

## Experiment No. 1: Divide and Conquer Strategy

**Date:**

**Aim:** Write a C program to implement the following program using divide and conquer strategy

- a. Merge Sort and binary search
- b. Quick sort and Minmax algorithm
- c. Finding Kth smallest element
- d. Strassen's Matrix Multiplication

### **THEORY:**

The Divide-and-Conquer strategy suggests splitting the inputs into  $k$  distinct subsets,  $1 < k < n$ , yielding  $k$  subproblems. These subproblems must be solved, and then a method must be found to combine sub-solutions into a solution of the whole. If the sub problems are still relatively large, then the divide-and- conquer strategy can possibly be reapplied. D And C is initially invoked as D And C(P), where P is the problem to be solved. Small(P) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. The subproblems  $P_1, P_2, \dots, P_k$  Combine is a function that determines the solution to P using the solutions to the  $k$  subproblem.

This technique can be divided into the following three parts:

1. Divide: This involves dividing the problem into smaller sub-problems.
2. Conquer: Solve sub-problems by calling recursively until solved.
3. Combine: Combine the sub-problems to get the final solution of the whole problem.

For divide-and-conquer based algorithms that produce sub-problems of the same type as the original problem, it is very natural to first describe such algorithms using recursion.

Some standard algorithms that follow Divide and Conquer algorithm are:- Quick Sort, Merge Sort, Find MinMax elements.

## **a) Merge Sort and Binary Search algorithm**

**Date:**

### **Problem Statement:**

- (i) Write a c program to sort a character array in ascending order containing the following elements [ 'M','O','C','T','W','E','R','Y','B','J','P'] using merge sort algorithm and search for 'R' and 'F' using binary search.
- (ii) Write a c program to sort an integer array in descending order containing the following elements [ 53,-20,23,11,92,66,-11,85,26,34] using merge sort algorithm .
- (iii) Write a c program to search for 'R' and 'F' using binary search on array [ 'M','O','C','T','W','E','R','Y','B','J','P']

### **Algorithm**

(i) Algorithm MergeSort(low, high)

// a[low:high] is a global array to be sorted. small(P) is true if there is only 1 element to sort.

//In this case the list already sorted.

{ if(low < high) then. // If there are more than 1 element

{

    // Divide P into subproblems. //

    Find where to split the set.

    mid:=[(low + high)/2];

    // Solve the subproblems

    MergeSort(low, mid);

    MergeSort(mid+1, high); //

    Combine the solutions.

    Merge(low, mid, high);

}

}

Algorithm Merge(low, mid, high)

// a[low:high] is a global array containing 2 sorted subsets in a[low:mid] and in a[mid:high].

//The goal is to merge these 2 sets into single set residing in a[low:high]. b[] is an // auxiliary set.

{ h:=low; i:=low; j:=mid + 1; while((h <=

mid) and (j <= high)) do

{ if(a[h] <= a[j]) then b[i]:=a[h];

    h:=h+1;

    else b[i]:=a[j]; j:=j+1;

    i:=i+1;

}

```

        if(h > mid) then for k:=j to
            high do b[i]:=a[k];
            i:=i+1;
        else for k:=h to mid do
            b[i]:=a[k]; i:=i+1;
        for k:=low to high do a[k]:=b[k];
    }

```

(ii) Algorithm BinarySearch(a, I, l, x)

```

// Given an array a[i:l] of elements in ascending order, 1 ≤ I ≤ l,
//determine whether x is present, and if so, return j such that x=a[j]; else return 0.
{ if(l = i) then //If Small(P)
{ if(x = a[I]) then return I;
  else return 0;
}
else
{
    // Reduce P into a smaller subproblem.
    mid:=[(I+l)/2]; if(x = a[mid]) then
    return mid; else if (x < a[mid]) then
    return BinarySearch(a, i, mid-1, x);
    else return BinarySearch(a, mid+1, l, x);
}
}

;

    MaxMin(mid+1, j, max1, min1);
    //Combine the solutions.
    if(max<max1) then max := min1;
    if(min>min1) then min := min1;
}
}

```

## Time and Space Complexity

### (i) Merge sort

Merge Sort is a recursive algorithm. If the time for merging operation is proportional to  $n$ , then the computing time for merge sort is described by the recurrence relation:

$$T(n) = 2T(n/2) + cn, \quad n > 1, \quad c \text{ is a constant.}$$

$$T(1) = a, \text{ where 'a' is a constant}$$

Applying repeated substitution method:

When  $n$  is a power of 2,  $n = 2^k$ ,

$$\Rightarrow \log n = \log_2 n$$

$$\Rightarrow \log_2 n = k$$

So we can replace  $k$  with  $\log_2 n$

We can solve the equation as follows:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \end{aligned}$$

.

$$\begin{aligned} &= 2^k T(1) + kcn \\ &= an + cn \log_2 n \end{aligned}$$

We see that  $2^k < n < 2^{k+1}$ , then  $T(n) \leq T(2^{k+1})$

Therefore,  $T(n) = \Theta(n \log n)$

Space Complexity:  $O(n)$

### (ii) Binary Search

- The time complexity of the binary search algorithm is  $O(\log n)$ .
- The best-case time complexity would be  $O(1)$  when the central index would directly match the desired value.
- The worst-case scenario could be the values at either extremity of the list or values not in the list.
- The time complexity of Binary Search can be written as

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

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is : o According to master's theorem

$$\begin{aligned}
 & a=1, b=2 \quad n^{\log(\text{base } b)a} = n^{\log(2)1} = 1 \quad T(n) = \\
 & n^{\log(2)1} * U(n) \\
 & U(n) \rightarrow h(n) = F(n) / (n^{\log(2)1}) \\
 & = c / (n^{\log(2)1}) \Rightarrow c \Rightarrow r=0 \\
 & \Rightarrow i=0 \\
 & (\log(n))^{i+1} / (i+1) = (\log(n))
 \end{aligned}$$

► Compute Space and Time complexity to perform Binary Search

Algorithm	Time Complexity	Space Complexity
<b>Algorithm BS(a, key)</b> { <b>Low:=1;</b> <b>High:=n</b> <b>While low&lt;=high do</b> { <b>Mid:=(low+high)/2;</b> <b>If a[mid] &lt; key then</b> <b>Low:=mid+1;</b> <b>Else if a[mid] &gt; key then</b> <b>High:=mid - 1;</b> <b>Else</b> <b>Return mid;</b> } <b>Return 0;</b> } 	1 1 n n-1 n-1 n-1 0 0 1 0 	1 for low 1 for high 1 for mid 1 for n 1 for key n for a[n] 
<b>Total</b>	<b>4n</b>	<b>S(P) = 5 + n</b>

Therefore, the time complexity is  $T(n) = 1 * \log(n) = O(\log(n))$

## 1) MERGE SORT

```
#include <stdio.h>
void Merge_desc(int a[], int low, int mid, int high)
{
    int i = low, j = mid + 1, k = low;
    int b[100];
    while (i <= mid && j <= high)
    {
        if (a[i] >= a[j])
        {
            b[k++] = a[i++];
        }
        else
            b[k++] = a[j++];
    }

    if (i > mid)
    {
        for (int m = j; m <= high; m++)
        {
            b[k++] = a[m];
        }
    }
    else
    {
        for (int m = i; m <= mid; m++)
        {
            b[k++] = a[m];
        }
    }

    for (int m = low; m <= high; m++)
    {
        a[m] = b[m];
    }
}

void MergeSort_desc(int a[], int low, int high)
{
    if (low < high)
    {
        int mid = (low + high) / 2;
        MergeSort_desc(a, low, mid);
        MergeSort_desc(a, mid + 1, high);
        Merge_desc(a, low, mid, high);
    }
}

void Merge(int a[], int low, int mid, int high)
{
    int i = low, j = mid + 1, k = low;
    int b[100];
    while (i <= mid && j <= high)
    {
        if (a[i] <= a[j])
        {
            b[k++] = a[i++];
        }
        else
            b[k++] = a[j++];
    }
}
```

```

        if (i > mid)
        {
            for (int m = j; m <= high; m++)
            {
                b[k++] = a[m];
            }
        }
        else
        {
            for (int m = i; m <= mid; m++)
            {
                b[k++] = a[m];
            }
        }

        for (int m = low; m <= high; m++)
        {
            a[m] = b[m];
        }
    }
    void MergeSort(int a[], int low, int high)
    {
        if (low < high)
        {
            int mid = (low + high) / 2;
            MergeSort(a, low, mid);
            MergeSort(a, mid + 1, high);
            Merge(a, low, mid, high);
        }
    }
    void Merge_Char(char a[], int low, int mid, int high)
    {
        int i = low, j = mid + 1, k = low;
        int b[100];
        while (i <= mid && j <= high)
        {
            if (a[i] <= a[j])
            {
                b[k++] = a[i++];
            }
            else
            {
                b[k++] = a[j++];
            }
        }
        if (i > mid)
        {
            for (int m = j; m <= high; m++)
            {
                b[k++] = a[m];
            }
        }
        else
        {
            for (int m = i; m <= mid; m++)
            {
                b[k++] = a[m];
            }
        }
        for (int m = low; m <= high; m++)
        {

```

```

        a[m] = b[m];
    }
}
void MergeSort_Char(char a[], int low, int high)
{
    if (low < high)
    {
        int mid = (low + high) / 2;
        MergeSort_Char(a, low, mid);
        MergeSort_Char(a, mid + 1, high);
        Merge_Char(a, low, mid, high);
    }
}
int main()
{
    int a[] = {9, 8, 7, 6, 5, 4, 7};
    int size = sizeof(a) / sizeof(a[0]);
    MergeSort_desc(a, 0, size - 1);
    for (int i = 0; i < size; i++)
    {
        printf("%d\t", a[i]);
    }
    char b[] = {'a', 'b', 'v'};
    int size_c = sizeof(a) / sizeof(a[0]);
    MergeSort_Char(b, 0, 2);
    printf("\nChar : \n");
    for (int i = 0; i < 3; i++)
    {
        printf("%c\t", b[i]);
    }
    // user input
    int d[100];
    printf("\nUser input\n");
    printf("Enter size\n");
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &d[i]);
    }
    MergeSort_desc(d, 0, n - 1);

    for (int i = 0; i < n; i++)
    {
        printf("%d\t", d[i]);
    }
    // user input
    char d_c[100];
    printf("\nUser input char\n");
    printf("Enter size\n");
    int n1;
    scanf("%d", &n1);
    for (int i = 0; i < n1; i++)
    {
        scanf("%s", &d_c[i]);
    }
    MergeSort_Char(d_c, 0, n1 - 1);

    for (int i = 0; i < n1; i++)
    {

```



```

        printf("%c\t", d_c[i]);
    }
    printf("\n");
    return 0;
}

```

O / P :

```

● PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> gcc mergeSort.c -o run
● PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> ./run
9      8      7      7      6      5      4
Char :
a      b      v
User input
Enter size
3
3
2
1
3      2      1
User input char
Enter size
4
b
a
a
d
○ a      a      b      d
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_>

```

## 2) BINARY SEARCH

```

#include <stdio.h>
#define input(x) scanf("%d", &x)

int binarySearch_Char(char a[],int low,int high,char x)
{
    if(low <= high){
        int mid = (low+high)/2;
        if(a[mid] == x) return mid;
        if(x < a[mid]){
            return binarySearch_Char(a,low,mid-1,x);
        }
        else if(x > a[mid]) return binarySearch_Char(a,mid+1,high,x);
    }
    else return -1;
}

int binarySearch(int a[], int low, int high, int x)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;
        if (a[mid] == x)
            return mid;
        if (x < a[mid])
        {
            return binarySearch(a, low, mid - 1, x);
        }
    }
}

```

```

    }
    else if (x > a[mid])
        return binarySearch(a, mid + 1, high, x);
    }
    else
        return -1;
}
int main()
{
    int a[100];
    int n;
    printf("Enter Size of Array\n");
    input(n);

    printf("Enter Array\n");
    for (int i = 0; i < n; i++)
    {
        input(a[i]);
    }
    printf("Enter Element To be Searched\n");
    int x;
    input(x);
    printf("Found at Position %d\t", binarySearch(a, 0, n-1, x));
}
O / P :

```

```

PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> gcc binarySearch.c -o run
● PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> ./run
Enter Size of Array
3
Enter Array
1
2
3
Enter Element To be Searched
3
Found at Position 2

```

### **b) Quick sort and Minmax algorithm**

**Date:**

#### **Problem Statement:**

- (i) write a c program to sort the following integers using quickSort [ 17, 89 , 11 , 56 , 43 , -11 , 85 , 49 ]
- (ii) write a c program to find the minimum and maximum element among the given set of numbers [ 17, 89 , 11 , 56 , 43 , -11 , 85 , 49 ]

#### **Algorithm**

(i) Algorithm MaxMin(i, j, max, min)  
//a[1:n] is a global array. 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 a[i:j] respectively.  
{  
    if(i==j) then max := min := a[i]; //Small(P)  
    else if(i=j-1) then // Another case of Small(P)  
        {  
            if(a[i]<a[j]) then  
                {  
                    max := a[j]; min := a[i];  
                }  
            else

```

        {
            max := a[i]; min := a[j];
        }
    }
else
{
    // if P is not small, divide P into subproblems.
    //Find where to split the set.
    mid := (i+j)/2;
    //Solve the subproblems.
    MaxMin(i, mid, max, min);
    MaxMin(mid+1, j, max1, min1);
    //Combine the solutions.
    if(max<max1) then max := min1;
    if(min>min1) then min := min1;
}
}

```

(ii)

```

Algorithm QuickSort( $p, q$ )
// Sorts the elements  $a[p], \dots, a[q]$  which reside in the global
// array  $a[1 : n]$  into ascending order;  $a[n + 1]$  is considered to
// be defined and must be  $\geq$  all the elements in  $a[1 : n]$ .
{
    if ( $p < q$ ) then // If there are more than one element
    {
        // divide  $P$  into two subproblems.
         $j := \text{Partition}(a, p, q + 1)$ ;
        //  $j$  is the position of the partitioning element.
        // Solve the subproblems.
        QuickSort( $p, j - 1$ );
        QuickSort( $j + 1, q$ );
        // There is no need for combining solutions.
    }
}

```

### Time and Space Complexity

(i) Quick Sort

Best case:

$$T(n) = T(k) + T(n-k-1) + O(n)$$

Since the left and right sub-arrays have equal sizes, we can simplify the above relation as:

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

Using the Master Theorem, we can solve this relation and get the time complexity for the best case as:

$$T(n) = O(n \log n)$$

Worst case:

$$T(n) = T(0) + T(n-1) + O(n)$$

Since the left sub-array is empty, we can simplify the above relation as:

$$T(n) = T(n-1) + O(n)$$

Expanding  $T(n-1)$ , we get:

$$T(n) = T(n-2) + O(n-1) + O(n)$$

$$T(n) = T(n-3) + O(n-2) + O(n-1) + O(n)$$

...

$$T(n) = T(0) + O(1) + O(2) + \dots + O(n-1) + O(n)$$

$$T(n) = O(1+2+\dots+n)$$

$$T(n) = O(n^2)$$

Therefore, the time complexity of QuickSort algorithm in the worst case is  $O(n^2)$ .

Average case:

$$T(n) = T(k) + T(n-k-1) + O(n)$$

Since  $k$  is a constant fraction of  $n$ , we can assume that both sub-arrays have roughly equal sizes. Therefore, we can use the same recurrence relation as the one used in the best case scenario:

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

Using the Master Theorem, we can solve this relation and get the time complexity for the average case as:

$$T(n) = O(n \log n)$$

Space Complexity :  $O(n)$

(ii) Min Max

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

This recurrence relation can be solved using the Master Theorem, which gives us:

$$T(n) = O(n)$$

This means that the time complexity of the MinMax algorithm is proportional to the size of the input array, and does not depend on the arrangement of the elements in the array.

Space Complexity :  $O(n)$

## **Code**

### **(i) Quick Sort**

```
#include <stdio.h>

#define forn(a,n) for (int i = 0; i < n; i++){printf("%d\t",a[i]);}

void swap(int *a,int *b){
    int t = *a;
    *a = *b;
    *b = t;
}

int partition(int a[],int low,int high)
{
    int pivot = a[high];
    int infinity_index = low-1;
    for(int j = low ; j < high ; j++){
        if(a[j] < pivot)
        {
            infinity_index++;
            swap(&a[infinity_index],&a[j]);
        }
    }
    swap(&a[infinity_index+1],&a[high]);
    return infinity_index+1;
}

void quickSort(int a[],int low,int high){
    //base Case
    if(low < high)
    {
        int pi = partition(a,low,high);
```

```

        quickSort(a,low,pi-1);
        quickSort(a,pi+1,high);
    }
    printf("\n");
    for(int i = low ; i < high + 1 ;i++)
    {
        printf("%d\t",a[i]);
    }
}
int main()
{
    int a[100];
    int n;
    scanf("%d",&n);
    for(int i = 0 ; i < n ;i++)
    {
        scanf("%d",&a[i]);
    }
    quickSort(a,0,n-1);
    printf("\nLast : \n");
    for(int i = 0 ; i < n ;i++)
    {
        printf("%d\t",a[i]);
    }
    printf("\n");
}

```

### **Output :**

```
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> gcc quickSort.c -o run
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> ./run
5
5 4 3 2 1

3

3      4
2      3      4

2      3      4      5
1      2      3      4      5
Last :
1      2      3      4      5
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_>
```

## Code :

### (ii) Min Max

```
#include "stdio.h"

void MinMax(int a[],int i,int j,int *max,int *min)
{
    int min1 = 0 , max1 = 0;

    if(i == j)
    {
        *max = a[i];
        *min = a[i];

        printf("max : %d min : %d\n",*max,*min);

        return;
    }
    else if(i == j-1)
    {
        if(a[i] > a[j]){
            *max = a[i];
            *min = a[j];
        }
        else{
            *max = a[j];
            *min = a[i];
        }
    }
}
```



```

    }

    printf("max : %d min : %d\n",*max,*min);

    return;
}

else{
    int mid = (i+j)/2;

    MinMax(a,i,mid,max,min);

    MinMax(a,mid+1,j,&max1,&min1);

    if(min1 < *min) *min = min1;

    if(max1 > *max) *max = max1;

    printf("max : %d min : %d\n",*max,*min);

    return;
}
}

int main()
{
    int a[100];

    int size;

    scanf("%d",&size);

    for(int i = 0 ; i < size ; i++)
    {
        scanf("%d",&a[i]);
    }

    int max = 0 , min = 0;

    MinMax(a,0,size-1,&max,&min);

    printf(" Final max : %d min : %d\n",max,min);
}

```

### **Output :**

```
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> gcc minmax.c -o run
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> ./run
5
1
2
3
4
5
max : 2 min : 1
max : 3 min : 3
max : 3 min : 1
max : 5 min : 4
max : 5 min : 1
Final max : 5 min : 1
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_>
```

### c) Finding kth smallest element

Date:

#### Problem Statement:

(i) find 6th and 2nd Smallest Element among the following

[25 , 33 , 13 , 56 , 95 , 110 , 80 , 72 , 66 , 88 , 76 ]

#### Algorithm :

```
// Original value for left = 0 and right = n-1
int kthSmallest(int A[], int left, int right, int K)
{
    if (left == right)
        return A[left]
    int pos = partition(A, left, right)
    count = pos - left + 1
    if ( count == K )
        return A[pos]
    else if ( count > K )
        return kthSmallest(A, left, pos-1, K)
    else
        return kthSmallest(A, pos+1, right, K-i)
}
```

```
int partition(int A[], int l, int r)
{
    int x = A[r]
    int i = l-1
    for ( j = l to r-1 )
    {
        if (A[j] <= x)
        {
            i = i + 1
            swap(A[i], A[j])
        }
    }
    swap(A[i+1], A[r])
    return i+1
}
```

## **Time and Space Complexity :**

Best case:

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

Using the Master Theorem, we can solve this relation and get the time complexity for the best case as:

$$T(n) = O(n)$$

Worst case:

$$T(n) = T(n-1) + O(n)$$

Using a similar argument as for the worst case of QuickSort, we can expand the recurrence relation as:

$$T(n) = T(n-2) + O(n-1) + O(n)$$

$$T(n) = T(n-3) + O(n-2) + O(n-1) + O(n)$$

...

$$T(n) = T(1) + O(2) + O(3) + \dots + O(n-1) + O(n)$$

$$T(n) = O(1+2+\dots+n-1) + O(n)$$

$$T(n) = O(n^2)$$

Therefore, the time complexity of the kth smallest element algorithm in the worst case is  $O(n^2)$ .

Average case:

$$T(n) = T(k) + T(n-k-1) + O(n)$$

Using a similar argument as for the average case of QuickSort, we can assume that both sub-arrays have roughly equal sizes. Therefore, we can use the same recurrence relation as the one used in the best case scenario:

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

Using the Master Theorem, we can solve this relation and get the time complexity for the average case as:

$$T(n) = O(n)$$

Space Complexity :  $O(\log(n))$

## **Code :**

### Kth Smallest Element (INT)

```
#include <stdio.h>
void swap(int *a,int *b){
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int a[],int low,int high)
{
    int pivot = a[high];
    int infinity_index = low-1;
    for(int j = low ; j < high ; j++){
        if(a[j] < pivot)
        {
```

```

        infinity_index++;
        swap(&a[infinity_index],&a[j]);
    }

}
swap(&a[infinity_index+1],&a[high]);
return infinity_index+1;
}
int kSmallest(int a[],int low,int high,int k){
    int pi = partition(a,low,high);
    if(pi == k)
    {
        printf("%d is the %dnd smallest element\n",a[pi],k+1);
        return a[pi];
    }
    else if(k < pi)
    {
        return kSmallest(a,low,pi-1,k);
    }
    else return kSmallest(a,pi+1,high,k);
}
int main()
{
    int a[] = {9,6,4,2,1,5,3};
    int low = 0 , high = 6;
    int k = 2;
    k--;
    printf("Enter size of Array\n");
    scanf("%d",&high);
    for(int i = 0 ; i < high ; i++)
    {
        scanf("%d",&a[i]);
    }
    high == 0? high = 0 : high--;
    // user input
    printf("Enter the Value of k\n");
    scanf("%d",&k);
    k == 0 ? k = 0 : k--;
    kSmallest(a,low,high,k);
}

```

## Kth Smallest (CHAR)

```

#include <stdio.h>
void swap(char *a,char *b){
    char t = *a;
    *a = *b;
    *b = t;
}
int partition(char a[],int low,int high)
{
    int pivot = a[high];
    int infinity_index = low-1;
    for(int j = low ; j < high ; j++){
        if(a[j] < pivot)
        {
            infinity_index++;
            swap(&a[infinity_index],&a[j]);
        }
    }
}

```

```

    }
    swap(&a[infinity_index+1],&a[high]);
    return infinity_index+1;
}
int kSmallest(char a[],int low,int high,int k){
    int pi = partition(a,low,high);
    if(pi == k)
    {
        printf("%c is the %dnd smallest element\n",a[pi],k+1);
        return a[pi];
    }
    else if(k < pi)
    {
        return kSmallest(a,low,pi-1,k);
    }
    else return kSmallest(a,pi+1,high,k);
}

int main()
{
    char a[100];
    int low = 0 , high = 6;
    int k = 2;
    k--;
    printf("Enter size of Array\n");
    scanf("%d",&high);
    for(int i = 0 ; i < high ; i++)
    {
        scanf(" %c",&a[i]);
    }
    high == 0? high = 0 : high--;
    // user input
    printf("Enter the Value of k\n");
    scanf("%d",&k);
    k == 0 ? k = 0 : k--;
    kSmallest(a,low,high,k);
}

```

### **Output :**

```

PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> gcc kSmallestElement.c -o run
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> ./run
Enter size of Array
3
1
2
3
Enter the Value of k
3
3 is the 3rd smallest element
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> gcc char_ksmallestElement.c -o run
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> ./run
Enter size of Array
3
a
b
c
Enter the Value of k
2
b is the 2nd smallest element
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_>

```

#### d) Strassen's Matrix Multiplication

Date:

#### Problem Statement:

Consider 4x4 Matrices A and B . Using Strassens's Matrix Multiplication find  $A \times B = C$  , where C is also a 4x4 matrix .

#### Algorithm :

```

Algorithm Strassens(a,n){
if(n<=2)
{
Q = A11(B12 - B22)
R = (A11 + A12)B22
S = (A21 + A22)B11
T = A22(B21 - B11)
P = (A11 + A22)(B11 + B22)
U = (A12 - A22)(B21 + B22)
V = (A11 - A21)(B11 + B12)
C11 = P + Q - T + V
C12 = R + T
C21 = Q + S
C22 = P + R - S + U
return C = [C11 C12; C21 C22];
}
else
{

```

```

//divide the matrix and combine;
c1 = Strassens(a,n/2) , c2 = Strassens(a,n/2);
c3 = Strassens(a,n/2) , c4 = Strassens(a,n/2);
c = combine(c1,c2,c3,c4);
return c
}
}

```

### **Time and Space Complexity**

$$T(n) = 7T(n/2) + O(n^2)$$

Using the Master Theorem, we can solve this relation and get the time complexity of Strassen's algorithm as follows:

$$a = 7, b = 2, d = 2$$

$$\log_b(a) = \log_2(7) \approx 2.81 > d$$

Therefore, we can use Case 1 of the Master Theorem, which gives us:

$$T(n) = \Theta(n^{\log_2(7)})$$

Space Complexity :  $O(n^2)$

### **Code :**

```

#include <stdio.h>

#define sint(x) scanf("%d", &x)

#define N 32 // can be changed

void add(int a[N][N], int b[N][N], int c[N][N], int n, int sn)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            c[i][j] = sn ? a[i][j] - b[i][j] : a[i][j] + b[i][j];
}

void copy(int a[N][N], int b[N][N], int r1, int r2, int c1, int c2)
{
    int n = r2 - r1;

    for (int i = r1, k = 0; i < r2 && k < n; i++, k++)
        for (int j = c1, l = 0; j < c2 && l < n; j++, l++)

```

```

        b[k][l] = a[i][j];
    }

void combine(int C11[N][N], int C12[N][N], int C21[N][N], int C22[N][N], int C[N][N], int n)
{
    for (int i = 0; i < n / 2; i++)
        for (int j = 0; j < n / 2; j++)
            C[i][j] = C11[i][j];

    for (int i = 0; i < n / 2; i++)
        for (int j = 0, k = n / 2; j < n / 2 && k < n; j++, k++)
            C[i][k] = C12[i][j];

    for (int i = 0, k = n / 2; i < n / 2 && k < n; i++, k++)
        for (int j = 0; j < n / 2; j++)
            C[k][j] = C21[i][j];

    for (int i = 0, k = n / 2; i < n / 2 && k < n; i++, k++)
        for (int j = 0, l = n / 2; j < n / 2 && l < n; j++, l++)
            C[k][l] = C22[i][j];
}

void strassen(int a[N][N], int b[N][N], int c[N][N], int n)
{
    if (n == 1)
    {
        c[0][0] = a[0][0] * b[0][0];
    }
    else
    {
        int mid = n / 2;

        int P[N][N], Q[N][N], R[N][N], S[N][N], T[N][N], U[N][N], V[N][N];
    }
}

```



```

int A11[N][N], A12[N][N], A21[N][N], A22[N][N];
int B11[N][N], B12[N][N], B21[N][N], B22[N][N];
int C11[N][N], C12[N][N], C21[N][N], C22[N][N];
int buf1[N][N], buf2[N][N];

copy(a, A11, 0, mid, 0, mid);
copy(a, A12, 0, mid, mid, n);
copy(a, A21, mid, n, 0, mid);
copy(a, A22, mid, n, mid, n);

copy(b, B11, 0, mid, 0, mid);
copy(b, B12, 0, mid, mid, n);
copy(b, B21, mid, n, 0, mid);
copy(b, B22, mid, n, mid, n);

add(A11, A22, buf1, mid, 0), add(B11, B22, buf2, mid, 0), strassen(buf1, buf2, P, mid);
add(A21, A22, buf1, mid, 0), strassen(buf1, B11, Q, mid);
add(B12, B22, buf2, mid, 1), strassen(A11, buf2, R, mid);
add(B21, B11, buf2, mid, 1), strassen(A22, buf2, S, mid);
add(A11, A12, buf1, mid, 0), strassen(buf1, B22, T, mid);
add(A21, A11, buf1, mid, 1), add(B11, B12, buf2, mid, 0), strassen(buf1, buf2, U, mid);
add(A12, A22, buf1, mid, 1), add(B21, B22, buf2, mid, 0), strassen(buf1, buf2, V, mid);

add(P, S, buf1, mid, 0), add(V, T, buf2, mid, 1), add(buf1, buf2, C11, mid, 0);
add(R, T, C12, mid, 0);
add(Q, S, C21, mid, 0);
add(P, R, buf1, mid, 0), add(U, Q, buf2, mid, 1), add(buf1, buf2, C22, mid, 0);

combine(C11, C12, C21, C22, c, n);
}
}

```

```

int main()
{
    int size = 4;

    int arr[N][N] = {{3, -7, 1, -9}, {6, -4, 8, -3}, {-2, 1, -3, 8}, {5, 9, -6, 5}};

    int brr[N][N] = {{-6, 4, -6, 9}, {8, -2, 7, 3}, {-9, 4, 5, -2}, {5, -3, 8, -7}};

    int crr[N][N];

    strassen(arr, brr, crr, size);

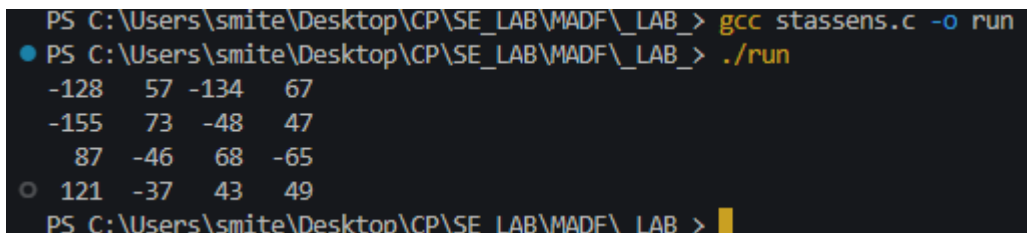
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
            printf("%4d ", crr[i][j]);

        printf("\n");
    }

    return 0;
}

```

### **Output :**



```

PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> gcc stassens.c -o run
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_> ./run
-128  57 -134  67
-155  73  -48  47
 87  -46  68  -65
121  -37  43   49
PS C:\Users\smite\Desktop\CP\SE_LAB\MADF\_LAB_>

```

### **CONCLUSION:**

Divide and Conquer strategy was studied. The programs for (a) Merger sort and binary search, (b) Quick sort and Minmax algorithm, (c) finding kth smallest element and (d) Strassen's Matrix multiplication algorithms were studied and implemented successfully.

