Sorting and selection

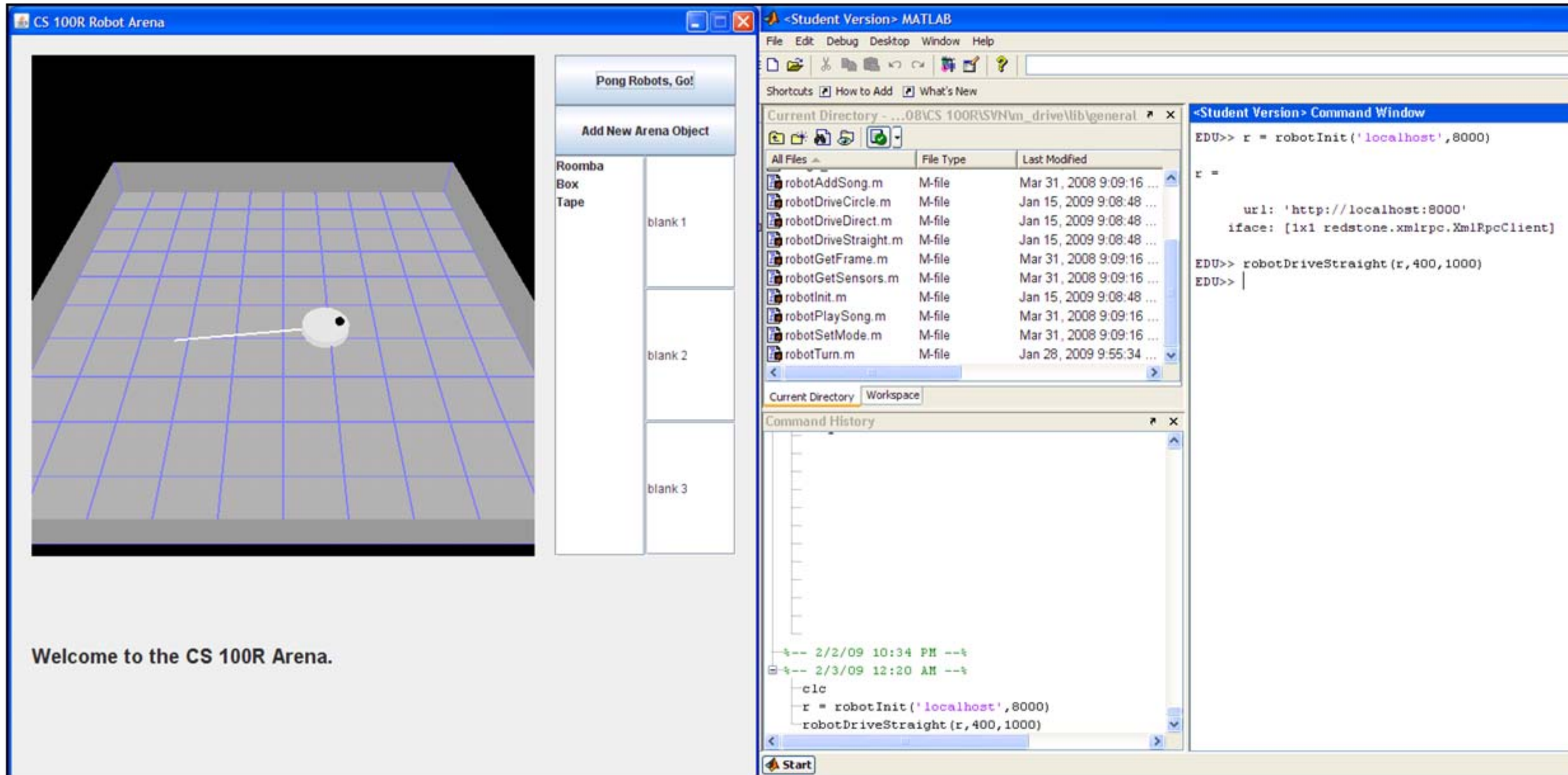
Prof. Noah Snavely
CS1114
http://cs1114.cs.cornell.edu

Cornell University
Computer Science

Administrivia

- Assignment 1 due Friday by 5pm
  - Please sign up for a demo slot using CMS
  - If you don't yet have a 100R account, please let me know
- Assignment 2 out on Friday
- Quiz 2 next Thursday 2/12
  - Coverage through next Tuesday
  - Closed book / closed note

Administrivia



- More robots will be working soon
  - If all of the working ones are in use, you can test your code with the robot arena

Logical operators

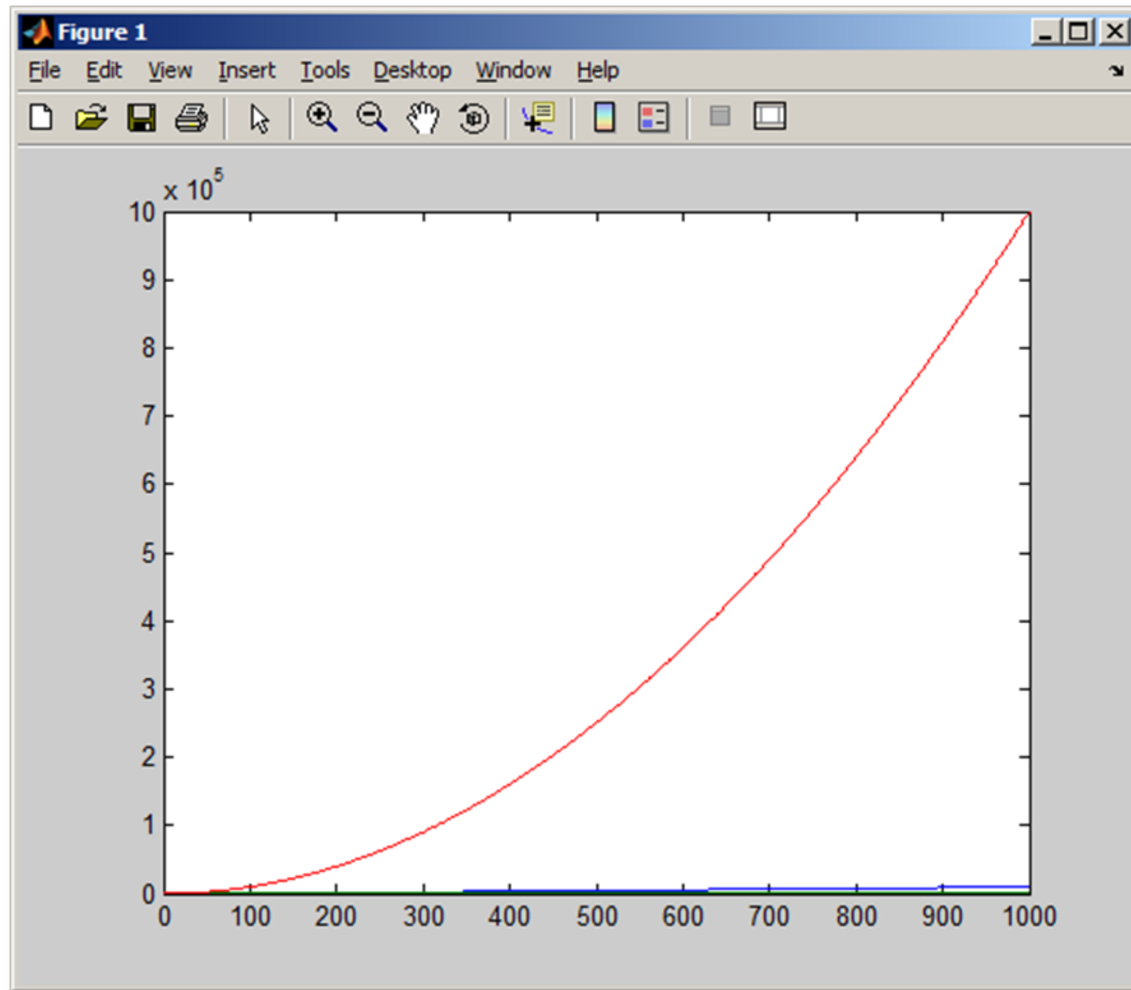
- && -- logical "and"
- || -- logical "or"

```
if (x == 1 && y == 2) || z == 3
    % do something interesting
else
    % do something else
end
```

What’s the difference between…

```
A(i) = A(i) + 2i;
A(i) = A(i) + 2 * i;
```

Plotting in Matlab



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 kth 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?

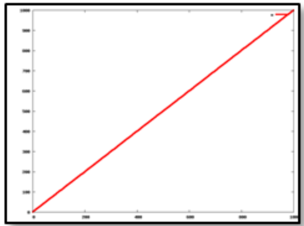
Recap from last time

- Big-O notation allows us to reason about speed without worrying about
  - Getting lucky on the input
  - Depending on our hardware

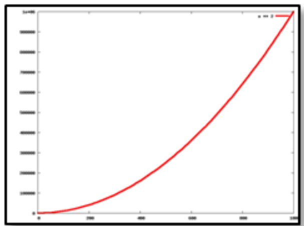
- Big-O of repeatedly removing the biggest element?
  - Worst-case (k = n/2, i.e., median) is quadratic, $O(n^2)$

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

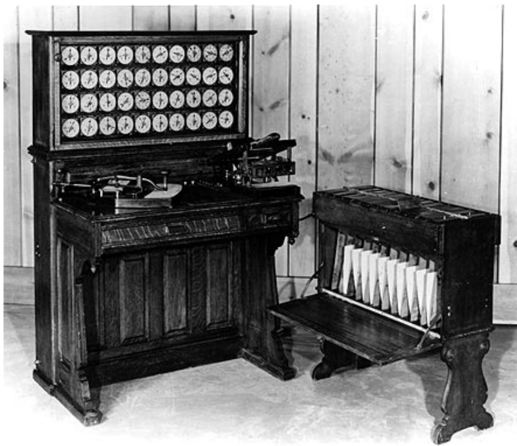


- Quadratic time algorithms, $O(n^2)$
  - Linear amount of work for every input item
  - Example: repeatedly removing max element

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:

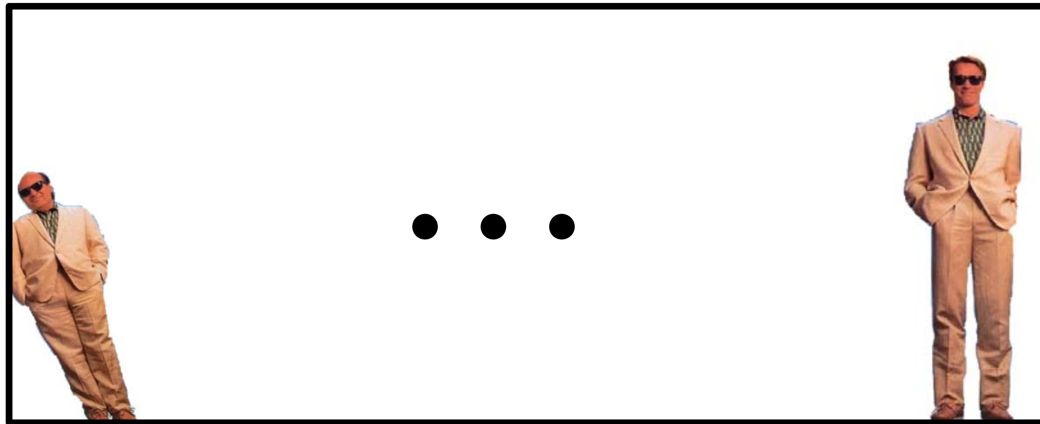
  `[10 2 5 30 4 8 19 102 53 3]`

- 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
```

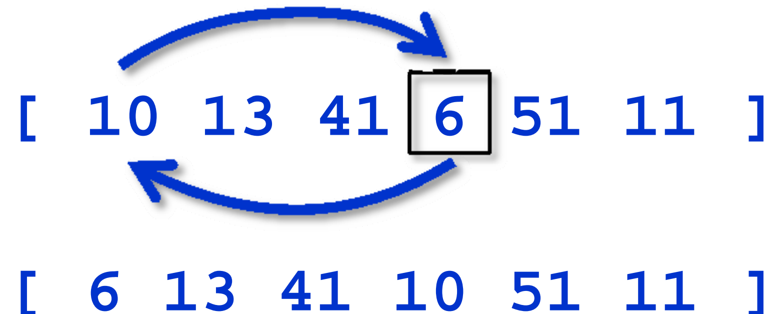
```
[ 10 13 41 6 51 11 ]
% After round 1,
%  m = 6, m_index = 4
```

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);
```

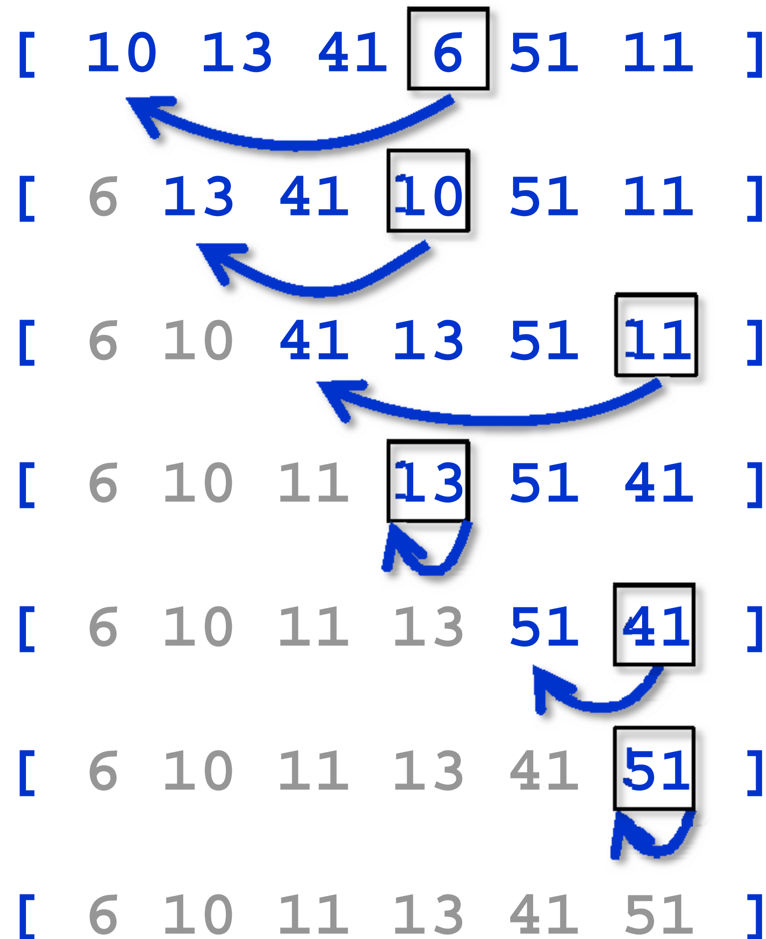
```
tmp = A(i);
A(i) = A(m_index);
A(m_index) = tmp;
```



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



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?

- 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. **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 ];
```

Cornell University

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?