1 Proof Techniques

13 October 2025

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg

version `2025-10-14 14:07`

Learning Outcomes

Unit 1: ***Proof Techniques***

1. Know logical *proof strategies* for proving implications, set inclusions, set equalities, and quantified statements.
2. Be able to use *mathematical induction* in simple proofs.
3. Know techniques for *proving termination* and *correctness* of procedures.

Outline

1.1 Digression: Random Shuffle

Random shuffle

Notation: $[0..n) = \{0, 1, \dots, n-2, n-1\}$

- **Goal:** Put an array $A[0..n)$ of n numbers into random order.
  More precisely: Any ordering of the elements $A[0], \dots, A[n-1]$ should be equally likely.
- A natural approach yields the following code

```
procedure myShuffle(A[0..n))
    for i := 0, ..., n − 1
        r := randomInt([0..n)) // A uniformly random number r with 0 ≤ r < n.
        Swap A[i] and A[r] // Swap A[i] to random position.
    end for
```

- Intuitively: *All elements are moved to a random index, so the order is random . . . right?*

Clicker Question

Select all statements that apply to myShuffle (for you).

A I have seen this shuffling algorithm (or a very similar method) before.

B I can understand the pseudocode for myShuffle (so I would be able to do an example by hand).

C It can generate all possible orderings of A (depending on the random numbers).

D myShuffle produces all possible orderings with the same probability.

E Assuming randomInt gives (perfect) uniform random numbers in the given range, myShuffle generates any ordering with equal probability.

→ sli.do/cs566

Random shuffle

- **Goal:** Put an array $A[0..n)$ of n numbers into random order.
  More precisely: Any ordering of the elements $A[0], \ldots, A[n-1]$ should be equally likely.
- A natural approach yields the following code

---

procedure myShuffle($A[0..n)$)
    for $i := 0, \ldots, n-1$
        $r :=$ randomInt($[0..n)$) // *A uniformly random number* r *with* $0 \le r < n$.
        Swap $A[i]$ and $A[r]$ // *Swap* $A[i]$ *to random position.*
    end for

- Intuitively: *All elements are moved to a random index, so the order is random . . . right?*

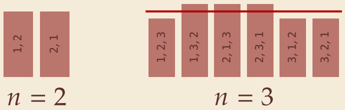
$n = 2$

Random shuffle

- **Goal:** Put an array $A[0..n)$ of n numbers into random order.
  More precisely: Any ordering of the elements $A[0], \ldots, A[n-1]$ should be equally likely.
- A natural approach yields the following code

```
procedure myShuffle(A[0..n))
    for i := 0, ..., n − 1
        r := randomInt([0..n)) // A uniformly random number r with 0 ≤ r < n.
        Swap A[i] and A[r] // Swap A[i] to random position.
    end for
```

- Intuitively: *All elements are moved to a random index, so the order is random ... right??*

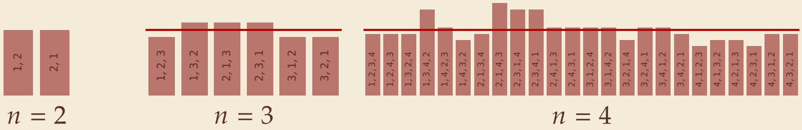


Random shuffle

- **Goal:** Put an array $A[0..n)$ of n numbers into random order.
  More precisely: Any ordering of the elements $A[0], \ldots, A[n-1]$ should be equally likely.
- A natural approach yields the following code

```
procedure myShuffle(A[0..n))
    for i := 0, ..., n − 1
        r := randomInt([0..n)) // A uniformly random number r with 0 ≤ r < n.
        Swap A[i] and A[r] // Swap A[i] to random position.
    end for
```

- Intuitively: *All elements are moved to a random index, so the order is random . . . right???*



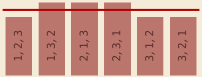
Random shuffle

- **Goal:** Put an array $A[0..n)$ of n numbers into random order.
  More precisely: Any ordering of the elements $A[0], \ldots, A[n-1]$ should be equally likely.
- A natural approach yields the following code

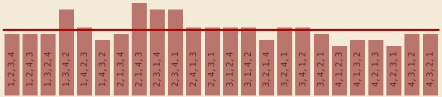
```
procedure myShuffle(A[0..n))
    for i := 0, ..., n − 1
        r := randomInt([0..n)) // A uniformly random number r with 0 ≤ r < n.
        Swap A[i] and A[r] // Swap A[i] to random position.
    end for
```

- Intuitively: *All elements are moved to a random index, so the order is random . . . right????*

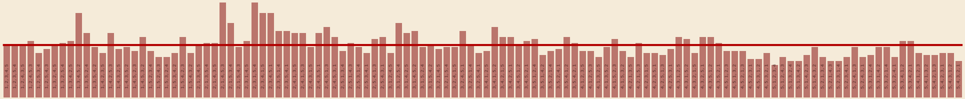
$n = 2$



$n = 3$



$n = 4$



$n = 5$

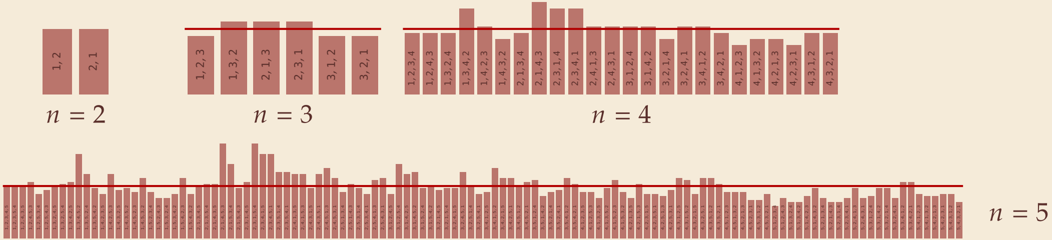
Random shuffle

- **Goal:** Put an array $A[0..n)$ of n numbers into random order.
  More precisely: Any ordering of the elements $A[0], \ldots, A[n-1]$ should be equally likely.
- A natural approach yields the following code

```
procedure myShuffle(A[0..n))
    for i := 0, ..., n − 1
        r := randomInt([0..n)) // A uniformly random number r with 0 ≤ r < n.
        Swap A[i] and A[r] // Swap A[i] to random position.
    end for
```

← **WRONG!**
DO NOT USE

- Intuitively: *All elements are moved to a random index, so the order is random . . . right????*



Clicker Question

Select all statements that apply to myShuffle (for you).

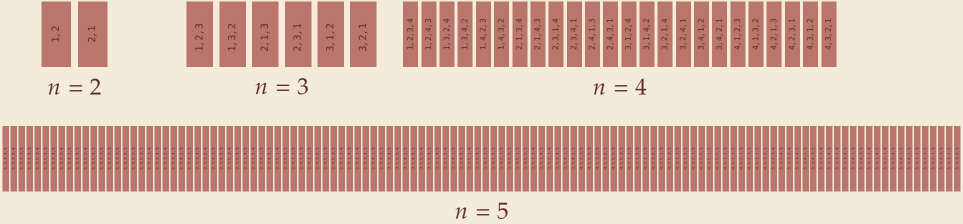
- **A** I have seen this shuffling algorithm (or a very similar method) before. ✓
- **B** I can understand the pseudocode for myShuffle (so I would be do an example by hand). ✓
- **C** It can generate all possible orderings of A (depending on the random numbers). ✓
- **D** ~~myShuffle produces all possible orderings with the same probability.~~
- **E** ~~Assuming randomInt gives (perfect) uniform random numbers in the given range, myShuffle generates any ordering with equal probability.~~

→ sli.do/cs566

Correct shuffle

- interestingly, a very small change corrects the issue

```
procedure shuffleKnuthFisherYates(A[0..n))
    for i := 0, ..., n − 1
        r := randomInt([i..n))
        Swap A[i] and A[r]
    end for
```



- looks good . . .
- . . . but how can we convince ourselves that it is correct, *beyond any doubt?*

1.2 Proof Templates

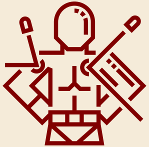
What is a *formal* proof?

A formal proof (in a logical system) is a **sequence of statements** such that each statement

1. is an *axiom* (of the logical system), or
2. follows from previous statements using the *inference rules* (of the logical system).

Among experts: Suffices to *convince a human* that a formal proof *exists*.

But: Use formal logic as guidance against faulty reasoning. $\rightsquigarrow$ bulletproof



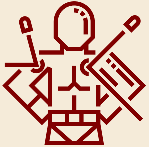
What is a *formal* proof?

A formal proof (in a logical system) is a **sequence of statements** such that each statement

1. is an *axiom* (of the logical system), or
2. follows from previous statements using the *inference rules* (of the logical system).

Among experts: Suffices to *convince a human* that a formal proof *exists*.

But: Use formal logic as guidance against faulty reasoning. $\rightsquigarrow$ bulletproof



Notation:

- Statements: $A \equiv$ "it rains", $B \equiv$ "the street is wet".
- Negation: $\neg A$ "Not A"
- And/Or: $A \wedge B$ "A and B"; $A \vee B$ "A or B or both"
- Implication: $A \Rightarrow B$ "If A, then B"; $\neg A \vee B$
- Equivalence: $A \Leftrightarrow B$ "A holds true *if and only if ('iff')* B holds true."; $(A \Rightarrow B) \wedge (B \Rightarrow A)$

Clicker Question

$A \Rightarrow B \equiv \neg A \vee B$

Is the following statement true?

"If $\underbrace{\text{the Earth is flat}}_{A}$, then $\underbrace{\text{ships can fall over its rim}}_{B}$."

A Yes

B No

C Neither

→ sli.do/cs566

Clicker Question

Is the following statement true?

"If the Earth is flat, then ships can fall over its rim."

A Yes ✓

B ~~No~~

C ~~Neither~~

→ sli.do/cs566

Implications

$\neg\neg A \equiv A$

To prove $A \Rightarrow B$, we can

$A \Rightarrow B \equiv \neg A \vee B$

$\equiv \neg(\neg B) \vee (\neg A)$

$\equiv \neg B \Rightarrow \neg A$

- directly derive B from A — *direct proof*
- prove $(\neg B) \Rightarrow (\neg A)$ — *indirect proof, proof by contraposition*
- assume $A \wedge \neg B$ and derive a contradiction — *proof by contradiction, reductio ad absurdum*
- distinguish cases, i. e., separately prove $(A \wedge C) \Rightarrow B$ and $(A \wedge \neg C) \Rightarrow B$. — *proof by exhaustive case distinction*

Clicker Question

n odd

$\Rightarrow \exists k \in \mathbb{N}_{\geq 0} : n = 2k+1$

Suppose we want to prove:
"If $n^2 \in \mathbb{N}_0$ is an even number, then n is also even."
For that we show that when n is odd, also n^2 is odd.
Which proof template do we follow?

$\Rightarrow n^2 = (2k+1)^2$

$= 4k^2 + 4k + 1$

$= 2\underbrace{(2k^2 + 2k)}_{k' \in \mathbb{N}_{\geq 0}} + 1$

$\Rightarrow n^2$ odd

- **A** direct proof: $A \Rightarrow B$
- **B** indirect proof: $(\neg B) \Rightarrow (\neg A)$ ✓
- **C** proof by contradiction: $A \land \neg B \Rightarrow \lightning$
- **D** proof by case distinction: $(A \land C) \Rightarrow B$ and $(A \land \neg C) \Rightarrow B$

→ sli.do/cs566

Clicker Question

Suppose we want to prove:
"If $n^2 \in \mathbb{N}_0$ is an even number, then n is also even."
For that we show that when n is odd, also n^2 is odd.
Which proof template do we follow?

A ~~direct proof: $A \Rightarrow B$~~

B indirect proof: $(\neg B) \Rightarrow (\neg A)$ ✓

C ~~proof by contradiction: $A \wedge \neg B \Rightarrow \lightning$~~

D ~~proof by case distinction: $(A \wedge C) \Rightarrow B$ and $(A \wedge \neg C) \Rightarrow B$~~

→ sli.do/cs566

Equivalences

To prove $A \Leftrightarrow B$,
we prove both implications $A \Rightarrow B$ and $B \Rightarrow A$ separately.

(Often, one direction is much easier than the other.)

Set Inclusion and Equality

To prove that a set S contains a set R, i. e., $R \subseteq S$,
we prove the implication $x \in R \Rightarrow x \in S$.

To prove that two sets S and R are equal, $S = R$,
we prove both inclusions, $S \subseteq R$ and $R \subseteq S$ separately.

1.3 Mathematical Induction

Quantified Statements

Notation

- Statements with parameters: $A(x) \equiv$ "x is an even number."
- Existential quantifiers: $\exists x : A(x)$ "There exists some x, so that $A(x)$."
- Universal quantifiers: $\forall x : A(x)$ "For all x it holds that $A(x)$."

  Note: $\forall x : A(x)$ is equivalent to $\neg\exists x : \neg A(x)$

Quantifiers can be nested, e. g., *ε-δ-criterion for limits*:

$$\lim_{x \to \xi} f(x) = a \quad :\Leftrightarrow \quad \forall \varepsilon > 0 \; \exists \delta > 0 : \left(|x - \xi| < \delta\right) \Rightarrow \left|f(x) - a\right| < \varepsilon.$$

To prove $\exists x : A(x)$, we simply list an example ξ such that $A(\xi)$ is true.

Clicker Question

Have you seen **proofs by *mathematical induction*** before?

- **A** Yes, could do it
- **B** Yes, but only vaguely remember
- **C** I've heard this term before, but ...
- **D** I have not heard "mathematical induction" before

→ *sli.do/cs566*

For-all statements

To prove $\forall x : A(x)$, we can

- derive $A(x)$ for an *"arbitrary but fixed value of x"*, or,
- for $x \in \mathbb{N}_0$, use ***induction***, i. e.,
  - prove $A(0)$, *induction basis*, and
  - prove $\forall n \in \mathbb{N}_0 : A(n) \Rightarrow A(n+1)$ *inductive step*

For-all statements

To prove $\forall x : A(x)$, we can

- derive $A(x)$ for an *"arbitrary but fixed value of x"*, or,
- for $x \in \mathbb{N}_0$, use ***induction***, i. e.,
  - prove $A(0)$, *induction basis*, and
  - prove $\forall n \in \mathbb{N}_0 : A(n) \Rightarrow A(n+1)$ *inductive step*

More general variants of induction:

- complete/strong induction
  inductive step shows $\big(A(0) \wedge \cdots \wedge A(n)\big) \Rightarrow A(n+1)$
- structural/transfinite induction
  works on any *well-ordered* set, e. g., binary trees, graphs, Boolean formulas, strings, ...

no infinite strictly decreasing chains

wohl-fundierte Ordnung / Noethersche Ordnung

1.4 Correctness Proofs

Formal verification

- verification: prove that a program computes the correct result

⇝ not our key focus in CS 566
but same techniques are useful for *reasoning* about algorithms

Here:

1. Prove that loop or recursive call eventually *terminates*.
2. Prove that a *loop* computes the *correct* result.

Proving termination

To prove that a recursive procedure $\texttt{proc}(x_1, \ldots, x_m)$ eventually terminates, we

- define a *potential* $\Phi(x_1, \ldots x_m) \in \mathbb{N}_0$ of the parameters
  (Note: $\Phi(x_1, \ldots x_m) \geq 0$ by definition!)

  $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$

  $\mathbb{N}_{\geq 1} = \{1, 2, \ldots\}$

- prove that every recursive call decreases the potential, i. e.,
  any recursive call $\texttt{proc}(y_1, \ldots, y_m)$ inside $\texttt{proc}(x_1, \ldots, x_m)$ satisfies

$$\Phi(y_1, \ldots, y_m) < \Phi(x_1, \ldots, x_m) \qquad \text{which means}$$

$$\Phi(y_1, \ldots, y_m) \leq \Phi(x_1, \ldots, x_m) - \mathbf{1}$$

Proving termination

To prove that a recursive procedure $\texttt{proc}(x_1, \ldots, x_m)$ eventually terminates, we

- define a *potential* $\Phi(x_1, \ldots x_m) \in \mathbb{N}_0$ of the parameters
  (Note: $\Phi(x_1, \ldots x_m) \geq 0$ by definition!)
- prove that every recursive call decreases the potential, i. e.,
  any recursive call $\texttt{proc}(y_1, \ldots, y_m)$ inside $\texttt{proc}(x_1, \ldots, x_m)$ satisfies

$$\Phi(y_1, \ldots, y_m) \ < \ \Phi(x_1, \ldots, x_m) \qquad \text{which means}$$
$$\Phi(y_1, \ldots, y_m) \ \leq \ \Phi(x_1, \ldots, x_m) \mathbf{- 1}$$

⇝ $\texttt{proc}(x_1, \ldots, x_m)$ terminates because
we can only strictly *decrease* the (integral) potential
a *finite* number of times from its initial value

- Can use same idea for a loop: show that potential decreases in each iteration.
  - ⇝ see tutorials for an example.

input $\longrightarrow$ $\boxed{\text{algorithm}}$ $\longrightarrow$ output

Hoare calculus

```
// state pre
program
// state post
```

pre condition

$\Downarrow$ program implies

post condition

```
// x ≥ 0
x := 5
// x == 5
```

Loop invariants

Goal: Prove that a *post condition* holds after execution of a (terminating) loop.

```
// (A) before loop
while cond do
    // (B) before body
    body
    // (C) after body
end while
// (D) after loop
```

For that, we

- find a *loop invariant I* (that's the tough part!)
- prove that I holds at (A)
- prove that $I \wedge cond$ at (B) imply I at (C)
- prove that $I \wedge \neg cond$ imply the desired post condition at (D)

Note: I holds before, during, and after the loop execution, hence the name.

Loop invariant – Example

- loop condition: $cond \equiv j < n$
- post condition (in line 13):
  $curMax = \max_{k\in[0..n-1]} A[k]$
  $[0..n)$
- loop invariant:
  $I \equiv \underbrace{curMax = \max_{k\in[0..\mathbf{j}-1]} A[k]}_{[0..j)} \wedge \underbrace{j \le n}$

```
1 // (A) before loop
2 while cond do
3     // (B) before body
4     body
5     // (C) after body
6 end while
7 // (D) after loop
```

```
1  procedure arrayMax(A,n)
2      // input: array of n elements, n ≥ 1
3      // output: the maximum element in A[0..n − 1]
4      curMax := A[0]; j := 1                          A[0..n)
5      // (A)
6      while j < n do
7          // (B)
8          if A[j] > curMax
9              curMax := A[j]
10         j := j + 1
11         // (C)
12     end while
13     // (D)
14     return curMax
```

We have to proof:

(i) I holds at (A) ✓

(ii) $I \wedge cond$ at (B) $\Longrightarrow$ I at (C) ✓

(iii) $I \wedge \neg cond$ $\Longrightarrow$ post condition ✓

ad (i) $curMax \overset{!}{=} \max_{k\in[0..j)} A[k] = A[0]$ ✓

$j = 1$

$j \overset{!}{\le} n$ ✓

$\overset{\shortparallel}{1}$

ad (iii) $j \le n \wedge j \ge n \Rightarrow j = n$

$curMax = \max_{k\in[0..j)} A[k] = \max_{k\in[0..n)} A[k]$

Loop invariant – Example

```
procedure arrayMax(A,n)
    // input: array of n elements, n ≥ 1
    // output: the maximum element in A[0..n − 1]
    curMax := A[0]; j := 1
    // (A)
    while j < n do
        // (B)
        if A[j] > curMax
            curMax := A[j]
        j := j + 1
        // (C)
    end while
    // (D)
    return curMax
```

$A[0..n)$

(ii) $I \wedge cond$ at (B) $\Rightarrow$ I at (C)

(iii) $I \wedge \neg cond$ $\Rightarrow$ post condition

ad (ii) $\quad curMax = \max\limits_{k \in [0..j)} A[k] \wedge j \leq n \wedge j < n$

1. Fall $\quad A[j] \leq curMax$

$$A[j] \leq curMax \underset{(I)}{=} \max_{k \in [0..j)} A[k] = \max_{k \in [0..j+1)} A[k]$$

$$curMax = \max_{k \in [0..j)} A[k] \quad (\text{da jetzt } j+1 \rightsquigarrow j)$$

$j < n$ bei (B) $\Rightarrow j \leq n-1 \Leftrightarrow j+1 \leq n$

$\Rightarrow$ bei (C) $j \leq n$

2. Fall $A[j] > curMax$

$j \leq n$ bei (C) genau wie oben

$$A[j] > curMax \underset{(I)}{=} \max_{k \in [0..j)} A[k]$$

$$\max_{k \in [0..j+1)} A[k] = A[j]$$

nach Zuweisung ist $curMax = A[j]$

$\Rightarrow$ $curMax = \max_{k \in [0..j)} A[k]$ bei (C)