# Section 2: Big O Notation

## 4. Intro to Big O

slides

<https://cs.slides.com/colt_steele/big-o-notation>

## 5. Timing our Code

An Example

Suppose we want to write a function that calculates the sum of all numbers from 1 up to (and including) some number n.

This is the one that comes to my mind first is to basically create a total variable accumulator and loop through all those numbers and add them in one at a time starting at 1.

<https://codesandbox.io/s/misty-wildflower-0mcjz?file=/src/app.js>

add\_up\_faster.js

function addUpTo(n) {

return n\*(n+1) /2;

}

console.log(addUpTo(100))

var time1 = performance.now();

addUpTo(1000000000);

var time2 = performance.now();

console.log(`Time Elapsed: ${(time2-time1)/1000} seconds.`)

## 6. Counting Operations

function addUpTo(n) {

return n\*(n+1) /2;

}

1 multiplication

1 addition

1 division

function addUpTo(n) {

let total = 0;

for (let i = 1; i <= n; i++) {

total += i;

}

return total;

}

1 assignment

1 assignment

n comparisons

n additions & n assignments

n additions & n assignments

## 7. Visualizing Time Complexities

### Performance Tracker

Choose a function and start plotting!

<https://rithmschool.github.io/function-timer-demo/>

## 8. Official Intro to Big O

Big O Notation is a way to formalize fuzzy counting

It allows us to talk formally about how the runtime of an algorithm grows as the inputs grow

We won’t care about the details, only the board trends

We say that an algorithm is O(f(n)) if the number of simple operations the computer has to do is eventually less than a constant times f(n) , as n increases

* f(n) could be linear (f(n) = n)
* f(n) could be quadratic (f(n)=n)
* f(n) could be constant (f(n)=1)
* f(n) could be something entirely different

Example

function addUpTo(n) {

return n\*(n+1) /2;

}

Always 3 operations

O(1)

function addUpTo(n) {

let total = 0;

for (let i = 1; i <= n; i++) {

total += i;

}

return total;

}

Number of operations is (eventually) bounded by a multiple of n(say,10n)

O(n)

Another Example

function countUpAndDown (n) {

console.log("Going up!");

for (let i = 0; i < n; i ++) { O(n)

console.log(i);

}

console.log("At the top! \nGoing down...");

for (let j = n-1; j>=0; j--) { O(n)

console.log(j)

}

console.log("Back dowm. Bye!")

}

Number of operations is (eventually) bounded by a multiple of n(say, 10n)

O(n)

OMG MOAR EXAMPLEZ

function printAllPairs(n) {

for (var i = 0; i < n ; i++) {

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

console.log(i,j);

}

}

}

O(n) operation inside of an O(n) operation

O(n)

## 9. Simplifying Big O Expressions

When determining the time complexity of an algorithm, there are some helpful rule of thumbs for big O expressions.

These rules of thumb are consequences of the definition of big O notation.

### Constants Don’t Matter

O(2n) >>> O(n)

O(500) >>> O(1)

O(13n) >>> O (n)

### Big O Shorthands

## 10. Space Complexity

Time complexity : How fast algorithms run with runtimes, how can we analyze the runtime of an algorithm as the size of the inputs increases

Use big O notation to analyze **space complexity:**how much additional memory do we need to allocate in order to run the code in our algorithm?

Sometimes you’ll hear the term **Auxiliary space complexity** to refer to space required by the algorithm, not including space taken up by the inputs.

**Determine the space complexity for the following function**

* **function logUpTo(n) {**
* **for (var i = 1; i <= n; i++) {**
* **console.log(i);**
* **}**
* **}**

O(1)

**Determine the space complexity for the following function**

* **function logAtMost10(n) {**
* **for (var i = 1; i <= Math.min(n, 10); i++) {**
* **console.log(i);**
* **}**
* **}**

O(1)

**Determine the space complexity for the following function**

* **function onlyElementsAtEvenIndex(array) {**
* **var newArray = Array(Math.ceil(array.length / 2));**
* **for (var i = 0; i < array.length; i++) {**
* **if (i % 2 === 0) {**
* **newArray[i / 2] = array[i];**
* **}**
* **}**
* **return newArray;**
* **}**

**O(n)**

**Determine the space complexity for the following function**

* **function subtotals(array) {**
* **var subtotalArray = Array(array.length);**
* **for (var i = 0; i < array.length; i++) {**
* **var subtotal = 0;**
* **for (var j = 0; j <= i; j++) {**
* **subtotal += array[j];**
* **}**
* **subtotalArray[i] = subtotal;**
* **}**
* **return subtotalArray;**
* **}**

**O(n)**

## 11. Logs and Section Recap

### Logarithms

### Logarithmic time complexity is great

#### O(1) > O(log n ) > O(n) > O( n log n) > O(n)

Certain searching algorithms have logarithmic time complexity.

Efficient sorting algorithms involve logarithms.

Recursion sometimes involves logarithmic space complexity.

### Recap

##### To analyze the performance of an algorithm, we use Big O Notation

##### Big O Notation can give us a high level understanding of the time or space complexity of an algorithm

##### Big O Notation doesn’t care about precision, only about general trends(linear? Quadratic? Constant? )

##### The time or space complexity (as measured by Big O ) depends only on the algorithm, not the hardware used to run the algorithm

* Big O Notation is everywhere, so get lots of practice