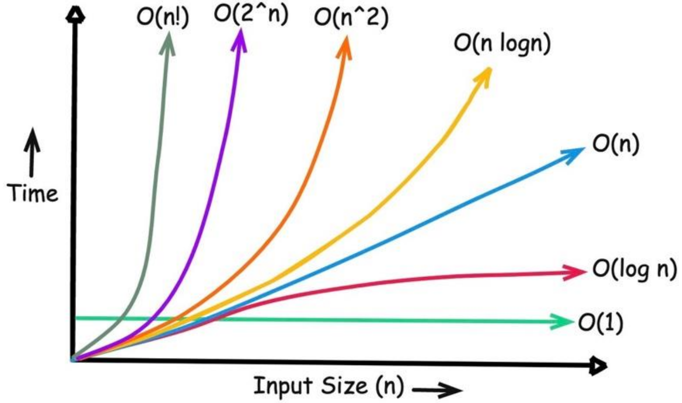
Lecture 3

Algorithm Analysis II



Algorithm Design

1. Formulate the problem precisely
2. Design an algorithm
3. Prove the algorithm is correct
4. Analyze its running time

Big-O Definition

Definition: The function $T(n)$ is $O\big(f(n)\big)$ if there exists 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

Big-O Definition

Definition: The function $T(n)$ is $O\big(f(n)\big)$ if there exists 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:

$$\begin{aligned} T(n) &= 2n^2 + n + 2 \\ &\leq 2n^2 + n^2 + 2n^2 \text{ for } n \geq 1 \\ &= 5n^2 \end{aligned}$$

c $f(n)$ n_0

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

Exercise I

Let $T(n) = 3n + 17\log_2 n + 1000$. Which of the following are true?
(Hint: it could be more than one)

i. $T(n)$ is $O(n^2)$

ii. $T(n)$ is $O(n)$

iii. $T(n)$ is $O(\log n)$

Exercise I

Let $T(n) = 3n + 17\log_2 n + 1000$. Which of the following are true?
(Hint: it could be more than one)

i. $T(n)$ is $O(n^2)$

ii. $T(n)$ is $O(n)$

iii. $T(n)$ is $O(\log n)$

Big-O bounds do not need to be tight!

Examples

- If $T(n) = n^2 + 10^6 n$ then $T(n)$ is $O(n^2)$
- If $T(n) = n^3 + 3n \log n$ then $T(n)$ is $O(n^3)$
- If $T(n) = n + 6$ then $T(n)$ is ***not*** $O(1)$

What Does Big-O Mean?

Worst-case analysis

- Running time guarantee for any input of size n
- Typically captures computational complexity in practice

Alternatives

- Average-case analysis
- Expected running time of a randomized algorithm
- Amortized (considering a sequence of operations)

Either not general enough or unwieldy

How to Use Big-O

- Study pseudocode to determine running time $T(n)$ for an algorithm as a function of n

$$T(n) = 2n^2 + n + 2$$

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

$$\begin{aligned} T(n) &= 2n^2 + n + 2 \\ &\leq 2n^2 + n^2 + 2n^2 \text{ for } n \geq 1 \\ &= 5n^2 \end{aligned}$$

- Next time, we will develop properties that simplify proving big-O bounds
  - You've likely come across some already in Data Structures!

Big-O in Practice

A way to categorize the 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 with input size n is $T(n)$
- Suppose $T(n)$ 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

Properties of Big-O

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

Example:

- $\underbrace{2n^2 + n + 1}_{f(n)}$ is $O(\underbrace{n^2}_{g(n)})$
- $\underbrace{n^2}_{g(n)}$ is $O(\underbrace{n^3}_{h(n)})$
- 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 of Big-O that

- $f(n) \leq cg(n)$ for all $n \geq n_0$
- $g(n) \leq c'h(n)$ for all $n \geq n'_0$

Let $n'' = \max\{n, n'\}$. Therefore, for all $n \geq n''$,

$$\begin{aligned} f(n) &\leq cg(n) \\ &\leq c\big(c'h(n)\big) \\ &= cc'h(n) \\ &= c''h(n). \end{aligned}$$

Properties of Big-O

Claims (Additivity):

- If f if $O(h)$ and g is $O(h)$, then $f+g$ is $O(h)$

$$3n^2 + n^4 \text{ is } O(n^5)$$

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

$$n^3 + 23n + n\log n \text{ is } O(n^3)$$

Significance of Additivity

- Okay to drop lower order terms

$$3n^2 + n \log n + 2n^4 \text{ is } O(n^4)$$

- Polynomials: Only the highest-degree term matters (with positive coefficient)

- You are using additivity when you ignore the running time of statements outside of for loops!

Other Useful Facts

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 $O(r^n)$ when $r > 1$

- ❖ Exponential functions grow faster than polynomials

Logarithm Review

Definition: $\log_b n$ is the unique number c such that $b^c = n$

Informally: the number of times you can divide n into b parts until each part has size one

Properties:

- Log of product equals sum of logs
  - $\log(xy) = \log(x) + \log(y)$
  - $\log(x^k) = k \log(x)$
- $\log_b(\cdot)$ is the inverse of $b^{(\cdot)}$
- $\log_a n = (\log_a b) \log_b n$

"Good" Running Time

Inefficiency

- ❖ We said that 2^n steps or worse is unacceptable in practice
- ❖ i.e. $O(2^n)$ or exponential running time is inefficient

"Good" Running Time

Inefficiency

- We said that 2^n steps or worse is unacceptable in practice
- i.e. $O(2^n)$ or exponential running time is inefficient

Efficiency

- An algorithm is *efficient* if it has a polynomial running time
- i.e. $O(n^k), k \geq 0$

"Good" Running Time

Inefficiency

- ❖ We said that 2^n steps or worse is unacceptable in practice
- ❖ i.e. $O(2^n)$ or exponential running time is inefficient

Efficiency

- ❖ An algorithm is *efficient* if it has a polynomial running time
- ❖ i.e. $O(n^k), k \geq 0$

Exceptions

- ❖ Some poly-time algorithms have large constants and exponents
- ❖ We sometimes use exponential-time algorithms when their worst case does not arise in practice

Exercise II

Suppose f is $O(g)$. Which of the following is true?

i. g is $O(f)$
ii. g is not $O(f)$
iii. g may be $O(f)$, depending on f and g

Exercise II

Suppose f is $O(g)$. Which of the following is true?

i. g is $O(f)$

ii. g is not $O(f)$

iii. g may be $O(f)$, depending on f and g

Big-Ω Definition

Informally, T grows at least as fast as f

Definition: The function $T(n)$ is $\Omega\big(f(n)\big)$ if there exists constants $c > 0$ and $n_0 \geq 0$ such that

$$T(n) \geq cf(n) \text{ for all } n \geq n_0$$

We say that f is an **asymptotic lower bound** for T

Big-Ω Definition

Informally, T grows at least as fast as f

Definition: The function $T(n)$ is $\Omega\big(f(n)\big)$ if there exists constants $c > 0$ and $n_0 \geq 0$ such that

$$T(n) \geq cf(n) \text{ for all } n \geq n_0$$

We say that f is an **asymptotic lower bound** for T

What's the difference between Big-O and Big-Ω?

Next Time

- Begin looking at tools for analyzing algorithms, e.g., Big-O notation