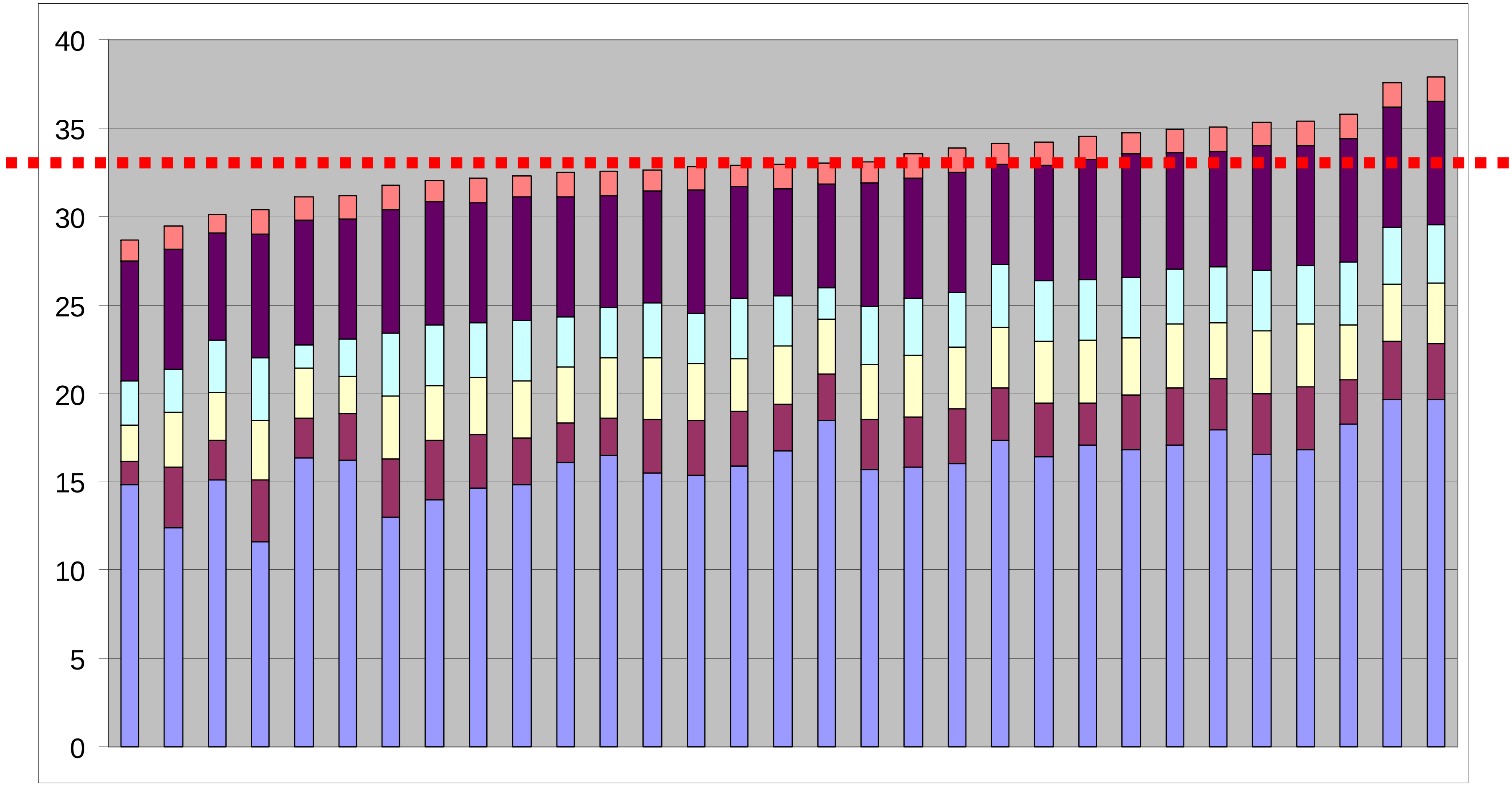
CSE 326: Data Structures
Sorting

James Fogarty

Autumn 2007

From Midterm Post-Mortem vs. Historical Average for this Course



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.

How fast can we sort?

- Heapsort, Mergesort, and Quicksort all run in O(N log N) best case running time
- Can we do any better?
- No, if the basic action is a comparison.

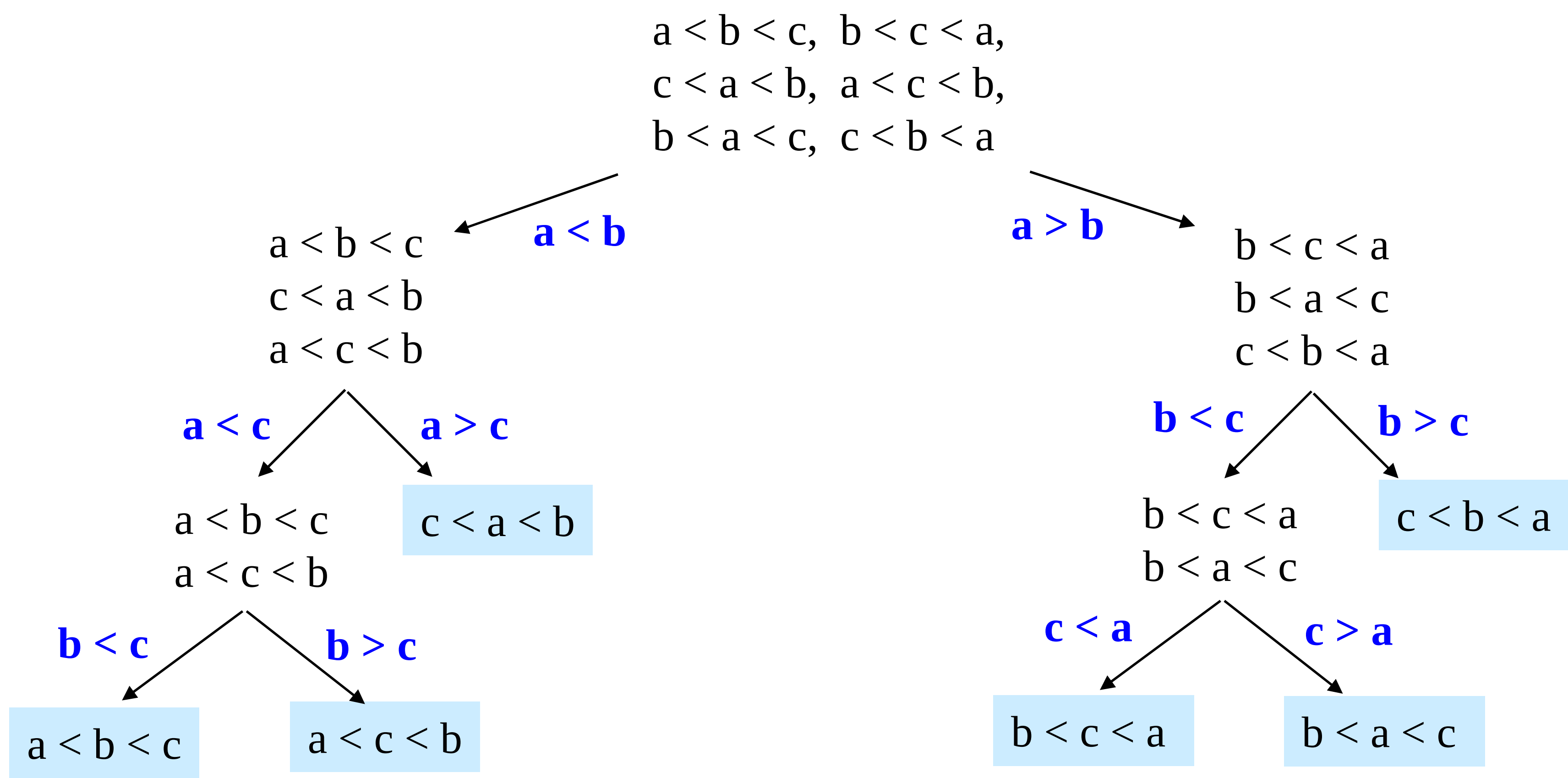
Sorting Model

- Recall our basic assumption: we can <u>only compare two elements at a time</u>
  - we can only reduce the possible solution space by half each time we make a comparison
- Suppose you are given N elements
  - Assume no duplicates
- How many possible orderings can you get?
  - Example: a, b, c (N = 3)

Permutations

- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
  - (a b c), (a c b), (b a c), (b c a), (c a b), (c b a)
  - 6 orderings = $3 \cdot 2 \cdot 1 = 3!$ (ie, “3 factorial”)
  - All the possible permutations of a set of 3 elements
- For N elements
  - N choices for the first position, (N-1) choices for the second position, …, (2) choices, 1 choice
  - $N(N-1)(N-2)\cdots(2)(1) =$ N! possible orderings

Decision Tree

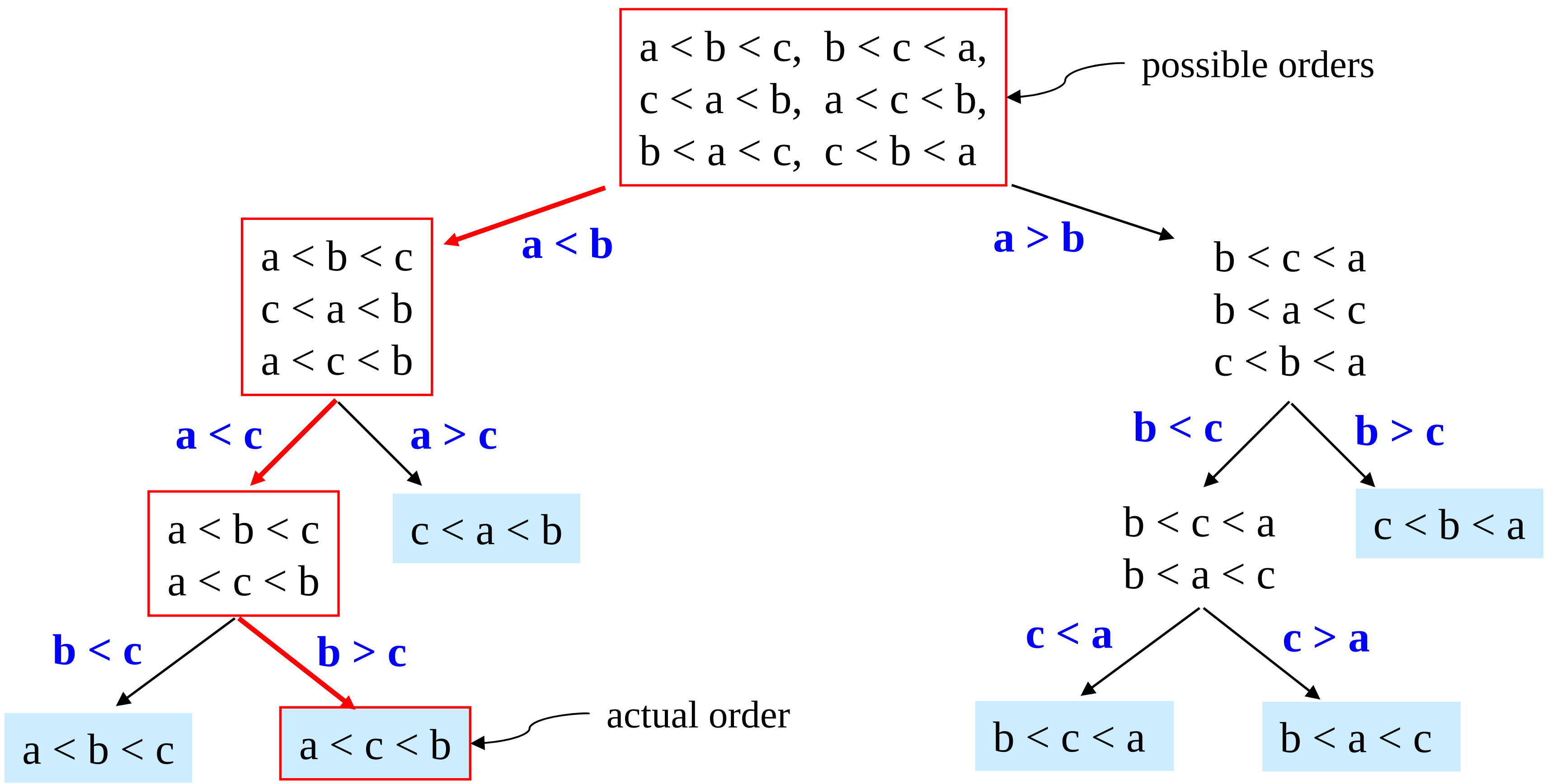


The leaves contain all the possible orderings of a, b, c

Decision Trees

- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
    - ie, the remaining solution space
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
  - How many leaves for N distinct elements?
    - N!, ie, a leaf for each possible ordering
- Only 1 leaf has the ordering that is the desired correctly sorted arrangement

Decision Tree Example



Decision Trees and Sorting

- Every sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
    - ie, by making comparisons
  - Each decision reduces the possible solution space by one half
- Run time is $\geq$ maximum no. of comparisons
  - maximum number of comparisons is the length of the longest path in the decision tree, i.e. the height of the tree

Lower bound on Height

- A binary tree of height h has **at most** *how many* leaves?

$$L \leq 2^h$$

- The decision tree has how many leaves:

$$L = N!$$

- A binary tree with L leaves has height **at least**:

$$h \geq \log_2 L$$

- So the decision tree has height:

$$h \geq \log_2(N!)$$

$\log(N!)$ is $\Omega(N\log N)$

$$\log(N!) = \log\big(N\cdot(N-1)\cdot(N-2)\cdots(2)\cdot(1)\big)$$

$$= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1$$

select just the first N/2 terms

$$\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log\frac{N}{2}$$

each of the selected terms is ≥ logN/2

$$\geq \frac{N}{2}\log\frac{N}{2}$$

$$\geq \frac{N}{2}(\log N - \log 2) = \frac{N}{2}\log N - \frac{N}{2}$$

$$= \Omega(N\log N)$$

Ω(N log N)

- Run time of any comparison-based sorting algorithm is **Ω(N log N)**
- Can we do better if we don’t use comparisons?

BucketSort (aka BinSort)

If all values to be sorted are *known* to be between 1 and *K*, create an array `count` of size *K*, **increment** counts while traversing the input, and finally output the result.

Example *K*=5. Input = (5,1,3,4,3,2,1,1,5,4,5)

`count` array	
1	
2	
3	
4	
5	

Running time to sort n items?

BucketSort Complexity: O(n+K)

- Case 1: K is a constant
  - BinSort is linear time
- Case 2: K is variable
  - Not simply linear time
- Case 3: K is constant but large (e.g. 2^{32})
  - ???

Fixing impracticality: RadixSort

- Radix = “The base of a number system”
  - We’ll use 10 for convenience, but could be anything

- Idea: BucketSort on each **digit**,
  least significant to most significant
  (lsd to msd)

Radix Sort Example (1st pass)

Bucket sort
by 1's digit

Input data

478
537
9
721
3
38
123
67

0	1	2	3	4	5	6	7	8	9
	72<u>1</u>		<u>3</u> 12<u>3</u>				53<u>7</u> 6<u>7</u>	47<u>8</u> 3<u>8</u>	<u>9</u>

After 1st pass

721
3
123
537
67
478
38
9

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.

Radix Sort Example (2nd pass)

After 1st pass

721
3
123
537
67
478
38
9

Bucket sort by 10's digit

0	1	2	3	4	5	6	7	8	9
03 09		721 123	537 38			67	478		

After 2nd pass

3
9
721
123
537
38
67
478

Radix Sort Example (3^{rd} pass)

After 2^{nd} pass

3
9
721
123
537
38
67
478

Bucket sort by 100's digit

0	1	2	3	4	5	6	7	8	9
003 009 038 067	123			478	537		721		

After 3^{rd} pass

3
9
38
67
123
478
537
721

Invariant: after k passes the low order k digits are sorted.

Your Turn

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

Radixsort: Complexity

- How many passes?
- How much work per pass?
- Total time?
- Conclusion?
- In practice
  - RadixSort only good for large number of elements with relatively small values
  - Hard on the cache compared to MergeSort/QuickSort

Summary of sorting

- Sorting choices:
  - $O(N^2)$ – Bubblesort, Insertion Sort
  - $O(N \log N)$ average case running time:
    - Heapsort: In-place, not stable.
    - Mergesort: $O(N)$ extra space, stable.
    - Quicksort: claimed fastest in practice, but $O(N^2)$ worst case. Needs extra storage for recursion. Not stable.
  - $O(N)$ – Radix Sort: fast and stable. Not comparison based. Not in-place.