Analysis of Algorithms | Set 4 (Analysis of Loops) - By GeeksForGeeks

We have discussed Asymptotic Analysis, Worst, Average and Best Cases and Asymptotic Notations in previous posts. In this post, analysis of iterative programs with simple examples is discussed.

1) O(1): Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

```
// set of non-recursive and non-loop statements
```

For example swap() function has O(1) time complexity.
A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
     // some O(1) expressions
}
```

2) O(n): Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}

for (int i = n; i > 0; i -= c) {
     // some O(1) expressions
}
```

3) O(n^c): Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have O(n^2) time complexity

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
       // some O(1) expressions
    }
}

for (int i = n; i > 0; i += c) {
    for (int j = i+1; j <=n; j += c) {
       // some O(1) expressions
}
```

For example Selection sort and Insertion Sort have O(n^2) time complexity.

4) O(Logn) Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

```
    for (int i = 1; i <=n; i *= c) {
        // some O(1) expressions
    }
    for (int i = n; i > 0; i /= c) {
        // some O(1) expressions
    }
```

For example Binary Search(refer iterative implementation) has O(Logn) time complexity.

5) O(LogLogn) Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced increased exponentially by a constant amount.

```
    // Here c is a constant greater than 1
    for (int i = 2; i <=n; i = pow(i, c)) {
        // some O(1) expressions
    }
    //Here fun is sqrt or cuberoot or any other constant root
    for (int i = n; i > 0; i = fun(i)) {
        // some O(1) expressions
    }
```

See this for more explanation.

How to combine time complexities of consecutive loops?

When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <=m; i += c) {
     // some O(1) expressions
}
for (int i = 1; i <=n; i += c) {
     // some O(1) expressions
}
Time complexity of above code is O(m) + O(n) which is O(m+n)
If m == n, the time complexity becomes O(2n) which is O(n).
```

How to calculate time complexity when there are many if, else statements inside loops?

Worst case time complexity is the most useful among best, average and worst. Therefore we need to consider worst case. We evaluate the situation when values in if-else conditions cause maximum number of statements to be executed.
For example consider the linear search function where we consider the case when element is present at the end or not present at all.
When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.

How to calculate time complexity of recursive functions?

Time complexity of a recursive function can be written as a mathematical recurrence relation. To calculate time complexity, we must know how to solve recurrences. We will soon be discussing recurrence solving techniques as a separate post.