

**DEERWALK INSTITUTE OF TECHNOLOGY**

**Tribhuvan University**

**Faculties of Computer Science**



**Bachelors of Science in Computer Science and  
Information Technology (BSc. CSIT)**

**Course: Design and Analysis of Algorithms (CSC-314)**  
**Year/Semester: 3/5**

**A Lab report on:  
Divide & Conquer Algorithms (II)**

Submitted by:  
Name: Nishant Khadka  
Roll: 1017

Submitted to:  
Sujan Shrestha  
Department of Computer Science

Date: 19-Aug-2023

# BINARY SEARCH

## Theory

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element. Otherwise, the search is called unsuccessful.

Linear Search and Binary Search are the two popular searching techniques. Here we will discuss the Binary Search Algorithm.

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

### Time Complexity

Case	Time Complexity
Best Case	$O(1)$
Average Case	$O(\log n)$
Worst Case	$O(\log n)$

## Algorithm

**Step 1:** Start

**Step 2:** Initialize variables:  $m$ ,  $flag = 0$

**Step 3:** Check if  $l \leq r$ :

- i. Calculate  $m$  as  $(l + r) \div 2$
- ii. If  $x$  equals  $a[m]$ :
  1. Update  $flag$  with  $m$
- iii. Else if  $x < a[m]$ :
  1. Recursively perform binary search on the left subarray:  
 $binarysearch(a, l, m - 1, x)$
- iv. Else:
  1. Recursively perform binary search on the right subarray:  
 $binarysearch(a, m + 1, r, x)$

**Step 4:** Return  $flag$

**Step 5:** Stop

## Pseudocode

```
biniter(a,l,r, x)
{
    int m;
    int flag = 0
    if(l<=r)
    {
        m = (l + r ) ÷ 2
        if (x == a[m])
            flag = m;
        elseif (x<a[m])
```

```

        return binarysearch(a,l,m-1,x)
    else
        return binarysearch(a,m+1,r,x)
    }
    return flag;
}

```

## Source Code

```

#include <stdio.h>

int binary_search(int a[], int l, int r, int x) {
    int m = 0;
    int flag = 0;
    if (l <= r) {
        m = (l + r) / 2;
        if (x == a[m]) {
            flag = m;
        } else if (x < a[m]) {
            flag = binary_search(a, l, m - 1, x);
        } else {
            flag = binary_search(a, m + 1, r, x);
        }
    }
    return flag;
}

int main() {
    printf("Nishant Khadka\n");
    printf("Roll: 1017\n\n");

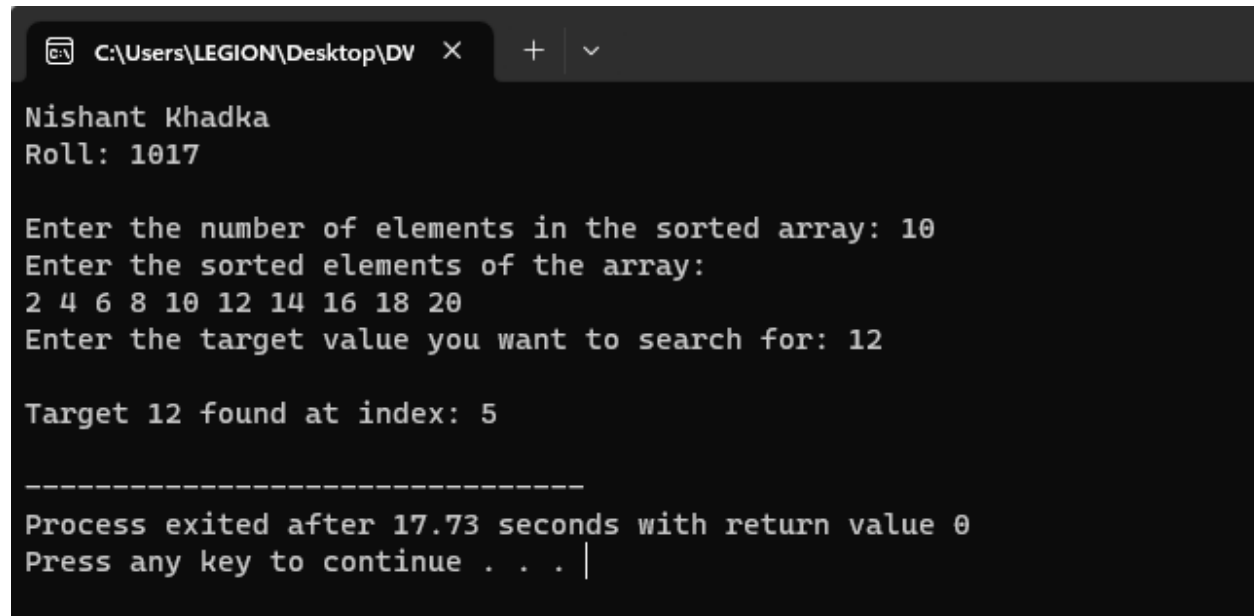
    int n;
    printf("Enter the number of elements in the sorted array: ");
    scanf("%d", &n);

    int sorted_array[n];
    printf("Enter the sorted elements of the array:\n");
}

```

```
for (int i = 0; i < n; i++) {  
    scanf("%d", &sorted_array[i]);  
}  
  
int target;  
printf("Enter the target value you want to search for: ");  
scanf("%d", &target);  
  
int result = binary_search(sorted_array, 0, n - 1, target);  
printf("\nTarget %d found at index: %d\n", target, result);  
  
return 0;  
}
```

## Output



```
C:\Users\LEGION\Desktop\DV X + v  
Nishant Khadka  
Roll: 1017  
  
Enter the number of elements in the sorted array: 10  
Enter the sorted elements of the array:  
2 4 6 8 10 12 14 16 18 20  
Enter the target value you want to search for: 12  
  
Target 12 found at index: 5  
  
-----  
Process exited after 17.73 seconds with return value 0  
Press any key to continue . . . |
```

## Discussion

The presented C program employs the binary search algorithm to efficiently locate a target value within a sorted array. User interaction is facilitated by input prompts for the number of elements in the sorted array, the array's elements themselves, and the desired target value for the search. The binary search algorithm systematically narrows down the search space, and the program outputs the index where the target value is found. By offering an interactive experience that demonstrates the practicality of binary search, this program underscores key programming concepts such as input handling, algorithm implementation, and structured output. This application provides a hands-on illustration of how binary search optimizes search operations in a sorted collection.

# MINMAX SEARCH

## Theory

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

**Time complexity:**  $O(b^d)$   $b$  is the branching factor and  $d$  is count of depth or ply of graph or tree.

**Space Complexity:**  $O(bd)$  where  $b$  is branching factor into  $d$  is maximum depth of tree similar to DFS.

## Algorithm

- Step 1:** Start MinMax( $l$ ,  $r$ ,  $max$ ,  $min$ )
- Step 2:** Initialize variables:  $max = min = A[l]$
- Step 3:** Check if  $l \leq r$ :
- Step 4:** Calculate mid as  $(l + r) \div 2$
- Step 5:** If  $x$  equals  $a[mid]$ :
  - a. Update  $max$  and  $min$  with  $a[mid]$

- Step 6:** Else if  $x < a[mid]$ :
- a. Recursively perform MinMax on the left subarray:  $\text{MinMax}(l, mid, max, min)$
- Step 7:** Else:
- a. Recursively perform MinMax on the right subarray:  $\text{MinMax}(mid + 1, r, max, min)$
- Step 8:** Return max and min
- Step 9:** Stop

## Pseudocode

Start

$\text{MinMax}(l, r, max, min)$

If  $l = r$ :

$max = min = A[l]$

Else If  $l = r - 1$ :

If  $A[l] < A[r]$ :

$max = A[r]$

$min = A[l]$

Else

$max = A[l]$

$min = A[r]$

Else

// Divide the problem

$mid = (l + r) / 2$

// Solve the subproblems

$\{min, max\} = \text{MinMax}(l, mid, max, min)$



```
{min1, max1} = MinMax(mid + 1, r, max1, min1)
```

```
// Combine the solutions
```

```
If max1 > max:
```

```
    max = max1
```

```
If min1 < min:
```

```
    min = min1
```

Stop

## Source Code

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

```
void MinMax(int l, int r, int A[], int *max, int *min) {
```

```
    if (l == r) {
```

```
        *max = *min = A[l];
```

```
    } else if (l == r - 1) {
```

```
        if (A[l] < A[r]) {
```

```
            *max = A[r];
```

```
            *min = A[l];
```

```
        } else {
```

```
            *max = A[l];
```

```
            *min = A[r];
```

```
        }
```

```
    } else {
```

```
        int mid = (l + r) / 2;
```

```
        int max1, min1;
```

```
        MinMax(l, mid, A, max, min);
```

```
        MinMax(mid + 1, r, A, &max1, &min1);
```

```
        if (max1 > *max) {
```

```
            *max = max1;
```

```
        }
```

```
        if (min1 < *min) {
```

```
        *min = min1;
    }
}

int main() {
    printf("Nishant Khadka\n");
    printf("Roll: 1017\n\n");

    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

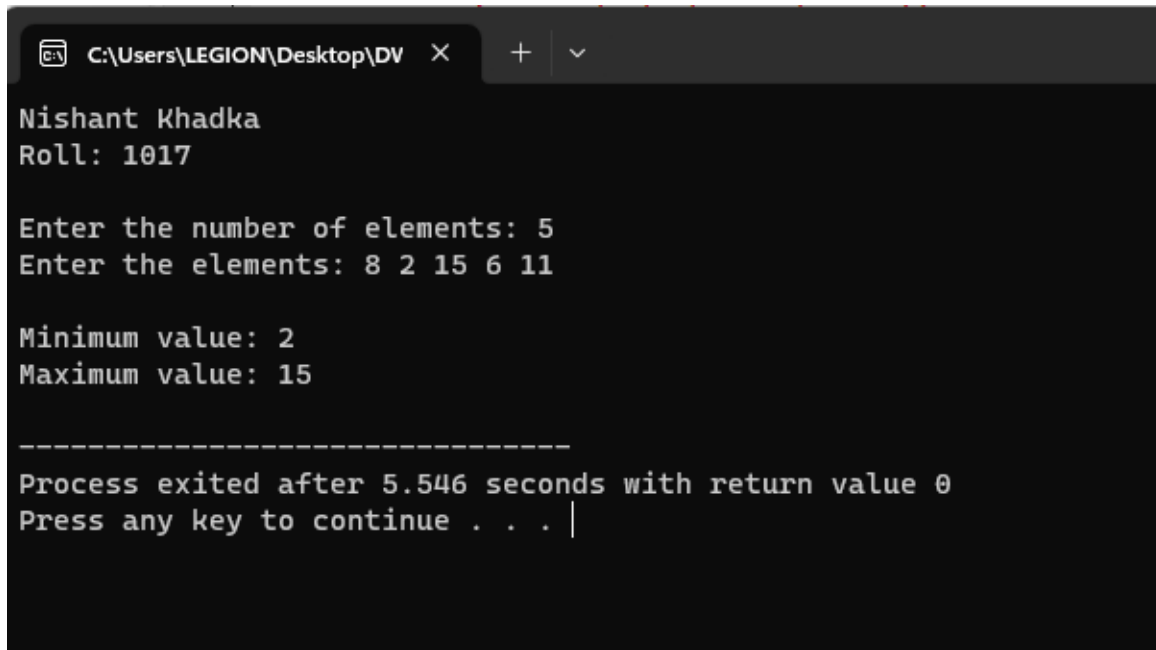
    int A[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &A[i]);
    }

    int max, min;
    MinMax(0, n - 1, A, &max, &min);

    printf("\nMinimum value: %d\n", min);
    printf("Maximum value: %d\n", max);

    return 0;
}
```

## Output

A screenshot of a Windows terminal window with a dark background. The window title bar shows the file path 'C:\Users\LEGION\Desktop\DV' and standard window controls. The terminal displays the output of a C program. It starts with the name 'Nishant Khadka' and roll number 'Roll: 1017'. Then it prompts for the number of elements, which is '5', and then the elements themselves, which are '8 2 15 6 11'. The program then outputs the 'Minimum value: 2' and 'Maximum value: 15'. A separator line of dashes follows. The final output shows the process exited after 5.546 seconds with a return value of 0, and a prompt to 'Press any key to continue' with a cursor on the line.

```
C:\Users\LEGION\Desktop\DV X + v
Nishant Khadka
Roll: 1017

Enter the number of elements: 5
Enter the elements: 8 2 15 6 11

Minimum value: 2
Maximum value: 15

-----
Process exited after 5.546 seconds with return value 0
Press any key to continue . . . |
```

## Discussion

The presented C program utilizes the MinMax algorithm to efficiently identify the minimum and maximum values within an array. User interaction is employed to input the number of elements in the array and the array's elements themselves. By employing a divide-and-conquer approach, the algorithm recursively dissects the problem into smaller segments, resolving the minimum and maximum values in each segment. The example output effectively showcases the program's capability to identify the minimum value (2) and maximum value (15) within an array of elements [8, 2, 15, 6, 11]. Through recursive division and analysis, the MinMax algorithm demonstrates its efficacy in solving the task of locating both the smallest and largest values within a collection.

# MERGE SORT

## Theory

Merge sort is the sorting technique that follows the divide and conquer approach. This article will be very helpful and interesting to students as they might face merge sort as a question in their examinations. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

### Time Complexity

Case	Time Complexity
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log n)$
Worst Case	$O(n \cdot \log n)$

## Algorithm

**Step 1:** Start

**Step 2:** mergeSort(arr[], left, right)

**Step 3:** If left < right:

i.  $\text{mid} = \text{left} + (\text{right} - \text{left}) / 2$

**Step 4:** // Sort the first and second halves

**Step 5:** Call mergeSort(arr, left, mid)

**Step 6:** Call mergeSort(arr, mid + 1, right)

**Step 7:** // Merge the sorted halves

**Step 8:** Call merge(arr, left, mid, right)

**Step 9:** merge(A[], left, mid, right)

**Step 10:** Initialize variables: i, j, k

**Step 11:** Calculate n1 as mid - left + 1

**Step 12:** Calculate n2 as right - mid

**Step 13:** Create temporary arrays: L[n1], R[n2]

**Step 14:** Copy data to temporary arrays:

**Step 15:** For i from 0 to n1 - 1:

i. Assign  $L[i] = A[\text{left} + i]$

**Step 16:** For j from 0 to n2 - 1:

i. Assign  $R[j] = A[\text{mid} + 1 + j]$

**Step 17:** Merge the temporary arrays back into A[l...r]:

**Step 18:** Initialize i as 0 // Initial index of first sub array

**Step 19:** Initialize j as 0 // Initial index of second sub array

**Step 20:** Initialize k as left // Initial index of merged sub array

**Step 21:** While  $i < n_1$  and  $j < n_2$ :

- i. If  $L[i] \leq R[j]$ :
  - 1. Assign  $A[k] = L[i]$
  - 2. Increment  $i$  by 1
- ii. Else:
  - 1. Assign  $A[k] = R[j]$
  - 2. Increment  $j$  by 1
- iii. Increment  $k$  by 1

**Step 22:** Copy the remaining elements of  $L[]$ , if any:

**Step 23:** While  $i < n_1$ :

- i. Assign  $A[k] = L[i]$
- ii. Increment  $i$  by 1
- iii. Increment  $k$  by 1

**Step 24:** Copy the remaining elements of  $R[]$ , if any:

**Step 25:** While  $j < n_2$ :

- i. Assign  $A[k] = R[j]$
- ii. Increment  $j$  by 1

**Step 26:** Increment  $k$  by 1

**Step 27:** Stop

## Pseudocode

```
mergeSort(arr[], left, right)
```

```
    if (left < right)
```

```
        mid = left + (right - left) / 2
```

```
        // Sort first and second halves
```

```
        mergeSort(arr, left, mid)
```

```
        mergeSort(arr, mid + 1, right)
```

```
        // Merge the sorted halves
```

```
        merge(arr, left, mid, right)
```

```
merge(A[], left, mid, right)
```

```
    i, j, k
```

```
    n1 = mid - left + 1
```

```
    n2 = right - mid
```

```
    // Create temporary arrays
```

```
    L[n1], R[n2]
```

```
    // Copy data to temporary arrays
```

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

```
        L[i] = A[left + i]
```

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

```
        R[j] = A[mid + 1 + j]
```

```
    // Merge the temporary arrays back into A[l...r]
```

```
    i = 0 // Initial index of first sub array
```

```
j = 0 // Initial index of second sub array
k = left // Initial index of merged sub array
while (i < n1 && j < n2)
    if (L[i] <= R[j])
        A[k] = L[i]
        i++
    else
        A[k] = R[j]
        j++
    k++
// Copy the remaining elements of L[], if any
while (i < n1)
    A[k] = L[i]
    i++
    k++
// Copy the remaining elements of R[], if any
while (j < n2)
    A[k] = R[j]
    j++
    k++
```



## Source Code

```
#include <stdio.h>

void merge(int A[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = A[left + i];
    for (j = 0; j < n2; j++)
        R[j] = A[mid + 1 + j];

    i = 0;
    j = 0;
    k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            A[k] = L[i];
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        A[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        A[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
```

```
        if (left < right) {
            int mid = left + (right - left) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

int main() {
    printf("Name: Nishant Khadka\n");
    printf("Roll: 1017\n\n");

    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

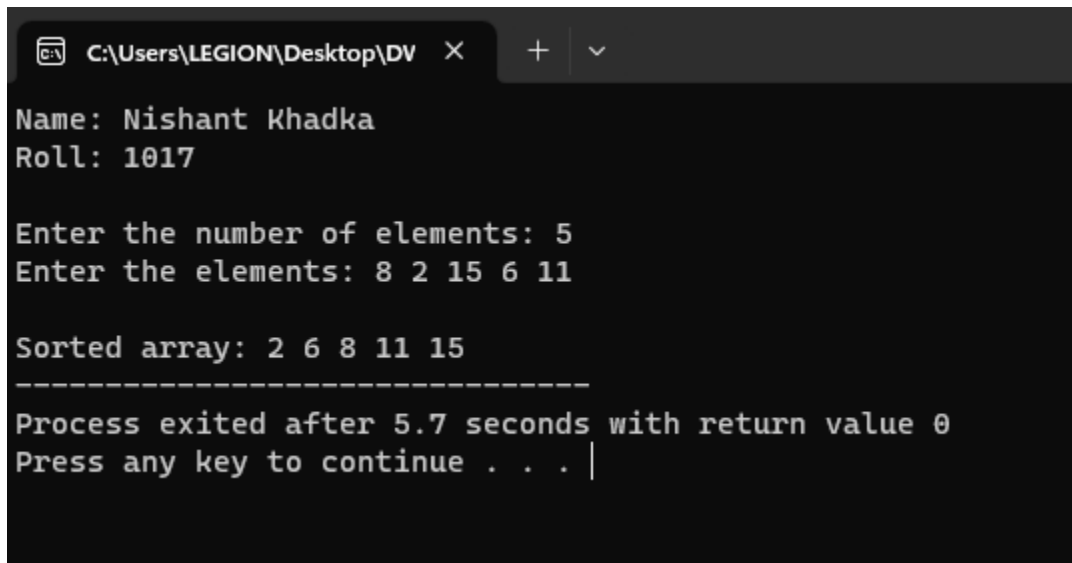
    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    mergeSort(arr, 0, n - 1);

    printf("\nSorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

## Output

A screenshot of a Windows command prompt window. The title bar shows the file path 'C:\Users\LEGION\Desktop\DV' and standard window controls. The terminal text is as follows:

```
Name: Nishant Khadka
Roll: 1017

Enter the number of elements: 5
Enter the elements: 8 2 15 6 11

Sorted array: 2 6 8 11 15
-----
Process exited after 5.7 seconds with return value 0
Press any key to continue . . . |
```

## Discussion

The presented C program employs the Merge Sort algorithm to efficiently sort an array of numbers. Through user input, the number of elements in the array and the array's elements are collected. The algorithm follows a divide-and-conquer strategy, breaking down the array into smaller segments, sorting them individually, and then merging them together to achieve a sorted overall array. The example output demonstrates the program's effectiveness in sorting an array of numbers, resulting in the sorted array [2, 6, 8, 11, 15]. By adhering to this systematic approach, the Merge Sort algorithm reliably demonstrates its capacity to sort arrays of different sizes.

# QUICK SORT

## Theory

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes  $n \log n$  comparisons in average case for sorting an array of  $n$  elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

**Divide:** In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

**Conquer:** Recursively, sort two subarrays with Quicksort.

**Combine:** Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element or the leftmost element of the given array.
- Select median as the pivot element.

## Algorithm

**Step 1:** Start  
**Step 2:** Quicksort(A, l, r)  
**Step 3:** if  $l < r$   
**Step 4:**  $p = \text{Partition}(A, l, r)$   
**Step 5:** Quicksort(A, l,  $p - 1$ )  
**Step 6:** Quicksort(A,  $p + 1$ , r)

**Step 7:** Partition(A, l, r)  
**Step 8:**  $x = l$   
**Step 9:**  $y = r$   
**Step 10:**  $p = A[l]$   
**Step 11:** while  $x < y$   
**Step 12:** while  $A[x] \leq p$   
    a.  $x++$   
**Step 13:** while  $A[y] \geq p$   
    a.  $y--$   
**Step 14:** if  $x < y$   
    a. swap( $A[x]$ ,  $A[y]$ )  
**Step 15:**  $A[l] = A[y]$   
**Step 16:**  $A[y] = p$   
**Step 17:** return y  
**Step 18:** Stop

## Pseudocode

Quicksort(A,l,r)

```
{  
    if(l<r)  
    {  
        p = Partition(A,l,r)  
        QuickSort(A,1,p-1)  
        Quicksort(A,p+1,r)  
    }  
}
```

Partition(A,l,r)

```
{  
    x=l;  
    y=r;  
    p=A[l]  
    while(x<y)  
    {  
        while(A[x]<=p)  
            x++  
        while(A[y]>=p)  
            y--  
        if(x<y)  
            swap(A[x],A[y])  
    }  
}
```

```
        A[l]=A[y]
        A[y]=p
        return y
    }
```

## Source Code

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int Partition(int A[], int l, int r) {
    int x = l;
    int y = r;
    int p = A[l];

    while (x < y) {
        while (A[x] <= p)
            x++;
        while (A[y] >= p)
            y--;
        if (x < y)
            swap(&A[x], &A[y]);
    }

    A[l] = A[y];
    A[y] = p;
    return y;
}

void Quicksort(int A[], int l, int r) {
    if (l < r) {
        int p = Partition(A, l, r);
        Quicksort(A, l, p - 1);
        Quicksort(A, p + 1, r);
    }
}
```

```
int main() {
    printf("Nishant Khadka\n");
    printf("Roll: 1017\n\n");

    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

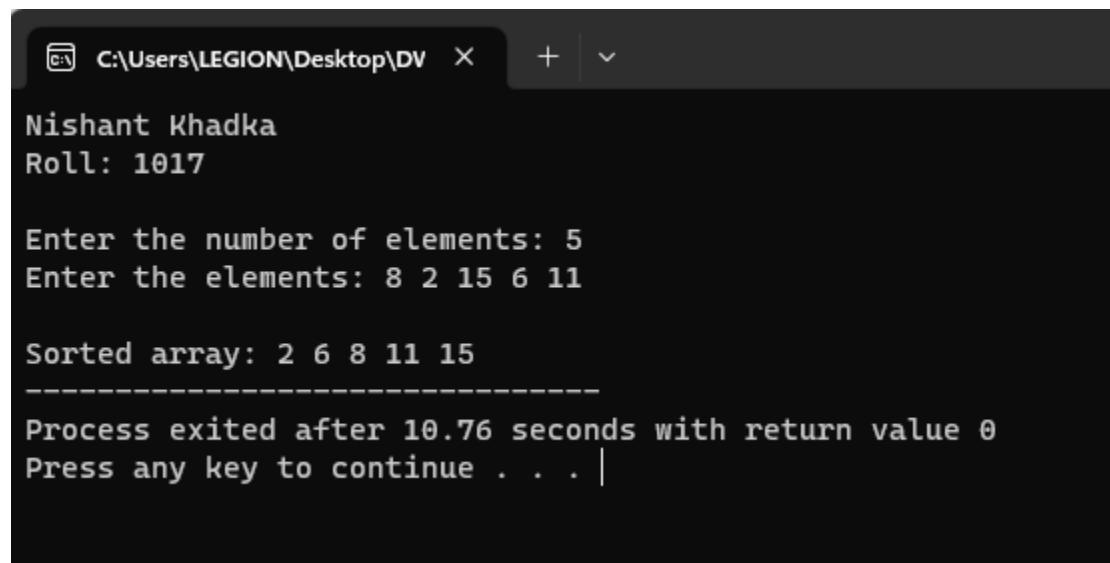
    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    Quicksort(arr, 0, n - 1);

    printf("\nSorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

## Output



```
C:\Users\LEGION\Desktop\DV X + v
Nishant Khadka
Roll: 1017

Enter the number of elements: 5
Enter the elements: 8 2 15 6 11

Sorted array: 2 6 8 11 15
-----
Process exited after 10.76 seconds with return value 0
Press any key to continue . . . |
```



## Discussion

The provided C program demonstrates the application of the QuickSort algorithm in efficiently sorting an array of integers. Through user interaction, the program collects the number of elements in the array and the array's elements. The algorithm employs a pivot element for partitioning the array into smaller segments, recursively sorts these segments, and then combines them to generate a fully sorted array. The example output illustrates the program's effectiveness in sorting an array of numbers [8, 2, 15, 6, 11], resulting in the sorted array [2, 6, 8, 11, 15]. Adhering to the QuickSort methodology, the program showcases its ability to handle arrays of various sizes and efficiently sort their contents.

# HEAP SORT

## Theory

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1<sup>st</sup> phase.

Before knowing more about the heap sort, let's first see a brief description of **Heap**.

Heap

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

Heap sort

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Heapsort is the in-place sorting algorithm.

## Algorithm

- Step 1:** Start
- Step 2:** Heapsort(A, n)
- Step 3:** for  $i = n/2$  down to 1  
    i. Maxheapify(A, n, i)
- Step 4:** for  $i = n$  down to 1  
    i. swap(A[1], A[i])  
    ii. Maxheapify(A, i - 1, 1)
- 
- Step 5:** Maxheapify(A, n, i)
- Step 6:** largest = i
- Step 7:** l = 2 \* i
- Step 8:** r = 2 \* i + 1
- Step 9:** if  $l \leq n$  and  $A[l] > A[\text{largest}]$   
    i. largest = l
- Step 10:** if  $r \leq n$  and  $A[r] > A[\text{largest}]$   
    i. largest = r
- Step 11:** if largest != i  
    i. swap(A[largest], A[i])  
    ii. Maxheapify(A, n, largest)
- Step 12:** Stop

## Pseudocode

# Input: Array A

# Output: Sorted array A

Heapsort(A, n)

{

    for( $i=n/2$ ;  $i \geq 1$ ;  $i--$ )

    {

```

        Maxheapify(A, n, i)
    }
    for(i=n; i>=1; i--)
    {
        swap(A[1],A[i])
        Maxheapify(A,i-1,1)
    }
}

```

```

Maxheapify(A,n,i)
{
    largest = i
    l = 2*i
    r = 2*i + 1
    if(l<=n && A[l] > A[largest])
        largest = l
    if(r<=n && A[r] > A[largest])
        largest = r
    if(largest !=i)
    {
        swap(A[largest],A[i])
        Maxheapify(A,n,largest)
    }
}

```

## Source Code

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void Maxheapify(int A[], int n, int i) {
    int largest = i;
    int l = 2 * i;
    int r = 2 * i + 1;

    if (l <= n && A[l] > A[largest])
        largest = l;
    if (r <= n && A[r] > A[largest])
        largest = r;
    if (largest != i) {
        swap(&A[largest], &A[i]);
        Maxheapify(A, n, largest);
    }
}

void Heapsort(int A[], int n) {
    for (int i = n / 2; i >= 1; i--) {
        Maxheapify(A, n, i);
    }
    for (int i = n; i >= 1; i--) {
        swap(&A[1], &A[i]);
        Maxheapify(A, i - 1, 1);
    }
}

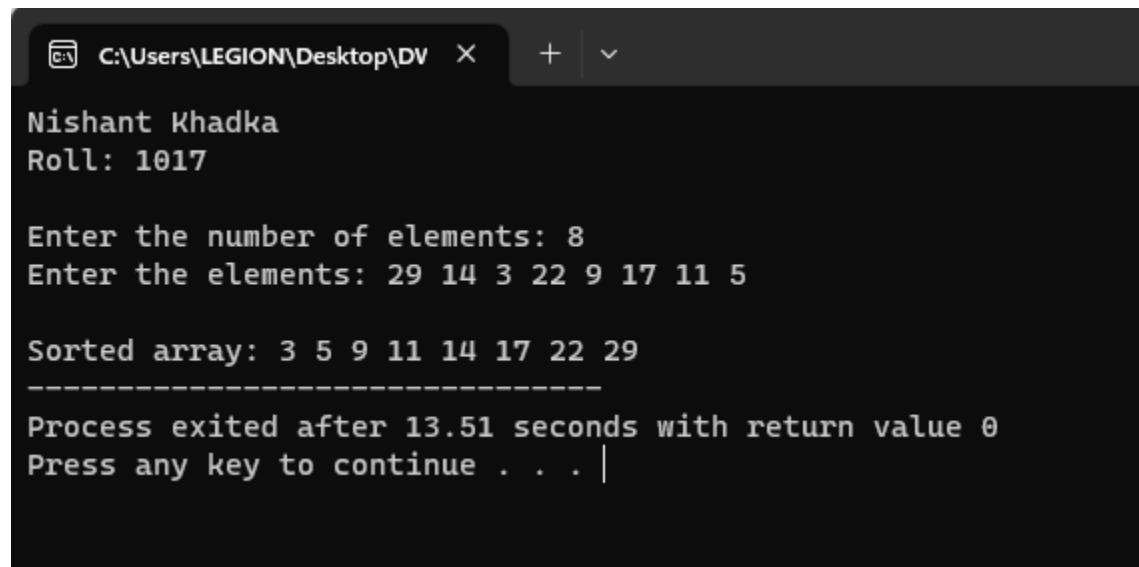
int main() {
    printf("Nishant Khadka\n");
    printf("Roll: 1017\n\n");

    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n + 1]; // Using 1-based indexing for HeapSort
    printf("Enter the elements: ");
```

```
for (int i = 1; i <= n; i++) {  
    scanf("%d", &arr[i]);  
}  
  
Heapsort(arr, n);  
  
printf("\nSorted array: ");  
for (int i = 1; i <= n; i++) {  
    printf("%d ", arr[i]);  
}  
  
return 0;  
}
```

## Output



```
C:\Users\LEGION\Desktop\DV X + v  
Nishant Khadka  
Roll: 1017  
  
Enter the number of elements: 8  
Enter the elements: 29 14 3 22 9 17 11 5  
  
Sorted array: 3 5 9 11 14 17 22 29  
-----  
Process exited after 13.51 seconds with return value 0  
Press any key to continue . . . |
```

## Discussion

The presented C program showcases the HeapSort algorithm's effectiveness in sorting an array of integers. User interaction is utilized to collect the number of elements in the array and the array's elements. The algorithm transforms the array into a max-heap, systematically sorts the heap, and then reverts it back into a fully

sorted array. The example output demonstrates the program's proficiency in sorting an array of numbers [29, 14, 3, 22, 9, 17, 11, 5], ultimately resulting in the sorted array [3, 5, 9, 11, 14, 17, 22, 29]. Through the HeapSort approach, the program effectively manages arrays of varying sizes and efficiently arranges their contents. The process showcases the algorithm's ability to sort numbers while adhering to the principles of max-heap construction and manipulation.

## **CONCLUSION**

The provided algorithms offer efficient solutions for sorting and searching. Binary Search narrows down the search range by dividing the sorted array, reducing search time logarithmically. Merge Sort employs a divide-and-conquer strategy, breaking an array into smaller parts, ensuring they're sorted, and then merging them for a fully sorted array. QuickSort selects a pivot, partitions the array, and recursively sorts subarrays. HeapSort creates a max-heap, sorts the array using heap properties, yielding a sorted array. These algorithms address different sorting and searching needs, providing stable and consistent solutions for handling varying data sizes efficiently.