Beyond Comparison Sorts; Intro to Multithreading

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- A binary tree of height h has at most how many leaves?
  $L \leq$ _______________

- A binary tree with L leaves has height at least:
  $h \geq$ _______________

- A decision tree has how many leaves: ________

Announcements

- 🎉🎉🎉 No quiz this week! 🎉🎉🎉

- Just one checkpoint ☺

Lecture Outline

- Comparison-based Sorting
  - **Theoretical lower bound**

- Beyond Comparison Sorts
  - BucketSort
  - RadixSort

- Sorting Conclusion

- Changing Another Major Assumption
  - Definitions: Parallelism vs Concurrency

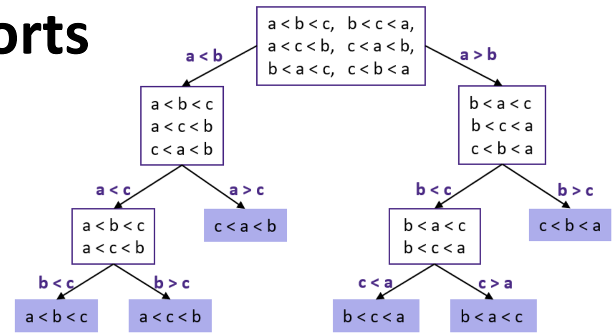
A Different View of Sorting

- Assume we have *n* elements, none are equal (ie, no duplicates)
  - ***n! permutations*** (possible orderings) of the elements. For *n*=3

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

- Assume an "OptimalSort"
  - Instead of describing how it ***works***, we'll describe what it ***knows and when it knows it***
  - Starts "knowing nothing"; "anything is possible"
  - Each binary: `a < b` or `b < a` comparison gains information, eliminating some possibilities
    - Each comparison eliminates (at most) half of remaining possibilities
  - In the end, narrows down to a single possibility

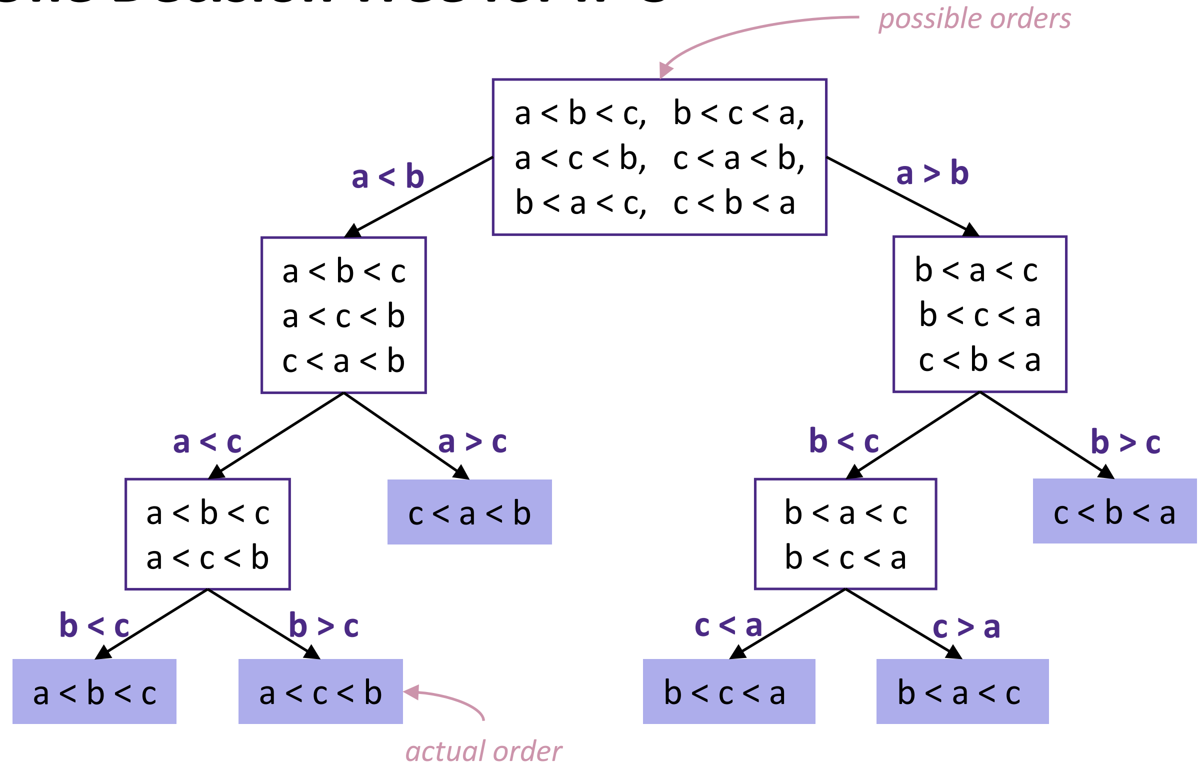
Representing Comparison Sorts



- Let's represent these binary comparisons as a binary tree!
- Called a *Decision Tree*
  - Nodes contain "set of remaining possible orderings"
  - The root contains all possible orderings; anything is possible
  - The leaves contain exactly one specific ordering
  - Edges are "answers from a comparison"

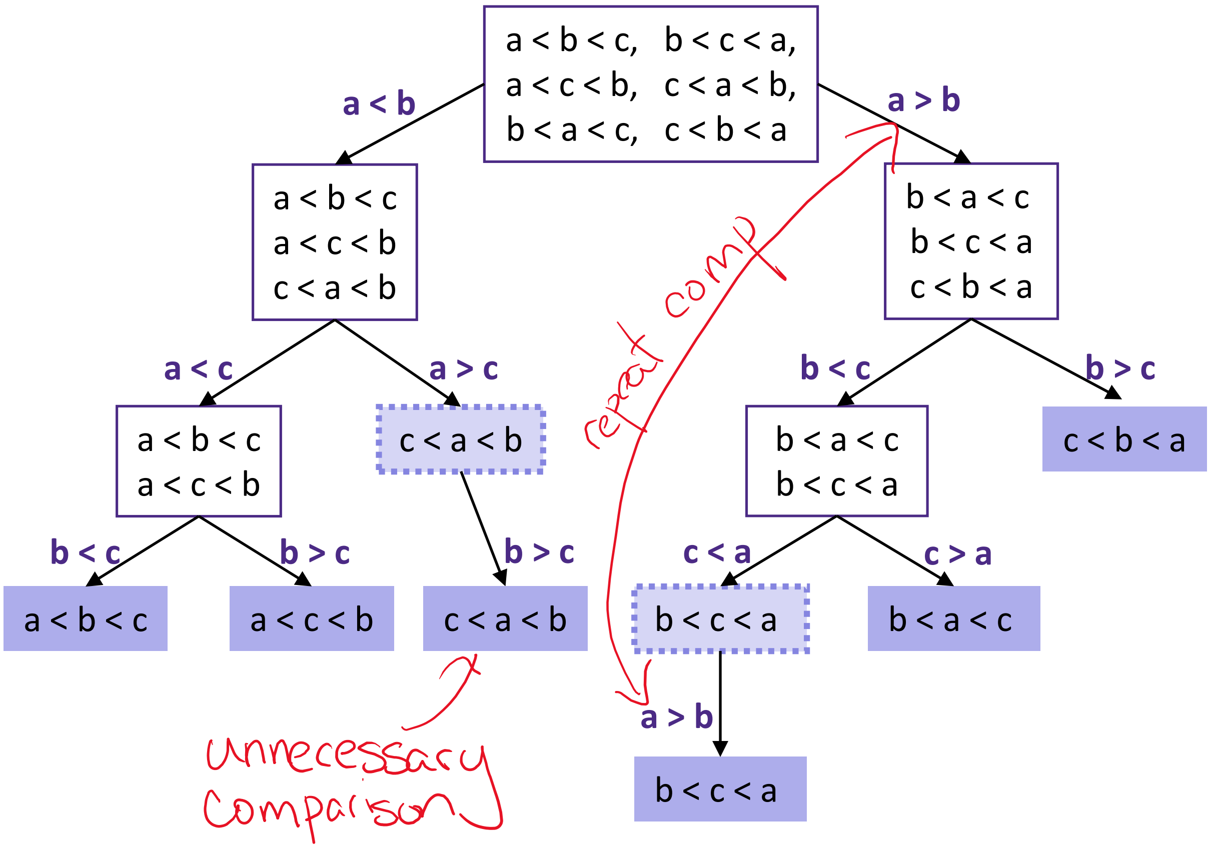
We are not actually building the tree; it's what our proof uses to represent "the most the algorithm could know so far"

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

Another Decision Tree for n=3



What the Decision Tree Tells Us

- Because any order is possible, any algorithm needs to ask enough questions to produce all $n!$ leaves (ie, orderings)
  - Each answer/ordering may lead to a different leaf
  - ***So the binary tree must be big enough to have n! leaves***

- Running *any* algorithm on *any* input will *at best* correspond to a root-to-leaf path in *some* decision tree with $n!$ leaves
  - Path length is the number of comparison operations needed
  - ***So no algorithm can have worst-case running time better than the height of a tree with n! leaves***
    - Because the worst-case number-of-comparisons for an algorithm is an input that yields to a longest path in algorithm's decision tree

Lower Bound on Height (1 of 2)

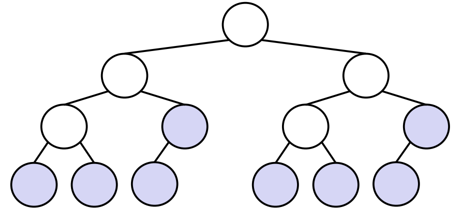
- A binary tree of height h has at most how many leaves?
  L ≤ 2^h

- 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 (2 of 2)



- 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 $\geq$ log2 (n/2)
$= (n/2)(\log_2 n - \log_2 2)$	property of logarithms
$= (1/2)n \log_2 n - (1/2)n$	arithmetic
"=" $\Omega (n \log n)$	

Lecture Outline

- Comparison-based Sorting
  - Theoretical lower bound
- Beyond Comparison Sorts
  - **BucketSort**
  - RadixSort
- Sorting Conclusion
- Changing Another Major Assumption
  - Definitions: Parallelism vs Concurrency

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*, put each element in its **bucket (a.ka. bin)**
  - If data is only integers, can store *count* of how many 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: 1,1,1,2,3,3,4,4,5,5,5

What is the running time?
How did the model change?

Analyzing BucketSort

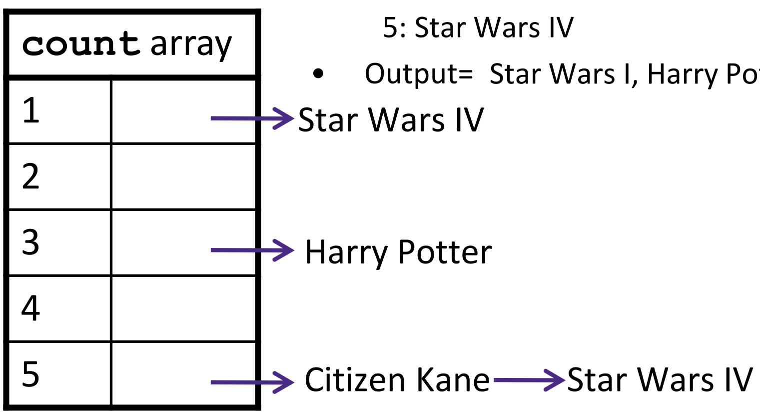
- Overall: $O(n+K)$
  - Linear in *n*, but also linear in *K*
  - $\Omega(n$ `log` $n)$ doesn' t apply because **this is not a comparison sort**

- Good when range, *K*, is smaller (or not much larger) than *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

BucketSort with Data

- Most real lists aren't just #'s; we have data too
  - Make 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: Citizen Kane
      3: Harry Potter movies
      1: Star Wars I
      5: Star Wars IV
    - Output= Star Wars I, Harry Potter movies, Citizen Kane, Star Wars IV



Lecture Outline

- Comparison-based Sorting
  - Theoretical lower bound
- Beyond Comparison Sorts
  - BucketSort
  - **RadixSort**
- Sorting Conclusion
- Changing Another Major Assumption
  - Definitions: Parallelism vs Concurrency

RadixSort

- Radix = “the base of a number system”
  - Examples will use 10 because we are used to that
  - Implementations may 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

RadixSort: Example (1 of 6)

0	1	2	3	4	5	6	7	8	9

Input:

333
143
591
65
332
491

First pass:

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

- List is sorted by first digit

RadixSort: Example (2 of 6)

0	1	2	3	4	5	6	7	8	9
	591 491	332	333 143		65				

Input:

333
143
591
65
332
491

First pass:

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

- List is sorted by first digit

Order is now:

591
491
332
333
143
65

gradescope

gradescope.com/courses/256241

0	1	2	3	4	5	6	7	8	9
	591 491	332	333 143		65				

Input:

333
143
591
65
332
491

Second pass:

1. BucketSort by **tens** digit, **stably**

Order is now:

0	1	2	3	4	5	6	7	8	9
	591 491	332	333 143		65				

0	1	2	3	4	5	6	7	8	9
			332 333	143		65			591 491

Input:

333
143
591
65
332
491

Second pass:

1. BucketSort by **tens** digit, **stably**

Notice: if we chop off the 100's place, these are now sorted

Order is now:

332
333
143
65
591
491

0	1	2	3	4	5	6	7	8	9
			332 333	143		065			591 491

0	1	2	3	4	5	6	7	8	9

Input:

333
143
591
65
332
491

Third pass:

1. BucketSort by **hundreds** digit, stably

Order is now:

0	1	2	3	4	5	6	7	8	9
			332 333	143		65			591 491

0	1	2	3	4	5	6	7	8	9
65	143		332 333						491 591

Input:

333
143
591
65
332
491

Third pass:

1. BucketSort by **<u>hundreds</u>** digit, stably

🎉 ***Only 3 digits; we're done!*** 🎉

Order is now:

65
143
332
333
491
591

Analysis of Radix Sort

- Performance depends on:
  - Input size: *n*
  - Number of buckets = Radix: *K*
    - 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 BucketSort: $O(K+n)$
  - *Each pass is a BucketSort!*
- Total work is $O(P\cdot(K+n))$
  - *We do 'P' passes, each of which is a BucketSort!*

Comparison to Comparison Sorts

- Compared to comparison sorts, radix sorts are 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

Lecture Outline

- Comparison-based Sorting
  - Theoretical lower bound
- Beyond Comparison Sorts
  - BucketSort
  - RadixSort
- **Sorting Conclusion**
- Changing Another Major Assumption
  - Definitions: Parallelism vs Concurrency

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.

Sorting: Summary (1 of 3)

- Simple $O(n^2)$ sorts can be fastest for small n
  - SelectionSort, InsertionSort:
    - The latter is linear for mostly-sorted!
    - Good for "below a cut-off" to help divide-and-conquer sorts
- "Fancy" $O(n \texttt{ log } n)$ sorts
  - HeapSort: not parallelizable
  - MergeSort: works as external sort
  - QuickSort: $O(n^2)$ in worst-case; cost of comparisons/copies often makes it fastest

Sorting : Summary (2 of 3)

- 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!

Sorting: Summary (3 of 3)

	Best-Case	Worst-Case	Randomized Case	In-Place?	Stable?	Notes
InsertionSort	$\Theta(N)$	$\Theta(N^2)$	$\Theta(N^2)$	Yes	Yes	Fastest for small or partially-sorted input
SelectionSort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^2)$	Yes	No	
In-Place HeapSort	$\Theta(N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Yes	No	Slow in practice
MergeSort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	No	Yes	Fastest stable sort
QuickSort *(1st-element pivot + 3-pass partition)*	$\Theta(N \log N)$	$\Theta(N^2)$	$\Theta(N \log N)$	No	Yes	>=2x slower than MergeSort
QuickSort *(Median-of-three pivot + Hoare partition + cutoffs)*	$\Omega(N)$	$O(N^2)$	$\Theta(N \log N)$	Yes	No	Fastest comparison sort
BucketSort	$\Theta(N+K)$	$\Theta(N+K)$	$\Theta(N+K)$	No	Yes	
RadixSort	$\Theta(P(B+N))$	$\Theta(P(B+N))$	$\Theta(P(B+N))$	No	Yes	

Lecture Outline

- Comparison-based Sorting
  - Theoretical lower bound
- Beyond Comparison Sorts
  - BucketSort
  - RadixSort
- Sorting Conclusion
- **Changing Another Major Assumption**
  - Definitions: Parallelism vs Concurrency

Sequential Programming: A Major Assumption

- So far, most / all of your study has assumed:

One thing happened at a time

- This is **sequential programming**: everything in one sequence

- Removing this assumption creates major challenges & opportunities
  - *Programming*: How to divide work among **threads of execution** and coordinate (**synchronize**) among them
  - *Algorithms*: How to utilize parallel activity to gain speed
    - More **throughput**: work done per unit time
  - *Data structures*: May need to support **concurrent access**
    - ie, multiple threads operating on data at the same time

A Simplified View of Computing History

- Writing *correct* and *efficient* multithreaded code is much more difficult than for sequential code
  - Especially in common languages like Java and C
- Roughly 1980-2005, computers got exponentially faster
  - Sequentially-written programs doubled in speed every couple years
  - So there was little motivation to write non-sequential code
- But nobody knows how to continue making computers faster
  - Increasing clock rate generates too much heat
  - Relative cost of memory access is too high
- But we *can* continue "making wires exponentially smaller" (**"Moore's 'Law'"**)
  - Result: multiple processors on the same chip ("**multicore**")

What to do with Multiple Processors/Cores?

- Next computer you buy will likely have 4 cores
  - Wait a few years and it will be 8, 16, 32, …
  - The chip companies have decided to do this (not a "law")
- What can you do with these processors?
  - Run multiple, totally different, programs at the same time
    - Already do that? It certainly *appears* that way, thanks to **time-slicing**
  - Run multiple, possibly different, tasks at the same time in one single program
    - Our focus for the next few lectures; it's more difficult!
    - Requires rethinking everything from asymptotic complexity to how to implement data-structure operations

Lecture Outline

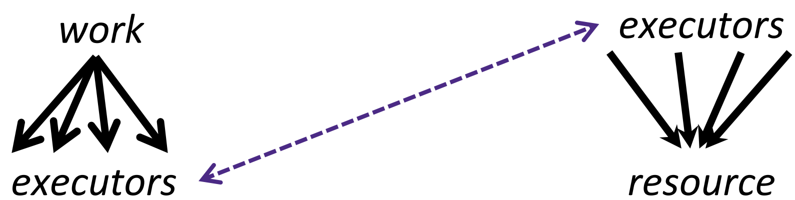
- Comparison-based Sorting
  - Theoretical lower bound
- Beyond Comparison Sorts
  - BucketSort
  - RadixSort
- Changing Another Major Assumption
  - **Definitions: Parallelism vs Concurrency**
- Shared Memory with Threads

Parallelism vs. Concurrency

- Note: Terms not yet standard but precision here is essential
  - (many programmers confuse these concepts)

Parallelism: Use extra executors to solve a problem faster

Concurrency: Manage access to shared resources



- There is some connection (confusion!) between them:
  - We commonly use threads for both parallelism and concurrency
  - If parallel computations access shared resources, the concurrency needs to be managed

Parallelism vs Concurrency: An Analogy

- **Sequential:** A program is like a cook making dinner
  - *One cook*: Makes gravy and stuffing one at a time!

efficiency

- **Parallelism**: *"Extra executors gets the job done faster!"*
  - *Multiple cooks*: One cook in charge of the gravy (and its onions), another in charge of the stuffing (and its onions)
    - Increase throughput via simultaneous execution!
    - Too many cooks means you spend all your time coordinating

correctness

- **Concurrency**: *"We need to manage a shared resource"*
  - *Multiple cooks*: One cook per dish, but only one cutting board
    - Correctness: Don't want spills or ingredient mixing
    - Efficiency: Who should use the boards and in what order?

Parallelism Example

- **Parallelism**: Using extra executors to solve a problem faster
- *Pseudocode* for summing an array:
  - No such 'FORALL' construct, but we'll see something similar
  - Bad style, but with 4 processors may get roughly 4x speedup

```
int sum(int[] arr) {
  int[] res = new int[4];
  int len = arr.length;
  FORALL (i=0; i < 4; i++) { // parallel iterations
    res[i] = sumRange(arr, i*len/4,(i+1)*len/4);
  }
  return res[0] + res[1] + res[2] + res[3];
}
int sumRange(int[] arr, int lo, int hi) {
  int result = 0;
  for(int j=lo; j < hi; j++)
    result += arr[j];
  return result;
}
```