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.

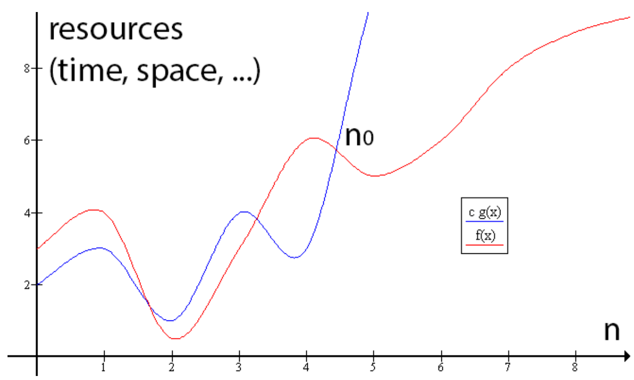
$O(g(n))$ is a *set* of functions, where $f(n) \in O(g(n))$ if and only if:

there is some ____________________ and some ____________________

such that ____________________ for all ____________________,

Although it isn't technically correct set notation, it is also common to write $f(n) = O(g(n))$.

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

Rank these big-O sets from left to right such that every big-O is a subset of everything to the right of it. (For instance, $O(n)$ goes farther to the left than $O(n!)$ because $O(n) \subset O(n!)$.) If two sets are the same, put them on top of each other.

$O(n!)$ $O(n)$ $O(4)$ $O(n\log n)$ $O(4n+3)$ $O(n^2 + 20000n + 3)$ $O(1)$ $O(n^2)$ $O(2^n)$
$O(\log n)$ $O(\log^2 n)$ $O(\log(\log n))$

Checkpoint 1

Using the formal definition of big-O, prove that $n^3 + 300n^2 \in O(n^3)$.

Simplest, tightest bounds

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

It's 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!

Checkpoint 2

Simplify the following big-O bounds without changing the sets they represent:

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

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

One interesting consequence of the second result in Checkpoint 2 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 3

Give the simplest, tightest bound for the following functions:

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

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

$h(x,y,z) = \max(x,y)+z^{16} \in$ ______________________________

Checkpoint 4

For the following two functions, determine the big-O bound:

```
int bigO_1(int n) {
  int[] A = alloc_array(int, n);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      A[i] += j;
    }
  }
  f(A, n); //assume f takes O(log(n)) time
  return A[n-1];
}
```

```
int bigO_2(int[] L, int n) {
  int[] A = alloc_array(int, n);

  for (int i = 0; i < n; i++)
    A[i] = L[i];

  for (int i = 0; i < n; i++) {
    c = n;
    while (c > 0) {
      L[i] += 122;
      c /= 4;
    }
  }
  return L[n/2];
}
```