Introduction to Algorithms

William Hendrix

Lecture 1

Survey

Who here:

- **loves** mathematics?
- would take this course even if it were not required?
- has taken discrete math?
- has written a proof before?
- has written a three-page proof?
- remember how to do proof by induction?
- would like to have an A in this course?
- is willing to work hard to get an A in this course?
- prefers real applications to theory?
- expects that their job after graduation will involve coding?

Any questions you have for me?

Syllabus

- Read at: usflearn.instructure.com

- Contact: ENB 343F or whendrix@usf.edu
- Office hours: Mondays 12-2 (TBD), Tuesdays at 6:30 (ENB 343F), Wednesdays at 2:00 (ENB 343F), Thursdays at 10-12 (ENB 325)
- TAs: Renhao Liu (renhaoliu@mail.usf.edu), Yu Peng (yupeng@mail.usf.edu)

- Feedback forms
- Course objectives, grading scale, exams, etc.
- Attendance policy
- Piazza

A Word of Warning

- Algorithms is not an easy class

- It requires:
  - Strong mathematical background
    - Writing proofs
    - Understanding graphs
    - Solving recurrences
  - Strong understanding of data structures
    - Arrays, lists, trees, graphs, heaps, hash tables
  - Creativity!
  - Good work ethic and academic integrity

- The projects require:
  - Strong programming skills
    - C, C++, or Java

Resources

- Read the textbook
- Ask questions on Piazza
- Visit office hours
- Solve practice problems
  - From the textbook or the web
- Contact me via email or Canvas

What is an algorithm?

- **algorithm:** a well-defined, step-by-step procedure for solving a *problem*

- **problem:** general task in which we are given some **input** and need to compute some corresponding **output**
  - **Example**
    - The *minimum problem*: given a set of values that can be compared, output the smallest value in that set

- **instance:** a particular input for a problem
- **solution:** the corresponding output for a problem instance
  - **Examples**
    - (5, 3, 14) -> 3
    - (-1, -1, -1, -1, -1) -> -1
    - ("Gallup", "Trot", "Canter") -> "Canter"

What some examples of algorithms?

- **Computer programs**

- Filling out tax forms
- Manufacturing processes
- Medical diagnostic tests
- Registering for classes
- How you wake up and get ready in the morning
- Driving to school
- Playing soccer

- Any solution that can be codified as a sequence of steps and decisions

Let's look at a concrete example

- **Problem:** sorting a list of numbers
- **Algorithm:**

Input:

```
data:  an array of integers to sort
n:     the number of values in data
```

Output: permutation of `data` such that `data[i]` ≤ `data[i+1]` for all `i`

Pseudocode:

```
for i = 1 to n
  Let m be the location of the min value in the array data[i..n]
  Swap data[i] and data[m]
end
return data
```

Wait… Sudoku?

- Pseudocode ("sue-doe-code")
  - *pseudo*- (false)
  - Algorithm description between code and English
  - Code-like to *eliminate ambiguity*
  - English-like for *simplicity*

Back to our example…

- **Problem:** sorting a list of numbers
- **Algorithm:**

Input:

`data:   an array of integers to sort`

`n:      the number of values in data`

Output: permutation of `data` such that `data[i]` ≤ `data[i+1]` for all `i`

Pseudocode:

```
for i = 1 to n
  Let m be the location of the min value in the array data[i..n]
  Swap data[i] and data[m]
end
return data
```

Walking it out

- **Problem:** sorting a list of numbers
- **Algorithm:**

Input:

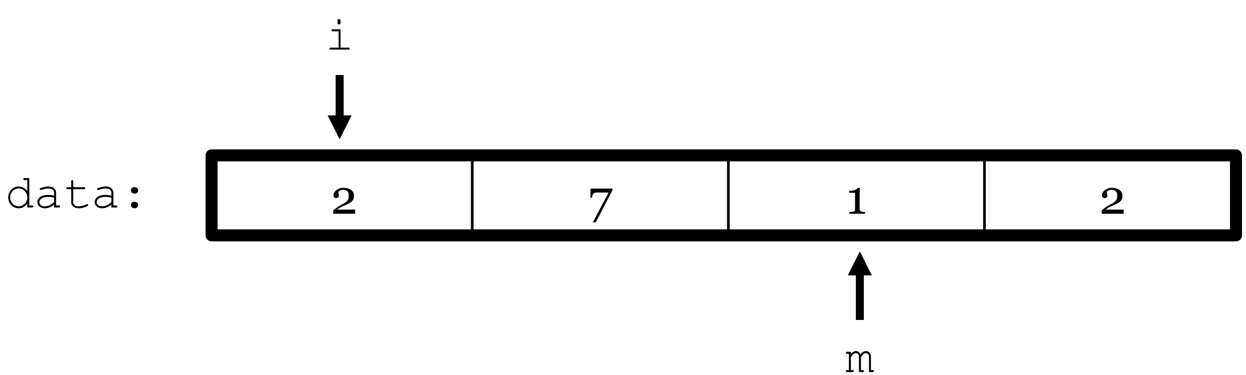
`data:   an array of integers to sort`

`n:      the number of values in data`

Output: permutation of `data` such that `data[i]` ≤ `data[i+1]` for all `i`

Pseudocode:

```
for i = 1 to n
  Let m be the location of the min value in the array data[i..n]
  Swap data[i] and data[m]
end
return data
```



Walking it out

- **Problem:** sorting a list of numbers
- **Algorithm:**

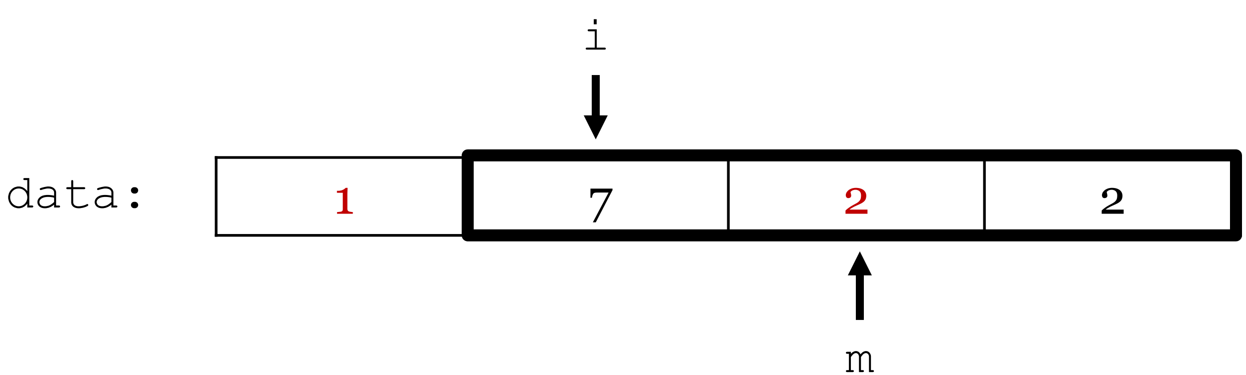
Input:

```
data:  an array of integers to sort
n:     the number of values in data
```

Output: permutation of `data` such that `data[i]` ≤ `data[i+1]` for all `i`

Pseudocode:

```
for i = 1 to n
  Let m be the location of the min value in the array data[i..n]
  Swap data[i] and data[m]
end
return data
```



Walking it out

- **Problem:** sorting a list of numbers
- **Algorithm:**

Input:

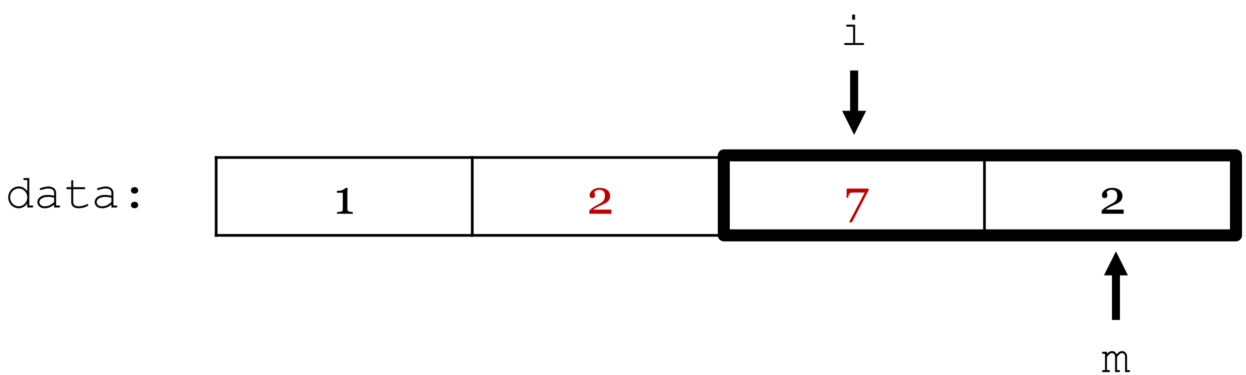
`data:` `an array of integers to sort`

`n:` `the number of values in data`

Output: permutation of `data` such that `data[i]` ≤ `data[i+1]` for all `i`

Pseudocode:

```
for i = 1 to n
  Let m be the location of the min value in the array data[i..n]
  Swap data[i] and data[m]
end
return data
```



Walking it out

- **Problem:** sorting a list of numbers
- **Algorithm:**

Input:

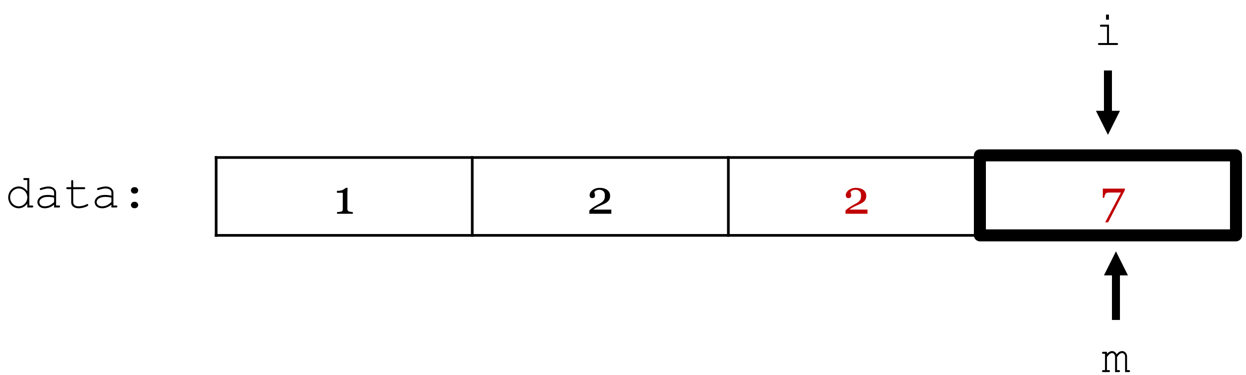
`data:   an array of integers to sort`

`n:      the number of values in data`

Output: permutation of `data` such that `data[i] ≤ data[i+1]` for all `i`

Pseudocode:

```
for i = 1 to n
  Let m be the location of the min value in the array data[i..n]
  Swap data[i] and data[m]
end
return data
```



What makes an algorithm good?

- **Correctness**
  - Always terminates
  - Always produces the correct output
  - Works for *every* problem instance

- **Efficiency**
  - Solves the problem quickly
    - Best case, average case, or worst case analysis
  - Uses few resources
    - Memory

- **Elegance**
  - Easy to understand
  - Easy to implement

How do we determine correctness?

- Proofs
  - Usually by **induction**
  - Or give **counterexample** to disprove

- Why don't we just test it?

1. Algorithm ≠ computer program
   - Writing programs is time-consuming and error-prone
2. Testing does not guarantee correctness

- Side note: why don't we prove that all programs work correctly?

1. Realistic programs are large and complex
2. The problem is not always easy to define
3. Code is *very* easy to misinterpret

Proving correctness

- **Problem:** sorting a list of numbers
- **Algorithm:** Selection Sort

Input:

```
data:  an array of integers to sort
n:     the number of values in data
```

Output: permutation of `data` such that `data[1] ≤ ... ≤ data[n]`

Pseudocode:

```
for i = 1 to n
  Let m be the location of the min value in the array data[i..n]
  Swap data[i] and data[m]
end
return data
```

How do we know this algorithm is correct?

- Answer depends on the algorithm!
- How does it solve the problem?
- Often prove by induction

Correctness of Selection Sort

- **Informal:** Each iteration of the `for` loop sorts one more element. After *n* iterations, the entire array is sorted.

- **More formal:** At the end of iteration *i* of the `for` loop, the first *i* items are less than or equal to everything that follows them. After *n* iterations, each `data[i]` will be less than or equal to `data[i+1]`.

Correctness of Selection Sort

- **Informal:** Each iteration of the **`for`** loop sorts one more element. After n iterations, the entire array is sorted.

- **More formal:** At the end of iteration i of the **`for`** loop, the first i items are less than or equal to everything that follows them. After n iterations, each `data[i]` will be less than or equal to `data[i+1]`.

Proof. In line 2 of the algorithm, we are selecting the minimum element from `data[i..n]`, and this element will be swapped with `data[i]` in line 3. Therefore `data[i]` will be less than or equal to all of the elements `data[i+1]`, `data[i+2]`, . . . , `data[n]` after iteration `i`. As this element cannot be selected by future iterations of the `for` loop, `data[i]` will not move afterwards.
At the end of the algorithm, every element `data[i]` will satisfy `data[i]` $\leq$ `data[i+1]` (except for `data[n]`), so `data[1]` $\leq \ldots \leq$ `data[n]`. $\square$

Coming up

- **Sign today's sign-in sheet!**
- **Feedback form 1** is posted on Canvas
  - Due at Exam 1
- **Homework 1** will be posted later this evening
  - Due next Tuesday

- **Recommended readings:** Chapter 1
- **Practice problems** (not required): Choose 1-2 problems from "Finding Counterexamples", "Proofs of Correctness", and "Induction" (problems 1-18 on pp. 27-29)