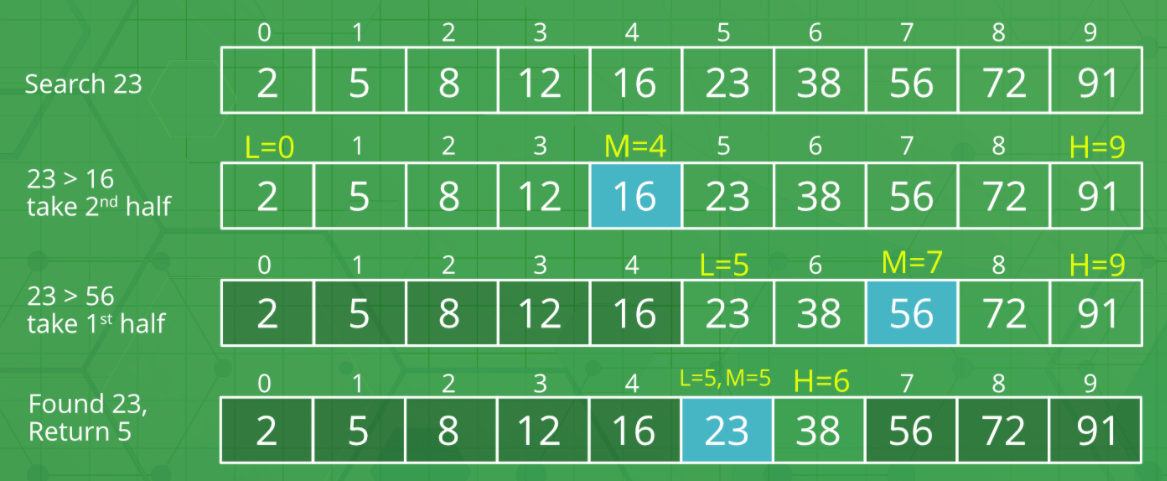
# CIS7 Unit 7 Lab: Search and Sort in C++

## Binary Search Basics

Search a sorted array by **repeatedly dividing the search interval in half**. Begin with an interval covering the whole array. **If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half**. **Otherwise, narrow it to the upper half**. Repeatedly check until the value is found or the interval is empty.



* **Iteration 1:**

Array: 2, 5, 8, 12, 16, 23, 38, 56, 72, 91

* + Select the middle element. (**here 16**)
  + Since 23 is greater than 16, so we divide the array into two halves and consider the sub-array after element 16.
  + Now this subarray with the elements after 16 will be taken into next iteration.
  + At Iteration 1, length of array = n
* **Iteration 2:**

Array: 23, 38, 56, 72, 91

* + Select the middle element. (**now 56**)
  + Since 23 is smaller than 56, so we divide the array into two halves and consider the sub-array before element 56.
  + Now this subarray with the elements before 56 will be taken into next iteration.
  + At Iteration 2, length of array = n⁄2
* **Iteration 3:**

Array: 23, 38

* + Select the middle element. (**now 23**)
  + Since 23 is the middle element. So the iterations will now stop.

At Iteration 3, length of array = (n⁄2)⁄2 = n⁄22

Therefore, after Iteration k, length of array = n⁄2k

After k divisions, the length of array becomes 1

Therefore, length of array = n⁄2k = 1 => n = 2k

Applying log function on both sides:

=> log2 (n) = log2 (2k)

=> log2 (n) = k log2 (2)

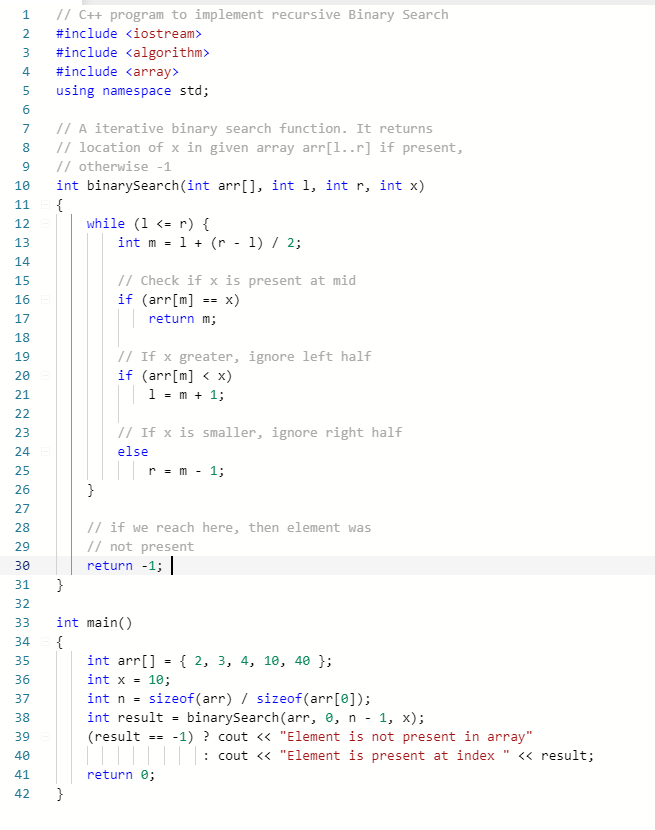
As (loga (a) = 1)

=> k = log2 (n)

**T(n) = log2 (n)**

## Binary Search: Iterative implementation

**Example 1**: Binary Search Index Of Array



The above program consists of the follow procedures:

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So it seeks in the right half.
4. Else (x is smaller), it seeks in the left half.
5. Write and run example 1 program. Provide screen capture of code and output.

A screenshot of a computer

Description automatically generated

1. Determine the number of comparisons of 5 element array using binary search.

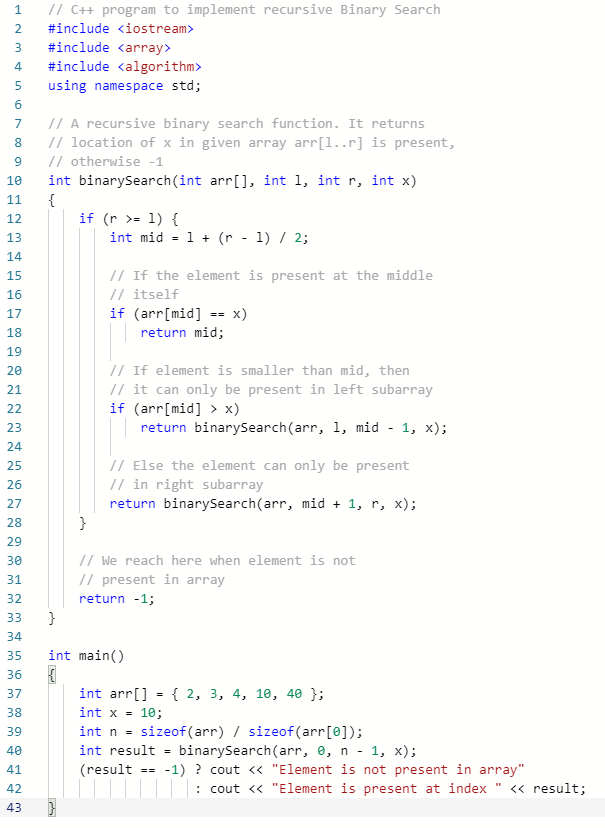
T(n) = Ln(5) / ln(2) = 2 comparisons

1. Explain the binary search processes given { 2, 3, 4, 10, 40 }, seeking 10. Explain the functionality of the program and how the algorithm is implemented.

Middle value (4) is selected and left and right are 2 and 40 respectively. 10 is greater than 4 so we set left to be 10 and right remains 40. Middle is now 10 and we check if middle = 10. Since middle = 10 we know 10 is in the array and its index is middle.

## Binary Search: Recursive Implementation

**Example 2**: Binary Search using recursion



The above program contains the following procedures:

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.
5. Write and run Example 2 in IDE. Provide screen capture of code and output.

A screenshot of a computer

Description automatically generated

1. Describe the main functionality of the program and how Binary Search is implemented in the program.

First the program checks if the right index is greater than or equal to the left index. If so it checks if the middle index is equal to x. If so the middle index is returned. If the array at middle index if greater than x we recursively call the function again but with the right index being equal to 1 index less than the middle. If the array at middle index is less than x then we recursively call the function with the left index being set to middle index + 1. This continues until either r is no longer greater than or equal to l until the array at the middle index = x;

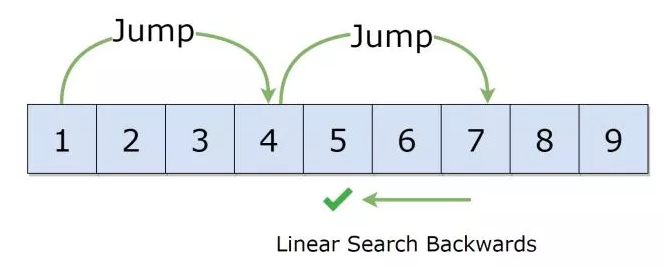
1. Compare Example 2 to Example 1 program. Which program is more efficient? Explain your answer.

The program in Example 1 is more efficient because recursion requires extra space to store the additional stack frames for each recursive call compared to the iterative approach in Example 1 which doesn’t require any additional space making it more efficient.

## Jump Search

Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

Instead of searching one-by-one, we search k-by-k. Let’s say we have a sorted array A, then jump search will look at A[1], A[1 + k], A[1 + 2k], A[1 + 3k] … and so on.



For example, suppose we have an array arr[] of size n and block (to be jumped) size m. Then we search at the indexes arr[0], arr[m], arr[2m]…..arr[km] and so on. Once we find the interval (arr[km] < x < arr[(k+1)m]), we perform a linear search operation from the index km to find the element x.

Optimal Jump size:

**C = √n**

In the below program, using array: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610}. Length of the array is 16. Jump search will find the value of 55 with the following steps assuming that the block size to be jumped is 4.

STEP 1: Jump from index 0 to index 4;

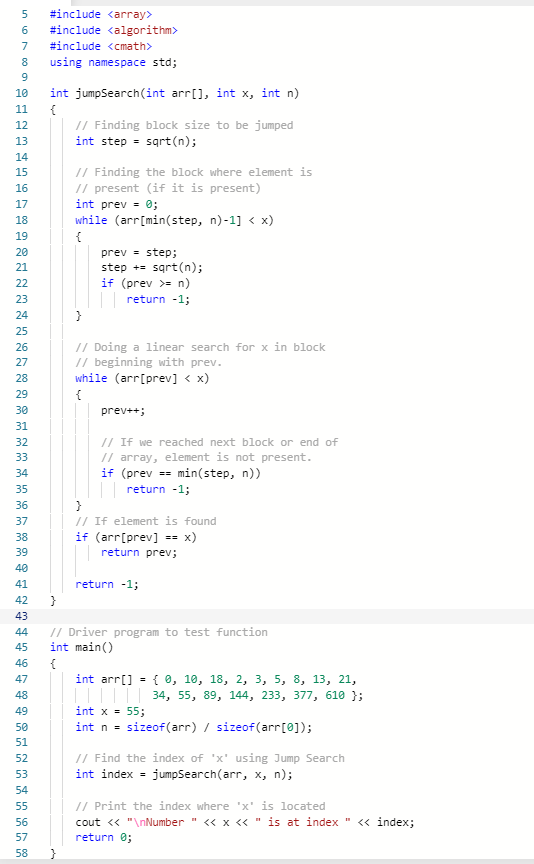
STEP 2: Jump from index 4 to index 8;

STEP 3: Jump from index 8 to index 12;

STEP 4: Since the element at index 12 is greater than 55 we will jump back a step to come to index 8

STEP 5: Perform linear search from index 8 to get the element 55.

**Example 3**: Jump Search



1. Write and run Example 3 in IDE. Provide screen capture of code and output.

A screenshot of a computer

Description automatically generated

1. Determine the “chunk” size given {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610}, length of the array is 16.

Chunk Size = sqrt(16) = 4

1. How many comparisons are needed to find the key, 55 in an array of 16 elements?

2 \* sqrt(n) = 2 \* 4 = 8 comparisons

1. Explain the program functionality and how the algorithm is implemented in the program for Jump search.

Algorithm jumps indexes by sqrt(n) each time until finding a chunk where element could be present. Once this chunk is found it starts at the beginning and performs a linear search to find the exact index of the element. If the element is present it returns the index of the element otherwise it returns -1.

Quick Sort comparing to std::sort() STL

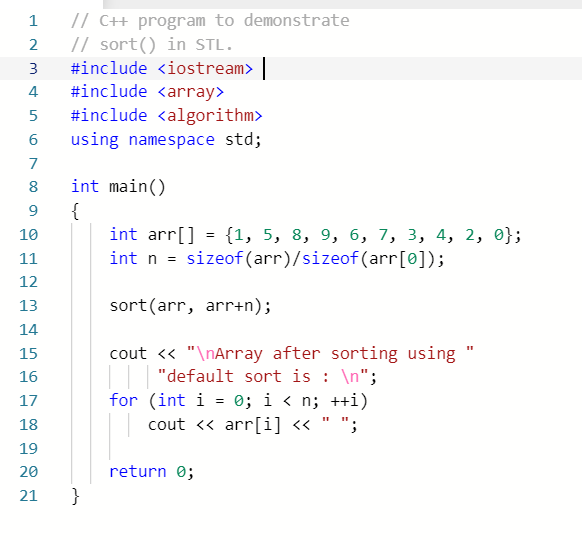
**sort()** takes a third parameter that is used to specify the order in which elements are to be sorted.

This function internally uses IntroSort. In more details it is implemented using hybrid of QuickSort, HeapSort and InsertionSort. By default, it uses QuickSort but if QuickSort is doing unfair partitioning and taking more than N\*logN time, it switches to HeapSort and when the array size becomes really small, it switches to InsertionSort.

*sort(startaddress, endaddress)*

* *startaddress*: the address of the first element of the array
* *endaddress*: the address of the next contiguous location of the last element of the array.
* sort() rearranges elements in the range of [*startaddress, endaddress*)

**Example 4**: Sort ascending order using STL



1. Write and run Example 4 program in IDE. Provide a screen capture of code and output.

A screenshot of a computer

Description automatically generated

1. Explain the sort functionality in the program. How are the values compared for order rearrangement?

The program uses the STL library sort to sort the array of integers. The sort takes two arguments to specify the beginning and end of the array to be sorted and then compares pairs of elements and swaps then so that they are in ascending order.

## Selection Sort

**Selection sort** algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array**.**

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

arr[] = 64 25 12 22 11

1. Find the minimum element in arr[0...4] and place it at beginning

**11** 25 12 22 64

1. Find the minimum element in arr[1...4] and place it at beginning of arr[1...4]

11 **12** 25 22 64

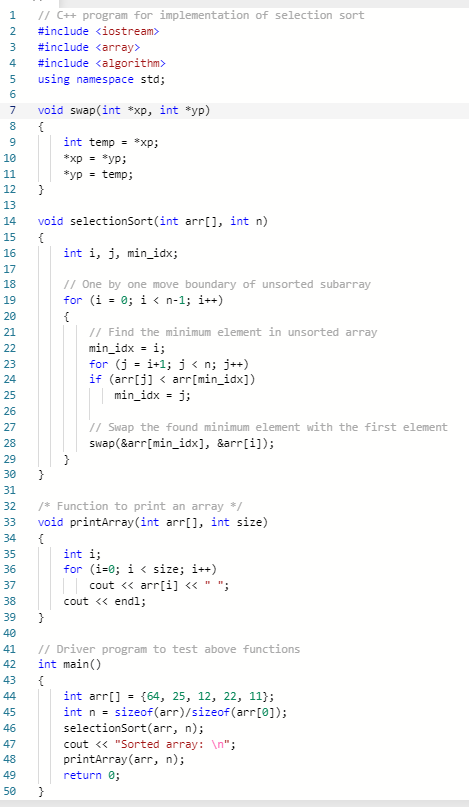
1. Find the minimum element in arr[2...4] and place it at beginning of arr[2...4]

11 12 **22** 25 64

1. Find the minimum element in arr[3...4] and place it at beginning of arr[3...4]

11 12 22 **25** 64

**Example 5**: Selection Sort



1. Write and run Example 5 in IDE. Provide screen capture of code and output.

A screenshot of a computer

Description automatically generated

1. Explain the implementation of selection sort algorithm in the program. How are the elements evaluated to rearrange the values from the lowest to the highest?

The algorithm is implemented using nested for loops. We start at index 0 and search through the rest of the array to find an element which is less than the value at index 0. Then we move to the index the right and continue doing this until the entire array is sorted in ascending order.

1. This program uses a nested for-loop, which provides T(n) = (n2). Determine the T(n) based on the program input.

T(n) =

### Merge Sort

**Merge Sort** is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.



**Example 6:** Merge Sort program

#include <iostream>

#include <array>

#include <algorithm>

using namespace std;

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

/\* create temp arrays \*/

int L[n1], R[n2];

/\* Copy data to temp arrays L[] and R[] \*/

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

L[i] = arr[l + i];

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

R[j] = arr[m + 1 + j];

/\* Merge the temp arrays back into arr[l..r]\*/

i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray

k = l; // Initial index of merged subarray

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

}

else {

arr[k] = R[j];

j++;

}

k++;

}

/\* Copy the remaining elements of L[], if there

are any \*/

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

/\* Copy the remaining elements of R[], if there

are any \*/

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

/\* l is for left index and r is right index of the

sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

if (l < r) {

// Same as (l+r)/2, but avoids overflow for

// large l and h

int m = l + (r - l) / 2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

/\* UTILITY FUNCTIONS \*/

/\* Function to print an array \*/

void printArray(int A[], int size)

{

int i;

for (i = 0; i < size; i++)

cout << (A[i]) << " ";

cout << ("\n");

}

/\* Driver program to test above functions \*/

int main()

{

int arr[] = { 12, 11, 13, 5, 6, 7 };

int arr\_size = sizeof(arr) / sizeof(arr[0]);

cout << ("Given array is \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

cout << ("\nSorted array is \n");

printArray(arr, arr\_size);

return 0;

}

The above program the merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

The program consists of the following procedures:

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves: middle m = (l+r)/2
2. Call mergeSort for first half: Call mergeSort(arr, l, m)
3. Call mergeSort for second half: Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3: Call merge(arr, l, m, r)
5. Write and run Example 6 program in IDE. Provide screen capture of code and output.

A screenshot of a computer

Description automatically generated

1. Merge sort program’s time complexity is: T(n) = nlogn. Determine T(n) for Example 6 program.

T(n) = 6Log(6)

1. Explain how Merge Sort algorithm is implemented in the program.

The Merge Sort algorithm is broken down into to functions. The Merge Sort function first subdivides the array into smaller portions until the array is only 1 element long. Then it uses the merge helper function to merge the subdivided arrays back into 1 complete sorted array.

Save screen captures and answers.

Login Canvas and submit document that contains screen captures, questions and answers.