Big O Notation

Analyzing the runtime of your algorithm

Eveline Visee

February 22, 2018

WonderDojo meetup @ FooCafé Stockholm

Table of contents

What is Big O Notation?

Runtime

You wrote an algorithm. It processes a list of numbers. How long will the computer take to run the algorithm on:

- a list of 1000 numbers?

Runtime

You wrote an algorithm. It processes a list of numbers. How long will the computer take to run the algorithm on:

- a list of 1000 numbers?
- a list of 2000 numbers?

Runtime

You wrote an algorithm. It processes a list of numbers. How long will the computer take to run the algorithm on:

- a list of 1000 numbers?
- a list of 2000 numbers?
- a list of 100.000 numbers?

Runtime

You wrote an algorithm. It processes a list of numbers. How long will the computer take to run the algorithm on:

- a list of 1000 numbers?
- a list of 2000 numbers?
- a list of 100.000 numbers?

You're not allowed to try it out and time it.

Space complexity

How much memory does your algorithm require for a list of 1000 numbers? And for 2000 numbers?

Space complexity

How much memory does your algorithm require for a list of 1000 numbers? And for 2000 numbers?

Finding the *space complexity* works just the same as analyzing the runtime.

Definition

Big O notation describes the runtime of an algorithm as a function of the size of the input.

Definition

Big O notation describes the runtime of an algorithm as a function of the size of the input.

This allows you to find the rate of increase of the runtime.

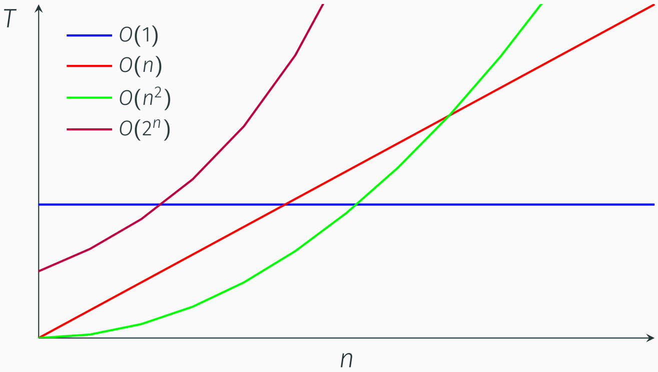
Definition

Big O notation describes the runtime of an algorithm as a function of the size of the input.

This allows you to find the rate of increase of the runtime.

For double the size of the input, will your algorithm take twice as long? Four times as long? More?

Example



Definition (continued)

For an algorithm that runs in time $O(n)$, its runtime is a constant times the size of the input.

$$c \cdot n$$

where c is a real number.

Definition (continued)

For an algorithm that runs in time $O(n)$, its runtime is a constant times the size of the input.

$$c \cdot n$$

where c is a real number.

For an algorithm that runs in time $O(n^2)$, its runtime is a constant times the square of the size of the input.

$$c \cdot n^2$$

A simple `for` loop

For loop: example

Finding the maximum number in a list of n random numbers.

- initialize $M = 0$
- compare each number in the list to the number max. If it is bigger, it becomes the new M

For loop: example

Finding the maximum number in a list of n random numbers.

- initialize $M = 0$
- compare each number in the list to the number max. If it is bigger, it becomes the new M

Runtime:

- initializing M has a small cost c_1
- iterating over the list takes n comparisons, each of cost c_2

For loop: example

Finding the maximum number in a list of n random numbers.

- initialize $M = 0$
- compare each number in the list to the number max. If it is bigger, it becomes the new M

Runtime:

- initializing M has a small cost c_1
- iterating over the list takes n comparisons, each of cost c_2
- Total cost $c_1 + n \cdot c_2$

Drop the non-dominant terms

Does this cost c_1 of initializing M really matter?

Drop the non-dominant terms

Does this cost c_1 of initializing M really matter?

No, because it does not grow as fast as the cost of iterating over the loop.

Drop the non-dominant terms

Does this cost c_1 of initializing M really matter?

No, because it does not grow as fast as the cost of iterating over the loop.

We only care about the fastest-growing term. Therefore we drop the smaller terms in our cost analysis.

Drop the constants

It does not matter if something has cost n, $0.1n$ or $100000n$. It is all $O(n)$ because what we care about is not the exact runtime, but the scaling behaviour.

Drop the constants

It does not matter if something has cost n, $0.1n$ or $100000n$. It is all $O(n)$ because what we care about is not the exact runtime, but the scaling behaviour.

Conclusion: our simple for loop example has runtime $O(n)$.

Drop the constants

It does not matter if something has cost n, $0.1n$ or $100000n$. It is all $O(n)$ because what we care about is not the exact runtime, but the scaling behaviour.

Conclusion: our simple for loop example has runtime $O(n)$.

Recap: What will happen when an algorithm with $O(n)$ runtime gets twice as much input?

Drop the constants

It does not matter if something has cost n, $0.1n$ or $100000n$. It is all $O(n)$ because what we care about is not the exact runtime, but the scaling behaviour.

Conclusion: our simple for loop example has runtime $O(n)$.

Recap: What will happen when an algorithm with $O(n)$ runtime gets twice as much input?

Answer: the runtime will double.

Double for loop

Double for loop

What happens, if we have an outer for loop for a list of length n, and an inner for loop over the same length, and inside the inner for loop we compare the two numbers in the list?

Double for loop

What happens, if we have an outer for loop for a list of length n, and an inner for loop over the same length, and inside the inner for loop we compare the two numbers in the list?

Answer: we make n^2 comparisons, so runtime $O(n^2)$.

Double for loop

What happens, if we have an outer for loop for a list of length n, and an inner for loop over the same length, and inside the inner for loop we compare the two numbers in the list?

Answer: we make n^2 comparisons, so runtime $O(n^2)$.

What happens if we double the lengths of both lists?

Double for loop

What happens, if we have an outer for loop for a list of length n, and an inner for loop over the same length, and inside the inner for loop we compare the two numbers in the list?

Answer: we make n^2 comparisons, so runtime $O(n^2)$.

What happens if we double the lengths of both lists?

We get $(2n)^2 = 4n^2$ comparisons: if the input is doubled, the runtime is multiplied by 4.

Repeatedly dividing by 3

Logarithm (1/3)

We start with a big number n and we keep dividing it by 3 until it is smaller than 3.

How many divisions do we make?

Logarithm (1/3)

We start with a big number n and we keep dividing it by 3 until it is smaller than 3.

How many divisions do we make?

If $3^{k-1} \leq n \leq 3^k$, we make k divisions, where $k = {}^3\log(n)$.

Logarithm (2/3)

Logarithm to which base?

Logarithm (2/3)

Logarithm to which base?

$$^{a}\log(x) = \frac{^{b}\log(x)}{^{b}\log(a)}$$

Does not matter, since $^{b}\log(a)$ is just a constant.

Conclusion: repeatedly dividing by 3 has runtime $O(\log n)$.

Logarithm (3/3)

What happens if we multiply the number n by 5, and run it through the same algorithm of dividing by 3?

Logarithm (3/3)

What happens if we multiply the number n by 5, and run it through the same algorithm of dividing by 3?

Answer: we get $\log(5n) = \log(5) + \log(n)$ divisions.

The knapsack problem

Knapsack: example

You have n items, each with their own weight $w_1, \ldots, w_n$. Since you're flying, the contents of your suitcase cannot be heavier than X. This means you have to leave some items at home. How do you fill up your suitcase as heavy as possible?

Knapsack: example

You have n items, each with their own weight $w_1, \ldots, w_n$. Since you're flying, the contents of your suitcase cannot be heavier than X. This means you have to leave some items at home. How do you fill up your suitcase as heavy as possible?

There are 2^n possibilities of how to fill up your suitcase. You have to consider each of them, so your runtime is $O(2^n)$.

Can we do better?

In the suitcase (knapsack) example, you get an exponential runtime. That’s not what you want at all!

Can we do better?

In the suitcase (knapsack) example, you get an exponential runtime. That's not what you want at all!

Because the knapsack problem is NP-complete, you can't do better if you want to be sure to get the best solution. But maybe you'll get an acceptable outcome with an *approximation algorithm*.

Advice

Practice

- Solve a ton of exercises.

Practice

- Solve a ton of exercises.
- Then solve some more.

Practice

- Solve a ton of exercises.
- Then solve some more.
- And more.

Practice

- Solve a ton of exercises.
- Then solve some more.
- And more.
- Math is great and so are you!

Practice

- Solve a ton of exercises.
- Then solve some more.
- And more.
- Math is great and so are you!
- Ask questions! Ask anyone, anytime, anywhere.