ACADEMY OF TECHNOLOGY

Lab Assignment 4

Paper name: Design and Analysis of Algorithms Lab

Code: PCC-CS494 | **Semester: 4^{th}**

Discipline: CSE | **Time: 2 Hours**

Date: March 28, 2023

1. Write a program in C or C++ to find a given $\boldsymbol{key}$ in array using ternary search algorithm.

Algorithm 1: SEARCH$(x[],\ l,\ r,\ key)$

// $x[1:n]$ is an array of n elements such that $x[1] \leq x[2] \leq .... \leq x[n]$.
// l and r indicate the left and right index respectively of array.
// key is the element which is to be searched within the array.
if $r \geq l$ **then**
  $mid_1 := l + \lfloor\frac{(r-l)}{3}\rfloor$;
  $mid_2 := r - \lfloor\frac{(r-l)}{3}\rfloor$;
  if $x[mid_1] = key$ **then return** mid_1;
  if $x[mid_2] = key$ **then return** mid_2;
  if $key < x[mid_1]$ **then return** SEARCH $(l, mid_1 - 1, key, x)$;
  else if$(key > x[mid_2])$ **then return** SEARCH $(mid_2 + 1, r, key, x)$;
  else return SEARCH $(mid_1 + 1, mid_2 - 1, key, x)$;
  return-1;
end

Algorithm 1 is a searching technique that is used to determine the position of a specific value in an array. Using an example verify that the Algorithm 1 correctly functioning in order to find whether an element exist in the list. What will be the recurrence relation for this algorithm. Solve the recurrence to find the time complexity of this algorithm? Is it better than binary search?

Which of the above two does less comparisons in worst case?

From the first look, it seems the ternary search does less number of comparisons as it makes $log_3 n$ recursive calls, but binary search makes $log_2 n$ recursive calls. Let us take a closer look. The following is recursive formula for counting comparisons in worst case of Binary Search.

$T(n) = T(n/2) + 2,\ T(1) = 1$

The following is recursive formula for counting comparisons in worst case of Ternary Search.

$T(n) = T(n/3) + 4,\ T(1) = 1$

In binary search, there are $2log_2 n + 1$ comparisons in worst case. In ternary search, there are $4log_3 n + 1$ comparisons in worst case.

Time Complexity for Binary search $= 2clog_2 n + O(1)$

Time Complexity for Ternary search $= 4clog_3 n + O(1)$

Therefore, the comparison of Ternary and Binary Searches boils down the comparison of expressions $2log_3 n$ and $log_2 n$. The value of $2log_3 n$ can be written as $\frac{2}{log_2 3} \times log_2 n$.

Since the value of $\frac{2}{log_2 3}$ is more than one, Ternary Search does more comparisons than Binary Search in worst case.

2. Write a program in C or C++ to find the maximum and minimum number from a given array using divide and conquer approach.

***Algorithm 2:* Max-Min $(x[], i, j, max, min)$**

```
// x[0 : n − 1] is a an array of n elements.
   Parameters i and j are integers, 1 ≤ i ≤ j ≤ n.
   The effect is to set max and min to the largest
   and smallest values in x[i : j], respectively.
if i = j then max := min := x[i]; // Small P
else if i = j − 1 then
    // Another case of Small P
    if a[i] < a[j] then
        max := x[j]; min = x[i];
    end
    else
        max := x[i]; min = x[j];
    end
end
else
    // If P is not small, divide P into sub-problems
    mid := ⌊(i+l)/2⌋; // Find where to split the set
    // Solve the sub-problems
    Max-Min(x[], i, mid, max, min);
    Max-Min(x[], mid + 1, j, max1, min1);
    // Combine the solutions
    if max < max1 then max := max1;
    if min > min1 then min := min1;
end
```

3. Write a program in C or C++ to sort a given array using Merge Sort algorithm.

Algorithm 3: **MergeSort***(low,high)*

Input: An array $arr[low : high]$ is a global array to be sorted.
Output: Sorted array such that $arr[i] \leq arr[i+1]$ for all $1 \leq i \leq n$

```
// Small(P) is true if there is only one element to sort.
if low < high then
    mid := ⌊(low+high)/2⌋; // divide P into sub problems.
    MergeSort(low, mid); // Solve the first sub problem.
    MergeSort(mid + 1, high);
    // Solve the second sub problem.
    Merge(low, mid, high); // Combine the solutions.
end
```

Algorithm 4: **Merge***(low, mid, high)*

Input: An array $arr[low : high]$ is a global array containing two sorted subsets $arr[low : mid]$ and in $arr[mid + 1 : high]$.
Output: The goal is to merge these two sets into a single set residing in $arr[low : high]$. $b[]$ is an auxiliary array.

```
k := low; i := low; j := mid + 1;
while i ≤ mid and j ≤ high do
    if arr[i] < arr[j] then
        b[k] := arr[i]; i := i + 1;
    end
    else
        b[k] := arr[j]; j := j + 1;
    end
    k := k + 1;
end
while i ≤ mid do
    b[k] := arr[i]; i := i + 1; k := k + 1;
end
while j ≤ high do
    b[k] := arr[j]; j := j + 1; k := k + 1;
end
for i := low to high do arr[i] := b[i];
```