# 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 next Thursday

## Neat CS talk today

- **Culturomics: Quantitative Analysis of Culture Using Millions of Digitized Books**
- Upson B-17 4:15pm



## Recap from last time

- How can we quickly compute the median / trimmed mean of an array?
  - The *selection* problem
- One idea: sort the array first
  - This makes the selection problem easier
- How do we sort?

## Recap from last time

- Last time we looked at one sorting algorithm, *selection* sort
- How fast is selection sort?



## Speed of selection sort

- Total number of comparisons:

$n + (n - 1) + (n - 2) + \ldots + 1$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

## Is this the best we can do?

- Maybe the problem of sorting *n* numbers is intrinsically $O(n^2)$
  - (i.e., maybe all possible algorithms for sorting *n* numbers are $O(n^2)$)
- Or maybe we just haven't found the right algorithm...
- Let's try a different approach
  - Back to the problem of sorting the actors...



## Sorting, 2nd attempt

- Suppose we tell all the actors
  - shorter than 5.5 feet to move to the left side of the room
- and all actors
  - taller than 5.5 feet to move to the right side of the room
  - (actors who are exactly 5.5 feet move to the middle)

`[ 6.0 5.4 5.5 6.2 5.3 5.0 5.9 ]`

↓

`[ 5.4 5.3 5.0 5.5 6.0 6.2 5.9 ]`

# Sorting, 2$^{nd}$ attempt

[ 6.0 5.4 5.5 6.2 5.3 5.0 5.9 ]

[ 5.4 5.3 5.0 | 5.5 | 6.0 6.2 5.9 ]

< 5.5 | | > 5.5

- Not quite done, but it's a start
- We've put every element on the correct side of 5.5 (the *pivot*)
- What next?

- *Divide and conquer*



# How do we select the pivot?

- How did we know to select 5.5 as the pivot?
- Answer: average-ish human height
- In general, we might not know a good value
- Solution: just pick some value from the array (say, the first one)

# Quicksort

This algorithm is called *quicksort*

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

- Example of a *recursive* algorithm (defined in terms of itself)



# Quicksort example

| | |
|---|---|
| Select pivot | [ 10 13 41 6 51 11 3 ] |
| Partition | [ 6 3 10 13 41 51 11 ] |
| Select pivot | [ 6 3 ] 10 [ 13 41 51 11 ] |
| Partition | [ 3 6 ] 10 [ 11 13 41 51 ] |
| Select pivot | [ 3 ] 6 10 [ 11 ] 13 [ 41 51 ] |
| Partition | 3 6 10 11 13 [ 41 51 ] |
| Select pivot | 3 6 10 11 13 41 [ 51 ] |
| Done | 3 6 10 11 13 41 51 |

# Quicksort – pseudo-code

```
function [ S ] = quicksort(A)
% Sort an array using quicksort
n = length(A);
if n <= 1
    S = A; return; % The base case
end

pivot = A(1);  % Choose the pivot
smaller = []; equal = []; larger = [];

% Compare all elements to the pivot:
%    Add all elements smaller than pivot to 'smaller'
%    Add all elements equal to pivot to 'equal'
%    Add all elements larger than pivot to 'larger'

% Sort 'smaller' and 'larger' separately
smaller = quicksort(smaller); larger = quicksort(larger); % This
   is where the recursion happens
S = [ smaller equal larger ];
```



# Quicksort and the pivot

- There are lots of ways to make quicksort fast, for example by swapping elements
  - We will cover these in section

# Quicksort and the pivot

- With a bad pivot this algorithm does quite poorly
  - Suppose we happen to always pick the smallest element of the array?
  - What does this remind you of?
- When can the bad case easily happen?



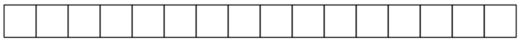
# Quicksort and the pivot

- With a good choice of pivot the algorithm does quite well
- Suppose we get lucky and choose the median every time
- How many comparisons will we do?
  - Every time **`quicksort`** is called, we have to:

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

# How many comparisons? (Lucky pivot 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 = ?)

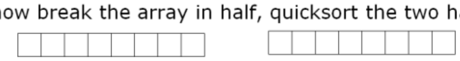


# How many comparisons? (Lucky pivot case)

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

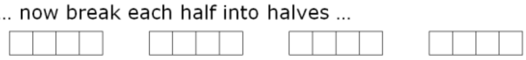
Round 1: Compare *n* elements to the pivot

… 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? (Lucky pivot case)

- During each round, we do a total of __ comparisons
- There are _________ rounds
- The total number of comparisons is __________
- With "lucky pivots" quicksort is O(__________)



## Can we expect to be lucky?

- Performance depends on the input
- "Unlucky pivots" (worst-case) give O($n^2$) performance
- "Lucky pivots" give O(________) performance
- For random inputs we get "lucky enough" – expected runtime on a random array is O(________)

# Questions?



# Recursion

- Recursion is cool and useful
  - Sierpinski triangle

- But use with caution

```
function x = factorial(n)
    x = n * factorial(n - 1)
end
```

## Back to the selection problem

- Can solve with quicksort
  - Faster (on average) than "repeated remove biggest"
- 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?