following cases arise:

1. Firstly, if C_j is infected at time $t \leq y$, a sequence of virus transmissions of this form exists: $(C_i, C_{l_1}, t_1), (C_{l_1}, C_{l_2}, t_2), ..., (C_{l_k}, C_j, t_k)$

   In Graph G, we have:
   $(C_i, t_1) \rightarrow (C_{l_1}, t_1) \rightarrow (C_{l_1}, t_2) \rightarrow (C_{l_2}, t_2)... \rightarrow (C_{l_j}, t_k)$
   Clearly, these tuples form a path devoid of any breaks and so we know that BFS will find this path.

2. Secondly, if a path from (C_i, x) to (C_j, t_k) is discovered such that $t_k \leq y$,
   $(C_i, t_1) \rightarrow (C_{l_1}, t_1) \rightarrow (C_{l_1}, t_2) \rightarrow (C_{l_2}, t_2)... \rightarrow (C_{l_j}, t_k)$

   Then, we know that every machine in along this sequence will be infected. Hence, the virus can spread from C_i to C_j within $x \leq t \leq y$.
   Hence, we have proved the correctness of the algorithm.

Time Complexity Analysis: I am analyzing the time complexity below:
Line 2 is constant time operation since it is basic declaration of the graph.
Line 3 takes $\mathcal{O}(n)$ time since we are declaring an array of size n.
Line 4 is called for each trace T which defines the upper bound for the number of edges in our Graph G. Thus, this takes $\mathcal{O}(m)$ time.
Lines 5-25 are constant time operations since they simply do comparisons and take unions, which is just adding a new vertex to the graph G.
Line 26 calls BFS on our graph which takes $\mathcal{O}(n + m)$ time.
Thus, in totality, our algorithm runs for $\mathcal{O}(n + m)$.

Problem 6

Problem Description:
Input: A directed graph G in the form of an adjacency list.
Output: The graph node index from which all other vertices in the graph are reachable.
Solution:
Description of the Algorithm: The main algorithm for this problem is written as the 'findMotherVertex' function. In general, mother vertex is a vertex in the graph from which all the other nodes in the graph are reachable. For finding the same, firstly, I take an array 'is_visited' that is initialized with all False and signifies whether a node has been visited or not. Each index of that array signifies a node in the graph.
Then, I loop over all the nodes through indices and check if the current node has already been visited by checking the same through 'is_visited' array. If the current node wasn't visited yet, then I run the Depth First Search through the graph starting from that node. During

the DFS, I mark all the nodes visited as visited by changing the corresponding index in the 'is_visited' as True. Every time, the DFS utility function is called, I update the last vertex from which we visited all other nodes making it the mother vertex. Thus, this full run gives us a potential mother vertex.
After all the nodes have been visited, i.e., at the end of the main loop, I re-initialize the is_visited array with False and again run the DFS starting at the mother vertex that we had got. After that DFS is completed, we check if there is any node or index in the is_visited array that holds a False which would signify checking for a node that was still not visited. If there is any such node that was not visited, then, we return -1. Otherwise, we return the mother vertex using which we had run the second DFS.

Pseudocode:

Algorithm 6: Find the graph node index from which all other nodes in the graph are reachable, if one exits. Otherwise, return -1.

```
Function findMotherVertex($G$, $n$):
    $is\_visited \leftarrow Array(size: n, init: False)$;
    $mother\_vertex \leftarrow -1$;
    for $idx \leftarrow 0$ to $n-1$ do
        if not $is\_visited[idx]$ then
            $DfsUtil(G, idx, is\_visited)$;
            $mother\_vertex = idx$;
    end for
    $is\_visited \leftarrow Array(size: n, init: False)$;
    $DfsUtil(G, mother\_vertex, is\_visited)$;
    for $idx \leftarrow 0$ to $n-1$ do
        if not $is\_visited[idx]$ then
            return $-1$;
    end for
    return $mother\_vertex$;
Function DfsUtil($G$, $node$, $is\_visited$):
    $is\_visited[node] \leftarrow True$;
    for $next\_node$ in $G[node]$ do
        if not $is\_visited[next\_node]$ then
            $DfsUtil(G, next\_node, is\_visited)$;
    end for
```

Correctness of the Algorithm:
As we are just using Depth First Search as our basis for the algorithm which we basically run twice for finding the mother vertex, the correctness of the algorithm is not needed here.

We assume that DFS is correct and thus, our algorithm is also correct. Since, we only keep track of the mother vertex (last vertex from which we visited other nodes in the graph) in addition to the DFS, the algorithm is correct.

Time Complexity Analysis: The time complexity of this algorithm is $\mathcal{O}(n + m)$ where n is the number of nodes/vertices in the graph and m is the number of edges in the graph. As we are simply applying DFS two times and the time complexity of a DFS algorithm is linear (which was covered in the class) which gives us $\mathcal{O}(2(n+m))$ which translates to $\mathcal{O}(n+m)$.