Analysis
of
Binary Search algorithm
and
Selection Sort algorithm

In this section we shall take up two representative problems in computer science, work out the algorithms based on the best strategy to solve the problems, and compute the time complexity of the algorithms. The two problems, one related to searching and the other related to data sorting are very common in many real world situations.

Binary Search Method:

Suppose we are given a number of integers stored in an array A, and we want to locate a specific target integer K in this array. If we do not have any information on how the integers are organized in the array, we have to sequentially examine each element of the array. This is known as *linear search* and would have a time complexity of *O(n) in the worst case*.

However, if the elements of the array are ordered, let us say in ascending order, and we wish to find out the position of an integer target K in the array, we need not make a sequential search over the complete array. We can make a faster search using the **Binary search method**.

The basic idea is to start with an examination of the middle element of the array. This will lead to 3 possible situations:

If this matches the target K, then search can terminate successfully, by printing out the index of the element in the array.
On the other hand, if K<A[middle], then search can be limited to elements to the left of A[middle]. All elements to the right of middle can be ignored.
If it turns out that K >A[middle], then further search is limited to elements to the right of A[middle].

If all elements are exhausted and the target is not found in the array, then the method returns a special value such as –1.

Here is one version of the Binary Search function:

```
int BinarySearch (int A[ ], int n, int K)
{
     int L=0,  Mid, R= n-1;
      while (L<=R)
     {
          Mid = (L +R)/2;
          if ( K= =A[Mid] )
                return Mid;
          else if ( K > A[Mid] )
               L = Mid + 1;
          else
                R = Mid – 1 ;
     }
     return  –1 ;
}
```

Complexity Analysis:

Let us now carry out an Analysis of this method to determine its time complexity. Since there are no “for” loops, we can not use summations to express the total number of operations. Let us examine the operations for a specific case, where the number of elements in the array n is 64.

When n= 64 BinarySearch is called to reduce size to n=32
When n= 32 BinarySearch is called to reduce size to n=16
When n= 16 BinarySearch is called to reduce size to n=8
When n= 8 BinarySearch is called to reduce size to n=4
When n= 4 BinarySearch is called to reduce size to n=2
When n= 2 BinarySearch is called to reduce size to n=1

Thus we see that BinarySearch function is called 6 times (6 elements of the array were examined) for n =64.
Note that $64 = 2^6$

Also we see that the BinarySearch function is called 5 times (5 elements of the array were examined) for n = 32.
Note that $32 = 2^5$

Let us consider a more general case where n is still a power of 2. Let us say $n = 2^k$.

Following the above argument for 64 elements, it is easily seen that after k searches, the while loop is executed k times and n reduces to size 1.
Let us assume that each run of the while loop involves at most 5 operations.
Thus total number of operations: 5k.
The value of k can be determined from the expression

$2^k = n$

Taking log of both sides

$k = \log n$

Thus total number of operations = $5 \log n$.

We conclude from there that the time complexity of the Binary search method is $O(\log n)$, which is much more efficient than the Linear Search method.

We shall study a recursive version of the Binary search method later when we study recursion.

Selection sort Method:

Suppose we are given a number of integers in an array A, and we want to sort the integers in increasing order. A number of methods are available to carry out sorting. Here we present the Selection sort method. As the name suggests, the method selects the smallest element in the array and puts it in proper place in the array. To start with, it would be the first position in the array. Once the first element has been positioned, this process is repeated by considering the remaining elements of the array.

Consider the array

[56 30 37 58 19 95 73 25]

The first pass locates the smallest element as 19 and swaps it with the first element 56 to result in the array

[**19** 30 37 58 56 95 73 25]

The process is repeated to find the next smallest from remaining elements which is 25. It is swapped with the 2nd element to result in the array

[**19 25** 37 58 56 95 73 30]

The process is now repeated for remaining elements of the array, till all elements have been put in proper places. Note that for this array of size 8, the process is to be executed 7 times. When 7 elements have been placed in proper position, the 8th element will be automatically in its proper place, as it would be largest element.

Here is the selection sort algorithm:

```
for i = 1 to n – 1
{
        minPosition = i;
        for j = i+1 to n
                        if A[j] < A[minPosition]        minPosition = j;

        swap A[i] with A[minPosition];

 }
```

Complexity Analysis:

We observe that the outer for loop variable i runs from 1 to n, while the inner for loop j runs from i+1 to n. One operation is being performed inside the innermost loop. Let us assume that swap involves just one operation. Note that for every run of the outer loop, there is one swap operation.

In terms of summations the total number of operations can be expressed as

$$\sum_{i=1}^{n-1} \left[\sum_{j=i+1}^{n} 1 + 1 \right]$$

The second summation can be split into two summations each starting from 1.

$$= \sum_{i=1}^{n-1} \left[\sum_{j=1}^{n} 1 - \sum_{j=1}^{i} 1 \right] + 1$$

$$= \sum_{i=1}^{n-1} [\, n - i \,] + 1$$

$$= \sum_{i=1}^{n-1} n+1 - \sum_{i=1}^{n-1} i$$

$$= (n+1)(n-1) - (n-1)n/2$$

$$= n^2 - 1 - n^2/2 + n/2$$

$$= n^2/2 + n/2 - 1$$

Thus the complexity of selection sort algorithm is $O(n^2)$.