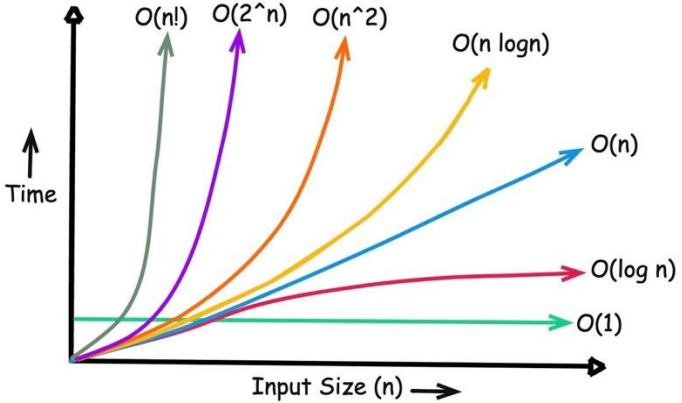
Lecture 2

Algorithm Analysis



Algorithm Design

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

Example: Search

Suppose you lose your place in a 256-page book

- Linear search
  - Search through the pages one-by-one until you find your spot
  - Requires searching up to 256 pages
- Binary search
  - Flip to the middle page; if you've seen it before, eliminate the first half; otherwise, eliminate the second half; repeat
  - Requires searching up to 8 pages

# pages	linear	binary
256	256	8
512	512	9
1024	1024	10
2048	2048	11
n	$\leq n$	$\leq \log(n)$

Counting Steps

Why measure running time by counting steps?

Counting Steps

Why measure running time by counting steps?

- Gives us a universal way of discussing algorithms
- e.g. abstracts away from implementation and hardware details

Counting Steps

Why measure running time by counting steps?

- Gives us a universal way of discussing algorithms
- e.g. abstracts away from implementation and hardware details

What is a step?

Counting Steps

Why measure running time by counting steps?

- Gives us a universal way of discussing algorithms
- e.g. abstracts away from implementation and hardware details

What is a step?

- Restrict to "primitive steps"
- Basic instructions such as $+,\ -,\ *,\ =,\ if$
- Accessing an item in memory (RAM model)

Counting Steps

Why measure running time by counting steps?

- Gives us a universal way of discussing algorithms
- e.g. abstracts away from implementation and hardware details

What is a step?

- Restrict to "primitive steps"
- Basic instructions such as $+,\ -,\ *,\ =,\ if$
- Accessing an item in memory (RAM model)

Measure running time as a function of input size (n)

"Good" Running Time

For many problems, there is a natural *brute-force* search algorithm that checks every possible solution

- Typically takes 2^n steps or worse
- Unacceptable in practice

Big difference in running time between brute-force and clever algorithms!

- Linear vs. binary search

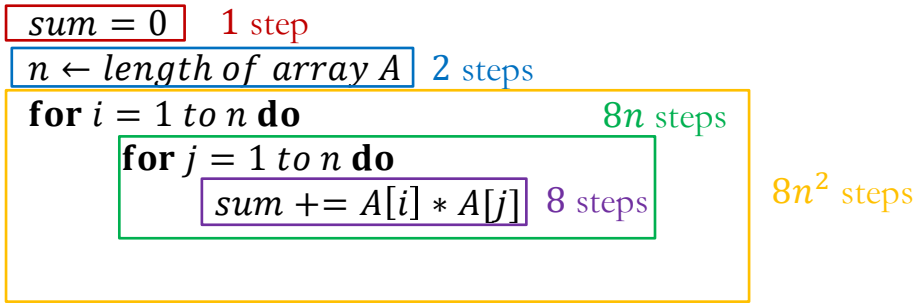
Example

What is the running time of this algorithm?

$$
\begin{aligned}
&sum = 0 \\
&n \leftarrow length\ of\ array\ A \\
&\textbf{for}\ i = 1\ to\ n\ \textbf{do} \\
&\qquad \textbf{for}\ j = 1\ to\ n\ \textbf{do} \\
&\qquad\qquad sum\ +\!= A[i] * A[j]
\end{aligned}
$$

Example

What is the running time of this algorithm?



Algorithm requires approx. $T(n) = 8n^2 + 8n + 3$ steps

- Would like to categorize this as "order n^2" or $O(n^2)$
- Allows us to ignore constants and lower-order terms
- Need tools to compare growth rates of functions: "asymptotic order notation" (big-O)

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!

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

"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

Next Time

- Properties of big-O
- Other asymptotic growth rate tools

Next Time

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