Sorting and Some Algorithm Principles

CS 4102, Algorithms

Prof. Floryan and Prof. Horton

Spring 2021

Topics

Topics in first part of this slide-deck:

- Readings: 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

Sorting Introduction

Sorting a Sequence: Defining the Problem

- The problem:
  - Given a sequence of items $\mathbf{a_0 \ldots a_n}$
    reorder it into a permutation $\mathbf{a'_0 \ldots a'_n}$
    such that $\mathbf{a'_i <= a'_{i+1}}$ 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 constraints like the basic operation
  - 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

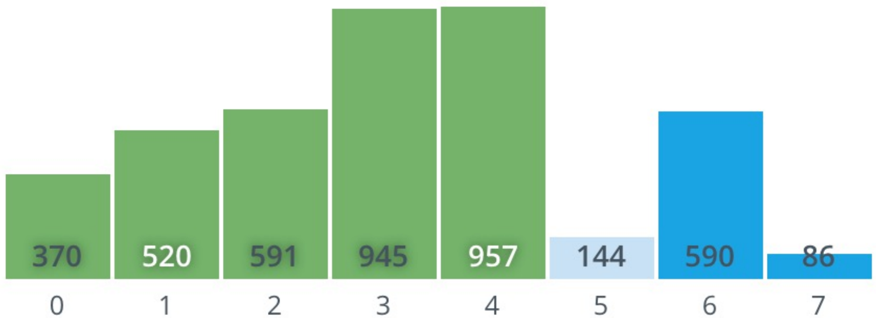
- **Comparison Sorts:** only compare keys and move items
- **Adjacent Sort:** Algorithms that sort by only swapping adjacent elements
  - e.g., bubble sort and insertion sort
  - ...these are a subset of comparison sorts.
- **Stable Sort:** A sorting algorithm is stable
  - when two items x and y occur 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 Sort:** 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

Insertion Sort



- The strategy:
  1. First section of list is sorted (say i-1 items)
  2. Increase this partial solution by…
  3. Shifting down next item beyond sorted section (i.e. the i^{th} item) down to its proper place in sorted section. (Must shift items up to make room.)
  4. Since one item alone is already sorted, we can put steps 1-3 in a loop going from the 2nd to the last item.
- 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 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

An Aside: Proving it right with Loop Invariants

- An important technique to prove algorithm correctness. (See CLRS or even Wikipedia.)
- Properties that hold true at these points:
  - Prior to first iteration (initialization)
  - If true before an iteration, then true after that iteration (maintenance)
  - When loop ends, properties still hold and tell us something useful about correctness (termination)
- Loop invariant for Insertion Sort:
  - For the for-loop governed by index j, the values A[0..j-1] are the elements originally stored in the sub-list but in sorted order

Properties of Insertion Sort

- We could have talked about bubble sort, selection sort,…
- Why Insertion Sort here?
  - 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: $$W(n) = \sum_{j=2}^{n}(j-1) = n(n-1)/2 = \Theta(n^2)$$
- Average Behavior
  - Average number of comparisons in inner-loop?

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

    - So for the j^{th} element, we do roughly j/2 comparisons
  - To calculate A(n), we note j goes from 2 to n

    $$A(n) = \sum_{j=2}^{n}\left(\frac{j}{2} + \frac{1}{2} - \frac{1}{j}\right) = \frac{n^2}{4} + \frac{3n}{4} - 1 - \sum_{j=2}^{n}\frac{1}{j} \approx \frac{n^2}{4}$$

- Best-case behavior? One comparison each time

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

Lower Bounds Proof for Adjacent Sorts

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. Θ(n lg n)) do exist
- But, 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 you how to make *lower-bounds arguments* about a problem
    - 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 $\mathbf{\Omega(n^2)}$ and can never be $\mathbf{o(n^2)}$
  - **We'll do this proof "live" session" in lecture!**

Mergesort and Divide and Conquer

Mergesort Overview

- General and practical sorting algorithm
- Good example of a **divide-and-conquer** algorithm
  - More on what that means next
  - 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 Strategy

- A divide-and conquer algorithm usually has the following structure:

```
solveProblem(input)
  if input is small, then solve directly (brute-force?)
  else if input is big
      divide problem into n smaller problems
      recursively invoke solveProblem on smaller problems
      combine solutions to small problems into bigger solution
      return bigger solution
```

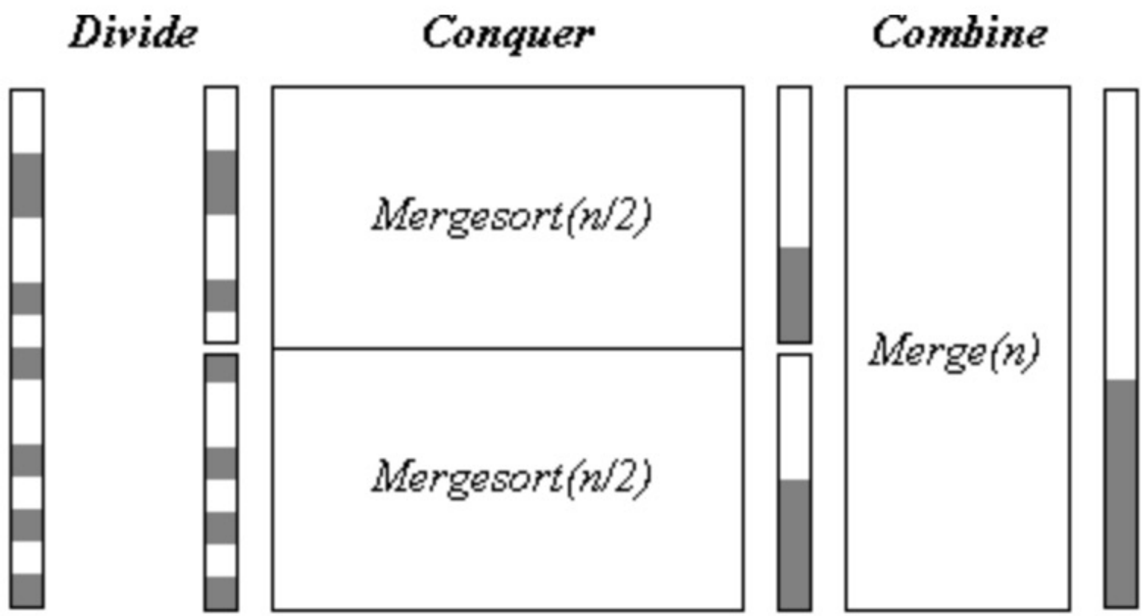
- Note: maybe solve all the smaller problem, or maybe just some of them.
- Runtime is sum of the times to divide, recursively solve, and combine

Mergesort and Divide and Conquer

- Base case:
  - Sublist is size 1. Already sorted!
- Divide:
  - Divide list into two sublists of equal size.
- Conquer:
  - Call mergesort recursively on each sublist.
  - Gives us a sorted left and right sublist.
- Combine:
  - Merge the sorted left and right sublists to get one larger sorted list.

Picture

- Note: in this diagram, think of the colored regions being small values that should get sorted to the front.



Mergesort code

- Specification:
  - Input: List *lst* and indexes *first*, and *last*, such that all elements lst[i] are defined for first <= i <= last.
  - Output: lst[first], …, lst[last] is sorted rearrangement of the same elements

```
def mergesort(lst, first, last):
    if first < last:
        mid = (first+last) // 2
        mergesort(lst, first, mid)
        mergesort(lst, mid+1, last)
        merge(lst, first, mid, last)
    return
```

  - Wait, where's the actual work happening?
  - Why do we need the 2nd and 3rd parameters? Wait for live session!

Merge: Pseudocode

- Most of the work done in merge
  - Comparisons, moves
  - Most implementations use a "scratch array"
    - An extra array of size n which is then copied back into
- The Problem:
  - Given two sorted sequences A and B, merge them to create one sorted sequence C
- Strategy:
  1. C is initially empty.
  2. Look at the first (current) items in A and B.
  3. The smallest of these should become the first (next) item in C
  4. Move that item to the end of C.
  5. You need to now compare the next item in that list to the current item in the other. Essentially, go to Step 2.
  6. When you've moved all items in one list, move the items in the other to the end.
- Time complexity of merge is linear, $\Theta(n)$

Mergesort Analysis

- What is the runtime $T(n)$? Add up the costs!
  - Divide the list: constant, **$\Theta(1)$**
  - Two recursive sorts: each costs **$T(n/2)$**
  - Merge: linear, **n** or close to it, so **$\Theta(n)$**
- Overall it's better than adjacent sorts!

$$T(n) = 2T(n/2) + n \in \Theta(n \log(n))$$

  - Uhhhhh...why is it that order class?
- Upcoming lectures and Chapter 4 of CLRS is all about "solving" *recurrence relations*
  - Getting a closed-form solution to a recursive formula

Summary

Where we are: We've used sorting to…

- See again how to apply ideas of counting operations
  - Including: 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 (well, in the live session)
  - One class of algorithms has a lower bound of $\Omega(n^2)$
  - To do better, must remove >1 inversion for each comparison
- Reason about algorithms and problems
  - Cost measures for an algorithm
  - Correctness: loop invariants
  - Lower-bound proof for a <u>problem</u> and <u>class</u> of algorithms