1 Proof Techniques

15 October 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2024-10-15 13:40`

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

- **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
```

$[0..n) = \{0, 1, \ldots, n-1\}$

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

Clicker Question

▶ 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
```

[0..n) = {0, 1, ..., n-1}

Select all s

- **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, ..., 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?*

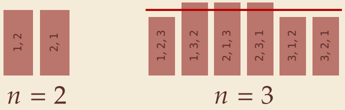
$n = 2$

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??*

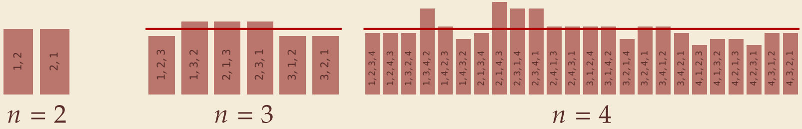


Random shuffle

- **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???*

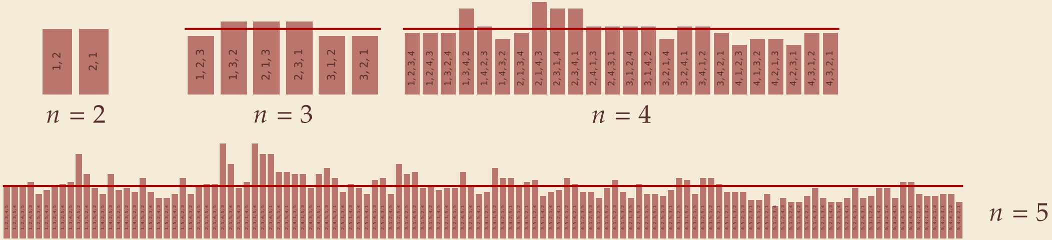


Random shuffle

- **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????*



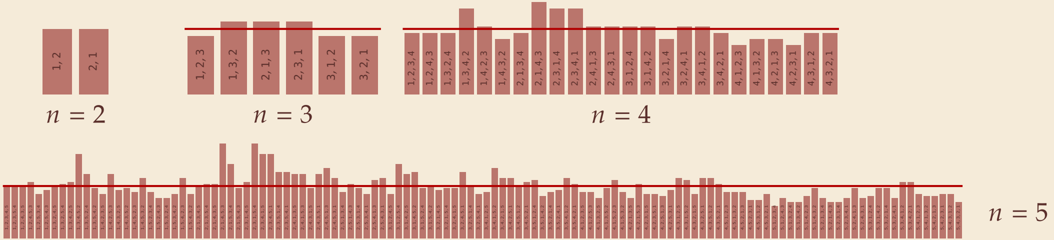
Random shuffle

- **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).

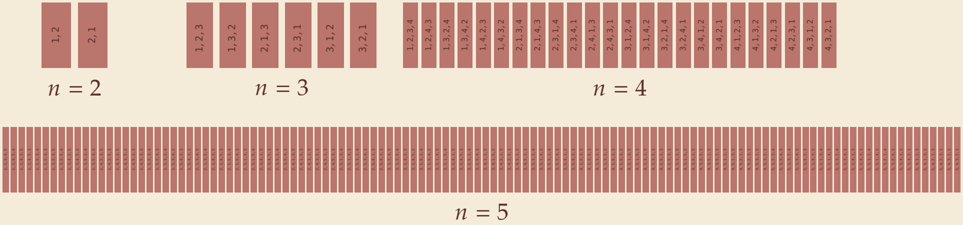
- A: I have seen this shuffling algorithm (or a very similar method) before. ✓
- B: I can understand the pseudocode for myShuffle (so I would be do an example by hand). ✓
- C: It can generate all possible orderings of A (depending on the random numbers). ✓
- D: ~~myShuffle produces all possible orderings 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

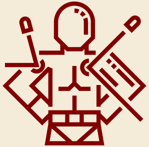
What is a *formal* proof?

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



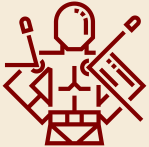
What is a *formal* proof?

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:

Aussagen

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

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

Clicker Question

Is the following statement true?

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

A ⇒ B

A Yes ✓

B ~~No~~

C ~~Neither~~

→ sli.do/cs566

Implications

To prove $A \Rightarrow B$, we can

- 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*

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

Clicker Question

n odd

$\rightsquigarrow \exists k : n = 2k+1$

$\rightsquigarrow n^2 = (2k+1)^2 = 4k^2 + 4k + 1$

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

$\Rightarrow n^2$ odd

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

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 $\neg\exists x : \neg A(x)$

$\forall x \in \mathbb{N} : \ldots$

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?

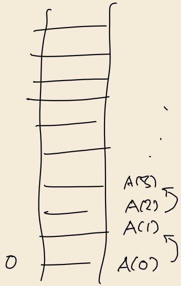
- A Yes, could do it
- B Yes, but only vaguely remember
- C I've heard this term before, but ...
- D I have not heard "mathematical induction" before

→ 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 vollständige Induktion
  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 `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 `proc`$(y_1, \ldots, y_m)$ inside `proc`$(x_1, \ldots, x_m)$ satisfies

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

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

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

$\mathbb{N}_{\geq 5} = \{5, 6, \ldots\}$

proc$(x_1, \ldots, x_m)$:
⋮
proc$(y_1, \ldots, y_m)$
⋮

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.

Loop invariants

Hoare Kalkül

{pre condition}
code ⇓
{post condition}

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):
  $curMax = \max_{k \in [0..n-1]} A[k]$
- loop invariant:
  $I \equiv curMax = \max_{k \in [0..j-1]} A[k] \wedge j \le n$

We have to proof:

(i) I holds at (A)

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

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

```
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
5      // (A)
6      while j < n do
7          // (B)
8          if A[i] > curMax
9              curMax := A[j]
10         j := j + 1
11         // (C)
12     end while
13     // (D)
14     return curMax
```

(i) $curMax = A[0] = \max_{k \in [0..j-1]} A[k]$

$= \max_{k \in [0..0]} A[k]$ ✓

$j = 1 \le n$ ✓

Loop invariant – Example

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

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

loop invariant:

$$I \equiv curMax = \max_{k \in [0..j-1]} A[k] \wedge j \leq n$$

```
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
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
```

(ii) Fallunterscheidung nach Bedingung in Z. 8

"I" **1. Fall** $A[j] > curMax$

bei (B)

$$A[j] > curMax \overset{I}{=} \max_{k \in [0..j-1]} A[k]$$

nach Z. 9 $curMax = A[j]$

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

nach Z. 10 $j \rightsquigarrow j+1$

$$curMax = \max_{k \in [0..j-1]} A[k]$$

✓

2. Fall $A[j] \leq curMax$

$$A[j] \leq curMax \overset{I}{=} \max_{k \in [0..j-1]} A[k]$$

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

nach Z. 10 $curMax = \max_{k \in [0..j-1]} A[k]$

✓

"I"

bei (B) $\quad j \le n \;\wedge\; j < n \;\rightsquigarrow\; j \le n-1$

nach Z.10 $\quad j \rightsquigarrow j+1$

$\Rightarrow \; j \le n$ bei (C) ✓

✓ (ii)

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

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

(iii) wir geben curMax zurück

(I) $\quad curMax = \max\limits_{k \in [0..j-1]} A[k]$

$j \le n \;\wedge\; \underbrace{\neg cond}_{\equiv\, j \ge n}$

$\rightsquigarrow \; j = n \quad \overset{(I)}{\rightsquigarrow} \quad curMax = \max\limits_{k \in [0..n-1]} A[k]$

✓ (iii)

```
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
5      // (A)
6      while j < n do
7          // (B)
8          if A[i] > curMax
9              curMax := A[j]
10         j := j + 1
11         // (C)
12     end while
13     // (D)
14     return curMax
```