CSE 332
Data Structures & Parallelism

Beyond Comparison Sorting

Melissa Winstanley

Spring 2024

Sorting: The Big Picture

Simple algorithms: $O(n^2)$

Insertion sort
Selection 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

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?

$n!$

- 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

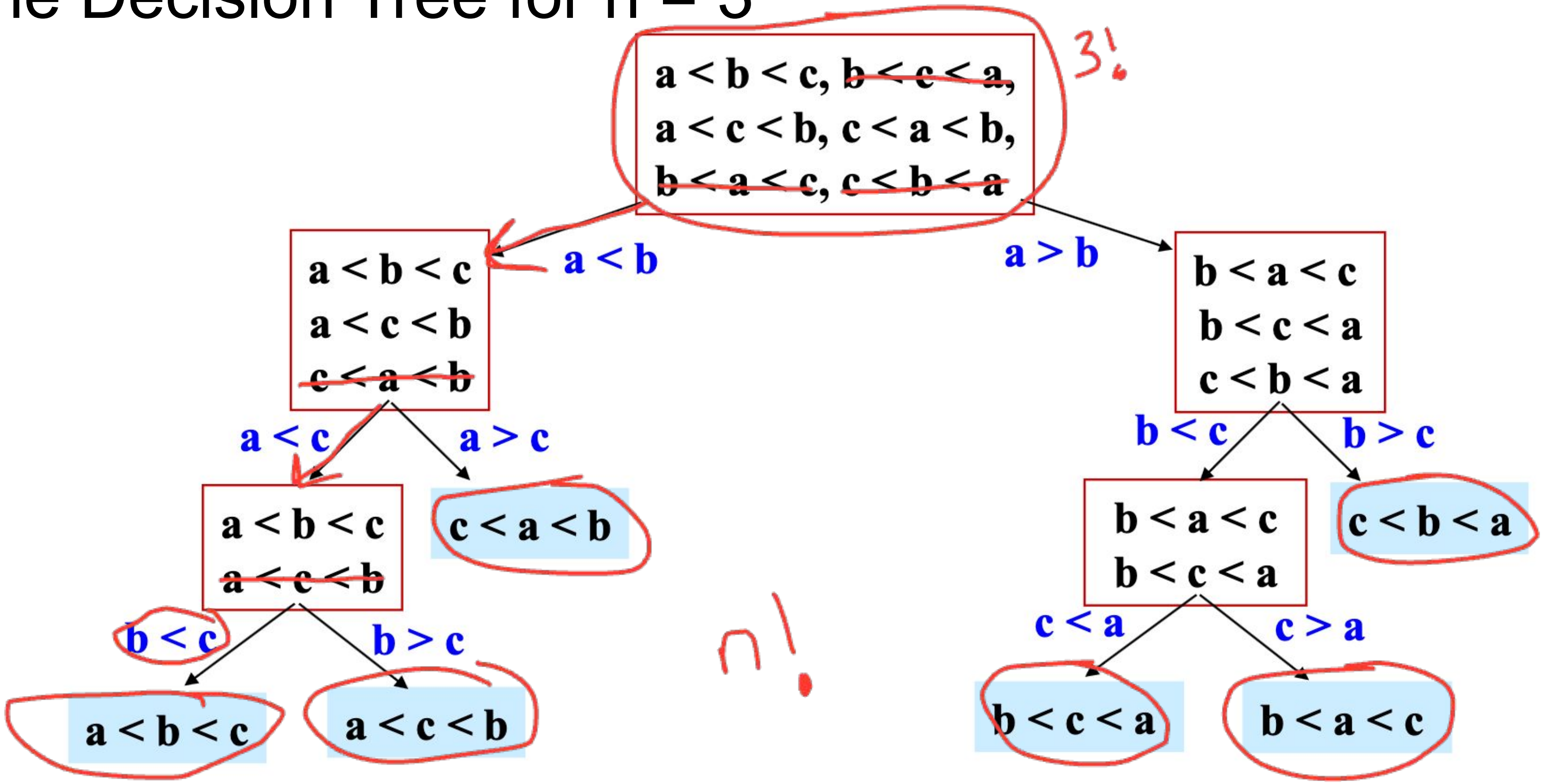
    a[0]<a[1]<a[2] a[0]<a[2]<a[1] a[1]<a[0]<a[2]
    a[1]<a[2]<a[0] a[2]<a[0]<a[1] a[2]<a[1]<a[0]

- In general, n choices for least element, then n-1 for next, then n-2 for next, …
    - $n(n-1)(n-2)\dots(2)(1) = \boxed{n!}$ possible orderings

Counting Comparisons

- Don’t know what the algorithm is, but it cannot make progress without doing comparisons
  - Eventually does a first comparison “is a < b ?"
  - Can use the result to decide what second comparison to do
  - Etc.: comparison k can be chosen based on first k-1 results
- Can represent this process as a **decision tree**
  - Nodes contain “set of remaining possibilities”
  - At root, anything is possible; no option eliminated
  - Edges are “answers from a comparison”
  - The algorithm does not actually build the tree; it’s what our ***proof*** uses to represent “the most the algorithm could know so far” as the algorithm progresses

One Decision Tree for n = 3



- The leaves contain all the possible orderings of a, b, c
- A different algorithm would lead to a different tree

Lower bound on Height

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

  $L \leq$ 2^h

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

  $h \geq$ $\log_2 L$

- The decision tree has how many leaves: $n!$

- So the decision tree has height:

  $h \geq$ $\log_2 n!$

Lower bound on height

- The height of a binary tree with L leaves is at least log2 L
- So the height of our decision tree, h:

$h \geq \mathbf{log}_2(n!)$	property of binary trees
$= \mathbf{log}_2(n*(n-1)*(n-2)\ldots(2)(1))$	definition of factorial
$= \mathbf{log}_2 n + \mathbf{log}_2(n-1) + \ldots + \mathbf{log}_2(1)$	property of logarithms
$\geq \mathbf{log}_2 n + \mathbf{log}_2(n-1) + \ldots + \mathbf{log}_2(n/2)$	keep first n/2 terms
$\geq (n/2)\,\mathbf{log}_2(n/2)$	each of the n/2 terms left is $\geq \log_2(n/2)$
$= (n/2)(\mathbf{log}_2\, n - \mathbf{log}_2\, 2)$	property of logarithms
$= (1/2)\, n\, \mathbf{log}_2 n - (1/2)n$	arithmetic
"=" $\Omega\,(n\, \mathbf{log}\, n)$	

Sorting: 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) ...	How??? Change the model - assume more than "compare(a,b)"	Bucket sort Radix sort	External sorting

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, no need to store more than a *count* of how many times that bucket has been used
- Output result via linear pass through array of buckets

`count` array	
1	III
2	I
3	II
4	II
5	III

- Example:

  $\boxed{K=5}$

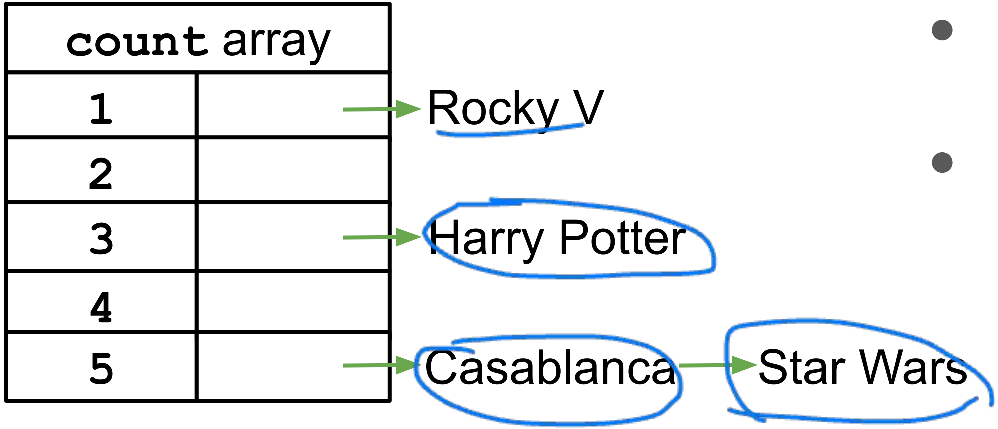
  Input: (5,1,3,4,3,2,1,1,5,4,5)

  Output: (1 1 1 2 3 3 4 4 5 5 5)
- Runtime: Step 1: n

  $\boxed{\text{Step 2: } n+K}$

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 O(1) (keep pointer to last element)



- Example: Movie ratings: 1=bad,… 5=excellent
- Input=

  5: Casablanca
  3: Harry Potter movies
  1: Rocky V
  5: Star Wars

Analyzing bucket sort

Performance depends on:

- Input size: **n**
- Number of buckets: **K**

- Work to put the data in buckets: n
- Work to pull data out of the buckets: n + K
- Overall: O(n + k)

Radix sort

976138

- 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
- **Invariant**: 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

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

First pass:

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

- List is sorted by first digit

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

Order now:

3
9
721
537
38
143
67
478

Second pass:

stable bucket sort by tens digit

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 tens digit

We're done!

Order now:

3
9
38
67
143
478
537
721

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: n+B
  - Each pass is a Bucket Sort
- Total work is P(n+B)
  - We do 'P' passes, each of which is a Bucket Sort

Recap: 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

Sorting massive data: External Sorting

Need sorting algorithms that **minimize disk 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

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)

- Mergesort can leverage multiple disks
- Weiss gives some examples

Sorting Summary

- 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
- $\Omega(n \log n)$ 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!