Sorting and Selection, Part 1

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

Administrivia

- Assignment 1 due Friday by 5pm
  - Please sign up for a demo slot using CMS (or demo before Friday)

- Assignment 2 out on Friday
“Corner cases”

- Sometimes the input to a function isn’t what you expect
  - What is the maximum element of a vector of length 0?

- When writing a function, you should try and anticipate such corner cases

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?
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 (complexity)

- 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
Asymptotic complexity

- Big-O only cares about the number of operations as $n$ (the size of the input) grows large ($n \to \infty$)

![Graphs of $f(n)$ and $g(n)$]

O(1) \quad O(n)

Complexity classes

- Big-O doesn’t care about constant coefficients
  - “Constant of proportionality” doesn’t matter

$0.001n = O(n)$

$1,000,000n = O(n)$

O($n^2$)
What is the complexity of:

1. Finding the 2nd biggest element (> all but 1)?

2. Finding the element bigger than all but 2%?
   - Assume we do this by repeated “find biggest”

3. Multiplying two n-digit numbers (using long multiplication)?

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 $k^{th}$ biggest or smallest (with what time complexity?)

- Very important principle! (encapsulation)
  - Divide your problem into pieces
    - One person (or group) can provide $\text{sort}$
    - The other person can use $\text{sort}$
  - As long as both agree on what $\text{sort}$ does, they can work independently
  - Can even “upgrade” to a faster $\text{sort}$
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, put him/her 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

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

Algorithm 1

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

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

```matlab
m = A(i); m_index = i;
for j = i+1:n
    if A(j) < m
        m = A(j); m_index = j;
    end
end
```

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)

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

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

After swapping:
```
% 10 13 41 6 51 11
```

Correct swaps:
```
% 6 13 41 10 51 11
```
Putting it all together

function [ A ] = selection_sort(A)
% Returns a sorted version of array A
n = length(A);
for i = 1:n
    % Find the smallest element in A(i:len)
    m = A(i); m_index = i;
    for j = 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

\[
\begin{array}{c|c|c|c|c|c|c}
10 & 13 & 41 & 6 & 51 & 11 \\
6 & 13 & 41 & 10 & 51 & 11 \\
6 & 10 & 41 & 13 & 51 & 11 \\
6 & 10 & 11 & 13 & 51 & 41 \\
6 & 10 & 11 & 13 & 41 & 51 \\
6 & 10 & 11 & 13 & 41 & 51 \\
\end{array}
\]
**Speed of selection sort**

- Let \( n \) be the size of the array
- How fast is selection sort?
  
  \[
  O(1) \quad O(n) \quad O(n^2) \quad ?
  \]

- How many comparisons (\(<\)) does it do?
- First iteration: \( n \) comparisons
- Second iteration: \( n - 1 \) comparisons
  ...
- \( n^{th} \) iteration: 1 comparison

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

- Work grows in proportion to \( n^2 \) → selection sort is \( O(n^2) \)
Other ideas for sorting?