**Big O Notation for Interviews**

# ****Why not just measure the runtime of the programs?****

We use Big O notation to measure the efficiency of the programs. When talking about efficiency is all about the execution time of the program. If you wondering why not just start the stopwatch when we start our program and stop when it’s done, please keep reading.

Why just don’t measure the runtime of the programs?

public static void print1000() {  
 Instant start = Instant.now();  
 for(int i = 0; i < 1000; i++) {  
 System.out.println("Hello, World");  
 }

Instant finish = Instant.now();  
 long timeElapsed = Duration.between(start, finish).toMillis();  
 System.out.println("Milliseconds: " + timeElapsed);  
}

Above an example of measuring the runtime of the execution for this method. I’m getting **Milliseconds: 65**and sometimes more or less. If you will try to execute this method on your machine, probably, you will get a different result. Because your machine is different than mine. It’s hard to measure the exact runtime for a program because in different machines it will be different. it depends on the speed of the processor and other parts of your machine. So instead of measuring runtime with seconds, **we use Big O notation to measure how fast runtime will grow based on input and considering the worst-case scenario.**

# ****Big O notation measurement is based on input****

If we would compare the runtime of the program, we would measure in seconds or milliseconds. For Big O notation there is **n**which represents the size of the input. So we can say things like

* Runtime grows on the order of the size of input —**O(n)**
* Runtime growth is quadratic based on the size of the input —**O(n²)**
* Runtime growth is constant for any size of input — **O(1)**

and so on.

Let’s see examples(Java):

public void printFirstElement(String[] str) {  
 System.out.println(str[0]);  
}

The time complexity for the above example is **O(1) — constant time** because it doesn’t really matter how big or small our input(String[] str), it will take constant steps for the program to execute.

public void iterateOver(int[] numbers) {  
 for (int i = 0; i < numbers.length; i++) {  
 System.out.println(numbers[i]);  
 }  
}

The time complexity of this above example is **O(n) — linear time**because we print a number of times or we can say we take a number of steps which same as the size of our input **n.**If numbers input will have 1 element we will need to take one step and if numbers will have 1000 elements then we need to take 1000 steps. So we need to do **n steps** where **n is our input**(size of our input).

public boolean isSevenThere(int[] numbers) {  
 for (int i = 0; i < numbers.length; i++) {  
 if (numbers[i] == 7) {  
 return true;  
 }  
 }

return false;  
}

The time complexity of this program is **O(n) — linear time.**Wait a minute Big O notation depends on input right? If our input array will look like this [7, 2, 3, 4, 5, 9] it will take only one step because our method will return true in the first iteration, and if our input array [8, 3, 4, 1, 5, 7] then it will be O(n) because we need iterate over the whole array till we reach the last element where is 7 and we will do n steps. Yes, that’s correct.**An important thing to remember is that Big O notation considers the worst-case scenario**. In our example worst-case scenario where 7 is the last element of an array or it doesn’t exist then we need to take n steps for our program which is O(n) — linear time.

public void sayHello(int num) {  
 for (int i = 0; i < num; i++) {  
 System.out.println("Hello");  
 }

for (int i = 0; i < num; i++) {  
 System.out.println("Hi");  
 }  
}

What’s the time complexity of this program? O(2n)? No, it’s **O(n) — linear time**even though actually it will do twice more steps every time. Thing is that printing 2 times more not dependent on input at all so we can ignore this kind of constant which will not depend on input size n.

public void sayHello(int num) {  
 System.out.println("Hello");  
 System.out.println("Hello");  
 System.out.println("Hello");  
 System.out.println("Hello");  
 System.out.println("Hello"); for (int i = 0; i < num; i++) {  
 System.out.println("Hello");  
 }  
}

We have here **O(n) — linear time.**We ignore parts that are not related to input at all. In our case, the worst case is when we have huge input, let’s say num is 10000 and 5 static prints above will become insignificant compared to the loop part where input will decide the number of steps.

public void printStr(String str) {  
 for (int i = 0; i < 100; i++) {  
 System.out.println(str);  
 }  
}

Loop in the program will not always mean O(n)**.**The time complexity of this program is **O(1) — constant time.**Even though we are printing input but the number of steps is constant and hardcoded as 100 and doesn’t matter what kind of input will come it will always take 100 steps.

public boolean isSumTarget(int numArr[], int target) {  
 for(int i = 0; i < numArr.length; i++) {  
 for(int j = i + 1; j < numArr.length; j++) {  
 if(numArr[i] + numArr[j] == target) {  
 return true;  
 }  
 }  
 }  
   
 return false;   
}

The time complexity for this program is **O(n²) — quadratic.**We are iterating over our array n time so n \* n times.

*Some common time complexities:  
- O(1) — constant time  
- O(n) — linear time  
- O(log n) — logarithmic time (binary search)  
- O(n²) — quadratic time  
- O(n³) — cubic time*