

## Tutorial-1

### How to calculate Running time of an algorithm?

We can calculate the running time of an algorithm reliably by running the implementation of the algorithm on a computer.

Alternatively we can calculate the running time by using a technique called algorithm analysis. We can estimate an algorithm's performance by counting the number of basic operations required by the algorithm to process an input of a certain size.

**Basic Operation:** The time to complete a basic operation does not depend on the particular values of its operands. So it takes a constant amount of time.

Examples: Arithmetic operation (addition, subtraction, multiplication, division), Boolean operation (AND, OR, NOT), Comparison operation, Module operation, Branch operation etc.

**Input Size:** It is the number of input processed by the algorithm.

Example: For sorting algorithm the input size is measured by the number of records to be sorted.

**Growth Rate:** The growth rate of an algorithm is the rate at which the running time (cost) of the algorithm grows as the size of the input grows. The growth rate has a tremendous effect on the resources consumed by the algorithm.

Consider the following simple algorithm to solve the problem of finding the 1<sup>st</sup> element in an array of n integers.

```
public int findFirstElement(int[] a){
    int firstElement = a[0];
    return firstElement;
}
```

It is clear that no matter how large the array is, the time to copy the value from the first position of the array is always constant (say k). So the time T to run the algorithm as a function of n,  $T(n) = k$ . Here T(n) does not depend on the array size n. We always assume T(n) is a non-negative value.

Consider another following algorithm to solve the problem of finding the smallest element in an array of n integers.

```
public int findSmallElement(int[] a){
    int smElement = a[0];
    for(int i=0; i<n ; i++)
        if(a[i] < smElement)
            smElement=a[i];
    return smElement;
}
```

Here the basic operation is to compare between two integers and each comparison operation takes a fixed amount of time (say k) regardless of the value of the two integers or their position in the array. In this algorithm the comparison operation is repeated n times due to for loop. So the running time of the above algorithm,  $T(n) = kn$ . The above algorithm is said to have linear growth rate.

Since for calculation of running time we want a reasonable approximation we have ignored the time required to increment the variable i, the time for actual assignment when a smaller value is found or time taken to initialize the variable smElement.

Consider another algorithm to solve the problem of finding the smallest element from a two dimensional array n rows and n columns.

```
public int findSmallElement(int[][] a){
    int smElement = a[0][0];
    for(int i=0; i<n ; i++)
        for(int j=0; j<n ; j++)
            if(a[i][j] < smElement)
                smElement=a[i][j];
    return smElement;
}
```

The total number of comparison operation occurs  $n*n=n^2$  times. So the running time of the algorithm,  $T(n) = kn^2$ . The above algorithm is said to have quadratic growth rate.

### Contiguous Subsequence Sums Example:

int[] a= {3, 4, 1, 3, 2, 7, 4, 4, 2, 6, 1, 4}

We shall compute all contiguous subsequence of length 5 for the array.

Array size n=12, subsequence length m=5.

Total number of subsequence = n - m + 1 = 12-5 +1=8.

#### # Using Brute force algorithm:

$$S_0 = a[0] + a[1] + a[2] + a[3] + a[4] = 3+4+1+3+2=13$$

$$S_1 = a[1] + a[2] + a[3] + a[4] + a[5]=4+1+3+2+7=17$$

$$S_2 = a[2] + a[3] + a[4] + a[5] + a[6]=1+3+2+7+4=17$$

$$S_3 = a[3] + a[4] + a[5] + a[6] + a[7]=3+2+7+4+4=20$$

$$S_4 = a[4] + a[5] + a[6] + a[7] + a[8]=2+7+4+4+2=19$$

$$S_5 = a[5] + a[6] + a[7] + a[8] + a[9]=7+4+4+2+6=23$$

$$S_6 = a[6] + a[7] + a[8] + a[9] + a[10]=4+4+2+6+1=17$$

$$S_7 = a[7] + a[8] + a[9] + a[10] + a[11]=4+2+6+1+4=17$$

Using Brute force algorithm total number of additions =  $8*4=32$ .

#### # Using previous subsequence ( $S_{k+1}=S_k + a[k+m] - a[k]$ )

$$S_0 = a[0] + a[1] + a[2] + a[3] + a[4] = 3+4+1+3+2=13$$

$$S_1 = S_0 + a[5] - a[0] = 13+7-3=17$$

$$S_2 = S_1 + a[6] - a[1] = 17+4-4=17$$

$$S_3 = S_2 + a[7] - a[2] = 17+4-1=20$$

$$S_4 = S_3 + a[8] - a[3] = 20+2-3=19$$

$$S_5 = S_4 + a[9] - a[4] = 19+6-2= 23$$

$$S_6 = S_5 + a[10] - a[5] = 23+1-7= 17$$

$$S_7 = S_6 + a[11] - a[6] = 17+4-4= 17$$

Total number of additions =18

#### Running Time Calculation Examples:

```
a) for(int i=0; i<n; i++)
    System.out.println("Algorithm analysis");
   for(int j=n; j>0; j--)
```

```
    System.out.println("Algorithm analysis");
```

The println() method takes a constant amount of time say c.

The println() method will be called n times due to 1<sup>st</sup> for loop and n times due to 2<sup>nd</sup> for loop. So total running time of the above algorithm

$$T(n) = (n+n)*c = 2nc$$

```
b) for(int i=n; i>0; i--)
    for(int j=n; j>i; j--)
```

```
    System.out.println("Algorithm analysis");
```

The loop variable for the outer loop is assigned to the values

n, n-1, n-2, ..... 1 resulting a total of n iterations. The inner loop is executed (n-i) times. The total number of calls to the println() method is (n-n) + (n - (n-1)) + (n - (n-2)) + ..... + (n-1) = 0+1+2+.....+n-1 =  $(n-1)*n/2$ .

$$T(n) = c* (n-1)*n/2.$$

```
c) for(int i=1; i<n; i=i*2)
    System.out.println("Algorithm analysis");
```

The loop variable i is assigned to the values 1, 2, 4,.....n. For simplicity of our calculation let us assume that n is a power of 2. Suppose the loop will be terminated after k number of iterations. So,  $n = 2^k$  ie.  $\log_2 n = k$ . Running time of the above algorithm,  $T(n) = c*k = c*\log_2 n$ .

```
d) for (int i=0; i<=n; i++)
    if(i % 10 == 0)
    for (int j=0; j<i; j++)
```

```
    System.out.println("Algorithm analysis");
```

The inner loop will be executed only when i is a multiple on 10 ie. i=0, 10, 20, ....., (n/10)\*10. The total number of calls to the println() method

is  $0 + 10 + 20 + 30 + \dots + (n/10) * 10 =$   
 $10 * (0 + 1 + 2 + 3 + \dots + n/10) = 10 * (n/10) * (n/10 + 1) / 2.$   
Running time of the above algorithm,  $T(n) = c * 10 * (n/10) * (n/10 + 1) / 2.$