CSE332: Data Abstractions

Lecture 14: Beyond Comparison Sorting

Tyler Robison

Summer 2010

The Big Picture

Simple algorithms: $O(n^2)$	Fancier algorithms: $O(n \log n)$	Comparison lower bound: $\Omega(n \log n)$	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 complexy
  - Maybe find something with $O(n)$ or $O(n \log \log n)$ (recall loglogn is smaller than logn)
  - 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
    - Show that the best we can do is O(nlogn), for the worst-case

Different View on 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, 6 possibilities:

$$a[0]<a[1]<a[2] \text{ or } a[0]<a[2]<a[1] \text{ or } a[1]<a[0]<a[2]$$

or

$$a[1]<a[2]<a[0] \text{ or } a[2]<a[0]<a[1] \text{ or } a[2]<a[1]<a[0]$$

- That is, these are the only possible permutations on the orderings of 3 distinct items
- Generalize to n (distinct) items:
  - *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

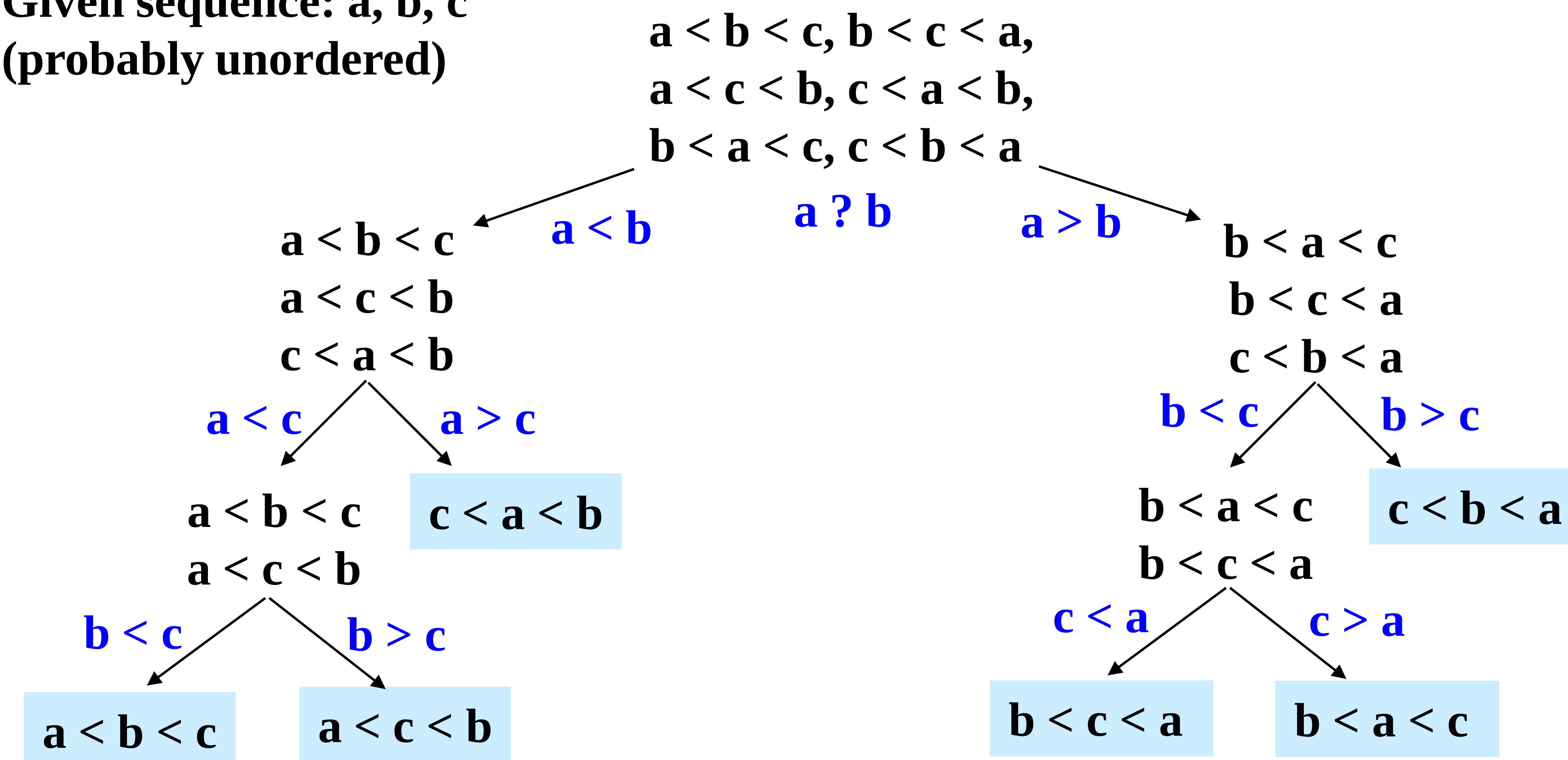
- A different way of thinking of sorting is that the sorting algorithm has to “find” the right answer among the n! possible answers
  - Starts “knowing nothing”; “anything’s possible”
  - Gains information with each comparison, eliminating some possibilities
    - Intuition: At best, each comparison performed 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
  - Note: This tree is not a data structure, it’s what our proof uses to represent “the most any algorithm could know”
- Aside: Decision trees are a neat tool, sometimes used in AI to, well, make decisions
  - At each state, examine information to reduce space of possibilities
  - Classical example: ‘Should I play tennis today?’; ask questions like ‘Is it raining?’, ‘Is it hot?’, etc. to work towards an answer

Decision tree for n=3

Given sequence: a, b, c (probably unordered)



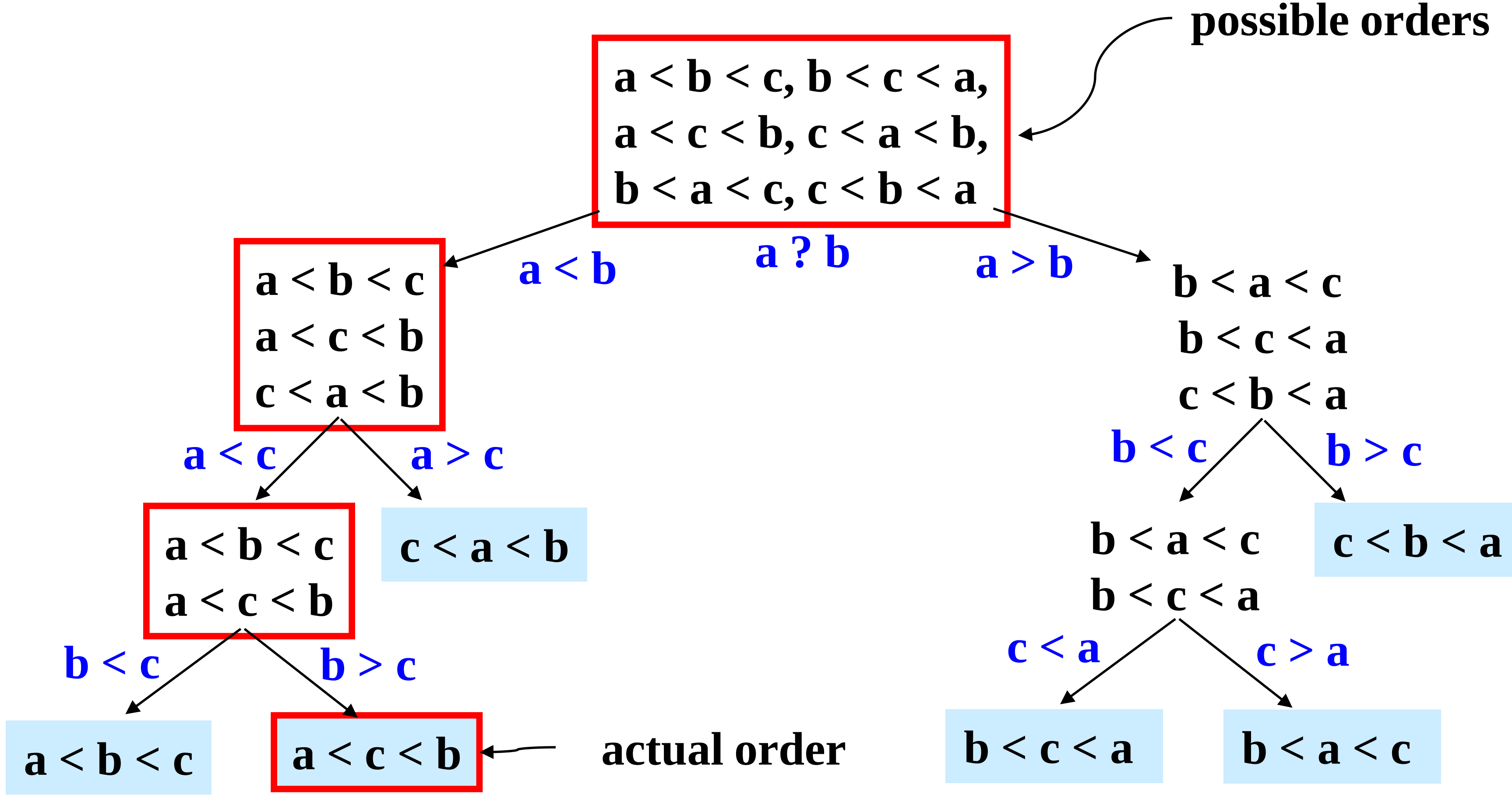
- **Each leaf is one outcome**
- **The leaves contain all outcomes; all possible orderings of a, b, c**

What the decision tree tells us

- A binary tree because each comparison has 2 possible 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

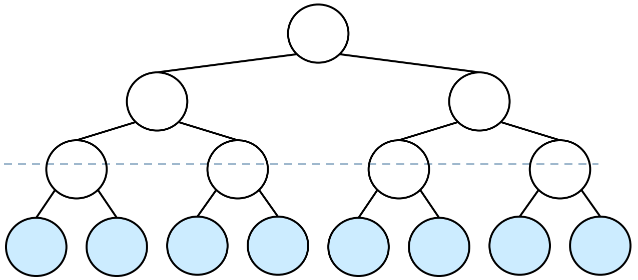
Example: Sorting some data a,b,c



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
- Great! Now how tall is that…
- Show that a binary tree with n! leaves has height $\Omega(n \log n)$
  - That is nlogn is the lower bound; the height must be at least that
  - Factorial function grows very quickly
- Then we'll conclude: 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 $\log_2 L$
  - If we pack them in as tightly as possible, each row has about 2x the previous row's nodes
- So the height of our decision tree, h:

$h \geq \log_2 (n!)$

$= \log_2 (n*(n-1)*(n-2)\ldots(2)(1))$	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

So $h \geq (1/2)n\log_2 n - (1/2)n$

“=“ $\Omega(n \log n)$

The Big Picture

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: Ω(n log n)

Specialized algorithms: O(n)

Bucket sort
Radix sort

Handling huge data sets

External sorting

Change the model – assume more than ‘compare(a,b)’

Non-Comparison Sorts

- Say we have a list of integers between 0 & 9 (ignore associated data for the moment)
  - Size of list to sort could be huge, but we'd have lots of duplicate values
  - Assume our data is stored in 'int array[]'; how about…

```
int[] counts=new int[10];
//init to counts to 0's
for(int i=0;i<array.length;i++) counts[array[i]]++;
```

  - Can iterate through array in linear time
  - Now return to array in sorted order; first counts[0] slots will be 0, next counts[1] will be 1…
  - We can put elements, in order, into array[] in O(n)
- This works because array assignment is sort of 'comparing' against every element currently in counts[] in constant time
  - Not merely a 2-way comparison, but an n-way comparison
  - Thus not under restrictions of nlogn for Comparison Sorts

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)
  - 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

If data is only integers, don't even need to store anything more than a *count* of how times that bucket has been used

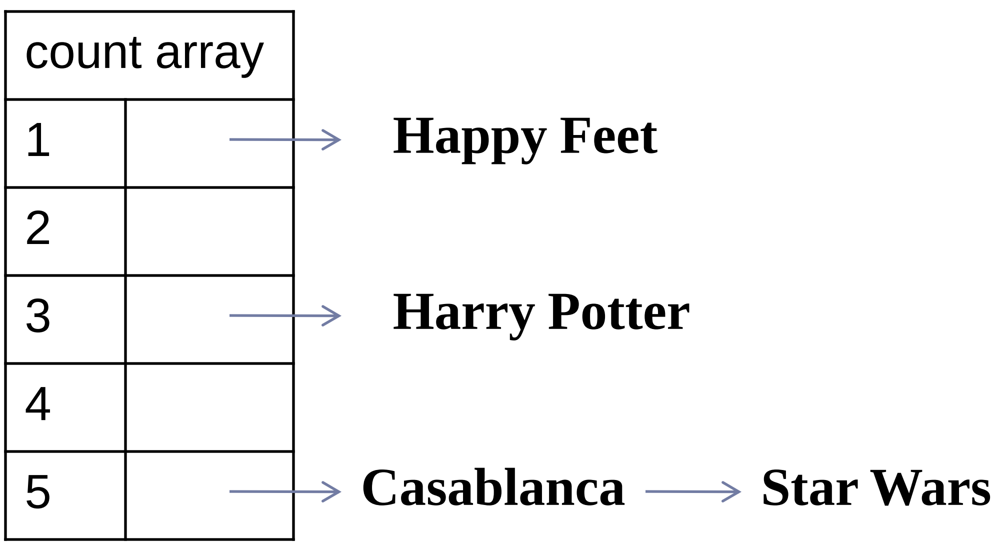
Analyzing bucket sort

- 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
  - 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
  - If $K \sim n^2$, not really linear anymore

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)



- Example: Movie ratings; scale 1-5;1=bad, 5=excellent

  Input=

  5: Casablanca

  3: Harry Potter movies

  5: Star Wars Original Trilogy

  1: Happy Feet

- Result: 1: Happy Feet, 3: Harry Potter, 5: Casablanca, 5: Star Wars
- This result is 'stable'; Casablanca still before Star Wars

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

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

Iterate through and collect into list

List is sorted by first digit

Order now: 721
3
143
537
67
478
38
9

Example

Radix = 10

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

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

Order now: 3
9
721
537
38
143
67
478

Example

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

Only 3 digits: We’re done

Order now: 3 9 38 67 143 478 537 721

Analysis

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

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

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