CSE332: Data Abstractions

Lecture 14: Beyond Comparison Sorting

Dan Grossman
Spring 2010

The Big Picture

Surprising amount of juicy computer science: 2-3 lectures…

Simple algorithms: O(n^2)	**Fancier algorithms: O(n log n)**	**Comparison lower bound: Ω(n log n)**	**Specialized algorithms: O(n)**	**Handling huge data sets**
Insertion sort **Selection sort** Shell sort …	**Heap sort** **Merge sort** **Quick sort (avg)** **…**		**Bucket sort** **Radix sort**	**External sorting**

How fast can we sort?

- Heapsort & mergesort have $O(n \log n)$ worst-case running time
- Quicksort has $O(n \log n)$ average-case running times
- These bounds are all tight, actually $\Theta(n \log n)$
- So maybe we need to dream up another algorithm with a lower asymptotic complexity, such as $O(n)$ or $O(n \log \log n)$
  - Instead: *prove* that this is *impossible*
    - *Assuming* our comparison *model*: The only operation an algorithm can perform on data items is a 2-element comparison

Permutations

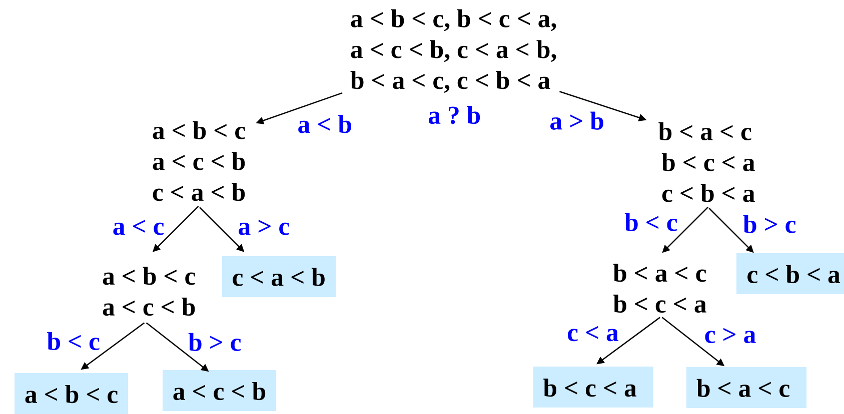
- Assume we have n elements to sort
  - And for simplicity, none are equal (no duplicates)
- How many permutations (possible orderings) of the elements?
- Example, n=3

  a[0]<a[1]<a[2] a[0]<a[2]<a[1] a[1]<a[0]<a[2]
  a[1]<a[2]<a[0] a[2]<a[0]<a[1] a[2]<a[1]<a[0]
- In general, n choices for least element, then n-1 for next, then n-2 for next, …
  - $n(n-1)(n-2)\ldots(2)(1) = n!$ possible orderings

Describing every comparison sort

- So every sorting algorithm has to "find" the right answer among the n! possible answers
- Starts "knowing nothing" and gains information with each comparison
  - Intuition: At best, each comparison can eliminate half of the remaining possibilities
- Can represent this process as a decision tree
  - Nodes are "remaining possibilities"
  - Edges are "answers from a comparison"
  - This is not a data structure, it's what our proof uses to represent "the most any algorithm could know"

Decision tree for n=3

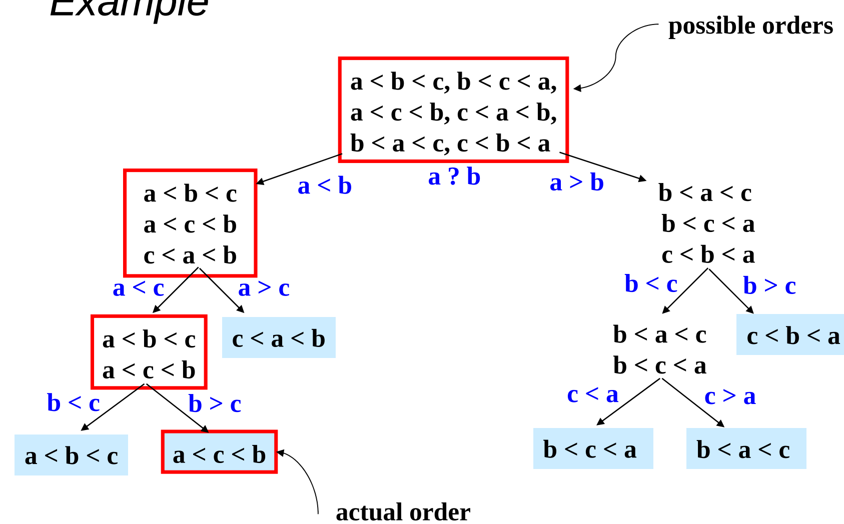


The leaves contain all the possible orderings of a, b, c

What the decision tree tells us

- A binary tree because each comparison has 2 outcomes
  - No duplicate elements
  - Assume algorithm not so dumb as to ask redundant questions
- Because any data is possible, any algorithm needs to ask enough questions to produce all n! answers
  - Each answer is a leaf (no more questions to ask)
  - So the tree must be big enough to have n! leaves
  - Running any algorithm on any input will at best correspond to one root-to-leaf path in the decision tree
  - So no algorithm can have worst-case running time better than the height of the decision tree

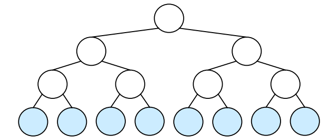
Example



Where are we

- Proven: No comparison sort can have worst-case running time better than the height of a binary tree with *n*! leaves
  - Turns out average-case is same asymptotically
- Now: a binary tree with n! leaves has height $\Omega(n \log n)$
  - Factorial function grows very quickly
- Conclusion: (Comparison) Sorting is $\Omega(n \log n)$
  - This is an amazing computer-science result: proves all the clever programming in the world can't sort in linear time

Lower bound on height



- The height of a binary tree with *L* leaves is at least $\log_2 L$
- So the height of our decision tree, *h:*

$h \geq \log_2 (n!)$	property of binary trees
$= \log_2 (n*(n-1)*(n-2)\ldots(2)(1))$	definition of factorial
$= \log_2 n + \log_2 (n-1) + \ldots + \log_2 1$	property of logarithms
$\geq \log_2 n + \log_2 (n-1) + \ldots + \log_2 (n/2)$	drop smaller terms (≥ 0)
$\geq (n/2) \log_2 (n/2)$	each of the n/2 terms left is $\geq \log_2 (n/2)$
$= (n/2)(\log_2 n - \log_2 2)$	property of logarithms
$= (1/2)n\log_2 n - (1/2)n$	arithmetic
"=" $\Omega(n \log n)$	

The Big Picture

Surprising amount of juicy computer science: 2-3 lectures…

Simple algorithms: $O(n^2)$
- **Insertion sort**
- **Selection sort**
- **Shell sort**
- **…**

Fancier algorithms: $O(n \log n)$
- **Heap sort**
- **Merge sort**
- **Quick sort (avg)**
- **…**

Comparison lower bound: $\Omega(n \log n)$

Specialized algorithms: $O(n)$
- **Bucket sort**
- **Radix sort**

Handling huge data sets
- **External sorting**

huh???
- Change the model – assume more than items can be compared!

BucketSort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and *K* (or any small range), create an array of size *K* and put each element in its proper bucket (a.k.a. bin)
  - *If* data is only integers, don't even need to store anything more than a *count* of how times that bucket has been used
- Output result via linear pass through array of buckets

`count` array	
1	3
2	1
3	2
4	2
5	3

- Example:
  K=5
  input (5,1,3,4,3,2,1,1,5,4,5)
  output: 1,1,1,2,3,3,4,4,5,5,5

Analyzing bucket sort

- Good when range, K, is smaller (or not much larger) than number of elements, n
  - Don't spend time doing lots of comparisons of duplicates!
- Bad when K is much larger than n
  - Wasted space; wasted time during final linear $O(K)$ pass
- Overall: $O(n+K)$
  - Linear in n, but also linear in K
  - $\Omega(n \mathbf{log}\ n)$ lower bound does not apply because this is not a comparison sort
- For data in addition to integer keys, use list at each bucket

Radix sort

- Radix = "the base of a number system"
  - Examples will use 10 because we are used to that
  - In implementations use larger numbers
  - For example, for ASCII strings, might use 128
- Idea:
  - Bucket sort on one digit at a time
    - Number of buckets = radix
    - Starting with *least* significant digit
    - Keeping sort *stable*
  - After k passes (digits), the last k digits are sorted
- Aside: Origins go back to the 1890 U.S. census

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

Input: 478
537
9
721
3
38
143
67

First pass:
bucket sort by ones digit

Order now: 721
3
143
537
67
478
38
9

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

Order was: 721
3
143
537
67
478
38
9

Second pass:
stable bucket sort by tens digit

Order now: 3
9
721
537
38
143
67
478

Example

0	1	2	3	4	5	6	7	8	9
3 **9**		**721**	**537** **38**	**143**		**67**	**478**		

Radix = 10

0	1	2	3	4	5	6	7	8	9
3 **9** **38** **67**	**143**			**478**	**537**		**721**		

Order was: 3, 9, 721, 537, 38, 143, 67, 478

Third pass:
stable bucket sort by 100s digit

Order now: 3, 9, 38, 67, 143, 478, 537, 721

Analysis

Input size: *n*
Number of buckets = Radix: *B*
Number of passes = “Digits”: *P*

Work per pass is 1 bucket sort: *O*(*B*+*n*)

Total work is *O*(*P*(*B*+*n*))

Compared to comparison sorts, sometimes a win, but often not
- Example: Strings of English letters up to length 15
  - 15*(52 + *n*)
  - This is less than *n* log n only if *n* > 33,000
  - Of course, cross-over point depends on constant factors of the implementations plus *P* and *B*
    - And radix sort can have poor locality properties

Last word on sorting

- Simple $O(n^2)$ sorts can be fastest for small *n*
  - selection sort, insertion sort (latter linear for mostly-sorted)
  - good for “below a cut-off” to help divide-and-conquer sorts
- *O*(*n* **`log`** *n*) sorts
  - heap sort, in-place but not stable nor parallelizable
  - merge sort, not in place but stable and works as external sort
  - quick sort, in place but not stable and $O(n^2)$ in worst-case
    - often fastest, but depends on costs of comparisons/copies
- Ω (*n* **`log`** *n*) is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small number of key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!