

# **SYSTEM PROGRAMMING LAB**

## **LAB PRACTICALS RECORD**

**(CSX - 326)**

## **COMPUTER SCIENCE AND ENGINEERING**



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING  
Dr. B R AMBEDKAR NATIONAL INSTITUTE OF TECHNOLOGY  
JALANDHAR – 144011, PUNJAB (INDIA)**

### **Submitted To:**

Ms. Rupali

Asst. Professor

Department of CSE

### **Submitted By:**

Aman Garg

13103050

6<sup>th</sup> Semester

**PROGRAM 1****SEARCHING- LINEAR | BINARY****Description:**

The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished. If the target value is less than the middle element's value, then the search continues on the lower half of the array; or if the target value is greater than the middle element's value, then the search continues on the upper half of the array. This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (and its associated element position is returned), or until the entire array has been searched (and "not found" is returned).

**Program:**

```
#include <bits/stdc++.h>
#include <iostream>
using namespace std;

void linearSearch(vector<int> & input, const int & key){
    // Given a key and an array, it linearly searches for the input key
    vector<int> :: iterator it = find(input.begin(), input.end(), key);
    if (it == input.end()) //we have reached end of the iterator
        cout << " Linear search couldn't locate the key: "<< key << endl;
    else
        cout << " Linear search located the key: "<< key << " at: "<<int(it - input.begin())<<endl;
}

void binarySearch(const vector<int> & input, const int & key){
    // Given a key and a sorted array, it searches for the input key by dividing into intervals
    vector<int> newInput (input);
    sort(newInput.begin(), newInput.end());

    int l = 0, r = newInput.size() - 1, mid;
    bool foundState = false;

    while (l <= r){
        mid = (l + r)/2 ;

        if (newInput[mid] == key){
            foundState = true;
            break;
        }
    }
```

```
        else if (newInput [mid] > key) //Key < [mid]. So, move to left interval
            r = mid - 1;
        else //Key > [mid]. So, move to right interval
            l = mid + 1;
    }

    if (foundState == true)
        cout << " Binary search located the key: "<< key << endl;
    else
        cout << " Binary search couldn't locate the key: "<< key << endl;
}

int main(){

    // Read test data
    ifstream inf("testFile");
    if (!inf){

        fprintf(stderr, "\nError opening test file\n");
        return -1;
    }

    vector<int> input;
    int searchKey;
    char c;

    while( (c = inf.get()) != EOF)
        input.push_back(int(c));

    inf.close();

    for (auto elem : input)
        cout << elem << " ";

    while (true){
        cout << "\n Enter search key: (-10 to quit) ";
        cin >> searchKey;
        if (searchKey == -10) break;

        linearSearch(input, searchKey);
        binarySearch(input, searchKey);
    }

    return 0;
}
```

```
49 32 51 48 32 55 56 32 50 32 49 48 32 51 57 56 51 32 50 50 32 45 49 48 32 45
2 56 57 32 53 54 32 56 57 32 55 56 32 55 55 32 49 50 56 10
Enter search key: (-10 to quit) 80
Linear search couldn't locate the key: 80
Binary search couldn't locate the key: 80

Enter search key: (-10 to quit) 41
Linear search couldn't locate the key: 41
Binary search couldn't locate the key: 41

Enter search key: (-10 to quit) 49
Linear search located the key: 49 at: 0
Binary search located the key: 49

Enter search key: (-10 to quit) 32
Linear search located the key: 32 at: 1
Binary search located the key: 32

Enter search key: (-10 to quit) 51
Linear search located the key: 51 at: 2
Binary search located the key: 51

Enter search key: (-10 to quit) 71
Linear search couldn't locate the key: 71
Binary search couldn't locate the key: 71

Enter search key: (-10 to quit) 56
Linear search located the key: 56 at: 6
Binary search located the key: 56

Enter search key: (-10 to quit) -10
aman@aman ~/Desktop/prog/Systems Prog/1-18 $
```

**PROGRAM 2****QUICKSORT | MERGESORT****Description:**

Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires  $O(N)$  extra storage,  $N$  denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have  $O(N\log N)$  average complexity but the constants differ. For arrays, merge sort loses due to the use of extra  $O(N)$  storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of  $O(n\log n)$ . The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

**Program (MERGESORT) :**

```
#include <bits/stdc++.h>
using namespace std;

void merge(vector<int> & arr, int low, int mid, int high){
    int i = low, j = mid + 1, k = low;
    vector<int> c(100);

    while (i <= mid && j <= high){
        if (arr[i] < arr[j]){
            c[k] = arr[i];
            k++; i++;
        }
        else{
            c[k] = arr[j];
            k++, j++;
        }
    }
    while (i <= mid){
        c[k] = arr[i];
        k++; i++;
    }
    while (j <= high){
        c[k] = arr[j];
        k++, j++;
    }
    for(i = low; i < k; i++)
        arr[i] = c[i];
}
```

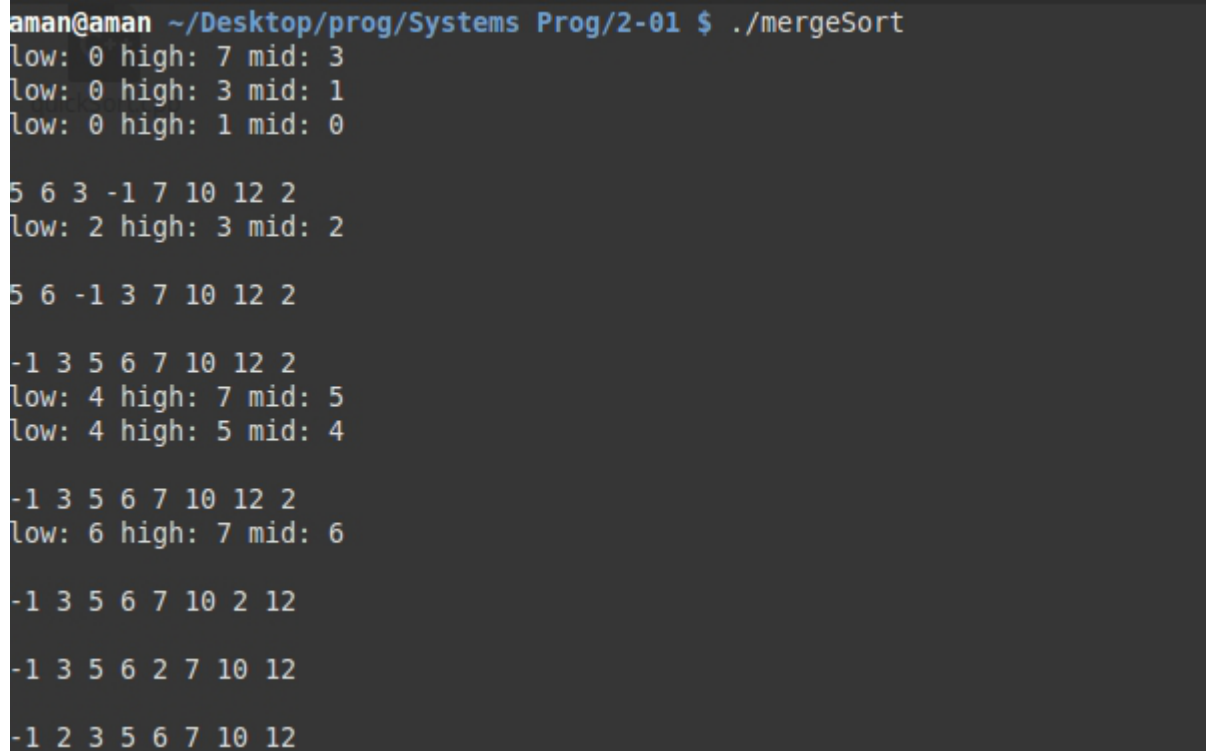
```
std::cout << endl;
for(auto a: arr)
    cout << a << " ";
cout << endl;
}
void mergeSort(vector<int> &arr, int low, int high){

    if (low < high){
        int mid = (low +high) /2;
        printf("low: %d high: %d mid: %d\n",low, high, mid );

        mergeSort(arr, low, mid);
        mergeSort(arr, mid +1, high);
        merge(arr, low, mid, high);
    }
    return;
}

int main(){

    vector<int> arr {6, 5, 3, -1, 7, 10, 12, 2};
    mergeSort(arr, 0, arr.size() -1);
    return 0;
}
```



```
aman@aman ~/Desktop/prog/Systems Prog/2-01 $ ./mergeSort
low: 0 high: 7 mid: 3
low: 0 high: 3 mid: 1
low: 0 high: 1 mid: 0

5 6 3 -1 7 10 12 2
low: 2 high: 3 mid: 2

5 6 -1 3 7 10 12 2

-1 3 5 6 7 10 12 2
low: 4 high: 7 mid: 5
low: 4 high: 5 mid: 4

-1 3 5 6 7 10 12 2
low: 6 high: 7 mid: 6

-1 3 5 6 7 10 2 12

-1 3 5 6 2 7 10 12

-1 2 3 5 6 7 10 12
```

**Program (QUICKSORT) :**

```
#include <bits/stdc++.h>
using namespace std;

int partition(vector<int> &A, int low, int high){
    int pivot = A[high];
    int pivotIndex = low;

    for (int i = low; i < high; i++){
        if (A[i] <= pivot){
            swap(A[i], A[pivotIndex]);
            pivotIndex ++;
        }
    }
    swap(A[high], A[pivotIndex]);
    for (auto a : A)
        cout << a <<" ";
    cout << endl <<endl;
    return pivotIndex;
}

void quickSort(vector<int> &A, int low, int high){
    if (low < high){
        printf("quick (%d, %d)\n", low, high);
        int pivotIndex = partition(A, low, high);
        quickSort(A, low, pivotIndex -1);
        quickSort(A, pivotIndex +1, high);
    }
}

int main(){

    vector<int> A {6, 5, 3, -1, 7, 10, 12, 2};

    cout << endl;
    quickSort(A, 0, A.size() -1);
    for (auto a : A)
        cout << a <<" ";
    cout << endl;
```

```
    return 0;  
}
```

```
aman@aman ~/Desktop/prog/Systems Prog/2-01 $ ./quickSort  
pivotIndex ++;  
quick (0, 7)  
-1 2 3 6 7 10 12 5  
  
quick (2, 7)  
-1 2 3 5 7 10 12 6  
  
a < pivotIndex;  
quick (4, 7)  
-1 2 3 5 6 10 12 7  
pivotIndex;  
  
quick (5, 7)  
-1 2 3 5 6 7 12 10  
  
quick (6, 7)  
-1 2 3 5 6 7 10 12  
if (a[low] < a[pivotIndex]) {  
    swap(a[low], a[pivotIndex]);  
    low++;  
}  
if (a[high] > a[pivotIndex]) {  
    swap(a[high], a[pivotIndex]);  
    high--;  
}  
if (low < high) {  
    quickSort(a, low, high);  
}  
printf("Sorted Array: ");  
for (i = 0; i < 8; i++) {  
    printf("%d ", a[i]);  
}  
printf("\n");  
}
```



## PROGRAM 3

### BUCKETSORT | HEAPSORT

**Description:**

Bucket sort, or bin sort, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a distribution sort, and is a cousin of radix sort in the most to least significant digit flavour. Bucket sort is a generalization of pigeonhole sort. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm. The computational complexity estimates involve the number of buckets.

Bucket sort works as follows:

1. Set up an array of initially empty "buckets".
2. Scatter: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. Gather: Visit the buckets in order and put all elements back into the original array.

**Program (BUCKETSORT) :**

```
#include <bits/stdc++.h>
using namespace std;

void bucketSort(vector<int> &A){
    int n = A.size();
    int minm = (*min_element(A.begin(), A.end()) / 10) * 10;
    int maxm = (*max_element(A.begin(), A.end()) / 10) * 10 + 10;
    int rangeM = (maxm - minm)/10;
    vector<vector<int> > buckets(rangeM);

    for(int i = 0; i < n; i++){
        int c = (A[i] / 10);
        buckets[c].push_back(A[i]);
    }

    for (int i = 0; i < buckets.size(); i++)
        sort(buckets[i].begin(), buckets[i].end());

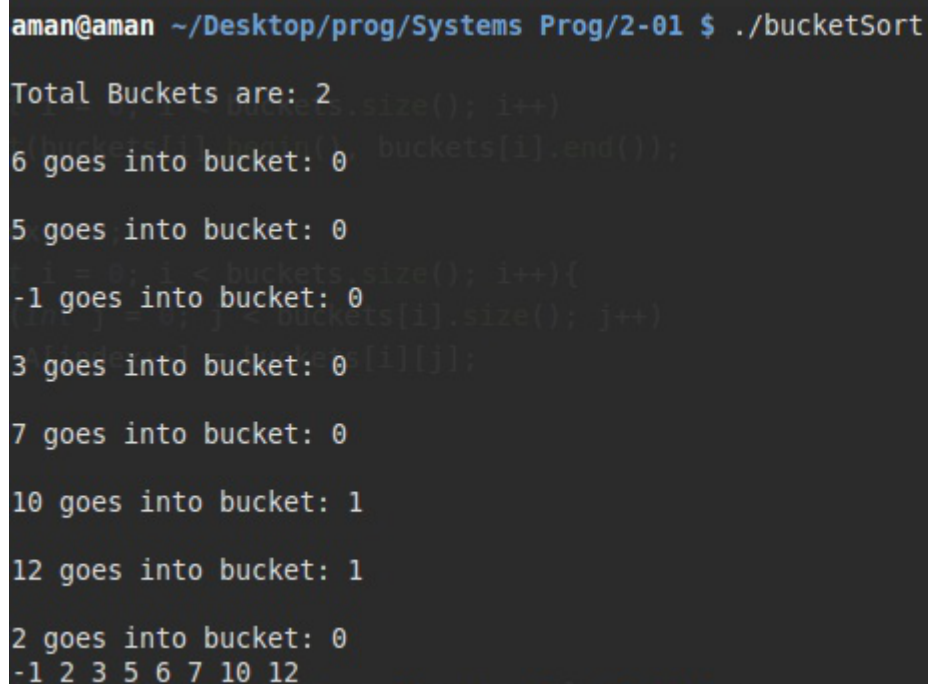
    int index = 0;
    for (int i = 0; i < buckets.size(); i++){
        for(int j = 0; j < buckets[i].size(); j++)
```

```
        A[index++] = buckets[i][j];
    }
}

int main(){
    vector<int> A {6, 5, -1, 3, 7, 10, 12, 2};
    bucketSort(A);

    for(auto a : A)
        cout << a << " ";
    cout << endl;

    return 0;
}
```



```
aman@aman ~/Desktop/prog/Systems Prog/2-01 $ ./bucketSort
Total Buckets are: 2
6 goes into bucket: 0
5 goes into bucket: 0
-1 goes into bucket: 0
3 goes into bucket: 0
7 goes into bucket: 0
10 goes into bucket: 1
12 goes into bucket: 1
2 goes into bucket: 0
-1 2 3 5 6 7 10 12
```

### Description (HeapSort):

Heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. The improvement consists of the use of a heap data structure rather than a linear-time search to find the maximum. Although somewhat slower in practice on most machines than a well-

implemented quicksort, it has the advantage of a more favorable worst-case  $O(n \log n)$  runtime.

Heapsort is an in-place algorithm, but it is not a stable sort.

**Program (HEAPSORT) :**

```
#include <bits/stdc++.h>
using namespace std;

int temp;

int left(int i){
    // get left child 2i + 1
    return 2*i + 1;
}

int right(int i){
    // get right child 2*i + 2
    return 2*i + 2;
}

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

void maxHeapify(vector<int> & arr, int i, int & heapSize){
    cout << "\n Max heapify called for index: "<<i<<" and HS: "<<heapSize<<endl;
    for (auto a : arr)
        cout << a << " ";
    cout << endl;

    int l = left(i), r = right(i);

    int largest = i;

    if (l < heapSize && arr[l] > arr[i])
        largest = l;
    if (r < heapSize && arr[r] > arr[largest])
        largest = r;
    if (largest == i) //we are fine. no heapify needed
        return;

    // else swap arr[i] with arr[largest]
    swap(&arr[largest], &arr[i]);
    maxHeapify(arr, largest, heapSize); }
```

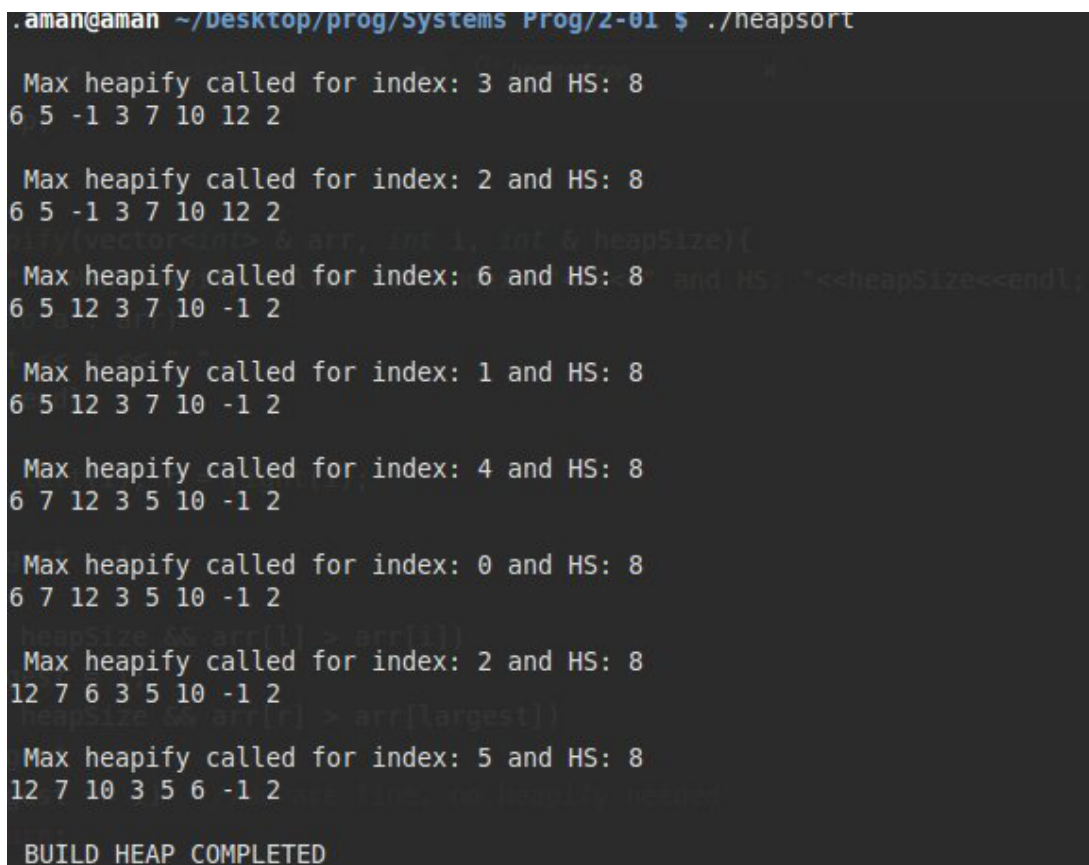
```
void buildHeap(vector<int> & arr, int & heapSize){
    int mid = (arr.size() -1) /2;
    for (int i = mid; i >= 0; i --)
        maxHeapify(arr, i, heapSize );
    cout <<"\n BUILD HEAP COMPLETED\n";
}
```

```
void heapSort(vector<int> &arr){
    int heapSize = arr.size();
    buildHeap(arr, heapSize);

    for (int i = arr.size() -1; i >= 0; i--){
        swap(&arr[0], &arr[i]);
        heapSize -= 1;
        maxHeapify(arr, 0, heapSize);
    }
}
```

```
int main(){
    vector<int> arr {6, 5, -1, 3, 7, 10, 12, 2};
    heapSort(arr);

    return 0;
}
```



```
.aman@aman ~/Desktop/prog/Systems Prog/2-01 $ ./heapsort

Max heapify called for index: 3 and HS: 8
6 5 -1 3 7 10 12 2

Max heapify called for index: 2 and HS: 8
6 5 -1 3 7 10 12 2

Max heapify called for index: 6 and HS: 8
6 5 12 3 7 10 -1 2

Max heapify called for index: 1 and HS: 8
6 5 12 3 7 10 -1 2

Max heapify called for index: 4 and HS: 8
6 7 12 3 5 10 -1 2

Max heapify called for index: 0 and HS: 8
6 7 12 3 5 10 -1 2

Max heapify called for index: 2 and HS: 8
12 7 6 3 5 10 -1 2

Max heapify called for index: 5 and HS: 8
12 7 10 3 5 6 -1 2

BUILD HEAP COMPLETED
```

```
Max heapify called for index: 0 and HS: 6  
-1 7 6 3 5 2 10 12
```

```
Max heapify called for index: 1 and HS: 6  
7 -1 6 3 5 2 10 12
```

```
Max heapify called for index: 4 and HS: 6  
7 5 6 3 -1 2 10 12
```

```
Max heapify called for index: 0 and HS: 5  
2 5 6 3 -1 7 10 12
```

```
Max heapify called for index: 2 and HS: 5  
6 5 2 3 -1 7 10 12
```

```
Max heapify called for index: 0 and HS: 4  
-1 5 2 3 6 7 10 12
```

```
Max heapify called for index: 1 and HS: 4  
5 -1 2 3 6 7 10 12
```

```
Max heapify called for index: 3 and HS: 4  
5 3 2 -1 6 7 10 12
```

```
Max heapify called for index: 0 and HS: 3  
-1 3 2 5 6 7 10 12
```

```
Max heapify called for index: 1 and HS: 3  
3 -1 2 5 6 7 10 12
```

```
Max heapify called for index: 0 and HS: 2  
2 -1 3 5 6 7 10 12
```

```
Max heapify called for index: 0 and HS: 1  
-1 2 3 5 6 7 10 12
```

```
Max heapify called for index: 0 and HS: 0  
-1 2 3 5 6 7 10 12
```