

Tips on Analyzing the Time Complexity of Algorithms that use Nested Loops

- if you have two loops; **the outer** and **inner** loop, and they are dependent on the problem size **n**, **the statements in the inner loop will be executed $O(n^2)$ times:**

```
for ( int i = 0; i < n; i++ ) {  
    for ( int j = 0; j < n; j++ ) {  
        // these statements are executed  $O(n^2)$  times  
    }  
}
```

```
for ( int i = 0; i < n / 2; i++ ) {  
    for ( int j = 0; j < n / 3; j++ ) {  
        // these statements are also executed  $O(n^2)$  times  
        // since both loops loop  $O(n)$  times, and  
        //  $O(n) * O(n) = O(n^2)$   
    }  
}
```

Tips on Analyzing the Time Complexity of Algorithms that use Nested Loops

- if you have **triple-nested loops**, all of which are dependent on the problem size **n**, the statements in **the innermost loop will be executed $O(n^3)$ times**:

```
for ( int i = 0; i < n; i++ ) {  
    for ( int j = 0; j < n; j++ ) {  
        for ( int k = 0; k < n; k++ ) {  
            // these statements are executed  $O(n^3)$  times  
        }  
    }  
}
```

Tips on Analyzing the Time Complexity of Algorithms that use Nested Loops

imagine a case with doubly-nested loops where only the outer loop is dependent on the problem size n , and the inner loop always executes a constant number of times, say 3 times:

```
for ( int i = 0; i < n; i++ ) {  
    for ( int j = 0; j < 3; j++ ) {  
        // these statements are executed  $O(n)$  times  
    }  
}
```

In this particular case, the inner loop will execute exactly 3 times for each of the n iterations of the outer loop, and so the total number of times the statements in the innermost loop will be executed is $3n$ or $O(n)$ times, *not* $O(n^2)$ times.

Tips on Analyzing the Time Complexity of Algorithms that use Nested Loops

imagine a third case: you have **doubly nested loops**, and the **outer loop is dependent on the problem size n** , but the **inner loop is dependent on the current value of the index variable of the outer loop**:

```
for ( int i = 0; i < n; i++ ) {  
    for ( int j = 0; j < i; j++ ) {  
        // these statements are executed  $O(n^2)$  times  
    }  
}
```

Analyzing Code



Simple statement

The simple statement takes $O(1)$ time.

```
1 | int x= n + 12;
```

if condition

```
if (condition) {  
    sequence of statements 1  
}  
else {  
    sequence of statements 2  
}
```

the worst-case time is the slowest of the two possibilities: $\max(\text{time}(\text{sequence 1}), \text{time}(\text{sequence 2}))$. For example, if sequence 1 is $O(N)$ and sequence 2 is $O(1)$ the worst-case time for the whole if-then-else statement would be $O(N)$.

for/while loops

The loops take N time to complete and take $O(n)$.

```
1 | for(int i=0;i<n;i++)  
2 | {  
3 | ..  
4 | ..  
5 | }
```

Nested loops

If the nested loops contain M and N size, the cost is $O(MN)$

```
1 | for(int i=0;i<n;i++)  
2 | {  
3 |     for(int i=0;i<m;i++){  
4 |         ..  
5 |         ..  
6 |     }  
7 | }
```

Analyzing Code



Example 1

$O(N)$

```
1 | for(int i = 0; i < n; i++)  
2 |     sum++;
```

Example 2

$O(N)$

```
1 | for(int i = 0; i < n; i+=2)  
2 |     sum++;
```

Example 3

$O(N^2)$

```
1 | for(int i = 0; i < n; i++)  
2 |     for( int j = 0; j < n; j++)  
3 |         sum++;
```

Analyzing Code



Example 4

$O(N)$

```
1 | for(int i = 0; i < n; i+=2)
2 |     sum++;
3 |     for(int j = 0; j < n; j++)
4 |         sum++;
```

Example 5

$O(n^3)$

```
1 | for(int i = 0; i < n; i++)
2 |     for( int j = 0; j < n * n; j++)
3 |         sum++;
```

Example 6

$O(N^2)$

```
1 | for(int i = 0; i < n; i++)
2 |     for( int j = 0; j < i; j++)
3 |         sum++;
```

Analyzing Code



Example 7

$O(n^5)$

```
1 | for(int i = 0; i < n; i++)
2 |     for(int j = 0; j < n * n; j++)
3 |         for(int k = 0; k < j; k++)
4 |             sum++;
```

Example 8

$O(\log(n))$

```
1 | for(int i = 1; i < n; i = i * 2)
2 |     sum++;
```

Example 9

$\log(n)$

```
1 | while(n > 1){
2 |     n = n / 2;
3 | }
```


Analyzing Code

Example 10

Find $O(n)$ for the following:

$$7n - 3$$

$$8n^2 \log n + 5n^2 + n$$

Simple Rule: Drop lower order terms and constant factors.

- $7n - 3$ is $O(n)$
- $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

Analyzing Code

Example 11

```
for (int i=0;i< n*n ;i++)  
    for (int j=i;j< i*i;j++)  
        for (int k=0;k<n;k++){  
            Statement(s);  
        }
```

// these statement are executed $O(n^7)$ times

More Examples

On board



Recurrence Relation

A recurrence relation, $T(n)$, is a recursive function of an integer variable n .

Like all recursive functions, it has one or more recursive cases and one or more base cases.

Example:

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2T(n/2) + bn + c & \text{if } n > 1 \end{cases}$$

The portion of the definition that does not contain T is called the **base case** of the recurrence relation; the portion that contains T is called the **recurrent or recursive case**.

Forming Recurrence Relations

Example 1: Write the recurrence relation for the following method:

```
public void f (int n) {  
    if (n==0)  
        System.out.println(n)  
    else{  
        System.out.println(n) ;  
        f(n-1) ;  
    }  
}
```

Recursive Function to
print number from
Given input down to 0.

1. The base case is reached when $n == 0$.
2. When $n > 0$, the method performs two basic operations and then calls itself, using ONE recursive call, with a parameter $n - 1$.

Forming Recurrence Relations

Example 1: Write the recurrence relation for the following method:

```
public void f (int n) {  
    if (n==0)  
        System.out.println(n)  
    else{  
        System.out.println(n);  
        f(n-1);  
    }  
}
```

$$T(n) = \begin{cases} c & , n = 0 \\ b + T(n-1) & , n > 0 \end{cases}$$

Forming Recurrence Relations

Example 2: Write the recurrence relation for the following method:

:

```
long fibonacci (int n) { // Recursively calculates Fibonacci number
    if( n == 1 || n == 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

the recurrence relation is:

$$T(n) = \begin{cases} c & , n = 1 \text{ or } n = 2 \\ b + T(n - 1) + T(n - 2) & , n > 2 \end{cases}$$

Forming Recurrence Relations

Example 3: Write the recurrence relation for the following method:

```
long power (long x, long n) {  
    if(n == 0)  
        return 1;  
    else if(n == 1)  
        return x;  
    else if ((n % 2) == 0)  
        return power (x, n/2) * power (x, n/2);  
    else  
        return x * power (x, n/2) * power (x, n/2);  
}
```

The recurrence relation is:

$$T(n) = \begin{cases} c & , n = 0 \text{ or } n = 1 \\ b + 2T(n/2) & , n > 2 \end{cases}$$

Solving Recurrence Relations

Example1: Form and solve the recurrence relation for the running time of factorial method and hence determine its big-O complexity:

```
long factorial (int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial (n - 1);  
}
```

First: The recurrence relation is:

$$T(n) = \begin{cases} c & , n = 0 \\ b + T(n-1) & , n \geq 1 \end{cases}$$

Second: By Substitution

Solving Recurrence Relations

$$T(n) = b + T(n-1)$$

$$T(n-1) = b + T(n-1-1) = b + T(n-2)$$

$$\rightarrow T(n-1) = b + T(n-2)$$

$$T(n) = b + [b + T(n-2)] = 2b + T(n-2)$$

$$T(n) = 2b + T(n-2)$$

$$T(n-2) = b + T(n-2-1) = b + T(n-3)$$

$$\rightarrow T(n-2) = b + T(n-3)$$

$$T(n) = 2b + [b + T(n-3)]$$

$$T(n) = 3b + T(n-3)$$

$$T(n) = b + T(n-1)$$

$$T(n) = 2b + T(n-2)$$

$$T(n) = 3b + T(n-3)$$

.

$$T(n) = kb + T(n-k)$$

The base case is reached when $n - k = 0$
 $\rightarrow k = n$, we then have:

$$T(n) = nb + T(n-n) = nb + T(0)$$

$$T(n) = nb + T(0)$$

$$T(n) = nb + c, \text{ } c \text{ is constant}$$

Therefore the method factorial is $O(n)$

Solving Recurrence Relations

Example 2: Analysis The following recurrence relation : (determine its big-O complexity)

$$T(n) = \begin{cases} c & , n = 1 \\ 2T(n/2) + n & , n > 1 \end{cases}$$

Solution: By Substitution

Solving Recurrence Relations

$$T(n) = 2T(n/2) + n$$

$$T(n/2) = 2T(n/4) + n/2$$

$$T(n) = 2[2T(n/4) + n/2] + n = 4T(n/4) + 2n$$

$$T(n) = 4T(n/4) + 2n$$

$$T(n/4) = 2T(n/8) + n/4$$

$$T(n) = 4[2T(n/8) + n/4] + 2n = 8T(n/8) + 3n$$

$$T(n) = 8T(n/8) + 3n$$

.

.

$$T(n) = 2^k T(n/2^k) + kn$$

$$T(n) = 2^k T(n/2^k) + kn$$

The base case is reached when $n/2^k = 1 \rightarrow n = 2^k$, we then have:

$$T(n) = n T(1) + n \log_2 n$$

$$T(n) = nc + n \log_2 n, \text{ c is constant}$$

$$\text{Therefore } T(n) = O(n \log_2 n)$$

Definition: $\log_x B = A$ means $X^A = B$

Solving Recurrence Relations

Example 3: Analysis The following recurrence relation : (determine its big-O complexity)

$$T(n) = \begin{cases} d & , n=1 \\ 2T\left(\frac{n}{2}\right) + b & , n>1 \end{cases}$$

Solution: By Substitution

Solving Recurrence Relations

$$T(n) = 2T\left(\frac{n}{2}\right) + b$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + b$$

$$T(n) = 2[2T\left(\frac{n}{2^2}\right) + b] + b$$

$$T(n) = 2^2 T\left(\frac{n}{2^2}\right) + (2*b) + b$$

$$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + b$$

$$T(n) = 2^2 [2T\left(\frac{n}{2^3}\right) + b] + (2*b) + b$$

$$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + (2^2 * b) + (2*b) + b$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + (2^{k-1} * b) + (2^{k-2} * b) + \dots + 2^0 * b$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + b \sum_{i=0}^{k-1} 2^i$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + b \left[\frac{2^k - 1}{2 - 1} \right]$$

Base case : $T(d)=1 \rightarrow \frac{n}{2^k} = 1 \rightarrow n=2^k \rightarrow K=\log_2 n$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + b \left[\frac{2^k - 1}{2 - 1} \right]$$

$$T(n) = n T(1) + b \left[\frac{n-1}{1} \right]$$

$\rightarrow T(n) = nc + bn - b$, c, b constants

Therefore $T(n) = O(n)$

$$\sum_{k=0}^{n-1} x^k = \frac{x^n - 1}{x - 1} \quad (x \neq 1)$$

Extra Exercises

❖ For each of the following program fragments, give an analysis of the running time (Big-Oh will suffice).

```
1. sum = 0
   for(i = 0; i < n; i++)
       for( j = 0; j < n * n; j++)
           sum++;
```

```
2. sum = 0
   for(i = 0; i < n; i++)
       for(j = 0; j < i * i; j++)
           for( k = 0; k < j; k++)
               sum++;
```

Extra Exercises

- ❖ For each of the following program fragments, give an analysis of the running time (Big-Oh will suffice).

```
3. sum = 0
   for(i = 0; i < n; i++)
       for(j = 1; j < i * i; j++)
           if( j % i == 0)
               for(k = 0; k < j; k++)  <== A
                   sum++;
```

Hint: A will run every time j is a multiple of i

Extra Exercises

❖ For each of the following loops, give an analysis of the running time (Big-Oh will suffice).

- `for (int i = 0; i < n; i += c)
 statement(s);`
- `for (int i = 0; i < n; i *= c)
 statement(s);`
- `for (int i = 0; i < n * n; i += c)
 statement(s);`

Extra Exercises

❖ For each of the following loops, give an analysis of the running time (Big-Oh will suffice).


- ```
for (int i = 0; i < n; i += c)
 for (int j = 0; j < n; i += c)
 statement;
```

- ```
for (int i = 0; i < n; i += c)
    statement;
for (int i = 0; i < n; i += c)
    for (int j = 0; j < n; i *= c)
        statement;
```

Question?



“Success is the sum of small efforts, repeated day in and day out.”
Robert Collier



Reference: 1. Andrew Rosenberg Lecture Notes
2. Rajesh Rao and Dan Suciu Lecture Notes
3. Prof. Charles E. Leiserson Lecture Notes