**About this Module**

A look at the importance of complexity analysis and the structure of this module.

Analysis of algorithms is the most important step for software development. A well-written algorithm is not only time-efficient but also space-efficient. The complexity of algorithms is measured using asymptotic notations. Using these notations, we can determine if an algorithm performs better than the other one. Coding interviews often include questions regarding complexity analysis of algorithms, or they might ask you to implement the most efficient algorithms. To answer such questions you should be able to calculate the complexity of any algorithm.

This module includes challenges in which you will be asked to calculate the time complexity of algorithms with varying difficulty. These algorithms will fall under the following categories:

* Simple Nested Loop
* Nested Loop with Addition
* Nested Loop with Subtraction
* Nested Loop with Multiplication

Let’s get started!

# Common Complexity Scenarios

This lesson summarizes our discussion of complexity measures and includes some commonly used examples and handy formulae to help you with your interview.

**We'll cover the following**

* + [List of Important Complexities](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#List-of-Important-Complexities-)
    - [Simple for-loop with an increment of size 1](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#Simple-for-loop--with-an-increment-of-size-1-)
    - [For-loop with increments of size k](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#For-loop-with-increments-of-size-k-)
    - [Simple nested For-loop](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#Simple-nested-For-loop-)
    - [Nested For-loop with dependent variables](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#Nested-For-loop-with-dependent-variables-)
    - [Nested For-loop With Index Modification](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#Nested-For-loop-With-Index-Modification-)
    - [Loops with log(n) time complexity](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#Loops-with-log(n)-time-complexity-)

## List of Important Complexities [**#**](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#List-of-Important-Complexities-)

The following list shows some common loop statements and how much time they take to execute.

### Simple for-loop with an increment of size 1 [**#**](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#Simple-for-loop--with-an-increment-of-size-1-)

for (int x = 0; x < n; x++) {  
    //statement(s) that take constant time  
}

**Running time Complexity** = T(n) = (2n+2) + cn = (2+c) n + 2(2*n*+2)+*cn*=(2+*c*)*n*+2. Dropping the leading constants \Rightarrow n + 2⇒*n*+2. Dropping lower order terms \Rightarrow O(n)⇒*O*(*n*).

**Explanation**: Java for loop increments the value x by 1 in every iteration from 0 till n-1 ([0, 1, 2, …, n-1]). So n is first 0, then 1, then 2, …, then n-1. This means the loop increment statement x++ runs a total of n*n* times. The comparison statement x < n ; runs n+1*n*+1 times. The initialization x = 0; runs once. Summing them up, we get a running time complexity of the for loop of n + n + 1 + 1= 2n+2*n*+*n*+1+1=2*n*+2. Now, the constant time statements in the loop itself each run n*n* times. Supposing the statements inside the loop account for a constant running time of c*c* in each iteration, they account for a total running time of cn*cn* throughout the loop’s lifetime. Hence the running time complexity is (2n+2) + cn(2*n*+2)+*cn*.

### For-loop with increments of size k [**#**](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#For-loop-with-increments-of-size-k-)

for (int x = 0; x < n; x+=k) {  
    //statement(s) that take constant time  
}

**Runing Time Complexity** = 2 + n(\frac{2+c}{k})2+*n*(​*k*​​2+*c*​​) = O(n)*O*(*n*)

**Explanation**: The initialization x = 0; runs once. Then, x gets incremented by k until it reaches n. In other words, x will be incremented to [0, k, 2k, 3k, \cdots, (mk) < n0,*k*,2*k*,3*k*,⋯,(*mk*)<*n*]. Hence, the incrementation part x+=k of the for loop takes floor(\frac{n}{k})*floor*(​*k*​​*n*​​) time. The comparison part of the for loop takes the same amount of time and one more iteration for the last comparison. So this loop takes 1+1+\frac{n}{k}+\frac{n}{k} = 2 + \frac{2n}{k}1+1+​*k*​​*n*​​+​*k*​​*n*​​=2+​*k*​​2*n*​​ time. While the statements in the loop itself take c\times\frac{n}{k}*c*×​*k*​​*n*​​ time. Hence in total, 2 + \frac{2n}{k}+\frac{cn}{k} = 2 + n(\frac{2+c}{k})2+​*k*​​2*n*​​+​*k*​​*cn*​​=2+*n*(​*k*​​2+*c*​​) times, which eventually give us O(n)*O*(*n*).

### Simple nested For-loop [**#**](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#Simple-nested-For-loop-)

for (int i=0; i<n; i++){  
    for (int j=0; j<m; j++){  
        //Statement(s) that take(s) constant time  
    }  
}

**Running Time Complexity** = nm(2+c)+2+4n =O(nm)*nm*(2+*c*)+2+4*n*=*O*(*nm*)

**Explanation:** The inner loop is a simple for loop that takes (2m+2) + cm(2*m*+2)+*cm* time and the outer loop runs it n*n* times so it takes n((2m+2) + cm)*n*((2*m*+2)+*cm*)time . Additionally, the initialization, increment and test for the outer loop take 2n+22*n*+2 time so in total, the running time is n((2m+2) + cm)+2n+2 = 2nm+4n+cnm+2 = nm(2+c)+4n+2*n*((2*m*+2)+*cm*)+2*n*+2=2*nm*+4*n*+*cnm*+2=*nm*(2+*c*)+4*n*+2, which is O(nm)*O*(*nm*).

### Nested For-loop with dependent variables [**#**](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#Nested-For-loop-with-dependent-variables-)

for (int i=0; i<n; i++){  
    for (int j=0; j<i; j++){  
        //Statement(s) that take(s) constant time  
    }  
}

**Running Time Complexity** = O(n^2)*O*(*n*​2​​)

**Explanation:** The outer loop runs n*n* times and for each time the outer loop runs, the inner loop runs i times. So, the statements in the inner loop do not run at the first iteration of the outer loop since i is 0 then; they run once at the second iteration of the outer loop since i is equal to 11 at that point, then they run twice, then thrice, until i is n-1*n*−1. So, they run a total of c + 2c + 3c + \cdots + (n-1)c*c*+2*c*+3*c*+⋯+(*n*−1)*c* times = c\left(\sum\_{i=1}^{n-1} i \right) = c\frac{(n-1)((n-1)+1)}{2} = \frac{cn(n-1)}{2}*c*(∑​*i*=1​*n*−1​​*i*)=*c*​2​​(*n*−1)((*n*−1)+1)​​=​2​​*cn*(*n*−1)​​. The initialization of j in the inner for loop runs once in each iteration of the outer loop. So, this statement incurs a running time of n*n*. In the first iteration of the outer for loop, the j < i statement runs once, in the second iteration it runs twice and so on. So, it incurs a total running time of 1 + 2 + \cdots + n = \frac{n(n+1)}{2}1+2+⋯+*n*=​2​​*n*(*n*+1)​​. In each iteration of the outer loop, the j++ statement runs one less time than the j < i statement, so it accounts for a running time of 0 + 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}0+1+2+⋯+(*n*−1)=​2​​*n*(*n*−1)​​. So in total, the inner loop has a running time of \frac{cn(n-1)}{2} + \frac{n(n+1)}{2} +\frac{n(n-1)}{2}+n​2​​*cn*(*n*−1)​​+​2​​*n*(*n*+1)​​+​2​​*n*(*n*−1)​​+*n*. The outer loop initialization, test and increment operations account for a running time of 2n+22*n*+2. That means the entire script has a running time of 2n + 2 + \frac{cn(n-1)}{2} + n + \frac{n(n+1)}{2} +\frac{n(n-1)}{2}2*n*+2+​2​​*cn*(*n*−1)​​+*n*+​2​​*n*(*n*+1)​​+​2​​*n*(*n*−1)​​ which is O(n^2)*O*(*n*​2​​)

### Nested For-loop With Index Modification [**#**](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#Nested-For-loop-With-Index-Modification-)

for (int i=0; i<n; i++){  
    i\*=2;  
    for (int j=0; j<i; j++){  
        // Statement(s) that take(s) constant time  
    }  
}

**Running Time Complexity** = O(n)*O*(*n*)

**Explanation:** Notice that the outer loop index variable is modified in the loop’s body. The first column in the following table shows the value of i immediately after entering the outer for loop. The second column shows the modified value of i after the i\*=2; statement is run.

| Outer Loop | Inner Loop |
| --- | --- |
| i = 0 | i = 0\*2 = **0** |
| i = 1 | i = 1\*2 = **2** |
| i = 3 | i = 3\*2 = **6** |
| \cdots⋯ | \cdots⋯ |
| i = (n-1)(*n*−1) | i = (n-1)\times2(*n*−1)×2 = **2(n-1)** |

A pattern is hard to decipher here. So, let’s simplify things. In the outer loop, i is doubled and then incremented each time. If we ignore the increment part, we will be slightly over-estimating the number of iteration of the outer for loop. That is fine because we are looking for an upper bound on the worst-case running time (Big O).

If i*i* keeps doubling, it will get from 11 to n-1*n*−1 in roughly log\_2{(n-1)}*log*​2​​(*n*−1) steps. With this simplification, the outer loop index goes (approximately) 1, 2, 4, \cdots, 2^{log\_2(n-1)}1,2,4,⋯,2​*log*​2​​(*n*−1)​​. We’ve ignored the iteration with i=0*i*=0, but it wouldn’t affect the result in Big O. If you are interested, you can add 11 to all the steps in the following calculations. This sequence can also be written as 2^0, 2^1, 2^2, \cdots, 2^{log\_2{(n-1)}}2​0​​,2​1​​,2​2​​,⋯,2​*log*​2​​(*n*−1)​​. This series also gives the number of iterations of the inner for loop. Thus, the total number of iterations of the inner for loop is: 2^0 + 2^1 + 2^2 + \cdots + 2^{log\_2{(n-1)}} = 2^{log\_2(n-1)+1}-1=2^{log\_2{(n-1)}}2-1 = 2(n-1)-1 = 2n - 32​0​​+2​1​​+2​2​​+⋯+2​*log*​2​​(*n*−1)​​=2​*log*​2​​(*n*−1)+1​​−1=2​*log*​2​​(*n*−1)​​2−1=2(*n*−1)−1=2*n*−3.

Therefore, the running time of the inner for loop is 2 (2n-3) + 2 + c(2n-3)2(2*n*−3)+2+*c*(2*n*−3) where c*c* is the running time of the statements in the body of the inner loop. This simplifies to 2n(2+c)-3c-42*n*(2+*c*)−3*c*−4. The contribution from the initialization, test, and increment operations of the outer for loop is 2log\_2{(n-1)}+22*log*​2​​(*n*−1)+2. So, the total running time is 2n(2+c)-3c-4+2log\_2{(n-1)}+22*n*(2+*c*)−3*c*−4+2*log*​2​​(*n*−1)+2. The term linear in n*n* dominates the others, and the time complexity is O(n)*O*(*n*).

Note that we could have done a rough approximation saying that the outer loop runs at most n*n* times, the inner loop iterates at most n*n* times each iteration of the outer for loop. That would lead us to conclude that the total running time is O(n^2)*O*(*n*​2​​). Mathematically that is correct, but it isn’t a tight bound.

### Loops with log(n) time complexity [**#**](https://www.educative.io/module/lesson/big-o-notation/JENPYQWgp3g#Loops-with-log(n)-time-complexity-)

i = //constant  
n = //constant  
k = //constant  
while (i < n){  
    i\*=k;  
    // Statement(s) that take(s) constant time  
}

**Running Time Complexity** = \log\_k(n)log​*k*​​(*n*) = O(\log\_k(n))*O*(log​*k*​​(*n*))

**Explanation:** A loop statement that multiplies/divides the loop variable by a constant such as the above takes \log\_k(n)log​*k*​​(*n*) time because the loop runs that many times. Let’s consider an example where n = 16, and k = 2:

| i | Count |
| --- | --- |
| 1 | 1 |
| 2 | 2 |
| 4 | 3 |
| 8 | 4 |
| 16 | 5 |

\log\_k(n) = \log\_2(16) = 4log​*k*​​(*n*)=log​2​​(16)=4

Now that you have all the tools necessary to solve the time complexity problems let’s look at some exercises in the next few lessons.

# Useful Formulae

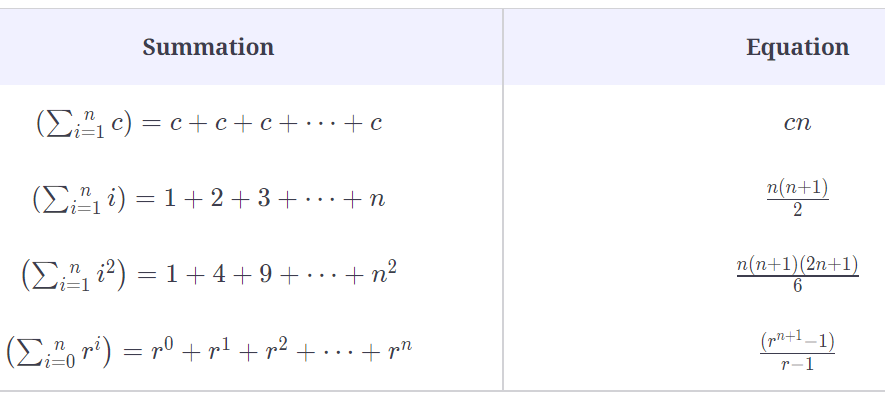
In this lesson, we'll study some mathematical formulae that would make calculating time complexity easier!

**We'll cover the following**

* + [Formulae](https://www.educative.io/module/lesson/big-o-notation/myJ6LYKG2gA#Formulae-)
  + [General Tips](https://www.educative.io/module/lesson/big-o-notation/myJ6LYKG2gA#General-Tips-)

## Formulae [**#**](https://www.educative.io/module/lesson/big-o-notation/myJ6LYKG2gA#Formulae-)

Here is a list of handy formulas which can be helpful when calculating the Time Complexity of an algorithm:



## General Tips [**#**](https://www.educative.io/module/lesson/big-o-notation/myJ6LYKG2gA#General-Tips-)

1. Every time a list or array gets iterated over c \times length*c*×*length* times, it is most likely in O(n)*O*(*n*) time.
2. When you see a problem where the number of elements in the problem space gets halved each time, it will most probably be in O(log n)*O*(*logn*) runtime.
3. Whenever you have a single nested loop, the problem is most likely in quadratic time.

In the next lesson, we will discuss some common scenarios and how you can calculate their complexities.

# Solution Review: Big O of a Nested Loop with Addition

This review provides a detailed analysis of the time complexity of the Nested Loop with Addition problem!

**We'll cover the following**

* + [Given Code](https://www.educative.io/module/lesson/big-o-notation/qV6Jp2AkWGp#Given-Code-)
  + [Solution Breakdown](https://www.educative.io/module/lesson/big-o-notation/qV6Jp2AkWGp#Solution-Breakdown-)
    - [Time Complexity](https://www.educative.io/module/lesson/big-o-notation/qV6Jp2AkWGp#Time-Complexity-)
  + [Illustration](https://www.educative.io/module/lesson/big-o-notation/qV6Jp2AkWGp#Illustration-)

## Given Code [**#**](https://www.educative.io/module/lesson/big-o-notation/qV6Jp2AkWGp#Given-Code-)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

class NestedLoop {

  public static void main(String[] args) {

    int n = 10; // 1 step --> Note: n can be anything. This is just an example

    int sum = 0; // 1 step

    double pie = 3.14; // 1 step

    for (int var = 0; var < n; var = var + 3) { // n/3 steps

      System.out.println("Pie: " + pie); // n/3 steps

      for (int j = 0; j < n; j = j + 2) {  // (n/3 \* n/2) steps

        sum++;   // (n/3 \* n/2) steps

        System.out.println("Sum = " + sum);  // (n/3 \* n/2) steps

     }

   }

  }

}





Run

Save

Reset

## Solution Breakdown [**#**](https://www.educative.io/module/lesson/big-o-notation/qV6Jp2AkWGp#Solution-Breakdown-)

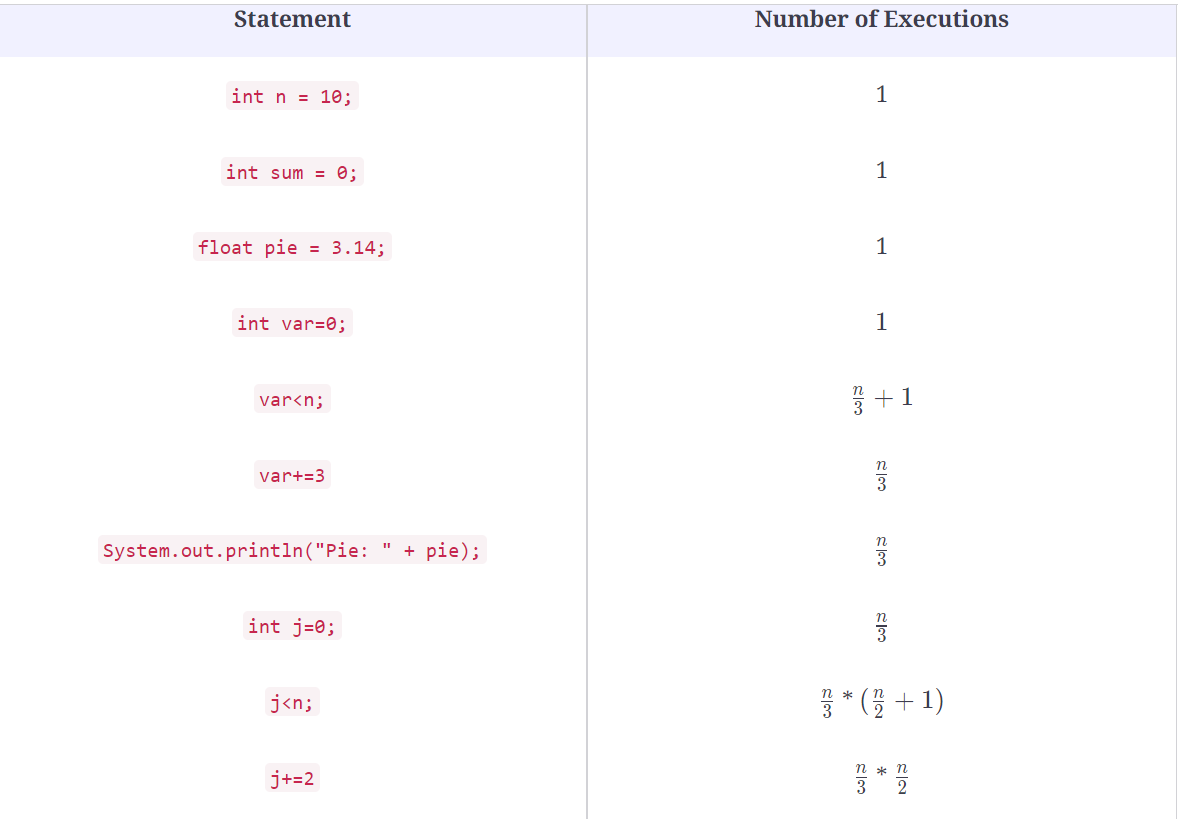
The first for loop on line **7** can be broken down into 3 parts:

* initialization
* comparison
* incrementation

Since the initialization (int var = 0) only happens once in the entire program, it takes 1 unit of time. The comparison (var < n) gets executed (\frac{n}{3} + 1)(​3​​*n*​​+1) times and the increment runs \frac{n}{3}​3​​*n*​​ times.

Similarly, (int j = 0) runs \frac{n}{3}​3​​*n*​​ times , the comparison (j < n) runs \frac{n}{3} \times \{\frac{n}{2}\ + 1\}​3​​*n*​​×{​2​​*n*​​ +1} (the test case runs once more than the whole loop where boundary check fails!) and the increment (j = j + 2) gets executed \frac{n}{2}​2​​*n*​​ times for each iteration of the outer loop–which makes it run a total of \frac{n}{3} \times \frac{n}{2} = \frac{n^2}{6}​3​​*n*​​×​2​​*n*​​=​6​​*n*​2​​​​ times.

See the following table for a more detailed line-by-line analysis of the calculation of time complexity.





### Time Complexity [**#**](https://www.educative.io/module/lesson/big-o-notation/qV6Jp2AkWGp#Time-Complexity-)

\text{Running Time Complexity} = 1 + 1 + 1 + 1 + \frac{n}{3} +1 + \frac{n}{3} + \frac{n}{3} + \frac{n}{3} \times (\frac{n}{2} + 1) + (\frac{n}{3} \times \frac{n}{2}) + (\frac{n}{3} \times \frac{n}{2}) + (\frac{n}{3} \times \frac{n}{2})Running Time Complexity=1+1+1+1+​3​​*n*​​+1+​3​​*n*​​+​3​​*n*​​+​3​​*n*​​×(​2​​*n*​​+1)+(​3​​*n*​​×​2​​*n*​​)+(​3​​*n*​​×​2​​*n*​​)+(​3​​*n*​​×​2​​*n*​​)

= 5 + \frac{5n}{3} + \frac{4n^2}{6}=5+​3​​5*n*​​+​6​​4*n*​2​​​​

= 5 + \frac{5n}{3} + \frac{2n^2}{3}=5+​3​​5*n*​​+​3​​2*n*​2​​​​

= \frac{15 + 5n + 2n^2}{3}=​3​​15+5*n*+2*n*​2​​​​

Now to find the Big O complexity,

1. Drop the leading constants => n + n^2*n*+*n*​2​​
2. Drop lower order terms => n^2*n*​2​​

Hence, Big O Time Complexity: **O(n2)**

## Illustration [**#**](https://www.educative.io/module/lesson/big-o-notation/qV6Jp2AkWGp#Illustration-)

Running Time Complexity0int n = 10; int sum = 0;double pie = 3.14;for (int var = 0; var < n; var = var + 3) { System.out.println("Pie: " + pie); for (int j = 0; j < n; j = j + 2) { sum++; System.out.println("Sum = " + sum); }}

We'll dry run this code to calculate its running time complexity.

**1** of 16

In the next lesson, let’s try calculating the time complexity of a for loop with subtraction.

# Solution Review: Big O of Nested Loop with Multiplication

This review provides a detailed analysis of the different ways to solve the Big O of Nested Loop with Multiplication.

**We'll cover the following**

* + [Given Code](https://www.educative.io/module/lesson/big-o-notation/B12JpONVzqJ#Given-Code-)
  + [Explanation](https://www.educative.io/module/lesson/big-o-notation/B12JpONVzqJ#Explanation)
    - [Time Complexity](https://www.educative.io/module/lesson/big-o-notation/B12JpONVzqJ#Time-Complexity)

## Given Code [**#**](https://www.educative.io/module/lesson/big-o-notation/B12JpONVzqJ#Given-Code-)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

class NestedLoop {

    public static void main(String[] args) {

        int n = 10; // O(time complexity of the called function)

        int sum = 0; //O(1)

        double pie = 3.14; //O(1)

        int var = 1;

        while(var < n) {  // O(log n)

            System.out.println("Pie: " + pie); // O(log n)

            for (int j = 0; j < var; j++) {  // 2n

                sum++;  //  (2n-1)

            }

            var \*= 2; // O(log n)

        } //end of while loop

        System.out.println("Sum: " + sum); //O(1)

    } //end of main

} //end of class





Run

Save

Reset

## Explanation[**#**](https://www.educative.io/module/lesson/big-o-notation/B12JpONVzqJ#Explanation)

The answer is O(n)*O*(*n*). Have a look at the slides below for an in-depth explanation of the answer.

Total for loop iterations= 2^(k+1)-1k is in O(log\_2(n))= 2^(log\_2(n)+1)-1= 2^(ceil(log\_2(n))-1<= 2(n - 1) - 1<= 2n - 3

Let's simplify the equation a bit (read the explanation below for details)

**43** of 61

### Time Complexity[**#**](https://www.educative.io/module/lesson/big-o-notation/B12JpONVzqJ#Time-Complexity)

The above slides give a detailed, step-by-step analysis of the code. Here, we provide a more summarized version.

The outer loop here runs log(n)*log*(*n*) times. In the first iteration of the outer loop, the body of the inner loop runs once. In the second iteration, it runs twice, and so on. The number of executions of the body of the inner loop increases in powers of 2. So, if k*k* is the number of iterations of the outer loop, the body of the inner loop runs a total of 1+2+4+8+\cdots+2^k1+2+4+8+⋯+2​*k*​​ times. This [geometric series](https://en.wikipedia.org/wiki/1_%2B_2_%2B_4_%2B_8_%2B_%E2%8B%AF) sums to 2^{k+1}-12​*k*+1​​−1. The inner loop condition requires that in the last time the inner loop runs, it runs at most n*n* times. This requires 2^k < n2​*k*​​<*n*, i.e., k < log\_2n*k*<*log*​2​​*n*, or k = \lfloor log\_2n \rfloor*k*=⌊*log*​2​​*n*⌋. This means that the geometric series sum to 2^{\lfloor log\_2n \rfloor+1} -12​⌊*log*​2​​*n*⌋+1​​−1 or 2^{\lceil log\_2n \rceil} -12​⌈*log*​2​​*n*⌉​​−1. In other words, the sum is **at least** n-1*n*−1. But we need an upper bound on this value, for Big O.

As you vary the value of n*n*, you will notice the following behavior:

| *n* | **Iterations of inner loop** |
| --- | --- |
| 1 | 0 |
| 2 | 1 |
| 3 | 3 |
| 4 | 3 |
| 5 | 7 |
| 6 | 7 |
| 7 | 7 |
| 8 | 7 |
| 9 | 15 |
| 10 | 15 |
| 11 | 15 |
| 12 | 15 |
| 13 | 15 |
| 14 | 15 |
| 15 | 15 |
| 16 | 15 |
| 17 | 31 |

The number of iterations of the inner loop changes on integer powers of 2, as you would expect with the \lceil n \rceil⌈*n*⌉ exponent. At n = 4*n*=4, the number of iteration is 33, at n = 8*n*=8, it is 77 etc. So, on values of n that are integer powers of 22, the number of iterations is n - 1*n*−1. That conforms to our lower bound on the number of iterations. The upper bound is evident on n = 2^i + 1*n*=2​*i*​​+1, where i = 1, 2, 3, 4, ...*i*=1,2,3,4,..., i.e., n = 3, 5, 9, 17, ...*n*=3,5,9,17,... You will notice that at at these values of n*n*, the number of iterations of the inner loop is one less than the next higher integer power of 2, i.e., 2(n - 1) - 1 = 2n - 32(*n*−1)−1=2*n*−3.

Thus, the number of iterations of the inner loop body is at least n - 1*n*−1 and at most 2n - 32*n*−3.

As described in the slides, the outer for loop statements account for 4\lceil log\_2 n \rceil4⌈*log*​2​​*n*⌉, and the total contribution of the inner loop is at least 8n - 128*n*−12. Thus, the total running time is at least 4\lceil log\_2 n \rceil + 8n - 124⌈*log*​2​​*n*⌉+8*n*−12. Dropping the constants and the lower order terms gives us n*n*.

Hence, the Big O Time Complexity is O(n)*O*(*n*)

In the next lesson, let’s solve another challenge involving nested loops with multiplication.