Chapter 1

**Big O notation**

Note: Run all js code inside Chrome Browser called JS-Snippets.

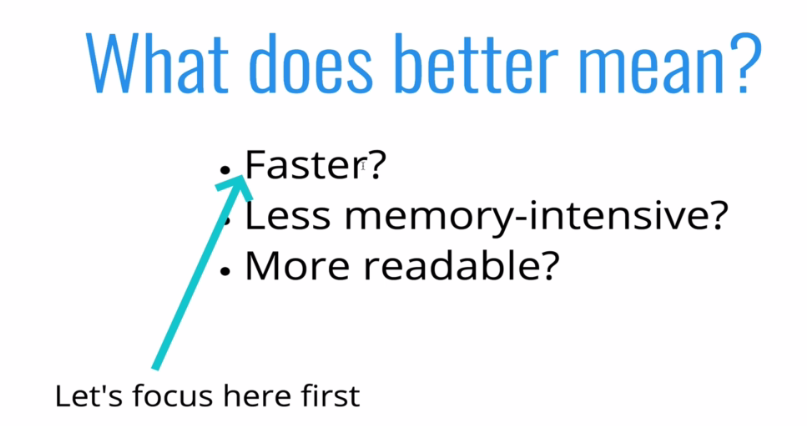
Chrome > Inspect > Source > Snippet

**1.1 Intro to big O**

* Objectives
* *need* for Big O Notation
* *Describe* Big O Notation
* *Simplify* Big O *Expressions*
* Define "*time complexity*" and "*space complexity*"
* *Evaluate* the *time complexity* and *space complexity* of different *algorithms* using *Big O* Notation
* Describe what a *logarithm* is
* Why big-O: There are multiple algorithms to solve a problem. There are different approaches and implementations to sove a specific problem.
* To find the best algorithm, we use big-O and calculate "*time complexity*" and "*space complexity*" and compare these for different algorithms.
* For Example: "Write a function that accepts a string input and returns a reversed copy". This problem has many solutions in Stack overflow. To find the best algorithm we have to use Big O.
* There are also following reasons that we need to consider Big O:
* Code-Performance: It's important to have a precise vocabulary to talk about how our code performs.
* Compare different approaches: Useful for discussing trade-offs between different approaches
* Find the critical points inside a code: When your code *slows* *down* or *crashes*, identifying *parts* of the code that are *inefficient* can help us find *pain points* in our applications
* Less important: it comes up in interviews!
* Example 1.1: Let's take a look at a more *concrete* *example* let's compare two solutions to the same problem. To write a function that calculates the sum of all numbers from ***1*** up to ***n***.

|  |  |
| --- | --- |
| // *-------- Using Loop  -----------*  **function** **addUpTo**(n) {  **let** total=0;  **for** (**let** i=1; i**<=** n; i++){          total += i;          }  **return** total;  }  console**.log**(**addUpTo**(6)) | // *Using mathematical formula*  **function** **addUpTo\_frmula**(n) {  **return** n\*(n+1)/2;  }  console**.log**(**addUpTo\_frmula**(6)) |

* Which one is better?



* To calculate *performance*, we use a *big number* and to calculate the time we use ***performance.now()*** function.

|  |  |
| --- | --- |
| // *-------- Using Loop  -----------*  **function** **addUpTo**(n) {  **let** total=0;  **for** (**let** i=1; i**<=** n; i++){          total += i;          }  **return** total;  }  // *console.log(addUpTo(6))*  **var** t1=performance**.now**();  **addUpTo**(1000000000);  **var** t2=performance**.now**();  console**.log**(`Time elapsed: ${(t2-t1)/1000} seconds`);  // *`` are not single quotes ''* | // *Using mathematical formula*  **function** **addUpTo\_frmula**(n) {  **return** n\*(n+1)/2;  }  // *console.log(addUpTo\_frmula(6))*  **var** t1=performance**.now**();  **addUpTo\_frmula**(1000000000);  **var** t2=performance**.now**();  console**.log**(`Time elapsed: ${(t2-t1)\*1000} Microseconds`); |
|  |  |

* The elapsed time varies. The difference is in Seconds and microsecond. Using formula gives a instant result. 10000 times faster than loop.

|  |
| --- |
| * Obviously second approach is good one. But measuring the performance of algorithm w.r.to ***'time'*** might be problematic. There are the following reasons:   ***The Problem with Time***   * *Different* *machines* will record *different* *times* * The *same* *machine* will record *different* *times*! * For *fast algorithms*, speed measurements may not be precise enough? |

Note:

**String Formatting in JavaScript:** In javascript, there is *no built-in* *string formatting function*. But there are other ways to format a string in *javascript* . 3 commonly used ways to format a string in javascript.

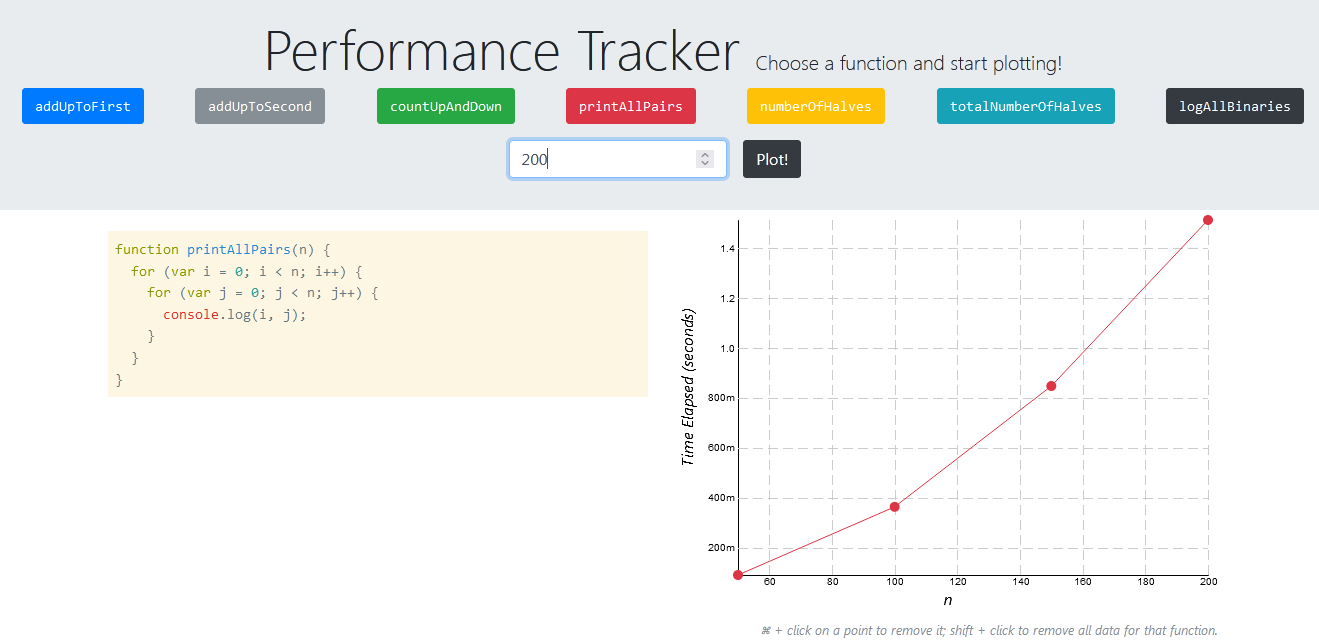
1. Using **{}** brackets within backticks **``**
2. Using **+** operator
3. By creating a format function

* Format String Using Backticks: Formatting string in *javascript* is best done by the use of *backticks* (``) and the *variables* are inserted *within* the *backticks* *wrapped* in *curly* braces ({}) preceded by a *dollar* sign ($).
* Example: `Hello ${name}`
* These variables are called *placeholders* within the *string* which are replaced with the *values* of the *variables*.

|  |  |
| --- | --- |
| **1.2 Counting operations**  Instead of *counting time*, we count *operations* inside an algorithm.   * Rather than counting seconds/time, which are so variable... * Let's count the number of simple operations the computer has to perform!   On the Right, we have 3 operations, whatever the value **n** is. |  |
| * Counting is getting hard! Depending on what we count, the number of operations can be as low as ***2n*** or as high as ***5n + 2***. * But regardless of the exact number, the ***number of operations*** grows roughly ***proportionally*** with ***n.*** |  |

* Performance tracker app: We have following app that plot the performance for 7-different functions. We can use the app for our demonstration. It is basically done by the timer at the background.

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



|  |  |
| --- | --- |
| // *-------- Using Loop  -----------*  **function** **addUpTo**(n) {  **let** total=0;  **for** (**let** i=1; i**<=** n; i++){          total += i;          }  **return** total;  }  console**.log**(**addUpTo**(6)) | // *Using mathematical formula*  **function** **addUpTo\_frmula**(n) {  **return** n\*(n+1)/2;  }  console**.log**(**addUpTo\_frmula**(6)) |
| Number of operations grows as n grows. | Number of operations is constant |

**1.3 The 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 *trends*
* We say that an algorithm is if the number of simple operations the computer has to do is eventually less than a constant times , as n increases
* could be *linear* ( )
* could be *quadratic* ( )
* could be *constant* ( )
* could be *something* entirely different! It could be logarithmic or exponential.
* When we talk about Big-O, we talking about worst case scenarios. Upper bound of runtime.

|  |  |
| --- | --- |
| **function** **countUpAndDown**(n) {      console**.log**("Going up!");  **for** (**var** i=0; i **<** n; i++) {        console**.log**(i);      }      console**.log**("At the top!\nGoing down...");  **for** (**var** j=n-1; j **>=** 0; j--) {        console**.log**(j);      }      console**.log**("Back down. Bye!");    } |  |

* We can think it as **O(2n)**, but in simpler term it is linear, so **O(n)**.

|  |  |
| --- | --- |
| * Nested loop (quadratic):   **function** **printAllPairs**(n) {  **for** (**var** i=0; i **<** n; i++) {  **for** (**var** j=0; j **<** n; j++) {      console**.log**(i, j);      }  }  } |  |

**1.4 Simplifying Big-O Expressions**

Rule of thumb: When determining the *time complexity* of an algorithm, there are some *helpful rule of thumb* for big O expressions. These *rules of thumb* are consequences of the *definition* of *big O notation*.

* Analyzing complexity with big O can get complicated. These rules won't ALWAYS work, but are a helpful starting point.

|  |  |
| --- | --- |
| 1. Constant doesn't matter: | **O(2n) ~ O(n);**  **O(500) ~ O(1);**  **O(15n2) ~ O(n2)** |
| 1. Smaller terms Doesn't matter : | **O(2n + 100) ~ O(n);**  **O(n + 500) ~ O(n);**  **O(15n2 + 50n + 400) ~ O(n2)** |

1. *Arithmetic operations* are *constant*
2. *Variable assignment* is *constant*
3. *Accessing elements* in an *array* (by index) or *object* (by key) is *constant*
4. In a *loop*, the *complexity* is the length of the *loop times* the complexity of whatever happens inside of the loop

* Notice the following exceptions:

**function** **logAtleast5**(n) {

**for** (**var** i=1; i **<=** Math**.max**(5, n); i++) { console**.log**(i); }

    }

* For ***n = 1,2, . . 5*** the operation is constant it loops ***5 times***. But when ***n>5*** the operations grows ***proportional*** to ***n***. So here we have ***O(n)***. Math**.max**(5, n) taking the maximum value.
* On the other hand, following loops ***n*** time as ***n<5***, but as ***n>5*** the loop stick to ***5***, growing ***n>5*** doesn't effect the operations. So the no. of operation is constant here. Hence ***O(1)***. Math**.min**(5, n) taking the minimum value.

**function** **logAtleast5**(n) {

**for** (**var** i=1; i **<=** Math**.min**(5, n); i++) {

      console**.log**(i);

    }

    }

**Some common Trends**

|  |  |
| --- | --- |
| * According to these trends we will compare our different algorithms in upcoming chapters. * We can see and are pretty good actually. * **O(n)** is better comparing to **O(n2)** and . |  |

**1.5 Space Complexity**

So far, we've been focusing on *time complexity*: how can we analyze the *runtime* of an *algorithm* as the *size* of the inputs *increases*?

* *Space complexity* is about the *memory* that taken by the *algorithm*. We can also 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*?
* What about the inputs: We know inputs has its own allocated-memory, but we do not consider it as Space-complexity. We consider the auxiliary space complexity.
* Auxiliary Space Complexity: 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*.
* When we talk about *space complexity*, technically we'll be talking about *auxiliary space complexity*.
* Rules of thumb for space complexity.

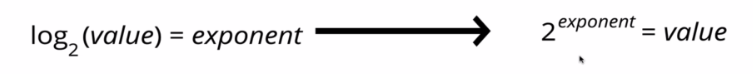
1. Most primitives (booleans, numbers, undefined, null) are constant space. Eg: ***100.6***, ***109999*** are considered constant space.
2. Strings require ***O(n)*** space (where ***n*** is the ***string length***)
3. Reference types are generally ***0( n)***, where ***n*** is the ***length*** (for arrays) or the ***number of keys*** (for objects)

|  |  |
| --- | --- |
| * Here space complexity is ***O(1)***, array is not changing, we just using its elements. The variables are "***total***" and "***i***" those two takes *constant space*, no new variables are created, we are just updating "***total***" and "***i***" for each step of the loop. |  |

|  |  |
| --- | --- |
| * On the other hand consider this example:   ***newArr***'s ***length*** depend on the input array ***arr***, if input array ***arr*** is larger ***newArr*** will be large. i.e space increases as input increases. Hence we have ***O(n)*** for the space complexity. |  |

**1.6 Logarithms**

* **Recall : *Logarithm*** is the inverse of ***exponent***.EG: **log10(100)=2,** or **,** it helps us to work with really big numbers.



* Note: In general ***log's base is 10***, now in our case as we study Data-Structure and algorithm ***our log's base is 2***.
* This isn't a math course, so here's a rule of thumb. The logarithm of a number roughly measures the number of times you can divide that number by 2 before you get a value that's less than or equal to one.
* Why we need logarithms
* 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!