Definition of Asymptotic Time Complexity

The National Institute of Standards and Technology define it as “The limiting behavior of the execution time of an algorithm when the size of the problem goes to infinity. This is usually noted in Big-O notation.”

Another way of putting it is: A way to describe how efficient an algorithm is relative to the size of the data set.

Purpose of it

What is the purpose of it?

This is key for us to have a quick way to determine how efficient an algorithm is versus another competing algorithm. Otherwise, arguments can ensue over which way is “better” and productivity can drop. This is more of a real-world consideration than theoretical. It helps us have a yardstick to measure algorithm efficiency quickly and consistently. This is just one way to measure how “good” an algorithm is depending on what we need to accomplish.

It is important to know that absolute efficiency is not always our goal. Asymptotic Time Complexity is a handy way to settle disputes, however, when performance is paramount.

The role of "Big-O" notation

When the term “n” is used in Big-O notation, it refers to the size of the data set. For example, assume we have an array data structure and this array has 20 elements in it. We would say that n is 20 since that is the maximum size of the data set.

Most of the time, we measure the efficiency of the algorithm relative to “n”. There are some algorithms, such as constant time algorithms, where the size of n is irrelevant, but this is not common for non-trivial problems.

Why do we use “n” instead of just counting the number of (machine) instructions?

For more complex problems (or problems that need to be scalable), counting the number of machine instructions is impractical or impossible. Think of operations where we have to access a database and process data of unknown (at compile time) size. Without knowing the size of the data set, you may not be able to accurately count the number of machine instructions generically.

By using a system relative to “n”, we are able to consider how the algorithm processing time grows with “n”. The process of growing with “n” might be unavoidable but we want it to grow **as slow as possible** with the size of the data set (n).

Examples of Big-O notation and common time classifications

Examples of asymptotic time with common algorithms

- Two-sum problem using two nested loops (n^2)
- Printing every element in an array (n)
- Reading a single element from an array (1)
- Basic sort like Selection or Insertion Sort (n^2)
- Reversing order of an array (n/2)* (With swap variable)

Different levels of common asymptotic time (in order of most efficient to least)

(Sometimes called “Order of Growth Classifications”)

- Constant Time - O(1)
- Logarithmic - O(log n) *(log is ALWAYS log 2)
- Linear - O(n)
- Linearithmic - O(n log n)
- Quadratic - O(n^2, etc.)
- Exponential - O(2^n)
- Factorial - O(*n!*)

Best case vs Worst case

Best case vs. Worst case (focus more on worst case as a guarantee)

Even with Asymptotic analysis, there may be certain algorithms that have a different “best case” and “worst case”. We generally want to worry more about the worst case in Computer Science since that gives us a guarantee of the “bare minimum” performance we can expect. Some exceptions due exist such as the quicksort sorting algorithm which has a worst case of n^2 but performs far better in the average case.

When you absolutely MUST have a baseline of performance, you will want to look at the worst case instead of the best case (or even the average case.)

Do we ignore constants?

*Do we ignore constants? (4n vs n*log n) Why do we?*

Sometimes you will see an algorithm in a textbook with a time complexity of “2n + n log n + 12” for example. The rules for our quick classifications are that:

- We generally ignore constants and treat them as 1
- We label the classification based on the least efficient part of the classification

So, for the above example, we would ignore non-1 constants and treat them as 1 so this would become “1n + n log n + 1”. We would then label the whole algorithm based on the least efficient which would be “n log n”. This is how we quickly classify the algorithm. Therefore, this hypothetical algorithm would be considered linearithmic.

Examples to test class knowledge of classification:

- A1 - 3n + log n + 3 (n) Linear
- A2 - n^2 + n log n + 12n (n^2) Quadratic
- A3 - n^3 + 100n + 2^n (2^n) Exponential
- A4 - log n + 13 (log n) Logarithmic
- A5 - 3 log n + 2n + n log n (n log n) Linearithmic

Examples of Common problems and optimal solutions

Some computer science problems already have the best (or optimal) possible solutions (in the worst case)

Examples:

- General Purpose Sort (n log n)
- Linear Search on unsorted array (n)
- Binary Search on sorted array (log n)
- Max/Min value on sorted array (constant or 1)
- Max/Min value on unsorted array (n)
- Copying elements of an array to another array (n)

We still need hardware and software optimization

Asymptotic time complexity doesn’t remove need to make hardware/software optimizations

(But only means for a given set of instructions, this is scalable for a data set)

Examples:

```
int x = 2 * 16; // Unoptimized

int x = 2 << 4; // Optimized (binary shifting is faster)

// Another example...

int number = 33;

int y = number % 2; // Unoptimized

int y = number & 1; // Optimized (bitwise operations are faster)
```

Therefore, we can still benefit from small optimizations even when using optimal algorithms from a Big-O standpoint.

NOTE: It is important to note that many modern compilers are making optimizations for you behind the scenes so you might not notice performance increases regardless of which way you code with above. This doesn't mean we shouldn't follow best practices as we might one day work with a compiler that doesn't make those optimizations for us and our performance can take a hit.

Algorithm Analysis wrap up

Algorithm analysis is largely based on Asymptotic time complexity (and space complexity). For theoretical Computer Science, this is a necessary idea to master (as proofs need to be written)

If you are at all interested in the theoretical side of Computer Science, you will need to become exceptionally familiar with this concept. When studying algorithms (not in this class thankfully!), you will need more than a quick classification of said algorithm. You will have to complete a mathematical proof detailing the precise time complexity of a given algorithm. This takes time and is quite involved and complex!

Thankfully, for this class, we are more concerned with quick classifications for purposes of real-world application or “rule-of-thumb” style judgment calls. Just know that there is a whole sub-field that goes in depth about this one concept that we spend a week on so this is far from everything there is to say about the subject!