CHAPTER TWO
2. TIME COMPLEXITY OF KNOWN ALGORITHMS

2.1. Searching algorithms

Organizing and retrieving information is at the heart of most computer applications, and searching is surely the most frequently performed of all computing tasks. Search can be viewed abstractly as a process to determine if an element with a particular value is a member of a particular set. The more common view of searching is an attempt to find the record within a collection of records that has a particular key value or those records in a collection whose key values meet some criterion such as falling within a range of values.

Searching is a process of looking for a specific element in a list of items or determining that the item is not in the list. There are two simple searching algorithms:

- Sequential Search, and
- Binary Search

2.1.1. Linear Search (Sequential Search)

The sequential search algorithm begins at the first position in the array and looks at each value in turn until K is found. Once K is found, the algorithm stops. It steps through the data sequentially until a match is found.

It compares the key element with each element in the list A[n], and returns the index i if the key is found, or -1 if the key is not found.

The basic idea is:

- ➢ Loop through the array starting at the first element until the value of target matches one of the array elements.
- ➢ Return the index if match is found, and if a match is not found, return –1.

Implementation example:

```
int Linear_Search(int list[ ], int key)
{   // n is size of the list
     int index=0;
     int found=0;
     do{
     if(key = = list[index])
              found=1;
     else
              index++;
     } while(found = = 0 && index < n);
     if(found= = 0)
               index=-1;
     return index;
}
```

There is a wide range of possible running times for the sequential search algorithm. The first integer in the array could have value K, and so only one integer is examined. In this case the running time is short. This is the best case for this algorithm, because it is not possible for sequential search to look at less than one value. Alternatively, if the last position in the array contains K, then the running time is relatively long, because the algorithm must examine N values. This is the worst case for this algorithm, because sequential search never looks at more than N values. If we implement sequential search as a program and run it many times on many different arrays of size N, or search for many different values of K within the same array, we expect the algorithm on average to go halfway through the array before finding the value we seek. On average, the algorithm examines about N/2 values.

The complexity of the algorithm is analyzed in three ways. First, we provide the cost of an unsuccessful search. Then, we give the worst-case cost of a successful search. Finally, we find the average cost of a successful search. Analyzing successful and unsuccessful searches separately is typical. Unsuccessful searches usually are more time consuming than are successful searches.

An unsuccessful search requires the examination of every item in the array, so the time will be O(N). In the worst case, a successful search, too, requires the examination of every item in the array because we might not find a match until the last item. Thus the worst-case running time for a successful search is also linear. On average, however, we search only half an array. That is, for every successful search in position i, there is a corresponding successful search in position N-1- i (assuming we start numbering from 0). However, N/2 is still O(N).

2.1.2. Binary Search

This searching algorithms works only on an ordered list. Suppose you have a sorted array, and you wish to search it for a particular value, *x*. The key idea is that since the array is sorted, you can narrow your focus to a section of the array that must contain *x* if *x* is in the array at all. In particular, *x* must come before every element greater than *x* and after every element less than *x*. We call the section of the array that could contain *x* the *section of interest*. We can iteratively home in on *x* with a loop that shrinks the section of interest while maintaining the loop invariant that *x* is in that section, if it is in the array at all.

Binary search is performed from the middle of the array rather than the end. We keep track of *low* and *high*, which delimit the portion of the array in which an item, if present, must reside. Initially, the range is from *0* to *N - 1*. If *low* is larger than *high*, we know that the item is not present, so we return *NOT-FOUND*. Otherwise, we let *mid* be the halfway point of the range (rounding down if the range has an even number of elements) and compare the item we are searching for with the item in position *mid*. If we find a match, we are done and can return. If the item we are searching for is less than the item in position *mid*, then it must reside in the range *low* to *mid-1*. If it is greater, then it must reside in the range *mid+l* to *high*.

The basic idea is:

- Locate midpoint of array to search
- Determine if target is in lower half or upper half of an array.
    - If in lower half, make this half the array to search
    - If in the upper half, make this half the array to search
- Loop back to step 1 until the size of the array to search is one, and this element does not match, in which case return –1.

Implementation example:

```
int Binary_Search(int list[ ], int k)
{ //n is size of the list
   int low = 0;
   int high = n-1;
   int found = 0;
   do{
       mid = (low + high)/2;
       if(key = = list[mid])
           found=1;
      else{
              if(key < list[mid])
                     high = mid-1;
              else
                     low = mid+1;
              }
       }while(found = = 0 && low < high);
     if(found = = 0)
       index = -1;
     else
       index = mid;
   return index;
   }
```

For an unsuccessful search, the number of iterations in the loop is *logN + 1*. The reason is that we halve the range in each iteration (rounding down if the range has an odd number of elements); we add 1 because the final range encompasses zero elements. For a successful search, the worst case is log N iterations because in the worst case we get down to a range of only one element. The average case is only one iteration better because half the elements require the worst case for their search, a quarter of the elements save one iteration, and only one in 2^i elements save *i* iterations from the worst case. The mathematics involves computing the weighted average by calculating the sum of a finite series. The end result, however, is that the running time for each search is O(log N).

Initially, the number of candidate entries is n; after the first call to Binary_Search, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. In general, after the i^{th} call to Binary_Search, the number of candidate entries remaining is at most $n/2^i$. In the worst case (unsuccessful search), the recursive calls stop when there are no more candidate entries. Hence, the maximum number of recursive calls performed, is the smallest integer m such that $n/2^m < 1$. In other words (recalling that we omit a logarithm's base when it is 2), $m > \log n$. Thus, we have $m = \log n + 1$, which implies that binary search runs in $O(\log n)$ time.

As an example, if we have 128 entries, then the remaining candidate entries after each iteration are 64, 32, 16, 8, 4, 2, 1, 0. Thus, the running time is O(logN).

2.2. Sorting Algorithms

Sorting is one of the most important operations performed by computers. Sorting is a process of reordering a list of items in either increasing or decreasing order. The following are simple sorting algorithms used to sort small-sized lists.

- Insertion Sort
- Selection Sort
- Bubble Sort

2.2.1. Insertion Sort

An insertion sort starts by considering the two first elements of the array data, which are data[0] and data[1]. If they are out of order, an interchange takes place. Then, the third element, data[2], is considered. If data[2] is less than data[0] and data[1], these two elements are shifted by one position; data[0] is placed at position 1, data[1] at position 2, and data[2] at position 0. If data[2] is less than data[1] and not less than data[0], then only data[1] is moved to position 2 and its place is taken by data[2]. If, finally, data[2] is not less than both its predecessors, it stays in its current position. Each element data[i] is inserted into its proper location j such that $0 \leq j \leq i$, and all elements greater than data[i] are moved by one position.

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

It's the most instinctive type of sorting algorithm. The approach is the same approach that you use for sorting a set of cards in your hand. While playing cards, you pick up a card, start at the beginning of your hand and find the place to insert the new card, insert it and move all the others up one place.

The basic idea:

Find the location for an element and move all others up, and insert the element.

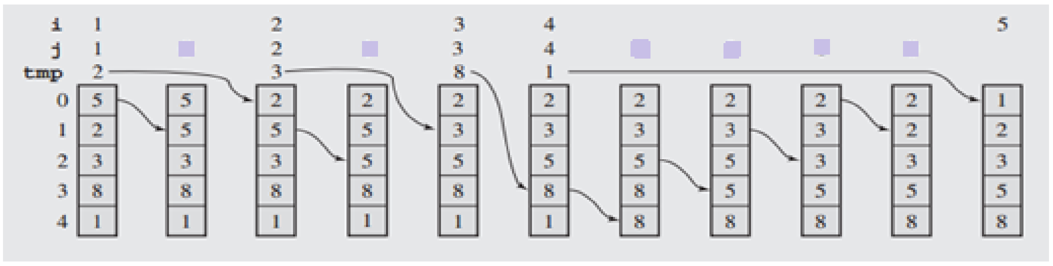
The process involved in insertion sort is as follows:

- The left most value can be said to be sorted relative to itself. Thus, we don't need to do anything.
- Check to see if the second value is smaller than the first one. If it is, swap these two values. The first two values are now relatively sorted.
- Next, we need to insert the third value in to the relatively sorted portion so that after insertion, the portion will still be relatively sorted.
- Remove the third value first. Slide the second value to make room for insertion. Insert the value in the appropriate position.
- Now the first three are relatively sorted.
- Do the same for the remaining items in the list.

Implementation example:

```
void Insertion_Sort(int list[ ] )
{
  int i, j, temp;
 for( i = 1; i < n; i++)
   {
     temp=list[i];
      for( j=i; j>0 && temp<list[j-1];j--)
          { // work backwards through the array finding where temp should go
               list[j]=list[j-1];
           }//end of inner loop
       list[j]=temp;
     }//end of outer loop
}//end of Insertion_Sort
```

The following figure shows how the array [5 2 3 8 1] is sorted by insertion sort.



Because an array having only one element is already ordered, the algorithm starts sorting from the second position, position *1*. Then for each element *temp* = *data[i]*, all elements greater than *temp* are copied to the next position, and *temp* is put in its proper place.

To find the number of comparisons performed by insertion sort algorithm, observe first that the outer for loop always performs n-1 iterations. The best case is when the data are already in order. Only one comparison is made for each position i, so there are n - 1 comparisons, which is O(n), , and 2(n - 1) moves, all of them redundant.

The worst case is when the data are in reverse order. In this case, for each i, the item data[i] is less than every item data[0], . . . , data[i-1], and each of them is moved by one position. For each iteration i of the outer for loop, there are i comparisons, and the total number of comparisons for all iterations of this loop is

$$\sum_{i=1}^{n-1}(i) = 1 + 2 + 3 + \ldots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

The number of times the assignment in the inner for loop is executed can be computed using the same formula. The number of times temp is loaded and unloaded in the outer for loop is added to that, resulting in the total number of moves

$$\frac{n(n-1)}{2} + 2\,(n - 1) = \frac{n^2+3n-4}{2} = O(n^2)$$

2.2.2. Selection Sort

Selection sort is an attempt to localize the exchanges of array elements by finding a misplaced element first and putting it in its final place. The element with the lowest value is selected and exchanged with the element in the first position. Then, the smallest value among the remaining elements data[1], . . . , data[n-1] is found and put in the second position. This selection and placement by finding, in each pass i, the lowest value among the elements data[i], . . . , data[n-1] and swapping it with data[i] are continued until all elements are in their proper positions.

The following pseudo code reflects the simplicity of the algorithm:

```
selectionsort(data[ ] , n)
  for i = 0 to n-2
    select the smallest element among data[i], . . . , data[n-1];
    swap it with data[i];
```

It is rather obvious that n-2 should be the last value for i, because if all elements but the last have been already considered and placed in their proper positions, then the nth element (occupying position n-1) has to be the largest.

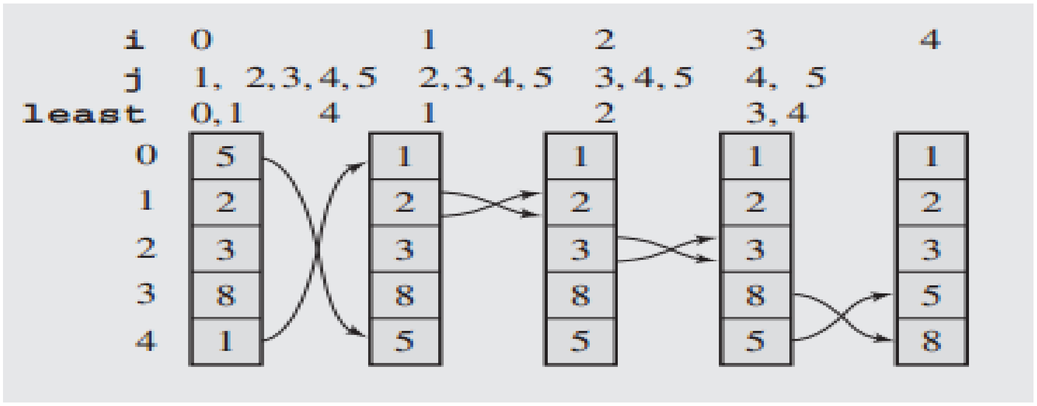
The basic idea:

- Loop through the array from i=0 to n-1.
- Select the smallest element in the array from i to n
- Swap this value with value at position i.

Implementation example:

```
void Selection_Sort(int data[ ] )
{
   int i,j, least;
  for( i = 0; i<n-1; i++ )
   {
     least = i;
     for( j = i+1; j<n; j++ )
       {
        if(data [j] < data [least])
            least = j;
      }//end of inner loop
      temp = data [least];
      data [least] = data [i];
      data [i] = temp;
   } //end of outer loop
}//end of Selection_Sort
```

The following figure shows how the array [5 2 3 8 1] is sorted by selection sort.



Note: *least* in the implementation and figure above is not the smallest element but its position.

The analysis of the performance of the function Selection_Sort ( ) is simplified by the presence of two for loops with lower and upper bounds. The outer loop executes *n - 1* times, and for each *i* between *0* and *n - 2*, the inner loop iterates *j = (n -1) - i* times. Because comparisons of keys are done in the inner loop, there are

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

comparisons. This number stays the same for all cases. There can be some savings only in the number of swaps. Note that if the assignment in the if statement is executed, only the index jis moved, not the item located currently at position j. Array elements are swapped unconditionally in the outer loop as many times as this loop executes, which is **n-1**. Thus, in all cases, items are moved the same number of times, **3(n - 1)**, i.e **O(n)**.

2.2.3. Bubble Sort

A bubble sort can be best understood if the array to be sorted is envisaged as a vertical column whose smallest elements are at the top and whose largest elements are at the bottom. The array is scanned from the bottom up, and two adjacent elements are interchanged if they are found to be out of order with respect to each other. First, items data[n-1] and data[n-2] are compared and swapped if they are out of order. Next, data[n-2] and data[n-3] are compared, and their order is changed if necessary, and so on up to data[1] and data[0]. In this way, the smallest element is bubbled up to the top of the array.

However, this is only the first pass through the array. The array is scanned again comparing consecutive items and interchanging them when needed, but this time, the last comparison is done for data[2] and data[1] because the smallest element is already in its proper position, namely, position 0. The second pass bubbles the second smallest element of the array up to the second position, position 1. The procedure continues until the last pass when only one comparison, data[n-1] with data[n-2], and possibly one interchange are performed.

A pseudo code of this algorithm is as follows:

```
bubblesort(data[ ] , n)
  for i = 0 to n - 2
    for j = n-1down to i+1
      swap elements in positions j and j-1 if they are out of order;
```

Bubble sort is the simplest algorithm to implement and the slowest algorithm on very large inputs.

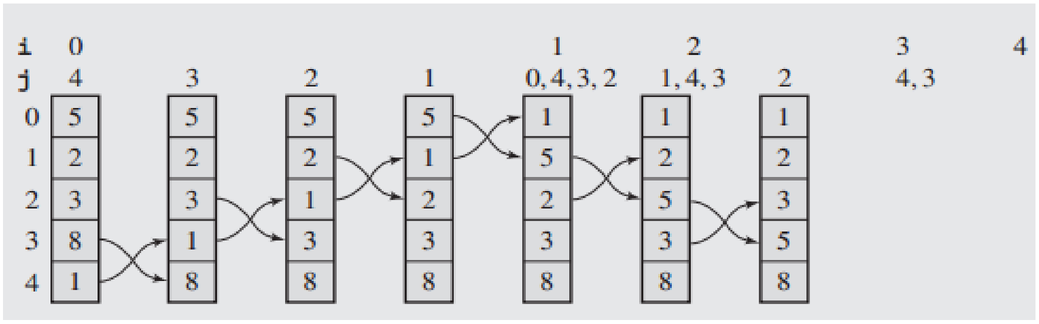
The basic idea of the algorithm is:

- Loop through array from i=0 to n and swap adjacent elements if they are out of order.

Implementation example:

```
void Bubble_Sort( data[ ] )
{
    int i, j, temp;
    for( i = 0; i < n-1; i++)
      {
        for( j = n-1; j > i; j--)
          {
              if(data [ j ] < data [ j-1 ] )
                  {
                    temp = data [ j ];
                    data [ j ] = data [ j-1 ];
                    data [ j-1 ] = temp;
                 }//swap adjacent elements
          }//end of inner loop
      }//end of outer loop
}//end of Bubble_Sort
```

The following figure shows how the array [5 2 3 8 1] is sorted by bubble sort.



The number of comparisons is the same in each case (best, average, and worst) and equals the total number of iterations of the inner for loop

$$\sum_{i=2}^{n-2}(n-1-i) = \frac{n(n-1)}{2} = O(n^2)$$

comparisons. This formula also computes the number of swaps in the worst case when the array is in reverse order. In this case, $3\,\frac{n(n-1)}{2} = O(n^2)$ moves have to be made. The best case, when all elements are already ordered, requires no swaps.