**Experiment No. 1: Divide and Conquer Strategy Date:**

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

strategy

1. Merge Sort and binary search
2. Quick sort and Minmax algorithm
3. Finding Kth smallest element
4. 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 P₁,Pz.....Pk 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, Strassen’s Matrix Multiplication.

**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','I','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','I','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, l, h, x)

// Given an array a[i:l] of elements in ascending order, 1<= I <= h,

//determine whether x is present, and if so, return j such that x=a[j]; else return 0.

{ if(l >= h) then //If Small(P)

{ if(x = a[I]) then return I;

else return 0;

}

else

{

// Reduce P into a smaller subproblem.

mid:=[(I+h)/2];

if(x = a[mid]) then return mid;

else if (x<a[mid]) then return Binary Search (a, i, mid-1, x);

else return BinarySearch(a, mid+1,h,x); }

}

**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, n>1, c is a constant.

T(1) = a, where ‘a’ is a constant

Applying repeated substitution method:

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

=> log n = log2k

=> log2n = k

So we can replace k with log2n

We can solve the equation as follows:

T(n) = 2(2T(n/4) + cn/2) + cn

= 4T(n/4) + 2cn

= 4(2T(n/8) + cn/4) + 2cn

. .

= 2k T(1) + kcn

= an + cn\*log2 n

We see that 2k < n < 2k+1, then T(n) <= T(2k+1)

Therefore, T(n) = Θ(n log n)

Space Complexity: O(n)

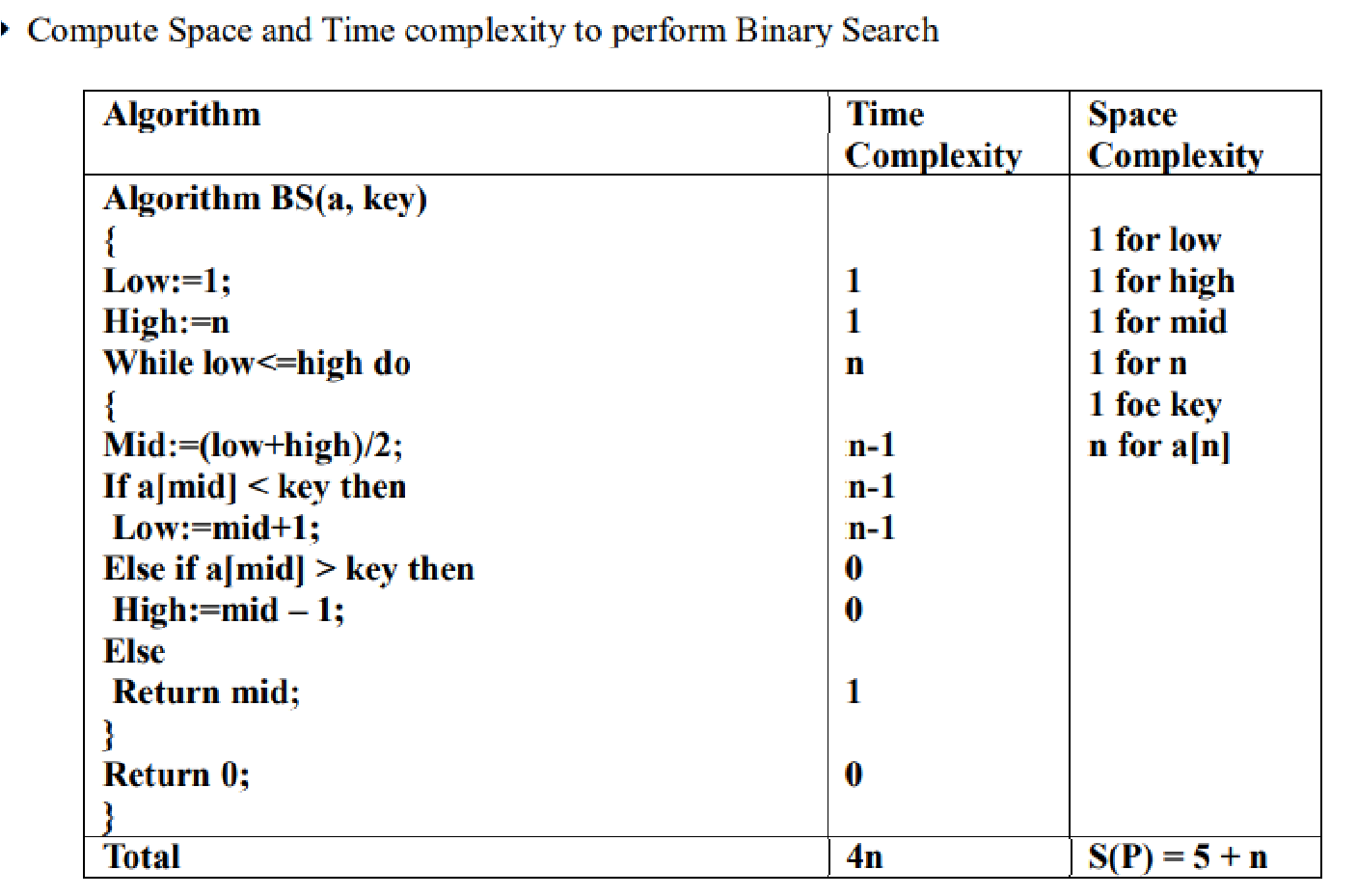
(ii) Binary Search

* + 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 :- According to Master’s theorem

a = 1, b = 2, k = 0 and p = 0  
bk = 1. So, a = bk and p > -1   
T(n) = O(nlogba logp+1n)  
T(n) = O(logn)



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

**Code:**

**Merge Sort(Char Array,Ascending):**

#include <stdio.h>

char a[100], b[100];

void merge(int low, int mid, int high)

{

int h = low, i = low, j = mid + 1;

while ((h <= mid) && (j <= high))

{

if (a[h] < a[j])

{

b[i++] = a[h++];

}

else

{

b[i++] = a[j++];

}

}

if (h > mid)

{

for (int k = j; k <= high; k++)

b[i++] = a[k];

}

else

{

for (int k = h; k <= mid; k++)

{

b[i++] = a[k];

}

}

for (int k = low; k <= high; k++)

a[k] = b[k];

}

void mergesort(int low, int high)

{

int mid;

if (low < high)

{

mid = (low + high) / 2;

mergesort(low, mid);

mergesort(mid + 1, high);

merge(low, mid, high);

}

}

int main()

{

int n;

printf("Enter the size of the array.\n");

scanf("%d", &n);

getchar();

printf("Enter the characters to be sorted.\n");

fgets(a, n + 1, stdin);

mergesort(0, n - 1);

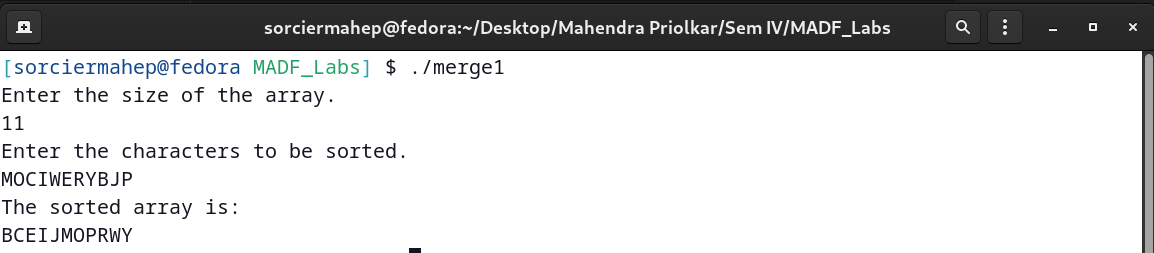
printf("The sorted array is:\n");

puts(a);

return 0;

}

**Output:**

****

**Code:**

**Merge Sort(Int Array, Descending):**

#include <stdio.h>

int a[100], b[100];

void merge(int low, int mid, int high)

{

int h = low, i = low, j = mid + 1;

while ((h <= mid) && (j <= high))

{

if (a[h] > a[j])

{

b[i++] = a[h++];

}

else

{

b[i++] = a[j++];

}

}

if (h > mid)

{

for (int k = j; k <= high; k++)

b[i++] = a[k];

}

else

{

for (int k = h; k <= mid; k++)

b[i++] = a[k];

}

for (int k = low; k <= high; k++)

{

a[k] = b[k];

}

}

void mergesort(int low, int high)

{

int mid;

if (low < high)

{

mid = (low + high) / 2;

mergesort(low, mid);

mergesort(mid + 1, high);

merge(low, mid, high);

}

}

int main()

{

int n;

printf("Enter the size of the array.\n");

scanf("%d", &n);

printf("Enter elements of the array.\n");

for (int i = 0; i < n; i++)

{

scanf("%d", &a[i]);

}

mergesort(0, n - 1);

printf("The sorted array is:\n");

for (int i = 0; i < n; i++)

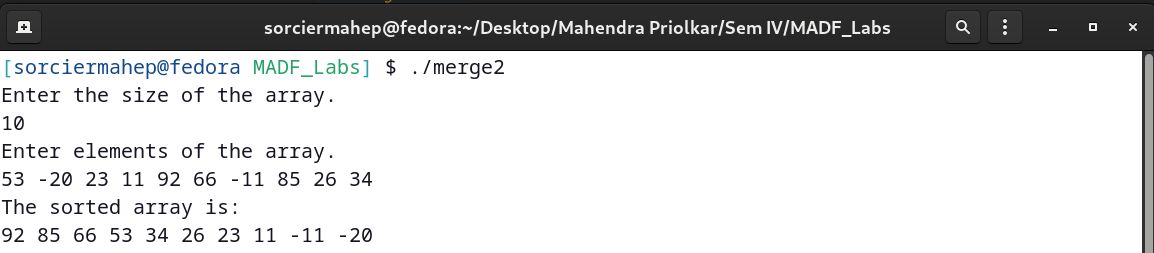
printf("%d ", a[i]);

printf("\n");

return 0;

}

**Output:**

****

**Code:**

**Binary Search:**

#include <stdio.h>

char a[100];

void bubble(int n)

{

for (int i = 0; i < n - 1; i++)

{

int exch = 0;

for (int j = 0; j < n - i - 1; j++)

{

if (a[j] > a[j + 1])

{

char temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

exch++;

}

}

if (!exch)

break;

}

}

int binsearch(char a[], int low, int high, char x)

{

if (low == high)

{

if (a[low] == x)

return low;

else

return -1;

}

else

{

int mid = (low + high) / 2;

if (a[mid] == x)

return mid;

else if (x > a[mid])

return binsearch(a, mid + 1, high, x);

else

return binsearch(a, low, mid - 1, x);

}

}

int main()

{

char x;

printf("Enter elem to be searched.\n");

scanf("%c", &x);

printf("Enter size of the array.\n");

int n;

scanf("%d", &n);

printf("Enter character array.\n");

scanf("%s", a);

bubble(n);

int pos = binsearch(a, 0, n - 1, x);

if (pos == -1)

printf("Elem not found.\n");

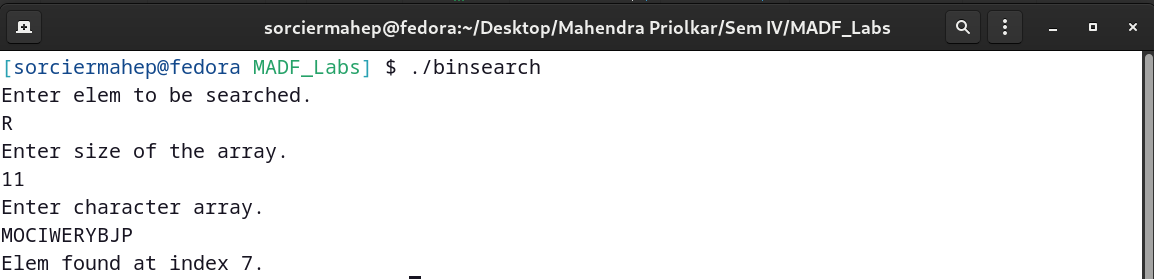
else

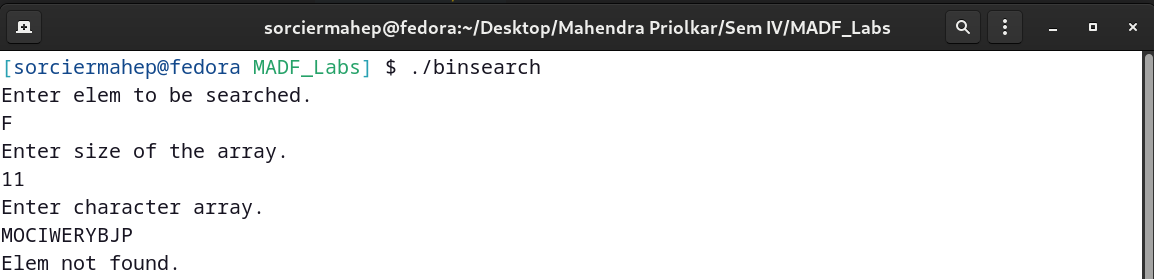
printf("Elem found at index %d.\n", pos);

return 0;

}

**Output:**

****

****

**b)Quick sort and Minmax algorithm**

**Date:**

**Problem Statement:**

i) Write a C program to Perform Quick Sort on the following list on integers to sort them in ascending order [55,11,33,23,-17,89,-11,72,43]

ii) Write a C program to Find the Min and Max element from the following list of integers using the Min Max Algorithm [55,11,33,23,-17,89,-11,72,43]

**Algorithm**

i)Quick Sort

Algorithm QuickSort(arr[], i, j)

// i -> starting index

// j -> upper index

// arr[] -> array to be sorted

{

if(i<j) then

{

p:=partition(arr, i, j);

QuickSort(arr, i, p-1);

QuickSort(arr, p+1, j);

}

}

Algorithm partition(arr, l, h)

{

pivot:=arr[h];

i:=l;

for j:=low to h-1 do

{

if(arr[j]<pivot) then

{

temp:=arr[j];

arr[j]:=arr[i];

arr[i]:=temp;

i:=i+1;

}

}

temp:=arr[j];

arr[j]:=arr[i];

arr[i]:=temp;

return i;

}

ii)Minmax

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;

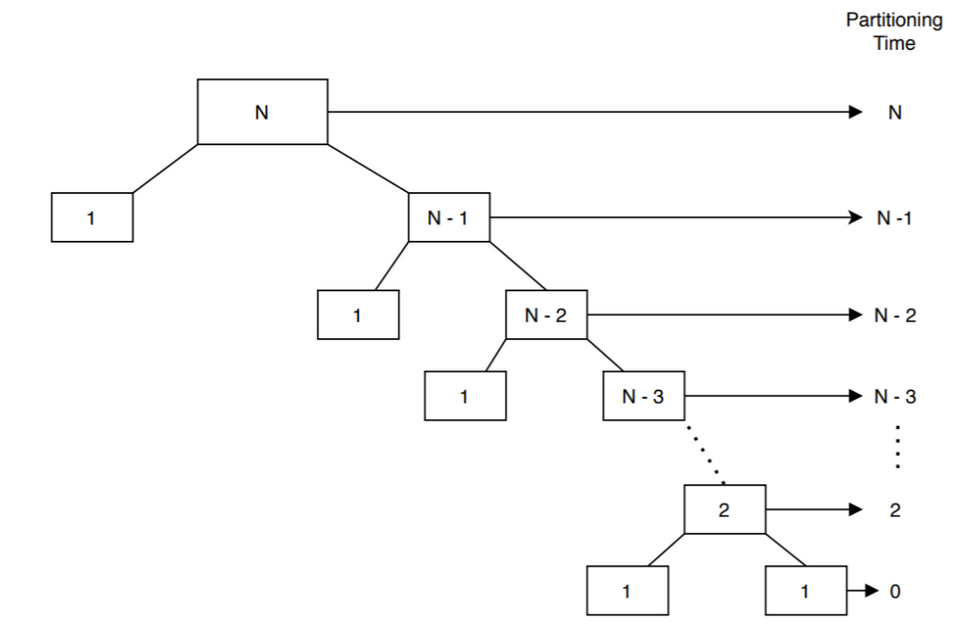
}

}

**Time and Space Complexity**

i)Quicksort

Worst Case:



In the worst case, after the first partition, one array will have  element and the other one will have  elements. Let’s say  denotes the time complexity to sort  elements in the worst case:

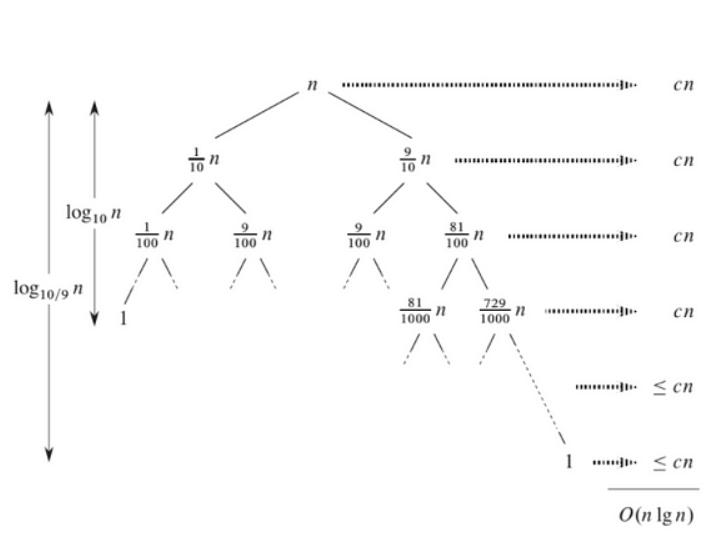
Again for the base case when   and , we don’t need to sort anything. Hence, the sorting time is  and 

Now, we’re ready to solve the recurrence relation we derived earlier:

, , , ,  , , , 

As a result, 

Average Case:



L(n) = 2U (n / 2) + Θ(n) lucky

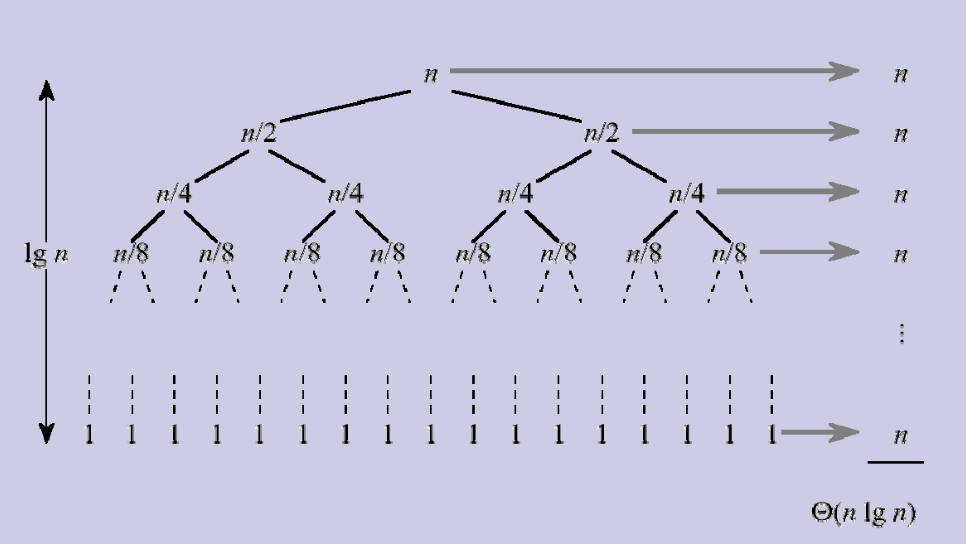
U (n) = L(n − 1) + Θ(n) unlucky

We consequently get

L(n) = 2( L(n / 2 − 1) + Θ(n / 2)) + Θ(n)

= 2 L(n / 2 − 1) + Θ (n)

= Θ(n log n)



The equal partition splits the array evenly.

The recurrence relation:-

T (n) = 2T (n / 2) + Θ(n)

a=2,b=2,d=1

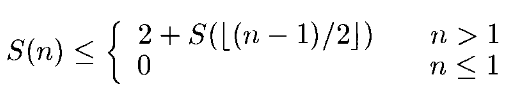
a=b^d

Using Master’s Theorem,

T(n)= Θ(n^logba\*log n)

T(n)= Θ(n\*log n)

Let S(n) be the maximum stack space needed. Then it follows that



S((n-1)/2) <= 2+ S((n-3)/4)

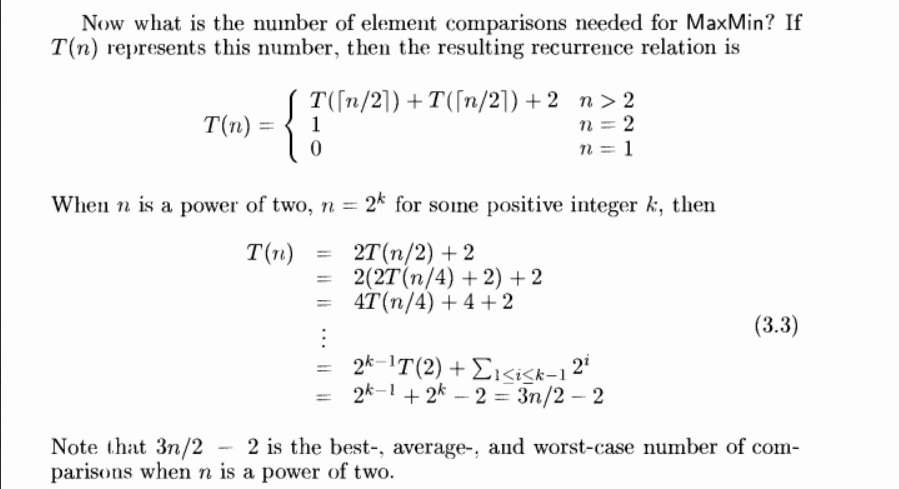
.

.

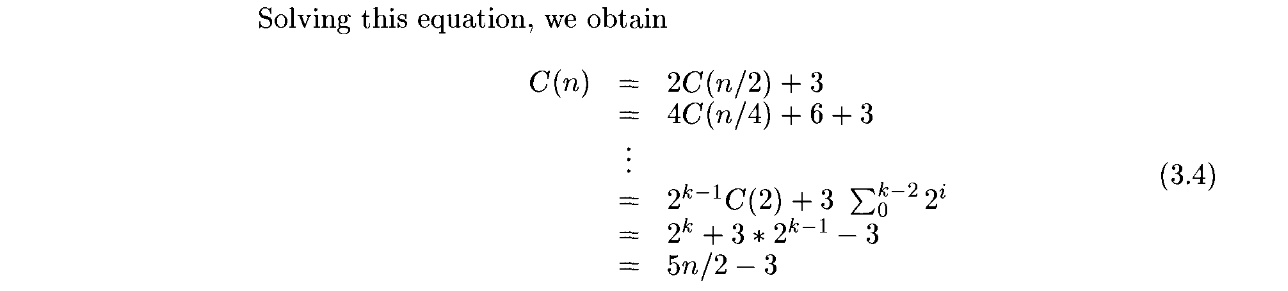
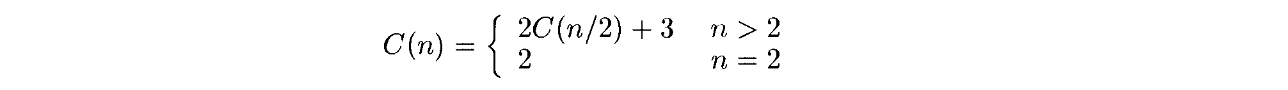
S(0) <= 2 log n

Thus it is less than 2 log n. Therefore space complexity will be O(log n).

ii) Minmax



If element comparisons have same cost as I,j comparisons:



Therefore the complexities in all the three cases will be O(n) when n is a power of 2.

There will be log2+ 1 levels of recursion and we need to save seven values

for each recursive call. Thus the space complexity is O(log n).

**Code**

**i)Quick Sort**

#include <stdio.h>

int a[100], n;

void swap(int \*a, int \*b)

{

int temp = \*a; \*a = \*b; \*b = temp;

}

void disp\_array(int a[], int n)

{

for (int i = 0; i < n; i++)

printf("%d ", a[i]);

printf("\n");

}

int Partition(int m, int p)

{

int v = a[m], i = m, j = p;

while (i < j)

{

do

{

i++;

} while (a[i] <= v);

do

{

j--;

} while (a[j] > v);

if (i < j)

{

swap(&a[i], &a[j]);

}

}

swap(&a[m], &a[j]);

return j;

}

void quicksort(int p, int q)

{

if (p < q)

{

disp\_array(a, n);

int j = Partition(p, q + 1);

printf("j=%d\n", j + 1);

quicksort(p, j);

quicksort(j + 1, q);

}

}

int main()

{

printf("Enter the size of the array.\n");

scanf("%d", &n);

printf("Enter the elements.\n");

for (int i = 0; i < n; i++)

scanf("%d", &a[i]);

a[n] = \_\_INT32\_MAX\_\_;

printf("The quicksort steps are:\n");

quicksort(0, n);

printf("The sorted elements are:\n");

for (int i = 0; i < n; i++)

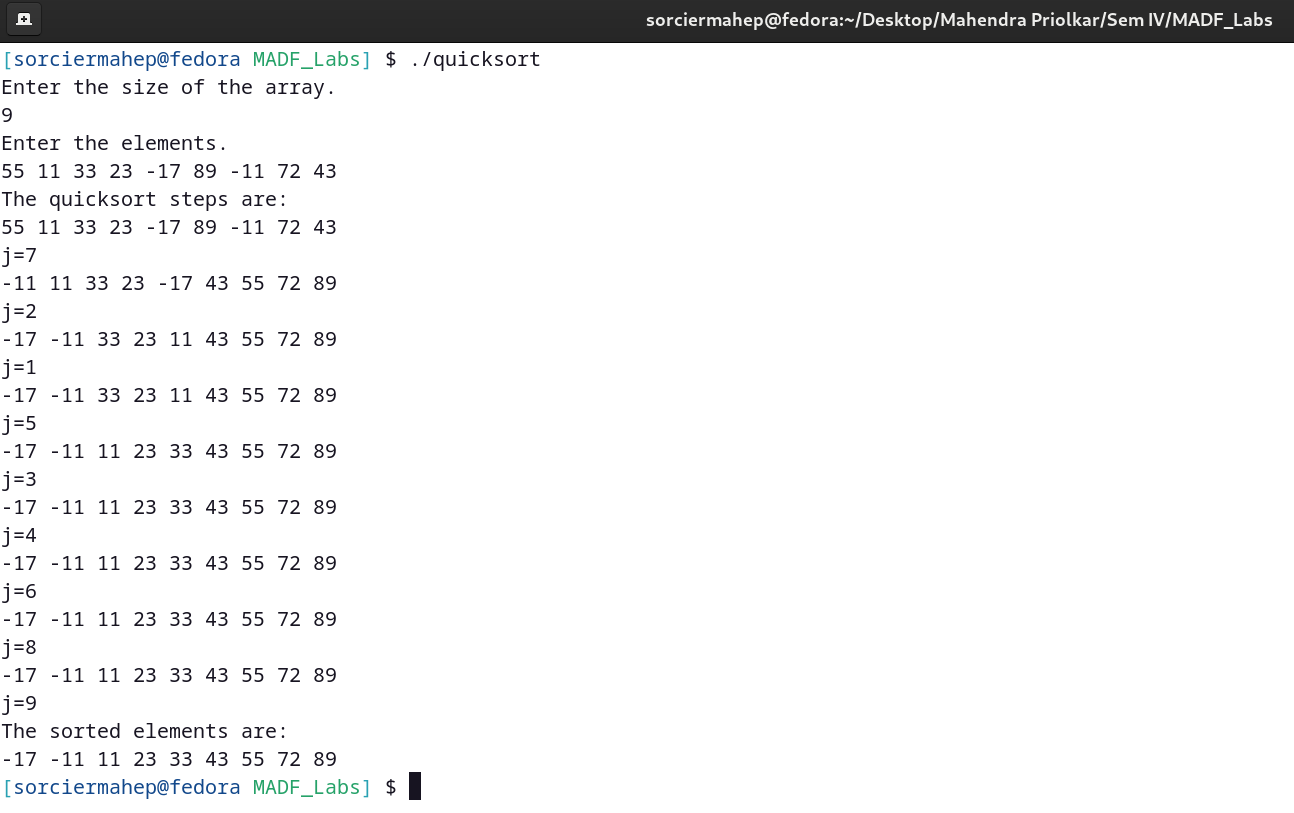
printf("%d ", a[i]);

printf("\n");

return 0;

}

**Output**



**Code**

**ii)Minmax**

#include <stdio.h>

int a[100];

void minmax(int i, int j, int \*max, int \*min)

{

if (i == j)

\*max = \*min = a[i];

else if (i == j - 1)

{

if (a[i] > a[j])

{

\*max = a[i];

\*min = a[j];

}

else

{

\*max = a[j];

\*min = a[i];

}

}

else

{

int mid = (i + j) / 2;

int min1, max1;

minmax(i, mid, max, min);

minmax(mid + 1, j, &max1, &min1);

if (max1 > \*max)

\*max = max1;

if (min1 < \*min)

\*min = min1;

}

}

int main()

{

int n, min, max;

printf("Enter the size of the array.\n");

scanf("%d", &n);

printf("Enter the elements.\n");

for (int i = 0; i < n; i++)

scanf("%d", &a[i]);

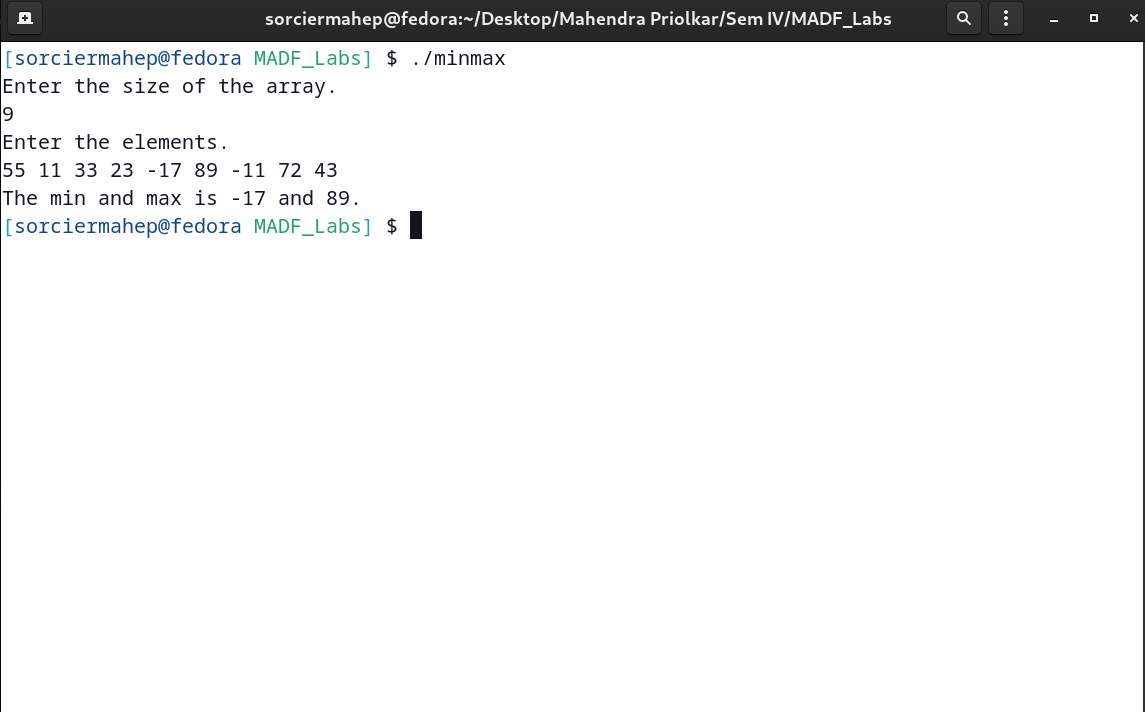
minmax(0, n - 1, &max, &min);

printf("The min and max is %d and %d.\n", min, max);

return 0;

}

**Output:**

****

**c)Finding kth smallest element**

**Date:**

**Problem Statement:**

i) Write a C program to find the kth smallest element using the kth smallest algorithm and find the 6th and the 10th element from the following list of integers

[23,92,72,87,65,45,68,79,89,17,08]

ii) Write a C program to find the kth smallest element using the kth smallest algorithm find the 6th and the 2nd element from the following list of characters

[‘H’, ‘Z’, ‘Q’, ‘U’, ‘L’, ‘J’, ‘O’, ‘T’, ‘W’, ‘V’, ‘A’]

**Algorithm**

Algorithm ksmall(a[], l, r, k)

// selects the kth smallest element in a[1:n] and places it in the kth position

{

if (r<k) then return;

repeat

{

p=partition(a[], 1, r);

if (p=k) then return;

else

{

if(p>k) then

r:=p-1;

else

l:=p+1;

}

} until (l<=r);

}

Algorithm partition(arr, l, h)

{

pivot:=arr[h];

i:=l;

for j:=low to h-1 do

{

if(arr[j]<pivot) then

{

temp:=arr[j];

arr[j]:=arr[i];

arr[i]:=temp;

i:=i+1;

}

}

temp:=arr[j];

arr[j]:=arr[i];

arr[i]:=temp;

return i;

}

**Time and Space Complexity**

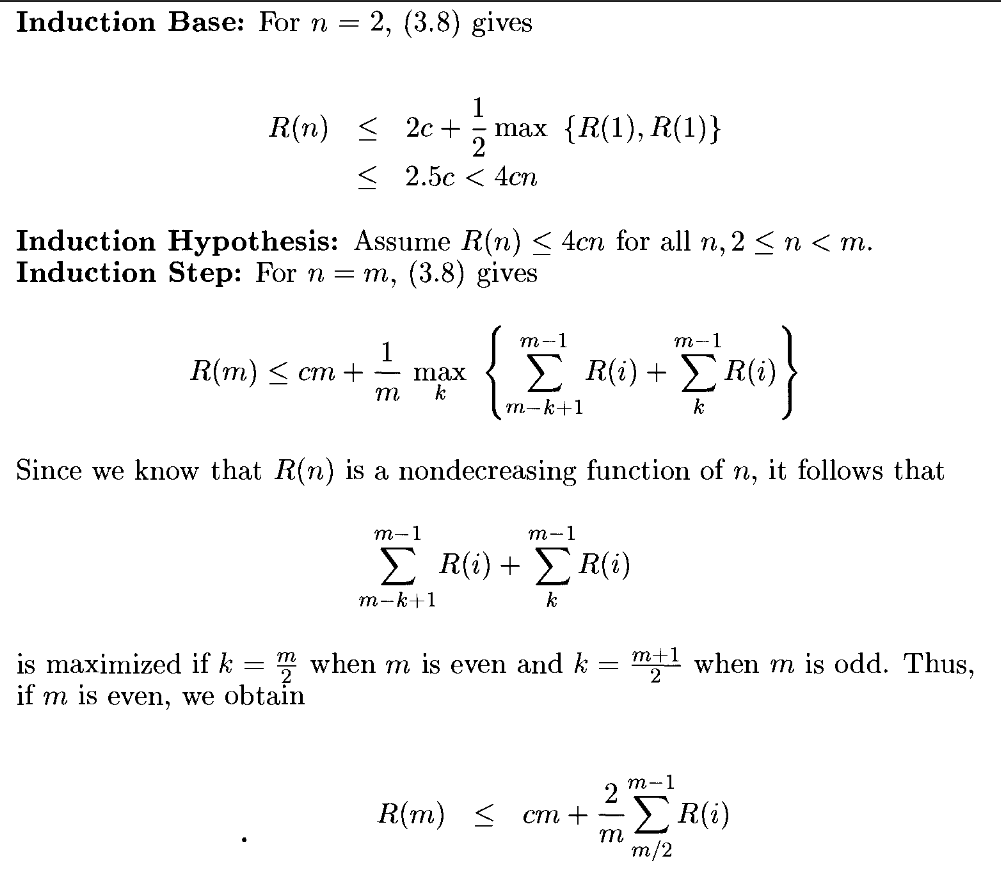
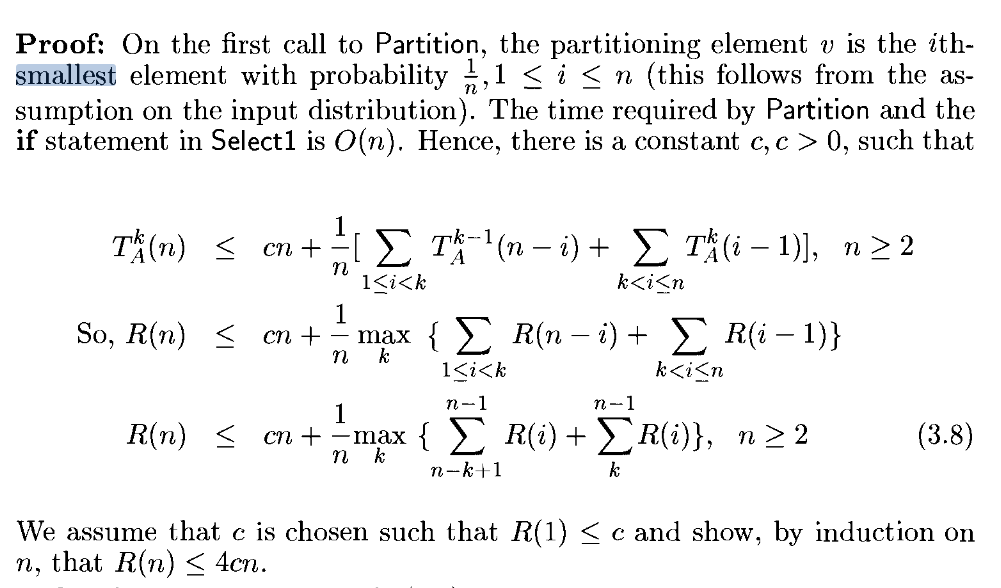
On each successive call to Partition,

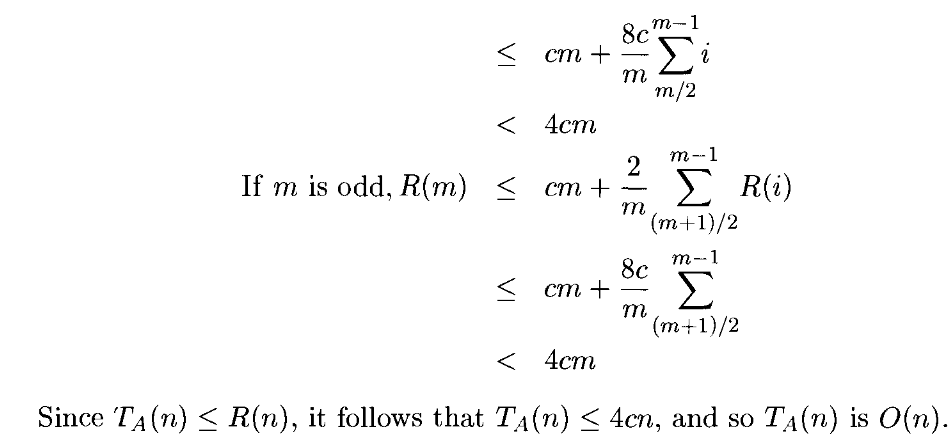
either m increases by at least one or j decreases by at least one. Initially

m = I and j = n + 1. Hence, at most n calls to Partition can be made.

Thus, the worst-case complexity of Selectl is O(n^2).

Average Case:





The space needed by this algorithm is O(n) i.e. Recursive depth(n)\* Size of one call(O(1))

**Code:**

**i)kth smallest(int array):**

#include <stdio.h>

int a[100];

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int Partition(int m, int p)

{

int v = a[m], i = m, j = p;

while (i < j)

{

do

{

i++;

} while (a[i] <= v);

do

{

j--;

} while (a[j] > v);

if (i < j)

{

swap(&a[i], &a[j]);

}

}

swap(&a[m], &a[j]);

return j;

}

int kthsmallest(int n, int k)

{

int low = 0;

int up = n;

do

{

int j = Partition(low, up);

if (k == j)

return a[j];

else if (k > j)

low = j + 1;

else

up = j;

} while (1);

}

int main()

{

int n;

printf("Enter the size of the array.\n");

scanf("%d", &n);

printf("Enter the elements.\n");

for (int i = 0; i < n; i++)

scanf("%d", &a[i]);

a[n] = \_\_INT32\_MAX\_\_;

int sel;

printf("Enter the order of smallest elem you want to display.\n");

scanf("%d", &sel);

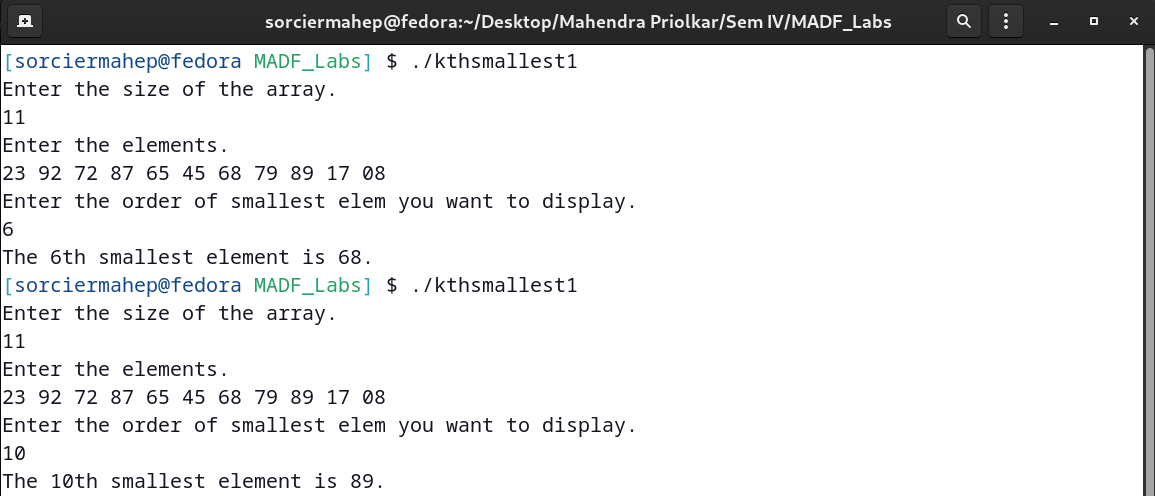
int num = kthsmallest(n, sel - 1);

printf("The %dth smallest element is %d.\n", sel, num);

return 0;

}

**Output:**

**Code:**

**kth smallest(char array):**

#include <stdio.h>

char a[100];

void swap(char \*a, char \*b)

{

char temp = \*a;

\*a = \*b;

\*b = temp;

}

int Partition(int m, int p)

{

char v = a[m], i = m, j = p;

while (i < j)

{

do

{

i++;

} while (a[i] <= v);

do

{

j--;

} while (a[j] > v);

if (i < j)

{

swap(&a[i], &a[j]);

}

}

swap(&a[m], &a[j]);

return j;

}

char kthsmallest(int n, int k)

{

int low = 0;

int up = n;

do

{

int j = Partition(low, up);

if (k == j)

return a[j];

else if (k > j)

low = j + 1;

else

up = j;

} while (1);

}

int main()

{

int n;

printf("Enter the size of the array.\n");

scanf("%d", &n);

getchar();

printf("Enter the elements.\n");

fgets(a, n + 1, stdin);

a[n] = '~';

a[n + 1] = '\0';

int sel;

printf("Enter the order of smallest elem you want to display.\n");

scanf("%d", &sel);

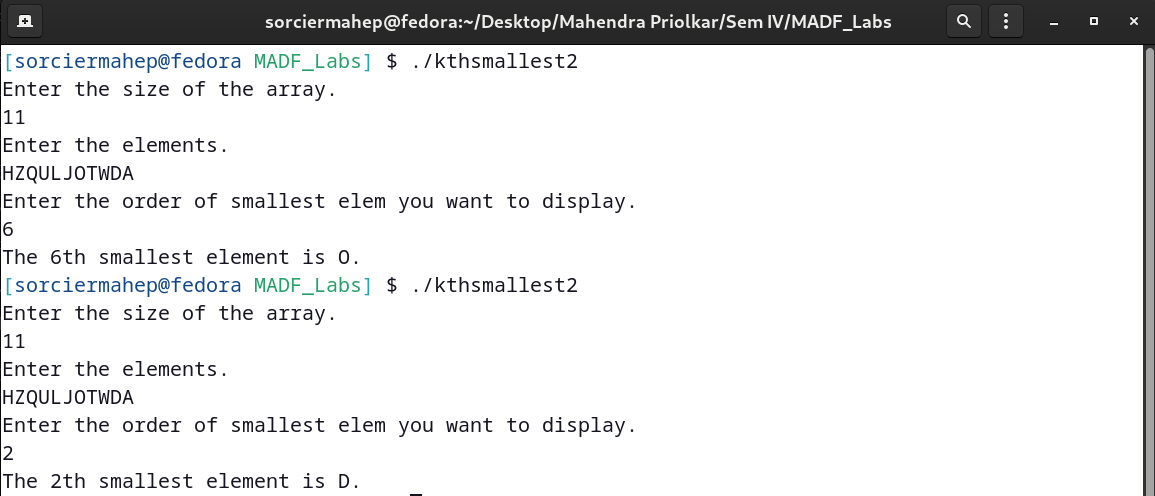
char ch = kthsmallest(n, sel - 1);

printf("The %dth smallest element is %c.\n", sel, ch);

return 0;

}

**Output:**

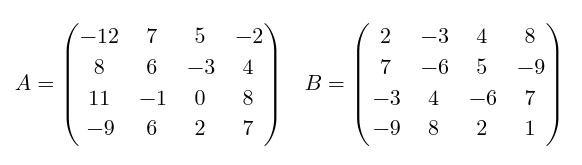
****

**d)Strassen’s Matrix Multiplication**

**Date:**

**Problem Statement:**

i) Write a C program to multiply two 4x4 matrices using Strassen’s matrix multiplication



**Algorithm**

Algorithm StrassenMult(n,a,b,c)

{

//nxn is the size of the matrix

// c = a \* b is a conventional resultant matrix

Partition a into four sub matrices a11, a12, a21, a22.

Partition b into four sub matrices b11, b12, b21, b22.

//Consider c to have four parts c11,c12,c21,c22.

if(n=2)

{

//p,q,r,s,t,u,v are values.

p=(a11+a22)\*(b11+b22);

q= (a21 + a22)\*(b11);

r= (a11)\*(b12 – b22);

s= (a22)\*(b21 – b11);

t= (a11 + a12)\*(b22);

u= ( a21 – a11)\*(b11 + b22);

v= (a12 – a22)\*(b21 + b22);

c11=p+s-t+v;

c12=r+t;

c21=q+s;

c22=p+r-q+u;

Combine c11,c12,c21,c22 into resultant matrix c.

}

else

{

Strassen ( n/2, a11 + a22, b11 + b22, p);

Strassen ( n/2, a21 + a22, b11, q);

Strassen ( n/2, a11, b12 – b22, r);

Strassen ( n/2, a22, b21 – b11, s);

Strassen ( n/2, a11 + a12, b22, t);

Strassen (n/2, a21 – a11, b11 + b22, u);

Strassen (n/2, a12 – a22, b21 + b22, v);

c11=p+s-t+v;

c12=r+t;

c21=q+s;

c22=p+r-q+u;

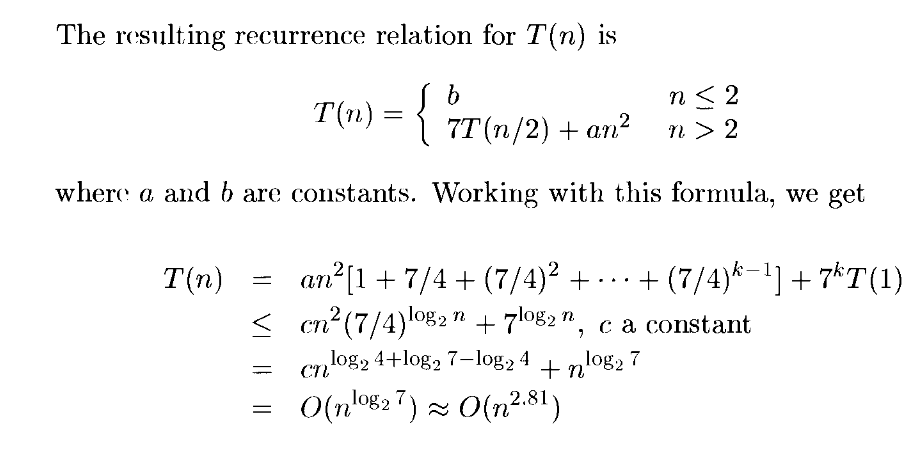
Combine c11,c12,c21,c22 into resultant matrix c.

}

return (c);

}

**Time and Space Complexity**

A new matrix is used to store the result of the multiplication. So, the space complexity is O(N^2).

**Code:**

#include <stdio.h>

#include <stdlib.h>

int \*\*matralloc(int \*\*a, int rows, int columns)

{

a = (int \*\*)malloc(rows \* sizeof(int \*));

for (int i = 0; i < rows; i++)

\*(a + i) = (int \*)malloc(columns \* sizeof(int));

return a;

}

void accept(int \*\*a, int rows, int columns)

{

for (int i = 0; i < rows; i++)

{

for (int j = 0; j < columns; j++)

scanf("%d", (\*(a + i) + j));

}

}

void display(int \*\*a, int rows, int columns)

{

for (int i = 0; i < rows; i++)

{

for (int j = 0; j < columns; j++)

{

printf("%5d ", \*(\*(a + i) + j));

}

printf("\n");

}

printf("\n");

}

int \*\*sum(int \*\*a, int \*\*b, int \*\*c, int r1, int c1, int r2, int c2)

{

for (int i = r1, l = 0, p = r2; i < r1 + 2; i++, l++, p++)

{

for (int j = c1, m = 0, q = c2; j < c1 + 2; j++, m++, q++)

\*(\*(c + l) + m) = \*(\*(a + i) + j) + \*(\*(b + p) + q);

}

return c;

}

int \*\*sub(int \*\*a, int \*\*b, int \*\*c, int r1, int c1, int r2, int c2)

{

for (int i = r1, l = 0, p = r2; i < r1 + 2; i++, l++, p++)

{

for (int j = c1, m = 0, q = c2; j < c1 + 2; j++, m++, q++)

\*(\*(c + l) + m) = \*(\*(a + i) + j) - \*(\*(b + p) + q);

}

return c;

}

int \*\*part(int \*\*a, int \*\*b, int r1, int c1)

{

for (int i = r1, l = 0; i < r1 + 2; i++, l++)

{

for (int j = c1, m = 0; j < c1 + 2; j++, m++)

\*(\*(b + l) + m) = \*(\*(a + i) + j);

}

return b;

}

void strassen(int \*\*a, int \*\*b, int \*\*c, int rows, int columns)

{

if (rows == 2 && columns == 2)

{

int p, q, r, s, t, u, v;

p = (a[0][0] + a[1][1]) \* (b[0][0] + b[1][1]);

q = (a[1][0] + a[1][1]) \* (b[0][0]);

r = (a[0][0]) \* (b[0][1] - b[1][1]);

s = (a[1][1]) \* (b[1][0] - b[0][0]);

t = (a[0][0] + a[0][1]) \* (b[1][1]);

u = (a[1][0] - a[0][0]) \* (b[0][0] + b[0][1]);

v = (a[0][1] - a[1][1]) \* (b[1][0] + b[1][1]);

c[0][0] = p + s - t + v;

c[0][1] = r + t;

c[1][0] = q + s;

c[1][1] = p + r - q + u;

}

else

{

int \*\*temp1 = NULL, \*\*temp2 = NULL;

temp1 = matralloc(temp1, 2, 2);

temp2 = matralloc(temp2, 2, 2);

int \*\*p = NULL, \*\*q = NULL, \*\*r = NULL, \*\*s = NULL, \*\*t = NULL, \*\*u = NULL, \*\*v = NULL;

p = matralloc(p, 2, 2);

q = matralloc(q, 2, 2);

r = matralloc(r, 2, 2);

s = matralloc(s, 2, 2);

t = matralloc(t, 2, 2);

u = matralloc(u, 2, 2);

v = matralloc(v, 2, 2);

strassen((temp1 = sum(a, a, temp1, 0, 0, 2, 2)), (temp2 = sum(b, b, temp2, 0, 0, 2, 2)), p, 2, 2);

strassen((temp1 = sum(a, a, temp1, 2, 0, 2, 2)), (temp2 = part(b, temp2, 0, 0)), q, 2, 2);

strassen((temp1 = part(a, temp1, 0, 0)), (temp2 = sub(b, b, temp2, 0, 2, 2, 2)), r, 2, 2);

strassen((temp1 = part(a, temp1, 2, 2)), (temp2 = sub(b, b, temp2, 2, 0, 0, 0)), s, 2, 2);

strassen((temp1 = sum(a, a, temp1, 0, 0, 0, 2)), (temp2 = part(b, temp2, 2, 2)), t, 2, 2);

strassen((temp1 = sub(a, a, temp1, 2, 0, 0, 0)), (temp2 = sum(b, b, temp2, 0, 0, 0, 2)), u, 2, 2);

strassen((temp1 = sub(a, a, temp1, 0, 2, 2, 2)), (temp2 = sum(b, b, temp2, 2, 0, 2, 2)), v, 2, 2);

int \*\*c00 = NULL, \*\*c01 = NULL, \*\*c10 = NULL, \*\*c11 = NULL;

c00 = matralloc(c00, 2, 2);

c01 = matralloc(c01, 2, 2);

c10 = matralloc(c10, 2, 2);

c11 = matralloc(c11, 2, 2);

c00 = sum(temp2 = sub(temp1 = (sum(p, s, temp1, 0, 0, 0, 0)), t, temp2, 0, 0, 0, 0), v, c00, 0, 0, 0, 0);

c01 = sum(r, t, c01, 0, 0, 0, 0);

c10 = sum(q, s, c10, 0, 0, 0, 0);

c11 = sum(temp2 = sub(temp1 = (sum(p, r, temp1, 0, 0, 0, 0)), q, temp2, 0, 0, 0, 0), u, c11, 0, 0, 0, 0);

for (int i = 0; i < 2; i++)

{

for (int j = 0; j < 2; j++)

{

\*(\*(c + i) + j) = \*(\*(c00 + i) + j);

\*(\*(c + i) + j + 2) = \*(\*(c01 + i) + j);

\*(\*(c + i + 2) + j) = \*(\*(c10 + i) + j);

\*(\*(c + i + 2) + j + 2) = \*(\*(c11 + i) + j);

}

}

printf("P:\n");

display(p, 2, 2);

printf("Q:\n");

display(q, 2, 2);

printf("R:\n");

display(r, 2, 2);

printf("S:\n");

display(s, 2, 2);

printf("T:\n");

display(t, 2, 2);

printf("U:\n");

display(u, 2, 2);

printf("V:\n");

display(v, 2, 2);

}

}

int main()

{

int \*\*a = NULL, \*\*b = NULL, \*\*c = NULL;

a = matralloc(a, 4, 4);

b = matralloc(b, 4, 4);

c = matralloc(c, 4, 4);

printf("Enter the elements of m1.\n");

accept(a, 4, 4);

printf("Enter the elements of m2.\n");

accept(b, 4, 4);

strassen(a, b, c, 4, 4);

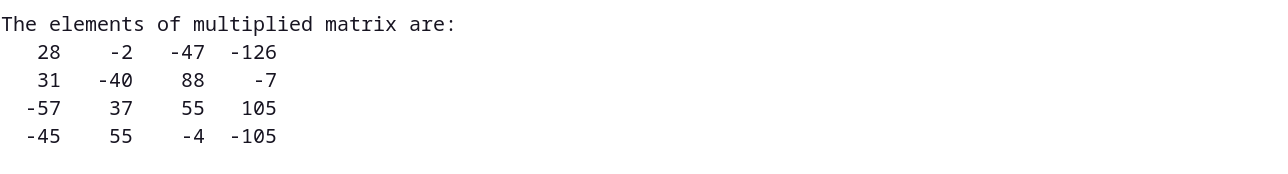
printf("The elements of multiplied matrix are:\n");

display(c, 4, 4);

return 0;

}

**Output**

****

**CONCLUSION:**

Divide and Conquer strategy was studied. The programs for (a) Merge 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.