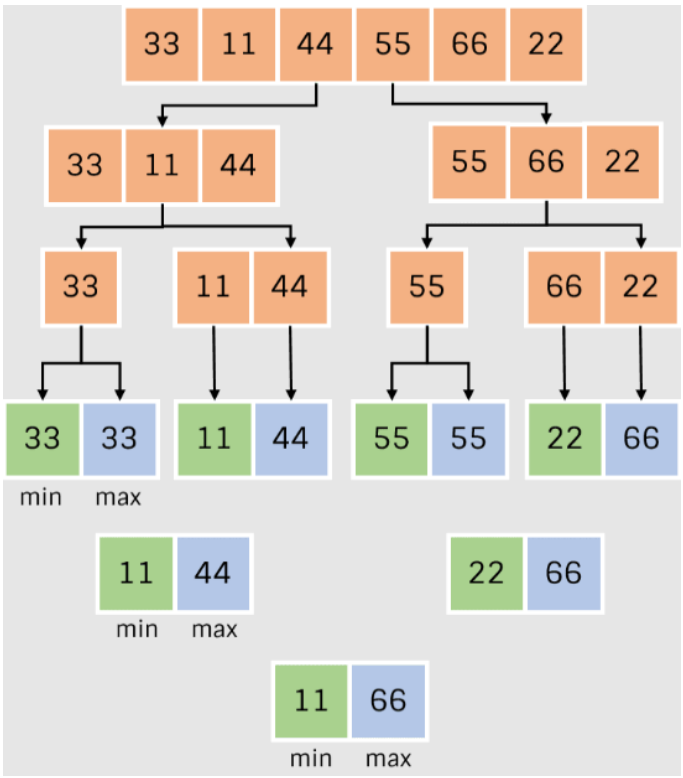
Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	03

AIM: **IMPLEMENT MIN MAX AND BINARY SEARCH USING DIVIDE AND CONQUER APPROACH**

THEORY:

- **MIN-MAX using DIVIDE & CONQUER APPROACH**
  - Divide: Divide array into two halves.
  - Conquer: Recursively find maximum and minimum of both halves.
  - Combine: Compare maximum of both halves to get overall maximum and compare minimum of both halves to get overall minimum.
  - Algorithm steps:
    Suppose function call minMax (X[], l, r) return maximum and minimum of the array, where l and r are the left and right end.
    - Divide array by calculating mid index i.e. mid = l + (r — l)/2
    - Recursively find the maximum and minimum of left part by calling the same function i.e. leftMinMax[2] = minMax(X, l, mid)
    - Recursively find the maximum and minimum for right part by calling the same function i.e. rightMinMax[2] = minMax(X, mid + 1, r)
    - Finally, get the overall maximum and minimum by comparing the min and max of both halves.
    - Store max and min in output[2] and return it.




```
if (leftMinMax[0] > rightMinMax[0])
    max = lminMax[0]
else
    max = rightMinMax[0]
if (leftMinMax[1] < rightMinMax[1])
    min = leftMinMax[1]
else
    min = rightMinMax[1]
```

- Base case 1: If the array size gets reduced to 1 during recursion, return that single element as both max and min.
- Base case 2: If the array size gets reduced to 2 during recursion, compare both elements and return maximum and minimum.
- Time and Space Complexities
  - The time complexity of the above solution is **O(n),** where n is the size of the input.
  - The auxiliary space required by the program is **O(n)** for recursion (call stack).

Code:

```
#include <stdio.h>
#include <stdlib.h>

#define n 10

int i, a[n];
int max, min;

void maxmin(int a[], int i, int j)
{
    int max1, min1, mid;
    if (i == j)
    {
        max = min = a[i];
    }
    else
    {
        if (i == j - 1)
        {
            if (a[i] < a[j])
            {
                max = a[j];
                min = a[i];
            }
            else
            {
                max = a[i];
                min = a[j];
            }
        }
        else
        {
            mid = (i + j) / 2;
            maxmin(a, i, mid);
            max1 = max;
```

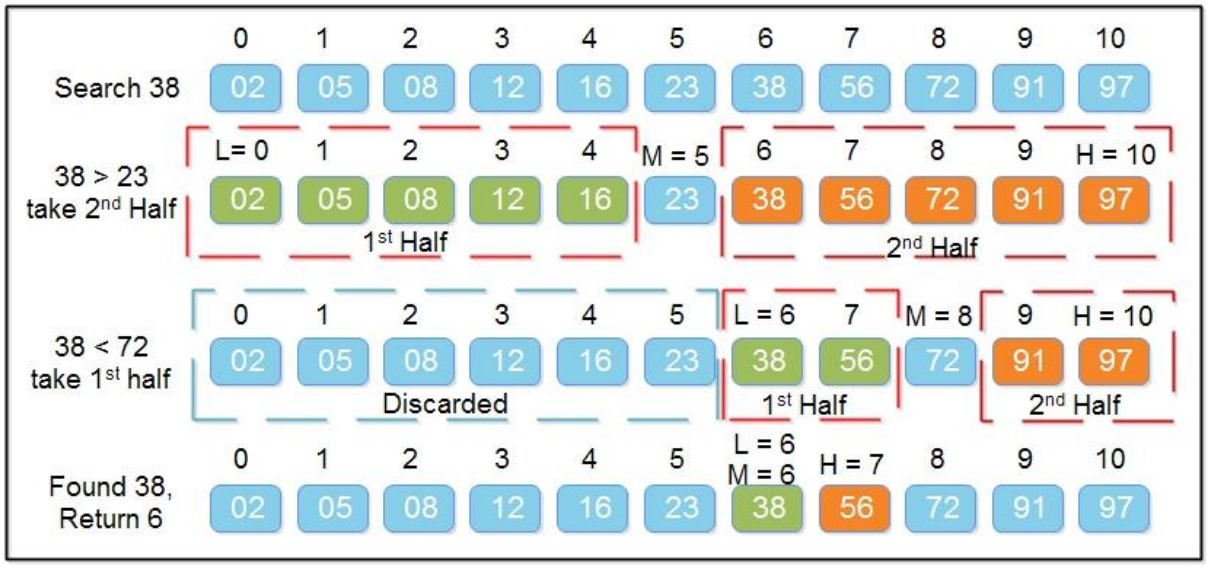
```
            min1 = min;
            maxmin(a, mid + 1, j);
            if (max < max1)
                max = max1;
            if (min > min1)
                min = min1;
        }
    }
}

int main()
{
    for (int i = 0; i < n; i++)
    {
        a[i] = rand();
    }

    for (int i = 0; i < n; i++)
    {
        printf("%d, ", a[i]);
    }

    max = a[0];
    min = a[0];

    maxmin(a, 0, n);
    printf("\nMinimum element in an array : %d\n", min);
    printf("Maximum element in an array : %d\n", max);

    return 0;
}
```

- Output:

```
n\gdb.exe' '--interpreter=mi'
41, 18467, 6334, 26500, 19169, 15724, 11478, 29358, 26962, 24464,
Minimum element in an array : 41
Maximum element in an array : 29358
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

BINARY SEARCH

- Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half.
- The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).



- Algorithm:

```
Sort the array in ascending order.
Set the low index to the first element of the array and the high index to the last element.
Set the middle index to the average of the low and high indices.
If the element at the middle index is the target element, return the middle index.
If the target element is less than the element at the middle index, set the high index to the middle index - 1.
If the target element is greater than the element at the middle index, set the low index to the middle index + 1.
Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.
```

- Time Complexity:
  - 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.
  - Binary search worst case differs from that. The worst-case scenario could be the values at either extremity of the list or values not in the list.

CODE:

```c
#include <stdio.h>
int binarySearch(int a[], int low, int high, int key){
    if (high >= low){
        int mid = low + (high-low)/2;
        if (key == a[mid]){
            return mid;
        }
        else if(a[mid] > key){
            binarySearch(a, low, mid-1, key);
        }
        else{
            binarySearch(a, mid+1, high, key);
        }
    }
    else{
        return -1;
    }
}
int main(){
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    int low = 0;
    int high = 9;
    printf("Enter any element: ");
    int num=0;
    scanf("%d", &num);
    int flag = binarySearch(a, low, high, num);
    if (flag == -1){
        printf("Element not found");
    }
    else{
        printf("Element found at %d index",flag);
    }
    return 0;
}
```

OUTPUT:

```
Enter any element: 4
Element found at 3 index
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>  & 'c:\Users\Jadhav\.vsc
```

```
Enter any element: 32
Element not found
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

A binary search algorithm has many benefits:

- Using each comparison eliminates half of the list that is not needed for further searches.
- This indicates whether the element to be searched is located before or after the current position within the list.
- This information can be used to limit your search.
- It works much better than linear searches for large data sets.