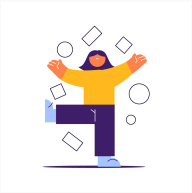
Lec 14: Time Complexity

Sam McCauley

March 13, 2026

Admin



- Midterm back; let me know if you have questions or something seems off
- Colloquium today on robot planning, also ethics (Mt Holyoke professor)
- Taco Tuesday on Tuesday!

The Challenge of Analyzing Time

- Different computers run at different speeds
- Computers are complicated! Adding two numbers together (for example) can take drastically different times depending on context.
- Good news: often times these details don't change much
- Example: It doesn't matter (too much) how fast I read if I'm scanning thousands of extra dictionary pages.
- This is why we focused on traversals until now.

Counting up time

- When we look at some Java code, how can we estimate how fast it is?
- Let's look at the operations the code requires
  - By operations, I mean built-in operations like `+`, `-`, `==`, `if`, `=`, array operations, etc.
  - If any methods are called, should count up their operations as well
- If we sum the time of all operations, we can figure out how long the code takes.
- Let's do a quick example

Counting up time example

```
1 int i = 0;       c1 time (integer variable assignment)
2 int count = 0; c2 time (integer variable assignment)
3 while(i < arr.length) { c3 time (access length, compare)
4     count += arr[i];  c4 time (array access, add, and assign)
5     i++;  c5 time (variable assignment and addition)
6 }
```

In total, this code takes time *at most*:

$$c1 + c2 + (c3 + c4 + c5) \cdot \texttt{arr.length}$$

Counting up time comparison

```
1  int count = arr[0] % 27; c6 time
```

In total, this code takes time at most $c6$.

If our array is at all large, this is going to be faster than the loop on *any* computer.

Let’s formalize this

Goal in analyzing efficiency:

- We don’t care that much about constants
- We care about scaling: what happens when the data in question is fairly large?
- Big-*O* notation: way of comparing two running times with this in mind

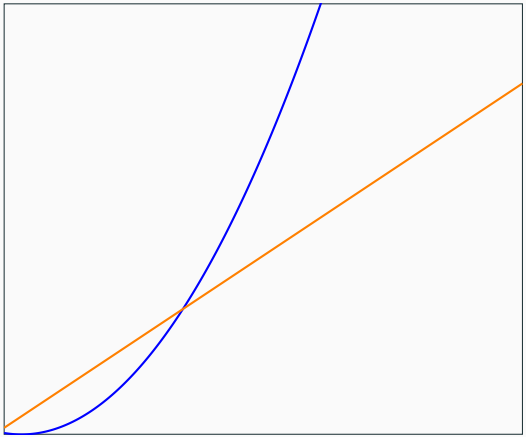
Big-O Notation

Definition

$f(n)$ is $O(g(n))$ if there exist constants $c > 1$ and n_0 such that for all $n > n_0$, $f(n) \leq c \cdot g(n)$

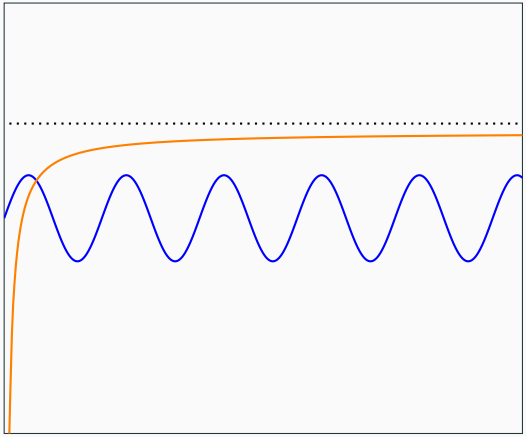
That is to say: If n is large enough ($n > n_0$), and if we ignore constants (we compare to $c \cdot g(n)$), then $g(n)$ is larger.

Plotting Big-O



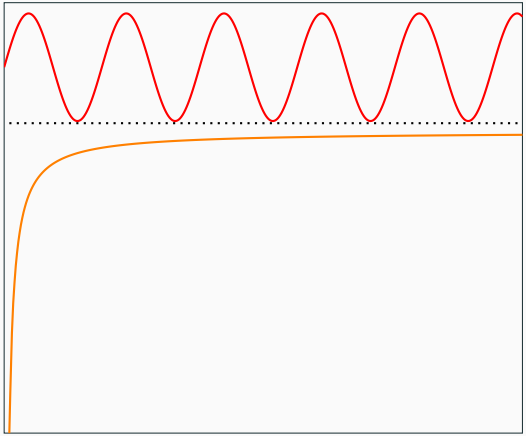
Let $g(n)$ be the blue function, and $f(n)$ be the orange function. If $g(n)$ and $f(n)$ continue increasing in the same way, then $f(n) = O(g(n))$.

Plotting Big-O



Let $g(n)$ be the blue function, and $f(n)$ be the orange function; assume that $f(n)$ is bounded above by the dotted line. Since $f(n) < c \cdot g(n)$, we still have $f(n) = O(g(n))$.

Plotting Big-O



Continued from last slide: once we multiply g by a constant, we obtain the plot shown in red; this is larger than f.

Proving Big-O

Reminder: $f(n)$ is $O(g(n))$ if there exist constants $c > 1$ and n_0 such that for all $n > n_0$, $f(n) \leq c \cdot g(n)$

- Let's say we have two functions $f(n)$ and $g(n)$, and we want to show that $f(n)$ is $O(g(n))$.
- We need to come up with a $c > 1$ and an n_0 such that for all $n > n_0$, $f(n) \leq c \cdot g(n)$

Counting up time example

```
1 int i = 0;     c1 time
2 int count = 0; c2 time
3 while(i < arr.length) { c3 time
4     count += arr[i];  c4 time
5     i++;  c5 time
6 }
```

Let's define $n =$ arr.length. That is: we are analyzing the running time in terms of the array length

In total, this code takes time at most:

$$f(n) = c_1 + c_2 + (c_3 + c_4 + c_5)n$$

Let's prove that this is $O(n)$.

Counting up time example

$$f(n) = c_1 + c_2 + (c_3 + c_4 + c_5)n$$

Let's prove that $f(n) = O(n)$.

Let's set $n_0 = 1$ and $c = c_1 + c_2 + c_3 + c_4 + c_5$. Then we want to show that for all $n > 1$, $f(n) \leq c \cdot g(n)$:

$$\begin{aligned} f(n) &= c_1 + c_2 + (c_3 + c_4 + c_5)n \\ &\leq c_1 n + c_2 n + (c_3 + c_4 + c_5)n \\ &= (c_1 + c_2 + c_3 + c_4 + c_5)n \\ &= c \cdot n \\ &= c \cdot g(n) \end{aligned}$$

So for $n > n_0$, $f(n) \leq c \cdot g(n)$; therefore, $f(n) = O(n)$

Counting up time example

```
1  int i = 0;     c1 time
2  int count = 0; c2 time
3  while(i < arr.length) { c3 time
4      count += arr[i];  c4 time
5      i++;  c5 time
6  }
```

This code takes $O(n)$ time.

Simplifying

- None of the c_i really mattered in the above analysis
- What we want to do: count the *number of operations* in a code segment
- Don't need to be too careful about it: `count += arr[i]` counting as 1 operation or 3 operations isn't going to change our final result
- Let's consider the above code again

Counting up time example

```
1  int i = 0;     2 operations
2  int count = 0;
3  while(i < arr.length) { 1 operation
4      count += arr[i];  4 operations
5      i++;  1 operation
6  }
```

This code takes $2 + 6n$ operations. Since each operation takes constant time, this is $O(n)$ time.

Counting up time comparison

```
1  int count = arr[0] % 27;  1 operation
```

If n is the length of the array, how long does this code take?

It takes a constant number of operations. $O(1)$ time.

(Note: $O(1)$ means *bounded above by a constant*—this function does not get larger as n increases. How does this relate to the definition of big-O?)

Wrapping Up Asymptotics

- We want to count the amount of time taken by a method
- Our analysis should apply regardless of how fast the computer is
- Idea: Look at how many operations are used. Use big-*O* notation
  - Ignore constants
  - Only care about sufficiently large inputs
- To be clear: you *never* need to count time of individual operations in this class. This was motivation. From now on, just count the number of operations; OK to ignore constants

Asymptotics Examples

Classic Examples

- What is the big-*O* running time for a method that traverses a list once?
  - $O(n)$
- What is the big-*O* running time for a method that traverses a list *twice*?
  - $O(n)$
- What is the big-*O* running time for a method that traverses a list n times?
  - $O(n^2)$
- What is the big-*O* running time for a Linked List method that only adjusts the references of the head and tail nodes?
  - $O(1)$

indexOf

```
1  public int indexOf(E element) {
2      for (int index = 0; index < numElems; index++) {
3          if (element.equals(arr[index])) {
4              return index;
5          }
6      }
7      return -1;
8  }
```

How long does this code take on a list of length n? $O(n)$

hasTwoSum

```java
public static boolean hasTwoSum(int[] arr, int target) {

    for (int i = 0; i < arr.length; i++) {
        for (int j = i + 1; j < arr.length; j++) {

            if (arr[i] + arr[j] == target) {
                return true; // found a pair
            }

        }
    }

    return false; // no pair found
}
```

How long does this code take on an array of length n? $O(n^2)$

isPalindrome Linked List

```
1  public boolean isPalindrome() {
2      Node<E> forward = head;
3      Node<E> backward = tail;
4      for(int i = 0; i < size(); i++) {
5          if(!forward.getData().equals(backward.getData())) {
6              return false;
7          }
8          forward = forward.getNext();
9          backward = backward.getPrevious();
10     }
11     return true;
12 }
```

How long does this code take on a linked list of length n? $O(n)$

isPalindrome Linked List With Get

```
1  public boolean isPalindrome() {
2      for(int i = 0; i < size(); i++) {
3          if(!get(i).equals(get(size() - i - 1)) {
4              return false;
5          }
6      }
7      return true;
8  }
```

How long does this code take on a linked list of length n? Each `get()` is $O(n)$ time and there are n of them. So: $O(n^2)$

Binary Search

```
1  private static int binarySearch(ArrayList<Integer> list, int
       target, int left, int right) {
2          if (left > right) {
3              return -1;
4          }
5          int mid = (left + right) / 2;
6          int midValue = list.get(mid);
7          if (midValue == target) {
8              return mid; // exact match
9          }
10         else if (target < midValue) {
11             return binarySearch(list, target, left, mid - 1);
12         }
13         else {
14             return binarySearch(list, target, mid+1, right);
15         }
16 }
```

How long does this code take on a list of length *n*?

Analyzing Binary Search

- We start out with an array of length n
- Each time the method is called, we do $O(1)$ work and call a method half the size
- So we call the method once for n, $n/2$, $n/4$, $n/8$, . . ., 1
- How many method calls is that?
- Want $n/2^k = 1$, so $2^k = n$. This means that $k = \log_2 n$
- $O(\log_2 n)$ time