0 Proof Techniques

28 September 2023

Sebastian Wild

COMP526 (Fall 2023)
University of Liverpool
version `2023-09-27 23:44`

Learning Outcomes

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.

Unit 0: ***Proof Techniques***

Outline

0.1 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
```

- 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/comp526`

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?*

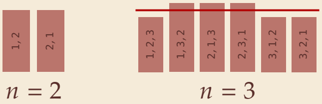
$n = 2$

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??*

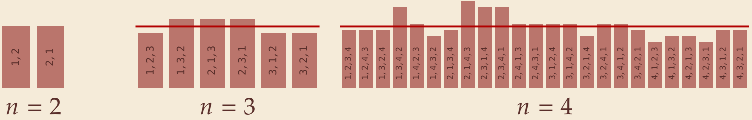


Random shuffle

- 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???*

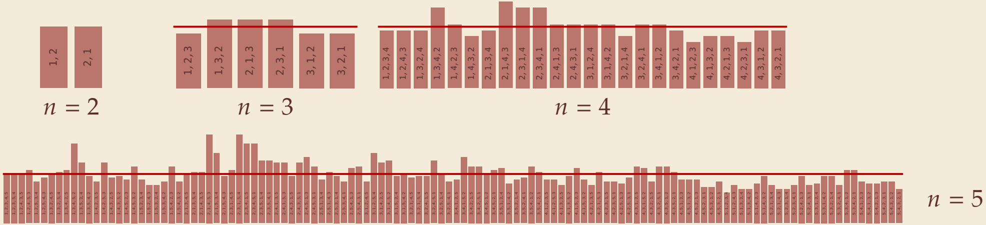


Random shuffle

- 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????*



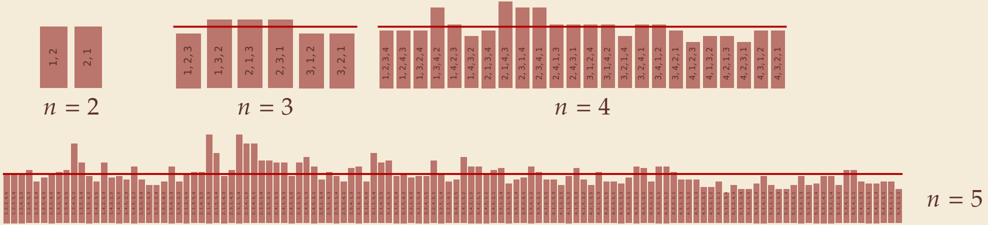
Random shuffle

- 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).

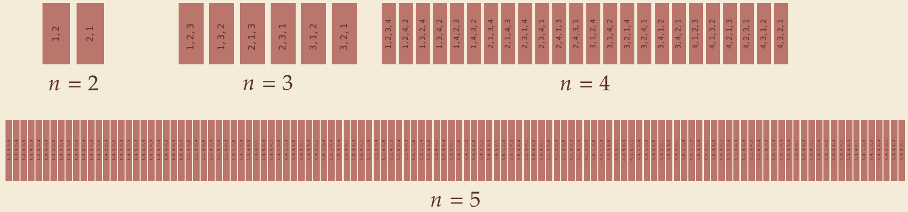
- **A** I have seen this shuffling algorithm (or a very similar method) before. ✓
- **B** I can understand the pseudocode for myShuffle (so I would be do an example by hand). ✓
- **C** It can generate all possible orderings of A (depending on the random numbers). ✓
- **D** ~~myShuffle produces all possible 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/comp526

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?*

0.2 Proof Templates

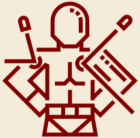
What is a *formal* proof?

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



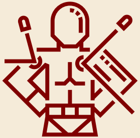
What is a *formal* proof?

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

Is the following statement true?

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

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

true

A Yes

B No

C Neither

→ sli.do/comp526

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/comp526`

Implications

To prove $A \Rightarrow B$, we can

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

$\neg B \Rightarrow \neg A$

- 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

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 $(A \wedge \neg C) \Rightarrow B$

n odd

$\Rightarrow n = 2k+1$ for some $k \in \mathbb{N}$

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

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

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

$\Rightarrow n^2$ odd

→ sli.do/comp526

Clicker Question

$A \Rightarrow B$

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 $(A \wedge \neg C) \Rightarrow B$~~

$\neg A$ $\neg B$

→ sli.do/comp526

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.

0.3 Mathematical Induction

Quantified Statements

A(6)

A(5)

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*:

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

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 ...

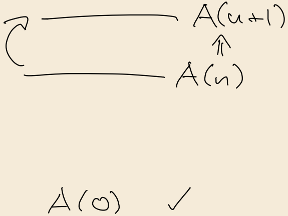
D I have not heard "mathematical induction" before

→ `sli.do/comp526`

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

  no infinite strictly decreasing chains

0.4 Correctness Proofs

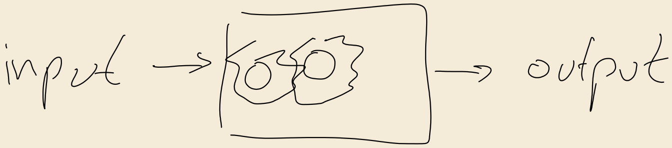
Formal verification

- verification: prove that a program computes the correct result

⇝ **not** our focus in COMP 526
but some 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!)

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

$\mathbb{N}_{\geq 1} = \{1, 2, 3, \ldots\}$

- 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

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

⇝ `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

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

```
I →  // (A) before loop
     while cond do
         // (B) before body
         body
         // (C) after body
I →  end while
I →  // (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.

pre condition —code→ post condition

(on variables, input, etc) (on variables & output)

Loop invariant – Example

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

```
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]; i := 1
5      // (A)
6      while i < n do
7          // (B)
8          if A[i] > curMax
9              curMax := A[i]
10         i := i + 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) $\Rightarrow$ I at (C) ✓

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

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

$i = 1 \leq n$ ✓

(ii) case distinction based on "$A[i] > curMax$" (in line 8)

case (a): true (new max at i)

$$\underline{A[i] > curMax} = \max_{k \in [0..i-1]} A[k] \quad (I)$$

line 9 changes $curMax$ to $A[i]$

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

line 10 changes i to $i+1$

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

red part of I holds at (C)

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

case (b) false $\quad A[i] \le curMax \overset{I}{=} \max\limits_{k \in [0..i-1]} A[k]$

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

after line 10

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

for $i \le n$: at (B) I : $i < n$

at line 10 $\quad i := i+1$

$\Rightarrow \quad i \le n$ ✓

√(i)

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

(iii) $I \wedge \neg cond$

$\equiv I \wedge i \geq n$

$\Rightarrow i = n$

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

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

post condition (in line 13):

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

- loop invariant:

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