Experiment no-3 Date:10/02/18

Aim:-Experiment on finding minimum and maximum numbers using divide and conquer approach

Theory:-

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **divide and conquer** is an [algorithm design paradigm](https://en.wikipedia.org/wiki/Algorithm_design_paradigm) based on multi-branched [recursion](https://en.wikipedia.org/wiki/Recursion). A divide-and-conquer [algorithm](https://en.wikipedia.org/wiki/Algorithm) works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Understanding and designing divide-and-conquer algorithms is a complex skill that requires a good understanding of the nature of the underlying problem to be solved. As when proving a [theorem](https://en.wikipedia.org/wiki/Theorem) by [induction](https://en.wikipedia.org/wiki/Mathematical_induction), it is often necessary to replace the original problem with a more general or complicated problem in order to initialize the recursion, and there is no systematic method for finding the proper generalization. These divide-and-conquer complications are seen when optimizing the calculation of a [Fibonacci number with efficient double recursion](https://en.wikipedia.org/wiki/Fibonacci_number#Matrix_form).

The correctness of a divide-and-conquer algorithm is usually proved by [mathematical induction](https://en.wikipedia.org/wiki/Mathematical_induction), and its computational cost is often determined by solving [recurrence relations](https://en.wikipedia.org/wiki/Recurrence_relation).

Algorithm for traditional method minimum and maximum:-

1.Let first element of array be initialized to maximum and minimum.

2.Traverse the array and change the minimum if current is smaller than minimum, simultaneously change the maximum if current is greater than maximum.

Algorithm for recursive method minimum and maximum:-

1.If array length is 1, minimum and maximum is same element.and return.

2.If array has two elements, find min and maximum elements of the two and return.

3. Divide the array into 2 parts. Find min and max of left subarray and right subarray. Compare the min of left and right and get minimum, compare max of right and left and get maximum. Return minimum and maximum.

*Complexity:*

     Now what is the number of element comparisons needed for MaxMin? If T(n) represents this number, then the resulting recurrence relation is

                           0                                              n=1

T(n) =      1                                              n=2

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

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

-for some positive integer k, then

  T(n) = 2T(n/2) + 2

           = 2(2T(n/4) + 2) + 2

           = 4T(n/4) + 4 + 2

           .

           .

           .

           = *2*k-1 T(2) + ∑(1≤i≤k-1) *2*k

           = *2*k-1 + *2*k – 2

           = 3n/2 – 2 = O(n)

Note that 3n/2 – 2 is the best, average, worst case number of comparison when n is a power of two.

*Comparisons with Straight Forward Method:*

        Compared with the 2n – 2 comparisons for the Straight Forward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than 3n/2 – 2 comparisons.

Program:-

#include<stdio.h>

typedef struct minmax {

int min, max;

} minmax;

minmax getMinMax(int arr[], int start, int end,int \*comp) {

minmax resultMinMax, mmLeft, mmRight;

if (start == end) {

resultMinMax.max = arr[start];

resultMinMax.min = arr[start];

return resultMinMax;

} else if (end == (start + 1)) {

if (arr[start] > arr[end]) {

resultMinMax.max = arr[start];

resultMinMax.min = arr[end];

(\*comp)++;

} else {

resultMinMax.max = arr[end];

resultMinMax.min = arr[start];

(\*comp)++;

}

return resultMinMax;

}

int mid = (start + end) / 2;

mmLeft = getMinMax(arr, start, mid,comp);

mmRight = getMinMax(arr, mid + 1, end,comp);

if (mmLeft.min < mmRight.min)

resultMinMax.min = mmLeft.min;

else

resultMinMax.min = mmRight.min;

if (mmLeft.max > mmRight.max)

resultMinMax.max = mmLeft.max;

else

resultMinMax.max = mmRight.max;

\*comp+=2;

return resultMinMax;

}

int main() {

int arr[] = {100, 80, 200, 60, 300, 500, 40,45,98,56,89,340,680,0,578,356,908,21,32,43,12,15,89,76,58};

int n = 26;

int min = arr[0], max = arr[0], i;int comp=0;

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

comp++;

if (arr[i] > max)

{

max = arr[i];

}

comp++;

if(min>arr[i])

{ min=arr[i];}

}

printf("traditional method took %d comparison \n",comp);

comp=0;

printf("Minimum element is %d\n", min);

printf("Maximum element is %d\n", max);

minmax answer = getMinMax(arr, 0, n - 1,&comp);

printf("Minimum element is %d\n", answer.min);

printf("Maximum element is %d\n", answer.max);

printf("D&C method took %d comparison \n",comp);

return 0;

}

**Output:-**

traditional method took 52 comparison

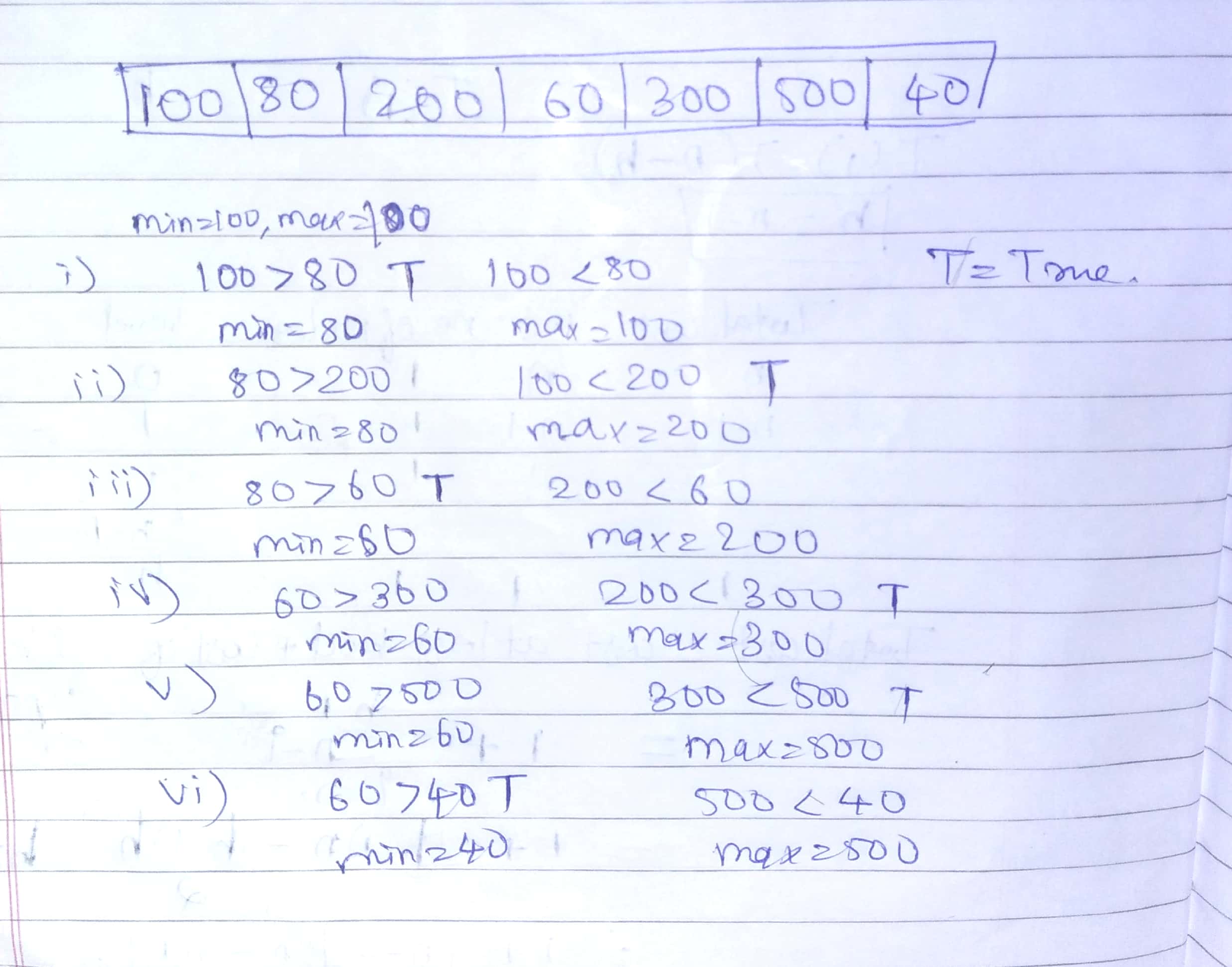
Minimum element is 0

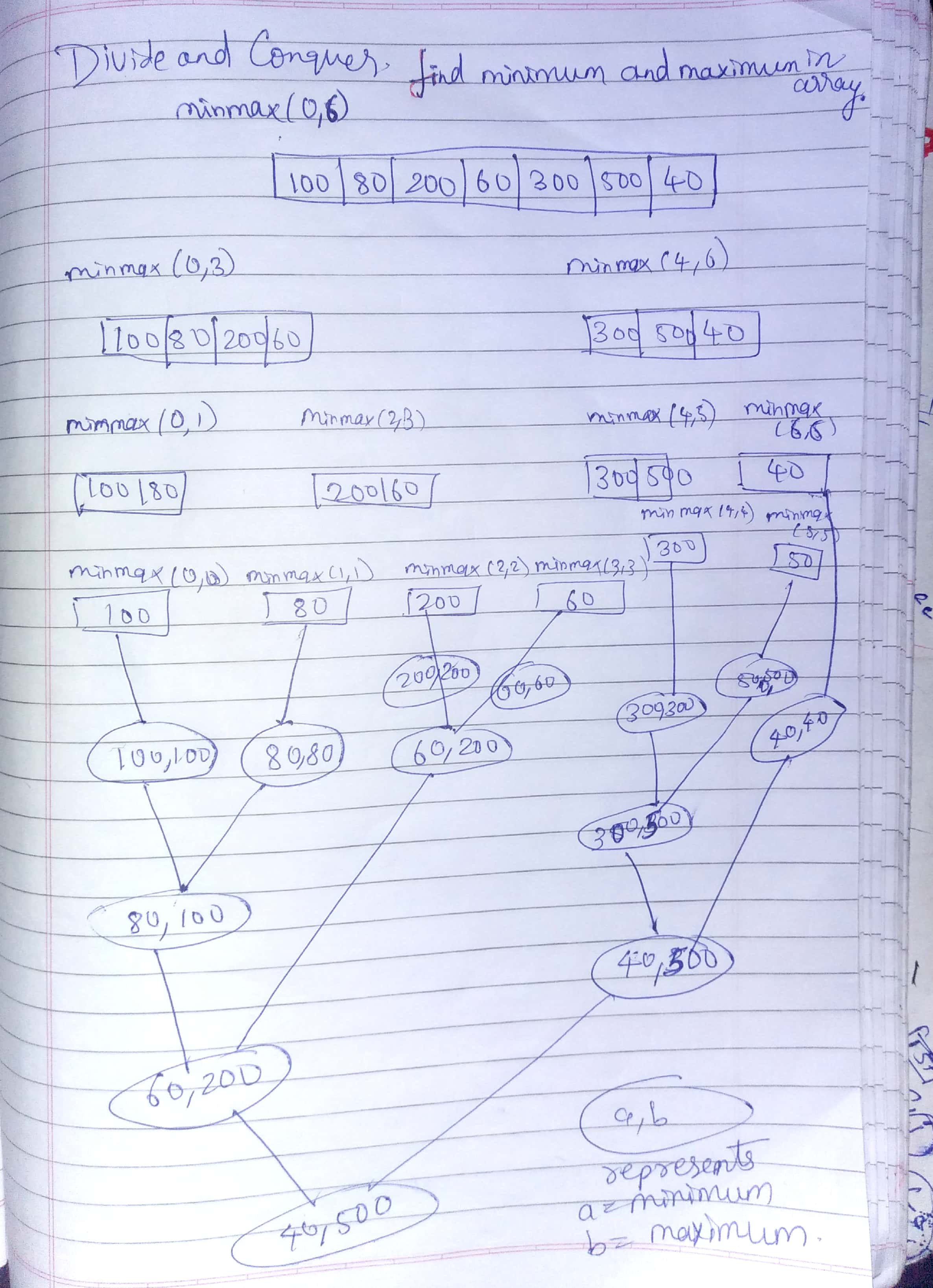
Maximum element is 908

Minimum element is 0

Maximum element is 908

D&C method took 40 comparison





**Conclusion:-**

                In traditional method the time complexity of this algorithm, we have to concentrate on the number of element comparisons. This algorithm requires 2(n-1) element comparisons in the best, average, and worst cases.

An immediate improvement is possible by realizing that the comparison min >a[i] is necessary only when a[i]>max is false.

                Now the Best case occurs when the elements are in increasing order. The number of element comparisons is n-1. The worst case occurs when the element are in decreasing order. In this case number of comparisons is 2(n-1).

But overall Divide and Conquer method is much better for large size array for general case.