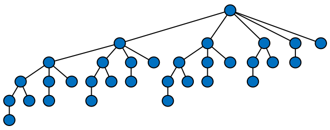
Lecture 4. Sorting, Models of Computation, Lower Bounds

CpSc 8400: Algorithms and Data Structures

Brian C. Dean



School of Computing
Clemson University
Spring, 2016

Models of Computation

- **Question:** What is the worst-case running time of the fastest possible algorithm for sorting n numbers?

A. O(n log n)
B. $O(n (\log \log n)^{1/2})$
C. O(n)
D. O(1)

Models of Computation

- **Answer:** It depends on our model of computation!

A. In the comparison model: O(n log n)

B. In the RAM model: $O(n (\log \log n)^{1/2})$ in expectation

C. In the RAM model with small word size: O(n)

D. In a (very unrealistic!) model where our machine has a “sort n integers” instruction: O(1)

3

Models of Computation

- In order to speak of the “running time” (number of fundamental operations) of an algorithm, we need to know what constitutes a fundamental operation.
- A *model of computation* defines the primitive operations in the abstract computing environment that our algorithm can use.
- Some algorithms look faster merely because they assume a more powerful model of computation.
  - Example: the *element uniqueness* problem (determining if an *n*-element array contains *n* distinct elements) requires $\Omega(n \log n)$ time in the comparison model, but can be solved in O(n) expected time on a RAM.
  - The O(n) RAM algorithm is not necessarily “better”…

4

The Random Access Machine (RAM) Model

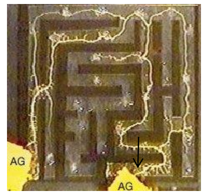
- Our default model of computation.
- Memory is a long array of *words* (b-bit integers).
- Can perform random access into memory: setting and retrieving a word based on its index takes O(1) time.
- Simple arithmetic operations also take O(1) time: adding two words, multiplying two words, comparing two words, etc…
- Fairly good abstract model of a modern digital computer. Or is it…?
- Tricky subtlety: what can we assume about b?

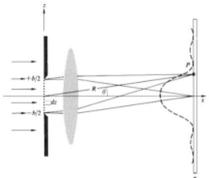
Other Models

- The **real RAM**.
  - Words can, if desired, hold real numbers. We can perform arithmetic on two real numbers in O(1) time.
  - Not realistic, but a useful simple model for problems that might require real numbers.
    - e.g., geometric problems where irrational distances might arise
  - Roundoff errors must be considered when actually implementing a real RAM algorithm.
- The **comparison-based** model.
  - Input elements not necessarily numeric, so can't use mathematical operations on them. Two input elements can only be compared.
  - Nice model for designing sorting/searching algorithms.

Exotic “Models of Computation”

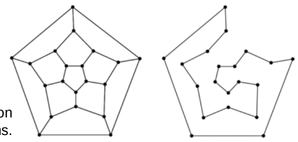
- Physical models
  (e.g., ball + string network models for computing shortest paths)
- Chemical models
  (e.g., DNA computing for Hamiltonian cycles)
- Biological models
  (e.g., computing shortest paths with slime molds)
- Optical models
  (e.g., computing a Fourier transform by diffraction of light)
- Quantum computation





Hecht. Optics, 3rd edition

L.M. Adleman. Molecular computation of solutions to combinatorial problems. *Science* 11 (1994).

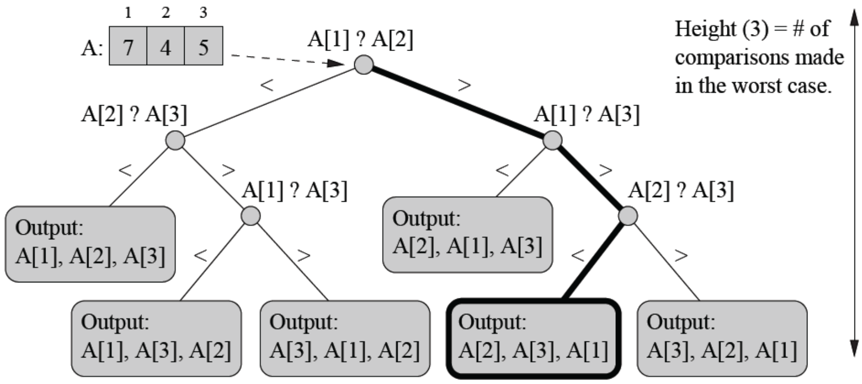


Lower Bounds for Comparison-Based Sorting Algorithms

- As it turns out, *any* algorithm that correctly sorts in the comparison-based model must make Ω(n log n) comparisons in the worst case, so there is an Ω(n log n) lower bound on the worst-case running time of any such algorithm.
- So an O(n log n) sorting algorithm has an **optimal** worst-case runtime in the comparison model.
- Much harder in general to prove lower bounds than upper bounds.
  – Most lower bounds we know are trivial: e.g., it take $\Omega(n^2)$ time to multiply two n x n matrices.

Decision Trees

- Any deterministic comparison-based algorithm can be modeled as an abstract *decision tree*:



9

Lower Bounds for Comparison-Based Sorting Algorithms

- A decision tree of height h has ≤ 2^h leaves
  (assume for simplicity that we are sorting distinct elements, so < and > are the only two possible comparisons)
- Suppose # of leaves is less than n!...
  - Then the **pigeonhole principle** tells us that two different input permutations end up at the same leaf.
  - These inputs "look the same" to our algorithm, since they generate the same comparison results.
  - Our algorithm would apply the same permutation to both inputs, sorting one of them incorrectly!
- Therefore, we must have $2^h \geq n!$ for our algorithm to sort correctly. By Stirling's approximation, we have $h \geq \log n! = \Theta(n \log n)$, so $h = \Omega(n \log n)$.

10

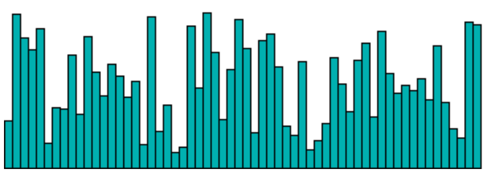
Lower Bounds Based on Reductions

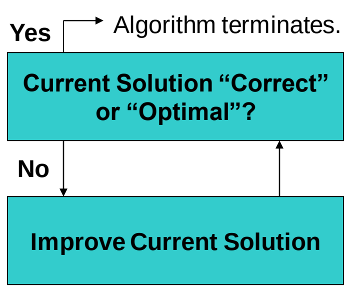
- A similar decision tree argument can be used to prove that an algorithm for element uniqueness requires $\Omega(n \log n)$ worst-case time in the comparison model.
- Both sorting and element uniqueness have worst-case lower bounds of $\Omega(n \log n)$ in the real RAM model, but these can be harder to prove!
- Now that we have a non-trivial lower bound for these problems, we can carry this bound over to other problems via **reductions.**
  - Example: It takes $\Omega(n \log n)$ worst-case time to determine the closest pair out of n points in the plane. *(careful: this not a comparison-based problem!)*

11

Bubble Sort (Iterative Refinement)

- Repeatedly scan array, swapping out-of-order pairs of consecutive elements.
- $O(n^2)$ time ($\Theta(n^2)$ in the worst case).
- Stable, in-place

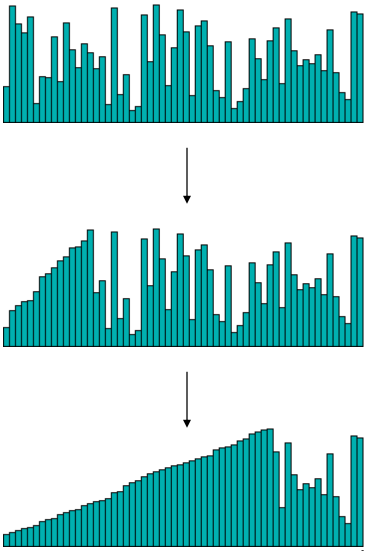




12

Insertion Sort (Incremental Construction)

- In each iteration, enlarge sorted prefix of array by one element.
- Runtime $\Theta(n + I) = O(n^2)$, where I is the number of **inversions** in our input.
  - Good for sorting nearly-sorted sequences. In fact, if $I = O(n)$, then insertion sort runs in linear time!
- Stable, in-place.



Selection Sort

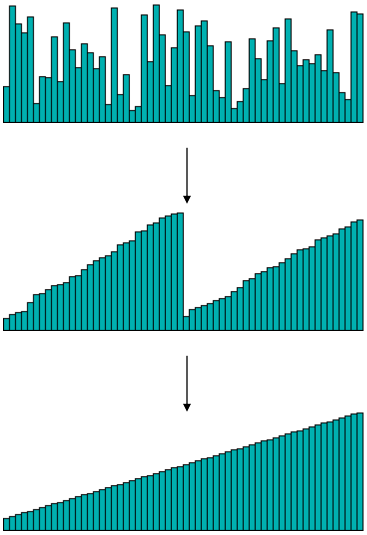
- Scan A[1…n] for smallest element, and swap it with A[1].
- Then scan A[2…n] for smallest element, and swap it with A[2]. Etc.

- What is its running time? Is it stable and/or in-place?
- How would you classify this algorithm?
- How does it compare to bubble sort and insertion sort?

Merge Sort (Divide and Conquer)

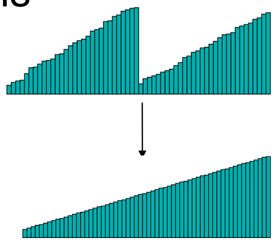
- Recursively sort first and second halves of array, then **merge** the two resulting sorted subarrays in linear time.

  (i.e., "divide, conquer, recombine")
- Runtime Θ(n log n), so it's an optimal comparison-based sort.
- Stable, but not in-place.

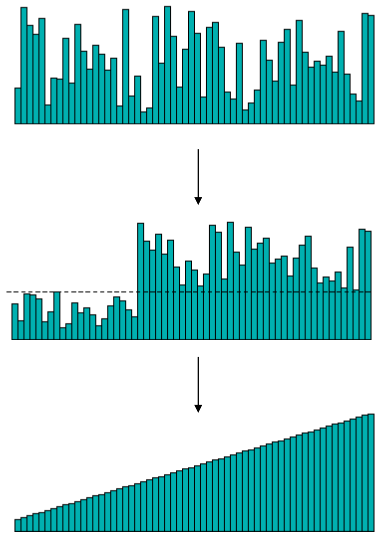


Merge Sort Analysis: Running Time per Element of Data

- It takes Θ(n) time to merge two lists of combined length n.
- Think of this as O(1) time per element taking part in the merge.
- Now look at a particular array element e. The total "work" we spend on e is equal to the number of merges in which e takes part: O(log n).
  - Why O(log n)? Each merge doubles the size of the sorted subarray containing e.
- So O(n log n) total work.



Quicksort (Divide and Conquer)



- In linear time, **partition** array based on the value of some **pivot** element.
- Then recursively sort left side (all elements ≤ pivot) and right side (≥ pivot).
- Typical implementation is in-place, but not stable.
- How to partition:
  – in linear time?
  – if some elements equal?

Quicksort Variants

- **Simple quicksort.** Choose pivot using a simple deterministic rule; e.g., first element, last element, median(A[1], A[n], A[n/2]).
  – $\Theta(n \log n)$ time if "lucky", but $\Theta(n^2)$ worst-case.
- **Deterministic quicksort.** Pivot on median (we'll see shortly how to find the median in linear time).
  – $\Theta(n \log n)$ time, but not the best in practice.
- **Randomized quicksort**. Choose pivot uniformly at random.
  – $\Theta(n \log n)$ time with high probability, and fast in practice (competitive with merge sort).

Further Thoughts

- Any sorting algorithm can be made **stable** at the expense of **in-place** operation

  (so we can implement quicksort to be stable but not in-place, or in-place but not stable).
- Memory issues:
  - Rather than sort large records, sort pointers to records.
  - Some advanced sorting algorithms only move elements of data O(n) total times.
  - How will caching affect the performance of our various sorting algorithms?

Back to Comparison-Based Sorting – An Obvious Question…

Algorithm	**Runtime**	**Stable**	**In-Place?**
Bubble Sort	$O(n^2)$	Yes	Yes
Insertion Sort	$O(n^2)$	Yes	Yes
Merge Sort	$\Theta(n \log n)$	Yes	No
Randomized Quicksort	$\Theta(n \log n)$ w/high prob.	No*	Yes*
Deterministic Quicksort	$\Theta(n \log n)$	No*	Yes*
Heap Sort	$\Theta(n \log n)$	No*	Yes*

- Is stable in-place sorting possible in O(n log n) time in the comparison-based model?

* = can be transformed into a stable, out-of-place algorithm

The Ideal Comparison-Based Sorting Algorithm…

- …would be **stable** and **in-place**.
- …would require only **O(n) moves** (memory writes)
- …would be **simple** and **deterministic**.
- …would run in **O(n log n)** time.

- We currently only know how to achieve limited combinations of the above properties…

21

Adaptive Sorting

- The ideal sorting algorithm would run in O(n log n) worst-case time, but would be even faster if the input is "nearly sorted"
  - E.g., Merge sort, with its Θ(n log n) running time, is much worse than insertion sort, in the event that we are sorting an already sorted array (which is not all that uncommon in practice…)
- How can we characterize unsortedness?

22

Adaptive Sorting

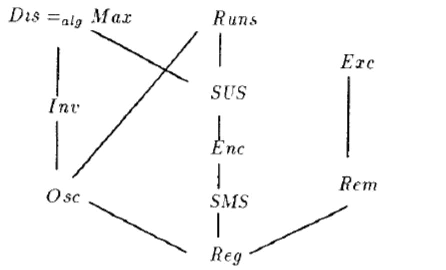


Figure 3. The partial order of the 11 measures of disorder. The ordering with respect to $\leq_{alg}$ is up the page; for example, *Inv* optimality implies *Dis* optimality.

Table 1. The Worst-Case Lower Bounds for Different Measures of Disorder

M	Lower bound: $\log \lVert below(M(X), \lvert X \rvert, M) \rVert$
Dis	$\Omega(\lvert X\rvert(1+\log[Dis(X)+1]))$
Exc	$\Omega(\lvert X\rvert + Exc(X)\log[Exc(X)+1])$
Enc	$\Omega(\lvert X\rvert(1+\log[Enc(X)+1]))$
Inv	$\Omega\left(\lvert X\rvert\left(1+\log\left[\frac{Inv(X)}{\lvert X\rvert}+1\right]\right)\right)$
Max	$\Omega(\lvert X\rvert(1+\log[Max(X)+1]))$
Osc	$\Omega\left(\lvert X\rvert\left(1+\log\left[\frac{Osc(X)}{\lvert X\rvert}+1\right]\right)\right)$
Reg	$\Omega(\lvert X\rvert + \log[Reg(X)+1])$
Rem	$\Omega(\lvert X\rvert + Rem(X)\log[Rem(X)+1])$
$Runs$	$\Omega(\lvert X\rvert(1+\log[Runs(X)+1]))$
SMS	$\Omega(\lvert X\rvert(1+\log[SMS(X)+1]))$
SUS	$\Omega(\lvert X\rvert(1+\log[SUS(X)+1]))$

From "A survey of adaptive sorting algorithms", by Estivill-Castro and Wood (1992)

23

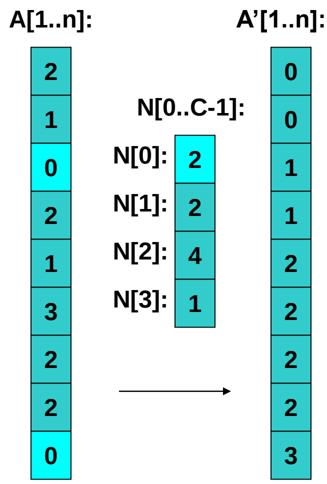
RAM Sorting Algorithms

- Suppose we are sorting n integers in the range 0…C – 1 in the RAM model of computation.
- **Counting sort**: O(n + C) time.
  - Sorts integers of magnitude C = O(n) in linear time.
- **Radix sort**: O(n max(1, $\log_n$C)) time.
  - Sorts integers of magnitude C = O(n^k), k = O(1), in linear time.

24

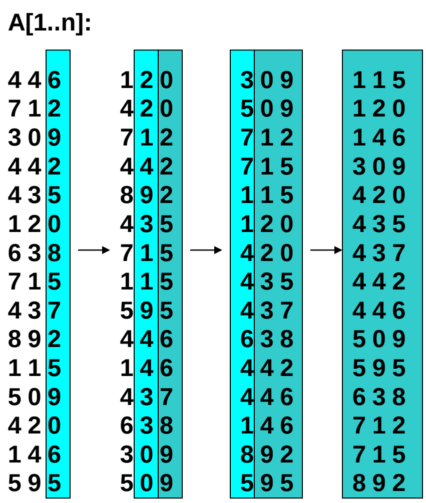
Counting Sort

- Scan A[1..n] in O(n) time and build an array N[0..C-1] of element counts.
- By scanning N, we then reconstruct A in sorted order in O(n + C) time.
- Ideally suited for C = O(n).
- Not in-place.
- Stable, if we're careful…



Radix Sort

- Write elements of A[1..n] in some base (radix), r. Typically, we set r = n.
- Sort on each digit, starting with the **least** significant, using a **stable** sort.
- # digits = $\log_n C$, which is constant if C = $n^{O(1)}$.
- Runtime O(n) if C = $n^{O(1)}$. (recall word size assumptions w/RAM)
- Stable, not in-place



Implications of Sorting Lower Bounds: Might This Be Possible?

- Can we build a comparison-based priority queue satisfying these bounds?
  - Insert: O(1) amortized time
  - Remove-Min: O(1) amortized time

Implications of Sorting Lower Bounds: Might This Be Possible?

- Can we build a comparison-based priority queue satisfying these bounds?
  - Insert: O(1) amortized time
  - Remove-Min: O(1) amortized time
- Can we achieve these bounds with a comparison-based min-aware data structure (a data structure that always knows its minimum element):
  - Insert: O(1) amortized time
  - Delete: O(1) amortized time

Implications of Sorting Lower Bounds: Might This Be Possible?

- Can we achieve these bounds with a comparison-based min-aware data structure (a data structure that always knows its minimum element):
  - Insert: O(1) time
  - Delete: O(1) time*
- This doesn't seem to contradict the Ω(n log n) lower bound on sorting, does it?
- These guarantees would be nice to have if we only occasionally need the functionality of a priority queue.

* Unless deleting the minimum, which takes O(log n) time.

Rank-Sensitive Priority Queues

- It turns out this is the "right" answer:
  - Insert: O(log (n/r)) time
  - Delete: O(log (n/r)) time

    (r = rank of the element being inserted or deleted, where r = 1 is the minimum and r = n is the maximum)
- So running time scales gracefully from O(log n) for operating on elements close to the minimum down to O(1) for elements far away from the minimum (say, in the larger 99% of the data set).