Chapter 2 - Searching and Sorting

Linear Search

Linear search or sequential search is a method for finding a particular value in a list. It consists of checking every one of the list elements, one at a time and in sequence, until the desired one is found or reaches to the end of list.

For a list with *n* items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed. The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case *n* comparisons are needed. This results in O(n) performance on a given list.

The following pseudocode describes linear search, where the result of the search is supposed to be either the location of the item where the desired item was found; or an invalid location -1, to indicate that the desired element does not occur in the list.

```
For each item in the list:
     if that item is the desired item,
         stop the search and return
the item's location.
 Return -1.
```

Advantages:

- The linear search is simple - It is very easy to understand and implement;
- It does not require the data in the array to be stored in any particular order.

Disadvantages:

- The linear search is inefficient - If the array being searched contains 20,000 elements, the algorithm will have to look at all 20,000 elements in order to find a value in the last element. In an average case, an item is just as likely to be found near the beginning of the array as near the end. Typically, for an array of N items, the linear search will locate an item in N/2 attempts. If an array has 50,000 elements, the linear search will make a comparison with 25,000 of them in a typical case. This is assuming that the search item is consistently found in the array. **N/2 is the average number of comparisons. The maximum number of comparisons is always N.**

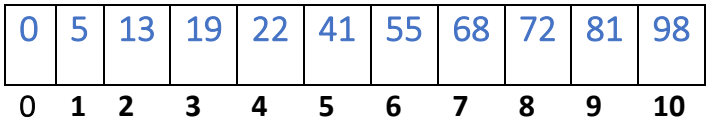
```
#include <stdio.h>

int linear_search(int[], int, int);

int main()
{
    int array[100], search, i, n, position;
    printf("Enter the number of elements in array\n");
    scanf("%d",&n);
    printf("Enter %d numbers\n", n);
    for ( i = 0 ; i < n ; i++ )
      scanf("%d", &array[i]);
    printf("Enter the number to search\n");
    scanf("%d",&search);

   position = linear_search(array, n, search);

   if ( position == -1 )
      printf("%d is not present in array.\n", search);
   else
      printf("%d is present at location %d.\n", search,
position+1);
 }

int linear_search(int a[], int n, int key)
{
   int i;

   for ( i = 0 ; i < n ; i++ )
   {
      if ( a[i] == key )
         return i;
   }
    return -1;
}
```

Binary Search

A **binary search** or **half-interval search** algorithm locates the position of an item in a sorted array. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned. Binary search algorithm typically halve the number of items to check with each successive iteration, thus locating the given item (or determining its absence) in logarithmic time. A binary search is a divide and conquer search algorithm.

```
if (value == middle element)
   value is found. return index of middle element
else if (value < middle element)
    Search left – half of list with the same method
else
    Search right – half of list with the same method
```

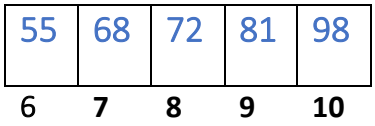
For example, consider the following sequence of integers sorted in ascending order and say we are looking for the number 55:



Index of the middle element is given by

mid = (low + high)/2 = (0+10)/2 = 5

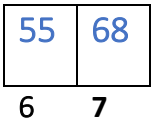
Middle element is element at index 5 i.e., 41. 41 is smaller than 55. From this we conclude not only that the element at index 5 is not the target value, but also that no element at indices between 0 and 4 can be the target value, because all elements at these indices are smaller than 41, which is smaller than the target value. This brings the search space down to indices 6 through 10:



Proceeding in a similar fashion, now

mid = (6 + 10) / 2 = 8

Middle element is at index 8 i.e., 72. 72 is bigger than 55. We chop off the second half of the search space and are left with:



mid = (6 + 7) / 2 = 6.5 = 6

This time middle element, which is at index 6 is equal to the search element. So, we can terminate the process by returning the index 6.

Binary search is much faster than linear search for most data sets. If you look at each item in order, you may have to look at every item in the data set before you find the one you are looking for. With binary search, you eliminate half of the data with each decision. If

there are n items, then after the first decision you eliminate n/2 of them. After the second decision you've eliminated 3n/4 of them. After the third decision you've eliminated 7n/8 of them. Etc. In other words, binary search is O(log n) . For a large data set, binary search would be much better than linear search. A standard binary search method can then be written as follows:

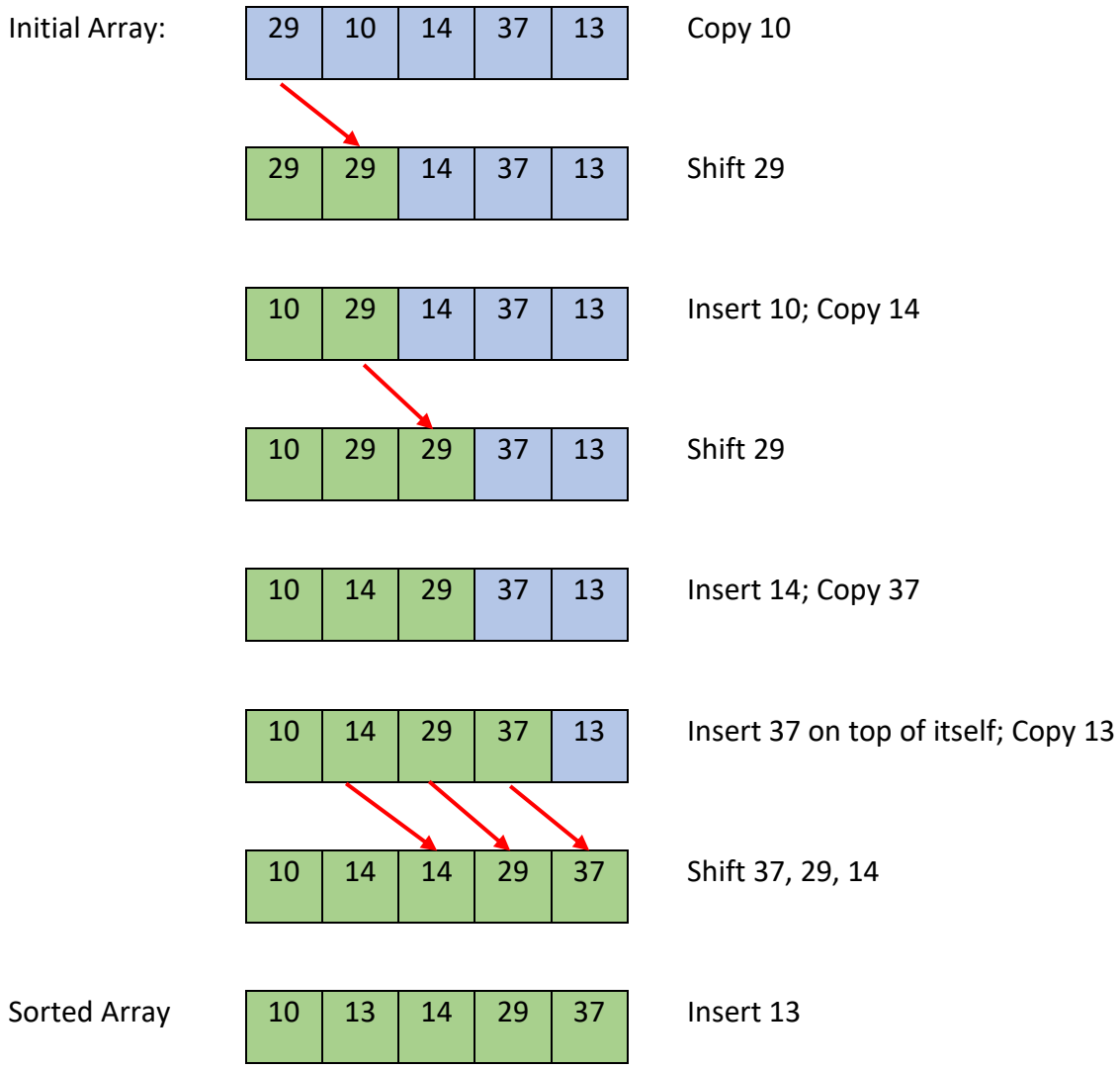
```
public int binarySearch(int a[], int n,  int target)
{
            int low = 0;
            int high = a.length - 1;
            int mid;
            while (low <= high)
            {
                        mid = (low + high) / 2;
                        if (target == a[mid])          //match
                                    return mid;
                        else
                        if (target < a[mid])
                                     //search low end of array
                                    high = mid - 1;
                        else
                                   //search high end of array
                         low = mid + 1;
              }
return -1;
}
```

Sorting Techniques

Sorting is a process of arranging objects according to some linear order such as £ for numbers. Sorting is of two types: Internal sorting and external sorting. Internal sorting takes place in the main memory of a computer. External sorting is necessary when the number of objects to be sorted is too large to fit in main memory.

Insertion Sort

One of the simplest sorting algorithms is the insertion sort. Insertion sort consists of n - 1 passes. In i^{th} pass we "insert" the i^{th} element $A[i]$ into its rightful place among $A[1]$, $A[2]$, . . . , $A[i-1]$, which were previously placed in sorted order. After doing this insertion, the objects occupying $A[1]$, . . . , $A[i]$ are in sorted order. The following example implements this strategy.




```
void insertionSort(int a[], int n)
{
      int t,i,j;
      for(i=1; i < n; i++)
      {
      t = a[i];
      for(j=i; t < a[j-1] && j > 0; j--)
            a[j] = a[j-1];
      a[j] = t;
      }
}
```

Insertion sort provides several advantages:

- Simple implementation
- Efficient for (quite) small data sets
- Adaptive, i.e. efficient for data sets that are already substantially sorted
- More efficient in practice than most other simple quadratic, i.e. O(n2) algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is O(n)
- In-place, i.e. only requires a constant amount O(1) of additional memory space

- Online, i.e. can sort a list as it receives it

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., $\Theta(n)$), because the test in the inner for loop always fails immediately. During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The worst case input is an array sorted in reverse order. In this case every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. For this case insertion sort has a quadratic running time (i.e., $O(n^2)$).

The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quick sort.

Selection Sort

Selection sort algorithm finds the smallest element in the list and put it in the first position during first pass. Then finds the second smallest element in the list and put it in the second position in pass 2. In the i^{th} pass, selects the element with the lowest key, among $A[i], \ldots, A[n]$, and swaps it with $A[i]$. As a result, after i passes, the i lowest records will occupy $A[1], \ldots, A[i]$, in sorted order. Following example illustrates selection sort.

Initial Array	23 78 45 8 32 46	Pass 1	Swap 23 and 8
	8 78 45 23 32 46	Pass 2	Swap 78 and 23
	8 23 45 78 32 46	Pass 3	Swap 45 and 32
	8 23 32 78 45 46	Pass 4	Swap 78 and 45
	8 23 32 45 78 46	Pass 5	Swap 78 and 46
Sorted Array	8 23 32 45 46 78	Pass 6	Swap 78 and 46

```
void selectionSort(int a[], int n)
{
      int t,i,j,k;
      for(i=0; i < n-1; i++)
      {
      k = i;
      for(j = i+1; j < n; j++)
            if (a[j] < a[k]) k = j;
      t = a[i];
      a[i] = a[k];
      a[k] = t;
      }
}
```

Selecting the lowest element requires scanning all n elements (this takes $n - 1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on, for $(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2 \in \Theta(n^2)$ comparisons.

Bubble Sort

Bubble Sort is an elementary sorting algorithm. It works by using the following steps:

1. In each pass, we compare adjacent elements and swap them if they are out of order until the end of the list. By doing so, the 1st pass ends up "bubbling up" the largest element to the last position on the list
2. The 2nd pass bubbles up the 2nd largest, and so on until, after N-1 passes, the list is sorted.

When no exchanges are required, the list is sorted. The algorithm gets its name from the way bigger elements "bubble" to the end of the list.

Bubble sort has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of $O(n \log n)$. Even other $O(n^2)$ sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.

The only significant advantage that bubble sort has over most other implementations, even quicksort, but not insertion sort, is that the ability to detect that the list is sorted is efficiently built into the algorithm. Performance of bubble sort over an already-sorted list (best-case) is $O(n)$. By contrast, most other algorithms, even those with better average-case complexity, are more complex. Insertion sort performs better

than bubble sort on a list that is almost sorted. Bubble sort should be avoided in case of large collections.

Following figure explains the process used by bubble sort.

			a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
PASS 1	i = 0	j						
		0	5	3	1	9	8	2
		1	3	5	1	9	8	2
		2	3	1	5	9	8	2
		3	3	1	5	9	8	2
		4	3	1	5	8	9	2
		5	3	1	5	8	2	9
PASS 2	i = 1	0	3	1	5	8	2	9
		1	1	3	5	8	2	9
		2	1	3	5	8	2	9
		3	1	3	5	8	2	9
		4	1	3	5	2	8	9
PASS 3	i = 2	0	1	3	5	2	8	9
		1	1	3	5	2	8	9
		2	1	3	5	2	8	9
		3	1	3	2	5	8	9
PASS 4	i = 3	0	1	3	2	5	8	9
		1	1	3	2	5	8	9
		2	1	2	3	5	8	9
PASS 5	i = 4	0	1	2	3	5	8	9
		1	1	2	3	5	8	9

```
void BubbleSort(int a[], int n)
{
      int t,i,j;
      for(i=0; i < n-1; i++)
            for(j = 0; j < n-i-1; j++)
            if (a[j] < a[j+1])
            {
                  t = a[j];
                  a[j] = a[j+1];
                  a[j+1] = t;
             }
}
```