Algorithm design

- Formulate the problem precisely
- Design an algorithm to solve the problem
- Prove the algorithm is correct
- Analyze the algorithm’s running time

Big-O: Motivation

What is the running time of this algorithm? How many “primitive steps” are executed for an input array A of size n?

```plaintext
sum = 0
n ← length of array A
for i = 1 to n do
    for j = 1 to n do
        sum += A[i]*A[j]
    end for
end for
```

The (worst-case) running time as a function of n is

\[ T(n) = 2n^2 + n + 2. \]

We would like to coarsely categorize this as \( O(n^2) \) — that is, ignore low-order terms and constant multipliers.

Big-O: Formal Definition

**Definition:** The function \( T(n) \) is \( O(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that

\[ T(n) \leq cf(n) \text{ for all } n \geq n_0 \]

We say that \( f \) is an asymptotic upper bound for \( T \).

**Example:**

\[ T(n) = 2n^2 + n + 2 \]

\[ \leq 2n^2 + n^2 + 2n^2 \text{ if } n \geq 1 \]

\[ T(n) \leq \frac{5}{c} n^2 \text{ if } n \geq \frac{1}{n_0} \]

So \( T(n) \) is \( O(n^2) \)

Big-O: Examples

**Claim** \( n^2 + 10^6n \) is \( O(n^2) \)

To prove this we need to show that

\[ n^2 + 10^6n \leq cn^2 \text{ for all } n \geq n_0 \]

**Clicker.** Which values of \( c \) and \( n_0 \) make this inequality true?

A. \( c = 2, n_0 = 10^6 \)
B. \( c = 10^6 + 1, n_0 = 1 \)
C. Both A and B
D. Neither A nor B

Big-O: Examples

- If \( T(n) = n^2 + 1000000n \) then \( T(n) \) is \( O(n^2) \)
- If \( T(n) = n^3 + n \log n \) then \( T(n) \) is \( O(n^3) \)
- If \( T(n) = 2\sqrt{\log n} \) then \( T(n) \) is \( O(n) \)
Clicker

Let \( f(n) = 4n^2 + 23n \log_2 n + 500 \). Which of the following are true?

A. \( f(n) \) is \( O(n^2) \)
B. \( f(n) \) is \( O(n^3) \)
C. Both A and B
D. Neither A nor B

The Big Idea: How to Use Big-O

Study pseudocode to determine running time \( T(n) \) of an algorithm as a function of \( n \):

\[
T(n) = 2n^2 + n + 2
\]

Prove that \( T(n) \) is asymptotically upper-bounded by simpler function using big-O definition:

\[
T(n) = 2n^2 + n + 2 \\
\leq 2n^2 + n^2 + 2n^2 \text{ if } n \geq 1 \\
\leq 5n^2 \text{ if } n \geq 1
\]

This is the right way to think about big-O, but too much work. We’ll develop properties of big-O that simplify proving big-O bounds, and use these properties to take shortcuts while analyzing algorithms (you probably learned the shortcuts without knowing formal justification).

Properties of Big-O Notation

Claim (Transitivity): If \( f \) is \( O(g) \) and \( g \) is \( O(h) \), then \( f \) is \( O(h) \).

Example:

\[
\frac{2n^2 + n + 1}{f(n)} \text{ is } O(\frac{n^2}{g(n)})
\]

\[
\frac{n^2}{g(n)} \text{ is } O(\frac{n^3}{h(n)})
\]

Therefore, \( 2n^2 + n + 1 \) is \( O(n^3) \)

Transitivity Proof

Claim (Transitivity): If \( f \) is \( O(g) \) and \( g \) is \( O(h) \), then \( f \) is \( O(h) \).

Proof: we know from the definition that

\[
\begin{align*}
    f(n) &\leq c_{g0} g(n) \quad \text{ for all } n \geq n_0 \\
g(n) &\leq c'h(n) \quad \text{ for all } n \geq n'_0
\end{align*}
\]

Therefore

\[
\begin{align*}
f(n) &\leq c_{g0} g(n) \quad \text{ if } n \geq n_0 \\
&\leq c_{g0} c'h(n) \quad \text{ if } n \geq n_0 \text{ and } n \geq n'_0 \\
&= \underbrace{c_{g0} c'}_{c} h(n) \quad \text{ if } n \geq \max\{n_0, n'_0\} \\
f(n) &\leq c'h(n) \quad \text{ if } n \geq n'_0
\end{align*}
\]

Know how to do proofs using Big-O definition.

Properties of Big-O Notation

Claims (Additivity):

- If \( f \) is \( O(h) \) and \( g \) is \( O(h) \), then \( f + g \) is \( O(h) \).

\[
\frac{3n^2 + n^3}{O(n^2)} \text{ is } O(n^3)
\]

- If \( f \) is \( O(g) \), then \( f + g \) is \( O(g) \)

\[
\frac{n^3 + 23n + n \log n}{g(n)} \text{ is } O(n^3)
\]

Significance of Additivity

- OK to drop lower order terms:

\[
2n^3 + 10n^3 + 4n \log n + 1000n \text{ is } O(n^3)
\]

- Polynomials: Only highest-degree term matters. If \( a_d > 0 \) then:

\[
a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d \text{ is } O(n^d)
\]

- You are using additivity when you ignore the running time of statements outside for loops!
Other Useful Facts: Log vs. Poly vs. Exp

Fact: \( \log_b(n) \) is \( O(n^d) \) for all \( b, d > 0 \)

All polynomials grow faster than logarithm of any base

Fact: \( n^d \) is \( O(r^n) \) when \( r > 1 \)

Exponential functions grow faster than polynomials

Logarithm review

Definition: \( \log_b(a) \) is the unique number \( c \) such that \( b^c = a \)

Informally: the number of times you can divide \( a \) into \( b \) parts until each part has size one

Properties:

▶ Log of product → sum of logs
  
  \[ \log(xy) = \log x + \log y \]
  
  \[ \log(x^k) = k \log x \]

▶ \( \log_b(\cdot) \) is inverse of \( b^\cdot \)
  
  \[ \log_b(b^n) = n \]
  
  \[ b^{\log_b(n)} = n \]

▶ \( \log_a n = \log_b b \cdot \log_b n \) (logs in any two bases are proportional)

(\( \text{const.} \))

When using big-O, it’s OK not to specify base. Assume \( \log_2 \) if not specified.

---

Big-O comparison

Which grows faster?

\[ n(\log n)^3 \quad \text{vs.} \quad n^{4/3} \]

simplifies to

\[ (\log n)^3 \quad \text{vs.} \quad n^{1/3} \]

simplifies to

\[ \log n \quad \text{vs.} \quad n^{1/9} \]

▶ We know \( \log n \) is \( O(n^d) \) for all \( d > 0 \)
  
  \[ \Rightarrow \log n \text{ is } O(n^{1/9}) \]
  
  \[ \Rightarrow n(\log n)^3 \text{ is } O(n^{4/3}) \]

Apply transformations (monotone, invertible) to both functions. Try taking log.

---

Logarithm review

Definition: \( \log_b(a) \) is the unique number \( c \) such that \( b^c = a \)

Informally: the number of times you can divide \( a \) into \( b \) parts until each part has size one

Properties:

▶ Log of product → sum of logs
  
  \[ \log(xy) = \log x + \log y \]
  
  \[ \log(x^k) = k \log x \]

▶ \( \log_b(\cdot) \) is inverse of \( b^\cdot \)
  
  \[ \log_b(b^n) = n \]
  
  \[ b^{\log_b(n)} = n \]

▶ \( \log_a n = \log_b b \cdot \log_b n \) (logs in any two bases are proportional)

\( \text{const.} \)

When using big-O, it’s OK not to specify base. Assume \( \log_2 \) if not specified.

---

Big-O: Correct Usage

Big-O: a way to categorize growth rate of functions relative to other functions.

Not: "the running time of my algorithm".

Correct Usage:

▶ The worst-case running time of the algorithm in input of size \( n \) is \( T(n) \).
  
  \[ T(n) \text{ is } O(n^3) \]
  
  The running time of the algorithm is \( O(n^3) \).

Incorrect Usage:

▶ \( O(n^3) \) is the running time of the algorithm. (There are many different asymptotic upper bounds to the running time of the algorithm.)