Lecture 1: Introduction and Proofs of Correctness

Sam McCauley

January 31, 2024

Welcome!

- I'm Sam
- This is algorithms (CS 256)
- Dialogue is encouraged! Please let me know if you have questions or comments.

What is This Course?

Day to day of Algorithms

- No coding in this class
- Focus is on high-level strategies (a.k.a. algorithms)
- English descriptions, pseudocode, proofs

Two Broad Questions about Algorithms

- Correctness: does this algorithm work?

- Running time: how fast is this algorithm?

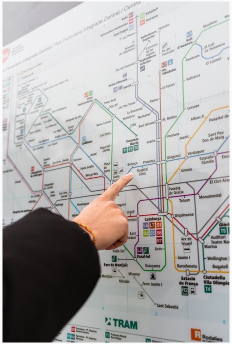
Why Algorithms?

1. Given a piece of code, or high-level strategy, does it work?
2. Does it *always* work?
3. Or: what does it do?
4. Is it fast?
5. If we move to another domain, will it still be fast?

Why Algorithms?

- It's a different way of thinking about computer science
- Some of you may use it a lot
- All of you (in my opinion) will benefit from having seen it

Proofs



- You and I will largely communicate via proofs
- Proofs: structure on top of intuition
- Remove *ambiguity*
- Strengthens intuition

Course Resources and Overview

Tools We'll Use

- Course website
- Overleaf/latex
- Gradescope

Questions about course resources?

Plan for Rest of Today

- Intro/review: reading pseudocode, expectations for proofs, etc.
- Use some likely-familiar algorithms as examples
  - And some algorithms that, probably, none of you have seen before
- Goal: Good foundation to get you started
- On Monday we'll move to the "Stable Matching" problem

Pseudocode

Pseudocode

- We will give algorithms in two ways in this course:
  - English descriptions, and
  - Pseudocode
- Code is a way for humans to *unambiguously* give computers instructions
- Pseudocode is a way for humans to communicate with each other
  - Keeps the structure of code
  - Does not rely on language-specific knowledge

Writing Pseudocode

- *Looks* very much like simple Python
- Basic keywords: `if`, `else`, `while`, etc.
- Basic arithmetic operations `+ - * / %`, use superscripts for exponents, write log
- Assume `0`-indexed arrays, inclusive `for` loops
- Explain any non-trivial steps in English
- Idea: make it as clear as possible!

Pseudocode Example 1

```
function findElement(A):
    minSoFar = A[0]
    for i = 1 to n-1:
        if A[i] < minSoFar:
            minSoFar = A[i]  # we found a new smallest
    return minSoFar
```

Pseudocode Example 2

It's OK to use sets in pseudocode if that's what you're comfortable with. Instead of library functions, write in English (if unambiguous!).

```
function findEven(A):
    B = ∅
    for x ∈ A:
        if x % 2 == 0:
            B = B ∪ {x}.
    Sort B using Merge Sort // O(n log n) time
    return B
```

This can be invaluable, but use carefully. Math notation is powerful, and some statements can be ambiguous or costly.

(Recall:) Two Questions about Algorithms

- Correctness: does this algorithm work?

- Running time: how fast is this algorithm?

Let’s start with correctness!

Algorithm Correctness

Correctness today

- We'll prove, in detail, that some algorithms are correct
- Some (but not all) review
- Correctness *can* be obvious, and is often omitted
  - We'll do some obvious proofs as practice
  - We'll talk about how short English explanations can be an effective alternative to formal proofs
  - We'll also do some non-obvious proofs

Algorithmic Invariants

Definition (Invariant)

If we stop an algorithm in the middle of its execution, what can we guarantee about its state?

- I love Invariants.
- Heart of all algorithms
- When looking at an algorithm for the first time, ask yourself what invariants it satisfies
- A proof by induction is a formal way of analyzing an invariant

Example 1: Selection Sort

```
selectionSort(A):
    for i = |A|-1 to 0:
        for j = 0 to i:
            if A[i] > A[j]:
                swap(A, i, j)

swap(A, i, j): // swaps A[i] and A[j]
    temp = A[i]
    A[i] = A[j]
    A[j] = temp
```

- What does the inner loop of selection sort do?
- *Intuitively*, in 1-2 sentences, why is this algorithm correct?
- How can we turn this into an inductive proof? [On Board #0]

Proofs in CS 256

- Proofs are a language for you to communicate with me
- Level of detail: judgment call
- Rule of thumb: imagine you're explaining to a skeptical classmate
    - They are trying to understand you; are willing to fill in details
    - But they are always asking questions
- Skeptical rubber duck explanation

Example 2: Insertion Sort

```
insertionSort(A):
    for i = 0 to |A| - 1:
        j = i
        while j > 0 and A[j-1] > A[j]:
            swap(A[j-1], A[j]) # swaps A[j-1] and A[j]
            j = j - 1
```

- What invariant can we guarantee after the outer loop executes *i* times?
- *Intuitively*, in 1-2 sentences, why is this algorithm correct?
- How can we turn this into an inductive proof? [On Board #1]

Insertion Sort Inductive Proof of Correctness

Theorem

After k iterations of the outer loop, the items in $A[0]$ through $A[k-1]$ are in sorted order.

Proof: By induction. **Base case:** for $k = 1$, $A[0]$ is always in sorted order.

Inductive step: Assume true for some $k \geq 1$. During the $k + 1$st iteration of the outer loop, the inner loop maintains that for any j: all items from $A[j]$ to $A[k]$ are in sorted order.

After the inner loop completes, all items from $A[0]$ to $A[j-1]$ are in sorted order, and are less than $A[j]$. Thus, when the $k + 1$st iteration of the outer loop completes, all items from $A[0]$ through $A[k]$ are in sorted order.

Insertion Sort 2-sentence Explanation of Correctness

The algorithm maintains the invariant that after k iterations of the outer loop, items in $A[0]$ through $A[k]$ are in sorted order. This is maintained because on the $k + 1$st iteration, the inner loop swaps $A[k + 1]$ with any larger element among the first k elements.