COMP 182 Algorithmic Thinking

Algorithm Efficiency

Luay Nakhleh
Computer Science
Rice University

Reading Material

- Chapter 3, Sections 2-3

Not All Correct Algorithms Are Created Equal

- We often choose the most efficient algorithm among the many correct ones.
- [In some cases, we might choose a slightly slower algorithm that's easier to implement.]
- Efficiency
  - Time: How fast an algorithm runs
  - Space: How much extra space the algorithm takes
  - There's often a trade-off between the two.

It's All a Function of the Input Size

- We often investigate an algorithm's complexity in terms of some parameter n that indicates the input size.
- For an algorithm that sorts a list, n is the number of elements in the list.
- For an algorithm that computes the cardinality of a set, n is the number of elements in the set.

It's All a Function of the Input Size

- In some cases, more than one parameter is needed to capture the input size.
- For an algorithm that computes the intersection of two sets A and B of sizes m and n, respectively, the input size is captured by both m and n.

It's All a Function of the Input Size

- An algorithm that operates on graphs:
  - If the graph is represented by its adjacency list, then the input size is often given by m (number of edges) and n (number of nodes).
  - If the graph is represented by its adjacency matrix, then the input size is often given by n alone (the number of nodes).
  - However, it the choice of the input size depends on the algorithm and the assumptions on the input.

It's All a Function of the Input Size

- Consider the following simple algorithm for testing whether integer p is prime:
  - divide p by every number between 2 and p-1; if the remainder is 0 in at least one case, return False, otherwise return True.
  - What is the input size?

Units for Measuring Running Time

- We'd like to use a measure that does not depend on extraneous factors such as the speed of a particular computer, the hardware being used, the quality of a program implementing the algorithm, or the difficulty of clocking the actual running time of the program.
- We usually focus on basic operations that the algorithm performs, and compute the number of times those basic operations are executed.

The Growth of Functions

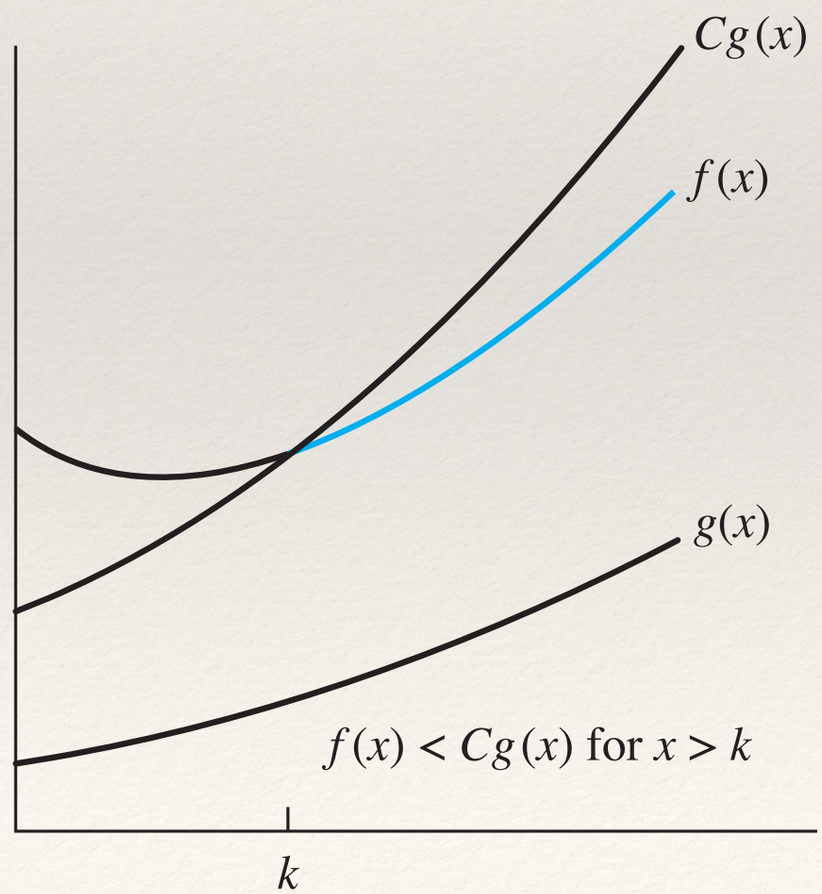
- A difference in running times on small inputs is not what distinguishes efficient algorithms from inefficient ones.
- When analyzing the complexity of an algorithm, we pay special attention to the <u>order of growth</u> of the number of steps that the algorithm takes as the input size increases.

Big-O

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|$$

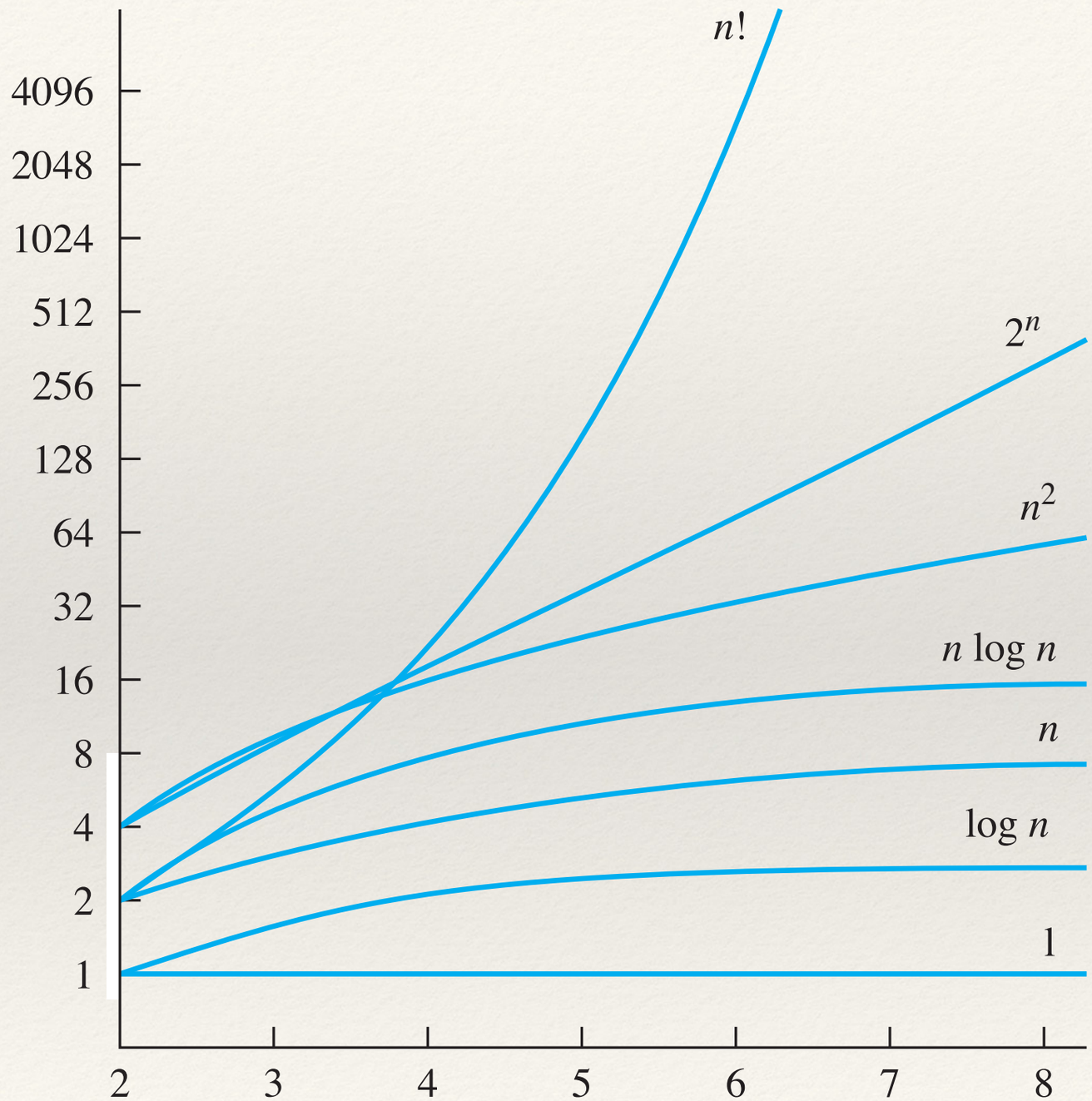
whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]



Big-O

- $n^2 = O(n^2)$
- $3n^2+100 = O(n^2)$
- $n^2 \neq O(n)$
- $5n^2 = O(n^3)$
- $1000 = O(1)$
- $n \neq O(\log n)$

Big-O


$n!$
2^n
n^2
$n \log n$
n
$\log n$
1
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
2
3
4
5
6
7
8

Big-O: Important Theorems

Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_{n-1}, a_n$ are real numbers. Then $f(x)$ is $O(x^n)$.

Suppose that $f_1(x)$ is $O(g_1(x))$ and that $f_2(x)$ is $O(g_2(x))$. Then $(f_1 + f_2)(x)$ is $O(\max(|g_1(x)|, |g_2(x)|))$.

Suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$. Then $(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$.

Big-O

- Of course, $17n^4-32n^2+5n+107=O(17n^4-32n^2+5n+107)$.
- If the running time of an algorithm is $f(x)$, the goal in using big-O notation is to choose a function $g(x)$ such that (1) $f(x) = O(g(x))$, and (2) $g(x)$ grows as slowly as possible so that to provide as tight an upper bound on $f(x)$ as possible.

Big-Omega

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that

$$|f(x)| \geq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-Omega of $g(x)$."]

Big-O and Big-Omega

$$f(x) = \Omega(g(x)) \leftrightarrow g(x) = O(f(x))$$

Big-Theta

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$. When $f(x)$ is $\Theta(g(x))$ we say that f is big-Theta of $g(x)$, that $f(x)$ is of *order* $g(x)$, and that $f(x)$ and $g(x)$ are of the *same order*.

Big-O, Big-Omega, and Big-Theta

$$f(x) = \Theta(g(x)) \leftrightarrow [f(x) = O(g(x)) \wedge f(x) = \Omega(g(x))]$$

Complexity of Algorithms

- The complexity analysis framework ignores multiplicative constants and concentrates on the order of growth of the number of steps to within a constant multiplier for large-size inputs.

Complexity: Problem vs. Algorithm

- The complexity of a problem is not a statement about a specific algorithm for the problem.
- The complexity of an algorithm pertains to that specific algorithm.

Complexity: Problem vs. Algorithm

- Example 1:
  - When we say that the (comparison-based) sorting problem is solvable on the order of $n \log n$ algorithm, this means that there exists an algorithm for sorting that runs in $n \log n$ time.
  - However, there are sorting algorithms that take more than $n \log n$ operations.

Complexity: Problem vs. Algorithm

- Example 2:
  - When we say that a problem is NP-Complete, it means that we do not know of any polynomial-time algorithm for it.
  - However, there could be two algorithms for solving the problem, one that takes on the order of 2^n operations and the other takes on the order of 5^n operations.

Worst, Best, and Average Cases

- For an algorithm and input size n:
  - Worst case: The input of size n that results in the largest number of operations.
  - Best case: The input of size n that results in the smallest number of operations.
  - Average case: The expected number of operations that the algorithm takes on an input of size n.

Worst, Best, and Average Cases

- Important: These cases are defined for a given input size n, so you cannot change the size to get worst or best complexities.
- For example, in analyzing the running time of a graph algorithm and using input size n and m, you cannot say the best case is when the graph has no edges, since that means you set m to 0 (this would be analogous to saying the best case of a sorting algorithm is an empty list!).

Worst, Best, and Average Cases

- The best-case of an algorithm is usually uninformative about the performance of the algorithm, but gives a lower bound (not necessarily tight) on the complexity of the algorithm.
- The average-case analysis is usually very hard to establish (we'll see some examples when we cover discrete probability) and requires making assumptions about the input.
- The worst-case of an algorithm bounds the running time of the algorithm from above and gives an upper bound on the complexity of the problem. It is the most commonly used case in algorithm complexity analyses.

Complexity of Algorithms

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Complexity of Algorithms

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

* more than 10^{100} years!

Complexity of Algorithms

Algorithm 3: LinearSearch.

Input: An array $A[0 \ldots n-1]$ of integers, and an integer x.
Output: The index of the first element of A that matches x, or -1 if there are no matching elements.

```
i ← 0;
while i < n and A[i] ≠ x do
    i ← i + 1;
if i ≥ n then
    i ← −1;
return i
```

Complexity of Algorithms

Algorithm 4: MatrixMultiplication.

Input: Two matrices $A[0 \ldots n-1, 0 \ldots k-1]$ and $B[0 \ldots k-1, 0 \ldots m-1]$.
Output: Matrix $C = AB$.

```
for i ← 0 to n − 1 do
    for j ← 0 to m − 1 do
        C[i, j] ← 0;
        for l ← 0 to k − 1 do
            C[i, j] ← C[i, j] + A[i, l] · B[l, j];
return C
```

Complexity of Algorithms

Algorithm 1: IsBipartite.

Input: Undirected graph $g = (V, E)$.
Output: *True* if g is bipartite, and *False* otherwise.

```
foreach Non-empty subset V_1 ⊂ V do
    V_2 ← V \ V_1;
    bipartite ← True;
    foreach Edge {u, v} ∈ E do
        if {u, v} ⊆ V_1 or {u, v} ⊆ V_2 then
            bipartite ← False;
            Break;
    if bipartite = True then
        return True;
return False;
```

(In)Tractability

- If there exists an polynomial-time worst-case algorithm for problem *P*, we say that *P* is <u>tractable</u>.
- If no polynomial-time worst-case algorithm exists for problem *P*, we say *P* is <u>intractable</u>.

P vs. NP

- Tractable problems belong to <u>class P</u>.
- Problems for which a solution can be verified in polynomial time belong to <u>class NP</u>.
- Class NP-Complete consists of problems with the property that if one of them can be solved by a polynomial-time worst-case algorithm, then all problems in class NP can be solved by polynomial-time worst-case algorithms.

P vs. NP

- The million-dollar question: Is P = NP?

Questions?