Sorting

Dr. Baldassano

Yu's Elite Education

Last week recap

- Algorithm: procedure for computing something
- Data structure: system for keeping track for information – optimized for certain actions
- "Good" algorithms have time and memory requirements that scale slowly with data size
- "Big O" notation gives scaling behavior for most expensive part of algorithm
  - O(log n) or O(n) great!
  - $O(n^2)$ may be okay
  - $O(2^n)$ or O(n!) very bad!

Sorting

- Sorting: taking a bunch of objects and putting them in order
- Why do we care?
  - An important piece of many other algorithms
  - A good example of lots of algorithms concepts
  - We can prove that we've found the best possible sorting algorithms (in big O sense)

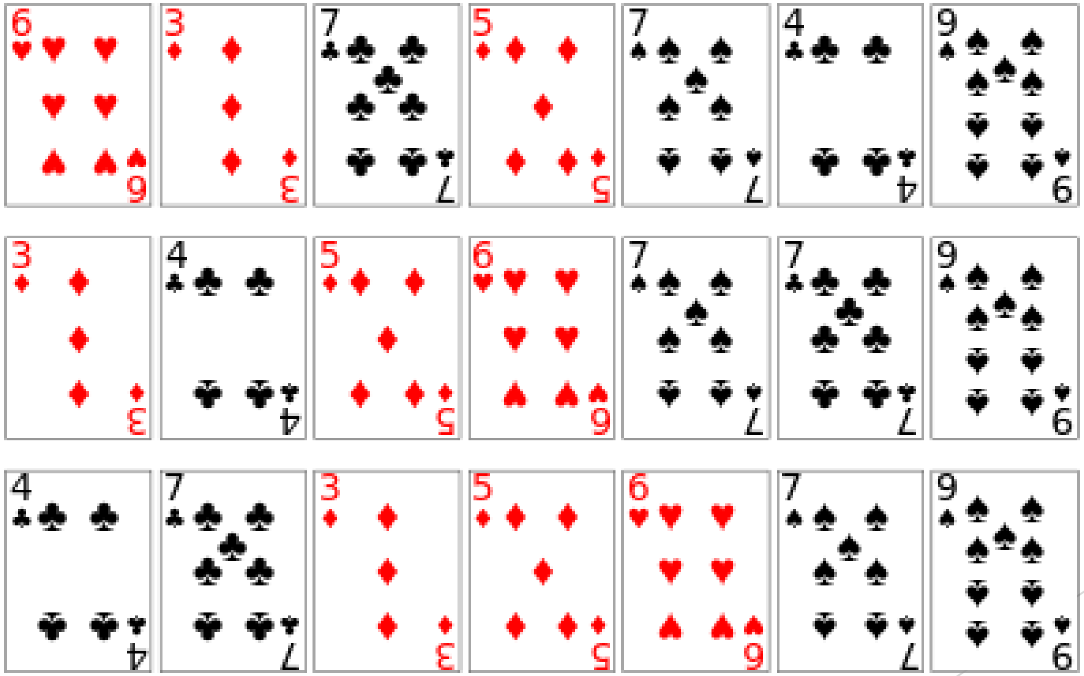
Last week's assignment

Stable sorting

- Sometimes the numbers we’re sorting are attached to a more complicated piece of data, so identical numbers correspond to different things
- Often want a sort to be stable: want identical numbers to remain in the same order after sorting

Stable sorting

- Sort first by number, then by suit
- Don't want second sort to mess up first one



Comparison Table

Name	Avg. Time	Memory	Stable?
Bubble	$O(n^2)$	O(1)	Yes

Things could be worse: Bogosort

- The stupidest possible sorting algorithm: randomly shuffle the items, then check to see if it is sorted
- There are O(n!) shuffles and each check takes O(n), so this has running time O(n*n!)
- Not stable
- At least it doesn’t require any extra memory!

Comparison Table

Name	Avg. Time	Memory	Stable?
Bubble	$O(n^2)$	O(1)	Yes
Bogo	O(n*n!)	O(1)	No

Insertion sort

- The most intuitive sorting algorithm
- One at a time, insert items into a sorted list on the left side

Insertion sort

- The most intuitive sorting algorithm
- One at a time, insert items into a sorted list on the left side

6 5 3 1 8 7 2 4

Big O of insertion sort

- Have to insert O(n) elements
- Will have to move O(n) elements on each insertion
- Avg running time $O(n^2)$, and stable
- In practice, insertion sort tends to be better than bubble sort
- Sometimes the very fastest sort for short lists (<10 elements)
- Variant called selection sort (more compares, fewer shifts)

Comparison Table

Name	Avg. Time	Memory	Stable?
Bubble	$O(n^2)$	O(1)	Yes
Bogo	O(n*n!)	O(1)	No
Insertion	$O(n^2)$	O(1)	Yes

Mergesort

- It’s easy to merge together two sorted lists:

  1 4 7 10 3 5 6 11

- “To sort a list, first sort the left side, then sort the right side, then merge the two lists together”
- This is a *recursive* sort, since mergesort will call itself on each half of the list

Mergesort

6 5 3 1 8 7 2 4

Big O for Mergesort

- There log(n) splitting levels
- Each element will have to be merged at each level
- Avg running time O(n * log(n))
- **BUT** requires extra O(n) memory
- Stable sort

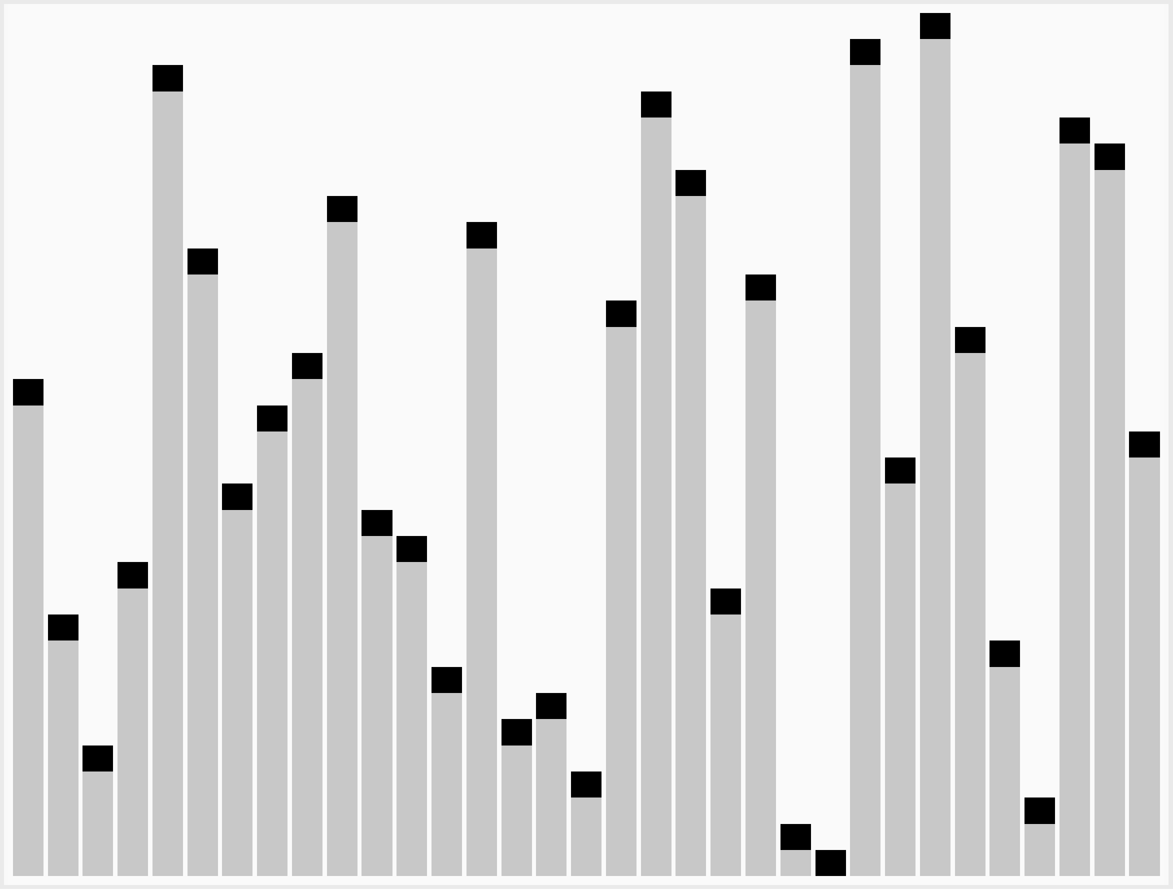
Comparison Table

Name	Avg. Time	Memory	Stable?
Bubble	$O(n^2)$	O(1)	Yes
Bogo	O(n*n!)	O(1)	No
Insertion	$O(n^2)$	O(1)	Yes
Merge	O(n*log(n))	O(n)	Yes

Quicksort

- Somewhat complicated, but probably the most common sorting algorithm used in practice
- Also recursive, but with opposite logic from mergesort:
- "To sort an array, first get the smaller items on the left and the larger items on the right, then sort the left and right arrays"
- Pick a "pivot" item to define small vs. large

Quicksort



Big O for Quicksort

- On average takes O(log n) splits, and each level of splitting looks at all O(n) items
- Avg running time O(n*log(n))
- Only requires O(log(n)) extra memory, to keep track of the recursive splits
- **BUT** not stable
  - Can be made stable, but requires some extra complexity and O(n) extra space
- Also, has $O(n^2)$ worst-case running time (if pivots are very unbalanced)

Comparison Table

Name	Avg. Time	Memory	Stable?
Bubble	$O(n^2)$	O(1)	Yes
Bogo	O(n*n!)	O(1)	No
Insertion	$O(n^2)$	O(1)	Yes
Merge	O(n*log(n))	O(n)	Yes
Quick	O(n*log(n))	O(log(n))	Depends

Real running time

	100	1,000	10,000	100,000	1,000,000
Bubble	0.050	5.93	445.92	44677.46	-
Insertion	0.015	1.72	126.41	12478.55	-
Merge	0.016	0.22	2.44	29.38	340.39
Quick	0.011	0.16	1.67	20.01	236.51

From: http://ddeville.me/2010/10/sorting-algorithms-comparison/

Visualizations

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

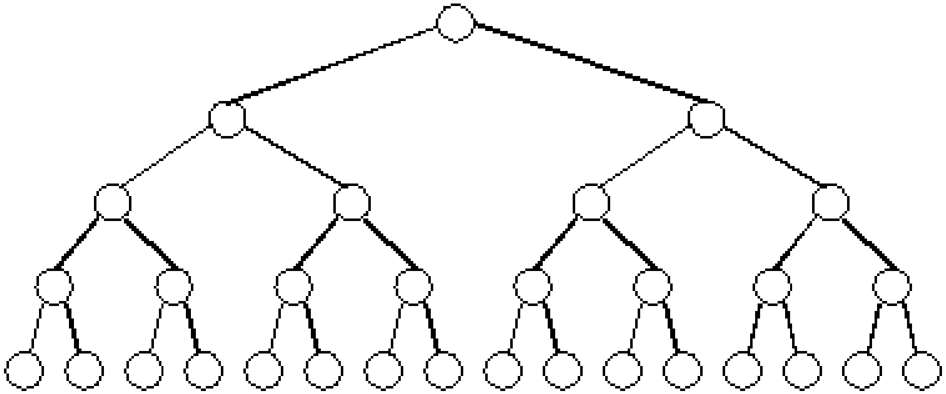
- http://bost.ocks.org/mike/algorithms/#sorting

Can we do better?

- Our best sorts are running in O(n log n)
- Is it possible to run faster?
- What is the minimum number of decisions a sorting algorithm needs to make?

Optimal sorting

- There are a total of n! possible ways to order a list – we need to pick one of these orders
- Every time we compare two numbers x,y in a sorting algorithm, we get one of two answers: x should go first, or y should go first
- D decisions -> 2^D possible outcomes



Optimal sorting

- We need $2^D = n!$
- Use Stirling's approximation: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
- Taking log of both sides, D = O(n log n)
- So any sort that works by making comparisons must have average running time at least O(n log n)

Doing the impossible

- Do a first pass, counting how many of each number there are
- Can then calculate where each number should go

Counting sort

- This is called counting sort
- This is a stable sort that runs in linear O(n) time!!

- How did we beat the theoretical bound?
- This is *not* a comparison sort – we never compare the items to one another

Non-comparison sorts

- Rather than comparing items, we directly calculate an item's position in the output list
- Catches:
  - The keys we're sorting need to come from a limited set
  - Requires O(n) extra space to store counting table and output array
- Comparison sorts are more general, requiring only some way to compare the items

Radix sort

- How to sort integers with 8 digits?
- Could use counting sort, with a huge table…
- Let’s use counting sort on each digit, repeating where necessary:

43028585	11474012	11474012
32820239	32820239	32820239
11474012	38572023	38572023
38572023	43028585	42581562
42581562	42581562	43028585

Radix sort

- Left-right: "Most Significant Digit" radix sort
- Have to keep track of create groups to sort within, and isn't stable
- Usually use "Least Significant Digit" radix sort, moving right to left

412	751	412	412
482	412	751	482
994	482	482	751
751	994	989	989
989	989	994	994

Big O for Radix Sort

- If number of digits is fixed, then we just need to do a fixed number of passes through n items
- Running time O(n)

- Works also for nonnumeric fixed-length sequences (e.g. fixed-length strings)

Sample problem

- Each of my friends is free for a different period of time on Saturday, e.g.
  - 10am-1pm for person 1,
  - 11am-5pm for person 2,
  - 9:30am–10:30pm for person 3
  - 12pm-4pm for person 4…
- What is the interval of time during which the most people are free?

One solution

- Convert to 24 hour time, and put all start and end times into a list, with each time tagged as start or end

  10S,13E, 11S, 17E, 9.5S, 10.5E, 12S, 16E

- Sort times using any sorting algorithm

  9.5S, 10S, 10.5E, 11S, 12S, 13E, 16E, 17E

- Move left to right, keeping track of #S - #E (this is the number of people free during this time)
- Whenever we reach a new maximum of #S - #E, record the current time, and set end time at next E
- Runs in O(n log n) time (or O(n) if times are integers)

Assignment: Mode of a list

- Generate a random list 10,000 integers between 1-100
- Write a program that finds the mode (the most common number) of the list
- What is the time complexity of your algorithm?
- Note: you can use an existing implementation of a sorting algorithm
- Also: vote for rescheduling December 10th class: December 7th (Mon) or 11th (Fri)