

DIVIDE AND CONQUER

AOA : Module 2

CONTENTS

- Binary Search
- Find Maximum and Minimum
- Merge Sort
- Quick Sort
- Strassen's Matrix Multiplication
- Heap Sort

INTRODUCTION

- Original problem is divided into similar kind of subproblems that are smaller in size and easy to be find.
- The solution of these small independent subproblems are combined to obtain the solution of whole problem.
- Divide and Conquer paradigm solves a problem in three steps at each level of recursion:
 1. Divide
 2. Conquer
 3. Combine

INTRODUCTION

- Time complexity to solve “Divide & Conquer” problem is given by recurrence relations.
- Recurrence relation is derived from algorithm and solved to calculate complexity.
- The general recurrence relation for divide and conquer is given as follows:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where, $T\left(\frac{n}{b}\right)$: time required to solve each subproblem

$f(n)$: time required to combine the solutions of all subproblems

BINARY SEARCH

- There are two approaches:
 1. Iterative or Non-recursive
 2. Recursive
- There is a linear Array ‘a’ of size ‘n’.
- Binary Search is one of the fastest searching algorithm.
- Binary Search can only be applied on “Sorted Arrays”- either ascending or descending order.
- We compare “key” with item in the middle position. If they are equal, search ends successfully.
- Otherwise,
 - if key is less than element present in the middle position,
 - then apply binary search on lower half,
 - else apply BINARY SEARCH on upper half of the array.
- Same process is applied to remaining half until match is found or there are no more elements left

BINARY SEARCH

Iterative Approach:

```
Algorithm IBinaryS(arr[ ], start, end, key){  
    int mid;  
    while(start<=end){  
        mid = (start + end)/2;  
        if (arr[mid] == key)  
            return 1;  
        if (arr[mid]<key)  
            start = mid+1;  
        else  
            end = mid-1;  
    }  
    return 0;  
}
```

Recursive Approach:

```
Algorithm RBinaryS(arr[ ], start, end, key){  
    int mid;  
    if (start > end) { return 0; }  
    else  
        mid = (start + end)/2;  
        if (key == arr[mid])  
            return (mid);  
        else  
            if (key < arr[mid]){  
                RBinaryS(arr[],key, start, mid-1)  
            }  
            else  
                RBinaryS(arr[],key, mid+1, end)  
    }  
}
```

FINDING MINIMUM AND MAXIMUM

Iterative Approach:

```
Algorithm MinMax(a[ ], n, max, min){  
    max=min=a[1];  
    for(i=2 to n)do  
    {  
        if(a[i]> max) then max=a[i];  
        if (a[i]< max) then min=a[i];  
    }  
}
```

Recursive Approach:

```
Algorithm MinMax(a[ ], l, h, max, min) {  
    if(l==h) then  
        max=min=a[l];  
    else if      (h-l==1), then  
    {  
        if(a[l]>=a[h]), then  
            max=a[l];  
            min=a[h];  
        else{  
            max=a[h];  
            min=a[l];  
        }  
    else{  
        Mid = (l+h)/2;  
        MinMax(l, mid, max, min);  
        MinMax(mid+1, h, max, min);  
        if(max< max1)  then max=max1;  
        if (min>min1)then, min = min1;  
    }  
}
```

FINDING MINIMUM AND MAXIMUM

Recursive Approach:

```
Algorithm MinMax(a[ ], l, h, max,  
min) {  
    if(l==h) then  
        max=min=a[l];  
    else if(h-l==1), then  
    {  
        if(a[l]>=a[h]), then  
            max=a[l];  
            min=a[h];  
        else{  
            max=a[h];  
            min=a[l];  
        }  
    }
```

```
else{  
    Mid = (l+h)/2;  
    MinMax(l, mid, max, min);  
    MinMax(mid+1, h, max1, min1);  
    if(a[max]< a[max1])  then  
        max=max1;  
    if (a[min]>a[min1])then,  
        min = min1;  
    }  
}
```

FINDING MINIMUM AND MAXIMUM

Time Complexity:

Recurrence relation :

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 2 & n > 2 \\ &= 1 & n = 2 \end{aligned}$$

MERGE SORT

- Simple and efficient algorithm for sorting a list of numbers
- Based on divide and Conquer paradigm
- Performed in three steps:
 - I. Divide:
 - i. List of n elements is divided into 2 sub-lists of $n/2$ elements
 - ii. Computes middle of the array, so it takes constant time $O(1)$.
 2. Conquer:
 1. Each half is sorted independently.
 2. Merge sort is recursively used to sort elements of smaller sub-lists.
 3. This step contributes $T(n/2) + T(n/2)$ to running time.

MERGE SORT

3. Combine:

- i. Two sorted halves are merged to obtain a sorted sequence
- ii. This requires merging of n elements into 1 list.
- iii. It contributes $O(n)$ to running time.

NOTE: The Key operation of merge sort is **Merging**

MERGE SORT ALGORITHM

```
mergeSort(arr[ ],low, high)
//arr is array, low is left sub-list, high is right sub-list
{
    if(low<high)
    {
        mid = (low+high)/2;
        mergeSort(arr, low, mid);
        mergeSort(arr,mid+1,high);
        merge(arr, low, mid, high);
    }
}
```

MERGE ALGORITHM

```
void merge(int arr[ ], int low, int mid,  
int high) {  
    int i = low;  
    int j = mid + 1;  
    int k = low;  
  
    /* create temp array */  
    int temp[5];
```

1

```
        while (i <= mid && j <= high) {  
            if (arr[i] <= arr[j]) {  
                temp[k] = arr[i];  
                i++;  
                k++;  
            }  
            else {  
                temp[k] = arr[j];  
                j++;  
                k++;  
            }  
        }
```

2

MERGE ALGORITHM

```
/* Copy the remaining elements  
of first half, if there are any */  
  
while (i <= mid) {  
    temp[k] = arr[i];  
    i++;  
    k++;  
}
```

3

```
/* Copy the remaining elements  
of 2nd half, if there are any */  
  
while (j <= high) {  
    temp[k] = arr[j];  
    j++;  
    k++;  
}
```

4

```
/* Copy the temp array to original array */  
for (int k = low; k <= high; k++) {  
    arr[k] = temp[k];  
}
```

5

MERGE SORT EXAMPLE

Example:

54	26	93	17	77	31	44	55
----	----	----	----	----	----	----	----