Data Structures and Algorithms 2

Lecture 2: Review, Reminders, Practice!

Co-instructors: Robbie Hott and Ray Pettit

Spring 2024

Readings in CLRS 4th edition:

- CLRS Chapter 2: insertion sort (if needed), book's pseudocode conventions
- CLRS Chapter 3: info on order-class (more than we cover in lecture), math review
- Note: book goes into more depth than we do, and topics we don't need. Use it to reinforce what's taught in lectures.

Measuring Work: Reminders!

From your earlier courses, remember…

We want a measure of an algorithm’s work that is

- Independent of hardware, language, programmer, etc.
- Doesn’t require us to implement the algorithm
- Described in terms of input size(s)

We count some operation in our algorithm

- Some “basic operation” that’s fundamental to the class of problems
- Or, something in the innermost nested loop
- Or sometimes, an expensive operation

Often “basic operation” defines a class of algorithms we’re comparing

E.g. sorting of keys only using key-comparisons

Input Sizes: Reminders

The nature of an input affects how much work we do, so….

Worst-case W(n)

- Maximum number of basic operations for any input of size n
- We often need this upper-bound on performance
- We reason about one or more *worst-case inputs*

Average case A(n)

- Harder to calculate because you need
  - The amount of work T_I for every input I, and
  - The probability it occurs, P_I (Do we know? Can we assume?)
  - $A(n) = \sum T_I P_I$
- We probably won't do this kind of calculation in CS3100.
  But sometimes we may talk about *average* or *expected* complexity

Analyzing Algorithms <u>and Problems</u>

Sometimes we talk about **problems** and their properties

- Feasible or tractable problems
- Intractable problems
  - Problems that seem to need exponential time complexity, $\Theta(k^n)$ where k is a constant > 1.
  - Examples: the classes of NP-hard problems, NP-Complete problems
- Unsolvable problems (e.g. the Halting Problem)
- **Lower bound** for the number of operations needed to solve a problem
  - In other words, can we prove that it's **impossible for <u>any</u> algorithm** to solve this problem in fewer than some number of operations?

Asymptotic Analysis and Order Classes

Remember the Big Picture?

We use **order classes** to categorize an algorithm's complexity. Examples:

- Insertion sort is $\Theta(n^2)$ in the worst-case, but it's $\Theta(n)$ in the best-case
- Quicksort is $\Theta(n^2)$ in the worst-case, but mergesort is $\Theta(n \log n)$ in the worst-case
- Searching a balanced search tree is $\Theta(\log n)$ in the worst-case

An order-class like $\Theta(n^2)$ is a set of functions that grow at the same rate

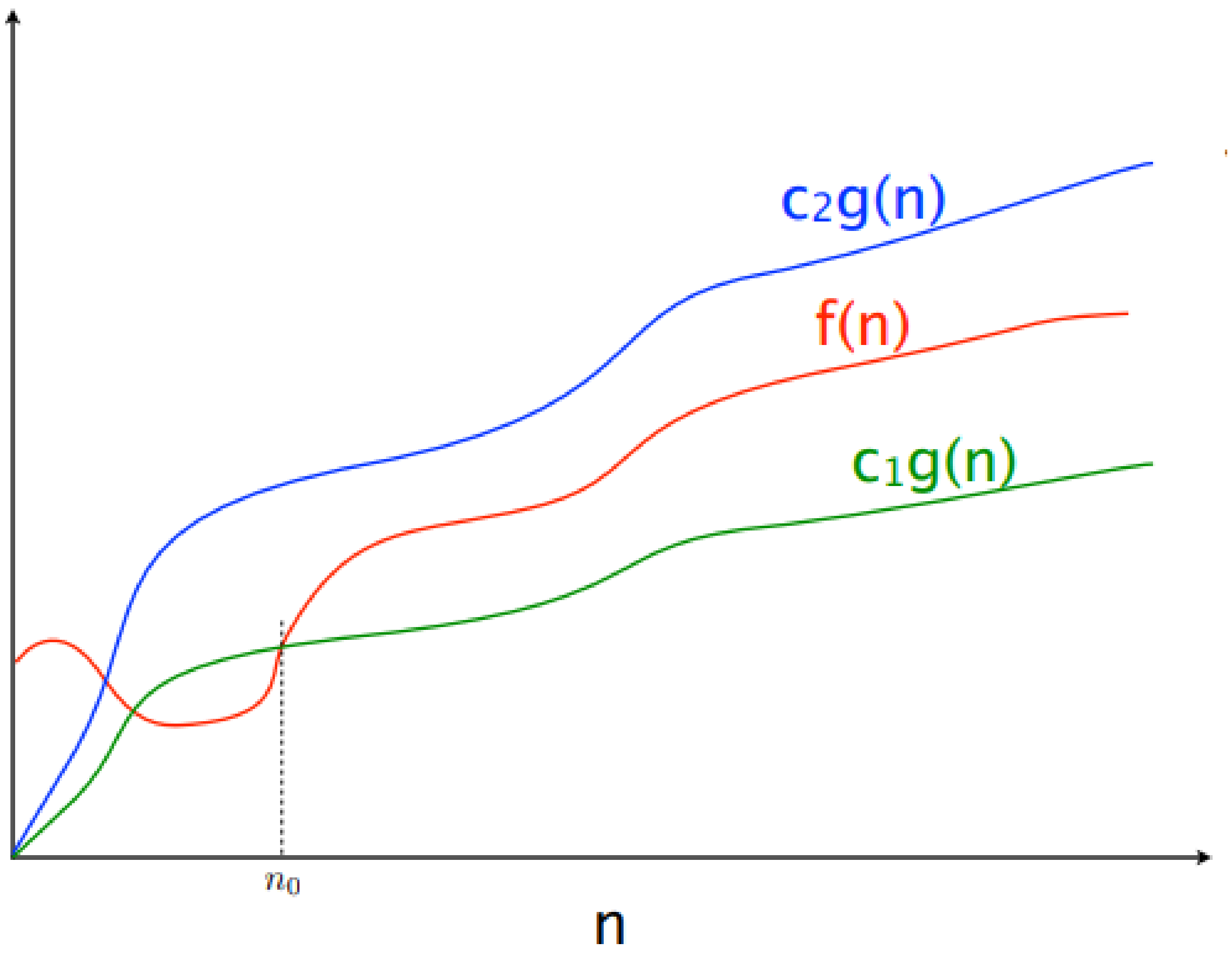
- Why is this a useful "label" to identify an algorithm's complexity?
- An analogy: Alex is an "A" student, which is a better category than someone who's a "B" student. And we're OK treating all "A" students as if they're equivalent in some way.
- In studying algorithms, being "equivalent" is about **asymptotic growth**

Asymptotic Bounds

The sets "big oh" O(g), "big theta" Θ(g), "big omega" Ω(g) – **Remember these meanings!**

- O(g): set of all functions that grow **no faster** than g, or **g is an asymptotic upper bound**
- Ω(g): set of all functions that grow **at least as fast** as g, or **g is an asymptotic lower bound**
- Θ(g): set of all functions that grow **at the same rate** as g, or **g is an asymptotic tight bound**

We'll see two mathematical ways to show some f(n) belongs to one of these sets



$$f(n) = O(g(n))$$

$$f(n) = \Theta(g(n))$$

$$f(n) = \Omega(g(n))$$

Again, we'll see two mathematical approaches to show some f(n) belongs to one of these sets

Asymptotic Notation*

Here's our first way to mathematically define these order classes

$\boldsymbol{O(g(n))}$

- At most within constant of g for large n
- $\{$functions $f \mid \exists$ constants $c, n_0 > 0$ s.t. $\forall n > n_0, f(n) \leq c \cdot g(n)\}$
- Set of functions that grow "in the same way" as <u>or</u> more *slowly* than g(n)

$\boldsymbol{\Omega(g(n))}$

- At least within constant of g for large n
- $\{$functions $f \mid \exists$ constants $c, n_0 > 0$ s.t. $\forall n > n_0, f(n) \geq c \cdot g(n)\}$
- Set of functions that grow "in the same way" as <u>or</u> more *quickly* than g(n)

$\boldsymbol{\Theta(g(n))}$

- "Tightly" within constant of g for large n
- $\Omega(g(n)) \cap O(g(n))$
- Set of functions that grow "in the same way" as g(n)

*CLRS Chapter 3

Asymptotic Notation Example

Show: $n \log n \in O(n^2)$

Asymptotic Notation Example

To Show: $n \log n \in O(n^2)$

- **Technique:** Find $c, n_0 > 0$ s.t. $\forall n > n_0, n \log n \le c \cdot n^2$
- **Proof:** Let $c = 1, n_0 = 1$. Then,
  $n_0 \log n_0 = (1) \log (1) = 0,$
  $c\, n_0^2 = 1 \cdot 1^2 = 1,$
  $0 \le 1.$

  $\forall n \ge 1, \log(n) < n \Rightarrow n \log n \le n^2$ □

Direct Proof!

Asymptotic Notation Example

Show: $n^2 \notin O(n)$

Asymptotic Notation Example

To Show: $n^2 \notin O(n)$

- **Technique: Contradiction**
- **Proof:** Assume $n^2 \in O(n)$. Then $\exists c, n_0 > 0$ s.t. $\forall n > n_0, n^2 \leq cn$
  Let us derive constant c. For all $n > n_0 > 0$, we know:
  $cn \geq n^2$,
  $c \geq n$.

  Since c is dependent on n, it is not a constant.
  Contradiction. Therefore $n^2 \notin O(n)$. □

Proof by Contradiction!

Proof Techniques

Direct Proof ✓

- From the assumptions and definitions, directly derive the statement

Proof by Contradiction ✓

- Assume the statement is true, then find a contradiction

Proof by Induction

Proof by Cases

More Asymptotic Notation

$\boldsymbol{o(g(n))}$

- Smaller than *any* constant factor of g for sufficiently large n
- {functions f : $\forall$ constants $c > 0$, $\exists n_0$ such that $\forall n > n_0, f(n) < c \cdot g(n)$}
- Set of functions that always grow more slowly than g(n)

Equivalently, ratio of $\frac{f(n)}{g(n)}$ is decreasing and tends towards 0:

$$f(n) \in o\big(g(n)\big) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Our 2nd way to mathematically define these classes, with a limit of a ratio

More Asymptotic Notation

$\boldsymbol{o(g(n))}$

- Smaller than *any* constant factor of g for sufficiently large n
- $\{$functions f : $\forall$ constants $c > 0, \exists n_0$ such that $\forall n > n_0, f(n) < c \cdot g(n)\}$
- Set of functions that always grow more slowly than g(n)

$\boldsymbol{\omega(g(n))}$

- Greater than *any* constant factor of g for large n
- $\{$functions f : $\forall$ constants $c > 0, \exists n_0$ such that $\forall n > n_0, f(n) > c \cdot g(n)\}$
- Set of functions that always grow more quickly than g(n)

Equivalently, $f(n) \in \omega\big(g(n)\big) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

A Way to Think about Order Classes

For “comparables”, we have 5 logical operators: $= \; < \; \leq \; > \; \geq$

How are the order classes we’ve defined like these?

Another Asymptotic Notation Example

Direct Proof

Show: $n \log n \in o(n^2)$

Proof Technique: Show the statement directly, using either definition

- For every constant $c > 0$, we can find an n_0 such that $\frac{\log n_0}{n_0} = c$. Then for all $n > n_0$, $n \log n < c\, n^2$ since $\frac{\log n}{n}$ is a decreasing function

$\forall$ constants $c > 0$, $\exists n_0$ such that $\forall n > n_0, f(n) < c \cdot g(n)$

Equivalently, $\lim_{n \to \infty} \frac{n \log n}{n^2} = \lim_{n \to \infty} \frac{\log n}{n} = 0$ (why is this true?)

Summary: Using Limit Definition

Want to prove f(n) belongs to some order class of g(n)?

Calculate this:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

If the result is….

$< \infty$, including the case in which the limit is 0, then $f \in O(g)$

> 0, including the case in which the limit is ∞, then $f \in \Omega(g)$

$= c$ and $0 < c < \infty$ then $f \in \Theta(g)$

$= 0$ then $f \in o(g)$

$= \infty$ then $f \in \omega(g)$

Math Reminders

Review Section 3.3 in CLRS (3.2 in 3rd edition) for some math we'll use:

- Polynomials
- Exponentials
  - $n^b \in o(\alpha^n)$ for any constant $\alpha > 1$
- Logarithms
  - Changing base means multiplying by a constant, so logs of any base are in the same order class $\theta(\log n)$
  - $\log n \in o(n^\alpha)$ for any constant $\alpha > 0$
- Factorials
  - Note: $n! \in o(n^n)$, $n! \in \omega(2^n)$, and $\log n! \in \theta(n \log n)$
- Functional iteration: $f^{(i)}(n)$

If You Ever See A Series Like These…

Arithmetic series

- The sum of consecutive integers:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Geometric series:

- For real $x \neq 1$

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

Polynomial Series

- The sum of squares: $\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6} \approx \frac{n^3}{3}$
- The general case is: $\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1}$
- Powers of 2: $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

Arithmetic-Geometric Series:

$$\sum_{i=1}^{k} i2^i = (k-1)2^{k+1} + 2$$

Algorithms You’ve Studied

Remember Searching?

Problem: Given a list and a target, return the location of the target in the list, or a sentinel value if not found

Sequential or Linear Search

- No assumptions on the order of values in the list
- Compares the target to keys of the list-items
- Always Θ(n). In the worst-case, *n* comparisons. On average, *n/2*

Binary Search

- List must be sorted
- Θ(log n) in the worst-case

Reminder: balanced search trees are also Θ(log n) in the worst-case

Remember Quadratic Sorts?

There are a number of comparison sorts that are $\Theta(n^2)$ in the worst-case:

Insertion sort, Selection sort, Bubble sort,…

Insertion sort

- $\Theta(n^2)$ in the worst-case, but it's $\Theta(n)$ in the best-case
    - If list is almost sorted, performance is close to linear
- It's *in-place* (extra storage is constant in size)
- For more info, CLRS Section 2.1 or other sources

Remember Mergesort?

A divide and conquer algorithm, usually implemented recursively on smaller sublists:

1. If a sublist is size 0 or 1, do nothing (it's already sorted)
   Otherwise:
2. Divide the (sub)list into two equal smaller sublists
3. Sort each of those recursively
4. Use a merge algorithm to combine the two sorted sublists

Note:

- Mergesort is $\Theta(n \log n)$ in the worst-case
- It's not in-place (we need $\Theta(n)$ extra storage for the merge)
- CLRS Section 2.3.1 is about mergesort, and there are other good sources

Remember Quicksort?

Also a divide and conquer algorithm, usually implemented recursively on smaller sublists:

1. If a sublist is size 0 or 1, do nothing (it's already sorted)
   Otherwise:
2. Use a **partition algorithm** to place some "pivot" value into it's correct position, that also makes sure items found before the pivot are smaller, and those after the pivot are larger
3. Sort the sublists on either side of the pivot recursively

Note:

- Quicksort is $\Theta(n \log n)$ in the best- and average-cases
- Could be $\Theta(n^2)$ in the worst-case but this can be avoided
- It's in-place (except for the stack needed for recursive calls)
- CLRS Chapter 7 is about quicksort, and there are other good sources

Remember Lower Bounds Proof for Sorting?

In CS2100, you saw a lower bounds proof that showed:
Any comparison sort has time-complexity of $\Omega(n \log n)$

Recall for a lower-bound proof, we make a logical argument about the problem itself, one that holds for <u>any</u> algorithm that solves the problem

Proof used a decision tree (next slide), which models how any sort must work:

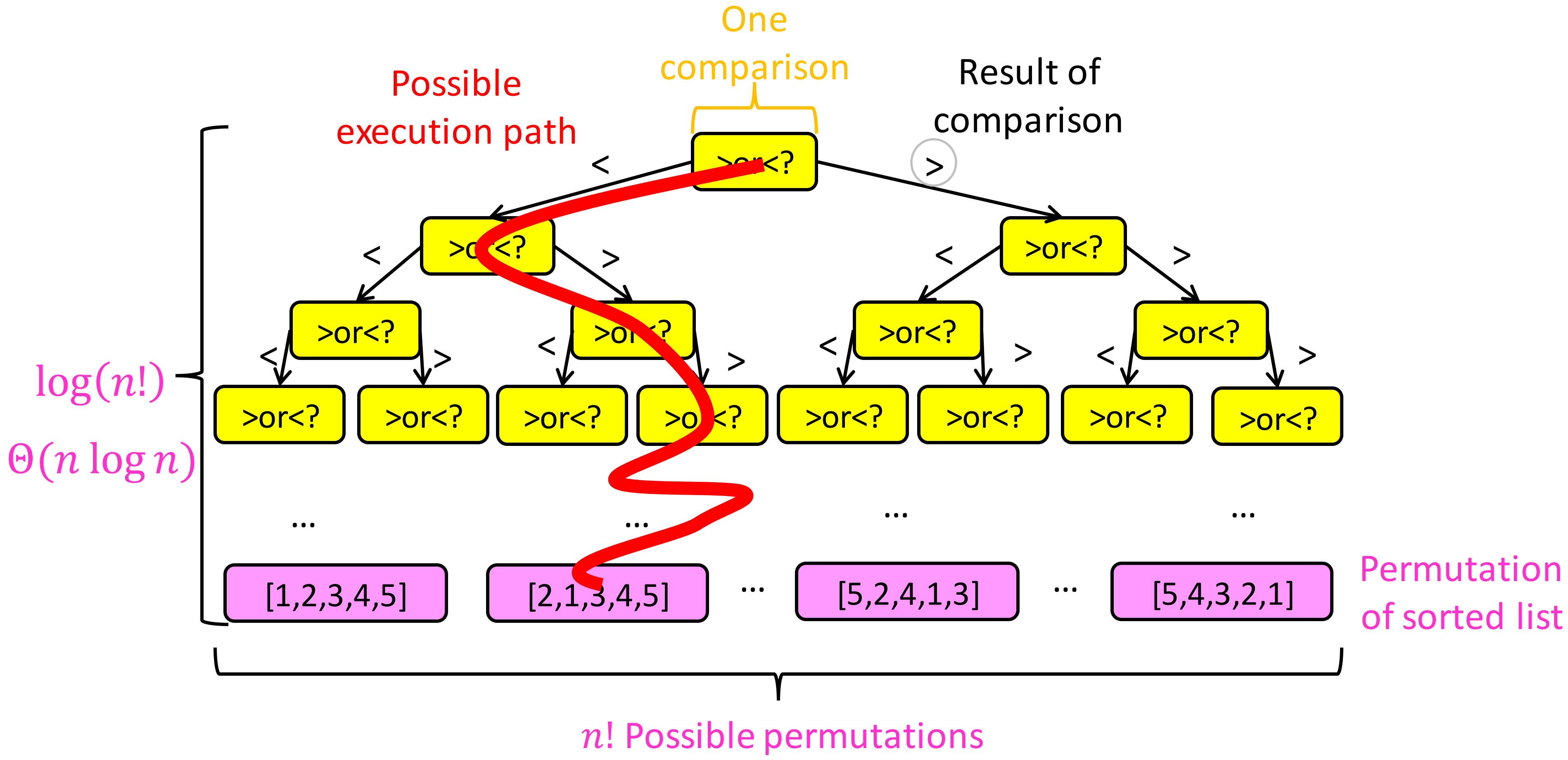
- Internal nodes represented a comparison between two keys
- Leaf nodes were permutations of list items (n! leaves)
- A correct sorting algorithm must trace a path through the tree to each of the leaves
- What's the longest path? The height of the tree.
- So how short can a tree be with n! leaves?

Decision Tree Argument

Worst case run time is the longest execution path

i.e., “height” of the decision tree

CLRS Section 8.1 describes this in more detail



What's Next?

Now that you've had some review

- Practice problems
- On to Graphs next lecture!