Algorithm design

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

Example: Binary vs. Linear Search

An elegant algorithm you can teach to a 5-year old. You lose your page in 256-page book:

- Linear search: 1, 2, 3, 4, ..., 256. search up to 256 pages
- Binary search: 128, 64, 32, 16, 8, 4, 2, 1. search up to 8 pages

<table>
<thead>
<tr>
<th># pages</th>
<th>linear</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>9</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>10</td>
</tr>
<tr>
<td>2048</td>
<td>2048</td>
<td>11</td>
</tr>
<tr>
<td>n</td>
<td>≤ n</td>
<td>≤ log(n)</td>
</tr>
</tbody>
</table>

Example: Binary vs. Linear Search

Board example: plot of $n$ vs. $\log(n)$

Take-aways:
- Measure running time (# steps) as function of input size ($n$)
- Need tools to compare growth rates of functions
- Big difference between brute-force and clever algorithms!

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
\( n \leftarrow \) length of array $A$
for $i = 1$ to $n$
do
  for $j = 1$ to $n$
do
    sum += $A[i] \cdot A[j]$

The (worst-case) running time as a function of $n$ has the form

$$T(n) = an^2 + bn + c$$

We would like to coarsely categorize this as “order $n^2$” or $O(n^2)$
- Ignore constants, lower-order terms
- Need tools to compare growth rates of functions: “asymptotic order notation” (big-O)

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 $O(n^2)$
Example:

Example: \( T(n) = 2n^2 + n + 2 \) is \( O(n^3) \)

\[
T(n) = 2n^2 + n + 2 \\
\leq 2n^3 + n^3 + 2n^3 \quad \text{if } n \geq 1 \\
T(n) \leq \frac{5}{c} n^3 \quad \text{if } n \geq \frac{1}{n_0}
\]

Big-O bounds do not need to be tight!

Big-O: Examples

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

To prove this we need to show that

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

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 \( T(n) = n^2 + 10^6 n \) 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 \( O(n) \)

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:

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

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:

\[
\frac{2n^2 + n + 1}{g(n)} \quad O(n^2) \\
\frac{n^2}{h(n)} \quad O(n^3) \\
\]

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
- \( f(n) \leq c_1g(n) \) for all \( n \geq n_0 \)
- \( g(n) \leq c_2h(n) \) for all \( n \geq n'_0 \)

Therefore
\[
f(n) \leq c_1g(n) \quad \text{if} \quad n \geq n_0
\]
\[
= c_1c_2^{-1} h(n) \quad \text{if} \quad n \geq \max\{n_0, n'_0\}
\]
\[
f(n) \leq c_3h(n) \quad \text{if} \quad n \geq n''_0
\]

Know how to do proofs using Big-O definition.

Significance of Additivity

- OK to drop lower order terms:
  \[
  2n^5 + 10n^3 + 4n \log n + 1000n \text{ is } O(n^5)
  \]
- Polynomials: Only highest-degree term matters. If \( a_d > 0 \) then:
  \[
  a_0 + a_1n + a_2n^2 + \ldots + a_dn^d \text{ is } O(n^d)
  \]
- You are using additivity when you ignore the running time of statements outside for loops!

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 \( \rightarrow \) sum of logs
  - \( \log(xy) = \log x + \log y \)
  - \( \log(b^x) = x \log b \)
- \( \log_b \) is inverse of \( b^c \)
  - \( \log_b(b^n) = n \)
  - \( b^{\log_b(n)} = n \)
- \( \log_{b_1} a = \log_b n \log_{b_1} b \) (logs in any two bases are proportional)

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

Properties of Big-O Notation

Claims (Additivity):
- If \( f \) is \( O(g) \) and \( g \) is \( O(h) \), then \( f + g \) is \( O(h) \).
- \( 3n^2 + n^4 \) is \( O(n^5) \)
- If \( f \) is \( O(g) \), then \( f + g \) is \( O(g) \)
  \[
  n^3 + 23n + n \log n \text{ is } O(n^3)
  \]

Other Useful Facts: Log vs. Poly vs. Exp

Fact: \( \log_b(n) \) is \( O(n^d) \) for all \( b > 1, d > 0 \)
All polynomials grow faster than logarithm of any base

Fact: \( n^d \) is \( O(n^r) \) when \( r > 1 \)
Exponential functions grow faster than polynomials

Big-O comparison

Which grows faster?
- \( n(\log n)^3 \) vs. \( n^{4/3} \)
  simplifies to
- \( (\log n)^3 \) vs. \( n^{1/3} \)
  simplifies to
- \( \log n \) vs. \( n^{1/9} \)
  - We know \( \log n \) is \( O(n^d) \) for all \( d > 0 \)
    \( \Rightarrow \) \( \log n \) is \( O(n^{1/9}) \)
    \( \Rightarrow \) \( n(\log n)^3 \) is \( O(n^{4/3}) \)

Apply transformations (monotone, invertible) to both functions.
Try taking log.
**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 $T(n)$.
- $T(n)$ 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.)