DIVIDE AND CONQUER

Given a function to compute on **n** inputs, the divide-and-conquer strategy suggests splitting the inputs into **k** distinct subsets, where **$1 < k \le n$**, yielding **k** subproblems. These subproblems must be solved, and then a method must be found to combine their solutions into a solution for the whole problem.

If the subproblems remain relatively large, the divide-and-conquer strategy can be reapplied recursively. Often, the subproblems generated are of the same type as the original problem. In such cases, the divide-and-conquer principle is naturally implemented using **recursion**. This process continues, breaking the problem into smaller and smaller subproblems, until they become small enough to be solved directly without further splitting.
To be more precise, consider the divide-and-conquer strategy when it splits the input into **two** subproblems of the same kind as the original problem. This **binary splitting** approach is common in many problems we will encounter.

We can define a **control abstraction**, which outlines the structure of a divide-and-conquer-based program. A **control abstraction** refers to a procedure where the flow of control is clear, but the primary operations are specified by other procedures whose exact implementation details are left undefined.
Let the **n** inputs be stored (or referenced) by the array **A(1:n)**. We assume this array is **global** to the algorithm. The procedure **DANDC** is a function initially invoked as DANDC(1,n).The function **DANDC(p, q)** solves a problem instance defined by the inputs **A(p:q)**.

Recurrence Equation – Divide and Conquer

In the **Divide and Conquer** strategy, a problem P of size n is solved by:

1. Checking if the problem is **small enough** to be solved directly.
2. If so, a direct solution is computed using a function S(P).
3. Otherwise, the problem is divided into k smaller **subproblems**: $P_1, P_2, ..., P_k$.
4. Each subproblem is solved recursively using the same approach.
5. The final result is obtained by **combining** the solutions of these subproblems using a function called Combine.

Let Small(P) be a Boolean function that returns true if the input size is small enough to be solved directly.
If the size of problem P is n, and the sizes of the k subproblems are $n_1, n_2, ..., n_k$, then the time complexity T(n) of the divide and conquer algorithm is defined by the following recurrence relation:

- $T(n) = g(n)$, if n is small
- $T(n) = T(n_1) + T(n_2) + ... + T(n_k) + f(n)$, otherwise

Where:

- T(n) is the time to solve the problem of size n
- g(n) is the time to solve small inputs directly
- f(n) is the time to divide the problem and combine the results

Standard Recurrence Form

Many divide-and-conquer algorithms follow a standard recurrence of the form:

- $T(n) = T(1)$, if $n = 1$
- $T(n) = a * T(n / b) + f(n)$, if $n > 1$

Where:

- a is the number of subproblems
- b is the factor by which the problem size is reduced
- f(n) is the time to divide and combine
- It is assumed that n is a power of b and T(1) is known

EXPERIMENT NO. : 1 **DATE :**

BINARY SEARCH

Aim:- Write a c program to implement recursive binary search on a given set of elements..

Problem statement:- Let a_i, $1 \leq i \leq n$, be a list of elements that are sorted in nondecreasing order. Determine whether a given element x is present in the list. If x is present, we are to determine a value j such that $a_j = x$. If x is not in the list, then j is to be set to zero. Let P=(n,ai,.....,al,x) denote an arbitrary instance of this search problem where n is the number of elements in the list, $a_i, \ldots, a_l$ is the list of elements and x is the element to be searched for.

Algorithm

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

Recurrence Equation

$$T(n) = \begin{cases} 1 & , \text{ if } n = 1 \\ T(n/2) + 1 & , \text{ if } n > 1 \end{cases}$$

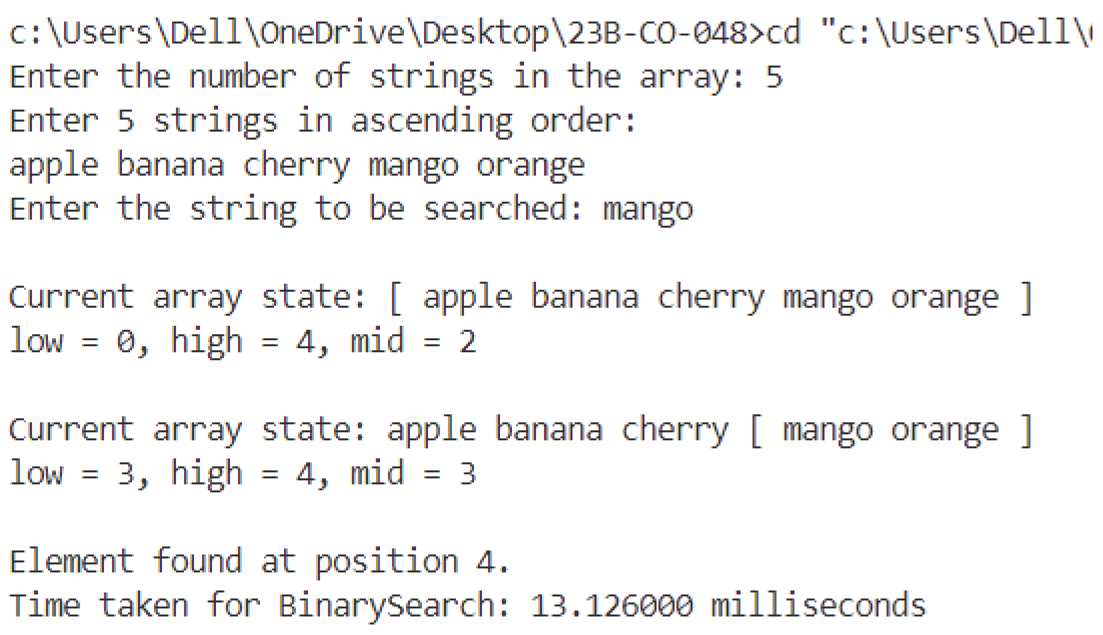
Time Complexity: O(logn)
Space Complexity: O(logn)

PROGRAM:

```
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

#define MAX 50

int n;
int BinarySearch(char a[][MAX], int low, int high, char *x);
void display(char a[][MAX], int low, int high);

int main() {
    int i, result;
    char arr[MAX][MAX], key[MAX];

    printf("Enter the number of strings in the array: ");
    scanf("%d", &n);

    if (n > 0) {
        printf("Enter %d strings in ascending order:\n", n);
        for (i = 0; i < n; i++) {
            scanf("%s", arr[i]);
        }

        printf("Enter the string to be searched: ");
        scanf("%s", key);
        printf("\n");

        struct timeval start_time, end_time;
        gettimeofday(&start_time, NULL);

        result = BinarySearch(arr, 0, n - 1, key);

        gettimeofday(&end_time, NULL);

        if (result == -1)
            printf("Element not found\n");
        else
            printf("Element found at position %d.\n", result + 1);

        double time_taken = (end_time.tv_sec - start_time.tv_sec) * 1000.0 +
                            (end_time.tv_usec - start_time.tv_usec) / 1000.0;

        printf("Time taken for BinarySearch: %.6f milliseconds\n", time_taken);
    }

    return 0;
}

int BinarySearch(char a[][MAX], int low, int high, char *x) {
    display(a, low, high);
    int mid;

    if (low > high) {
        return -1;
    }

    mid = (low + high) / 2;

    int cmp = strcmp(a[mid], x);
```

```
      if (cmp == 0)
        return mid;
      else if (cmp < 0)
        return BinarySearch(a, mid + 1, high, x);
      else
        return BinarySearch(a, low, mid - 1, x);
    }

    void display(char a[][MAX], int low, int high) {
      int i;
      printf("Current array state: ");
      for (i = 0; i < n; i++) {
        if (i == low)
          printf("[ ");
        printf("%s ", a[i]);
        if (i == high)
          printf("] ");
      }
  printf("\n");
  printf("low = %d, high = %d, mid = %d\n\n", low, high, (low + high) / 2);
}
```

OUTPUT:



```
c:\Users\Dell\OneDrive\Desktop\23B-CO-048>cd "c:\Users\Dell\
Enter the number of strings in the array: 5
Enter 5 strings in ascending order:
apple banana cherry mango orange
Enter the string to be searched: mango

Current array state: [ apple banana cherry mango orange ]
low = 0, high = 4, mid = 2

Current array state: apple banana cherry [ mango orange ]
low = 3, high = 4, mid = 3

Element found at position 4.
Time taken for BinarySearch: 13.126000 milliseconds
```

Conclusion:

The given code performs a binary search to locate a specific string within a sorted array of strings. By repeatedly comparing the target string (item) with the middle element of the current range (arr[mid]), the algorithm efficiently reduces the search space by half with each recursive call. Throughout the process, the program outputs the current values of low, mid, and up, providing a clear view of how the search converges toward the desired element

EXPERIMENT NO. : 2 **DATE :**

MERGE SORT

Aim:- Write a c program to implement a merge sort on a given set of elements.

Problem statement:- Given a sequence of n elements a[1],....,a[n], they are to be split into two sets a[1],...a[n/2] and a[n/2 +1],...,a[n]. Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.

Algorithm

```
Algorithm MergeSort(low, high)
// a[low :high] is a global array to be sorted. Small(P) is true if there is only one element to sort.In
this //case the list is already sorted.
{
if (low< high) then //If there are more than one element  {
// DivideP into subproblems.
// Find where to split the set.
mid:=[(low+high)/2];
// Solve the subproblems.
 MergeSort(low,mid);
 MergeSort(mid+1,high);
// Combinethe solutions.
      Merge(low, mid,high);
}
}

Algorithm Merge(low, mid,high)
//a[low :high] is a global array containing two sorted subsets in a [low:mid] and in
//a[mid+1 : high].The goal is to merge these two sets into a single set residing in a[low :high]. b[ ] is
//an auxiliary global array.
{
      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];
}
```

Recurrence relation

T(n) = { a if n = 1, a is a constant
2T(n/2) + cn if n > 1, c is a constant

Time Complexity

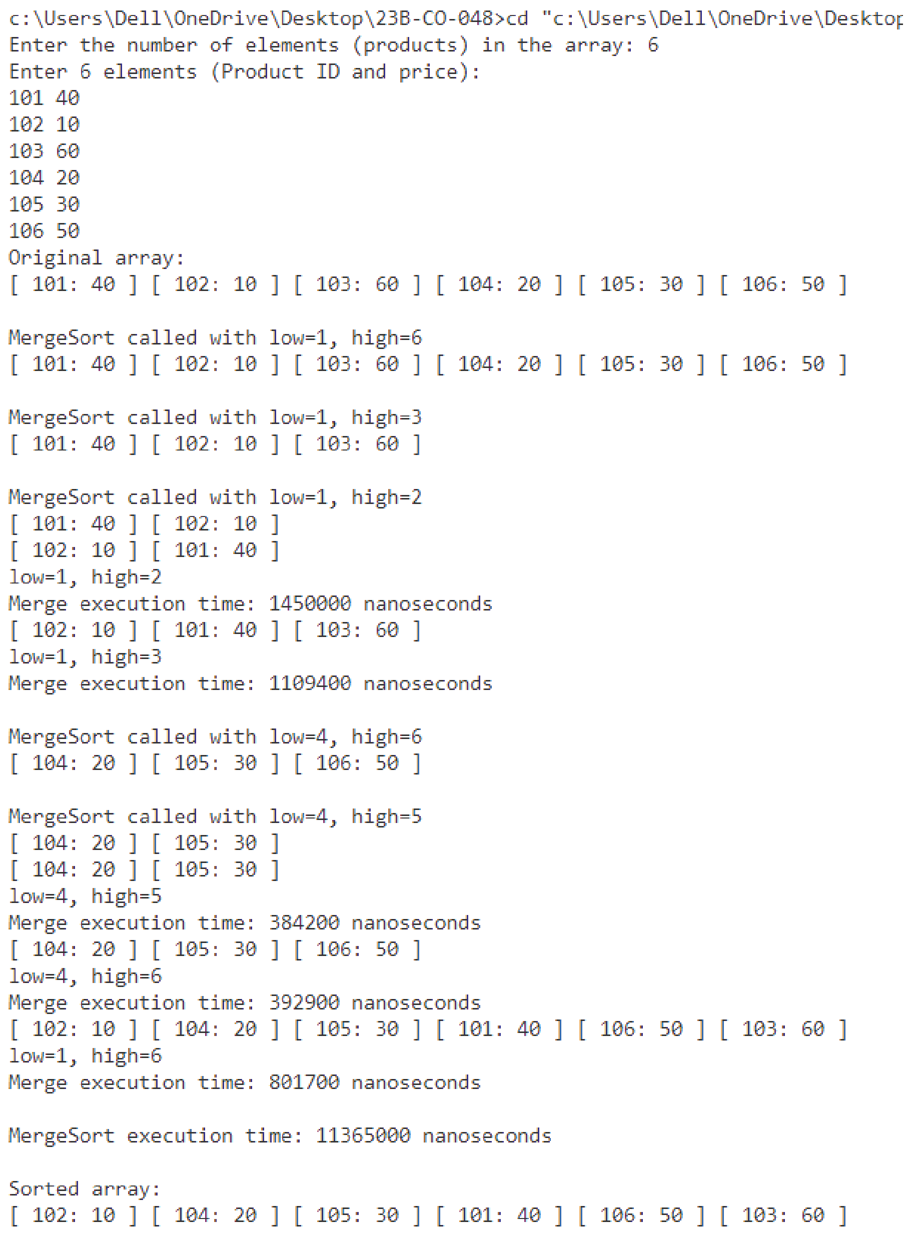
Best Case: O(n log n)
Worst Case: O(n log n)
Average Case: O(n log n)

Space Complexity: Auxiliary Space: O(n)

PROGRAM:

```
#include <stdio.h>
#include <windows.h>
#define MAX 50
typedef struct {
    int id;
    int value;
} Element;

void MergeSort(int low, int high);
void Merge(int low, int mid, int high);
void display(int low, int high);

Element a[MAX], b[MAX];
int n;
long long getTimeNanoseconds() {
    LARGE_INTEGER frequency, now;
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&now);
    return (now.QuadPart * 1000000000LL) / frequency.QuadPart;
}
int main() {
    int i;
    long long start, end;

    printf("Enter the number of elements (products) in the array: ");
    scanf("%d", &n);

    printf("Enter %d elements (Product ID and price):\n", n);
    for (i = 1; i <= n; i++) {
        scanf("%d %d", &a[i].id, &a[i].value);
    }
    printf("Original array:\n");
    for (i = 1; i <= n; i++) {
        printf("[ %d: %d ] ", a[i].id, a[i].value);
    }
    printf("\n");

    start = getTimeNanoseconds();
    MergeSort(1, n);
    end = getTimeNanoseconds();
    printf("\nMergeSort execution time: %lld nanoseconds\n", end - start);

    printf("\nSorted array:\n");
    for (i = 1; i <= n; i++) {
        printf("[ %d: %d ] ", a[i].id, a[i].value);
    }
    printf("\n");

    return 0;
}

void MergeSort(int low, int high) {
    if (low < high) {
        printf("\nMergeSort called with low=%d, high=%d\n", low, high);
        display(low, high);

        int mid = (low + high) / 2;
        MergeSort(low, mid);
        MergeSort(mid + 1, high);

        long long start = getTimeNanoseconds();
        Merge(low, mid, high);
        long long end = getTimeNanoseconds();

        printf("Merge execution time: %lld nanoseconds\n", end - start);
    }
}

void Merge(int low, int mid, int high) {
    int i = low, h = low, j = mid + 1;

    while (h <= mid && j <= high) {
        if (a[h].value <= a[j].value) {
            b[i++] = a[h++];
        } else {
            b[i++] = a[j++];
        }
    }
    while (h <= mid) {
        b[i++] = a[h++];
    }

    while (j <= high) {
        b[i++] = a[j++];
    }

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

    display(low, high);
    printf("low=%d, high=%d\n", low, high);
}
void display(int low, int high) {
    for (int i = low; i <= high; i++) {
        printf("[ %d: %d ] ", a[i].id, a[i].value);
    }
    printf("\n");
}
```

OUTPUT:



```
c:\Users\Dell\OneDrive\Desktop\23B-CO-048>cd "c:\Users\Dell\OneDrive\Deskto
Enter the number of elements (products) in the array: 6
Enter 6 elements (Product ID and price):
101 40
102 10
103 60
104 20
105 30
106 50
Original array:
[ 101: 40 ] [ 102: 10 ] [ 103: 60 ] [ 104: 20 ] [ 105: 30 ] [ 106: 50 ]

MergeSort called with low=1, high=6
[ 101: 40 ] [ 102: 10 ] [ 103: 60 ] [ 104: 20 ] [ 105: 30 ] [ 106: 50 ]

MergeSort called with low=1, high=3
[ 101: 40 ] [ 102: 10 ] [ 103: 60 ]

MergeSort called with low=1, high=2
[ 101: 40 ] [ 102: 10 ]
[ 102: 10 ] [ 101: 40 ]
low=1, high=2
Merge execution time: 1450000 nanoseconds
[ 102: 10 ] [ 101: 40 ] [ 103: 60 ]
low=1, high=3
Merge execution time: 1109400 nanoseconds

MergeSort called with low=4, high=6
[ 104: 20 ] [ 105: 30 ] [ 106: 50 ]

MergeSort called with low=4, high=5
[ 104: 20 ] [ 105: 30 ]
[ 104: 20 ] [ 105: 30 ]
low=4, high=5
Merge execution time: 384200 nanoseconds
[ 104: 20 ] [ 105: 30 ] [ 106: 50 ]
low=4, high=6
Merge execution time: 392900 nanoseconds
[ 102: 10 ] [ 104: 20 ] [ 105: 30 ] [ 101: 40 ] [ 106: 50 ] [ 103: 60 ]
low=1, high=6
Merge execution time: 801700 nanoseconds

MergeSort execution time: 11365000 nanoseconds

Sorted array:
[ 102: 10 ] [ 104: 20 ] [ 105: 30 ] [ 101: 40 ] [ 106: 50 ] [ 103: 60 ]
```

<u>Conclusion:</u> The code effectively demonstrates the implementation of the Merge Sort algorithm for sorting an array in ascending order. The merge_sort function recursively divides the array into two halves, sorts each half, and then combines them using the merge function. The merging process ensures that two sorted subarrays are efficiently combined into a single sorted array in ascending order, highlighting the power of divide-and-conquer in sorting algorithms.

EXPERIMENT NO. : 3 **DATE :**

FINDING MINIMUM AND MAXIMUM ELEMENT

Aim:- Write a C program to find Minimum and maximum element in the given array.

Problem statement:- Let P =(n,a[i],...,a[j]) denote an arbitrary instance of the problem. Here n is the number of elements in the list and we have to find maximum and minimum elements in this list. The situation of i=j and i=j-1 are handled separately. For the sets containing more than two elements, the midpoint is determined and two new subproblems are generated. When the maxima and minima of these subproblems are determined, the two maxima are compared and the two minima are compared to get the solution for the entire set.

Algorithm:

```
AlgorithmMaxMin(i, j,max,min)
// a[l: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:=max1;
if (min >min1) then min:=min1;
}
}
```

Recurrence relation:

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & \text{if } n > 2 \\ 1 & \text{if } n = 2 \\ 0 & \text{if } n = 1 \end{cases}$$

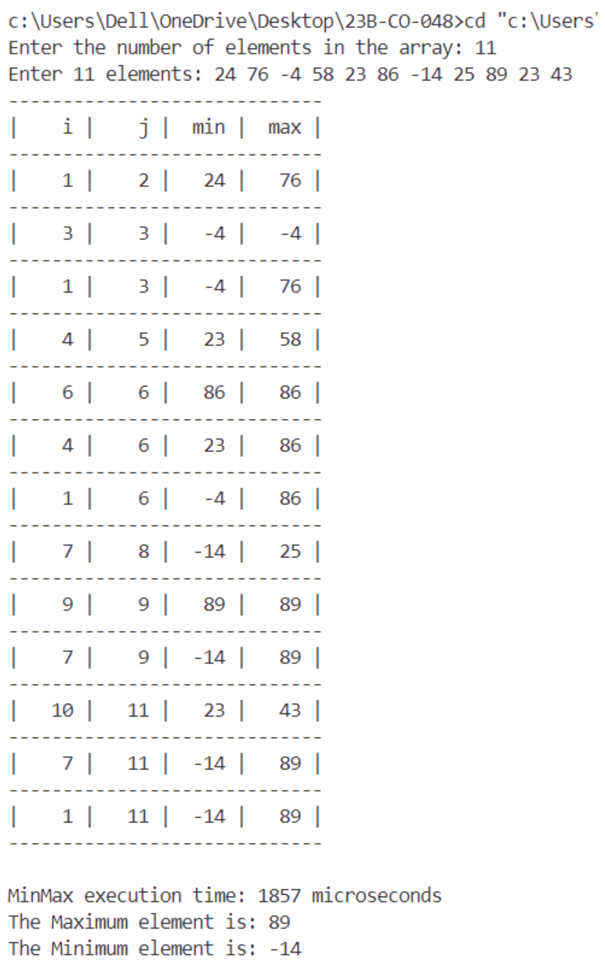
Time Complexity: O(n)

Space Complexity: O(logn)

PROGRAM:

```
#include<stdio.h>
#include <time.h>
#define MAX 50
int a[MAX];
void MinMax(int i,int j,int* min,int* max);
void display(int i,int j,int min,int max);

int main() {
    int i,n,min,max;
    printf("Enter the number of elements in the array :");
    scanf("%d",&n);
    printf("Enter %d elements :",n);
    for(i=1;i<=n;i++)  {
        scanf("%d",&a[i]);    }
    printf("-----------------------------\n");
    printf("| %4s | %4s | %4s | %4s |\n","i","j","min","max");
    printf("-----------------------------\n");
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
    MinMax(1,n,&min,&max);
    clock_gettime(CLOCK_MONOTONIC, &end);
   long long minmaxTime = (end.tv_sec - start.tv_sec) * 1000000LL + (end.tv_nsec / 1000);
    printf("\nMinMax execution time: %lld microseconds\n", minmaxTime);
    printf("The Maximum element is : %d\n",max);
    printf("The Minimum element is : %d\n",min);  }

void MinMax(int i,int j,int* min,int* max)  {
    int mid,min1,max1;
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
    if(i==j) {
        *min=a[i];
        *max=a[i];
        display(i,j,*min,*max);
    }
    else if(i==j-1)  {
        if(a[i]>=a[j])  {
            *max=a[i];
            *min=a[j];
            display(i,j,*min,*max);
        }
        else {
            *max=a[j];
            *min=a[i];
            display(i,j,*min,*max);
        }
    }
    else{
        mid=(i+j)/2;
        MinMax(i,mid,min,max);
        MinMax(mid+1,j,&min1,&max1);
        if(*min>min1)
            *min=min1;
        if(*max<max1)
            *max=max1;
        display(i,j,*min,*max);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
long long minmaxSegmentTime = (end.tv_sec - start.tv_sec) * 1000000LL + (end.tv_nsec /
1000);
    printf("MinMax segment execution time: %lld microseconds\n", minmaxSegmentTime);
}

void display(int i,int j,int min,int max)  {
    printf("| %4d | %4d | %4d | %4d |\n",i,j,min,max);
    printf("-----------------------------\n");
}
```

OUTPUT:



```
c:\Users\Dell\OneDrive\Desktop\23B-CO-048>cd "c:\Users'
Enter the number of elements in the array: 11
Enter 11 elements: 24 76 -4 58 23 86 -14 25 89 23 43
-----------------------------
|    i |    j |  min |  max |
-----------------------------
|    1 |    2 |   24 |   76 |
-----------------------------
|    3 |    3 |   -4 |   -4 |
-----------------------------
|    1 |    3 |   -4 |   76 |
-----------------------------
|    4 |    5 |   23 |   58 |
-----------------------------
|    6 |    6 |   86 |   86 |
-----------------------------
|    4 |    6 |   23 |   86 |
-----------------------------
|    1 |    6 |   -4 |   86 |
-----------------------------
|    7 |    8 |  -14 |   25 |
-----------------------------
|    9 |    9 |   89 |   89 |
-----------------------------
|    7 |    9 |  -14 |   89 |
-----------------------------
|   10 |   11 |   23 |   43 |
-----------------------------
|    7 |   11 |  -14 |   89 |
-----------------------------
|    1 |   11 |  -14 |   89 |
-----------------------------

MinMax execution time: 1857 microseconds
The Maximum element is: 89
The Minimum element is: -14
```

Conclusion:- The above code implements a function MinMax that finds the minimum and maximum elements of an array using a divide-and-conquer approach. The MinMax function works recursively, dividing the array into smaller subarrays until each subarray has one or two elements. During the recursion, the MinMax function prints the values of i, j, the current min, and max.

EXPERIMENT NO. : 4 **DATE :**

QUICK SORT

Aim:- Write a C program to sort an array of structures using quick sort.

Problem statement:- a[] is an array of ‘n’ elements. Within a[m],a[m+1],.....,a[p+1] the elements are rearranged in such a manner that if initially t=a[m], then after completion a[q]=t for some q between m and p-1, a[k]<=t for m<=k<q and a[k]>=t for q<k<p. In quick sort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in a[1:n] such that a[i] ≤ a[j] for all i between 1 to m and all j between m+1 and n. Thus the elements in a[1:m] and a[m+1:n] can be independently sorted. The rearrangement of the elements is accomplished by picking some element of a[] called as pivot and then reordering the other elements so that all elements appearing before pivot are less than or equal to pivot and all elements appearing after pivot are greater than pivot. This rearranging is referred to as partitioning.

Algorithm

```
Algorithm Partition(a,m,p)
// Within a[m],a[m+1],... ,a[p-1] the elements are rearranged in such a manner that if initially  t =
//a[m], then after completion a[q] = t for some q between m and p-1,a[k] ≤ t for m <k <q, and a[k] ≥ t
//for q <k <p. q is returned. Set a[p] = oo.
{  v :=a[m]; i :=m; j :=p;
   repeat {
repeat
      i:=i+1;
      until(a[i] ≥ v);
repeat
                j :=j -1;
until(a[j] ≤ u);
      if (i <j) then Interchange(a,i,j);
     }until(i ≥ j);
 a[m] :=a[j]; a[j] :=v;  return j;
 }

 Algorithm Interchange(a,i,j)
// Exchange a[i] with a[j].
 {  p :=a[i];   a[i]:=a[j];  a[j]:=p;
 }

Algorithm QuickSort(p,q)
// Sorts the elements a[p],..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 ≥ all the elements in a[1:n].
{if (p <q) then //If there are more than one element
{     // divideP into two subproblems.
                j :=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.  }   }
```

Recurrence Relation:

Best case: T(n)=2T(n/2)+O(n)

Average case: T(n)=2T(n/2)+O(n)

Worst case: T(n)=T(n−1)+O(n)

Time Complexity

Best case: O(nlogn)

Average case: O(nlogn)

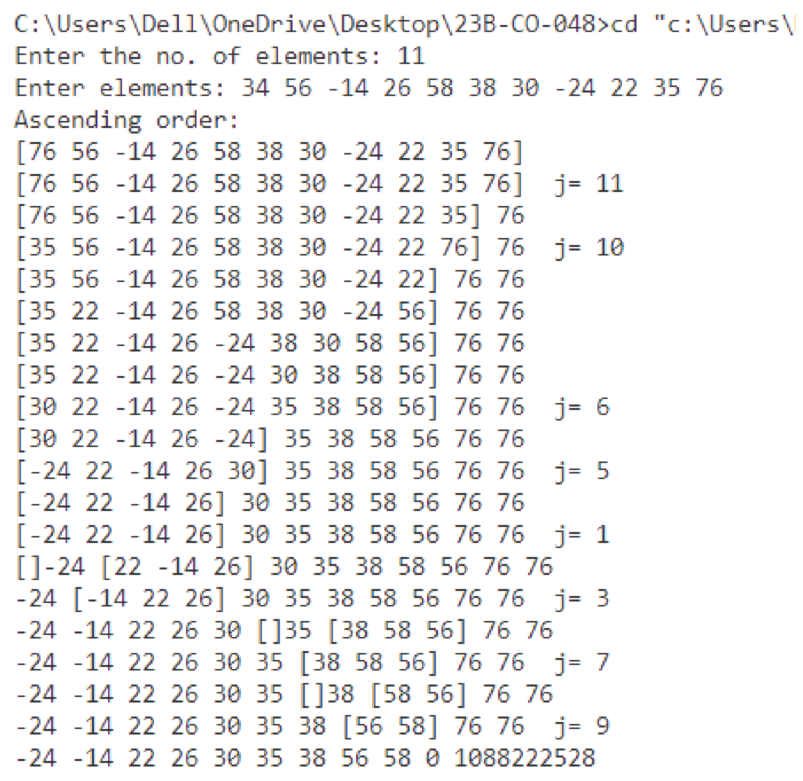
Worst case: O(n2)

Space Complexity: O(1)

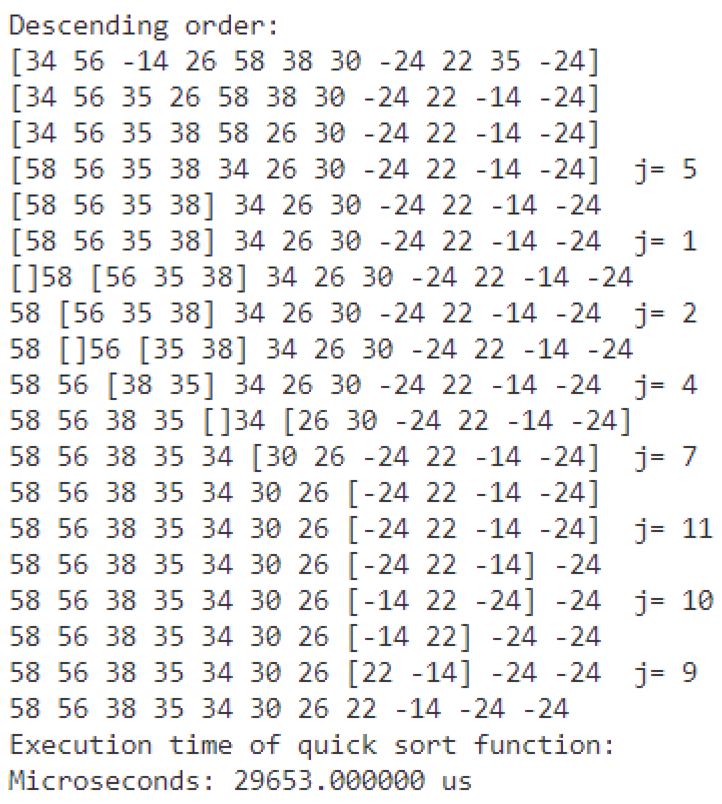
PROGRAM:

```
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#define MAX 10

int n, flag, flag2;

void print(int arr[], int low, int up);
void interchange(int arr[], int i, int j) {
    int p = arr[i];
    arr[i] = arr[j];
    arr[j] = p;
}
int partition(int arr[], int low, int up) {
    int pivot = arr[low], i = low + 1, j = up;
    while (i <= j) {
        while (arr[i] <= pivot) i++;
        while (arr[j] > pivot) j--;
        if (i < j) {
            interchange(arr, i, j);
            print(arr, low, up);
        }
    }
    arr[low] = arr[j];
    arr[j] = pivot;
    print(arr, low, up);
    printf(" j= %d", j);
    return j;
}

int partition2(int arr[], int low, int up) {
    int pivot = arr[low], i = low + 1, j = up;
    while (i <= j) {
        while (arr[i] >= pivot) i++;
        while (arr[j] < pivot) j--;
        if (i < j) {
            interchange(arr, i, j);
            print(arr, low, up);
        }
    }
    arr[low] = arr[j];
    arr[j] = pivot;
    print(arr, low, up);
    printf(" j= %d", j);
    return j;
}

void quick_sort(int arr[], int low, int up) {
    int j;
    if (low > up) {
        flag = 1;
        return;
    }
    if (low < up) {
        print(arr, low, up);
        j = partition(arr, low, up);
        quick_sort(arr, low, j - 1);
        quick_sort(arr, j + 1, up);
    }
}

void quick_sort2(int arr[], int low, int up) {
    int j;
    if (low > up) {
        flag = 1;
        return;
    }
    if (low < up) {
        print(arr, low, up);
        j = partition2(arr, low, up);
        quick_sort2(arr, low, j - 1);
        quick_sort2(arr, j + 1, up);
    }
}

void print(int arr[], int low, int up) {
    printf("\n");
    for (int i = 1; i <= n; i++) {
        if (flag == 1 && i == low - 1) {
            printf("[]");
            flag = 0;
        }
        if (i == low) printf("[");
        printf("%d", arr[i]);
        if (i != up) printf(" ");
        if (i == up) printf("] ");
    }
}
int main() {
    int arr[MAX], arr2[MAX];
    struct timeval start, end;
    double t_ms, t_us;
    long s, us;
    printf("Enter the no. of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (int i = 1; i <= n; i++) {
        scanf("%d", &arr[i]);
    }
    for (int i = 1; i <= n; i++) {
        arr2[i] = arr[i];
    }
    printf("Ascending order:");
    gettimeofday(&start, NULL);
    quick_sort(arr, 1, n);
    gettimeofday(&end, NULL);
    s = end.tv_sec - start.tv_sec;
    us = end.tv_usec - start.tv_usec;
    t_us = s * 1000000 + us;
    printf("\n");
    for (int i = 1; i <= n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n\nDescending order:");
    quick_sort2(arr2, 1, n);
    printf("\n");
    for (int i = 1; i <= n; i++) {
        printf("%d ", arr2[i]);
    }
    printf("\nExecution time of quick sort function:\n");
    printf("Microseconds: %f us\n", t_us);
    return 0;
}
```

OUTPUT:



```
C:\Users\Dell\OneDrive\Desktop\23B-CO-048>cd "c:\Users\
Enter the no. of elements: 11
Enter elements: 34 56 -14 26 58 38 30 -24 22 35 76
Ascending order:
[76 56 -14 26 58 38 30 -24 22 35 76]
[76 56 -14 26 58 38 30 -24 22 35 76]  j= 11
[76 56 -14 26 58 38 30 -24 22 35] 76
[35 56 -14 26 58 38 30 -24 22 76] 76  j= 10
[35 56 -14 26 58 38 30 -24 22] 76 76
[35 22 -14 26 58 38 30 -24 56] 76 76
[35 22 -14 26 -24 38 30 58 56] 76 76
[35 22 -14 26 -24 30 38 58 56] 76 76
[30 22 -14 26 -24 35 38 58 56] 76 76  j= 6
[30 22 -14 26 -24] 35 38 58 56 76 76
[-24 22 -14 26 30] 35 38 58 56 76 76  j= 5
[-24 22 -14 26] 30 35 38 58 56 76 76
[-24 22 -14 26] 30 35 38 58 56 76 76  j= 1
[]-24 [22 -14 26] 30 35 38 58 56 76 76
-24 [-14 22 26] 30 35 38 58 56 76 76  j= 3
-24 -14 22 26 30 []35 [38 58 56] 76 76
-24 -14 22 26 30 35 [38 58 56] 76 76  j= 7
-24 -14 22 26 30 35 []38 [58 56] 76 76
-24 -14 22 26 30 35 38 [56 58] 76 76  j= 9
-24 -14 22 26 30 35 38 56 58 0 1088222528
```



```
Descending order:
[34 56 -14 26 58 38 30 -24 22 35 -24]
[34 56 35 26 58 38 30 -24 22 -14 -24]
[34 56 35 38 58 26 30 -24 22 -14 -24]
[58 56 35 38 34 26 30 -24 22 -14 -24]  j= 5
[58 56 35 38] 34 26 30 -24 22 -14 -24
[58 56 35 38] 34 26 30 -24 22 -14 -24  j= 1
[]58 [56 35 38] 34 26 30 -24 22 -14 -24
58 [56 35 38] 34 26 30 -24 22 -14 -24  j= 2
58 []56 [35 38] 34 26 30 -24 22 -14 -24
58 56 [38 35] 34 26 30 -24 22 -14 -24  j= 4
58 56 38 35 []34 [26 30 -24 22 -14 -24]
58 56 38 35 34 [30 26 -24 22 -14 -24]  j= 7
58 56 38 35 34 30 26 [-24 22 -14 -24]
58 56 38 35 34 30 26 [-24 22 -14 -24]  j= 11
58 56 38 35 34 30 26 [-24 22 -14] -24
58 56 38 35 34 30 26 [-14 22 -24] -24  j= 10
58 56 38 35 34 30 26 [-14 22] -24 -24
58 56 38 35 34 30 26 [22 -14] -24 -24  j= 9
58 56 38 35 34 30 26 22 -14 -24 -24
Execution time of quick sort function:
Microseconds: 29653.000000 us
```

<u>Conclusion:-</u> This program demonstrates the practical application of the Quick Sort algorithm on a collection of student data. It showcases both ascending and descending order sorting with the option to visualize the sorting process at every step. Additionally, the use of partitioning and recursion highlights key concepts in sorting algorithms and provides insights into how Quick Sort efficiently handles large datasets by dividing and conquering.

EXPERIMENT NO. : 5 **DATE :**

Kth SMALLEST ELEMENT

Aim:- Write a C program to find kth smallest element in the array

Problem statement:- Given n elements a[1:n] and are required to determine the kth-smallest element.For this the partition algorithm is used to obtain an efficient solution. If the partitioning element v is positioned at a[j] then j-1 elements are less than or equal to a[j] and n-j elements are greater than or equal to a[j].

Algorithm

```
Algorithm Select1(a,n,k)
// Selects The kth-smallest element in a[1:n] and places it in the kth position of a[]. The remaining
//elements are rearranged such that a[m] ≤ a[k] for 1 ≤ m <k, and a[m] ≥ a[k] for k < m ≤ n.
{
low :=1;up :=n + 1;
a[n+ 1]:=oo;  // a[n+ 1] is set to infinity.
Repeat
{
        // Each time the loop is entered , 1≤ low ≤ k ≤ up≤ n+ 1.
        j :=Partition(a,low,up); // j is such that a[j] is the jth-smallest value in a[ ].
        if (k = j) then return;
        else if (k < j) then up :=j; // j is the new upper limit.
         else low :=j + 1; // j + 1 is the new lower limit.
}until(false);
}
```

Recurrence relation:

$$T_A(n) = \frac{1}{n} \sum_{1 \le k \le n} T_A^k(n)$$

Time Complexity: O(n)
Worst case: O(n2)

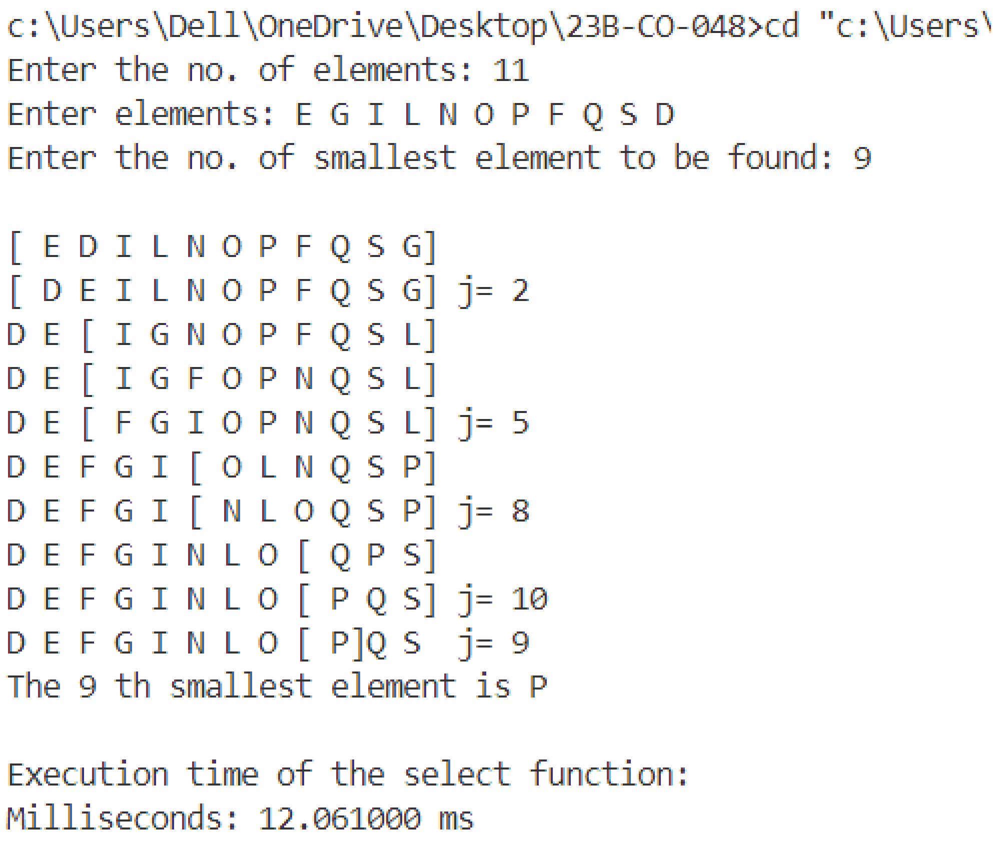
Space Complexity: O(1)

PROGRAM:

```c
#include <stdio.h>
#include <limits.h>
#include <time.h>
#include <sys/time.h>
#define MAX 15

int n;

void interchange(char arr[], int i, int j) {
   char p = arr[i];
   arr[i] = arr[j];
```

```
    arr[j] = p;
}

void print(char arr[], int low, int up) {
    printf("\n");
    for (int i = 1; i <= n; i++) {
        if (i == low) {
            printf("[ ");
        }
        printf("%c", arr[i]);
        if (i != up - 1)
            printf(" ");
        if (i == up - 1)
            printf("]");
    }
}

int partition(char arr[], int low, int up) {
    char pivot = arr[low];
    int i = low + 1, j = up;
    while (i <= j) {
        while (arr[i] <= pivot) i++;
        while (arr[j] > pivot) j--;
        if (i < j) {
            interchange(arr, i, j);
            print(arr, low, up);
        }
    }
    arr[low] = arr[j];
    arr[j] = pivot;
    print(arr, low, up);
    printf(" j= %d", j);
    return j;
}

char select(char a[], int low, int up, int k) {
    int j;
    low = 1;
    up = n + 1;
    a[n + 1] = CHAR_MAX;
    while (low < up) {
        j = partition(a, low, up);
        if (k == j)
            return a[k];
        else if (k < j)
            up = j;
        else
            low = j + 1;
    }
    return '\0';
}

int main() {
    int k;
    char arr[MAX], m;
    struct timeval start, end;
    double t_ms, t_us, t_ns;
    long s, us;
    printf("Enter the no. of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (int i = 1; i <= n; i++) {
        scanf(" %c", &arr[i]);
    }
    printf("Enter the no. of smallest element to be found: ");
    scanf("%d", &k);
    gettimeofday(&start, NULL);
    m = select(arr, 1, n, k);
    gettimeofday(&end, NULL);
    printf("\nThe %d th smallest element is %c\n", k, m);
    s = end.tv_sec - start.tv_sec;
    us = end.tv_usec - start.tv_usec;
    t_us = s * 1000000 + us;
    t_ms = t_us / 1000.0;
    t_ns = t_us * 1000.0;
    printf("\nExecution time of the select function:\n");
    printf("Milliseconds: %f ms\n", t_ms);
    return 0;
}
```

OUTPUT:



```
c:\Users\Dell\OneDrive\Desktop\23B-CO-048>cd "c:\Users\
Enter the no. of elements: 11
Enter elements: E G I L N O P F Q S D
Enter the no. of smallest element to be found: 9

[ E D I L N O P F Q S G]
[ D E I L N O P F Q S G] j= 2
D E [ I G N O P F Q S L]
D E [ I G F O P N Q S L]
D E [ F G I O P N Q S L] j= 5
D E F G I [ O L N Q S P]
D E F G I [ N L O Q S P] j= 8
D E F G I N L O [ Q P S]
D E F G I N L O [ P Q S] j= 10
D E F G I N L O [ P]Q S  j= 9
The 9 th smallest element is P

Execution time of the select function:
Milliseconds: 12.061000 ms
```

Conclusion:

This program efficiently implements the Quickselect algorithm to find the *k*-th smallest element in an array of characters. It leverages partitioning similar to Quick Sort to progressively narrow the search space and ultimately select the desired element. The inclusion of a visual representation of the partitioning process makes it easier to follow the algorithm's behavior.

EXPERIMENT NO. : 6 **DATE :**

STRASSEN'S MATRIX MULTIPLICATION

Aim:- Write a C program to multiply two matrices using Strassen's Matrix Multiplication.

Problem statement:- Let A and B be two n x n matrices. The product matrix C=AB is also an n x n matrix whose i,jth element is formed by taking the elements in the ith row of A and the jth column of B and multiplying them to get matrix C.Multiplication of two matrices requires O(N^3) running time but we can reduce this time to O(N^2.81) by using this approach. If A and B are two square matrices, then they are partitioned into four square submatrices, each submatrix have dimensions n/2 x n/2.

Algorithm

```
Algorithm Strassens(A,B,C)
//A[1:2][1:2], B[1:2][1:2] are two matrices that need to be multiplied and stored in C
{ P := (A[1][1] + A[2][2]) × (B[1][1] + B[2][2]);
Q := (A[2][1] + A[2][2]) × B[1][1];
R := A[1][1] × (B[1][2] - B[2][2]);
S := A[2][2] × (B[2][1] - B[1][1]);
T := (A[1][1] + A[1][2]) × B[2][2];
U := (A[2][1] - A[1][1]) × (B[1][1] + B[1][2]);
V := (A[1][2] - A[2][2]) × (B[2][1] + B[2][2]);
C[1][1] := P + S - T + V;
C[1][2] := R + T;
C[2][1] := Q + S;
C[2][2] := P + R - Q + U;
}
```

Recurrence relation:

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

Time Complexity:

$$\begin{aligned} T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^k T(1) \\ &\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n},\ c \text{ a constant} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \approx O(n^{2.81}) \end{aligned}$$

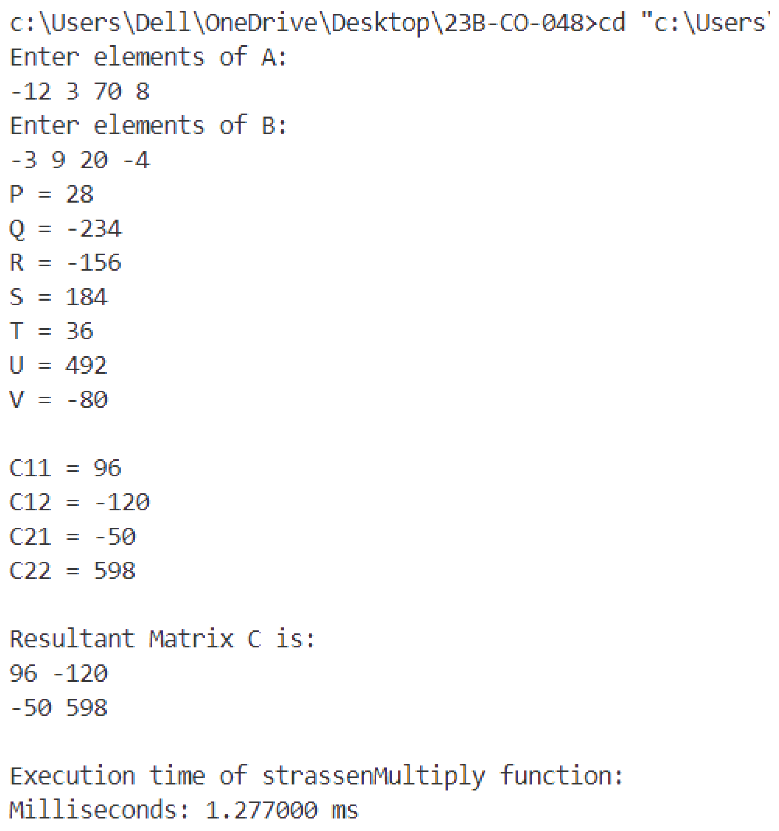
Space Complexity: O(1)

PROGRAM:

```
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

void strassenMultiply(int A[2][2], int B[2][2], int C[2][2]) {
   int P,Q,R,S,T,U,V;
   P = (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]);
   printf("P = %d\n", P);
   Q = (A[1][0] + A[1][1]) * B[0][0];
   printf("Q = %d\n", Q);
   R = A[0][0] * (B[0][1] - B[1][1]);
   printf("R = %d\n", R);
   S = A[1][1] * (B[1][0] - B[0][0]);
   printf("S = %d\n", S);
   T = (A[0][0] + A[0][1]) * B[1][1];
   printf("T = %d\n", T);
   U = (A[1][0] - A[0][0]) * (B[0][0] + B[0][1]);
   printf("U = %d\n", U);
   V = (A[0][1] - A[1][1]) * (B[1][0] + B[1][1]);
   printf("V = %d\n\n", V);
   C[0][0] = P + S - T + V;
   printf("C11 = %d\n", C[0][0]);
   C[0][1] = R + T;
   printf("C12 = %d\n", C[0][1]);
   C[1][0] = Q + S;
   printf("C21 = %d\n", C[1][0]);
   C[1][1] = P + R - Q + U;
   printf("C22 = %d\n\n", C[1][1]);
}

int main() {
   int A[2][2], B[2][2], C[2][2];
   struct timeval start, end;
   double t_ms, t_us;
   long s, us;
   printf("Enter elements of A:\n");
   for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 2; j++) {
         scanf("%d", &A[i][j]);
      }
   }
   printf("Enter elements of B:\n");
   for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 2; j++) {
         scanf("%d", &B[i][j]);
      }
   }
   gettimeofday(&start, NULL);
   strassenMultiply(A, B, C);
   gettimeofday(&end, NULL);
   s = end.tv_sec - start.tv_sec;
   us = end.tv_usec - start.tv_usec;
```

```
    t_us = s * 1000000 + us;
    t_ms = t_us / 1000.0;
    printf("Resultant Matrix C is:\n");
    printf("%d %d\n", C[0][0], C[0][1]);
    printf("%d %d\n", C[1][0], C[1][1]);
    printf("\nExecution time of strassenMultiply function:\n");
    printf("Milliseconds: %f ms\n", t_ms);
    return 0;
}
```

OUTPUT:



```
c:\Users\Dell\OneDrive\Desktop\23B-CO-048>cd "c:\Users
Enter elements of A:
-12 3 70 8
Enter elements of B:
-3 9 20 -4
P = 28
Q = -234
R = -156
S = 184
T = 36
U = 492
V = -80

C11 = 96
C12 = -120
C21 = -50
C22 = 598

Resultant Matrix C is:
96 -120
-50 598

Execution time of strassenMultiply function:
Milliseconds: 1.277000 ms
```

Conclusion:

Strassen’s algorithm significantly optimizes matrix multiplication by reducing the time complexity from $O(n^3)$ to approximately $O(n^{2.81})$, making it more efficient for larger matrices. This program effectively demonstrates the use of Strassen's method on two 2x2 matrices, showcasing its ability to minimize the number of multiplications compared to the conventional matrix multiplication approach.