**Group Members**:

Pravir Kadian - 19BCE10006

Abhishek Srivastava - 19BCE10071

**DAA Group Activity-1**

**Question-1  
  
Solutions**

1. Searching linearly: Increment the loop by 1
2. Divide and Conquer: Tournament Method
3. Comparison in pairs: Increment the loop by 2

**1. Searching linearly: Increment the loop by 1**

We initialize both minimum and maximum element to the first element and then traverse the array, comparing each element and update minimum and maximum whenever necessary.

***Pseudo-Code***

**int**[] getMinMax(**int** A[], **int** n)

{

**int** max = A[0]

**int** min = A[0]

**for** ( i = 1 to n-1 )

{

**if** ( A[i] > max )

max = A[i]

**else** **if** ( A[i] < min )

min = A[i]

}

*// By convention, let ans[0] = maximum and ans[1] = minimum*

**int** ans[2] = {max, min}

**return** ans

}

***Complexity Analysis***

At every step of the loop, we are doing 2 comparisons in the worst case. Total no. of comparisons (in worst case) = 2\*(n-1) = 2n - 2

Time complexity = O(n), Space complexity = O(1)

In the best case, a total of n-1 comparisons have been made. (**How?**)

***Critical ideas to think!***

* We have initialized maximum and minimum with the first element of the array - why?
* What would be the best case and worst case input?
* How can we decrease the number of comparisons made here?

**2. Divide and Conquer : Tournament Method**

Another way to do this could be by following the divide and conquer strategy. Just like the merge sort, we could divide the array into two equal parts and recursively find the maximum and minimum of those parts. After this, compare the maximum and minimum of those parts to get the maximum and minimum of the whole array.

***Solution Steps***

1. Write a recursive function accepting the array and its start and end index as parameters
2. The base cases will be

* If array size is 1, return the element as both max and min
* If array size is 2, compare the two elements and return maximum and minimum

3. The recursive part is

* Recursively calculate and store the maximum and minimum for left and right parts
* Determine the maximum and minimum among these by 2 comparisons

4. Return max and min.

***Pseudo Code***

**int**[] findMinMax(**int** A[], **int** start, **int** end)

{

**int** max;

**int** min;

**if** ( start == end )

{

max = A[start]

min = A[start]

}

**else** **if** ( start + 1 == end )

{

**if** ( A[start] < A[end] )

{

max = A[end]

min = A[start]

}

**else**

{

max = A[start]

min = A[end]

}

}

**else**

{

**int** mid = start + (end - start)/2

**int** left[] = findMinMax(A, start, mid)

**int** right[] = findMinMax(A, mid+1, end)

**if** ( left[0] > right[0] )

max = left[0]

**else**

max = right[0]

**if** ( left[1] < right[1] )

min = left[1]

**else**

min = right[1]

}

*// By convention, we assume ans[0] as max and ans[1] as min*

**int** ans[2] = {max, min}

**return** ans

}

***Complexity Analysis***

For counting the number of comparisons, since this is a recursive function, let us define the recurrence relation :

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

T(2) = 1

T(1) = 0

We can solve **this** recurrence relation by master method/recursion tree method.

**if** n is a power of 2

T(n) = 3n/2 - 2

Time complexity = O(n) and space complexity = O(logn) (For recursion call stack)

If n is a power of 2, the algorithm needs exactly 3n/2–2 comparisons to find min and max. If it's not a power of 2, it will take a few more(not significant).

***Critical ideas to think!***

* How do we analyze the recursion by the master's theorem and recursion tree method?
* How is the space complexity derived to be O(logn)?
* Why there are 2 base cases? What if we remove the base case with array size 2?
* Why prefer mid = start + (end - start)/2 over (start + end)/2 when calculating middle of the array ?
* Can the number of comparisons be decreased further?

**3. Comparison in Pairs : Increment the loop by 2**

In this approach, we pick array elements in pairs and update the min and max. If the array size is odd, we initialize the first element as both min and max, and if it's even, we compare the first two elements and initialize min and max accordingly.

***Solution Steps***

1. Create max and min variables.
2. Check for the size of the array

* If odd, initialize min and max to the first element
* If even, compare the elements and set min to the smaller value and max to the bigger value

3. Traverse the array in pairs

4. For each pair, compare the two elements and then

* Compare the bigger element with max, update max if required.
* Compare the smaller element with min, update min if required.

5. Return max and min.

***Pseudo Code***

**int**[] findMinMax(**int** A[], **int** n)

{

**int** max, min

**int** i

**if** ( n is odd )

{

max = A[0]

min = A[0]

i = 1

}

**else**

{

**if** ( A[0] < A[1] )

{

max = A[1]

min = A[0]

}

**else**

{

max = A[0]

min = A[1]

}

i = 2

}

**while** ( i < n )

{

**if** ( A[i] < A[i+1] )

{

**if** ( A[i] < min )

min = A[i]

**if** ( A[i+1] > max )

max = A[i+1]

}

**else**

{

**if** ( A[i] > max )

max = A[i]

**if** ( A[i+1] < min )

min = A[i+1]

}

i = i + 2

}

*// By convention, we assume ans[0] as max and ans[1] as min*

**int** ans[2] = {max, min}

**return** ans

}

***Complexity Analysis***

Time Complexity is O(n) and Space Complexity is O(1).

For each pair, there are a total of three comparisons, first among the elements of the pair and the other two with min and max.

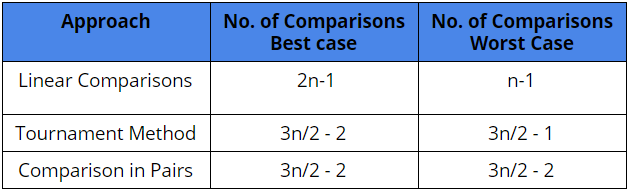
Total number of comparisons:-

* If n is odd, 3 \* (n-1) / 2
* If n is even, 1 + 3\*(n-2)/2 = 3n/2-2

***Critical ideas to think!***

* Why min and max are initialized differently for even and odd sized arrays?
* Why incrementing the loop by 2 help to reduce the total number of comparsion ?
* Is there any other way to solve this problem? Think.
* In which case, the number of comparisons by method 2 and 3 is equal?

**Comparison of different solutions**



**Question-2**

Binary Search

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do [linear search](http://quiz.geeksforgeeks.org/linear-search/)**.**The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.

**Binary Search:** Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

**Recursive**implementation of Binary Search

|  |
| --- |
| // C++ program to implement recursive Binary Search  #include <bits/stdc++.h>  using namespace std;    // A recursive binary search function. It returns  // location of x in given array arr[l..r] is present,  // otherwise -1  int binarySearch(int arr[], int l, int r, int x)  {      if (r >= l) {          int mid = l + (r - l) / 2;            // If the element is present at the middle          // itself          if (arr[mid] == x)              return mid;            // If element is smaller than mid, then          // it can only be present in left subarray          if (arr[mid] > x)              return binarySearch(arr, l, mid - 1, x);            // Else the element can only be present          // in right subarray          return binarySearch(arr, mid + 1, r, x);      }        // We reach here when element is not      // present in array      return -1;  }    int main(void)  {      int arr[] = { 2, 3, 4, 10, 40 };      int x = 10;      int n = sizeof(arr) / sizeof(arr[0]);      int result = binarySearch(arr, 0, n - 1, x);      (result == -1) ? cout << "Element is not present in array"                     : cout << "Element is present at index " << result;      return 0;  } |

**Output :**

Element is present at index 3