CS4102 Algorithms

Spring 2022

Warm up

What’s the sum? – Gauss

$$1 + 2 + 3 + \cdots + 98 + 99 + 100$$

Simplify:

$$1 + 2 + 3 + \cdots + (n - 1) + n =$$

$$1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$$

Announcements

- Homework 1A Basic is out!
- Prof. Hott’s Office hours
  - Mondays 3-5pm (4-5pm primarily 4102)
  - Fridays 12-2pm (1-2pm primarily 4102)
- Horton’s office hours: See course website
- Policy change: max size of collaboration groups is 4 total

Topics in first part of this slide-deck:

- Some of this material is from CLRS, Chapter 2
- Goals for this lecture:
  - Review the sorting problem and some “basic” algorithms, while using this to review (or introduce) some principles of algorithm analysis
- Topics:
  - The sorting problem
  - Insertion Sort
    - Including a lower-bounds proof
  - Mergesort
    - Including an overview of Divide and Conquer
  - Solving recurrences: tree method and the unrolling method

Sorting a Sequence: Defining the Problem

- The problem:
  - Given a sequence of items $\boldsymbol{a_0}$ … $\boldsymbol{an}$
    reorder it into a permutation $\boldsymbol{a'_0}$ … $\boldsymbol{a'_n}$
    such that $\boldsymbol{a'_i} <= \boldsymbol{a'_{i+1}}$ for all pairs
    - Specifically, this is sorting in *non-descending order*…
- We'll mostly focus on a restricted form of this problem:
  "Sorting using comparison of keys"
  - The **basic operation** we'll count in our analysis will be a comparison of two items' key-values. Why?
    - General: can sort anything
    - Controls decisions, so total operations often proportional
    - Can be an expensive operation (e.g. when keys are large strings)

Some Observations

- We assume non-descending order for simplicity
  - Our analysis results apply for other orderings
  - You know a comparison-function can be used in practice (e.g. Java's Comparable interface)
- In analyzing a problem and algorithms that solve it, sometimes it's important to define assumptions like what we're counting, i.e. the basic operation here
  - Example: *binary search* is an optimal algorithm for searching using key comparisons, but *hashing* can be faster in practice.
- Swapping items is often expensive
    - We can apply same techniques to count swapping, as a separate analysis

Sorting: More Terminology

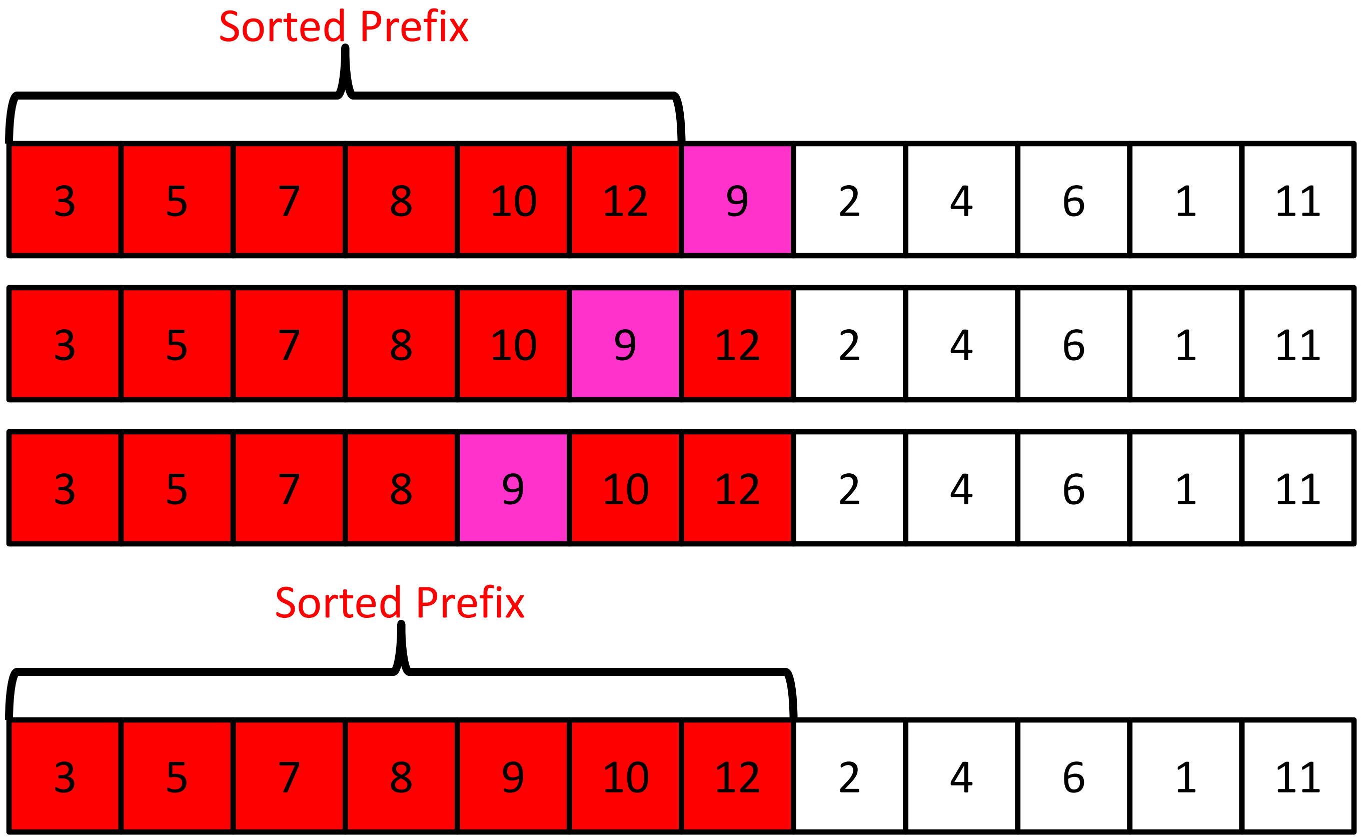
- **Comparison Sorts:** only compare keys and move items
- **Adjacent Sort:** Algorithms that sort by only swapping adjacent elements
  - e.g., bubble sort and insertion sort
  - ...these are a subset of comparison sorts.
- **Stable Sort:** A sorting algorithm is stable
  - when two items x and y occur in the relative order x,y in the original list AND x==y, then x and y appear in the same relative order x,y in the final sorted list.
  - Why would we want this?
- **In-Place Algorithm:** the algorithm uses at most $\Theta(1)$ extra space
  - e.g., allocating another array of size $\Theta(n)$ is NOT allowed.

Why Do We Study Sorting?

- An important problem, often needed
  - Often users want items in some order
  - Required to make many other algorithms work well.
    - Example: To use binary search, sequence must be sorted first. The search algorithm is optimal and requires $\Theta(\log n)$ comparisons.
- And, for the study of algorithms…
  - A history of solutions
  - Illustrates various design strategies and data structures
  - Illustrates analysis methods
  - Illustrates how we prove something about optimality for this problem

Insertion Sort

Idea: Maintain a sorted list prefix, extend that prefix by "inserting" the next element



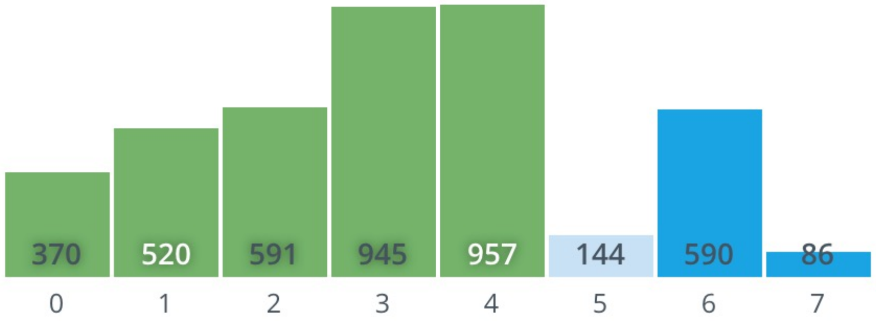
Sorted Prefix

3	5	7	8	10	12	9	2	4	6	1	11
3	5	7	8	10	9	12	2	4	6	1	11
3	5	7	8	9	10	12	2	4	6	1	11

Sorted Prefix

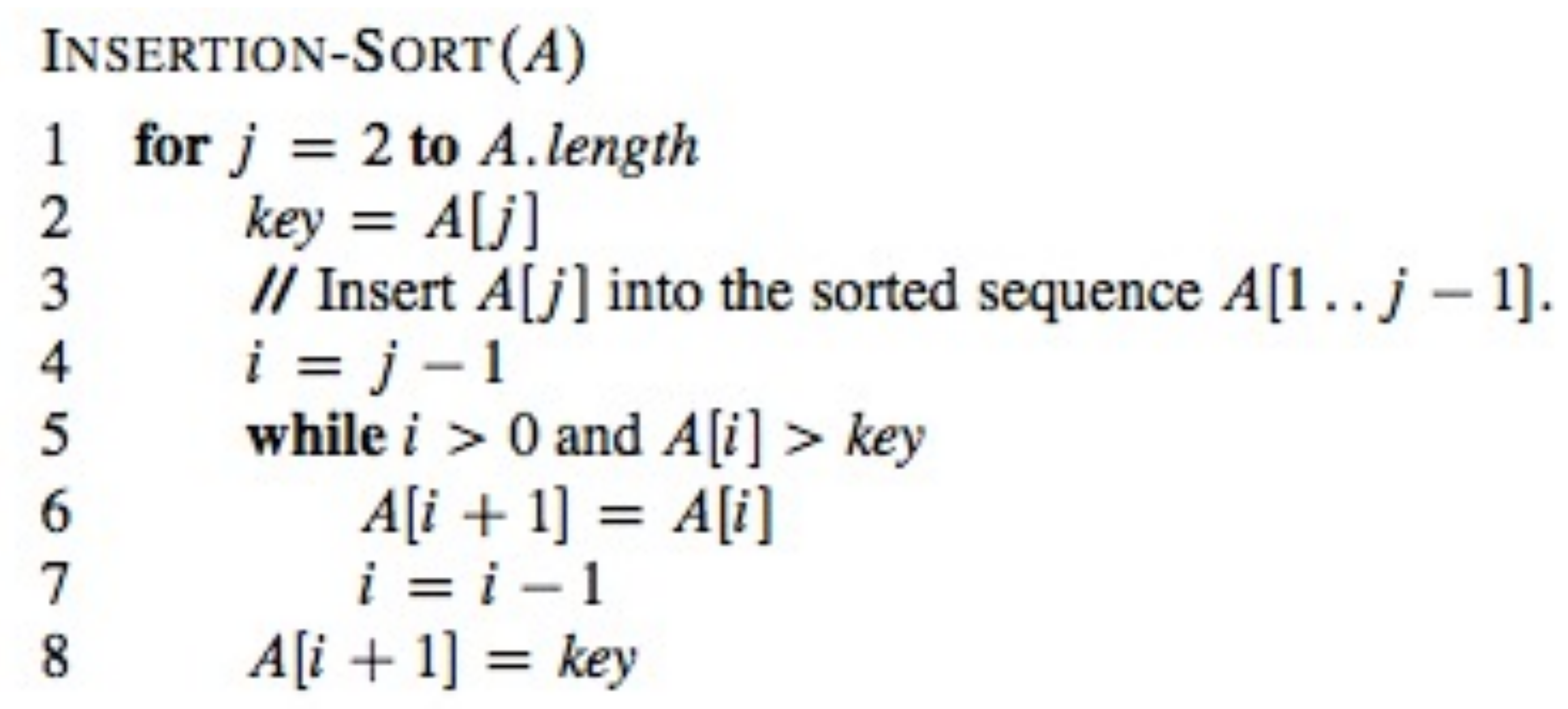
3	5	7	8	9	10	12	2	4	6	1	11

Insertion Sort



- The strategy:
  1. First section of list is sorted (say $i - 1$ items)
  2. Increase this partial solution by…
     1. Shifting down the next item beyond sorted section (i.e. the i^{th} item) down to its proper place in sorted section.
     2. Note: must shift items up to make room – **Adjacent Sort!**
  3. Start at $i\ =\ 2$, since one item alone is already sorted.
- Note: Example of general strategy
  - – Extend a partial solution by increasing its size by one.
  - – Some call this: *decrease and conquer*

Insertion Sort: Pseudocode



```
INSERTION-SORT($A$)
for $j = 2$ to $A.length$
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1 \mathinner{\ldotp\ldotp} j-1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

Note: *CLRS pseudocode indexes lists from 1, not from 0.*

Properties of Insertion Sort

- We could have talked about bubble sort, selection sort,…
- Why are we focusing on Insertion Sort?
    - Easy to code
    - In-place
    - What's it like if the list is sorted?
        - Or almost sorted?
    - Fine for small inputs.   Why?
    - Is it stable? Why?

Insertion Sort: Analysis

- Worst-Case: $W(n) = \sum_{j=2}^{n} (j-1) = n(n-1)/2 = \Theta(n^2)$

- Best-case behavior? One comparison each time

$$B(n) = \sum_{j=2}^{n} 1 = n-1$$

- Average Behavior
  - – We won't do the math for that, but it's about $n^2/4$

Insertion Sort: Best of a breed?

- We know that I.S. is one of many quadratic sort algorithms, and that log-linear sorts (i.e., $\Theta(n \lg n)$) do exist
- Can we learn something about I.S. that tells us what it is about I.S. that "keeps it" in the slower class?
  - – Yes, by a *lower-bounds argument* for adjacent sort algorithms
  - – This is our first example about how to make *lower-bounds arguments* about <u>a problem</u>
    - E.g. "it's impossible for any algorithm to solve this problem in better than…."
  - – We'll show that sorting a list by **only swapping adjacent elements** is $\boldsymbol{\Omega(n^2)}$ and can never be $\boldsymbol{o(n^2)}$

Removing Inversions

- Define an *inversion* in a sequence: A pair of elements that are out of order
  - E.g. [2, 4, 1, 5, 3] not sorted and has 4 inversions:
    - Pairs: (2,1) (4,1) (4,3) (5,3)
  - *Any correct sort* must fix each of these inversions
    - Ex: they must at some point swap 4 and 1
  - What's the maximum possible number of inversions?

    $n(n-1)/2$ *all possible pairs*
    - This really can occur, e.g. [5, 4, 3, 2, 1]

Removing Inversions – Lower Bound

- Consider an adjacent sort algorithm
  - Reminder: sorts by only swapping adjacent elements
- Each adjacent swap can only remove at most one inversion!
- There are $n(n-1)/2$ inversions, e.g., [5, 4, 3, 2, 1]

Theorem: Any algorithm that sorts by comparison of keys and removes at most one inversion after each comparison must do at least $n(n-1)/2$ comparisons in the worst case.

Lower Bound and Insertion Sort

- Insertion Sort *only swaps adjacent elements*
    - Each “new” element compared with element to left (”slide in”)
    - Insertion sort *only* removes at most one inversion for each key comparison it does
    - Insertion sort must do at least $\frac{n(n-1)}{2} = O(n^2)$ comparisons
- **Therefore:** Insertion Sort is optimal for the set of algorithms that only swap adjacent elements.
    - I.e. adjacent sorts

Lower Bound is General

- **Important:** we just proved a time-complexity result about **the problem** that applies to <u>any</u> algorithm of this type that solves it!
  - Lower bounds proofs are about **the problem**, and can be used to show an algorithm is optimal (or close to optimal)
- Meaning: for any algorithm to be $o(n^2)$, it must swap elements that are not adjacent!

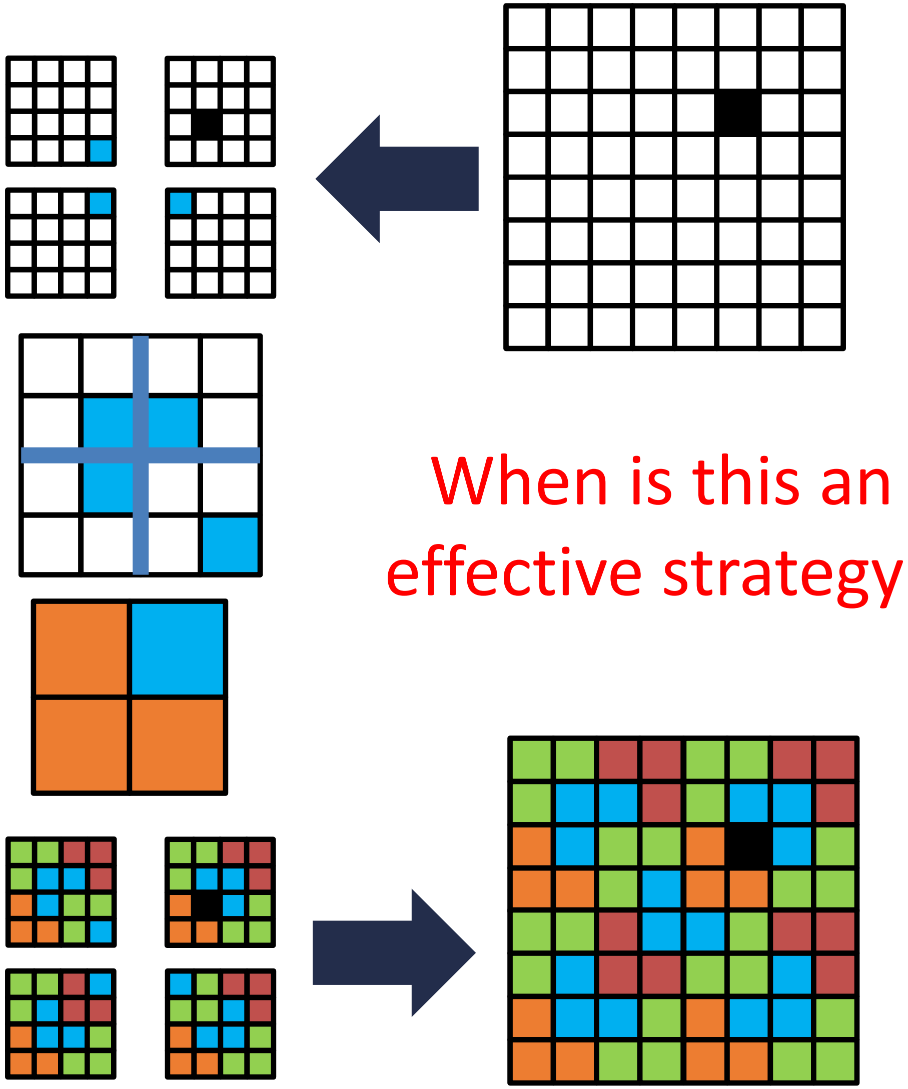
Mergesort Overview

- General and practical sorting algorithm
- Good example of a **divide-and-conquer** algorithm
  - Recursion leads to a more efficient solution in the worst-case than adjacent sorts
  - It's $o(n^2)$ or $\Theta(n \lg n)$ to be more precise

Divide and Conquer

[CLRS Chapter 4]

- **Divide**:
  - Break the problem into multiple subproblems, each smaller instances of the original
- **Conquer**:
  - If the suproblems are "large":
    - Solve each subproblem recursively
  - If the subproblems are "small":
    - Solve them directly (base case)
- **Combine**:
  - Merge solutions to subproblems to obtain solution for original problem



When is this an effective strategy?

Generic Divide and Conquer Solution

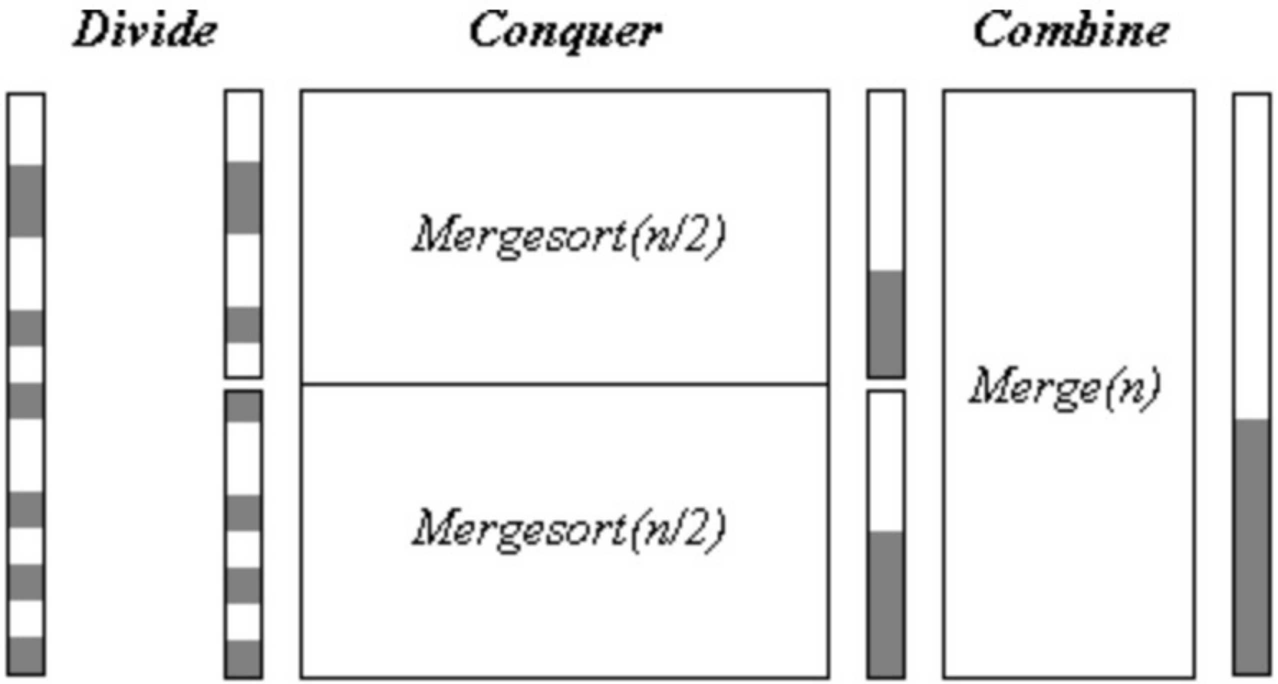
```
def myDCalgo(problem):
    if baseCase(problem):
        solution = solve(problem) #brute force if necessary
        return solution
    subproblems = Divide(problem)
    for sub in subproblems:
        subsolutions.append(myDCalgo(sub))
    solution = Combine(subsolutions)
    return solution
```

Merge Sort: Divide and Conquer Sorting

- **Divide**:
  - Break n-element list into two lists of $^{n}/_{2}$ elements
- **Conquer**:
  - If $n > 1$:
    - Sort each sublist recursively
  - If $n = 1$:
    - List is already sorted (base case)
- **Combine**:
  - Merge together sorted sublists into one sorted list

A Visualization

- Note: in this visualization, think of the gray regions being larger values that should get sorted to the end (the bottom).



Merge

- **Combine**: Merge sorted sublists into one sorted list
- We have:
  - 2 sorted lists (L_1, L_2)
  - 1 output list (L_{out})

```
While ($L_1$ and $L_2$ not empty):
    If $L_1[0] \leq L_2[0]$:
        $L_{out}$.append($L_1$.pop())
    Else:
        $L_{out}$.append($L_2$.pop())
$L_{out}$.append($L_1$)
$L_{out}$.append($L_2$)
```

$O(n)$

Analyzing Divide and Conquer

1. Break into smaller subproblems
   - Define smaller subproblems, how to divide and combine their results
2. Use recurrence relation to express recursive running time
   - **Divide:** $D(n)$ time,
   - **Conquer:** recurse on small problems, size s
   - **Combine:** $C(n)$ time
   - **Recurrence:**

$$T(n) = D(n) + \sum T(s) + C(n)$$

3. Use asymptotic notation to simplify

Analyzing Merge Sort

1. Break into smaller subproblems
2. Use recurrence relation to express recursive running time
3. Use asymptotic notation to simplify

Divide: 0 comparisons

Conquer: recurse on 2 small subproblems, size $\frac{n}{2}$

Combine: n comparisons

Recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \in \Theta(\mathrm{n} \log \mathrm{n})$$

Let's see why!

Recurrence Solving Techniques

Four methods for solving recurrences

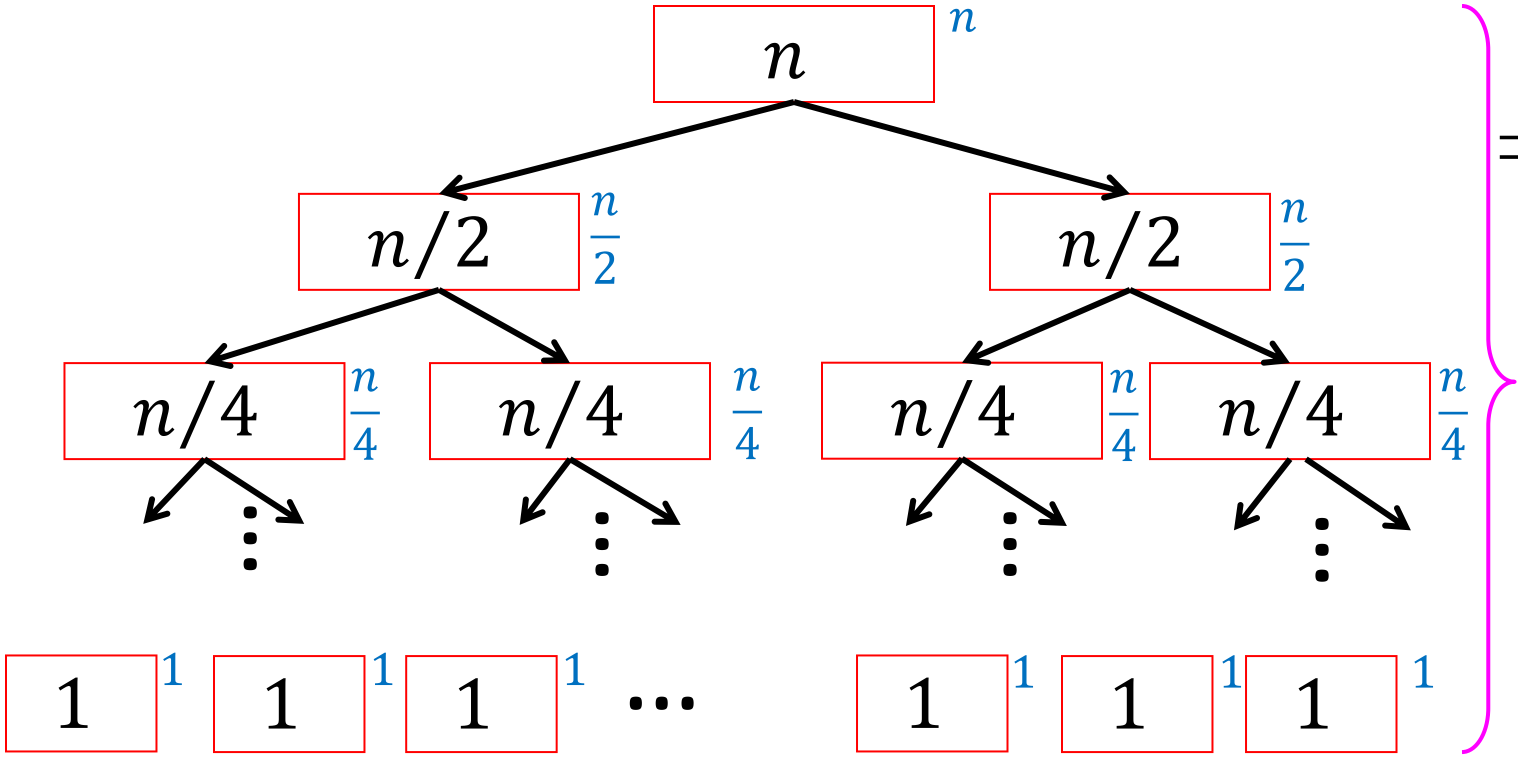
- Unrolling: expand the recurrence

- Tree: get a picture of recursion
- ❓✓ Guess/Check: Substitution by guessing the solution and using induction to prove

- “Cookbook”: Use magic (a.k.a. Master Theorem)

Tree method

$$T(n) = 2T(\frac{n}{2}) + n$$



n $\quad n$

$n/2$ $\quad \frac{n}{2}$ $\qquad n/2$ $\quad \frac{n}{2}$

$n/4$ $\quad \frac{n}{4}$ $\qquad n/4$ $\quad \frac{n}{4}$ $\qquad n/4$ $\quad \frac{n}{4}$ $\qquad n/4$ $\quad \frac{n}{4}$

1 $\,^1$ $\quad 1$ $\,^1$ $\quad 1$ $\,^1$ $\quad \cdots \quad 1$ $\,^1$ $\quad 1$ $\,^1$ $\quad 1$ $\,^1$

$\Rightarrow$ n total / level

$\log_2 n$ levels of recursion

$$T(n) = \sum_{i=1}^{\log_2 n} n = n \log_2 n$$

Unrolling

The strategy:

1. Replace the recursive calculation for the smaller value with what you get if you "plug in" the smaller value into the original recurrence
2. Repeat and describe the general pattern
3. Use the base case to find how many repetitions
4. Finally plug in base case value and simplify

Let's do this for Mergesort's recurrence!

Unrolling, Step 1

1. Replace the recursive calculation for the smaller value with what you get if you “plug in” the smaller value into the original recurrence

$$T(n) = 2T\left({}^{n}/_{2}\right) + n \longrightarrow T\left({}^{n}/_{2}\right) = 2T\left({}^{n}/_{4}\right) + {}^{n}/_{2}$$

$$= 2[???] + n$$

$$= 2\left[2T\left({}^{n}/_{4}\right) + {}^{n}/_{2}\right] + n = 4T\left({}^{n}/_{4}\right) + 2n$$

Unrolling, Step 2: Repeating

1. Replace the recursive calculation for the smaller value with what you get if you "plug in" the smaller value into the original recurrence
2. Repeat and describe the general pattern

$$T(n) = 2T\left({}^{n}/_{2}\right) + n \quad\longrightarrow\quad T\left({}^{n}/_{4}\right) = 2T\left({}^{n}/_{8}\right) + {}^{n}/_{4}$$

$$= 2\left[2T\left({}^{n}/_{4}\right) + {}^{n}/_{2}\right] + n \quad = 4T\left({}^{n}/_{4}\right) + 2n$$

$$= 4\left[2T\left({}^{n}/_{8}\right) + {}^{n}/_{4}\right] + 2n = 8T\left({}^{n}/_{8}\right) + 3n$$

…

Unrolling, Step 2: General Pattern

1. Replace the recursive calculation for the smaller value with what you get if you "plug in" the smaller value into the original recurrence
2. Repeat and <u>describe the general pattern</u>

Let i count repetitions

$$T(n) = 2T\left({}^{n}/_{2}\right) + n \qquad \boxed{i = 1}$$

$$= 2\left[2T\left({}^{n}/_{4}\right) + {}^{n}/_{2}\right] + n \quad = 4T\left({}^{n}/_{4}\right) + 2n \qquad \boxed{i = 2}$$

$$= 4\left[2T\left({}^{n}/_{8}\right) + {}^{n}/_{4}\right] + 2n = 8T\left({}^{n}/_{8}\right) + 3n \qquad \boxed{i = 3}$$

…

$$= 2^{i}\,T\left({}^{n}/_{2^{i}}\right) + i\,n$$

Unrolling, Steps 3 and 4: General Pattern

3. Use the base case to find how many repetitions
4. Finally plug in base case value and simplify

$$T(n) = 2T\left({}^{n}/_{2}\right) + n$$

$$= 2^i\, T\left({}^{n}/_{2^i}\right) + i\, n$$

Let i count repetitions

When will we stop repeating this? At the base case: $T(1) = 0$

When $T\left({}^{n}/_{2^i}\right)$ is $T(1)$ then ${}^{n}/_{2^i} = 1$ or $i = log_2 n$

$$2^i\, T\left({}^{n}/_{2^i}\right) + i\, n = 2^{lg\, n}\, T\left({}^{n}/_{2^{lg\, n}}\right) + (\lg n)\ n$$

$$= n\, T(1) + n \lg n = n \lg n$$

We have the closed form of the recurrence!

Conclusion

Using the unrolling technique, we've found the closed form for:

Mergesort's Recurrence:

$$T(1) = 0,\ T(n) = 2T\left(\frac{n}{2}\right) + n = n\ log\ n$$

Mergesort is a log-linear sort, $\Theta(\mathbf{n\ log\ n})$

Things to think about:

- Would the Θ order class change if
  - T(1) was a non-zero constant
  - The cost to combine was not exactly n but still $\Theta(n)$
- Practice unrolling on the cheer-for-pizza recurrence!

Where we are: We’ve used sorting to…

- See again how to apply ideas of counting operations
  - For insertion sort, we’ve discussed: worst, average, best case
- See two different strategies for the same problem
  - Insertion sort: “decrease and conquer”
  - Mergesort: divide and conquer
  - Introduced some new concepts: in-place, stable
- Prove a lower-bound that shows
  - One class of algorithms has a lower bound of $\Omega(n^2)$
  - To do better, must remove >1 inversion for each comparison
- Methods for solving recurrences
  - Seen the tree method
  - Learned the unrolling method