

APPLIED ALGORITHMS \$ APPLIED
APPLIED ALGORITHMS \$
CS \$ APPLIED ALGORITHMI
DALGORITHMS \$ APPLIE
ED ALGORITHMS \$ APPLI
GORITHMICS \$ APPLIEDAL
HMICS \$ APPLIEDALGORIT
ICS \$ APPLIEDALGORITHM
IEDALGORITHMS \$ APPL
ITHMICS \$ APPLIEDALGOR
LGDALGORITHMS \$ APPLIE
MICS \$ APPLIEDALGORIT
ORITHMICS \$ APPLIE
PLIEDALGORITHMS \$ AP
PPLIEDALGORITHMS \$ A
RITHMICS \$ APPLIEDALGO
S \$ APPLIEDALGORIT
THMICS \$ APPLIEDALGORI



Proof Techniques

3 February 2022

Sebastian Wild

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 Proof Techniques

- 0.1 Proof Templates
- 0.2 Mathematical Induction
- 0.3 Correctness Proofs

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."; $(A \Rightarrow B) \wedge (B \Rightarrow A)$

0.1 Proof Templates

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*

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.

0.2 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*:

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


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

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

no infinite strictly decreasing chains

0.3 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 $\text{proc}(x_1, \dots, x_m)$ eventually terminates, we

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

$$\begin{aligned}\Phi(y_1, \dots, y_m) &< \Phi(x_1, \dots, x_m) && \text{which means} \\ \Phi(y_1, \dots, y_m) &\leq \Phi(x_1, \dots, x_m) - \mathbf{1}\end{aligned}$$

↪ $\text{proc}(x_1, \dots, x_m)$ 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.

```
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
```

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: $cond \equiv i < n$

▶ post condition (in line 13):

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

▶ loop invariant:

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

We have to prove:

- (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,  $n \geq 1$ 
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  $curMax$ 
```
