The Idiots Guide To Big O Notation

corejava
interview questions . com

BIG O NOTATION

I hate big O notation. For as long as I can remember it's been my biggest achilles heel. It's just something I've never managed to successfully motivate myself to learn about despite knowing it's going to come up in every single interview. I'll get asked to implement an algorithm which I'll do via brute force to start with: "What is the Big O of that?". "I don't know Big O but I know it's slow".

It's fairly obvious, but this is the wrong approach. Take the time to really read this chapter and maybe even do some extra research. This seems to be a question that comes up consistently at interviews so it's worth taking the time.

What on earth is Big O?

Big O is the way of measuring the efficiency of an algorithm and how well it scales based on the size of the dataset. Imagine you have a list of 10 objects, and you want to sort them in order. There's a whole bunch of algorithms you can use to make that happen, but not all algorithms are built equal. Some are quicker than others but more importantly the speed of an algorithm can vary depending on how many items it's dealing with. Big O is a way of measuring how an algorithm scales. Big O references how **complex** an algorithm is.

Big O is represented using something like O(n). The O simply denoted we're talking about big O and you can ignore it (at least for the purpose of the interview). *n* is the thing the complexity is in relation to; for programming interview questions this is almost always the size of a collection. The complexity will increase or decrease in accordance with the size of the data store.

Below is a list of the Big O complexities in order of how well they scale relative to the dataset.

O(1)/Constant Complexity: Constant. This means irrelevant of the size of the data set the algorithm will always take a constant time.

1 item takes 1 second, 10 items takes 1 second, 100 items takes 1 second.

It always takes the same amount of time.

O(log n)/Logarithmic Complexity: Not as good as constant, but still pretty good. The time taken increases with the size of the data set, but not proportionately so. This means the algorithm takes longer per item on smaller datasets relative to larger ones.

1 item takes 1 second, 10 items takes 2 seconds, 100 items takes 3 seconds.

If your dataset has 10 items, each item causes 0.2 seconds latency. If your dataset has 100, it only takes 0.03 seconds extra per item. This makes log n algorithms very scalable.

O(n)/Linear Complexity: The larger the data set, the time taken grows proportionately.

1 item takes 1 second, 10 items takes 10 seconds, 100 items takes 100 seconds.

O(n log n): A nice combination of the previous two. Normally there's 2 parts to the sort, the first loop is O(n), the second is O(log n), combining to form O(n log n).

1 item takes 2 seconds, 10 items takes 12 seconds, 100 items takes 103 seconds.

O(n ^2^)/Quadratic Complexity: Things are getting extra slow.

1 item takes 1 second, 10 items takes 100, 100 items takes 10000.

O(2 ^n^): Exponential Growth! The algorithm takes twice as long for every new element added.

1 item takes 1 second, 10 items takes 1024 seconds, 100 items takes 1267650600228229401496703205376 seconds.

It is important to notice that the above is not ordered by the best to worst complexity. There is no "best" algorithm, as it completely hinges on the size of the dataset and the task at hand. It is also important to remember the code maintenance cost; a more complex algorithm may result in an incredibly quick sort, but if the code has become unmaintainable and difficult to debug is that the right thing to do? It depends on your requirements.

There is also a variation in complexity within the above complexities. Imagine an algorithm which loops through a list exactly two times. This would be O(2n) complexity, as it's going through your lists length (n) twice!

Why does this matter?

Simply put: *an algorithm that works on a small dataset is not guaranteed to work well on a large dataset*. Just because something is lightning fast on your machine doesn't mean that it's going to work when you scale up to a serious dataset. You need to understand exactly what your algorithm is doing, and what it's big O complexity is, in order to choose the right solution.

There are three things we care about with algorithms: **best case, worst case and expected case**. In reality we only actually care about the latter two, as we're a bunch of pessimists. If you ask an algorithm to sort a pre-sorted list it's probably going to do it much faster than a completely jumbled list. Instead we want to know the worst case (the absolutely maximum amount of steps the algorithm could take) and the expected case (the likely or average number of steps the algorithm could take). Just to add to the fun, these can and often are different.

Examples

Hopefully you're with me so far, but let's dive into some example algorithms for sorting and searching. The important thing is to be able to explain what complexity an algorithm is. Interviewers love to get candidates to design algorithms and then ask what the complexity of it is.

O(1)

Irrelevant of the size, it will always return at constant speed. The javadoc for Queue states that it is *"constant time for the retrieval methods (`peek`, `element`, and `size`)".* It's pretty clear why this is the case. For peek, we are always returning the first element which we always have a reference to; it doesn't matter how many elements follow it. The size of the list is updated upon element addition/removal, and referencing this number is just one operation to access no matter what the size of the list is.

O(log n)

The classic example is a Binary search. You're a massive geek so you've obviously alphabetised your movie collection. To find your copy of "Back To The Future", you first go to the middle of the list. You discover the middle film is "Meet The Fockers", so you then head to the movie in between the start and this film. You discover this is "Children of Men". You repeat this again and you've found "Back to the Future". There's a great interactive demo of binary search available online at Armstrong State University.

Although adding more elements will increase the amount of time it takes to search, it doesn't do so proportionally. Therefore it is O(log n).

O(n)

As discussed in the collections chapter, LinkedLists are not so good (relatively speaking) when it comes to retrieval. It actually has a complexity of O(n) for the **worst case**: to find an element T, which is the last element in the list, it is necessary to navigate the entire list of n elements. As the number of elements increases so does the access time in proportion.

O(n log n)

The best example of O(n log n) is a **merge sort**. This is a divide and conquer algorithm. Imagine you have a list of integers. We divide the list in two again and again until we are left with with a number of lists with 1 item in: each of these lists is therefore sorted.We then merge each list with it's neighbour (comparing the first elements of each every time). We repeat this with the new composite list until we have our sorted result.
To explain why this is O(n log n) is a bit more complex. In the above example of 8 numbers, we have 3 levels of sorting:

- 4 list sorts when the list sizes are 2
- 2 list sorts when the list sizes are 4
- 1 list sort when the list size is 8

Now consider if I were to double the number of elements to 16: this would only require one more level of sorting. Hopefully you recognise this is a log n scale.

However, on each level of sorting a total of n operations takes place (look at the red boxes in the diagram above). This results in (n * log n) operations, e.g. O(n log n).

O(n^2)

The Bubble Sort algorithm is everyone's first algorithm in school, and interestingly it is quadratic complexity. If you need a reminder; we go through the list and compare each element with the one next to it, swapping the elements if they are out of order. At the end of the first iteration, we then start again from the beginning, with the caveat that we now know the last element is correct.

Imagine writing the code for this; it's two loops of n iterations.

```
public int[] sort(int[] toSort){
    for (int i = 0; i < toSort.length -1; i++) {
        boolean swapped = false;
        for (int j = 0; j < toSort.length - 1 - i; j++) {
            if(toSort[j] > toSort[j+1]){
                swapped = true;
                int swap = toSort[j+1];
                toSort[j + 1] = toSort[j];
                toSort[j] = swap;
            }
        }
        if(!swapped)
            break;
    }
        return toSort;
}
```

This is also a good example of best vs worst case. If the list to sort is already sorted, then it will only take one iteration (e.g. n) to sort. However, in the worst case we have to go through the list n times and each time looping another n items (less how many loops we have done before) which is slow.

You may notice that it's technically less than n^2 as the second loop decreases each time. This gets ignored because as the size of the data set increases this impact of this becomes more and more marginal and tends towards quadratic.

O(2^n)

Exponential growth! Any algorithm where adding another element dramatically increases the processing time. Take for example trying to find combinations; if I have a list of 150 people and I would like to find every combination of groupings; everyone by themselves, all of the groups of 2 people, all of the groups of 3 people etc. Using a simple program which takes each person and loops through the combinations, if I add one extra person then it's going to increase the processing time exponentially. Every new element will double processing time.

In reality O(2^n) algorithms are not scalable.

How to figure out Big O in an interview

This is not an exhaustive list of Big O. Much as you can O(n^2), you can also have O(n^ 3^) (imagine bubble sort but with an extra loop). What the list on this page should allow you to do is have a stab in the dark at figuring out what the big O of an algorithm is. If someone is asking you this during an interview they probably want to see how you try and figure it out. Break down the loops and processing.

- Does it have to go through the entire list? There will be an *n* in there somewhere.
- Does the algorithm's processing time increase at a slower rate than the size of the data set? Then there's probably a *log n* in there.
- Are there multiple loops? You're probably looking at *n^2 or n^3*.
- Is access time constant irrelevant of the size of the dataset? *O(1)*

Sample question

I have an array of the numbers 1 to 100 in a random number. One of the numbers is missing. Write an algorithm to figure out what the number is and what position is missing.

There are many variations of this question all of which are very popular. To calculate the missing number we can add up all the numbers we do have, and subtract this from the expected answer of the sum of all numbers between 1 and 100. To do this we have to iterate the list once. Whilst doing this we can also note which spot has the gap.

```
public class BlankFinder{

public void findTheBlank(int[] theNumbers) {
        int sumOfAllNumbers = 0;
        int sumOfNumbersPresent = 0;
        int blankSpace = 0;

        for (int i = 0; i < theNumbers.length; i++) {
            sumOfAllNumbers += i + 1;
            sumOfNumbersPresent += theNumbers[i];
            if (theNumbers[i] == 0)
                blankSpace = i;
        }

        System.out.println("Missing number = " + (sumOfAllNumbers -
```

sumOfNumbersPresent) + “ at location ” + blankSpace +“ of the array”);

```
     }

    public static void main(String[] args) {
            new BlankFinder().findTheBlank(new int[]
{7,6,0,1,3,2,4});
       }
       //Missing number = 5 at location 2 of the array
}
```

Caveat: you can also calculate sumOfAllNumbers using (theNumbers.length+1) * (theNumbers.length) / 2.0). *I would never remember that in an interview though.*

What is the big O of your algo?

Our algorithm iterates through our list once, so it’s **O(n).**