**Module #1 Introduction, Arrays, and Complexity Analysis**

CS 315: Data Structures and Algorithms

Richard S. Stansbury

Embry-Riddle Aeronautical University

Daytona Beach, FL

# Asymptotic Complexity

|  |
| --- |
| Important Note:  In the definitions of big-O covered below and within the textbook, t(N) represents number of steps or true complexity of the algorithm given the problem size N. It uses f(N) as the order of complexity with respect to N that meets the definition of big-O.  In other readings, you might find t(N) represented as f(N) and f(N) represented as g(N). These are semantically the same meaning, but a different form of notation between different authors.  So, if reading the supplementary material, make sure you take note of the type of notation used by the author by first reviewing their formal definition of big-O. |

## Big-O Notation

### Concept

**Introduced by Paul Bachmann in 1892**

Notation: O(f(n)) where f(N) is some function defined in terms of problem size N.

e.g. reads as big-O of N squared.

Qualitative metric. Not intended to be quantitative measure of performance.

Saying an algorithm is implies that its performance can be described by an **asymptotic curve** that represents its upper bound as N becomes sufficiently large.

**Note to self:** Draw on board a simple example of an asymptote.

Relies upon simplifications.

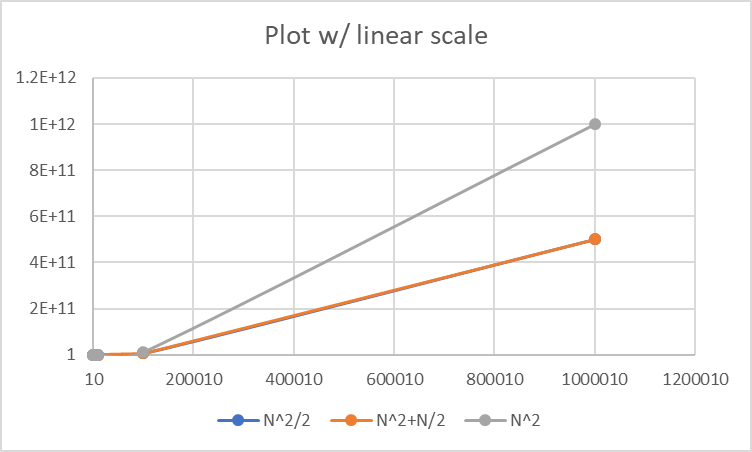
Standard Simplifications of Big-O

1. Eliminate any term whose contribution to the total ceases to become significant as N becomes large.
   1. i.e. drop all lower order terms
2. Eliminate any constant factors.

To understand this, think about our selection sort algorithm’s performance estimate as approximately . Can we determine a better asymptote to represent our big-O upper limit?

Observe what happens as we pick apart this expression.

|  |  |  |  |
| --- | --- | --- | --- |
| N |  |  |  |
| 10 | 50 | 5 | 55 |
| 100 | 5000 | 50 | 5,050 |
| 1,000 | 500,000 | 500 | 500,500 |
| 10,000 | 50,000,000 | 5,000 | 50,005,000 |
| 100,000 | 5,000,000,000 | 50,000 | 5,000,050,000 |
| 1,000,000 | 500,000,000,000 | 500,000 | 500,000,500,000 |



* In actuality, the lower order term N/2 does not play much of a role.
  + We can drop it and our understanding of the performance as N grows does not change.
  + We would now have .
* Which given our second simplification rule, we could change to simplify to as the approximation and
  + Same curve shape, but it would just be double each value.
* So, for selection sort in describing how it scales over problem size, N, we can say
  + Constants and lower order terms didn’t influence the shape significantly
  + We could approximate the shape using N^2 as an upper bound as N grows.
  + So, for this problem, we would say that it is **O().**

### Formal Definition:

iff for all

Where

* + t(N) is the time complexity of the algorithm for data set of size N.
  + is a constant
* What this is saying that if an algorithm’s true complexity is less than or equal to the approximation of complexity multiplied by some coefficient c for all N greater than some minimum
* **Our goal** is to find the that is smallest that exists where we can find a valid and such that the definition’s constraint holds.

Please note that asymptotic complexity approximations are not just for handling time complexity. Space complexity can also be described in terms of big-O.

Example #1:

public static boolean hasValue(int [] arr, int val) {

for (int i=0; i < arr.length; i++) {

if (arr[i] == val) return true;

}

return false;

}

What is the Big-O?

* + In the worst-case, the loop would iterate over all N items where N = arr.length
  + Show counting steps on board.
  + O(N)

Since this is a relatively simple problem, we can explain much of this without complex analysis:

First, let’s determine how many operations happen inside the loop. We see that inside the loop, there is a comparison and a return. This would take a constant amount of time to complete so let’s simply call it a O(1) operation.

Next, we must observe how many times the interior of the loop is called. We see that it iterates from i=0 to i=n-1. . Therefore, the iterations of the loop are O(N).

Finally, we take note of the remaining operations, which is the initialization of i, the final comparison of the loop, and the final return. These are all constant time operations.

So, we can see that we have basically: O(N) + O(1) = O(N).

Alternatively, we can count the operations:

* 1 op: int i = 0
* Loop, executes n=arr.length times.
  + 1 op: i < arr.length
  + 2 op: if (arr[i] == val) return true;
    - Comparison and return
  + 1 op: i++;
* 1 op: i < arr.length

This gives us: 1 + n\*4+1 = 4n+2 = O(N)

Example #2:

public static void printArr(int [] arr) {

for (int i=0; i < arr.length; i++) {

System.out.println(arr[i]);

}

}

What is the Big-O?

* + No matter what, the algorithm will always iterate over the array’s entire length.
  + Show counting steps on board.
  + O(N)

Simple analysis:

* We have a single loop that is going from i=0 to N where n = arr.length
* Print statement inside the loop is a constant time operation
* O(N)

More complex:

* 1 op: int i=0
* N loops:
  + 1 op: i < arr.length
  + 1 op: println
  + 1 op: i++
* 1 op: I < arr.length

1 + N\*3 + 1 = O(3N+2) = O(N)

**Now switch to worksheets for more complex problems.**

## Common Big-O Asymptote classes

– Constant e.g. fixed set of operations, accessing location in array, etc.

– Logarithmic growthy (typically log base 2). e.g. binary search of sorted array and binary search tree lookup.

– Linear. e.g. traversal of an array or list, finding an item in an unsorted list.

– linearithmetic or loglinear – Heap sort, best case quick sort, merge sort, fast Fourier transforms.

– Quadratic (or polynomial complexity with polynomial 2 growth) – e.g. bubble sort, selection sort, insertion sort, worst case for quick sort.

– Polynomial – often scene when you have two or more nested loops where the loops iterate incrementally (index+1) or iterates through elements of a collection.

– Exponential– e.g. traveling sales person using dynamic programming / constraint satisfaction problem formulation

– factorial– e.g. brute force search of traveling salesperson, or enumerating all possible partitions within a set.

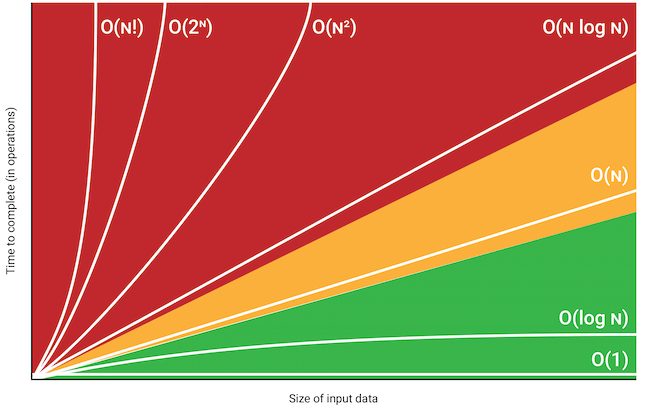


Figure : https://danielmiessler.com/study/big-o-notation/

## Other Asymptotic complexities

Two additional asymptotes may be considered with complexity analysis, which are defined by **Big-Omega (asymptotic lower bound)** and **Big-Theta (asymptotic tight bound).**

### Big-Omega Notation

Big-Omega is the lower-bound asymptote formally defined as:

iff for all

Where

* + t(N) is the time complexity of the algorithm for data set of size N.
  + is a constant

Example #1:

public static boolean hasValue(int [] arr, int val) {

for (int i=0; i < arr.length; i++) {

if (arr[i] == val) return true;

}

return false;

}

What is the Big-?

* + In the best-case scenario, the first item is the value we are looking for and we return true in the first iteration of the for loop.
  + Since this would be a single step, we know that it would be a constant time operation.
  + Therefore, we can say that the algorithm will do no better than a constant time operation.

Example #2:

public static void printArr(int [] arr) {

for (int i=0; i < arr.length; i++) {

System.out.println(arr[i]);

}

}

What is the Big-?

* + No matter what, the algorithm will always iterate over the array’s entire length.
  + Show counting steps on board.
  + (N)

### Big-Theta Notation

Big-O is the asymptotic upper bound and big-Omega is the asymptotic lower bound. It is not uncommon for these two values to be different. If is the same for both big-O and big-Omega, then we say that the asymptote represents the “tight bounds” of the algorithm, which is known as big-theta.

iff for all

Where

* + is the time complexity of the algorithm for data set of size N.
  + and are constants

Example #1:

public static boolean hasValue(int [] arr, int val) {

for (int i=0; i < arr.length; i++) {

if (arr[i] == val) return true;

}

return false;

}

What is the Big-?

* + We cannot find a big theta for this problem. It cannot be tightly bound.
  + We already showed that the tightest big-O is O(N) and the tightest big-Omega is
  + Since these two asymptotes are not equal, we do not have a big-theta.

Example #2:

public static void printArr(int [] arr) {

for (int i=0; i < arr.length; i++) {

System.out.println(arr[i]);

}

}

What is the Big-?

* + We have shown that for this algorithm we have the same runtime for the best and worst case. and
  + Since both have the same asymptote, we can say that this algorithm is tightly bound at