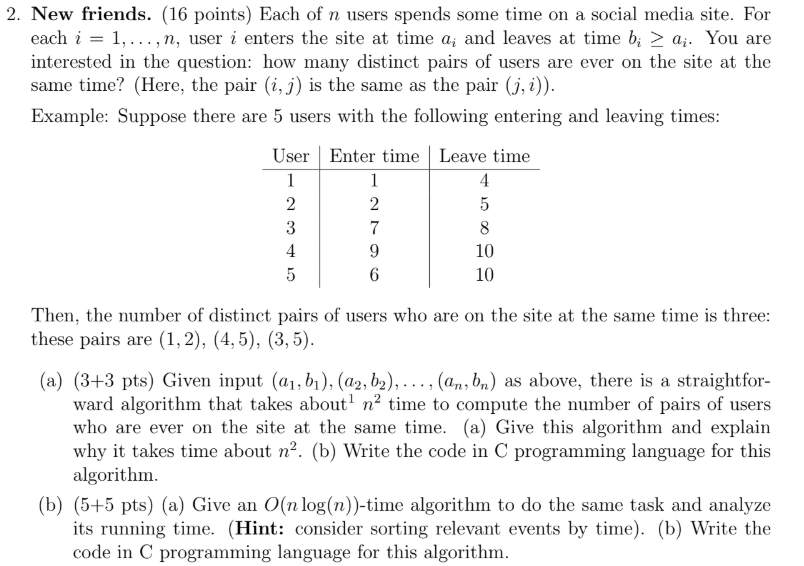


I hope to advanced algorithms and improve my problem solving skills to understand, analyze and solve such problems that usually came in challenges, contests, interviews, etc. This will also help me in future as this course will make us realize the bad coding mistakes people usually make and how to prevent our self from doing the same.



(a). Algorithm

Algorithm **I**: Time Complexity O(n^2)

This Program will check how many unique pairs of students will be

online at the same time based on their entry and exit times

The Algorithm Uses Two Nested Loops Which Go over N times

**Input**:

n <= total no of students

people <= [(entry, exit), ...] Array defining a students' entry/exit time

no\_of\_pairs <= 0

**FOR** i from 0 to n - 1:

**FOR** j from (i+1) to n - 1:

**IF** (person[i].entry < person[j].exit and person[j].entry < person[i].exit):

Increment no\_of\_pairs by 1

(a). C Program:

#include <stdio.h>

typedef struct person{

int entry;

int exit;

} Person;

int main(int argc, char const \*argv[]){

int noOfPeople;

scanf("%d", &noOfPeople);

Person people[noOfPeople];

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

scanf("%d %d", &people[i].entry, &people[i].exit);

}

int distinctPairs = 0;

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

for (int j = i + 1; j < noOfPeople; j++){

if(people[i].entry < people[j].exit && people[j].entry < people[i].exit){

distinctPairs++;

}

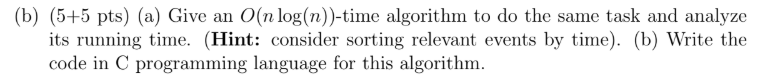
}

}

printf("Possible distinct pairs are %d", distinctPairs);

return 0;

}



1. . Prerequisite Algorithms:

1-Binary Search

Algorithm **BINARYSEARCH**: Time Complexity O(log(n))

This Program will search for the greatest no. less than or equal

to the Value given in the Sorted Array

**BINARYSEARCH**(Array, val)

l <= 0, r <= Array.Length -1

**WHILE** l < r:

**GET** Middle Element and compare with val

mid <= l + (r - l) / 2;

**IF** val is less than Array[mid]:

**SEARCH** for val in right half of sorted Array:

l <= mid + 1

**ELSE**

**SEARCH** for val in left half of sorted Array:

r <= mid

**IF** Array[l] == val:

RETURN l

**ELSE**

RETURN l - 1

2-Merge Sort

Algorithm **MERGESORT**: Time Complexity O(n\*log(n))

**MERGESORT**(arr[], l, r)

**IF** l == r

**RETURN**

**ELSE**

Find the middle point to divide the array into two halves:

middle m <= (l+r)/2

Do **MERGESORT** for first half:

Call **MERGESORT**(arr, l, m)

Do MERGESORT for second half:

Call **MERGESORT**(arr, m+1, r)

MERGE the two halves sorted in step 2 and 3:

Call **MERGE**(arr, l, m, r)

Algorithm MERGE: Time Complexity O(n)

**MERGE**(Array, l, m, r):

// Array1 is Array[l:m], Array2 is Array[m+1:r]

Create a Duplicate array of Total Length:

Temp <= Array of Size [ Array1.Length + Array2.Length ]

index <= 0

Initialize Index for both Array:

i <= 0, j <= 0

**WHILE** index < Temp.Length:

**IF** any Array becomes empty

**IF** i == Array1.Length :

Temp[index++] <= Array2[j++]

**IF** j == Array2.Length :

Temp[index++] <= Array1[i++]

**CONTINUE**

INSERT the lower Element:

**IF** Array1[i] < Array2[j]:

Temp[index++] <= Array1[i++]

**ELSE**

Temp[index++] <= Array2[j++]

(b) Algorithm

Algorithm II: Time Complexity **O(n\*log(n))**

This Program will check how many unique no of pairs of students will be

available online based on their entry and exit times

The First Part of the Problem User MERGESORT which takes **O(n\*log(n))** Time

In Second Part we BINARYSEARCH the Value in Array over a loop of size **n** so

net Complexity will be **O(n) \* O(log(n))** => **O(n\*log(n))**

Input:

n <= Total no of students

Array defining a student structure with entry/exit time

people <= [(entry, exit), ...]

no\_of\_pairs <= 0

Apply MERGESORT on people Array

**FOR** i from 0 to n - 1:

GET Value When the ith person Leaves

val <= people[i].exit

Find How many people were logged in before ith person left

found\_index <= **BINARYSEARCH**(people, val)

Add No. of Pairs of the respective person

**Increment** no\_of\_pairs by found\_index - i

(b) C Program

#include <stdio.h>

typedef struct person {

int entry;

int exit;

} Person;

int binarySearchPersonEntry(Person Array[], int val, int start, int end)

{

while (start <= end) {

int m = start + (end - start) / 2;

if (Array[m].entry == val)

return m;

if (Array[m].entry < val)

start = m + 1;

else

end = m - 1;

}

return start - 1;

}

int compare(Person p1, Person p2){

if (p1.entry < p2.entry) {

return 1;

}

else {

if (p1.exit < p2.exit) {

return 1;

}

else {

return 0;

}

}

}

void mergeSort(Person Array[], int start, int end)

{

if (start == end) {

return;

}

int mid = (start + end - 1) / 2;

mergeSort(Array, start, mid);

mergeSort(Array, mid + 1, end);

// Merging Arrayay

int length = end - start + 1;

Person Temp[length];

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

Temp[i] = Array[i + start];

}

int i = 0, j = 0, m = mid - start + 1;

for (int c = start; c <= end; c++){

if(i >= m){

Array[c] = Temp[m + j++];

}

else if (j + mid >= end)

{

Array[c] = Temp[i++];

}

else if (compare(Temp[i], Temp[m + j]))

{

Array[c] = Temp[i++];

}

else

{

Array[c] = Temp[m + j++];

}

}

}

int main(int argc, char const \*argv[])

{

int noOfPeople;

scanf("%d", &noOfPeople);

Person people[noOfPeople];

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

scanf("%d %d", &people[i].entry, &people[i].exit);

}

mergeSort(people, 0, noOfPeople - 1);

int distinctPairs = 0;

int begin = 0, end = 0, current\_online = 0;

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

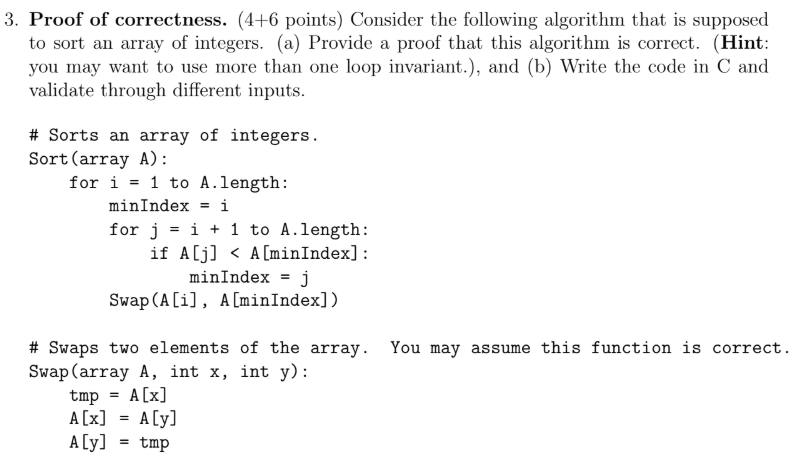
int k = binarySearchPersonEntry(people, people[i].exit - 1, i+1, noOfPeople-1) - i;

distinctPairs = k;

}

printf("No. of distinct pairs are: %d.\n", distinctPairs);

}



**Inductive Hypothesis:** After iteration of *i* in the outer loop the array A[0 : i] will be sorted and all the Elements in A[i+1 :end] are greater than elements in A[0 : i].

**Base Case:** At beginning of outerLoop *i = 1* The Array A[0:1] will containsingle element which is already sorted since it is the only element in the Array.

**Inductive Step:** Suppose that the inductive hypothesis holds for *i-1* so A[0 : i-1] is sorted and all the elements in Array A[i : end] are larger than all elements in the Previous Array, after the i-1th iteration. We want to show that Array A[0 : i] is sorted and contains the least i members of entire Array A after the *ith* iteration.

Suppose that *kth* element is the smallest integer in the remaining Unsorted Array A[i : end]. Then the effect of the inner loop is to swap the position of ith element with kth element, which will convert the existing array

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A[0] | A[1] | … | A[i-1] | A[i] | … | A[k] | … | A[end] |

Into following:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A[0] | A[1] | … | A[i-1] | A[k] | … | A[i] | … | A[end] |

We can now claim that the following list is sorted:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A[0] | A[1] | A[2] | … | A[i-1] | A[k] |

This is because the first i-1 elements are already sorted and Since all the remaining elements were larger than all the elements in Array A[0 : i-1]. Hence A[k] > A[i-1]. Also Since A[k] was the smallest in the Array A[i : end] hence all the remaining elements A[i+1: end] are larger than A[k] Satisfying the second condition. Since Both the conditions are satisfied and element A[k] is in the right place, this proves the claim.

Thus, After the ith iteration completes the Array A[0: i] is sorted and this establishes the inductive hypothesis for i.

Hence the Correctness of Algorithm is Proved.

(b)- C Program

#include <stdio.h>

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

\*a = \*a + \*b;

\*b = \*a - \*b;

\*a = \*a - \*b;

}

void sort(int arr[], int length){

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

int pos = i;

for (int j = i + 1; j < length; j++){

if(arr[j] < arr[pos]){

pos = j;

}

}

if(i != pos){

swap(&arr[i], &arr[pos]);

}

}

}

int main(){

int n;

printf("Enter length: ");

scanf("%d", &n);

int arr[n];

printf("Enter numbers: ");

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

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

}

sort(arr, n);

printf("Sorted Array: ");

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

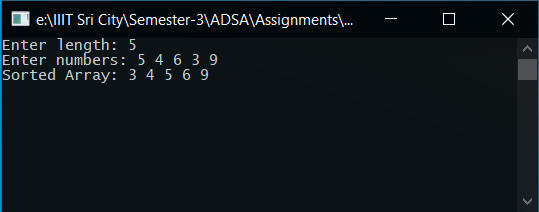
printf("%d ", arr[i]);

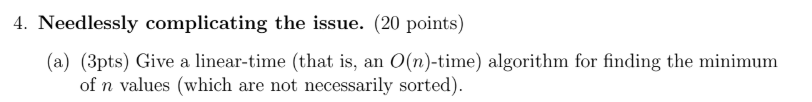
}

printf("\n");

}

Output





(A)- Algorithm

**Algorithm**: Time Complexity **O(n)**

This Program will Search the Array for the least element and will return it

The Algorithm Uses a Single Loop in which we will match each element

with the minimum element and update the minimum element if turns out to be smaller

Since Every element is matched once the complexity will be **O(n)**

Input:

n <= total no of students

arr <= [...] Array of Numbers

**DEFINE** first element as the least

min = arr[0]

**FOR** i from 1 to n:

**IF** arr[i] is less than min:

**DEFINE** min as arr[i]

min <= arr[i]

RETURN min

(b)

Since The array is not sorted, hence the smallest element can lie at any position, Hence In the worst case the program/Algorithm has to check each and every element, If we do not check the value of any element there is probability that the unchecked element is the least element, which will lead to wrong output. So, a Minimum of n comparison are required in worst case to find least element in an Array.



(c)- C Program

#include <stdio.h>

int findMinimum(int arr[], int length){

int min = arr[0];

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

if (arr[i] < min) {

min = arr[i];

}

}

return min;

}

int main(){

int n;

printf("Enter length: ");

scanf("%d", &n);

int arr[n];

printf("Enter numbers: ");

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

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

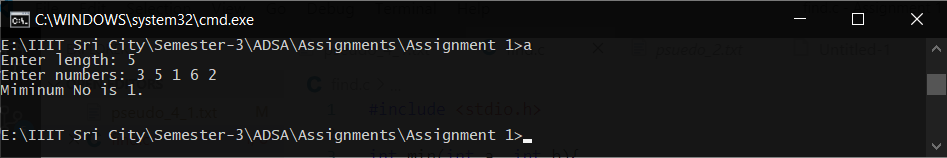
}

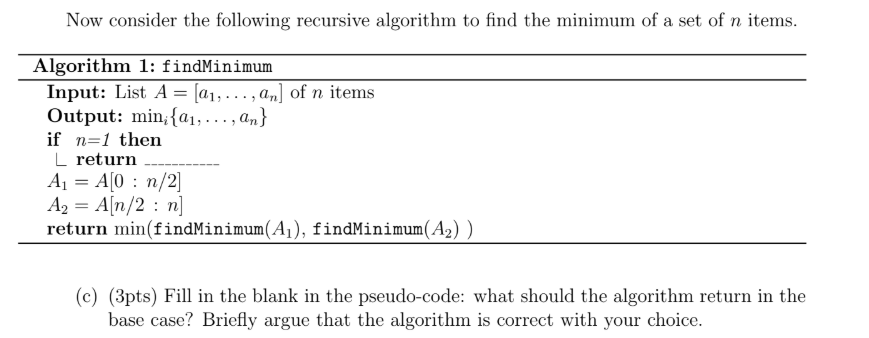
printf("Miminum No is %d.\n", findMinimum(arr, n));

printf("\n");

}

Output:







(d)-Algorithm

**Input:** List A = *{a1, a2, …, an}* of *n* items

**Output:** min*i*{a1, a2, …, an}

**IF** n =1 then

**RETURN** a*1*

A1 <= A[0 : n/2]

A2 <= A[n/2 : n]

**RETURN** min(findMinimum(A1), findMinimum(A2) )

The Algorithm will first compare the results of sub-problems of left and right arrays. When the base case occurs it will just return the element itself. This will create a Tree-like recursive call where each node will return the minimum element of its children and ultimately we will find the minimum of array.



(e)-Running Time Analysis

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  | | --- | --- | --- | --- | | Level | # Problem | Time | Complexity | | 0 | 1 | 1 | 1 | | 1 | 2 | 1 | 2 | | 2 | 4 | 1 | 4 | | … | … | … | …. | | … | … | … | …. | | Log2(n) | n | 1 | n | |

Since the process of Dividing and Conquering takes O(1) time we can find the net complexity of the Algorithm as 1 + 2 + 4 + … +n/2 + n = 2n - 1 = O(n)

Since This method Also takes Linear time it of same complexity as of previous Problem.



(f)- C Program

#include <stdio.h>

int min(int a, int b){

return ((a > b) ? b : a);

}

int findMinimum(int arr[], int start, int end){

if(start == end){

return arr[start];

}

int mid = start + (end - start) / 2;

return(min(findMinimum(arr, start, mid), findMinimum(arr, mid+1, end)));

}

int main(){

int n;

printf("Enter length: ");

scanf("%d", &n);

int arr[n];

printf("Enter numbers: ");

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

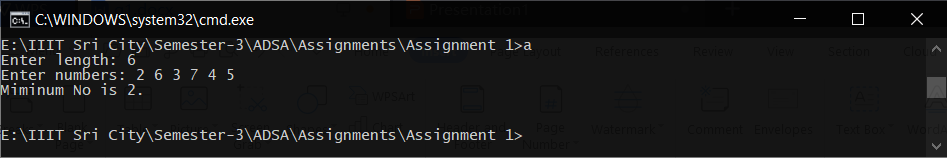
scanf("%d", &arr[i]);

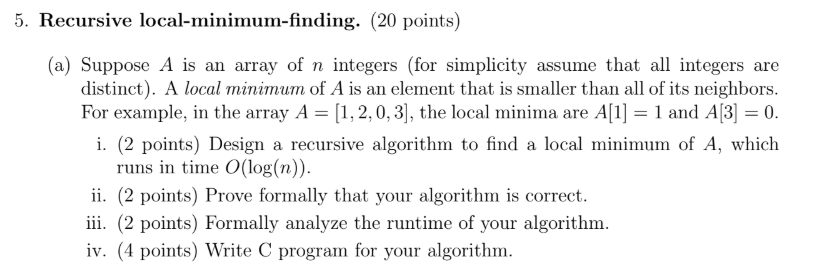
}

printf("Miminum No is %d.\n", findMinimum(arr,0, n-1));

printf("\n");

}





1. Algorithm

Algorithm: **FINDLOCALMINIMA** Time Complexity: **O(log(n))**

We compare middle element with its neighbors. If middle element is smaller than both of its neighbors, then we return it.

If the middle element is greater than its left neighbor, then there is always a local minima in left half.

If the middle element is greater than its right neighbor, then there is always a local minima in right half.

**FINDLOCALMINIMA**(arr, start, end, n):

GET index of middle element

mid <= (low + high) / 2

**IF** neighbour are there:

**IF** ((mid == 0 || arr[mid-1] > arr[mid]) and

(mid == n-1 || arr[mid+1] > arr[mid]))

**RETURN** mid

**IF** Left Half Contains the minima serach in left-Half

otherwise serach in right-half:

**IF** mid > 0 and arr[mid-1] < arr[mid]:

**RETURN** FINDLOCALMINIMA(arr, low, (mid -1), n)

**ELSE**

**RETURN** FINDLOCALMINIMA(arr, (mid + 1), high, n);

1. Correctness of Proof:

**Recursive Invariant:** If the middle element is smaller than both its neighbour than it is the local minima, If right neighbour element is smaller than the middle element then it is guaranteed that there will always be a local minima in right-half of the array, similarly if left neighbour element is smaller than the middle element

then there must be local minima in the left-half of array

**Inductive Hypothesis:** If the recursive invariant is true than the algorithm will find

**Base Case:** When n = 1 then the array has a single element hence it is a local minima.

**Inductive Step:** When we check the middle element there are following cases

**Case I:** When the middle element is less than both of its neighbour, it will return the element.

**Case II:** If the middle element is greater than right neighbour, The recursive algorithm will search the right-half sub array A[mid+1: end], Now either

**“Case I”** the middle of this element is the local minima or **“Case II”** The function will search right-half of this array or **“Case III”** The function will search left-half of this array recursively till solution is reached.

**Case III:** If the middle element is greater than left neighbour, The recursive algorithm will search the left-half sub array A[start: mid], Now either

**“Case I”** the middle of this element is the local minima or **“Case II”** The function will search right-half of this array or **“Case III”** The function will search left-half of this array recursively till solution is reached.

Hence the algorithm will ultimately return a local mining.

(III)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  | | --- | --- | --- | --- | | Level | # Problems | Time | Complexity | | 0 | 1 | 1 | 1 | | 1 | 1 | 1 | 1 | | 2 | 1 | 1 | 1 | | … | … | … | … | | … | … | … | … | | Log2(n) | 1 | 1 | 1 | |

Since the process of Dividing and Conquering has conditional statements only it takes O(1) time and only a Single branch is formed we can find the net complexity of the Algorithm as 1 + 1 + … log2(n) +1 times = O(log(n))

So This method takes Logarithmic time.

(IV)- C Program

#include <stdio.h>

int local\_minima(int arr[], int low, int high, int length){

int mid = low + (high - low)/2;

if ((mid == 0 || arr[mid-1] > arr[mid]) && (mid == length-1 || arr[mid+1] > arr[mid]))

return mid;

else if (mid > 0 && arr[mid-1] < arr[mid])

return local\_minima(arr, low, (mid -1), length);

return local\_minima(arr, (mid + 1), high, length);

}

int main(){

int n;

printf("Enter length: ");

scanf("%d", &n);

int arr[n];

printf("Enter numbers: ");

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

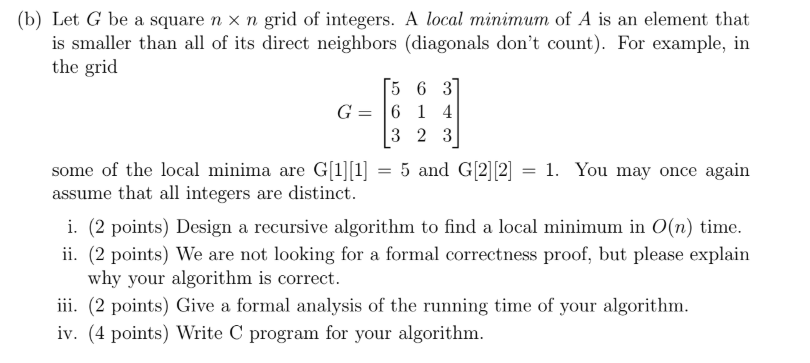
scanf("%d", &arr[i]);

}

printf("Local Mimima is %d.\n", local\_minima(arr, 0, n-1, n));

printf("\n");

}



(I)- Algorithm (Check Is Minima at given position)

**Algorithm:** ISMINIMA Time Complexity **O(1)**

This Algorithm check the Neighbour of arr[x][y] and returns 1

If it is smaller than all otherwise returns 0

**ISMINIMA**(arr[][], int x, int y, int len){

check <= 1

**IF** arr[x][y] is a local minima

**IF** arr[x][y] is More than Top Element:

check <= 0

**IF** arr[x][y] is More than Bottom Element:

check <= 0

**IF** arr[x][y] is More than Left Element:

check <= 0

**IF** arr[x][y] is More than Right Element:

check <= 0

**RETURN** check;

}

Algorithm **FINDLOCALMINIMA2**: Time Complexity: O(n)

This Program Finds the index for a local Minima in a 2D n\*n matrix

It checks the boundry and middle row and colums and checks if the

element in the center is Minimum, if not we recurse this problem

for smaller quadrants.

**FINDLOCALMINIMA2**(A[][], Xi, Yi, Xn, n):

**IF** Xi is larger than or equal to Xn:

**RETURN** A[Yi][Yi]

Min<= arr[Yi][Xi]

MinX <= Yi

MinY <= Xi

MiddleX <= Xi + (Xn - Xi)/2

MiddleY <= Yi + (Xn - Xi)/2

length <= Xn - Xi

CHECK through the columns

FOR i from Xi to length:

FOR j from Yi to length:

check if local minima

IF ISMINIMA(A, i, j, length)

RETURN A[j][i]

ELSE

IF A[j][i] less than Min:

min <= A[j][i]

i += MiddleX

CHECK through the the rows

FOR i from Yi to length:

FOR j from Xi to length:

IF ISMINIMA(A, i, j, length)

check FOR local minima

RETURN A[j][i]

else IF A[j][i] less than Min:

min <= A[j][i]

i += MiddleX

Search in Smaller Quadrants:

IF A[MinX][maxIndexX] > A[MinX][minIndex - 1]:

IF MinX < MiddleY:

RETURN FINDLOCALMINIMA2(arr, MinY - MiddleX, Yi, MinY, n);

else:

RETURN FINDLOCALMINIMA2(arr, MinY-MiddleX, Yi+MiddleY, MinY, n);

(II)- Correctness Intuition

When We Check for Condition of minimum of boundary and middle row and column We also check the Adjacent (Top,Down,Left,Right) elements as well. In case it is not the minimum element, We find the minimum corner and recurse into that quadrant. Since We are keeping the minimum element each time in the quadrant the min value does not increases, we keep doing that until we find a local minima for the quadrant which will also be local minima for the entire matrix.

1. Running Time Analysis

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  | | --- | --- | --- | --- | | Level | # Problems | Time | Complexity | | 0 | 1 | n | c\*n | | 1 | 1 | n/2 | c\*n/2 | | 2 | 1 | n/4 | c\*n/4 | | … | … | … | … | | … | … | … | … | | Log2(n) | 1 | 1 | c | |

Since the process of Dividing and Conquering has conditional statements only it takes O(n) time and only a Single branch is formed we can find the net complexity of the Algorithm as c( 1 + 2 + … n/2 + n) times = 2c\*n - c = **O(n)**

So This method takes Linear time.

1. - C Program

#include <stdio.h>

#include <stdlib.h>

int isMinima(int \*arr[], int x, int y, int len){

int is\_minima = 1;

if(x + 1 <= len){

if(arr[y][x + 1] < arr[y][x]){

is\_minima = 0;

}

}

if(y + 1 <= len){

if(arr[y + 1][x] < arr[y][x]){

is\_minima = 0;

}

}

if(y - 1 >= 0){

if(arr[y - 1][x] < arr[y][x]){

is\_minima = 0;

}

}

if(x - 1 >= 0){

if(arr[y][x - 1] < arr[y][x]){

is\_minima = 0;

}

}

return is\_minima;

}

int localMinima(int \*arr[], int x0, int y0, int x1, int length){

if(x0 >= x1){

return arr[y0][x0];

}

int min = arr[y0][x0];

int MinIx= x0, MinIy= y0;

int midX = x0 + (x1 - x0)/2;

int midY = y0 + (x1 - x0)/2, lastY = y0 + (x1 - x0);

int len = x1 - x0;

//Checking in the columns

for(int i = x0; i <= x0 + len; i+=(x1-x0)/2){

for(int j = y0; j <= y0 + len; j++){

if(isMinima(arr, i, j, length)){

return arr[j][i];

}

else if(arr[j][i] < min){

min = arr[j][i];

MinIx= i;

MinIy= j;

Output

}

}

}

// Checking in the rows

for(int j = y0; j <= y0 + len; j+=(x1-x0)/2){

for(int i = x0; i <= x0 + len; i++){

if(isMinima(arr, i, j, length)){

return arr[j][i];

}

else if(arr[j][i] < min){

min = arr[j][i];

MinIx= i;

MinIy= j;

}

}

}

if(MinIx- 1 >= 0){

if(arr[minIndY][minIndX] > arr[minIndY][MinIx- 1]){

if(MinIy< midY){

return localMinima(arr, MinIx- midX, y0, minIndX, length);

}

else

{

return localMinima(arr, MinIx- midX, y0 + midY, minIndX, length);

}

}

}

else if(MinIx+ 1 < length){

if(arr[minIndY][minIndX] > arr[minIndY][MinIx+ 1]){

if(MinIy< midY){

return localMinima(arr, minIndX, y0, MinIx+ midX, length);

}

else

{

return localMinima(arr, minIndX, y0 + midY, MinIx+ midX, length);

}

}

}

else if(MinIy- 1 >= 0){

if(arr[minIndY][minIndX] > arr[MinIy- 1][minIndX]){

if(MinIx< midX){

return localMinima(arr, x0, MinIy- midY, midX, length);

}

else

{

return localMinima(arr, midX, MinIy- midY, x1, length);

}

}

}

else if(MinIy+ 1 < length){

if(arr[minIndY][minIndX] > arr[MinIy+ 1][minIndX]){

if(MinIx< midX){

return localMinima(arr, x0, MinIy+ midY, midX, length);

}

else

{

return localMinima(arr, midX, MinIy+ midY, x1, length);

}

}

}

return 0;

}

int main(){

int n;

printf("Enter length: ");

scanf("%d", &n);

int \*\*arr = (void\*)malloc(sizeof(int\*) \* n);

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

\*(arr + i) = (void\*)malloc(sizeof(int) \* n);

}

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

printf("Enter Row: ");

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

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

}

}

printf("Miminum No is %d.\n", localMinima(arr, 0, 0, n-1, n));

printf("\n");

}

