CS 1114:
Sorting and selection (part one)

Prof. Graeme Bailey

http://cs1114.cs.cornell.edu

(notes modified from Noah Snavely, Spring 2009)

Cornell University
Computer Science

Where we are so far

- Finding the lightstick
  - Attempt 1: Bounding box (not so great)
  - Attempt 2: Centroid isn't much better
  - Attempt 3: Trimmed mean
    - Seems promising
    - But how do we compute it quickly?
    - The obvious way doesn't seem so great…
    - But do we really know this?

How do we define fast?

- We want to think about this issue in a way that doesn't depend on either:
  A. Getting really lucky input
  B. Happening to have really fast hardware

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
- We considered an algorithm that repeatedly removes the largest element
  - How fast is this algorithm?

A more general version of trimmed mean

- Given an array of *n* numbers, find the k^{th} largest number in the array
- Strategy:
  - Remove the biggest number
  - Do this *k* times
  - The answer is the last number you found

How fast is this algorithm?

- An elegant answer exists
- You will learn it in later CS courses
  - But I'm going to steal their thunder and explain the basic idea to you here
  - It's called "big-O notation"

- Two basic principles:
  - Think about the average / worst case
    - Don't depend on luck
  - Think in a hardware-independent way
    - Don't depend on the chip!

Performance of our algorithm

- What value of *k* is the worst case?
  - ~~$k = n$~~ we can easily fix this
  - $k = n/2$

- How much work will we do in the worst case?

1. Examine *n* elements to find the biggest
2. Examine *n*-1 elements to find the biggest

… keep going …

n/2. Examine *n*/2 elements to find the biggest

How much work is this?

- How many elements will we examine in total?

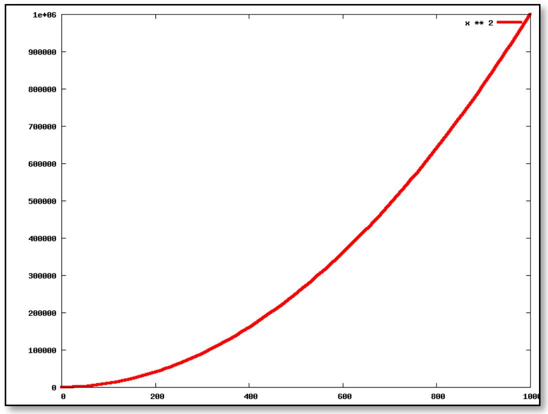
$$\underbrace{n + (n - 1) + (n - 2) + \ldots + n/2}_{n\ /\ 2 \text{ terms}}$$

$$= ?$$

- We don't really care about the exact answer
  - It's bigger than $(n / 2)^2$

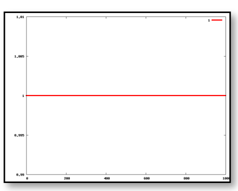
How much work is this?

- The amount of work grows *in proportion* to n^2

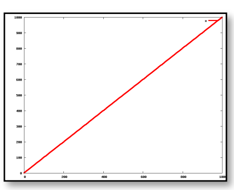


- We say that this algorithm is $O(n^2)$
- *[Blackboard discussion of O(g(n)) and o(g(n))]*

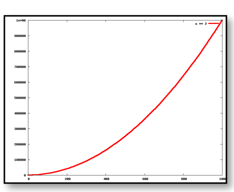
Classes of algorithm speed



- Constant time algorithms, $O(1)$
  - Do not depend on the input size
  - Example: find the first element



- Linear time algorithms, $O(n)$
  - Constant amount of work for every input item
  - Example: find the largest element



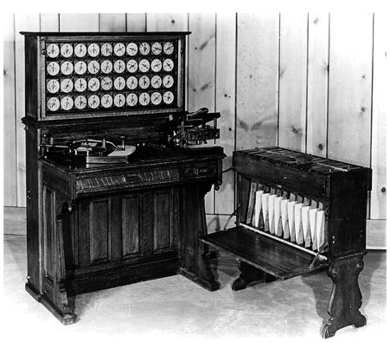
- Quadratic time algorithms, $O(n^2)$
  - Linear amount of work for every input item
  - Example: repeatedly removing max element

Different hardware only affects the parameters (i.e., line slope)

How to do selection better?

- If our input were sorted, we can do better
  - Given 100 numbers in increasing order, we can easily figure out the 5th biggest or smallest
- Very important principle! (encapsulation)
  - Divide your problem into pieces
    - One person (or group) can provide `sort`
    - The other person can use `sort`
  - As long as both agree on what `sort` does, they can work independently
  - Can even "upgrade" to a faster `sort`

How to sort?



- Sorting is an ancient problem, by the standards of CS
  - First important "computer" sort used for 1890 census, by Hollerith (the 1880 census took 8 years, 1890 took just one)
- There are many sorting algorithms

How to sort?

- Given an array of numbers:

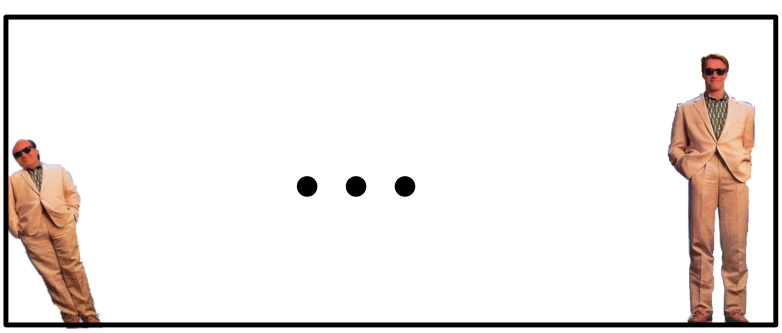
  [10 2 5 30 4 8 19 102 53 3]

- How can we produce a sorted array?

  [2 3 4 5 8 10 19 30 53 102]

How to sort?

- A concrete version of the problem
  - Suppose I want to sort all actors by height



  - How do I do this?

Sorting, 1st attempt

- Idea: Given *n* actors

1. Find the shortest actor (D. Devito), put him first
2. Find the shortest actor in the remaining group, put him/her second

… Repeat …

n. Find the shortest actor in the remaining group (one left), put him/her last

Sorting, 1st attempt

Algorithm 1

1. Find the shortest actor put him first
2. Find the shortest actor in the remaining group, put him/her second

… Repeat …

n. Find the shortest actor in the remaining group put him/her last

- What does this remind you of?
- This is called *selection sort*
- After round *k*, the first *k* entries are sorted

Selection sort – pseudocode

```
function [ A ] = selection_sort(A)
% Returns a sorted version of array A
%   by applying selection sort
%   Uses in place sorting
n = length(A);
for i = 1:n
    % Find the smallest element in A(i:n)
    % Swap that element with something (what?)
end
```


Filling in the gaps

- `% Find the smallest element in A(i:n)`
- We pretty much know how to do this

```
m = 10000; m_index = -1;
for j in i:n
    if A(j) < m
        m = A(j); m_index = j;
    end
end
```

```
[ 10 13 41 6 51 11 ]
% After round 1,
%  m = 6, m_index = 4
```

Filling in the gaps

- `% Swap the smallest element with something`
- `% Swap element A(m_index) with A(i)`

```
A(i) = A(m_index);
A(m_index) = A(i);
```

```
tmp = A(i);
A(i) = A(m_index);
A(m_index) = tmp;
```

[10 13 41 6 51 11]

[6 13 41 10 51 11]

Putting it all together

```
function [ A ] = selection_sort(A)
% Returns a sorted version of array A
len = length(A);
for i = 1:len
    % Find the smallest element in A(i:len)
    m = 10000; m_index = -1;
    for j in i:n
        if A(j) < m
            m = A(j); m_index = j;
        end
    end
    % Swap element A(m_index) with A(i)
    tmp = A(i);
    A(i) = A(m_index);
    A(m_index) = tmp;
end
```

Example of selection sort

```
[ 10 13 41 6 51 11 ]
[ 6 13 41 10 51 11 ]
[ 6 10 41 13 51 11 ]
[ 6 10 11 13 51 41 ]
[ 6 10 11 13 51 41 ]
[ 6 10 11 13 41 51 ]
[ 6 10 11 13 41 51 ]
```


Speed of selection sort

- Let n be the size of the array
- How fast is selection sort?

$O(1)$ $O(n)$ $O(n^2)$?

- How many comparisons (<) does it do?
- First iteration: n comparisons
- Second iteration: $n - 1$ comparisons

…

- n^{th} iteration: 1 comparison

Speed of selection sort

- Total number of comparisons:

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

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

- Work grows in proportion to n^2 → selection sort is $O(n^2)$

Is this the best we can do?

- Wait and see !!!!

- Don't forget to spend time in the lab getting help on hw 1 – we have a really fun exercise coming up for the next homework ☺