Sorting and selection – Part 2

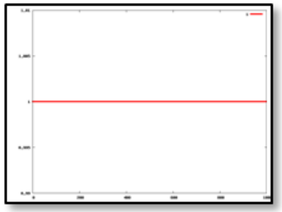
Prof. Noah Snavely
CS1114
http://cs1114.cs.cornell.edu

Cornell University
Computer Science

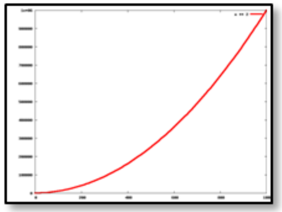
Administrivia

- Assignment 1 due tomorrow by 5pm
- Assignment 2 will be out tomorrow
  - Two parts: smaller part due next Friday, larger part due in two weeks
- Quiz 2 next Thursday 2/12
  - Coverage through next Tuesday
    (topics include running time, sorting)
  - Closed book / closed note

Recap from last time: sorting



- If we sort an array, we can find the k^{th} largest element in constant (O(1)) time
  - For all k, even for the median ($k = n/2$)



- Sorting algorithm 1: Selection sort
  - Running time: $O(n^2)$

?

- Sorting algorithm 2: Quicksort
  - Running time: O(?)

Quicksort

1. Pick an element (**pivot**)
2. Compare every element to the pivot and **partition** the array into elements < pivot and > pivot
3. Quicksort these smaller arrays separately

Quicksort: worst case

- With a bad pivot this algorithm does quite poorly
  - Degrades to selection sort
  - Number of comparisons will be $O(n^2)$
- The worst case occurs when the array is already sorted
  - We could choose the average element instead of the first element

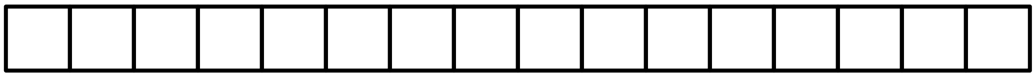
Quicksort: best case

- With a good choice of pivot the algorithm does quite well
- What is the best possible case?
  – Selecting the median
- How many comparisons will we do?
  – Every time **`quicksort`** is called, we have to:

```
% Compare all elements to the pivot
```

How many comparisons? (best case)

- Suppose **`length(A) == n`**



- Round 1: Compare *n* elements to the pivot

... now break the array in half, quicksort the two halves ...

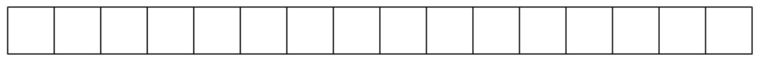
- Round 2: For each half, compare *n* / 2 elements to each pivot (total # comparisons = n)

... now break each half into halves ...

- Round 3: For each quarter, compare *n* / 4 elements to each pivot (total # comparisons = n)

How many comparisons? (best case)

Suppose `length(A) == n`



Round 1: Compare *n* elements to the pivot

… now break the array in half, quicksort the two halves …

Round 2: For each half, compare *n* / 2 elements to the pivot (total # comparisons = ?)

… now break each half into halves …

Round 3: For each quarter, compare *n* / 4 elements to the pivot (total # comparisons = ?)

⋮

How many rounds will this run for?

How many comparisons? (best case)

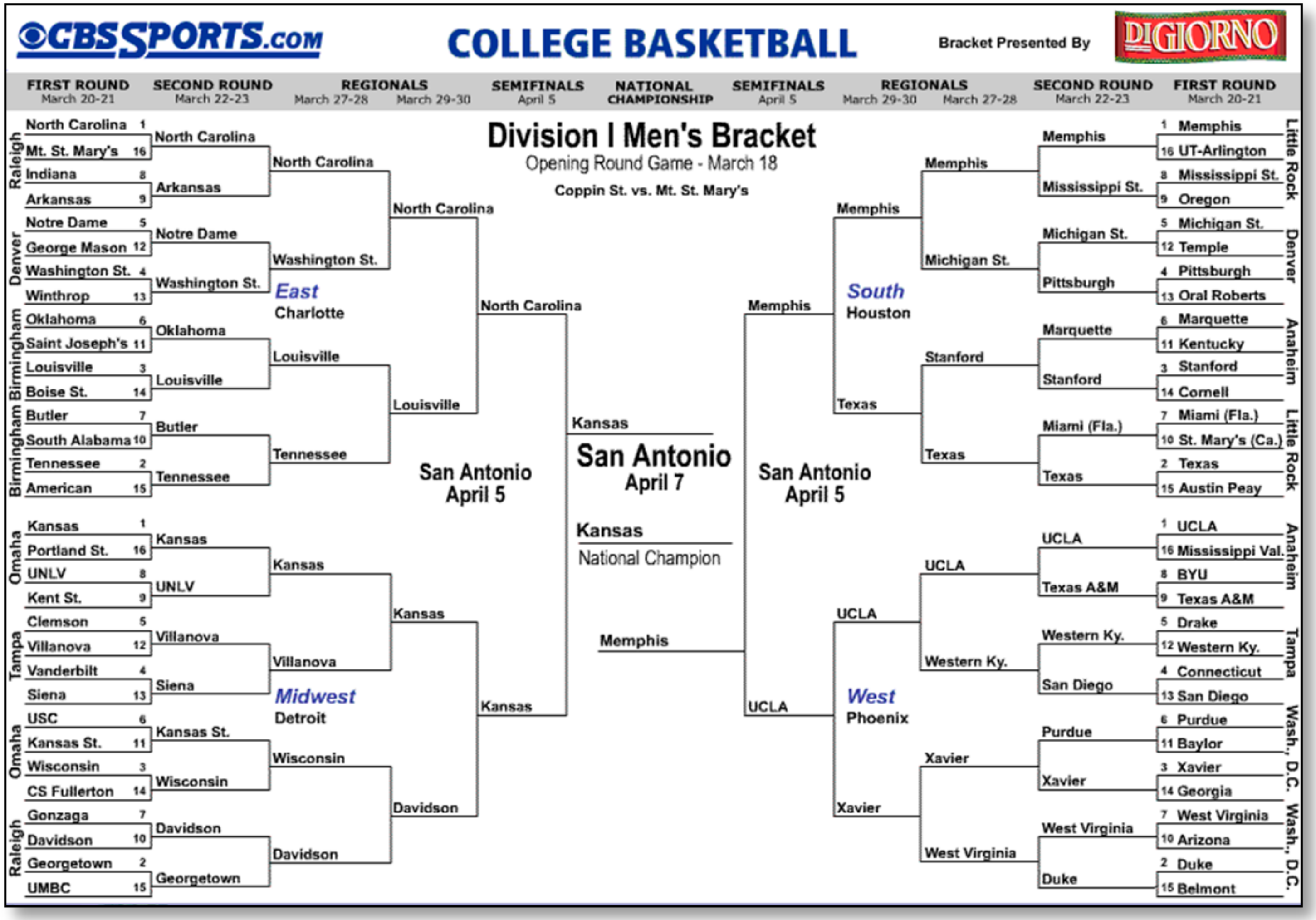
- During each round, we do a total of *n* comparisons
- There are log n rounds
- The total number of comparisons is n log n
- In the best case quicksort is O(n log n)

Can we expect to be lucky?

- Performance depends on the input
- “Unlucky pivots” (worst-case) give O(n^2) performance
- “Lucky pivots” give O(n log n) performance
- For random inputs we get “lucky enough” – expected runtime on a random array is O(n log n)
- Can we do better?

Back to the selection problem

- Can solve with sorting
- Is there a better way?
- Rev. Charles L. Dodgson's problem
  - Based on how to run a tennis tournament
  - Specifically, how to award 2^{nd} prize fairly



- How many teams were in the tournament?
- How many games were played?
- Which is the second-best team?

Finding the second best team

- Could use quicksort to sort the teams

- Step 1: Choose one team as a pivot (say, Arizona)
- Step 2: Arizona plays every team
- Step 3: Put all teams worse than Arizona in Group 1, all teams better than Arizona in Group 2 (no ties allowed)
- Step 4: Recurse on Groups 1 and 2
- ... eventually will rank all the teams ...

Quicksort Tournament

Quicksort Tournament

Step 1: Choose one team (say, Arizona)

Step 2: Arizona plays every team

Step 3: Put all teams worse than Arizona in Group 1, all teams better than Arizona in Group 2 (no ties allowed)

Step 4: Recurse on groups 1 and 2

... eventually will rank all the teams ...

- (Note this is a bit silly – AZ plays 63 games)
- This gives us a ranking of all teams
  - What if we just care about finding the 2nd-best team?

Modifying quicksort to select

- Suppose Arizona beats 36 teams, and loses to 27 teams

36 teams < A < 27 teams

Group 1 — Group 2

- If we just want to know the 2nd-best team, how can we save time?

Modifying quicksort to select – Finding the 2nd best team

36 teams < < 27 teams

Group 1 Group 2

16 teams < Cornell < 10 teams

Group 2.1 Group 2.2

7 teams < < 2 teams

Modifying quicksort to select – Finding the 32nd best team

36 teams < < 27 teams

Group 1 Group 2

20 teams < < 15 teams

Group 1.1 Group 1.2

- Q: Which group do we visit next?
- The 32nd best team overall is the 4th best team in Group 1

Find k^{th} largest element in A (< than k-1 others)

A = [6.0 5.4 5.5 6.2 5.3 5.0 5.9]

MODIFIED QUICKSORT(A, k):

- Pick an element in A as the pivot, call it x
- Divide A into A1 (<x), A2 (=x), A3 (>x)
- If k < length(A3)
  - MODIFIED QUICKSORT (A3, k)
- If k > length(A2) + length(A3)
  - Let j = k – [length(A2) + length(A3)]
  - MODIFIED QUICKSORT (A1, j)
- Otherwise, return x

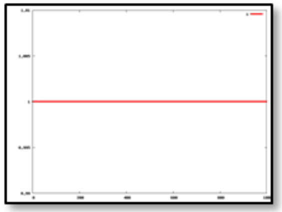
Modified quicksort

MODIFIED QUICKSORT(A, k):

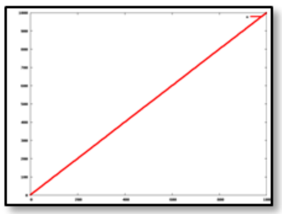
- Pick an element in A as the pivot, call it x
- Divide A into A1 (<x), A2 (=x), A3 (>x)
- If k < length(A3)
  - Find the element < k others in A3
- If k > length(A2) + length(A3)
  - Let j = k – [length(A2) + length(A3)]
  - Find the element < j others in A1
- Otherwise, return x

- We'll call this *quickselect*
- Let's consider the running time…

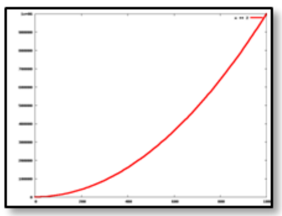
What is the running time of:



- Finding the 1st element?
  - O(1) (effort doesn't depend on input)



- Finding the biggest element?
  - O(n) (constant work per input element)



- Finding the median by repeatedly finding and removing the biggest element?
  - $O(n^2)$ (linear work per input element)

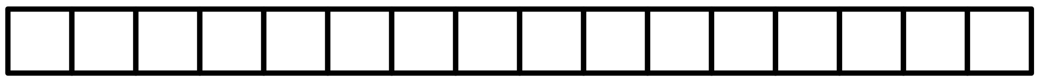
- Finding the median using quickselect?
  - Worst case? O(n^2)
  - Best case? O(n)

Quickselect – "medium" case

- Suppose we split the array in half each time (i.e., happen to choose the median as the pivot)
- How many comparisons will there be?

How many comparisons? ("medium" case)

- Suppose **`length(A) == n`**



- Round 1: Compare *n* elements to the pivot

… now break the array in half, quickselect one half …

- Round 2: For remaining half, compare *n / 2* elements to the pivot (total # comparisons = n / 2)

… now break the half in half …

- Round 3: For remaining quarter, compare *n / 4* elements to the pivot (total # comparisons = n / 4)

How many comparisons? ("medium" case)

Number of comparisons =

$$n + n / 2 + n / 4 + n / 8 + \ldots + 1$$

$$= ?$$

→ The "medium" case is O(n)!

Quickselect

- For random input this method actually runs in linear time (beyond the scope of this class)
- The worst case is still bad
- Quickselect gives us a way to find the k^{th} element without actually sorting the array!

Quickselect

- It's possible to select in *guaranteed* linear time (1973)
  - Rev. Dodgson's problem
  - But the code is a little messy
    - And the analysis is messier

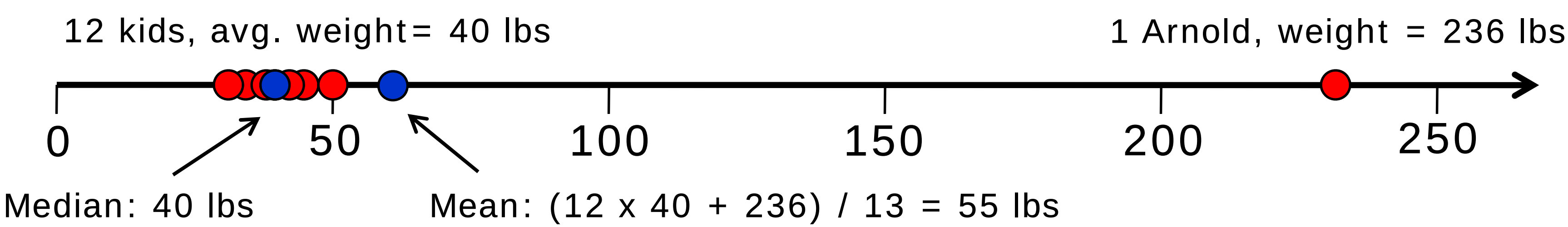
    http://en.wikipedia.org/wiki/Selection_algorithm
- Beyond the scope of this course

Back to the lightstick

- By using quickselect we can find the 5% largest (or smallest) element
  - This allows us to efficiently compute the trimmed mean

What about the median?

- Another way to avoid our bad data points:
  - Use the median instead of the mean



Median vector

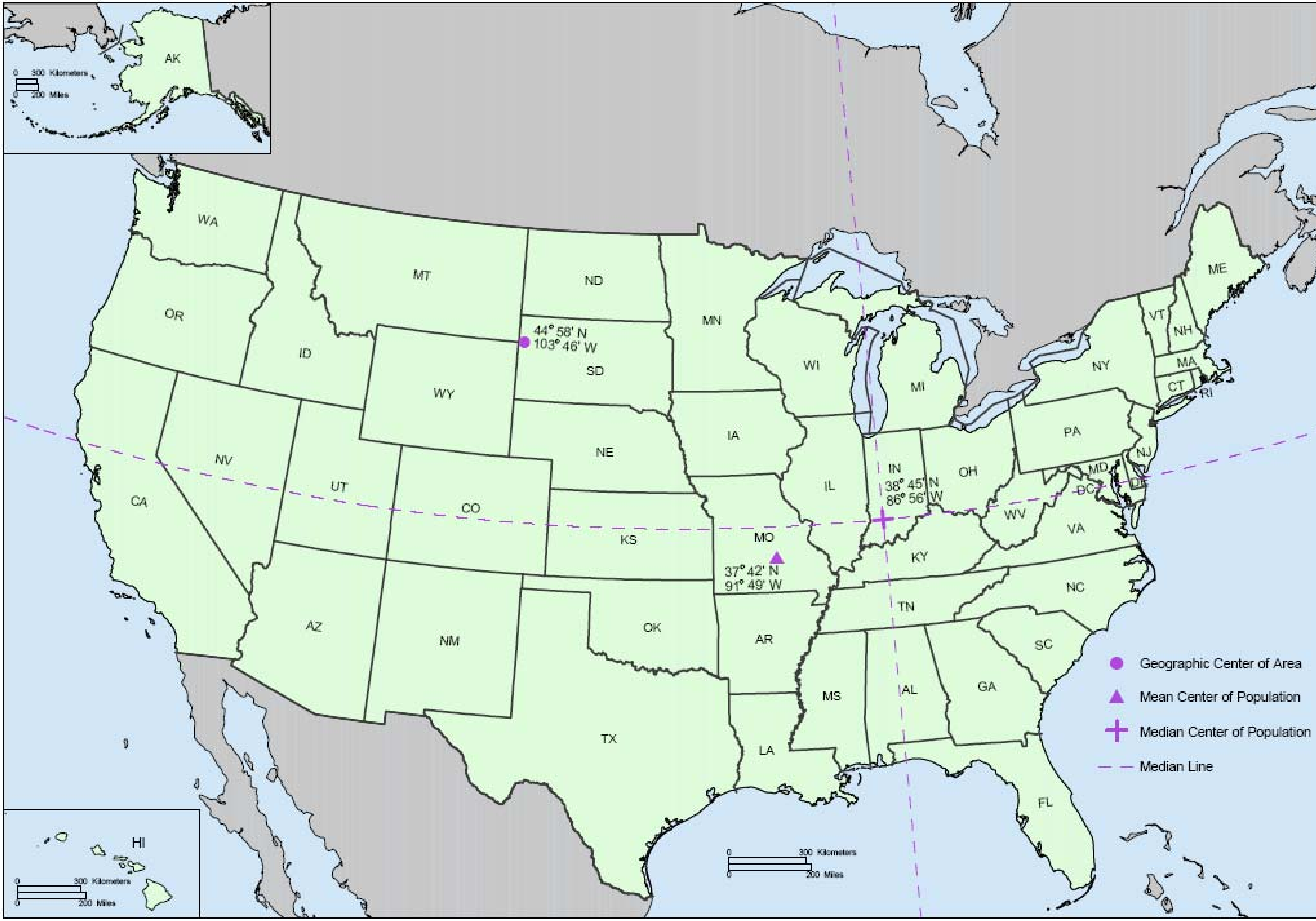
- Mean, like median, was defined in 1D
  - For a 2D mean we used the centroid
  - Mean of x coordinates and y coordinates separately
    - Call this the “mean vector”

  - Does this work for the median also?

What is the median vector?

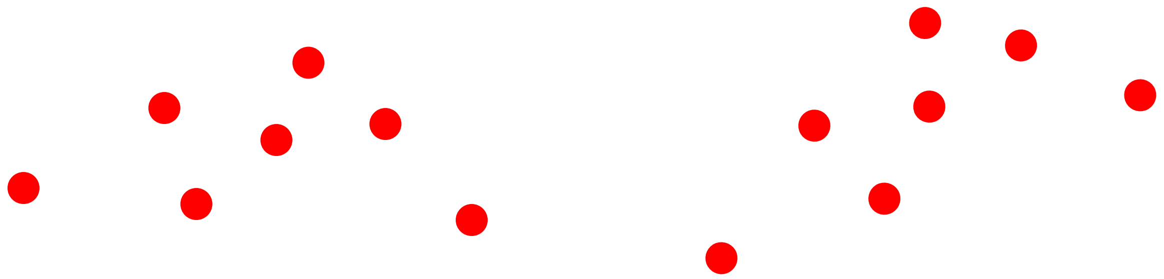
- In 1900, statisticians wanted to find the "geographical center of the population" to quantify westward shift
- Why not the centroid?
  - Someone being born in San Francisco changes the centroid much more than someone being born in Indiana
- What about the "median vector"?
  - Take the median of the x coordinates and the median of the y coordinates separately

Position of the Geographic Center of Area, Mean and Median Centers of Population: 2000



Median vector

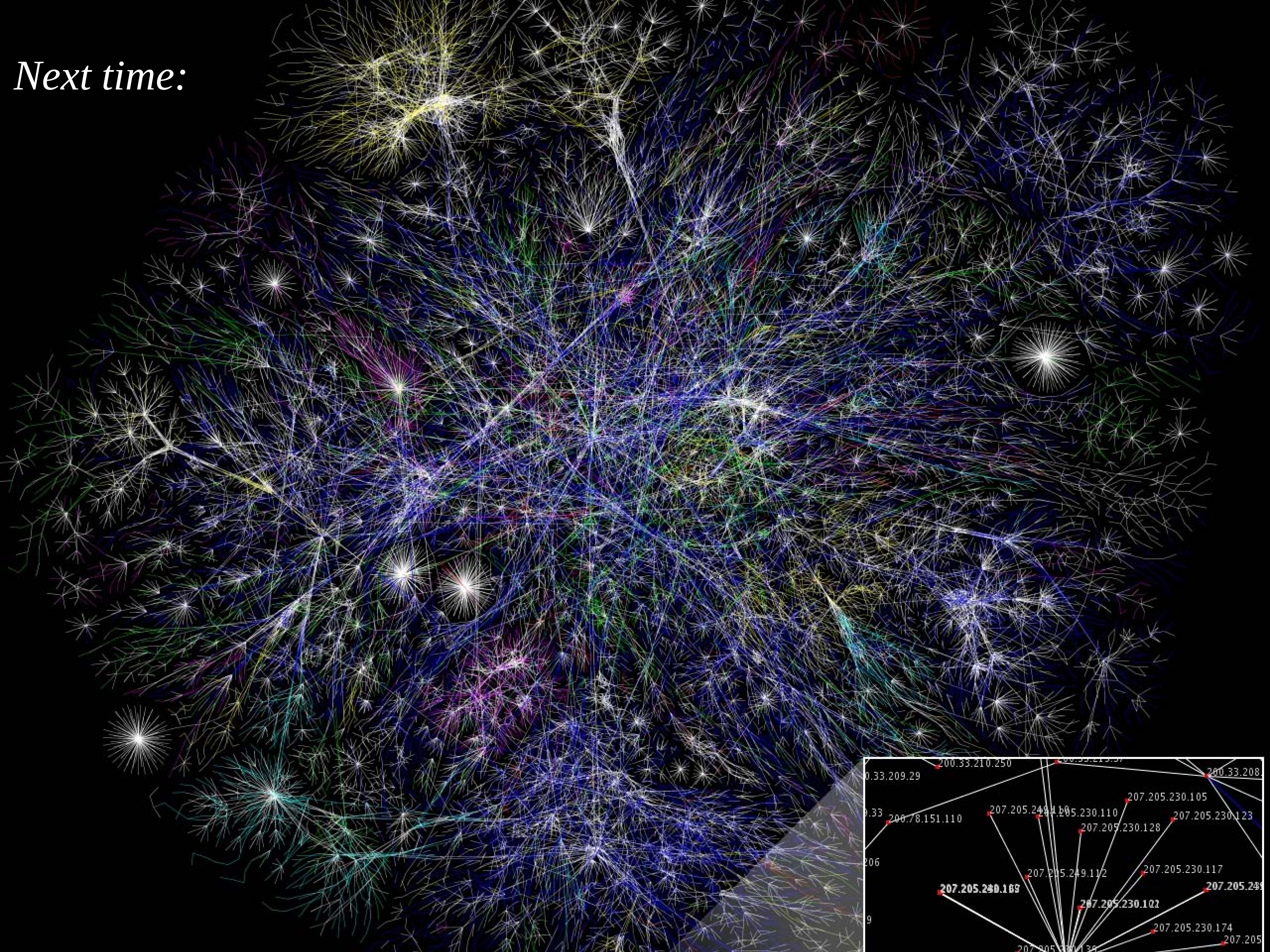
- A little thought will show you that this doesn't really make a lot of sense
  - Nonetheless, it's a common solution, and we will implement it for CS1114
  - In situations like ours it works pretty well
- It's almost never an actual datapoint
- It depends upon rotations!



Can we do even better?

- None of what we described works that well if we have widely scattered red pixels
  - And we can't figure out lightstick orientation
- Is it possible to do even better?
  - Yes!
- We will focus on:
  - Finding "blobs" (connected red pixels)
  - Summarizing the shape of a blob
  - Computing orientation from this
- We'll need brand new tricks!

Next time:


200.33.210.250
207.205.230.105
207.205.230.110
200.78.151.110
207.205.230.128
207.205.230.123
207.205.249.112
207.205.230.117
207.205.230.174