**ICS 103 – Computer Programming in C**

**Lab # 11 Searching & Sorting**

**Objective:**

* Learn how to search arrays using linear and binary searchs.
* Learn how to sort an array using selection and bubble sorts.

**Searching:**

A common problem encountered in programming is the need to search an array in order to find the location of the desired value. Suppose you are looking for a student with ID 995203 in a list of ID’s, then you search for this ID in the list from the beginning till you find it or reach the end of the array and the ID is not there.

The searching algorithms we will use are the linear search and the binary search.

**Linear search:**

This is a general search strategy. Here, we search for the element from the beginning of the array. If we find the target element in the array, we stop searching and return the position of the element in the array; otherwise we continue till the end of the array. If we reach the end of the array and did not find the target, the search returns –1 as an indication that it is not in the array.

This is the code of linear search:

**int linear\_search(int a[], int n, int key){**

**int i;**

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

 **if(key == a[i])**

 **return i ;**

**}**

**return -1;**

**}**

**Binary search:**

If the array we are searching is sorted and the sorting order is known, then we can use a better strategy than linear search to look for the target element. The binary search makes use of the fact that the elements are ordered and does not scan the whole array. The steps of binary search, for an array that is sorted in increasing order, are:

1. if the middle element m is equal to the target, then return its position and we are done.
2. else:
	1. if the target < middle element, then search the first half of the array and repeat step 1.
	2. if the target > middle, then search second half of the array and repeat step1.
	3. if the array is exhausted, then the target is not in the array and the search returns -1.

In the implementation, we use two variables (first and last) that indicate the start and the end of the sub-array being searched. Through first and last we can get the middle element and specify the boundary of the array for the next pass if we do not find the target in the current pass.

This is code for binary search:

**int binary\_search(int x[], int first, int last, int key) {**

**int middle;**

**if(last<first)**

 **return -1; // key not in array**

**middle=(first+last)/2;**

**if(key==x[middle])**

 **return middle;**

 **else if (key<x[middle])**

 **return binary\_search(x, first,middle-1, key);**

**else**

 **return binary\_search(x, middle+1,last, key);**

 **}**

**Example1**: The following figure is an example of binary search in which we are searching for **22**:

 Note: The original call to the **binary\_search** function, in this case, is:

 **binary\_search(a, 0, arraySize – 1, 22);**

 where ***a*** is an int array of size 12. The array ***a*** corresponds to array ***x*** in the function.



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration# | first | last | middle | comparison | comment |
| 1 | 0 | 11 | (0+11) / 2 = 5 | 22 > x[5] | first = middle + 1 |
| 2 | 6 | 11 | (6+11) / 2=8 | 22 < x[8] | last = middle - 1 |
| 3 | 6 | 7 | (6+7) / 2 = 6 | 22 **==** x[6] | Since **target == x[middle]** the recursion stops with success i.e., **target** is found |

**Example2**: Use binary\_search to search for **6** in the array of the above example

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration# | first | last | middle | comparison | comment |
| 1 | 0 | 11 | (0+11) / 2 = 5 | 6 < x[5] | last = middle - 1 |
| 2 | 0 | 4 | (0+4) / 2 = 2 | 6 < x[2] | last = middle - 1 |
| 3 | 0 | 1 | (0+1) / 2 = 0 | 6 > x[0] | first = middle + 1 |
| 4 | 1 | 1 | (1+1) / 2 = 1 | 6 < x[1] | last = middle - 1 |
| 5 | 1 | 0 |  |  | Since **last < first**, the recursion stops with failure i.e., **target** is not found |

**Sorting:**

Another interesting problem in programming is to sort the elements of an array in increasing or decreasing order. The use of sorted arrays is very obvious. Having sorted records makes it easier to locate a particular element. Also if you want the output of these records (for example students ID’s along with their grades) to be displayed on screen sorted, then using a sort strategy becomes necessary.

In this lab we will discuss two sorting algorithms: Selection Sort and Bubble sort.

**Selection Sort:**

Selection sort is a simple straightforward algorithm to sort an array of numbers and it works as follows. First, we find the minimum element and store it in A[0]. Next, we find the minimum of the remaining elements and store it in A[1]. We continue this way until the second largest element is stored in A[size – 1]. The following figure explains the process of selection sort:



This is the code for selection sort:

**void selection\_sort(int x[], int size){**

 **int k,j,minpos,temp;**

 **for (k=0; k < size - 1; k++){**

 **minpos = k; // initialize location of min value**

 **// go over the elements to find location of minimum value**

 **for(j = k+1; j < size; j++){**

 **if(x[j] < x[minpos])**

 **minpos = j;**

 **}**

 **// bring minimum value which is at minpos at index k**

 **temp = x[minpos];**

 **x[minpos] = x[k];**

 **x[k] = temp;**

 **}**

**}**

**Burble Sort:**

* The idea of Bubble (or exchange) sort is to scan through the list and swap each pair of adjacent elements that are in the wrong order.
* The process is repeated each time from index zero to one less than the previous limit until either the list is exhausted or until a pass that involve no swap is encountered.
* At the end of first pass, the largest element will move (or bubble up) to the end of the list.
* At the end of the second swap, the second largest will move to its right place, etc.
* The following table shows a trace of how bubble sort works.

****

This is the code for bubble sort:

**void bubble\_sort(double a[], int size) {**

 **int i, pass = 1, swap\_occurs;**

 **double temp;**

 **do{**

 **swap\_occurs = 0;**

 **for(i = 1; i <= size - pass; i++) {**

 **if(a[i - 1] > a[i]) {**

 **temp = a[i – 1];**

 **a[i-1] = a[i];**

 **a[i] = temp;**

 **swap\_occurs = 1;**

 **}**

 **}**

 **pass++;**

 **} while (swap\_occurs && pass <= size-1);**

**}**

**Lab Work:**

**Question 1:**

Write a program that will generate an array of 10 integer numbers from 0 to 30 using the function **rand()** present in **stdlib.h** library. The function **rand()** will return a pseudo random number from **0** to **RAND\_MAX** which is the maximum value in the **int** type. In order to limit your number from **0** to **n**, make the returned **value** to be **value % (n+1)**. By using **rand()** only, your program will generate the same sequence during each execution. To avoid this problem insert the following statement at the beginning of your main function:

**srand(time(NULL));**

 Ask the user to enter a **target value** (between 0 and 30) to check if it is present in the array by using linear search. If present your program should display its position, otherwise print a message saying that the value is not in the array.

**Question 2:**

The binary search function shown above is recursive; rewrite the same function using a loop instead of recursion. To test your iterative binary search function repeat **question 1** by using binary search instead of linear search.

**Note: you should sort your array before searching**

**Question 3:**

Write a logical function **sorted\_inc** that receives an array of int values and **n** representing the number of values. The function will return **1** if the array values are sorted in increasing order, **0** otherwise.

Write a similar function **sorted\_dec** to check if the array values are sorted in decreasing order or not.

Write another version of selection\_sort; **selection\_sort\_dec** so that it sorts the array in decreasing order.

Write a program that will generate an array x1 of 20 random integer values from 1 to 100. Use similar procedure as in question 1, but now the lower part is not 0.

Print the array on the screen.

Check if it is by chance sorted in increasing or decreasing order by using the functions sorted\_inc and sorted\_dec. If the array is not sorted (this is what is expected), sort the array by using the original and modified selection sort functions i.e. increasing and decreasing. Print your array after each sorting.