Beyond Comparison Sorting

CSE 373
Data Structures & Algorithms
Ruth Anderson

Today's Outline

- **Admin**:
  - HW #5 – Graphs, due Thurs March 1 at 11pm
  - HW #6 – last homework, on sorting, individual project, no Java programming, coming soon, due Thurs March 8.
- **Sorting**
  - Comparison Sorting
  - *Beyond* Comparison Sorting

The Big Picture

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

Comparison Sorting

So far we have only talked about *comparison sorting*:

Assume we have *n comparable* elements in an array and we want to rearrange them to be in increasing order:

Input:

- An array **A** of data records
- A key value in each data record
- A comparison function (consistent and total)
  - Given keys a & b, what is their relative ordering? <, =, >?
  - Ex: keys that implement Comparable or have a Comparator that can handle them

Effect:

- Reorganize the elements of **A** such that for any `i` and `j`, if `i < j` then `A[i] ≤ A[j]`

An algorithm doing this is a comparison sort

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
- 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: we actually KNOW that this is *impossible!!*
  - *(See end of slide deck for proof)*
- *In particular, it is impossible assuming* our comparison *model*: The only operation an algorithm can perform on data items is a 2-element comparison

The Big Picture

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???

- Change the model – assume more than 'compare(a,b)'

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.ka. 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	
2	
3	
4	
5	

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

03/02/2012 7

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.ka. 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

What is the running time?

03/02/2012 8

Analyzing bucket sort

- Overall: $O(n+K)$
  - Linear in *n*, but also linear in *K*
  - $\Omega(n \log n)$ lower bound does not apply because this is not a comparison sort
- Good when range, *K*, is smaller (or not much larger) than number of elements, *n*
  - We 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
- For data in addition to integer keys, use list at each bucket

03/02/2012 9

Bucket Sort with Data

- Most real lists aren't just #'s; we have data
- Each bucket is a list (say, linked list)
- To add to a bucket, place at end in O(1) (say, keep a pointer to last element)

count array	
1	→ **Rocky V**
2	
3	→ **Harry Potter**
4	
5	→ **Casablanca** → **Star Wars**

- Example: Movie ratings; scale 1-5;1=bad, 5=excellent
  Input=
  5: Casablanca
  3: Harry Potter movies
  5: Star Wars Original Trilogy
  1: Rocky V

•Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
•This result is 'stable'; Casablanca still before Star Wars

03/02/2012 10

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, sort with Bucket Sort
    - Keeping sort *stable*
  - Do one pass per digit
  - After *k* passes, the last *k* digits are sorted
- Aside: Origins go back to the 1890 U.S. census

03/02/2012 11

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:
1. bucket sort by ones digit
2. Iterate thru and collect into a list

Order now: 721
3
143
537
67
478
38
9

List is sorted by first digit. →

03/02/2012 12

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

If we chop off the 100's place, these #s are sorted →

Example

Radix = 10

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

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

Only 3 digits: We're done! →

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

Student Activity

RadixSort

- Input:126, 328, 636, 341, 416, 131, 328

BucketSort on lsd:

0	1	2	3	4	5	6	7	8	9

BucketSort on next-higher digit:

0	1	2	3	4	5	6	7	8	9

BucketSort on msd:

0	1	2	3	4	5	6	7	8	9

Analysis of Radix Sort

Performance depends on:

- Input size: *n*
- Number of buckets = Radix: *B*
  - e.g. Base 10 #: 10; binary #: 2; Alpha-numeric char: 62
- Number of passes = "Digits": *P*
  - e.g. Ages of people: 3; Phone #: 10; Person's name: ?

- Work per pass is 1 bucket sort: ___________
  - Each pass is a Bucket Sort
- Total work is ____________
  - We do 'P' passes, each of which is a Bucket Sort

Analysis of Radix Sort

Performance depends on:

- Input size: *n*
- Number of buckets = Radix: *B*
  - Base 10 #: 10; binary #: 2; Alpha-numeric char: 62
- Number of passes = "Digits": *P*
  - Ages of people: 3; Phone #: 10; Person's name: ?

- Work per pass is 1 bucket sort: $O(B+n)$
  - Each pass is a Bucket Sort
- Total work is $O(P(B+n))$
  - We do 'P' passes, each of which is a Bucket Sort

Comparison to Comparison Sorts

Compared to comparison sorts, sometimes a win, but often not

- Example: Strings of English letters up to length 15
  - Approximate run-time: $15*(52 + n)$
  - 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
- Not really practical for many classes of keys
  - Strings: Lots of buckets

Sorting massive data

- Need sorting algorithms that minimize disk/tape access time:
  - Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
  - Mergesort scans linearly through arrays, leading to (relatively) efficient sequential disk access
- MergeSort is the basis of massive sorting
- In-memory sorting of reasonable blocks can be combined with larger mergesorts
- Mergesort can leverage multiple disks

External Sorting

- For sorting massive data
- Need sorting algorithms that minimize disk/tape access time
- **External sorting** – Basic Idea:
  - Load chunk of data into Memory, sort, store this "run" on disk/tape
  - Use the Merge routine from Mergesort to merge runs
  - Repeat until you have only one run (one sorted chunk)
  - Text gives some examples

Features of Sorting Algorithms

In-place

- Sorted items occupy the same space as the original items. (No copying required, only O(1) extra space if any.)

Stable

- Items in input with the same value end up in the same order as when they began.

Examples:

- Merge Sort - not in place, stable
- Quick Sort - in place, not stable

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
  - 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
- $\Omega(n \log n)$ is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small maximum key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!

Extra Slides: Proof of Comparison Sorting Lower Bound

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

A Different View of Sorting

- 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 Different View of Sorting

- 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, six possibilities
  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)…(2)(1) = *n*! possible orderings

Describing every comparison sort

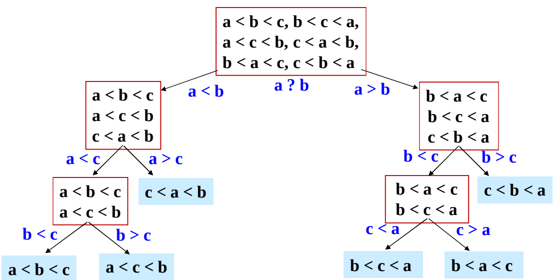
- A different way of thinking of sorting is that the sorting algorithm has to “find” the right answer among the n! possible answers
  - Starts “knowing nothing”, “anything is possible”
  - Gains information with each comparison, eliminating some possiblities
    - Intuition: At best, each comparison can eliminate half of the remaining possibilities
  - In the end narrows down to a single possibility

Representing the Sort Problem

- Can represent this sorting process as a *decision tree:*
  - **Nodes** are sets of “remaining possibilities”
  - At root, anything is possible; no option eliminated
  - **Edges** represent comparisons made, and the node resulting from a comparison contains only consistent possibilities
    - Ex: Say we need to know whether a<b or b<a; our root for n=2
    - A comparison between a & b will lead to a node that contains only one possibility (either a<b or b<a)

Note: *This tree is not a data structure*, it's what our proof uses to represent “the most any algorithm could know”

Decision tree for n=3

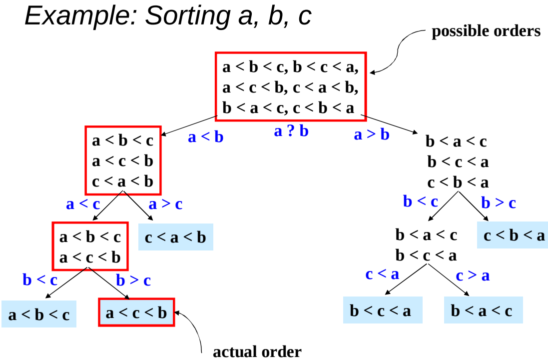


The leaves contain all the possible orderings of a, b, c

What the decision tree tells us

- A **binary** tree because each comparison has 2 outcomes
  - Perform only comparisons between 2 elements; binary result
    - Ex: Is a<b? Yes or no?
  - We assume no duplicate elements
  - Assume algorithm doesn't 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: Sorting a, b, c



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
- Fine, *how tall is a binary tree with n! leaves?*

Now: Show that a binary tree with n! leaves has height $\Omega(n \log n)$

- That is, n log n is the lower bound, the height must be at least this, could be more, (in other words your comparison sorting algorithm could take longer than this, but it won't be faster)
- Factorial function grows very quickly

Then we'll conclude that: (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!

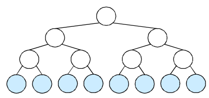
Lower bound on Height

- A binary tree of height h has **at most** *how many* leaves?
  L ≤ ________________
- A binary tree with L leaves has height **at least**:
  h ≥ ________________
- The decision tree has how many leaves: _______
- So the decision tree has height:
  h ≥ ________________

Lower bound on Height

- A binary tree of height h has **at most** *how many* leaves?
  L ≤ 2^h
- A binary tree with L leaves has height **at least**:
  h ≥ $\log_2 L$
- The decision tree has how many leaves: N!
- So the decision tree has height:
  h ≥ $\log_2 N!$

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
$\geq \log_2 n + \log_2 (n-1) + \ldots + \log_2 (n/2)$	keep first n/2 terms
$\geq (n/2) \log_2 (n/2)$	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
"=" $\Omega(n \log n)$	