CSE 373
Data Structures & Algorithms

Lecture 15

Sorting (III)

Sorting Recap

- Selection Sort
- Bubble Sort
- Insertion Sort
- Shell Sort

Sorting Recap

- Heapsort
- Mergesort
  - Does the sorting bottom-up
- Quicksort
  - Does the sorting top-down, as part of recursion

MergeSort and Massive Data

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

How fast can we sort?

Heapsort and Mergesort both have
O(*N* log *N*) **worst** case running time.

Heapsort, Mergesort, and Quicksort also have
O(*N* log *N*) **average** case running time.

Can we do any better?

Permutations

- How many possible orderings are there?

- Example: a, b, c

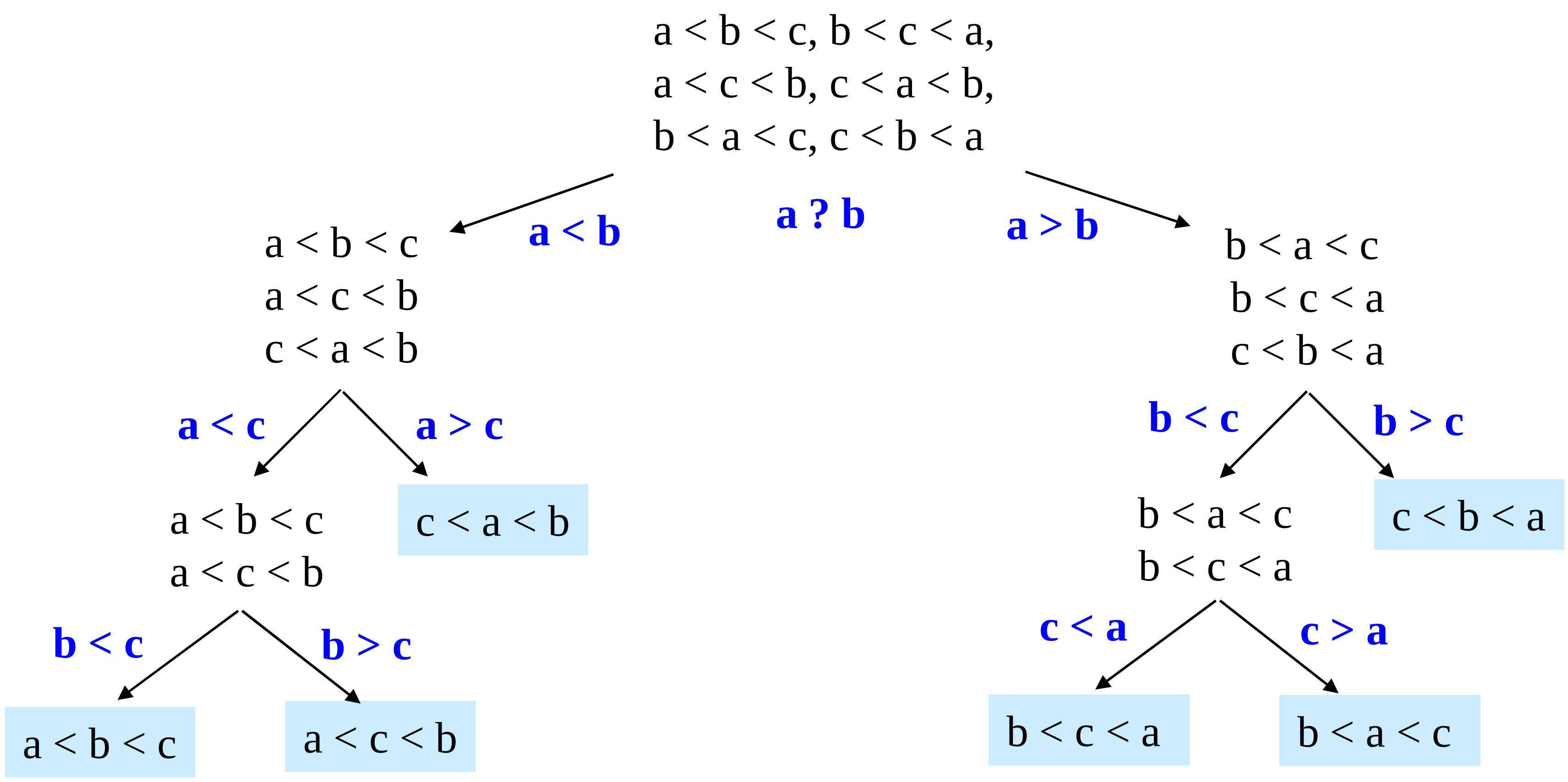
$a < b < c$	$b < a < c$	$c < a < b$
$a < c < b$	$b < c < a$	$c < b < a$

Permutations

- For 3 elements
  - 6 orderings = 3∙2∙1 = 3! (i.e., "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). . . (2)(1)= <u>*N*! possible orderings</u>

Decision Tree

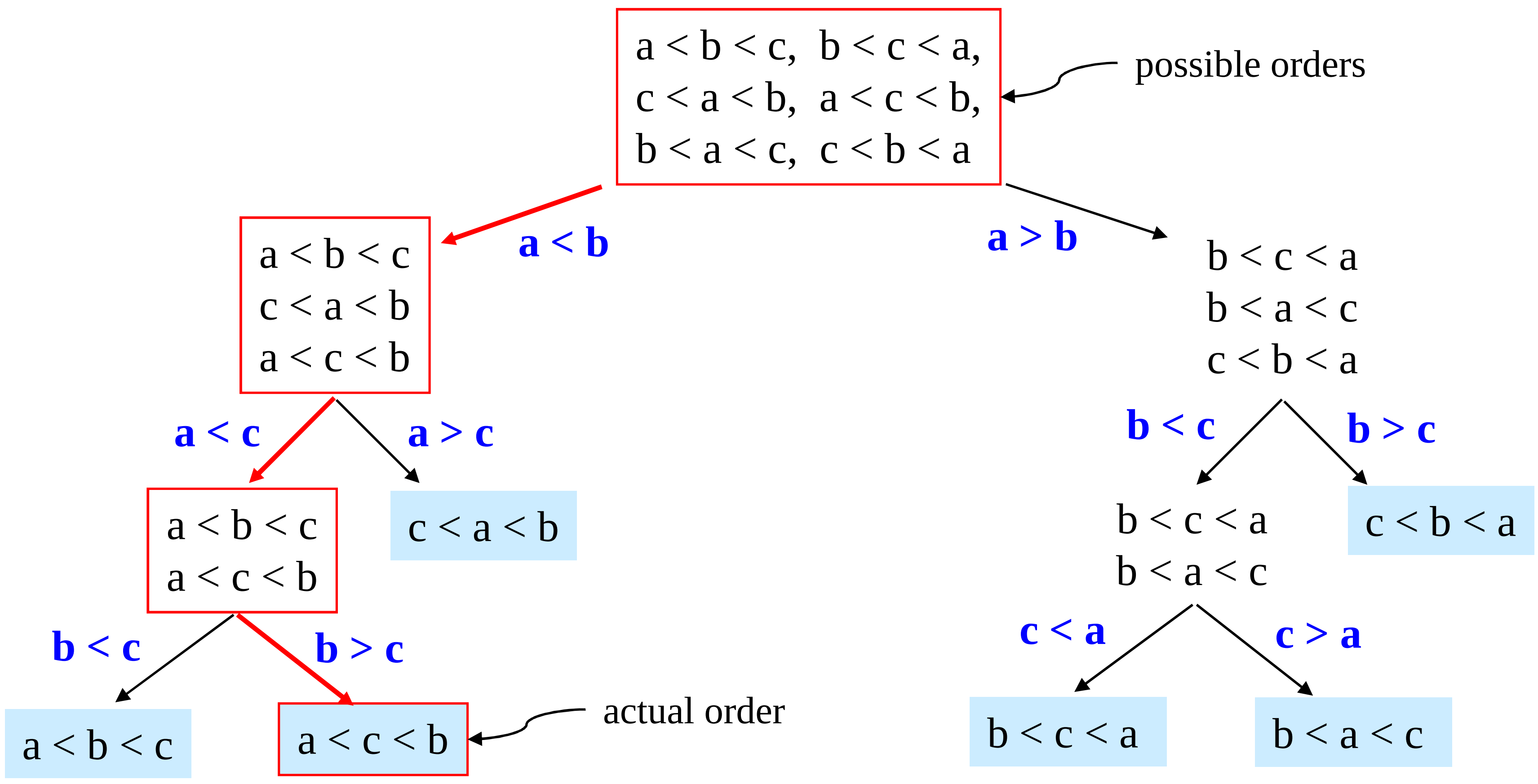


The leaves contain all the possible orderings of a, b, c

Decision Trees

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

Decision Tree Example

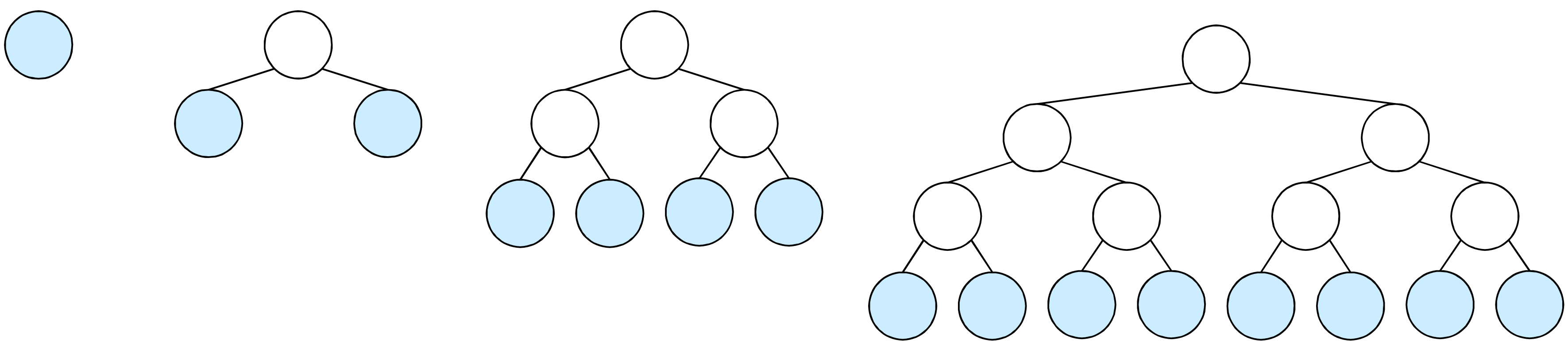


Decision Trees and Sorting

- Every sorting algorithm corresponds to this type of decision tree
- Finds correct leaf by following edges
  - In other words, by making comparisons
- Think about worst case run time
  - Worst case $\geq$ maximum comparisons
  - Maximum comparisons is the length of the longest path in the decision tree
  - Which is the height of the tree.

How many leaves on a tree?

- Suppose you have a binary tree of height h.
  How many leaves in a perfect tree?



- We can prune a perfect tree to make any binary tree of same height. Can # of leaves increase?

Lower bound on Height

- 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:

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

$\log(N!)$

$\log(N!)$

log(*N*!)

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

$\log(N!)$

$$\begin{aligned}\log(N!) &= \log\big(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)\big) \\ &= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1\end{aligned}$$

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

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

log(*N*!)

$$\log(N!) = \log\big(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)\big)$$
$$= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1$$
$$\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log \frac{N}{2}$$
$$\geq \frac{N}{2} \log \frac{N}{2}$$

select just the first N/2 terms

each of the selected terms is ≥ logN/2

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

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

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

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

each of the selected terms is ≥ logN/2

BucketSort (aka BinSort)

If all values are *known* to be between 1 and *K,*

BucketSort (aka BinSort)

If all values are *known* to be between 1 and *K*, create an array **count** of size *K*,

BucketSort (aka BinSort)

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

BucketSort (aka BinSort)

If all values 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	

BucketSort (aka BinSort)

If all values 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	1

BucketSort (aka BinSort)

If all values 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	1
2	
3	
4	
5	1

BucketSort (aka BinSort)

If all values 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	1
2	
3	1
4	
5	1

BucketSort (aka BinSort)

If all values 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	1
2	
3	1
4	1
5	1

BucketSort (aka BinSort)

If all values 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	1
2	
3	2
4	1
5	1

BucketSort (aka BinSort)

If all values 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	1
2	1
3	2
4	1
5	1

BucketSort (aka BinSort)

If all values 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
2	1
3	2
4	1
5	1

BucketSort (aka BinSort)

If all values 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	3
2	1
3	2
4	1
5	1

BucketSort (aka BinSort)

If all values 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	3
2	1
3	2
4	1
5	2

BucketSort (aka BinSort)

If all values 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	3
2	1
3	2
4	2
5	2

RadixSort

- Radix = “The base of a number system”
  - We’ll use 10 for convenience
  - Use a larger number in any implementation
  - ASCII Strings, for example, might use 128
- Idea:
  - BucketSort on one digit at a time
    - Requires stable sort!
  - After sort k, the last k digits are sorted
  - Set number of buckets: B = radix.

Radix Sort Example (1st pass)

Input data

478
537
9
721
3
38
123
67

Bucket sort
by 1's digit

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

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 (2^{nd} pass)

After 1^{st} 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
<u>0</u>3 <u>0</u>9		7<u>2</u>1 1<u>2</u>3	5<u>3</u>7 <u>3</u>8			<u>6</u>7	4<u>7</u>8		

After 2^{nd} pass

3
9
721
123
537
38
67
478

Radix Sort Example (3rd pass)

After 2nd 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 3rd pass

3
9
38
67
123
478
537
721

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

RadixSort

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

BucketSort on lsd:

	341 131					126 636 416		328 328	
0	1	2	3	4	5	6	7	8	9

BucketSort on next-higher digit:

	416	126 328 328	131 636	341					
0	1	2	3	4	5	6	7	8	9

BucketSort on msd:

	126 131		328 328 341	416		636			
0	1	2	3	4	5	6	7	8	9

Output: 126, 131, 328, 328, 341, 416, 636

Radixsort: Complexity

In our examples, we had:

- Input size, N
- Number of buckets, B = 10
- Maximum value, $M < 10^3$
- Number of passes, P = 3

How much work per pass?

Same as BucketSort!

O(B + N)

Total time?

O(P * (B + N))

Choosing the Radix

Run time is roughly proportional to:

$$P(B+N) = \log_B M (B+N)$$

Can show that this is minimized when:

$$B \log_e B \approx N$$

In theory, then, the best base (radix) depends only on N.

For fast computation, prefer $B = 2^b$. Then best b is:

$$2^b \log_e 2^b \approx N \quad \rightarrow \quad b + \log_2 b \approx \log_2 N$$

Example:

- N = 1 million (i.e., ~2^{20}) 64 bit numbers, $M = 2^{64}$
- $\log_2 N \approx 20 \rightarrow b = 16$
- $B = 2^{16}$ = 65,536 and $P = \log_{(2^{16})} 2^{64} = 4$.

In practice, memory word sizes, space, other architectural considerations, are important in choosing the radix.

Summary of sorting

$O(N^2)$ average, worst case:

- **Selection Sort**, **Bubblesort**, **Insertion Sort**

$O(N \log N)$ worst case:

- **Heapsort**: In-place, not stable.
- **Mergesort**: $O(N)$ extra space, stable, massive data.
- **Quicksort**: $O(N \log N)$ average case; claimed fastest in practice, but $O(N^2)$ worst case. Recursion/stack requirement. Not stable.

$\Omega(N \log N)$ worst and average case:

- **Any comparison-based sorting algorithm**

$O(N)$

- **Radix Sort**: fast and stable. Not comparison based. Not in-place. Poor memory locality can undercut performance. If N distinct keys, then each has $O(\log N)$ bits: back to $O(N \log N)$