**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.

1. **Merge Sort and Binary Search algorithm**

**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, 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, 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 : o According to master’s theorem

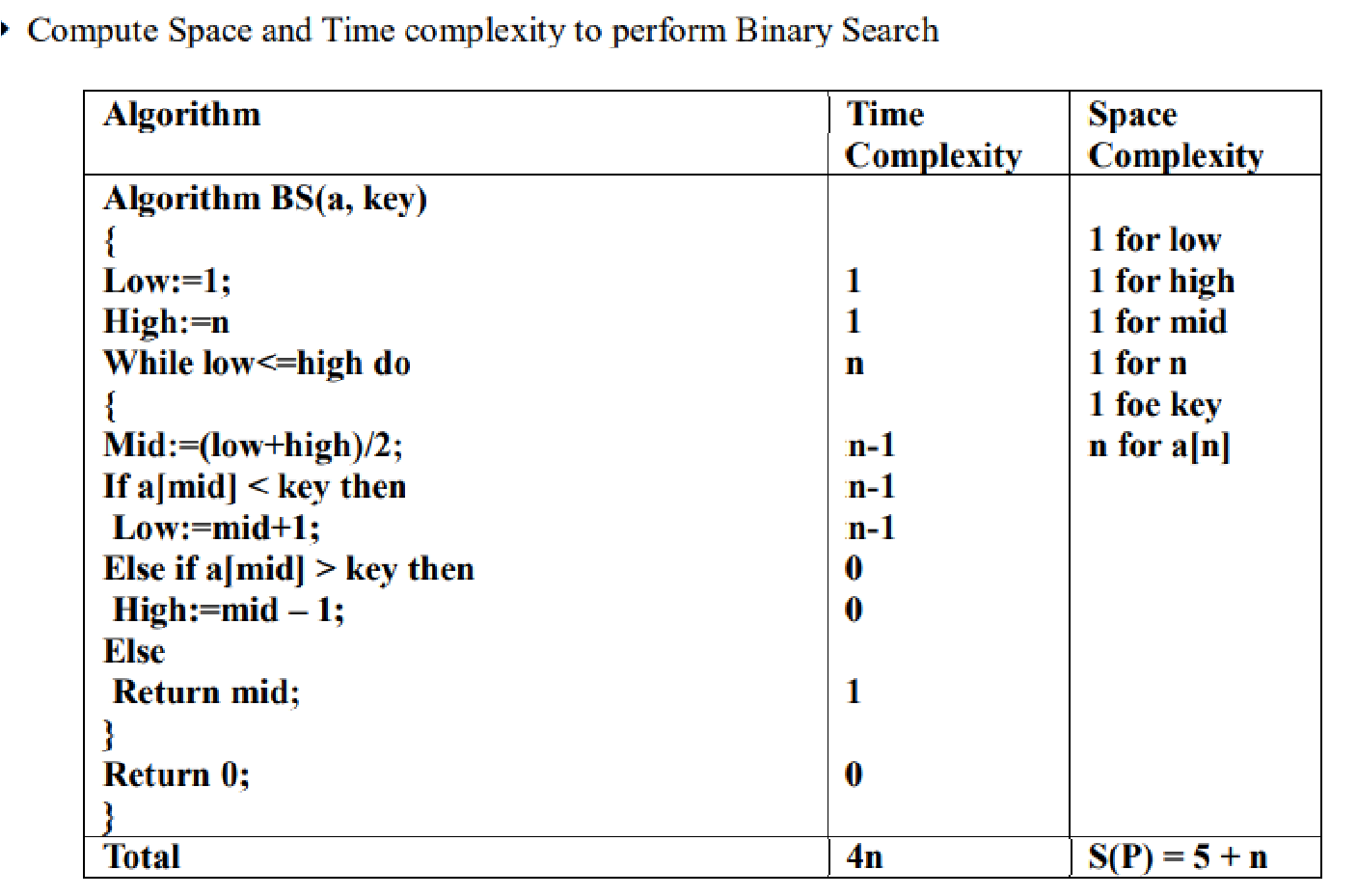
**a=1,b=2 n^(log(base b)a)=n^(0) =1 T(n) = n^(0)\*U(n)**

**U(n)->h(n)=F(n)/(n^0)**

**= c/(n^0) =>c=>r=0**

**=>i=0**

**(Log(n))^i+1/(i+1) =(Log(n))**



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

**Code**

**Program 1:**

#include <stdio.h>  
  
char a[50];  
  
void merge(int low,int mid,int high){  
 char b[50];  
 int i=low,j=mid+1,k=low;  
 while(i<=mid && j<=high){  
 if(a[i]<a[j])  
 b[k++]=a[i++];  
 else  
 b[k++]=a[j++];  
 }  
 while(i<=mid)  
 b[k++]=a[i++];  
 while(j<=high)  
 b[k++]=a[j++];  
 for(i=low;i<=high;i++)  
 a[i]=b[i];  
}  
  
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 binarysearch(int low, int high, char c) {  
 if(low==high) {  
 if(a[low]==c)  
 return low;  
 else   
 return 0;  
 }  
 int mid=(low+high)/2;  
 if(a[mid]==c)  
 return mid;  
 else if (c>a[mid])  
 return binarysearch(mid,high,c);  
 else  
 return binarysearch(low,mid-1,c);   
}  
  
int main(){  
   
 int n,s,p,i;  
 printf("enter the no of elements\n");  
 scanf("%d",&n);  
 printf("enter the elements\n");  
 for(i=0;i<n;i++){  
 scanf(" %c",&a[i]);  
 }  
 mergesort(0,n-1);  
 printf("\nSorted array is:\n");  
 for(i=0;i<n;i++){  
 printf("%c ",a[i]);  
 }  
}

**Output**

enter the no of elements

11

enter the elements

M O C I W E R Y B J P

Sorted array is:

B C E I J M O P R W Y

--------------------------------

Process exited after 15.77 seconds with return value 11

Press any key to continue . . .

**Program 2:**

#include <stdio.h>  
  
int a[50];  
  
void merge(int low,int mid,int high){  
 int b[50];  
 int i=low,j=mid+1,k=low;  
 while(i<=mid && j<=high){  
 if(a[i]>a[j])  
 b[k++]=a[i++];  
 else  
 b[k++]=a[j++];  
 }  
 while(i<=mid)  
 b[k++]=a[i++];  
 while(j<=high)  
 b[k++]=a[j++];  
 for(i=low;i<=high;i++)  
 a[i]=b[i];  
}  
  
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,s,p,i;  
 printf("enter the no of elements\n");  
 scanf("%d",&n);  
 printf("enter the elements\n");  
 for(i=0;i<n;i++){  
 scanf("%d",&a[i]);  
 }  
 mergesort(0,n-1);  
 printf("Sorted array is:\n");  
 for(i=0;i<n;i++){  
 printf("%d ",a[i]);  
 }  
}

**Output:**

enter the no of elements

10

enter the elements

53 -20 23 11 92 66 -11 85 26 34

Sorted array is:

92 85 66 53 34 26 23 11 -11 -20

--------------------------------

Process exited after 21.38 seconds with return value 10

Press any key to continue . . .

**Program 3:**

#include <stdio.h>

#include <stdlib.h>

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

#define ch(x) scanf(" %c", &x);

void charmerge(char arr[], int left, int mid, int right)

{

int i, j, k;

int n1 = mid - left + 1;

int n2 = right - mid;

char L[n1], R[n2];

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

L[i] = arr[left + i];

for (j = 0; j < n2; j++)

R[j] = arr[mid + 1 + j];

i = 0;

j = 0;

k = left;

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

void charmergesort(char arr[], int left, int right)

{

if (left < right)

{

int mid = left + (right - left) / 2;

charmergesort(arr, left, mid);

charmergesort(arr, mid + 1, right);

charmerge(arr, left, mid, right);

}

}

int bs(char arr[], char item, int low, int up)

{

int mid;

if (low > up)

return -1;

mid = (low + up) / 2;

if (item < arr[mid])

bs(arr, item, low, mid - 1);

else if (item > arr[mid])

bs(arr, item, mid + 1, up);

else

return mid;

}

int main()

{

char ch;

int charsize,i;

printf("Enter the size of the array: ");

sint(charsize);

char charr[charsize];

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

for (i = 0; i < charsize; i++)

ch(charr[i]);

charmergesort(charr, 0, charsize - 1);

while (1)

{

int c;

printf("Enter 1 to search\n2 to exit\n");

sint(c);

if (c == 2)

exit(1);

else

{

printf("Enter the character to be searched: ");

ch(ch);

if (bs(charr, ch, 0, charsize - 1) == -1)

printf("%c not found\n", ch);

else

printf("%c found\n", ch);

}

}

return 0;

}

**Output:**

Enter the size of the array: 11

Enter the character array

M O C I W E R Y B J P

Enter 1 to search

2 to exit

1

Enter the character to be searched: R

R found

Enter 1 to search

2 to exit

1

Enter the character to be searched: F

F not found

Enter 1 to search

2 to exit

2

--------------------------------

Process exited after 33.56 seconds with return value 1

Press any key to continue . . .

1. **Quick sort and Minmax algorithm**

**Problem Statement:**

a) Write a c program to sort an integer array containing the following elements [ 55, 11, 33, 23, -17, 72, -11, 89, 43] using quicksort algorithm .

b) Write a c program to find min and max element in an integer array containing the following elements [ 55, 11, 33, 23, -17, 72, -11, 89, 43] using min max algorithm .

**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.1) 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] = ∞.

{ v:= a[m]; i :=m;j := p;

repeat

{

repeat

i:= i + 1;

until (a[i] >= v);

repeat

j:=j-1;

until (a[i] <=v); if (i<j) then Interchange(a, i, j);

} until (i >= j);

a[m]:= a[j]; a[j]:=u; return j; }

2. Algorithm Interchange(a,i,j)

// Exchange a[i] with a[j]

{ p:= a[i]; a[i] : = a[j];

A[j] : = p;

}

3.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

{

// divide P 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.

}

}

**Time and Space Complexity**

(i) Let T(n) represent the number of comparisons needed for MaxMin, then the recurrence relation is:

T(n) = 0 if n=1

=1 if n=2

=T(n/2)+T(n/2)+2 if n>2

T(n)=2T(n/2)+2 --- (1)

From (1)

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

Substitute (2) in (1)

T(n)=2[2T(n/4)+2]+2 =4T(n/4)+4+2----(3)

From (1)

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

Substitute (4) in (2)

T(n)= 4[2T(n/48+2]+4+2

=8t(n/8)+8+4+2

…..

…..

……

2^k-1 T(2^k/2^k-1)+2^k-2

2^k-1 T(2)+2^k-2

=2^k-1+2^k-2

=n/2+n-2

=3n/2-2

=O(n)

Space Complexity: O(1)

(ii)Let T(n) represent the number of comparisons needed for Quicksort, then the recurrence relation is:

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

T(n) = 1 if n=1

= 2 T(n/2)+n if n>1 T(n)=2T(n/2)+n --- (1)

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

Substitute (2) in (1)

T(n)=2[2T(n/4)+/(n2)]+n =4T(n/4)+2n----(3)

By substituting T(n/4) in ---(3)

T(n)=8T(n/8)+3n

T(n)= nT(n/n)+ logn\*n

T(n) =n+nlogn= O(nlogn)

This represents best and avg case.

The worst case is O(n^2).

Space Complexity: O(logn)

**Code**

**Program 1:**

#include <stdio.h>

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

// utility functions for qs

int size, count = 1;

void sw(int a[], int i, int j)

{

int t = a[i];

a[i] = a[j], a[j] = t;

}

void dis(int a[], int s)

{

int i;

for (i = 0; i < s; i++)

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

printf("\n");

}

int part(int a[], int l, int r)

{

int j;

int piv = a[r], i = l; // we choose the rightmost element is the pivot

for (j = l; j < r; j++)

{

if (a[j] < piv)

{

sw(a, i, j);

i++;

}

}

sw(a, i, r);

return i;

}

void qs(int a[], int l, int r)

{

// step 1: find the partition index

if (l < r)

{

int p = part(a, l, r);

printf("pass %d: ", count), count++;

dis(a, size);

qs(a, l, p - 1);

qs(a, p + 1, r);

}

}

int main()

{

int i;

printf("Enter size of the array: ");

sint(size);

int a[size];

for (i = 0; i < size; i++)

sint(a[i]);

printf("Original Array: ");

dis(a, size);

qs(a, 0, size - 1);

printf("Sorted Array: ");

dis(a, size);

return 0;

}

**Output:**

Enter size of the array: 9

55 11 33 23 -17 89 -11 72 43

Original Array: 55 11 33 23 -17 89 -11 72 43

pass 1: 11 33 23 -17 -11 43 55 72 89

pass 2: -17 -11 23 11 33 43 55 72 89

pass 3: -17 -11 23 11 33 43 55 72 89

pass 4: -17 -11 11 23 33 43 55 72 89

pass 5: -17 -11 11 23 33 43 55 72 89

pass 6: -17 -11 11 23 33 43 55 72 89

Sorted Array: -17 -11 11 23 33 43 55 72 89

--------------------------------

Process exited after 39.38 seconds with return value 0

Press any key to continue . . .

**Program 2:**

#include<stdio.h>

#define MAX 100

int a[MAX];

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 max1,min1;

minmax(i,mid,max,min);

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

if(max1>\*max)

\*max=max1;

if(min1<\*min)

\*min=min1;

}

}

int main()

{

int min,max,n,i;

printf("enter the no of elements - ");

scanf("%d",&n);

printf("enter the elements\n");

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

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

}

printf("The min and max elements:\n");

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

printf("Min = %d\nMax = %d",min,max);

}

**Output:**

enter the no of elements - 9

enter the elements

55 11 33 23 -17 89 -11 72 43

The min and max elements:

Min = -17

Max = 89

--------------------------------

1. **Finding kth smallest element**

**Problem Statement:**

1. Write a c program to find the Kth smallest element in an integer array containing the following elements [23, 92, 72, 87, 65, 45, 68, 79, 89, 17, 8] using Kth smallest algorithm

i) Find 6th element. ii) Find 10th element.

2. Write a c program to find the Kth smallest element in a character array containing the following elements [‘H’, ‘Z’, ‘Q’ ,’U’, ‘L’, ‘J’,’O’,’ T’,’

W’,’ D’, ‘A’] using Kth smallest algorithm

i) Find 6th element. ii) Find 2nd element

**Algorithm**

Algorithm Select kth smallest(a,n,k)

//selects the kth smallest element in a[1:n]

//and places it in the kth position of a[]

{

low:=1; up:=n+1; a[n+1] = ∞; repeat {

j= Partition(a,low,up); if k =j then return; else if k>j then low:=j+1; else up: = j;

}until(false)

}

**Time and Space Complexity**

**3a]** Time Complexity: The worst-case time complexity for this algorithm is O(n), but it can be improved if we choose the pivot element randomly. If we randomly select the pivot, the expected time complexity would be linear, O(n).

Space Complexity: O(logn) average for recursion call stack

**3b]**

**Time Complexity:**

This algorithm takes O(nm) time where n is the length of the string and m is the length of the substring to be searched. n time to loop through the characters of the string and m time is taken to calculate the previous index of the mismatched character in the substring.

**Space Complexity:**

Takes auxiliary space of O(1) or total space complexity of O(n+m) where n, m are the lengths of the string, substring respectively if we consider the input sizes.

**Code**

**Program 1:**

#include <stdio.h>

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

int size;

// utility functions

void sw(int a[], int i, int j)

{

int t = a[i];

a[i] = a[j], a[j] = t;

}

void dis(int a[], int s)

{

int i;

for (i = 0; i < s; i++)

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

printf("\n");

}

int part(int a[], int l, int r)

{

int j;

int piv = a[r], i = l; // we choose the rightmost element is the pivot

for (j = l; j < r; j++)

{

if (a[j] < piv)

{

sw(a, i, j);

i++;

}

}

sw(a, i, r);

dis(a, size);

return i;

}

void ksmall(int a[], int l, int r, int x)

{

if (r < x)

{

printf("out of bounds\n");

return;

}

do

{

int p = part(a, l, r);

printf("pivot: %d\n", p);

if (p == x)

{

printf("The kth smallest element is: %d\n", a[p]);

return;

}

else if (p > x)

r = p - 1;

else

l = p + 1;

} while (l <= r);

printf("The kth smallest element is not present\n");

return;

// if (l <= r)

// {

// int p = part(a, l, r);

// printf("pivot: %d\n", p);

// if (p == x)

// {

// printf("%dth smallest element is: %d\n", ++x, a[p]);

// return;

// }

// else if (p > x)

// ksmall(a, l, p - 1, x);

// else if (p < x)

// ksmall(a, p + 1, r, x);

// }

// else

// printf("The element is not present\n");

}

int main()

{

int x,i;

printf("Enter size of the array: ");

sint(size);

int a[size];

for (i = 0; i < size; i++)

sint(a[i]);

// int a[] = {42, 5, 90, 33, 28, 88, 56, 54, 11, 23, 76};

// size = sizeof(a) / sizeof(a[0]);

while (x != -1)

{

printf("Enter the kth value to be searched: ");

sint(x);

x--;

ksmall(a, 0, size - 1, x);

}

return 0;

}

**Output:**

Enter size of the array: 11

23 92 72 87 65 45 68 79 89 17 8

Enter the kth value to be searched: 6

8 92 72 87 65 45 68 79 89 17 23

pivot: 0

8 17 23 87 65 45 68 79 89 92 72

pivot: 2

8 17 23 65 45 68 72 79 89 92 87

pivot: 6

8 17 23 65 45 68 72 79 89 92 87

pivot: 5

The kth smallest element is: 68

Enter the kth value to be searched: 10

8 17 23 65 45 68 72 79 87 92 89

pivot: 8

8 17 23 65 45 68 72 79 87 89 92

pivot: 9

The kth smallest element is: 89

**Program 2:**

#include <stdio.h>

#define sch(x) scanf(" %c", &x);

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

int size;

void sw(char a[], int i, int j)

{

char t = a[i];

a[i] = a[j], a[j] = t;

}

void dis(char a[], int s)

{

int i;

for (i = 0; i < s; i++)

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

printf("\n");

}

int part(char a[], int l, int r)

{

int j;

int piv = a[r], i = l; // we choose the rightmost element is the pivot

for (j = l; j < r; j++)

{

if (a[j] < piv)

{

sw(a, i, j);

i++;

}

}

sw(a, i, r);

dis(a, size);

return i;

}

void ksmall(char a[], int l, int r, int x)

{

if (r < x)

{

printf("out of bounds\n");

return;

}

do

{

int p = part(a, l, r);

printf("pivot: %d\n", p);

if (p == x)

{

printf("The kth smallest element is: %c\n", a[p]);

return;

}

else if (p > x)

r = p - 1;

else

l = p + 1;

} while (l <= r);

printf("The kth smallest element is not present\n");

return;

// if (l <= r)

// {

// int p = part(a, l, r);

// printf("pivot: %d\n", p);

// if (p == x)

// {

// printf("%dth smallest element is: %c\n", ++x, a[p]);

// return;

// }

// else if (p > x)

// ksmall(a, l, p - 1, x);

// else if (p < x)

// ksmall(a, p + 1, r, x);

// }

// else

// printf("The element is not present\n");

}

int main()

{

// int size;

int i;

printf("Enter size of the array: ");

sint(size);

char a[size];

for (i = 0; i < size; i++)

sch(a[i]);

// int a[]={42, 5, 90, 33, 28, 88, 56, 54, 11, 23, 76};

// size=sizeof(a)/sizeof(a[0]);

int x;

while (x != -1)

{

printf("\nEnter the kth value to be searched: ");

sint(x);

x--;

ksmall(a, 0, size - 1, x);

}

return 0;

}

**Output:**

Enter size of the array: 11

H Z Q U L J O T W D A

Enter the kth value to be searched: 6

A Z Q U L J O T W D H

pivot: 0

A D H U L J O T W Z Q

pivot: 2

A D H L J O Q T W Z U

pivot: 6

A D H L J O Q T W Z U

pivot: 5

The kth smallest element is: O

Enter the kth value to be searched: 2

A D H L J O Q T U Z W

pivot: 8

A D H L J O Q T U Z W

pivot: 7

A D H L J O Q T U Z W

pivot: 6

A D H L J O Q T U Z W

pivot: 5

A D H J L O Q T U Z W

pivot: 3

A D H J L O Q T U Z W

pivot: 2

A D H J L O Q T U Z W

pivot: 1

The kth smallest element is: D

1. **Strassen’s Matrix Multiplication**

**Problem Statement:**

**3b]** Write a c program to multiply the following matrices using Stressan’s Matrix Multiplication algorithm.

i) Matrix A is: Matrix B is:

9 1 5 -6

4 2 -3 -7 ii)Matrix A is:

-12 7 5 -2

8 6 -3 4

11 -1 0 8

-9 6 2 7

Matrix B is:

2 -3 4 8

7 -6 5 -9

-3 4 -6 7

-9 8 2 1

**Algorithm**

P= (A11+A22) (B11+ B22)

1. = (A21 + A22) B11
2. = A11(B12-B22)
3. = A22 (B21-B11)
4. = (A11 + A12) B22
5. = (A21-A11)(B11+ B12)
6. (A12-A22)(B21+ B22)

C11= P+S-T+V

C12 = R+T

C21 = Q+S

C22 = P+R-Q+U

**Time and Space Complexity**

**Time complexity:**

The time complexity of Strassen's algorithm for matrix multiplication is O(n^log2(7)), which is approximately O(n^2.81). This is faster than the traditional algorithm for matrix multiplication, which has a time complexity of O(n^3). However, in practice, Strassen's algorithm is not always faster than the traditional algorithm for small matrices due to the overhead involved in the recursive calls. The crossover point depends on the implementation and hardware used.

**Space complexity:**

The space complexity of Strassen's algorithm for matrix multiplication is O(n^log2(7)), which is approximately O(n^2.81). This is because the algorithm involves recursive calls that split the input matrices into submatrices, and each recursive call requires additional space to store these submatrices. Specifically, the space complexity of Strassen's algorithm can be expressed as:

S(n) = 7S(n/2) + O(n^2)

where S(n) is the space required to multiply two n × n matrices using Strassen's algorithm, and the O(n^2) term represents the space required to store the result matrix. Solving this recurrence relation using the master theorem yields a space complexity of O(n^log2(7)).

**Code**

#include<stdio.h>

#define m 4

void print(int n,int arr[][n])

{

int i,j;

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

{

for(j=0;j<n;j++)

{

if(arr[i][j]<0)

printf("%d ",arr[i][j]);

else

printf(" %d ",arr[i][j]);

}

printf("\n");

}

}

void add\_sub(int size\_a,int size\_b,int size\_c,int a[size\_a][size\_a],int b[size\_b][size\_b],int c[size\_c][size\_c],int add,int a\_row,int a\_col,int b\_row,int b\_col)

{

int i,j;

if(add==0)

add=-1;

for(i=0;i<size\_c;i++)

for(j=0;j<size\_c;j++)

c[i][j]=a[i+a\_row][j+a\_col]+b[i+b\_row][j+b\_col]\*add;

}

void zero(int n,int a[n][n])

{

int i,j;

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

for(j=0;j<n;j++)

a[i][j]=0;

}

void direct\_mul(int n,int a[n][n],int b[n][n],int c[n][n])

{

zero(n,c);

int i,j,k;

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

for(j=0;j<n;j++)

for(k=0;k<n;k++)

c[i][j]+=(a[i][k]\*b[k][j]);

}

void mul(int n,int a[n][n],int b[n][n],int c[n][n])

{

if(n==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 p[n/2][n/2],q[n/2][n/2],r[n/2][n/2];

int s[n/2][n/2],t[n/2][n/2],u[n/2][n/2],v[n/2][n/2];

int temp1[n/2][n/2],temp2[n/2][n/2];

int i,j;

zero(n/2,p);

zero(n/2,q);

zero(n/2,r);

zero(n/2,s);

zero(n/2,t);

zero(n/2,u);

zero(n/2,v);

zero(n/2,temp1);

zero(n/2,temp2);

//calc for p

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

add\_sub(n,n,n/2,a,a,temp1,1,0,0,n/2,n/2);

add\_sub(n,n,n/2,b,b,temp2,1,0,0,n/2,n/2);

mul(n/2,temp1,temp2,p);

//calc for q

// printf("q = (a[1][0]+a[1][1])\*b[0][0];\n");

add\_sub(n,n,n/2,a,a,temp1,1,n/2,0,n/2,n/2);

for(i=0;i<n/2;i++)

for(j=0;j<n/2;j++)

temp2[i][j]=b[i][j];

mul(n/2,temp1,temp2,q);

//calc for r

// printf("r = a[0][0]\*(b[0][1]-b[1][1]);\n");

for(i=0;i<n/2;i++)

for(j=0;j<n/2;j++)

temp1[i][j]=a[i][j];

add\_sub(n,n,n/2,b,b,temp2,0,0,n/2,n/2,n/2);

mul(n/2,temp1,temp2,r);

//calc for s

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

for(i=0;i<n/2;i++)

for(j=0;j<n/2;j++)

temp1[i][j]=a[i+n/2][j+n/2];

add\_sub(n,n,n/2,b,b,temp2,0,n/2,0,0,0);

mul(n/2,temp1,temp2,s);

//calc for t

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

add\_sub(n,n,n/2,a,a,temp1,1,0,0,0,n/2);

for(i=0;i<n/2;i++)

for(j=0;j<n/2;j++)

temp2[i][j]=b[i+n/2][j+n/2];

mul(n/2,temp1,temp2,t);

//calc for u

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

add\_sub(n,n,n/2,a,a,temp1,0,n/2,0,0,0);

add\_sub(n,n,n/2,b,b,temp2,1,0,0,0,n/2);

mul(n/2,temp1,temp2,u);

//calc for v

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

add\_sub(n,n,n/2,a,a,temp1,0,0,n/2,n/2,n/2);

add\_sub(n,n,n/2,b,b,temp2,1,n/2,0,n/2,n/2);

mul(n/2,temp1,temp2,v);

printf("p = \n");

print(n/2,p);

printf("q = \n");

print(n/2,q);

printf("r = \n");

print(n/2,r);

printf("s = \n");

print(n/2,s);

printf("t = \n");

print(n/2,t);

printf("u = \n");

print(n/2,u);

printf("v = \n");

print(n/2,v);

//cal c00

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

add\_sub(n/2,n/2,n/2,p,s,temp1,1,0,0,0,0);

add\_sub(n/2,n/2,n/2,temp1,t,temp2,0,0,0,0,0);

add\_sub(n/2,n/2,n/2,temp2,v,temp1,1,0,0,0,0);

for(i=0;i<n/2;i++)

for(j=0;j<n/2;j++)

c[i][j]=temp1[i][j];

//cal c01

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

add\_sub(n/2,n/2,n/2,r,t,temp1,1,0,0,0,0);

for(i=0;i<n/2;i++)

for(j=0;j<n/2;j++)

c[i][j+n/2]=temp1[i][j];

//cal c10

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

add\_sub(n/2,n/2,n/2,q,s,temp1,1,0,0,0,0);

for(i=0;i<n/2;i++)

for(j=0;j<n/2;j++)

c[i+n/2][j]=temp1[i][j];

//cal c11

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

add\_sub(n/2,n/2,n/2,p,r,temp1,1,0,0,0,0);

add\_sub(n/2,n/2,n/2,temp1,q,temp1,0,0,0,0,0);

add\_sub(n/2,n/2,n/2,temp1,u,temp2,1,0,0,0,0);

for(i=0;i<n/2;i++)

for(j=0;j<n/2;j++)

c[i+n/2][j+n/2]=temp2[i][j];

}

}

int main()

{

// int a[m][m]={{5,-6},{4,2}};

// int b[m][m]={{9,1},{-3,-7}};

int a[m][m]={{12,7,5,-2},{8,6,-3,4},{11,-1,0,8},{-9,6,2,7}};

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

printf("A = \n");

print(m,a);

printf("B = \n");

print(m,b);

int c[m][m]={0};

mul(m,a,b,c);

printf("Calculated ans = \n");

print(m,c);

int i,j;

printf("\n\nActual ans :\n");

direct\_mul(m,a,b,c);

print(m,c);

}

**Output**

PS C:\Users\P JEEVESH NAIDU> cd "c:\Users\P JEEVESH NAIDU\OneDrive\Desktop\" ; if ($?) { gcc matrix.c -o matrix } ; if ($?) { .\matrix }

A =

12 7 5 -2

8 6 -3 4

11 -1 0 8

-9 6 2 7

B =

2 -3 4 8

7 -6 5 -9

-3 4 -6 7

-9 8 2 1

p =

87 -27

77 -25

q =

71 -75

77 -57

r =

141 -58

98 -52

s =

-128 112

-122 112

t =

-92 124

-10 45

u =

-102 115

-102 -85

v =

25 -35

66 -82

Calculated ans =

76 -74 49 66

31 -40 88 -7

-57 37 55 105

-45 55 -4 -105

Actual ans :

76 -74 49 66

31 -40 88 -7

-57 37 55 105

-45 55 -4 -105

**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.