CSE 417 Autumn 2025

Lecture 2: Algorithm Correctness

Glenn Sun

Concept check quizzes

Many students haven’t done it yet 🙁

Remember: unlimited submissions, and are marked “incomplete” ***until you get every question right***.

One-time extension for everyone until **11:59pm tonight**!

Be sure to get those in on time going forward.

Homework 1

HW 1 out right after class, due next Friday at 11:59pm.

- **Problem 1 (Grading ChatGPT):** Read ChatGPT’s response to a question about stable matchings, and explain where the LLM made mistakes.

- **Problem 2 (Business profit):** Write a super simple algorithm for a basic task, and prove its correctness (today’s lecture!)

Video submission option

Optional! Written solutions are always accepted.

- Use a **whiteboard**, **paper**, or **tablet + stylus** to sketch key points while speaking, as if you were teaching a colleague.
- Please **do not** use slides, read a script, or edit your video.
- You may have some key points written before starting the video (e.g. problem statement, pseudocode, etc.)

Watch the example video on Canvas/course website!

Extension options

- **Problem 1X.1 (Stable matching verifier):** Correctly solve the question from Problem 1 by writing (part of) a Java program that performs a brute-force verification.

- **Problem 1X.2 (Using LLMs):** Reflect on your experience answering Problem 1 and share your thoughts on when it is appropriate to use LLMs.

Pick one to do!

Homework 0: Sample

HW 0 has been out on Canvas/course website since last night

Includes:

- One sample problem
- One sample written solution
- LaTeX source for the written solution
- One sample video solution

What is correctness?

Correctness defined

Algorithm: A list of unambiguous instructions to solve a class of computational problems

An algorithm is **correct** for a given problem if it has:

1. **Soundness:** Running it never raises exceptions/errors
2. **Termination:** All loops terminate
3. **Validity:** The output meets the problem specification

Gale–Shapley revisited (1/2)

Input: Proposers P, receivers R, and preference lists

Goal: A stable matching

```
1. while there is a free proposer $p \in P$ do
2.     Let $r$ be the best receiver that $p$ has not yet proposed to.
3.     if $r$ is also free then
4.         Have $r$ accept $p$.
5.     if $r$ is paired to $p'$ but prefers $p > p'$ then
6.         Have $r$ accept $p$ and also leave $p'$.
7. return all matches
```

Gale–Shapley revisited (2/2)

Q: What does “no exceptions” mean in Gale–Shapley?

A: In line 2, p has not yet exhausted their entire list.

Q: What does “loops terminate” mean in Gale–Shapley?

A: Every proposer gets matched in finite time.

Q: What does “meets specification” mean in Gale–Shapley?

A: The final set of matches is a stable matching.

Sum of an array (1/2)

Input: Array $A[1 \dots n]$ of numbers

Goal: The sum of the elements in A

```
1. Let $\mathbf{sum} = 0$.
2. for $i = 1, \dots, n$ do
3.     Update $\mathbf{sum} = \mathbf{sum} + A[i]$.
4. return $\mathbf{sum}$
```

Sum of an array (2/2)

Q: Explain why “no exceptions” is true for this algorithm.

A: The only line that can raise an exception is array access in line 3. By line 2, we have $\mathbf{1 \leq i \leq n}$ during line 3, so we are good.

Q: Explain why “loops terminate” is true for this algorithm.

A: For-loops always terminate!

Q: Explain why “meets specification” is true for this algorithm?

Seems obvious, but formally, we can **explain iteration-by-iteration**.

Loop invariants (1/3)

Instead of jumping to why something is true at the end of the loop, it is easier to break it down step-by-step.

We want: After iteration $\boldsymbol{n}$, we have $\mathbf{sum} = \boldsymbol{A}[\mathbf{1}] + \cdots + \boldsymbol{A}[\boldsymbol{n}]$.

Instead, show: After iteration $\boldsymbol{i}$, we have $\mathbf{sum} = \boldsymbol{A}[\mathbf{1}] + \cdots + \boldsymbol{A}[\boldsymbol{i}]$.

after iteration #: before $\mathbf{1}$ $\mathbf{2}$ $\boldsymbol{i-1}$ $\boldsymbol{i}$ $\boldsymbol{n}$

🏁

Loop invariants (2/3)

A **loop invariant** is property that is true:

- right before the loop starts, and
- after every iteration

Note: The "invariant" property *can* depend on the iteration index $\boldsymbol{i}$.

"Invariant" = "always stays true"

Example: After iteration $\boldsymbol{i}$, we have $\mathbf{sum} = \boldsymbol{A}[\mathbf{1}] + \cdots + \boldsymbol{A}[\boldsymbol{i}]$.

Loop invariants (3/3)

Example: After iteration $\boldsymbol{i}$, we have $\mathbf{sum} = \boldsymbol{A}[\mathbf{1}] + \cdots + \boldsymbol{A}[\boldsymbol{i}]$.

Proof. **Before the loop starts:** We set $\mathbf{sum} = \mathbf{0}$, which is the sum of an empty array.

After iteration $\boldsymbol{i}$:

- The previous iteration left us with $\mathbf{sum} = \boldsymbol{A}[\mathbf{1}] + \cdots + \boldsymbol{A}[\boldsymbol{i} - \mathbf{1}]$.
- We update $\mathbf{sum} = \mathbf{sum} + \boldsymbol{A}[\boldsymbol{i}]$, so at the end of this iteration, $\mathbf{sum} = \boldsymbol{A}[\mathbf{1}] + \cdots + \boldsymbol{A}[\boldsymbol{i}]$ as we wanted.

Summary so far

Algorithm correctness means "**no exceptions**", "**loops terminate**", and "**meets specification**".

For sum of an array, the first two are easy.

We proved that $\mathbf{sum} = \boldsymbol{A}[\mathbf{1}] + \cdots + \boldsymbol{A}[\boldsymbol{n}]$ after iteration $\boldsymbol{n}$ by breaking it down and proving $\mathbf{sum} = \boldsymbol{A}[\mathbf{1}] + \cdots + \boldsymbol{A}[\boldsymbol{i}]$ after every iteration $\boldsymbol{i}$.

Properties that stay true through a loop are called **loop invariants**.

Practice with more loop invariants

Max element in array (1/4)

Input: Array $A[1 \dots n]$ of numbers

Goal: The largest number in A (defined as $-\infty$ if A is empty)

```
1. Let $\mathbf{max} = -\infty$.
2. for $i = 1, \dots, n$ do
3.     if $\mathbf{max} < A[i]$ then
4.         Update $\mathbf{max} = A[i]$.
5. return $\mathbf{max}$
```

Max element in array (2/4)

Q: Explain why “no exceptions” is true for this algorithm.

A: Array access in lines 3-4 are within bounds, because line 2 gives $\boldsymbol{1 \leq i \leq n}$.

Q: Explain why “loops terminate” is true for this algorithm.

A: For-loops always terminate!

Max element in array (3/4)

Q: What do we need to show for “meets specification”?

A: At the end of the program, the variable $\mathbf{max}$ holds the largest number in the array $\boldsymbol{A}$.

Q: Can you break this down into a loop invariant that is true after every iteration?

A: After iteration $\boldsymbol{i}$, the variable $\mathbf{max}$ holds the largest number in the subarray $\boldsymbol{A}[\mathbf{1} \ldots \boldsymbol{i}]$.

Max element in array (4/4)

Claim. After iteration $\boldsymbol{i}$, the variable $\mathbf{max}$ holds the largest number in the subarray $\boldsymbol{A}[\mathbf{1} \dots \boldsymbol{i}]$.

Proof. **Before the loop starts:** The largest number in the empty subarray $\boldsymbol{A}[\mathbf{1} \dots \mathbf{0}]$ is defined to be $-\infty$, and $\mathbf{max} = -\infty$.

After iteration $\boldsymbol{i}$: By the previous iteration, $\mathbf{max}$ starts out holding the largest number in $\boldsymbol{A}[\mathbf{1} \dots \boldsymbol{i} - \mathbf{1}]$.

(Then let’s look at the code again to see what happens.)

Proving code with if-statements (1/3)

Input: Array $A[1 \dots n]$ of numbers

Goal: The largest number in A (defined as $-\infty$ if A is empty)

```
1. Let $\mathbf{max} = -\infty$.
2. for $i = 1, \dots, n$ do
3.     if $\mathbf{max} < A[i]$ then
4.         Update $\mathbf{max} = A[i]$.
5. return $\mathbf{max}$
```

What kinds of inputs trigger the if statement?

Break into cases based on this!

Proving code with if-statements (2/3)

The largest number of $\boldsymbol{A[1 \dots n]}$ is either in $\boldsymbol{A[1 \dots i-1]}$ or is $\boldsymbol{A[i]}$.

Case 1 (Largest number of $\boldsymbol{A[1 \dots n]}$ is in $\boldsymbol{A[1 \dots i-1]}$):

- Recall that by the previous iteration, $\mathbf{max}$ starts out holding the largest number in $\boldsymbol{A[1 \dots i-1]}$.
- This fact, with the case assumption, implies that $\mathbf{max} \geq \boldsymbol{A[i]}$.
- Thus, we don't enter the if statement and $\mathbf{max}$ doesn't change, which is correct.

Proving code with if-statements (3/3)

The largest number of $A[1 \dots n]$ is either in $A[1 \dots i-1]$ or is $A[i]$.

Case 2 (Largest number of $A[1 \dots n]$ is $A[i]$):

- Recall that by the previous iteration, $\mathbf{max}$ starts out holding the largest number in $A[1 \dots i-1]$.
- This fact, with the case assumption, implies that $\mathbf{max} < A[i]$.
- Thus, we enter the if statement and $\mathbf{max}$ is updated to $A[i]$, which is correct.

Small tangent on pseudocode

Pseudocode: Description of code intended for human reading.

Why write pseudocode over code?

- Easier to read and think about
- Communicate with people who don’t know Java

Why write pseudocode over English?

- Precise communication with no ambiguity

Pseudocode detail depends on context (1/2)

Input: Array $A[1 \ldots n]$ of numbers

```
1. for $i = 1, \ldots, n$ do
2.     Let $A[j]$ be the maximum
       element of $A[i \ldots n]$.
3.     Swap $A[i]$ and $A[j]$.
4. return $A$
```

✅

```
1. for $i = 1, \ldots, n$ do
2.     Let $\mathbf{maxIndex} = i$.
3.     for $j = i, \ldots, n$ do
4.         if $A[j] > A[\mathbf{maxIndex}]$ then
5.             Update $\mathbf{maxIndex} = j$.
6.     Let $\mathbf{temp} = A[i]$.
7.     Replace $A[i] = A[\mathbf{maxIndex}]$.
8.     Replace $A[\mathbf{maxIndex}] = \mathbf{temp}$.
9. return $A$
```

⚠️

Pseudocode detail depends on context (2/2)

Input: Array $A[1 \dots n]$ of numbers

```
1. Let $\mathbf{max} = -\infty$.
2. for $i = 1, \dots, n$ do
3.     if $\mathbf{max} < A[i]$ then
4.         Update $\mathbf{max} = A[i]$.
5. return $\mathbf{max}$
```

✅

```
1. return the maximum
   element of $A[1 \dots n]$
```

❌

Selection sort (1/6)

Input: Array $A[1 \ldots n]$ of numbers

Goal: A permutation of A that is sorted in decreasing order

```
1.  for $i = 1, \ldots, n$ do
2.      Let $A[j]$ be the maximum element of $A[i \ldots n]$.
3.      Swap $A[i]$ and $A[j]$.
4.  return $A$
```

Selection sort (2/6)

Q: Explain why “no exceptions” is true for this algorithm.

A: Two things:

1. Array access on i is within bounds because $1 \leq i \leq n$ (line 1).
2. Maximum element $A[j]$ exists because $i \geq 1$, so $A[1 \dots i]$ is nonempty.

Note: The concept of “error” in pseudocode is broader than code: whenever you say “let x be the …,” make sure it exists!

Q: “loops terminate”? A: For-loops always terminate!

Selection sort (3/6)

Q: What are some loop invariants that will help us show “meets specification”?

A: Here are some natural ideas:

1. After every iteration, array $\boldsymbol{A}$ is a permutation of the original.
2. After iteration $\boldsymbol{i}$, subarray $\boldsymbol{A}[\mathbf{1} \dots \boldsymbol{i}]$ is sorted in decreasing order.

Selection sort (4/6)

1. After every iteration, array A is a permutation of the original.

Proof. **Before the loop starts:** A is unchanged.

After each iteration: By the previous iteration, A starts out as a permutation of the original array.

Because we only modify A by swapping elements, it remains a permutation of the original at the end of this iteration.

Selection sort (5/6)

2. After iteration $\boldsymbol{i}$, subarray $\boldsymbol{A}[\mathbf{1} \dots \boldsymbol{i}]$ is sorted in decreasing order.

Proof. **Before the loop starts:** $\boldsymbol{A}[\mathbf{1} \dots \mathbf{0}]$ is empty.

After each iteration: By the previous iteration, $\boldsymbol{A}[\mathbf{1} \dots \boldsymbol{i} - \mathbf{1}]$ starts out sorted in decreasing order.

To show $\boldsymbol{A}[\mathbf{1} \dots \boldsymbol{i}]$ ends up sorted, we need $\boldsymbol{A}[\boldsymbol{i} - \mathbf{1}] \geq \boldsymbol{A}[\boldsymbol{i}]$.

(Then let's look at the code again to see what happens.)

Selection sort (6/6)

Input: Array $A[1 \ldots n]$ of numbers

Goal: A permutation of A that is sorted in decreasing order

```
1. for $i = 1, \ldots, n$ do
2.     Let $A[j]$ be the maximum element of $A[i \ldots n]$.
3.     Swap $A[i]$ and $A[j]$.
4. return $A$
```

Stuck because this iteration doesn't give any information about $A[i-1]$!

Instead, *strengthen the loop invariant* to know more about $A[i-1]$.

Strengthening the invariant (1/3)

2. After iteration $\boldsymbol{i}$, subarray $\boldsymbol{A[1 \dots i]}$ is sorted in decreasing order **and contains the $\boldsymbol{i}$ largest elements of $\boldsymbol{A}$**.

Proof. **Before the loop starts:** $\boldsymbol{A[1 \dots 0]}$ is empty and certainly contains the 0 largest elements of $\boldsymbol{A}$.

After each iteration: By the previous iteration,

1. $\boldsymbol{A[1 \dots i-1]}$ is sorted in decreasing order, and
2. $\boldsymbol{A[1 \dots i-1]}$ has the $\boldsymbol{i-1}$ largest elements of $\boldsymbol{A}$.

Need to prove both at the end of the iteration.

Strengthening the invariant (2/3)

After each iteration: By the previous iteration,

1. $A[1 \ldots i-1]$ is sorted in decreasing order, and
2. $A[1 \ldots i-1]$ has the $i-1$ largest elements of A.

To prove that $A[1 \ldots i]$ is sorted in decreasing order at the end:

By (2), $A[i-1] \geq A[j]$ where j is the maximum of $A[i \ldots n]$.

We swap $A[i]$ and $A[j]$, so at the end, $A[i-1] \geq A[i]$ and thus combined with (1), $A[1 \ldots i]$ is sorted in decreasing order.

Strengthening the invariant (3/3)

After each iteration: By the previous iteration,

1. $\boldsymbol{A[1 \dots i-1]}$ is sorted in decreasing order, and
2. $\boldsymbol{A[1 \dots i-1]}$ has the $\boldsymbol{i-1}$ largest elements of $\boldsymbol{A}$.

To prove that $\boldsymbol{A[1 \dots i]}$ has the $\boldsymbol{i}$ largest elements of $\boldsymbol{A}$ at the end:

Because $\boldsymbol{j}$ was the maximum of $\boldsymbol{A[i \dots n]}$, by (2) it must be the $\boldsymbol{i}$th largest element of $\boldsymbol{A}$.

After swapping, $\boldsymbol{A[1 \dots i]}$ has the $\boldsymbol{i}$ largest elements of $\boldsymbol{A}$.

Integer square root (1/2)

Input: A positive integer n

Goal: An integer k such that $(k-1)^2 < n \leq k^2$

```
1. Let $k = 1$.
2. while $k^2 < n$ do
3.     Update $k = k + 1$.
4. return $k$
```

Integer square root (2/2)

Q: Explain why “no exceptions” is true for this algorithm.

A: Nothing to show!

Q: Explain why “loops terminate” is true for this algorithm.

A: We can give an **upper bound on the number of iterations**.

After i iterations, we have $k = i + 1$. (This is a loop invariant!)

After $n - 1$ iterations, $k = n$. Check the while condition to see that $k^2 = n^2 \geq n$ (since $n \geq 1$), so the loop exits within $n - 1$ iterations.

Proving code with while loops (1/3)

To prove "meets specification" with a while loop, it's usually not enough to use loop invariants. Must use

loop invariant + while termination condition

Example:

- loop invariant: $(\boldsymbol{k} - \mathbf{1})^{2} < \boldsymbol{n}$
- termination condition: $\boldsymbol{n} \leq \boldsymbol{k}^{2}$

Together, these two facts are exactly the specification.

Proving code with while loops (2/3)

Claim. After every iteration, $(k-1)^2 < n$.

Proof. **Before the loop starts:** We have $k = 1$ and $n \geq 1$, so $(k-1)^2 = (1-1)^2 < 1 \leq n$ as desired.

After each iteration: By the while condition, the iteration started with $k^2 < n$. Then we incremented k, so at the end of the iteration, this fact says $(k-1)^2 < n$.

Didn't use previous iteration!

Can jump to loop end instead of arguing iteration-by-iteration.

Proving code with while loops (3/3)

Claim. After every iteration, $(k-1)^2 < n$.

Revised proof. By the while condition, each iteration started with $k^2 < n$. Then we incremented k, so at the end of the iteration, this fact says $(k-1)^2 < n$.

Either way, this property + loop termination condition proves "meets specification" for our algorithm.

Looking forward

In HW1 and HW2, we may ask you to state and proof loop invariants.

In homeworks after that, we recognize that this is very tedious, so feel free to just state the key loop invariants without proof (as long as you can prove them in your head).

Induction

In other courses, you may have heard of a similar concept called induction. Induction proves statements like

$$1 + 4 + 9 + 16 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

with a **base case** ($n = 1$) and **inductive step** (assume true for $i - 1$, prove for i).

If this is familiar to you, proving loop invariants is exactly induction on the number of iterations. Otherwise, **ignore this.**

Final reminders

HW1 released at 11:30am!

I have OH now-12:30pm:

- Meet at front of classroom, we'll walk over together
- CSE (Allen) 214 if you're coming later

Nathan has online OH 12–1pm:

- Link on Canvas/course website
- https://washington.zoom.us/my/nathanbrunelle