CS 1114: Sorting and selection (part two)

Prof. Graeme Bailey

http://cs1114.cs.cornell.edu

(notes modified from Noah Snavely, Spring 2009)

Cornell University Computer Science

Recap from last time

- We looked at the "trimmed mean" problem for locating the lightstick
  - Remove 5% of points on all sides, find centroid
- This is a version of a more general problem:
  - Finding the k^{th} largest element in an array
  - Also called the "selection" problem
- To solve this, we try first solving the sorting problem (then getting the k^{th} largest is easy)
  - How fast is this algorithm?

Is this the best we can do?

- Let's try a different approach
- Suppose we tell all the actors
  - shorter than 5.5' to move to the left side of the room
- and all actors
  - taller than 5.5' to move to the right side of the room
  - (actors who are exactly 5.5' 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, 2nd 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?
- Do this again on each side: ask shorter group to pivot on (say) 5.3', taller group to pivot on 6.0'
- *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. Compare every element to the pivot and **partition** the array into elements < pivot and > pivot
3. Quicksort these smaller arrays separately

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;
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 called recursion
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?
  - Number of comparisons will be $O(n^2)$
- When can the bad case easily happen?
- The worst case occurs when the array is already sorted
  - We could choose the average element instead of the first element

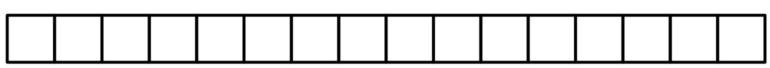
Quicksort: best case

- With a good choice of pivot the algorithm does quite well
- What is the best possible case?
  - Selecting the median (how easy is this to find?)
- 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 …

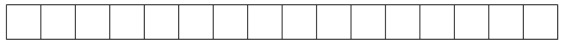
- Round 2: For each half, compare *n* / 2 elements to each pivot (total # comparisons = n)

… 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)

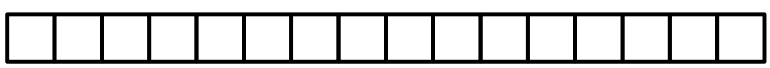
- During each round, we do a total of *n* comparisons
- There are log n rounds
- The total number of comparisons is n log n
- In the best case quicksort is O(n log n)

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?

Another sort of sort
(step 1, decompose)

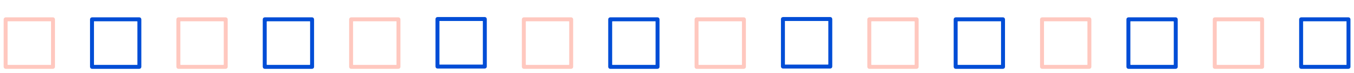
- Suppose `length(A) == n`



- Round 1: Break the array in half

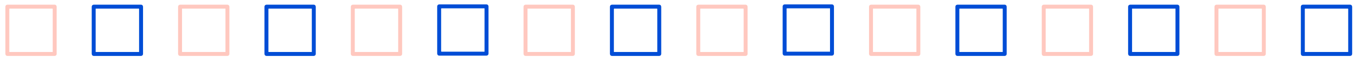
- Round 2: Now break each half in half …

- Round log(n): Now have a lot (n) of small arrays!



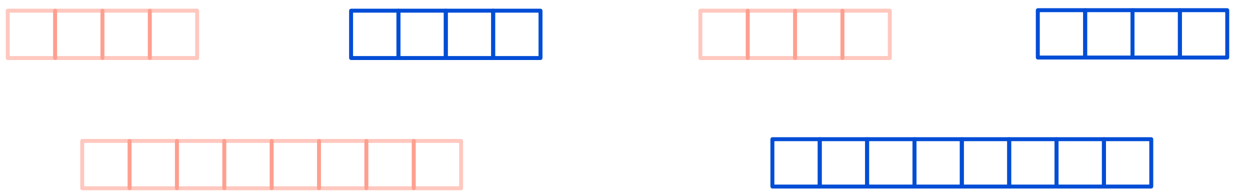
Another sort of sort
(step 2, recombine)

- It's easy to sort an array of length one! So …

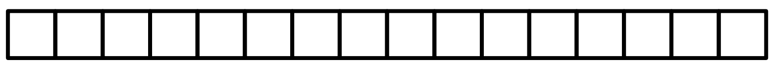


- Round 1: Merge them pairwise in the correct order

- Round 2: Now merge these…



- Round log(n): Now we have one sorted array!



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 2nd prize fairly

- We'll dodge this until next time ……..