

Computer Science & Engineering 423/823

Design and Analysis of Algorithms

Lecture 01 — Shall We Play A Game?

Stephen Scott

sscott@cse.unl.edu

Introduction

- ▶ In this course, I assume that you have learned several fundamental concepts on basic data structures and algorithms
- ▶ Let's confirm this
- ▶ What do I mean ...

... when I say: “Asymptotic Notation”

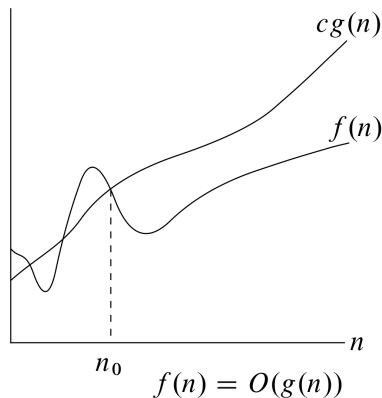
- ▶ A convenient means to succinctly express the growth of functions
 - ▶ Big- O
 - ▶ Big- Ω
 - ▶ Big- Θ
 - ▶ Little- o
 - ▶ Little- ω
- ▶ Important distinctions between these (**not interchangeable**)

Asymptotic Notation

... when I say: “Big-O”

Asymptotic upper bound

$$O(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq f(n) \leq c g(n)\}$$



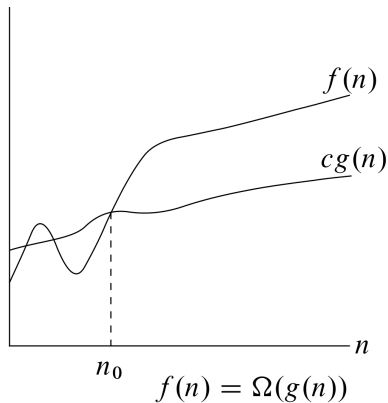
Can **very loosely and informally** think of this as a “ $\leq$ ” relation between functions

Asymptotic Notation

... when I say: “Big-Ω”

Asymptotic lower bound

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c g(n) \leq f(n)\}$$



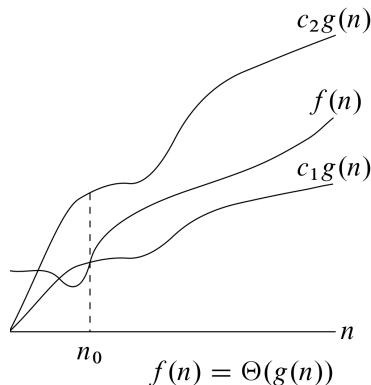
Can **very loosely and informally** think of this as a “ $\geq$ ” relation between functions

Asymptotic Notation

... when I say: “Big- Θ ”

Asymptotic tight bound

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$



Can **very loosely and informally** think of this as a “=” relation between functions

Asymptotic Notation

... when I say: “Little-o”

Upper bound, not asymptotically tight

$$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq f(n) < c g(n)\}$$

Upper inequality strict, and holds for **all** $c > 0$

Can **very loosely and informally** think of this as a “ $<$ ” relation between functions

Asymptotic Notation

... when I say: “Little- ω ”

Lower bound, not asymptotically tight

$$\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c g(n) < f(n)\}$$

$$f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$$

Can **very loosely and informally** think of this as a “ $>$ ” relation between functions

... when I say: “Upper and Lower Bounds”

- ▶ Most often, we analyze algorithms and problems in terms of **time complexity** (number of operations)
- ▶ Sometimes we analyze in terms of **space complexity** (amount of memory)
- ▶ Can think of **upper** and **lower** bounds of time/space for a specific **algorithm** or a general **problem**

Upper and Lower Bounds

... when I say: “Upper Bound of an Algorithm”

- ▶ The most common form of analysis
- ▶ An algorithm A has an **upper bound** of $f(n)$ for input of size n if there exists **no input** of size n such that A requires more than $f(n)$ time
- ▶ E.g., we know from prior courses that Quicksort and Bubblesort take no more time than $O(n^2)$, while Mergesort has an upper bound of $O(n \log n)$
 - ▶ (But why is Quicksort used more in practice?)
- ▶ **Aside:** An algorithm's lower bound (not typically as interesting) is like a best-case result

Upper and Lower Bounds

... when I say: “Upper Bound of a Problem”

- ▶ A problem has an **upper bound** of $f(n)$ if there exists **at least one** algorithm that has an upper bound of $f(n)$
 - ▶ I.e., there exists an algorithm with time/space complexity of at most $f(n)$ on **all** inputs of size n
- ▶ E.g., since Mergesort has worst-case time complexity of $O(n \log n)$, the problem of sorting has an upper bound of $O(n \log n)$

Upper and Lower Bounds

... when I say: “Lower Bound of a Problem”

- ▶ A problem has a **lower bound** of $f(n)$ if, for **any** algorithm A to solve the problem, there exists **at least one** input of size n that forces A to take at least $f(n)$ time/space
- ▶ This pathological input depends on the specific algorithm A
- ▶ E.g., there is an input of size n (reverse order) that forces Bubblesort to take $\Omega(n^2)$ steps
- ▶ Also e.g., there is a different input of size n that forces Mergesort to take $\Omega(n \log n)$ steps, but none exists forcing $\omega(n \log n)$ steps
- ▶ Since **every** sorting algorithm has an input of size n forcing $\Omega(n \log n)$ steps, the sorting problem has a **time complexity lower bound** of $\Omega(n \log n)$
 - ⇒ Mergesort is **asymptotically optimal**

Upper and Lower Bounds

... when I say: “Lower Bound of a Problem” (2)

- ▶ To argue a lower bound for a problem, can use an **adversarial** argument:
An algorithm that simulates **arbitrary** algorithm A to build a pathological input
- ▶ Needs to be in some general (algorithmic) form since the nature of the pathological input depends on the specific algorithm A
- ▶ Can also **reduce** one problem to another to establish lower bounds
 - ▶ **Spoiler Alert:** This semester we will show that if we can compute convex hull in $o(n \log n)$ time, then we can also sort in time $o(n \log n)$; this cannot be true, so convex hull takes time $\Omega(n \log n)$

... when I say: “Efficiency”

- ▶ We say that an algorithm is **time-** or **space-efficient** if its worst-case time (space) complexity is $O(n^c)$ for constant c for input size n
- ▶ I.e., polynomial in the size of the input
- ▶ **Note on input size:** We measure the size of the input in terms of the **number of bits** needed to represent it
 - ▶ E.g., a graph of n nodes takes $O(n \log n)$ bits to represent the nodes and $O(n^2 \log n)$ bits to represent the edges
 - ▶ Thus, an algorithm that runs in time $O(n^c)$ is efficient
 - ▶ In contrast, a problem that includes as an input a numeric parameter k (e.g., threshold) only needs $O(\log k)$ bits to represent
 - ▶ In this case, an efficient algorithm for this problem **must** run in time $O(\log^c k)$
 - ▶ If instead polynomial in k , sometimes call this **pseudopolynomial**

... when I say: “Recurrence Relations”

- ▶ We know how to analyze non-recursive algorithms to get asymptotic bounds on run time, but what about recursive ones like Mergesort and Quicksort?
- ▶ We use a **recurrence relation** to capture the time complexity and then bound the relation asymptotically
- ▶ E.g., Mergesort splits the input array of size n into two sub-arrays, recursively sorts each, and then merges the two sorted lists into a single, sorted one
- ▶ If $T(n)$ is time for Mergesort on n elements,

$$T(n) = 2T(n/2) + O(n)$$

- ▶ Still need to get an asymptotic bound on $T(n)$

Recurrence Relations

... when I say: “Master Theorem” or “Master Method”

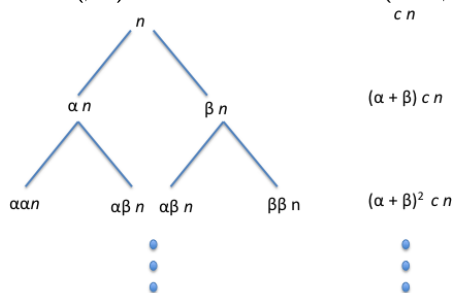
- ▶ **Theorem:** Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined as $T(n) = aT(n/b) + f(n)$. Then $T(n)$ is bounded as follows:
 1. If $f(n) = O(n^{\log_b a - \epsilon})$ for constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
 2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
 3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for constant $c < 1$ and sufficiently large n , then $T(n) = \Theta(f(n))$
- ▶ E.g., for Mergesort, can apply theorem with $a = b = 2$, use case 2, and get $T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$

Recurrence Relations

Other Approaches

Theorem: For recurrences of the form $T(\alpha n) + T(\beta n) + O(n)$ for $\alpha + \beta < 1$,
 $T(n) = O(n)$

Proof: Top $T(n)$ takes $O(n)$ time ($= cn$ for some constant c). Then calls to $T(\alpha n)$ and $T(\beta n)$, which take a total of $(\alpha + \beta)cn$ time, and so on



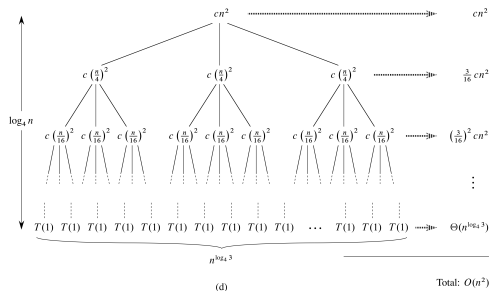
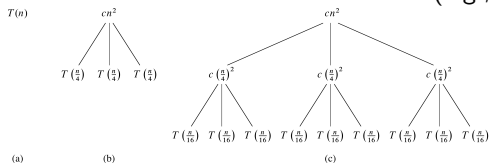
Summing these infinitely yields (since $\alpha + \beta < 1$)

$$cn(1 + (\alpha + \beta) + (\alpha + \beta)^2 + \cdots) = \frac{cn}{1 - (\alpha + \beta)} = c'n = O(n)$$

Recurrence Relations

Still Other Approaches

Previous theorem special case of **recursion-tree method**: (e.g., $T(n) = 3T(n/4) + O(n^2)$)

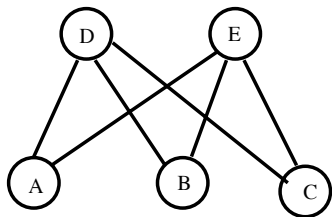


Another approach is **substitution method** (guess and prove via induction)

Graphs

... when I say: “(Undirected) Graph”

A **(simple, or undirected)** graph $G = (V, E)$ consists of V , a nonempty set of vertices and E a set of **unordered** pairs of distinct vertices called **edges**



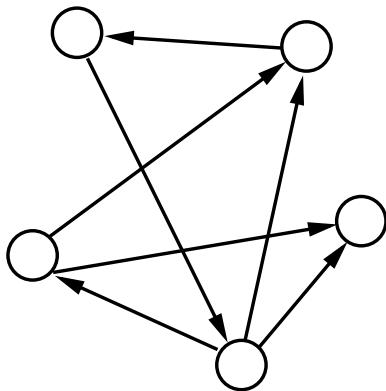
$$V = \{A, B, C, D, E\}$$

$$E = \{ (A,D), (A,E), (B,D), \\ (B,E), (C,D), (C,E) \}$$

Graphs

... when I say: "Directed Graph"

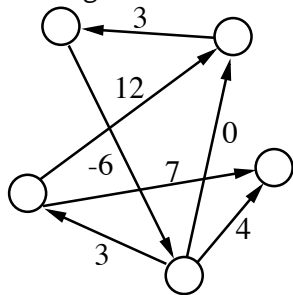
A **directed** graph (digraph) $G = (V, E)$ consists of V , a nonempty set of vertices and E a set of *ordered* pairs of distinct vertices called *edges*



Graphs

... when I say: "Weighted Graph"

A **weighted** graph is an undirected or directed graph with the additional property that each edge e has associated with it a real number $w(e)$ called its *weight*



Graphs

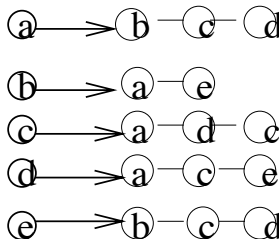
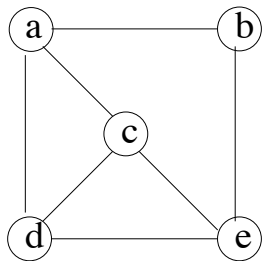
... when I say: “Representations of Graphs”

- ▶ Two common ways of representing a graph: **Adjacency list** and **adjacency matrix**
- ▶ Let $G = (V, E)$ be a graph with n vertices and m edges

Graphs

... when I say: "Adjacency List"

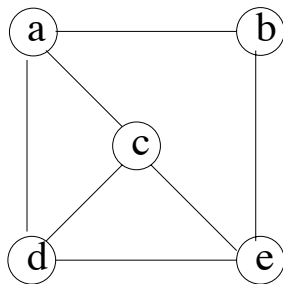
- ▶ For each vertex $v \in V$, store a list of vertices adjacent to v
- ▶ For weighted graphs, add information to each node
- ▶ How much is space required for storage?



Graphs

... when I say: "Adjacency Matrix"

- ▶ Use an $n \times n$ matrix M , where $M(i,j) = 1$ if (i,j) is an edge, 0 otherwise
- ▶ If G weighted, store weights in the matrix, using ∞ for non-edges
- ▶ How much is space required for storage?



	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	0	1
c	1	0	0	1	1
d	1	0	1	0	1
e	0	1	1	1	0

Algorithmic Techniques

... when I say: “Dynamic Programming”

- ▶ **Dynamic programming** is a technique for solving **optimization problems**, where we need to choose a “best” solution, as evaluated by an **objective function**
- ▶ **Key element:** Decompose a problem into **subproblems**, optimally solve them recursively, and then combine the solutions into a final (optimal) solution
- ▶ **Important component:** There are typically an exponential number of subproblems to solve, but many of them overlap
 - ⇒ Can re-use the solutions rather than re-solving them
- ▶ Number of distinct subproblems is polynomial
- ▶ Works for problems that have the **optimal substructure property**, in that an optimal solution is made up of optimal solutions to subproblems
 - ▶ Can find optimal solution if we consider all possible subproblems
- ▶ Example: All-pairs shortest paths

Algorithmic Techniques

... when I say: “Greedy Algorithms”

- ▶ Another optimization technique
- ▶ Similar to dynamic programming in that we examine subproblems, exploiting optimal substructure property
- ▶ **Key difference:** In dynamic programming we considered all possible subproblems
- ▶ In contrast, a greedy algorithm at each step commits to just one subproblem, which results in its **greedy choice** (locally optimal choice)
- ▶ Examples: Minimum spanning tree, single-source shortest paths

Algorithmic Techniques

... when I say: “Divide and Conquer”

- ▶ An algorithmic approach (not limited to optimization) that splits a problem into sub-problems, solves each sub-problem recursively, and then combines the solutions into a final solution
- ▶ E.g., Mergesort splits input array of size n into two arrays of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, sorts them, and merges the two sorted lists into a single sorted list in $O(n)$ time
 - ▶ Recursion bottoms out for $n = 1$
- ▶ Such algorithms often analyzed via recurrence relations

Conclusion

- ▶ This was a deliberately brief overview of concepts you should already know
- ▶ I expect you to understand it well during lectures, homeworks, and exams
- ▶ **It is all covered in depth in the textbook!**