## Big-O definition

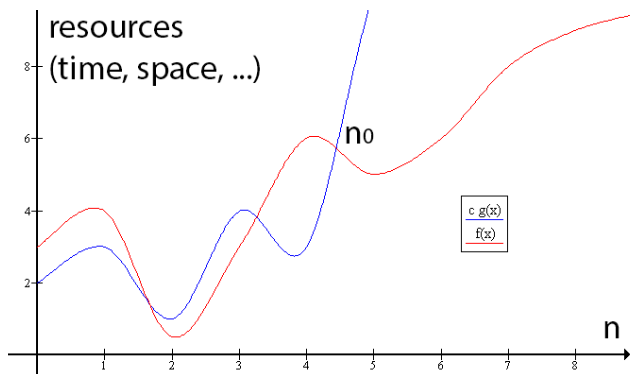
The definition of big-O has a lot of mathematical symbols in it, and so can be very confusing at first. Let's familiarize ourselves with the formal definition and get an intuition behind what it's saying.

$O(g(n))$ is a *set* of functions, where $f(n) \in O(g(n))$ if and only if:

there is some ______________________ and some ______________________

such that ______________________ for all ______________________.

## Big-O intuition




*To the left of $n_0$, the functions can do anything.*
*To its right, $c\,g(n)$ is always greater than or equal to $f(n)$.*

Intuitively, $O(g(n))$ is the set of all functions that $g(n)$ can outpace in the long run (with the help of a constant scaling factor). For example, $n^2$ eventually outpaces $3n\log(n)+5n$, so $3n\log(n)+5n \in O(n^2)$. Because we only care about long run behavior, we generally can discard constants and can consider only the most significant term in a function.

There are actually *infinitely many functions* that are in $O(g(n))$: If $f(n) \in O(g(n))$, then $\frac{1}{2}f(n) \in O(g(n))$ and $\frac{1}{4}f(n) \in O(g(n))$ and $2f(n) \in O(g(n))$. In general, for any constants $k_1, k_2$, $k_1 f(n) + k_2 \in O(g(n))$.

## Checkpoint 0

Using the formal definition of big-O, prove that $n^3 + 9n^2 - 7n + 2 \in O(n^3)$.

$c =$ ______________, $n_0 =$ ______________

**To show:** ______________________________ (expand $c$ and $n_0$)

A. $n \geq$ ______________________________ by assumption

B. ______________________________ by ______________

C. ______________________________ by ______________

D. ______________________________ by ______________

E. ______________________________ by ______________

F. ______________________________ by ______________

## Simplest, tightest bounds

Something that will come up often with big-O is the idea of a *simple* and *tight* bound on the runtime of a function.

It's technically correct to say that linear search is in $O(3n + 2)$ where $n$ is the length of the input array, but $O(3n + 2)$ consists of the exact same functions as $O(n)$, which is simpler.

It's also technically correct to say that binary search, which takes around $\log n$ steps on an array of length $n$, is $O(n!)$, since $n! > \log n$ for all $n > 0$ but it's not very useful. If we ask for a *tight* bound, we want the closest bound you can give. For binary search, $O(\log n)$ is a tight bound because no function that grows more slowly than $\log n$ provides a correct upper bound for binary search.

**Unless we specify otherwise, we want the simplest, tightest bound!**

## Complexity Classes

Big-O sets in simplest and tightest form are used to summarize the complexity of a given function — for example $n^3 + 9n^2 - 7n + 2 \in O(n^3)$ highlights that $n^3 + 9n^2 - 7n + 2$ is a cubic function. As such, big-O sets in simplest and tightest form are called *complexity classes*.

When working with functions with a single argument, say $n$, the most common complexity classes we will encounter in this course are

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(2^n) \subset O(n!)$$

Every function in the big-O set on the left of the *subset symbol* ($\subset$) is also a function in the big-O set on the right (but not necessarily vice versa) — for example $O(\log n) \subset O(n)$ says that every function in $O(\log n)$ is also in $O(n)$.

We use big-O sets in simplest and tightest form also to classify functions with multiple arguments.

## Checkpoint 1

For each of the following big-O sets, give an equivalent big-O set in simplest and tightest form.

$O(3n^{2.5}+2n^2)$ can be written more simply as ______________________________

$O(\log_{10} n+\log_2(7n))$ can be written more simply as ______________________________

One interesting consequence of this second result is that $O(\log_i n) = O(\log_j n)$ for all $i$ and $j$ (as long as they're both greater than 1), because of the change of base formula:

$$\log_i n = \frac{\log_j n}{\log_j i}$$

But $\frac{1}{\log_j i}$ is just a constant! So, it doesn't matter what base we use for logarithms in big-O notation.

When we ask for the *simplest, tightest bound* in big-$O$, we'll usually take points off if you write, for instance, $O(\log_2 n)$ instead of the simpler $O(\log n)$.

## Checkpoint 2

Give the complexity class of the following functions:

$f(n) = 16n^2+5n+2 \in$ ______________________________

$g(n, m) = n^{1.5}\times 16m \in$ ______________________________

$h(x, y) = \max(x, y)+x^2 \in$ ______________________________

# Checkpoint 3

Determine the big-O complexity of the following function.

```
1  int bigO_1(int k) {
2    int[] A = alloc_array(int, k); // allocating an k-length array takes O(k) time
3    for (int i = 0; i < k; i++) {
4      for (int j = 1; j < k; j*=2) {
5        A[i] += j;
6      }
7    }
8    int p = 0;
9    while (p < 10) {
10     f(A, k); //assume f takes O(k) time
11     p++;
12   }
13   return A[k-1];
14 }
```

Always write your complexity in terms of the input variables!

- Line 2 takes time in $O($ ______________________ $)$
- The loop on lines 3–7 runs ______________________ times
  - The loop on lines 4–6 runs ______________________ times
    - Each run of line 5 takes time in $O($ ______________________ $)$

    Therefore the loop on lines 4–6 takes time in $O($ ______________________ $)$

  Therefore the loop on lines 3–7 takes time in $O($ ______________________ $)$
- Line 8 takes time in $O($ ______________________ $)$
- The loop on lines 9–12 runs ______________________ times
  - Each run of line 10 takes time in $O($ ______________________ $)$
  - Each run of line 11 takes time in $O($ ______________________ $)$

  Therefore the loop on lines 9–12 takes time in $O($ ______________________ $)$
- Line 13 takes time in $O($ ______________________ $)$

Thus, the function `bigO_1` takes time in $O($ ______________________ $)$ to run altogether

# Checkpoint 4

Determine the big-O class of the following function. You may use the lines on the right for scratch work.

```
int bigO_2(int[] L, int m, int n)
//@requires \length(L) == m && m > 0;
{
  int[] A = alloc_array(int, n);    // ____________________

  for (int i = 0; i < n; i++) {     // ____________________
    for (int j = i; j < n; j++) {   // ____________________
      A[i] = i * j;                 // ____________________
    }                               // ____________________
  }                                 // ____________________
  int c = m;                        // ____________________

  while (c > 0) {                   // ____________________
    L[c] += 122;                    // ____________________
    c /= 4;                         // ____________________
  }                                 // ____________________
  return L[m/2];                    // ____________________
}                                   // ____________________
```

The big-O class of this function is ______________________________.