COMP_SCI 214: Data Structures and Algorithms

Searching and Sorting

PROF. SRUTI BHAGAVATULA

Announcements

- Self-eval 1 due tonight
- HW 1 resubmission due tonight

Announcements

- The two worksheets to be released late tonight (Canvas quizzes)
  - Complexity (Tuesday) and graphs (Chapter 10 of draft textbook)
  - Graphs quiz is to get you up to speed on fundamentals so we can go deeper in class
  - Instant feedback and 10 attempts for each; make sure to
    - understand why you got something wrong and learn from it
    - work on your own
    - not just guess until you get it right

Searching an array

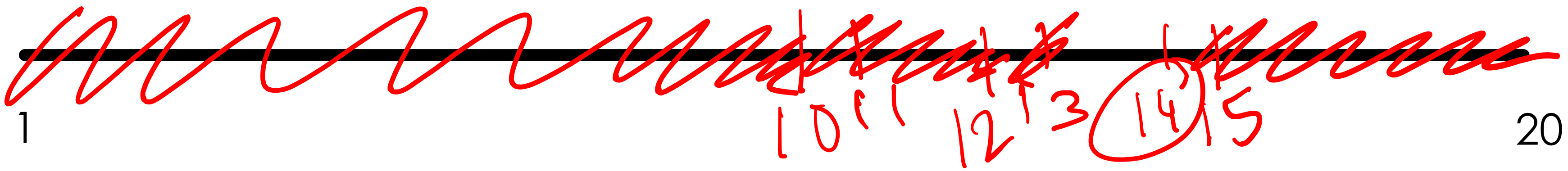
Linear search

- Search that takes linear time: O(n)
- Same complexity on sorted and unsorted arrays
- Worst case is $O(n)$
  - Average and best cases may be cheaper
  - **Usually care about worst case (but not always)**
- Can sortedness get us a better worst case cost?

```
def linear_search(numbers, target):
    for x in numbers:
        if x > target:
            return False
        if x == target:
            return True
    return False
```

Game: Guessing a number

- Your job: Think of a number between **[1, 20]**
- My job: guess your number in 5 guesses or less



- # steps to guess: 1 + 1 + 1 + 1 + 1

Game: Guessing a number (round 2)

- Your job: Think of a number between **[1, 20]**
- My job: guess your number in 5 guesses or less

1 20

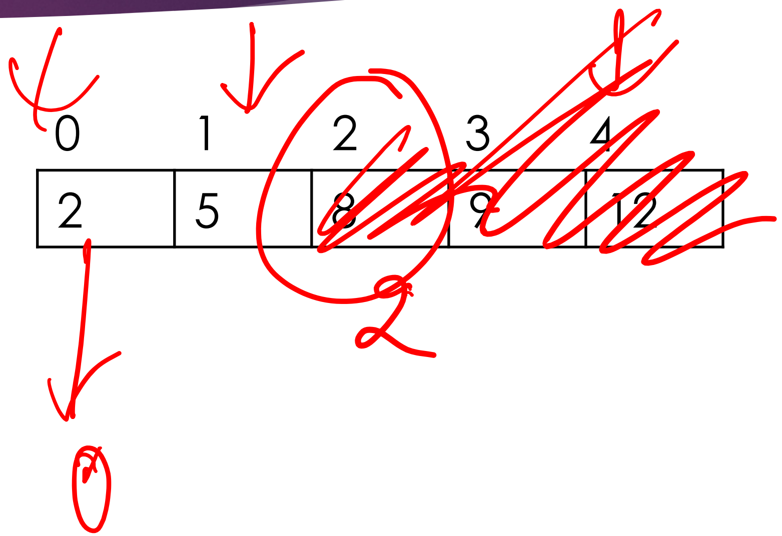
- # steps to guess:

Binary search

- Type of divide-and-conquer algorithm (or decrease-and-conquer)
- Search space is reduced by half each time
- #steps for worst case:
  - I guess 10. You tell me too high (+1 step)
  - I guess 5. You tell me too low. (+1 step)
  - I guess 7. You tell me too low. (+1 step)
  - I guess 8. You say too low. (+1 step)
  - I guess 9. You say that's correct! (+1 step)
  - Total guesses = $5 \approx \log_2 20$ (#times 20 can divide by 2 before reaching 0)

Binary search pseudocode

1. State: `start = 0` and `end = length-1`
2. Look for `midpoint` position in the array between start and end: `(end+start)/2`
3. Check if target is equal to, less than, or more than value at `midpoint`
   a. If equal: we found it, return true
   b. If less than: `end = midpoint - 1`
   c. If more than: `start = midpoint + 1`
   d. If `start` > `end`: `return false`
4. Repeat steps 2-3



Binary search code

```
def binary_search (numbers, target):
  # look for `target` between indices `low` and `high`
  def helper (low, high):
    # empty range -> not found
    if low > high: return False
    let mid = (low + high) // 2
    if numbers[mid] == target: return True
    elif numbers[mid] < target:
      return helper(mid+1, high)
    else: # numbers[mid] > target:
      return helper(low, mid-1)

  return helper(0, numbers.len()-1)
```

Binary search complexity

- Logarithmic worst-case complexity
  - $O(\log n)$
- Constant best-case complexity
  - $O(1)$ (midpoint of the whole array is the target)
  - Not what matters (best case is rare)

Another example: Looking for a word in a dictionary

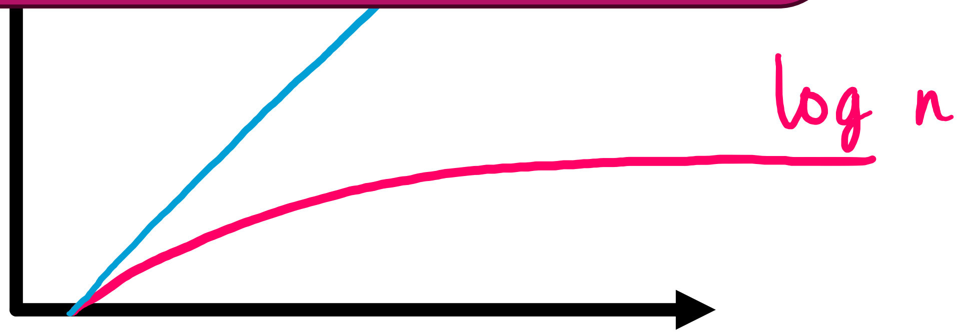
- Dictionary is sorted
- Say it 171,476 words

- Linear search would take a maximum of 171,476 steps
- Binary search takes a maximum of $\log_2 171{,}476 = 18$ steps

O(log n) is a big deal

n	10	100	1000	10K	100K	1M	10M	100M
$\log_2 n$							3.3	26.6

Considered as good as O(1) as n gets larger



Pre-condition for binary search

- Sorted array is a precondition for binary search
- We'll need a way to sort an array

Sorting

Many sorting algorithms

- Selection sort
- Bubble sort
- Merge sort
- Quicksort
- Insertion sort
- Heap sort
- Counting sort
- Radix sort

Each with their own time and space complexities and tradeoffs.

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

Many sorting algorithms

- Selection sort
- Bubble sort
- Merge sort
- Quicksort
- Insertion sort
- Heap sort (later in the quarter)
- Counting sort
- Radix sort

We'll look at some today

All achieve the same task (parallel to ADTS?)
- Sorting is like an abstract algorithm type
- The specific algorithm is the implementation

Characteristics of sorting algorithms

- In-place vs. out-of-place
- Stable vs. unstable
- Comparisons

What can we sort?

- Most collections of data
  - Arrays
  - Linked lists
  - Other collections?

Selection sort on an array (descending)

9	13	3	2	18

max

18	13	3	2	9

18	13	3	2	9

18	13	9	2	3

18 13 9 3 2

Selection sort complexity

1. For each element in the array at index i n
   1. Find the maximum element in the array from i to end $n-i$
   2. Swap element at index i with maximum element above $O(1)$

$$\frac{n + (n-1) + (n-2) + (n-3) \ldots\ldots 1}{}$$

$$= 1 + 2 + 3 + 4 \ldots\ldots n$$

$$= \frac{n(n+1)}{2} = \frac{n^2}{\cancel{2}} + \frac{1}{\cancel{2}} \in O(n^2)$$

Selection sort code for array

sort-selection-vec.rkt

```
def selection_sort (unsorted):
   for i in range(unsorted.len()-1):
      # not shown (see code)
      let max = find_largest_vec(unsorted, i)
      # not shown (see code)
      swap(unsorted, i, max)
```

Selection sort on a linked list

sort-selection.rkt

- Same complexity with a slightly different implementation
  - Code on Canvas

1. Create an empty list called “sorted”
2. Find and remove the minimum element in the unsorted linked list
3. Add removed element to end of “sorted”
4. Repeat steps 2-4 until unsorted array is empty

Many sorting algorithms

- ~~Selection sort: $O(n^2)$~~
- Bubble sort
- Merge sort
- Quicksort
- Insertion sort
- Heap sort (later in the quarter)
- Counting sort
- Radix sort

Merge sort

- Another divide-and-conquer algorithm

0	1	2	3	4	5
4	2	1	3	7	0

Vector notation for clarity; works even better with linked lists!

Merge sort

- Another divide-and-conquer algorithm

0	1	2	3	4	5
4	2	1	3	7	0

- Step 1: Split into two sub-lists (even and odd indices)

4	1	7

2	3	0

Merge sort

- Another divide-and-conquer algorithm

0	1	2	3	4	5
4	2	1	3	7	0

- Step 1: Split into two sub-lists (even and odd indices)

4	1	7

2	3	0

- Step 2: Recursively sort each sub-list

1	4	7

0	2	3

Merge sort

- Another divide-and-conquer algorithm

0	1	2	3	4	5
4	2	1	3	7	0

- Step 1: Split into two sub-lists (even and odd indices)

4	1	7

2	3	0

- Step 2: Recursively sort each sub-list

1	4	7

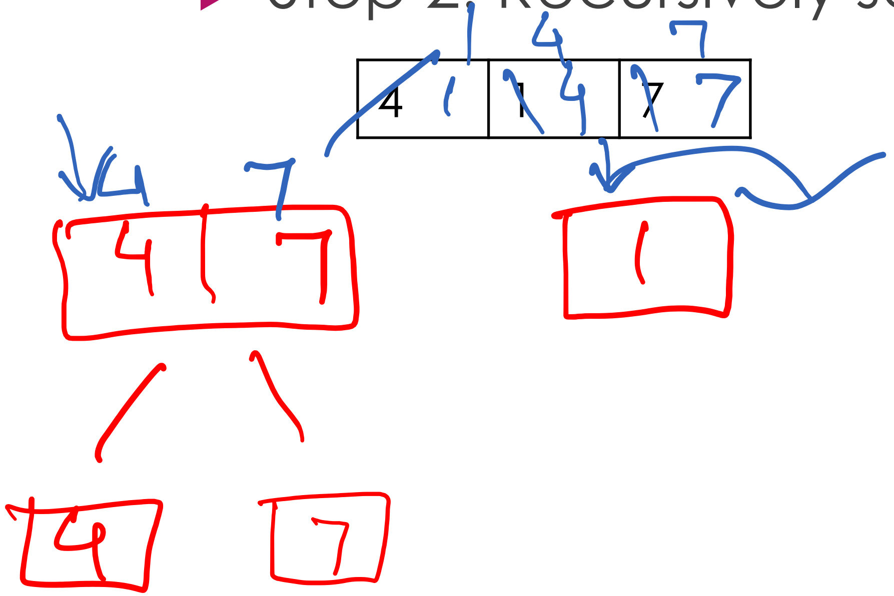
0	2	3

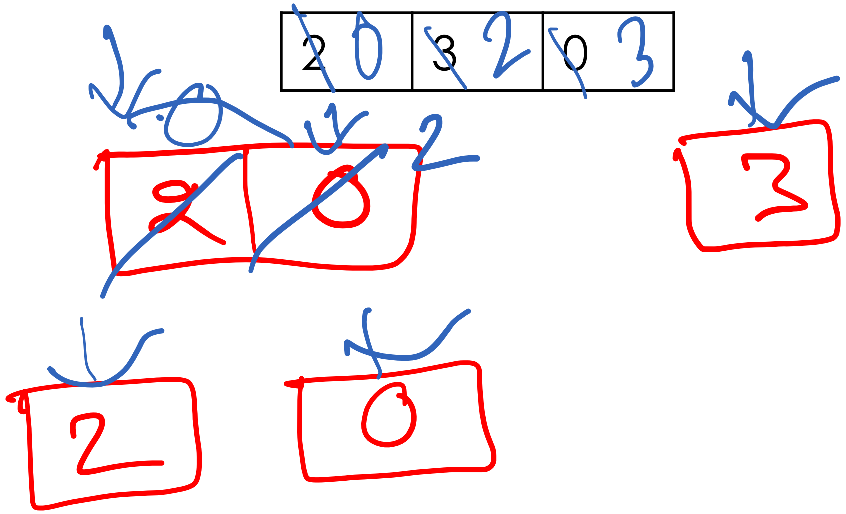
- Step 3: Merge the two sub-lists by looking at each element in both in-turn

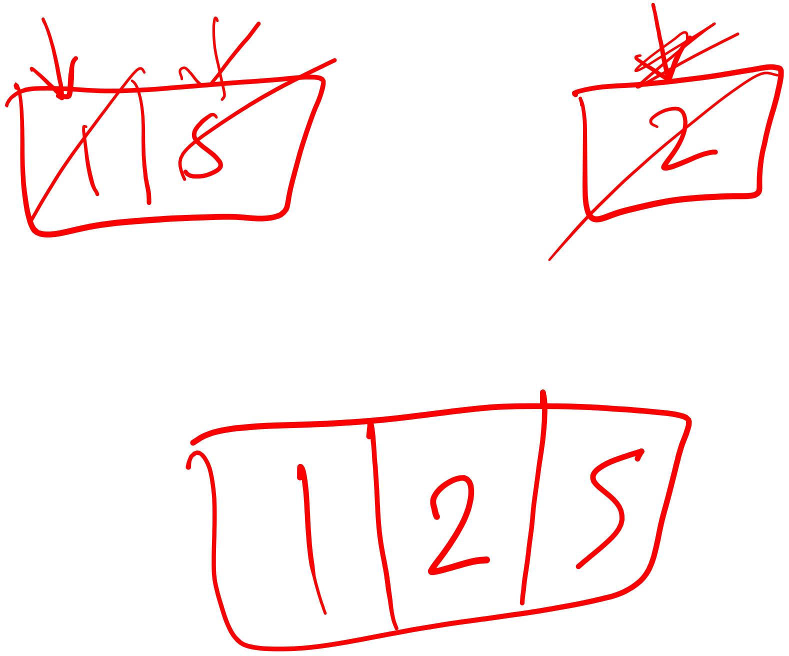
0	1	2	3	4	7

Breaking down Step 2

- Step 2: Recursively sort each sub-list






1 8
2
1 2 5

Merge sort code for a linked list

sort-merge.rkt

```
# : List[Number] -> List[Number]
def merge_sort(lst):
  # Base cases
  if lst is None or lst.next is None: return lst
  # Step 1: split into two sub-lists
  let o = odds(lst)
  let e = evens(lst)
  # Step 2: recursively sort each sub-list
  o = merge_sort(o)
  e = merge_sort(e)
  # Step 3: merge the two sub-lists
  return merge(o, e)
```

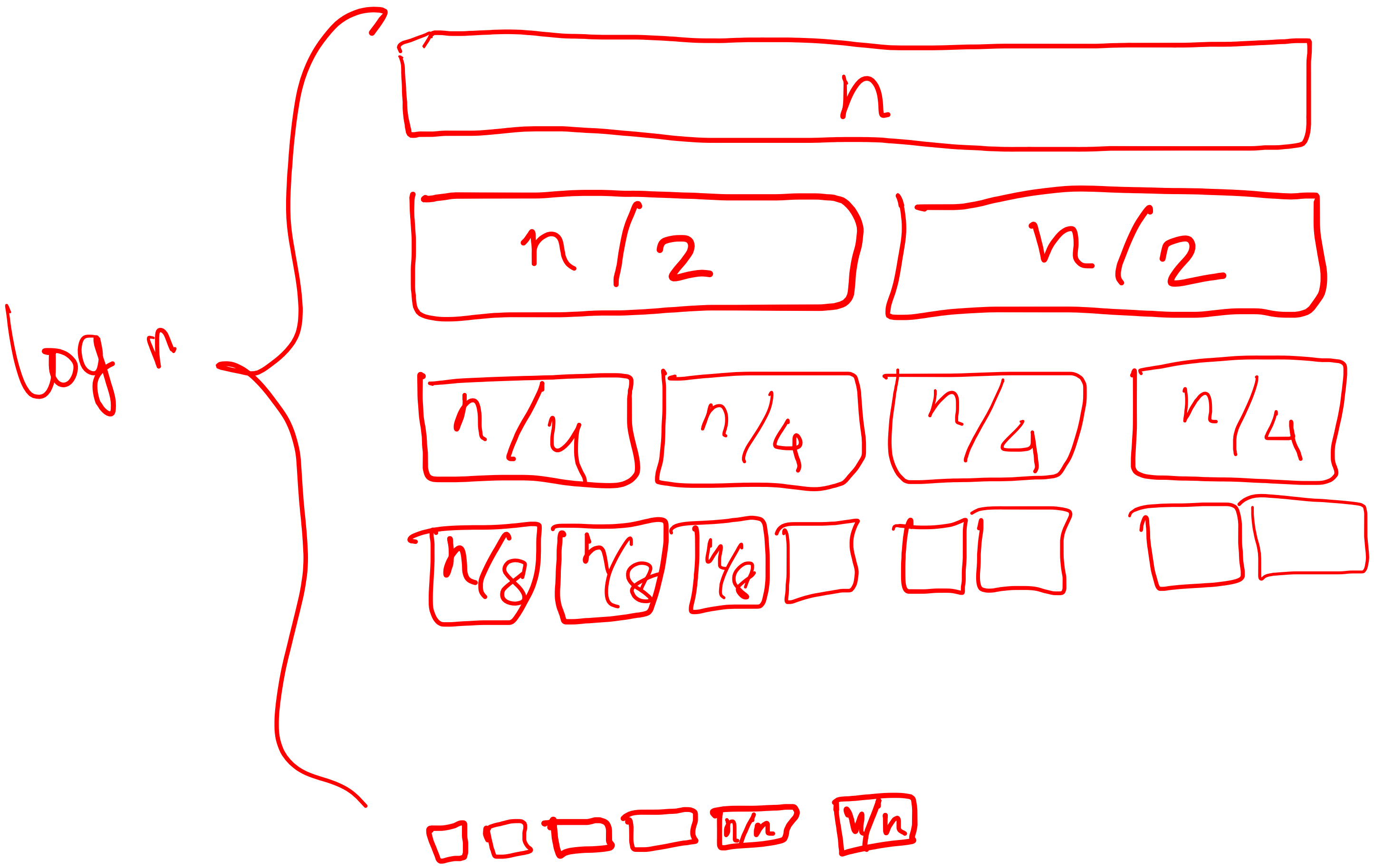
1

} n

One pass of MS is O(n) w/o rec. calls.

} n

Merge sort complexity



steps:

n

$\cancel{2} \times \frac{n}{\cancel{2}}$

$\cancel{4} \times \frac{n}{\cancel{4}}$

$\cancel{8} \times \frac{n}{\cancel{8}}$

$\cancel{n} \times \frac{n}{\cancel{n}}$

$\log n * n$

$\in O(n \log n)$

Pause

- Anything unclear or other questions?

Many sorting algorithms

- ~~Selection sort: $O(n^2)$~~
- Bubble sort
- ~~Merge sort: $O(n \log n)$~~
- Quicksort
- Insertion sort
- Heap sort (later in the quarter)
- Counting sort
- Radix sort

Quick sort

- Another divide-and-conquer algorithm

0	1	2	3	4	5	6	7	8
4	2	8	5	2	1	9	5	3

Vector notation for clarity; works even better with linked lists!

Quick sort

- Another divide-and-conquer algorithm

0	1	2	3	4	5	6	7	8
4	2	8	5	2	1	9	5	3

- Step 1: Choose one element as the $pivot$; let's pick the first element

Quick sort

- Another divide-and-conquer algorithm

0	1	2	3	4	5	6	7	8
4	2	8	5	2	1	9	5	3

Vector notation for clarity; works even better with linked lists!

- Step 1: Choose one element as the $pivot$; let's pick the first element
- Step 2: Partition the list: $elts\ <\ pivot$, and $elts \geq pivot$

2	1	3

4

7	8	5	9	5

Quick sort

- Another divide-and-conquer algorithm

0	1	2	3	4	5	6	7	8
4	2	8	5	2	1	9	5	3

Vector notation for clarity; works even better with linked lists!

- Step 1: Choose one element as the $pivot$; let's pick the first element
- Step 2: Partition the list: $elts < pivot$, and $elts \geq pivot$

2	1	3

4

7	8	5	9	5

- Step 3: Sort each sub-list recursively

Quick sort

- Another divide-and-conquer algorithm

0	1	2	3	4	5	6	7	8
4	2	8	5	2	1	9	5	3

Vector notation for clarity; works even better with linked lists!

- Step 1: Choose one element as the $pivot$; let's pick the first element
- Step 2: Partition the list: $elts < pivot$, and $elts \geq pivot$

1	2	3

4

5	5	7	8	9

- Step 3: Sort each sub-list recursively

Quick sort

- Another divide-and-conquer algorithm

0	1	2	3	4	5	6	7	8
4	2	8	5	2	1	9	5	3

Vector notation for clarity; works even better with linked lists!

- Step 1: Choose one element as the $pivot$; let's pick the first element
- Step 2: Partition the list: $elts\ <\ pivot$, and $elts \geq pivot$

1	2	3

4

5	5	7	8	9

- Step 3: Sort each sub-list recursively
- Step 4: Append them back together

1	2	3	4	5	5	7	8	9

Quick sort code for a linked list

sort-quicksort.rkt

```
def quicksort(lst):
    # Base cases
    if lst is None or lst.next is None: return lst
    # Step 1: choose a pivot
    let pivot = lst.data # first element
    # Step 2: partition the list (filter() not shown; see code)
    let below = filter(lambda x: x <  pivot, lst.next)
    let above = filter(lambda x: x >= pivot, lst.next)
    # Step 3: sort each sublist recursively
    below = quicksort(below)
    above = quicksort(above)
    # Step 4: append them back together (append() not shown; see code)
    return append(below, cons(pivot, above))
```

1

1

n

n

1 pass:

n work

O(n)

Quick sort complexity

- With a good pivot, array is divided roughly $\log n$ times before resulting in all singleton arrays
- This is similar to merge sort if a good pivot is picked on average
- Quicksort has $O(n \log n)$ average-case complexity

Picking the pivots

- First element pivot in the following array is 1

1	2	3	4	5	6	7	8	9

  - What's the complexity?
    - Every element in the same sub-list
    - Partition into sub-lists: n times
    - $O(n)$ at each recursive call → $O(n^2)$

  - Any other pivot would have been better

Picking the pivots

- What about the middle element: 5

1	2	3	4	5	6	7	8	9

  - What's the complexity?
    - Similar to original example: O(n log n)
  - Much better!

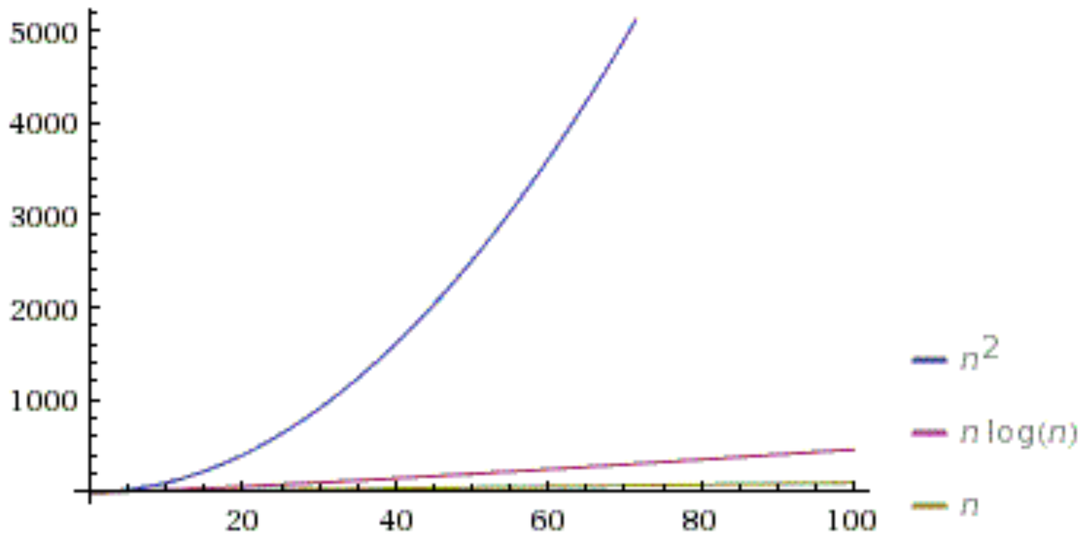
Picking the pivots

- No good strategy to guess the best pivot
- Often we pick:
  - Random element
  - Middle element
  - Median-of-three element (median of first, last, and middle element)
- Average-case complexity for such pivots is O(n log n)
  - First time we're caring about non-worst case complexity

Many sorting algorithms

- ~~Selection sort: $\Theta(n^2)$~~
- Bubble sort
- ~~Merge sort: $\Theta(n \log n)$~~
- ~~Quicksort: $\Theta(n \log n)$ average-case~~
- Insertion sort
- Heap sort (later in the quarter)
- Counting sort
- Radix sort

$n \log n$ vs. n^2



- $n \log n$ grows faster than n but much slower than n^2

In-place vs. Out-of-place sorts

- In-place sorting: sorting a vector* by modifying it, without relying on separate auxiliary storage
  - * needs $O(1)$ get/set_nth, so lists wouldn't work
- Out-ot-place sorting: sorting a vector or list by creating a separate vector or list for the output
  - Any intermediate steps may create extra storage too

Contrasting in-place and out-of-place

- Why would an in-place sort be useful?
  - Use less space; no need for extra DS allocations
  - Useful for large inputs! (or small computers)
- Why would an out-of-place sort be useful?
  - Don’t modify their inputs -> can still use the original
  - Often simpler to implement (like our quick sort)
- Both are useful in different contexts: choose wisely!

In-place or out-of-place?

- Selection sort
  - In-place
- Merge sort
  - Out-of-place
- Quick sort
  - Out-of-place
  - But can also be made in-place (messy but possible) → code on Canvas

In-class exercise (5 minutes)

- You run a small convenience store and are sorting products by their popularity so you can decide what to order more or less of. You have ~200 products to sort and have a really old computer.

1. Would you opt to use (a) selection sort, (b) merge sort, or (c) either?
2. Justify your reasoning clearly for your above answer in 1-2 sentences. Be specific in your reasoning.

What about more complex data?

- The world is not made up of such arrays or collections of numbers
  - Numbers are easy to sort
  - Characters and strings also straightforward
  - Beyond that?

- One element can contain many pieces of information
  - E.g., a student: name, year, major, etc.
  - What do we sort on? What if I want to rank across >1 field?

Case study

- Say we want to sort restaurants by their ratings on Yelp

Restaurant	*	$
Eat Unique	3.5	1
Ramen Bar	4	2
Smoke	3.5	2
ROOH	4	3
DiAnoia's Eatery	4.5	2
Geja's cafe	4	3
Ethiopian Diamond	4	2

→

Restaurant	*	$
DiAnoia's Eatery	4.5	2
Geja's cafe	4	3
Ethiopian Diamond	4	2
ROOH	4	3
Ramen Bar	4	2
Smoke	3.5	2
Eat Unique	3.5	1

Case study

Restaurant	*	$
Eat Unique	3.5	1
Ramen Bar	4	2
Smoke	3.5	2
ROOH	4	3
DiAnoia's Eatery	4.5	2
Geja's cafe	4	3
Ethiopian Diamond	4	2

→

Restaurant	*	$
DiAnoia's Eatery	4.5	2
Geja's cafe	4	3
Ethiopian Diamond	4	2
ROOH	4	3
Ramen Bar	4	2
Smoke	3.5	2
Eat Unique	3.5	1

- Issue: the computer doesn't know how to compare two elements/rows:
  - (Eat Unique, 3.5, 1) < (Ramen Bar, 4, 2)?
  - We need to tell the sort how to compare to elements (select a sort key):
    `lambda r1, r2: r1.stars > r2.stars`

Selection sort revisited

- Selection sort in descending order

```
def selection_sort (unsorted):
   for i in range(unsorted.len()-1):
      # not shown: finds min in array(i+1, len)
      let max = find_largest_vec(unsorted, i)

      # not shown: swap values at index i and max
      swap(unsorted, i, min)
```

Selection sort with custom comparator

- Only modify the part of `find_largest_vec` that checks <, >, or =

What it would have looked like before:

```
def find_largest_vec(vec, idx):
    let max = idx
    for i in range(idx+1, vec.len()):
        if vec[i] > vec[max]:
            max = i
    return max
```

What it would look like now:

```
def find_largest_vec(vec, idx, cmp_f):
    let max = idx
    for i in range(idx+1, vec.len()):
        if cmp_f(vec[i], vec[max]):
            max = i
    return max
```

Selection sort with custom comparator

- Only modify the part of `find_largest_vec` that checks <, >, or =

Also allows us to use the same function for both descending and ascending (just specify in the comparator function)

Other sorts can be adapted in similar ways

What it would have lo

```
def find_largest_
    let max = idx
    for i in range
        if vec[i] > vec[max]:
            max = i
    return max
```

```
mp_f):

en()):
        if cmp_f(vec[i], vec[max]):
            max = i
    return max
```

Think beyond the code: Be careful with comparators

- A user-defined comparator is not vetted and may have ethical issues
- May introduce biases if sort key inherently shows bias

- Case study: Sort applicants by their qualification to get accepted to college
  - Sorting by race, gender, age obviously bad (and illegal)
  - What are features we can sort by or what keys could cause a biased sort?
  - Would you consider sorting on "number of extracurriculars" as the sort key to be ethical according to the above requirement?