Big O Notation

By Roshan Lamichhane

Big-O notation is used in the analysis of algorithms to describe an algorithm's usage of computational resources. The worst case running time, or memory usage, of an algorithm is often expressed as a function of the length of its input using big O notation.

– In 6.00 we generally seek to analyze the worst-case running time. However it is not unusual to see a big-O analysis of memory usage.

– An expression in big-O notation is expressed as a capital letter "O", followed by a function (generally) in terms of the variable n, which is understood to be the size of the input to the function you are analyzing.

– This looks like: O(n)

– If we see a statement such as: f(x) is O(n) it can be read as "f of x is big Oh of n"; it is understood, then, that the number of steps to run f(x) is linear with respect to |x|, the size of the input x.

A description of a function in terms of big O notation only provides an upper bound on the growth rate of the function.

– This means that a function that is O(n) is also, technically, O(n2), O(n3), etc

– However, we generally seek to provide the tightest possible bound. If you say an algorithm is O(n3), but it is also O(n2), it is generally best to say O(n2).

Why use Big O Notation?

Big-O notation allows us to compare different approaches for solving problems, and predicts how long it might take to run an algorithm on a very large input.

Here is a list of Big O Functions:

– **O(1):** Constant time. O(1) = O(10) = O(2100) - why? Even though the constants are huge, they are still constant. Thus if you have an algorithm that takes 2100 discrete steps, regardless of the size of the input, the algorithm is still O(1) - it runs in constant time; *it is not dependent upon the size of the input*.

– **O(lg n):** Logarithmic time. This is faster than linear time; O(log10 n) = O(ln n) = O(lg n) (traditionally in Computer Science we are most concerned with lg n, which is the base-2 logarithm – why is this the case?). The fastest time bound for search.

– **O(n):** Linear time. Usually something when you need to examine every single bit of your input.

– **O(n lg n):** This is the fastest time bound we can currently achieve for sorting a list of elements.

– **O(n2):** Quadratic time. Often this is the bound when we have nested loops.

– **O(2n):** Really, REALLY big! A number raised to the power of n is slower than n raised to any power.

Here are some questions for you:

1. Does O(100n2) = O(n2)?

2. Does O(1 4n3) = O(n3)?

3. Does O(n) + O(n) = O(n)?

The answers to all of these are Yes! Why?

Big-O notation is concerned with the long term, or limiting, behavior of functions. If you're familiar with limits, this will make sense - recall that

$$\lim_{x\to\infty} x^2 = \lim_{x\to\infty} 100x^2 = \infty$$

basically, go out far enough and we can’t see a distinction between 100x^2 and ^2. So, when we talk about big-O notation, we always drop coecient multipliers - because they don’t make a difference.

Examples:

Constant Time Algorithms – *O(1)*

```
int n = 1000;
System.out.println("Hey - your input is: " + n);
```

Logarithmic Time Algorithms – *O(log n)*

```
for (int i = 1; i < n; i = i * 2){
    System.out.println("Hey - I'm busy looking at: " + i);
}
```

Linear Time Algorithms – *O(n)*

```
for (int i = 0; i < n; i++) {
    System.out.println("Hey - I'm busy looking at: " + i);
}
```

O(n log n)

```
for (int i = 1; i <= n; i++){
    for(int j = 1; j < 8; j = j * 2) {
        System.out.println("Hey - I'm busy looking at: " + i + " and " + j);
    }
```

Quadratic time

```
 public static void printAllPossibleOrderedPairs(int[] items) {
    for (int firstItem : items) {
        for (int secondItem : items) {
            System.out.println(firstItem + ", " + secondItem);
        }
    }
}
```

O(2n)

```
public void solve(int n, String start, String auxiliary, String end) {
if (n == 1) {
System.out.println(start + " -> " + end);
} else {
solve(n - 1, start, end, auxiliary);
System.out.println(start + " -> " + end);
solve(n - 1, auxiliary, start, end); }
```