

Big O Notation

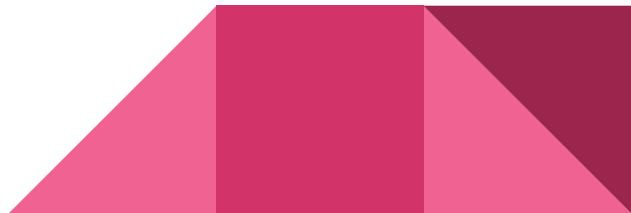
Why Does Algorithm Efficiency Matter?

Imagine searching for a name in a phone book...

- **Method A:** Start at page 1, check every name until you find it
- **Method B:** Open to the middle, decide if your name comes before or after, repeat

Which is faster for 1,000,000 names?

The algorithm you choose matters enormously as data grows.



Do Now

How can we compare the performance of different algorithms ?

Should we record the start time, run the algorithm, record the end time, and then calculate the running time?



COMPLEXITY

There are two types of complexity:

- **Time complexity:** number of steps involved to run an algorithm.
- **Memory complexity:** amount of memory it takes to run an algorithm.

We should be concern about **time complexity** when developing our algorithms.



How do we measure the time complexity of an algorithm?

Looking at:

1. The worst case?
2. The best case?
3. The average case?



How do we measure the time complexity of an algorithm?

Looking at the best case does not help. You rarely going to have the best case.

Looking at the average case is not going to tell you the absolute worst time complexity.

Looking at the worst case can help us to understand what to expect from the algorithm. That is why is more helpful to look at the worst case.



Example: Algorithm Add Sugar to Tea

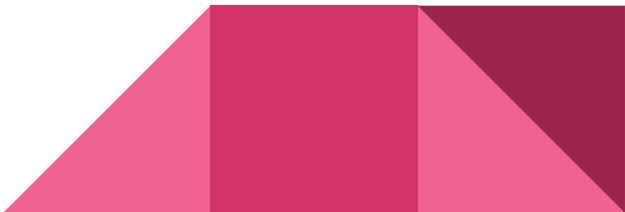
1. Fetch the bowl containing the sugar
2. Get a spoon
3. Scoop out sugar using the spoon
4. Pour the sugar from the spoon into the tea
5. Repeat steps 3 and 4 until you have added the desired amount of sugar



Example: Algorithm Add Sugar to Tea

Number of sugars	Steps required
1	4
2	6
3	8
4	10

Time complexity depends on the number of sugars someone wants on their tea.
How do we measure time complexity?



The Big O Notation

Big O notation describes how an algorithm's runtime grows as the input size increases. In other words, it indicates the complexity related to the **number of items** that an algorithm has to deal with.

It answers: "If I double my data, how much longer will this take?"

It is written as a capital O followed by an expression in parenthesis. Example: $O(n)$

It is conventional to designate the number of items by n .



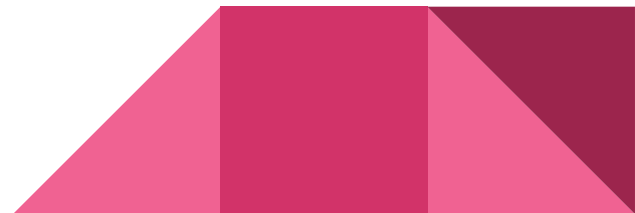
Big O notation for the number of sugars algorithm

- Number of desired sugars = n
- Total number of steps = $2 + 2n$ (as n grows the number of steps grows)
- The 2 in $2n$ and the $+2$ remain constant (do not factor into the time complexity)
- Time complexity = $O(n)$
- Linear time complexity

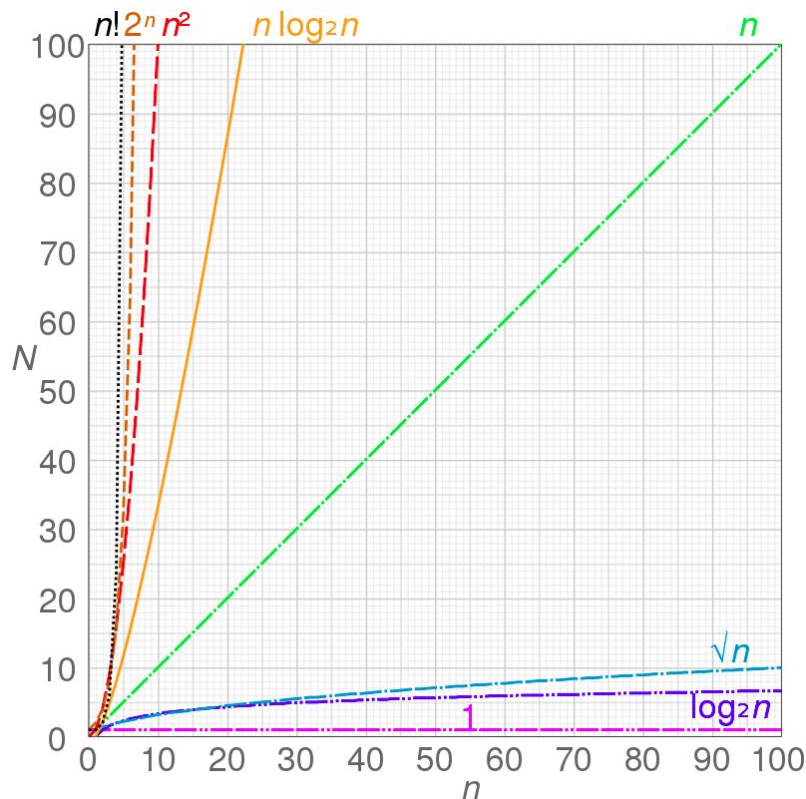


Big O values

Big O values		
$O(1)$	Constant	Excellent
$O(\log n)$	Logarithmic (base2)	Good
$O(n)$	Linear	Fair
$O(n \log n)$	$n \log n$	Bad
$O(n^2)$	Quadratic	Horrible



Big O Notation



Graphs of functions commonly used in the analysis of algorithms, showing the number of operations N versus input size n for each function.

Source: Wikipedia

Constant Time: $O(1)$

An algorithm does not depend on the input size n . The run time will always be the same regardless of the input size.

Example: Return the first element of an array. Even if the array has 1 million elements, the time complexity will be constant.



Linear Time: $O(n)$

When the running time of an algorithm increases linearly with the size of the input (a function iterates over the input size of n).

Example: An algorithm that calculates the factorial of any inputted number. If you input 5 then you are to loop through and multiply 1 by 2 by 3 by 4 and by 5 and then output 120.




Logarithm Time $O(\log n)$

Similar to linear time complexity, except that the runtime does not depend on the input size but rather on half the input size. When the input size decreases on each iteration or step.

This method is the second best because an algorithm runs for half the input size rather than the full size.

The input size decreases with each iteration.

Example: Binary search functions, which divide a sorted array based on the target value.



Quadratic Time $O(n^2)$

When an algorithm has nested iteration (nested loops).

Example: An outer loop runs n times, and the inner loop will run n times for each iteration of the outer loop, which will give total n^2 prints. If the array has ten items, ten will print 100 times (10^2).



Rules for Determining Big O

Rule 1: Drop the constants

- $O(2n) \Rightarrow O(n)$
- $O(500) \Rightarrow O(1)$

Rule 2: Drop lower-order terms

- $O(n^2 + n) \Rightarrow O(n^2)$
- $O(n + \log n) \Rightarrow O(n)$

Rule 3: Different inputs = different variables

```
for (int a : arrA) {    // O(a)
    // do something
}
for (int b : arrB) {    // O(b)
    // do something
}
// Total: O(a + b), NOT O(n)
```



Analyzing Nested Loops

Independent loops - ADD

```
for (int i = 0; i < n; i++) { } // O(n)
for (int j = 0; j < n; j++) { } // O(n)
```



Total: $O(n + n) = O(n)$

Nested loops - MULTIPLY

```
for (int i = 0; i < n; i++) {      // n times
    for (int j = 0; j < n; j++) {  // n times each
        // do something
    }
}
```



Total: $O(n \times n) = O(n^2)$



Key Takeaways

1. Big O describes how **runtime scales with input size**
2. We **focus on worst case** and **dominant terms**
3. Recognize the patterns:
 - **Single loop** through data => **$O(n)$**
 - **Nested loops** => **$O(n^2)$**
 - **Halving each step** => **$O(\log n)$**



Classwork Time :)

https://github.com/novillo-cs/apcsa_material/tree/main/classwork/01_12_time_complexity

