FREE eBook

LEARNING

big-o

Free unaffiliated eBook created from
Stack Overflow contributors.

#big-o

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: big-o

It is an unofficial and free big-o ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official big-o.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with big-o

Remarks

This section provides an overview of what big-o is, and why a developer might want to use it.

It should also mention any large subjects within big-o, and link out to the related topics. Since the Documentation for big-o is new, you may need to create initial versions of those related topics.

Examples

What is Big-O Notation?

Big-O notation is a notation used to talk about the long-term growth rates of functions. It's often used in the analysis of algorithms to talk about the runtime of an algorithm or related concepts like space complexity.

In common usage, big-O notation is used to talk about how an algorithm's runtime scales as a size of the input. For example, we'd say that selection sort has a runtime of $O(n^2)$ because the runtime grows quadratically as a function of the size of the array to sort. That is, if you double the size of the input, the runtime of selection sort should roughly double. When using big-O notation, the convention is to drop coefficients and to ignore lower-order terms. For example, while technically it is not wrong to say that binary search runs in time $O(2 \log_2 n + 17)$, it's considered poor style and it would be better to write that binary search runs in time $O(\log n)$.

Formally, big-O notation is used to quantify the long-term behavior of a function. We say that $f(n) = O(g)$ (sometimes denoted $f(n) \in O(g(n))$ in some sources) if there are fixed constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. This formal definition accounts for why we don't care about low-order terms (they can be subsumed by making c larger and increasing n_0) and constant factors (the c term absorbs them). This formal definition is often used in the rigorous analysis of algorithms but is rarely used colloquially.

Calculating Big-O for your code

One way to calculate the Big-O value of a procedure you've written is to determine which line of code runs the most times in your function, given your input size *n*. Once you have that number, take out all but the fastest-growing terms and get rid of the coefficents - that is the Big-O notation of your function.

For instance, in this function, each line runs exactly once, and for the same amount of time regardless of how large `a` is:

```
int first(int[] a){
   printf("Returning the first element of a");
   return a[0];
}
```

The function itself might take 1 millisecond ((1 ms) * n^0) or 100 milliseconds ((100 ms) * n^0) - the exact value would depend on the power of the computer involved and what `printf()` is printing to. But because those factors don't change with the size of `a`, they don't matter for Big-O calculations - they are constant coefficients, which we remove. Hence, this function has a Big-O value of *O(1)*.

In this function, line 3 (`sum += a[i];`) runs once for each element in `a`, for a total of `a.length` (or *n*) times:

```
int sum(int[] a){
   int sum = 0;
   for (int i = 0; i < a.length; i++){
        sum += a[i];
   }
   return sum;
}
```

The statements `i++` and `i < a.length` each also run *n* times - we could have picked those lines, but we don't have to. Also, `int sum = 0;`, `int i = 0`, and `return sum;` each run once, which is fewer than `n` times - we ignore those lines. It doesn't matter how long `sum += a[i]` takes to run - that is a coefficent that depends on the power of the computer - so we remove that coefficient. Hence, this function is *O(n)*.

If there are multiple code paths, big-O is usually calculated from the worst case. For instance, even though this function can perhaps exit immediately no matter how large `a` is (if `a[0]` is `0`), a case still exists that causes line 6 to run `a.length` times, so it's still O(n):

```
int product(int[] a){
    int product = 0;
    for (int i = 0; i < a.length; i++){
        if (a[i] == 0)
            return 0;
        else
            product *= a[i];
    }
    return product;
}
```

Read Getting started with big-o online: https://riptutorial.com/big-o/topic/1340/getting-started-with-big-o

Chapter 2: Calculating Big-O

Examples

O(n) functions.

Functions that are `O(n)` increase the number of operations linearly, as the input gets very large. A simple example of a function that is O(n) would be the linear search algorithm, which runs once for the size of the input.

The following pseudo-code would be `O(n)`, because it will always be bounded above by the input size, as the algorithm will never run more times then the input size.

```
function LinearSearch (SearchArray, SearchFor)
    for each element in SearchArray
        if the element is SearchFor
            return the index of element
```

Read Calculating Big-O online: https://riptutorial.com/big-o/topic/4307/calculating-big-o

Credits

S. No	Chapters	Contributors
1	Getting started with big-o	Community, intboolstring, templatetypedef, TheHansinator
2	Calculating Big-O	intboolstring