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
- Second part of this slide-deck: Quicksort

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

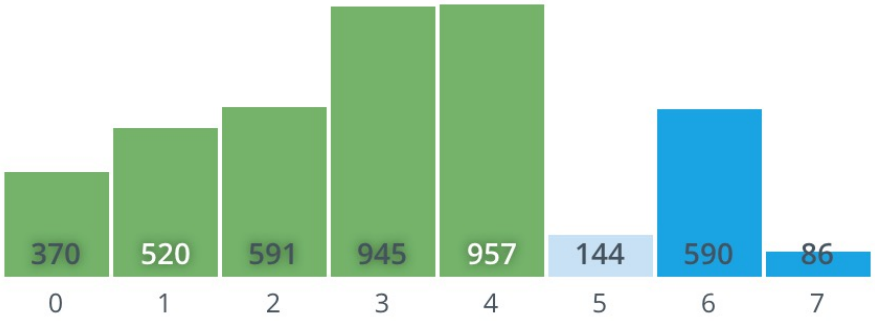
- **Comparison Sorts:** only compare keys and move items
- **Adjacent Sort:** Algorithms that sort by only swapping adjacent elements
  - e.g., bubble sort and insertion sort
  - …these are a subset of comparison sorts.
- **Stable Sort:** A sorting algorithm is stable
  - when two items x and y occur 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 Θ(1) extra space
  - e.g., allocating another array of size Θ(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

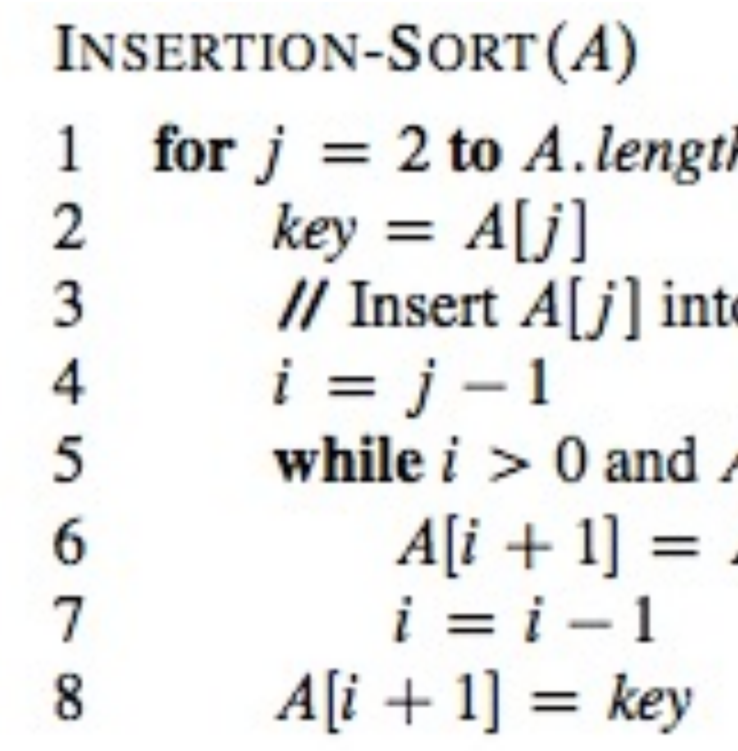
Insertion Sort

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 jth 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 **$\Omega(n^2)$** and can never be **$o(n^2)$**
  - **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
```

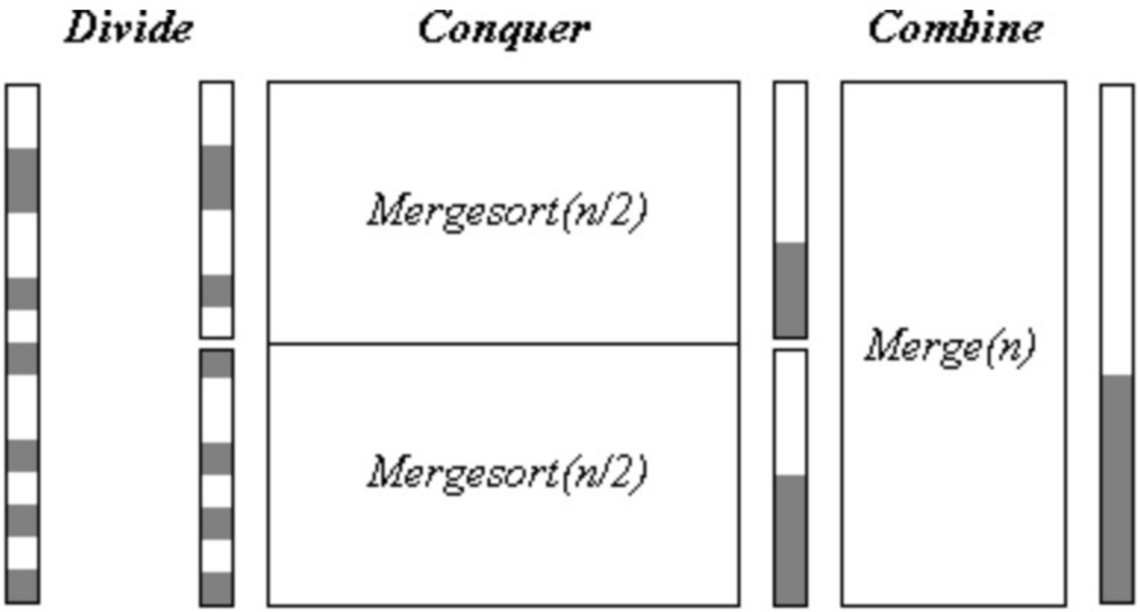
- Note: maybe solve all the smaller problem, or maybe just some of them.
- Runtime is sum of the times to divide, recursively solve, and combine

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, Θ(n)

Mergesort Analysis

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

  T(n) = 2T(n/2) + n = Θ(n log(n))
  - Uhhhhh...why?
- Next lecture 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 problem and class of algorithms

Quicksort and Partition

Readings: CLRS Chapter 7 (not 7.4.2)

Quicksort: Introduction

- Developed by C.A.R. (Tony) Hoare (a Turing Award winner)
  http://www.wikipedia.org/wiki/C._A._R._Hoare
  - Published in 1962
- Classic divide and conquer, but…
  - Mergesort does no comparisons to divide, but a lot to combine results (i.e. the merge) at each step
  - Quicksort does a lot of work to divide, but has nothing to do after the recursive calls. No work to combine.
    - If we're using arrays. Linked lists? Interesting to think about this!
- Dividing done with algorithm often called ***partition***
  - Sometimes called *split*. Several variations.

Quicksort's Strategy

- Called on subsection of array from *first* to *last*
  - Like mergesort
- First, choose some element in the array to be the ***pivot*** element
  - Any element! Doesn't matter for correctness.
  - Often the first item. For us, the last. Or, we often move some element into the last position (to get better efficiency)
- Second, call ***partition***, which does two things:
  - Puts the pivot in its proper place, i.e. where it will be in the correctly sorted sequence
  - All elements below the pivot are less-than the pivot, and all elements above the pivot are greater-than
- Third, use quicksort recursively on both sub-lists

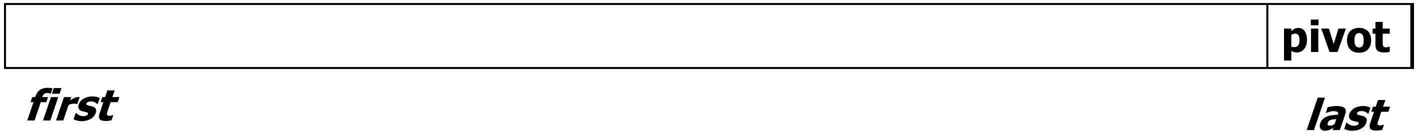
Quicksort is Divide and Conquer

- Divide: select pivot element p, Partition(p)
- Conquer: recursively sort left and right sublists
- Combine: Nothing!

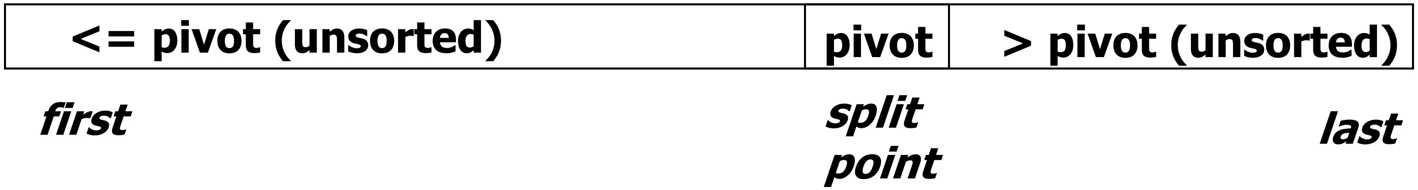
Contrast to mergesort,
where divide is simple and combine is work

Quicksort’s Strategy (a picture)

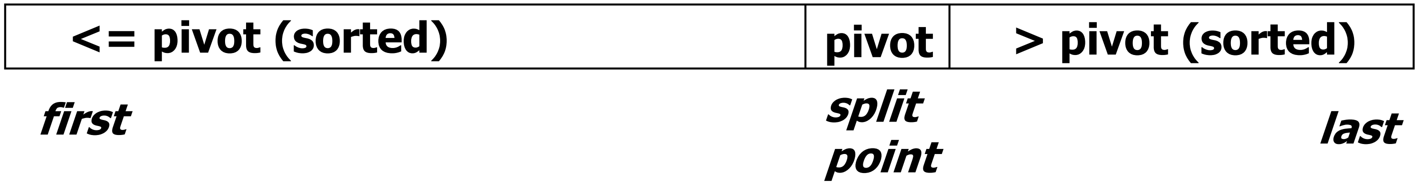
- Use last element as pivot (or pick one and move it there)



  - After call to partition…



  - Now sort two parts recursively and we’re done!



  - Note that splitPoint may be anywhere in *first..last*
  - Note our assumption that all keys are distinct

Quicksort Code

Input Parameters: *list, first, last*

Output Parameters: *list*

```
def quicksort(list, first, last):
    if first < last:
        q = partition(list, first, last)
        quicksort(list, first, q-1)
        quicksort(list, q+1, last)
    return
```

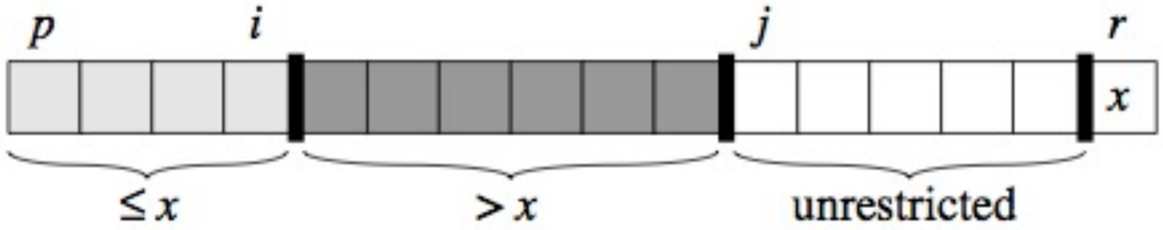
Partition Does the Dirty Work

- Partition rearranges elements
  - How? How many comparisons? How many swaps?
- How? Two well-known algorithms
  - In this chapter of CLRS, Lomuto’s algorithm
  - In the exercises, the original: Hoare’s algorithm. (Page 185. Look at on your own.)
  - Important:
    - Both are in-place!
    - Both are linear.

Strategy for Lomuto’s Partition

Note: book uses *p* and *r* for what we called *first* and *last* in earlier slides

- Invariant: At any point:
  - *i* indexes the right-most element <= *pivot*
  - *j-1* indexes the right-most element > *pivot*



- Strategy:
  - Look at next item *a[j]*
  - If that item > pivot, all is well!
  - If that item < pivot, increment *i* and then swap items at positions *i* and *j*
  - When done, swap pivot with item at position *i+1*
- Number of comparisons: n-1

Efficiency of Quicksort

- Partition divides into two sub-lists, perhaps unequal size
  - Depends on value of pivot element
- Recurrence for Quicksort

  $T(n)$ = partition-cost +

  $T($size of 1st section$) + T($size of 2nd section$)$
- If divides equally, $T(n) = 2\,T(n/2) + n-1$
  - Just like mergesort
  - Solve by substitution or master theorem

    $T(n) \in \Theta(n \lg n)$
- This is the best-case. But…

Worst Case of Quicksort

- What if divides in most unequal fashion possible?
  - One subsection has size 0, other has size n-1
  - T(n) = T(0) + T(n-1) + n-1
  - What if this happens every time we call partition recursively?

$$W(n) = \sum_{k=2}^{n}(k-1) \in \Theta(n^2)$$

  - Uh oh. Same as insertion sort.
    - "Sorry Prof. Hoare – we have to take back that Turing Award now!"

Quicksort's Average Case

- Good if it divides equally, bad if most unequal.
  - Remember: when subproblems size 0 and n-1
  - Can worst-case happen?
    Sure! Many cases. One is when elements already sorted. Last element is max, pivot around that. Next pivot is 2nd max…
- What's the average?
  - Much closer to the best case
  - A bad-split then a good-split is closer to best-case (pp. 176-178)
  - To prove A(n), fun with recurrences!
  - The result: If all permutations are equal, then
    A(n) ≅ 1.386 n lg n (for large n)
- So very fast on average.
- And, we can take simple steps to avoid the worst case!

Avoiding Quicksort’s Worst Case

- Make sure we don’t pivot around max or min
  - Find a better choice and swap it with last element
  - Then partition as before
- Recall we get best case if divides equally
  - Could find median. But this costs Θ(n). Instead…
  - Choose a **random element** between first and last and swap it with the last element
  - Or, estimate the median by using the “median-of-three” method
    - Pick 3 elements (say, first, middle and last)
    - Choose median of these and swap with last. (Cost?)
    - If sorted, then this chooses real median. Best case!

Tuning Quicksort's Performance

- In practice quicksort runs fast
  - A(n) is log-linear, and the "constants" are smaller than mergesort and heapsort
  - Often used in software libraries
  - So worth tuning it to squeeze the most out of it
  - <u>Always</u> do something to avoid worst-case
- Sort small sub-lists with (say) insertion sort
  - For small inputs, insertion sort is fine
    - No recursion, function calls
  - Variation: don't sort small sections at all.
    After quicksort is done, sort entire array with **insertion sort**
    - It's efficient on almost-sorted arrays!

Quicksort’s Space Complexity

- Looks like it’s in-place, but there’s a *recursion stack*
  - Depends on your definition: some people define *in-place* to **not** include stack space used by recursion
    - E.g. our CLRS algorithms textbook
    - Other books and people do “count” this
  - How much goes on the stack?
    - If most uneven splits, then Θ(n).
    - If splits evenly every time, then Θ(lg n).
- Ways to reduce stack-space used due to recursion
  - Various books cover the details (not ours, though)
  - First, remove 2nd recursive call (tail-recursion)
  - Second, always do recursive call on smaller section

Summary: Quicksort

- Divide and conquer where divide does the heavy-lifting
- In worst-case, efficiency is $\Theta(n^2)$
  - But it's practical to avoid the worst-case
- On average, efficiency is $\Theta(n \lg n)$
- Better space-complexity than mergesort.
- In practice, runs fast and widely used
  - Many ways to tune its performance
- Various strategies for Partition
  - Some work better if duplicate keys
- More details? See Sedgewick's algorithms textbook
  - He's the expert! PhD on this under Donald Knuth