Lecture 1

Part 1: Introduction

Contents

- Course Information p.2
- Algorithm Design and Analysis Overview p.4
- Pseudo-Code, RAM model p.8
- Math and Program Correctness Proofs Review p.15

[document finalized]

Course Information

Computing Science Theory Courses

272 Formal Systems and Logic in Computing Science

An introduction to fundamental discrete structures used for the design and analysis of algorithms, including:

- Propositional and predicate logic
- Sets
- Functions and arithmetic
- Proofs by induction
- Pre- and post conditions, loop invariants
- Relations
- Graphs
- Number theory

204 Algorithms I

- Introduction to algorithms
- Analysis: correctness, worst/average/best case behaviour, asymptotical runtime
- Algorithms: sorting and searching, optimization, graph algorithms
- Design techniques: divide-and-conquer, dynamic programming, greedy

304 Algorithms II

- More advanced algorithms, and their design and analysis, complexity, notion of reduction, NP-completeness

474 Formal Languages, Automata and Computability

- More formal approach to models, complexity, and computability
- Computational limitations

Algorithm Design and Analysis Overview

Basic Concepts

- Algorithm: A well-defined step-by-step procedure that computes an output from a given input, i.e. computes a function
- Problem: Set of inputs satisfying certain properties for which an algorithm computes outputs with certain properties.
  Examples:
  - Given natural numbers x and y, compute $x \cdot y$
  - Sort integer array $A[0..n-1]$ in non-decreasing order
  - Given a map, find the shortest route between two points
- Problem Instance: a specific input for an algorithm that solves a problem
  Examples:
  - $(7, 8)$ is a problem instance for the multiplication algorithm
  - $A[0..4] = (0, -2, 4, 10, -5)$ is one for sorting

- Issues for a given algorithm
  - Correctness — does the algorithm conform to the input/output specification? This includes proving that the algorithm stops for all valid inputs.
  - Analysis of resource requirements
    Time, Space, Bandwidth, ...
  - Quality of the results (exact, approximation?, sometimes incorrect?)
  - Optimality in terms of required resources.
    E.g., finding the maximum element in an array of size n can't be done faster than linear time because every element must be visited at least once for the algorithm to be correct.
- Algorithm design concepts
  - Choosing the right data structures: how to store and organize for fast access and manipulation?
  - Design techniques:
    Greedy, Divide and conquer, Dynamic programming, exhaustive search, etc.

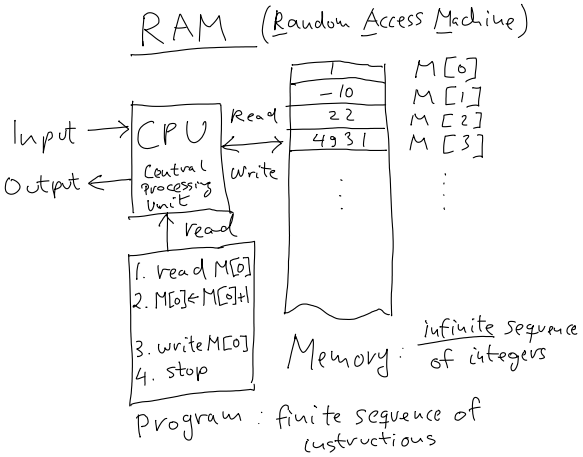
Methodologies for Analyzing Algorithms

- How do we know whether an algorithm is fast or space efficient?
- Several factors involved: implementation language, compiler, operating system, the way it is implemented, test data, computer hardware (CPU, memory, disk, etc.), and so on.
- The runtime often increases as the input size increases
- So one way is to measure time in terms of input size
- Then run experiments for different input sizes and see how the runtime increases

- Problem with experimental analysis:
  - We cannot run against all possible inputs, sometimes there are infinitely many!
  - Even inputs of the same size may result in different runtimes
  - Some factors (like CPU, memory, implementation, etc.) can vary significantly; so test results are very dependent on them.
- So we need an analytic way of measuring the runtime independent of the environmental factors (CPU speed, compiler, implementation, etc.)
- Idea:
  - select abstract computer model that is simpler but sufficiently close to modern computer hardware
  - express runtime on this model (number of execution cycles) in terms of input size

RAM Model and Pseudo-Code

Lecture 2 Model of Computation



- Components
  - Input device
  - Output device
  - CPU (Central Processing Unit)
  - M: memory locations (each can store an integer) M[0], M[1], M[2], ...

  - Program: fixed, finite, user-defined instruction sequence

- Properties
  - CPU has direct access to any mem location (by index)
  - move data between memory cells
  - compare data and branch
  - binary arithmetic operations
  - read from Input to memory
  - write from memory to Output
- Primitive operations:
  - assign a value to a memory cell
  - (conditional) jump to instruction
  - arithmetic operations $(+, -, *, /)$
  - comparisons

Example: RAM Program that computes $f(n) = n^n$ for $n \in \mathbb{N}$

```
1. read M[0]              // read number n into M[0]
2. M[1] ← M[0]            // copy M[0] into M[1] (counter)
3. M[2] ← 1               // initialize M[2] with 1 (prod.)
4. if M[1] ≤ 0 goto 8     // continue at 5. if M[1] > 0
5. M[1] ← M[1] - 1        // decrement counter by 1
6. M[2] ← M[2] * M[0]     // multiply product by M[0] (n)
7. goto 4                 // repeat
8. write M[2]             // print product
9. stop                   // done
```

This programming language resembles assembler languages used for modern CPUs such as AMD's x86 or Sun's SPARC architectures. It is Too cumbersome, we need something more high-level, with less emphasis on irrelevant details.

Pseudo-code representation of above RAM program:

```
// input:  natural number n
// output:  n^n

function powern(n)
counter ← n                  // assignment
product ← 1
while counter > 0 do
   counter ← counter − 1
   product ← product ∗ n
end
return product
```

This program representation is much more readable and easier to understand.

Describing Algorithms with Pseudo-Code

- Supported statements:

```
while a < b do     // while-loop, exited when cond. fails
   a ← a + 1       // assignment
end

for i ← 1 to n do  // for-loop
   ...             // loop body executed with i = 1, 2, ..n
end                // body not executed if n < 1

repeat             // repeat-loop
   ...             // loop body executed at least once
until a < b        // as long as condition is false

if a > b then      // if-then-else
   ...             // executed when condition true
else               // optional else-branch
   ...             // executed when condition false
end

sort(A[1..n])      // method calls with parameters

x ← max(3, 4, 5)   // function calls

return             // return to caller from method
return value       // return to caller from function
```

- Supported types: integers and Boolean values
- Supported operations
  - Integer operations $+ \quad - \quad * \quad /$ (rounding down)
  - Integer/Boolean relations $= \quad \neq \quad > \quad < \quad \geq \quad \leq$
  - Boolean connectives (AND OR NOT)
- Conventions
  - syntax will not be strict as long as the meaning is clear, e.g. the return type of functions will be implicit
  - indentation reflects block structure
  - // : remainder of line is a comment
  - array indexing: $A[i]$ for i-th cell of array A.

Another Pseudo-Code Example

```
// input:  integer array A[0..n-1]
// output:  maximum element in A[0..n-1]

function max(A[0..n − 1])
max ← A[0]
for i ← 1 to n − 1 do
   if A[i] > max then
      max ← A[i]
   end
end
return max
```

The following program is identical with the for-loop replaced by a while-loop:

```
function max(A[0..n − 1])
max ← A[0]
i ← 1
while i ≤ n − 1 do
   if A[i] > max then
      max ← A[i]
   end
   i ← i + 1
end
return max
```


Math and Program Correctness Proofs Review

Math Review

Here we review math concepts and notations introduced in CMPUT 272 that we rely upon in this course:

- logic
- sets
- functions
- summation
- proof techniques
- proving program correctness

If you are unfamilar with some of the concepts make sure you catch up right away, by consulting Wikipedia, appendixes of algorithm textbooks, or CMPUT 272 lecture notes. Also, get your algebra skills for manipulating equations in good shape. We will need it.

Logic

Predicate: a function mapping objects to true or false.

$P(n) :=$ "The program stops with input n"

Logical connectives: and ($\wedge$), or ($\vee$), not ($\neg$), implies ($\Rightarrow$), equivalent ($\Leftrightarrow$)

$$(P(x) \wedge \neg Q(x)) \Rightarrow R(x)$$

Quantified Expressions: Expressions made of predicates, logical connectives, and quantifiers ($\forall$: for all, $\exists$: exists)

"for all inputs n that are natural numbers, our algorithm halts"

written as: $\forall n \in \mathbb{N}\ H(n)$

"there exists a natural number n such that $n > 10$"

written as: $\exists n \in \mathbb{N}\ (n > 10)$

Sets: unordered collections of distinct objects

$$S = \{s_1, s_2, \ldots\}$$

s_i are called elements of S

$U = \{a, b, c\},\ V = \{a, c, d, e\},\ \emptyset$ empty set

Element relation: $x \in S$ indicates that x is an element of S, $x \notin S$ indicates it isn't.

$$a \in U,\ c \notin \emptyset$$

$|S|$: cardinality of S = number of elements in S

$$|U| = 3,\ |V| = 4,\ |\emptyset| = 0$$

Sequences: $A = (a_1, a_2, \ldots, a_n)$ "n-tupel"

ordered collection of items

(a_1, a_2) "pair"

$(a, b) \neq (b, a)$, but $\{a, b\} = \{b, a\}$

Set operations

Intersection: $U \cap V = \{x \mid x \in U \wedge x \in V\} = \{a, c\}$

"set of all x such that x is in U and in V"

Union: $U \cup V = \{x \mid x \in U \vee x \in V\} = \{a, b, c, d, e\}$

"set of all x such that x is in U or in V"

Difference: $U \backslash V = U - V = \{x \mid x \in U \wedge x \notin V\} = \{b\}$

"set of all x such that x is in U but not in V"

Numbers

$\mathbb{N} = \{0, 1, 2, \ldots\}$: natural numbers

$\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$: integers

$\mathbb{Q}$: rational numbers, e.g. $\frac{2}{3}, \frac{-7}{2}$

$\mathbb{R}$: real numbers

$\mathbb{R} \backslash \mathbb{Q}$ = irrational numbers, can't be expressed as integer fractions

$E.g.\ \sqrt{2} = 1.4142..., e = 2.718182..., \pi = 3.141592...$

Functions

functions map elements from the domain set to unique elements of the range set

$$f : \text{Domain} \to \text{Range}$$

$$x \mapsto f(x)$$

Important numerical functions mapping $\mathbb{R}$ to $\mathbb{R}$:

powers x^c, $c \in \mathbb{R}$ constant

examples: $x, x^2, x^{1/2} = \sqrt{x}$

polynomials $a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i$

$a_i \in \mathbb{R}$ constants, ex. $1 + 2x + 4x^2$

exponentials c^x, $c \in \mathbb{R}$ constant

$2^x, e^x$ $\quad c^0 = 1,\ c^1 = c,\ c^{-1} = 1/c,\ c^n c^m = c^{n+m}$

Logarithms $\log_c x, \quad c > 0$

"logarithm to base c of x"

Definition: $c^{\log_c(x)} = x$, ex. $2^{\log_2(x)} = x$

$\log_2(1024) = 10$, because $2^{10} = 1024$

$\log_3(9) = 2$, because $3^2 = 9$

Common logarithms:

$\log_2 \equiv \log, \lg$

$\log_e \equiv \ln$ ("natural logarithm", base $e = 2.718182...$)

Rules:

$$\log_c(1) = 0$$

$$\log_c(a \cdot b) = \log_c(a) + \log_c(b)$$

$$\log_c(d^x) = x \cdot \log_c(d)$$

$$\log_a(x) = \log_a(b) \cdot \log_b(x)$$

Sums

problem: compute $\sum_{i=1}^{1000} i = 1 + 2 + 3 + \cdots + 1000$

500 pairs: $1+1000, 2+999, 3+998, \ldots$ each worth 1001

sum $= 500 \cdot 1001 = 500500$

In general for n even:

$$\sum_{i=1}^{n} i = (n+1) \cdot \frac{n}{2}$$

For odd n:

Small example $n = 5$: $1 + 2 + 3 + 4 + 5 = (1 + 5) + (2 + 4) + 3$ (middle element 3)

$n = 1001:$ 500 pairs: $1 + 1001, 2 + 1000, 3 + 999, \ldots$ + middle element? (501)

$$\sum_{i=1}^{n} i = \underbrace{(n+1) \cdot \frac{n-1}{2}}_{\text{pairs}} + \underbrace{\frac{n+1}{2}}_{\text{middle}} = (n+1) \cdot \frac{n}{2}, \text{ as well}$$

Finite summation rules

$$\sum_i c \cdot a_i = c \sum_i a_i$$

$$\sum_i (a_i \pm b_i) = \sum_i a_i \pm \sum_i b_i$$

Example:

$$\sum_i 3(2^i + i) = 3 \sum_i 2^i + 3 \sum_i i$$

Lecture 3

Application: arithmetical progression series

$$\sum_{i=0}^{n} (a \cdot i + b) = a \sum_{i=0}^{n} i + \sum_{i=0}^{n} b$$

$$= a \sum_{i=0}^{n} i + b(n+1)$$

$$= a \cdot (n+1)\frac{n}{2} + b(n+1)$$

E.g., $\sum_{i=0}^{4}(2 \cdot i + 1) = 1 + 3 + 5 + 7 + 9 = ?$
$n = 4, a = 2, b = 1 \rightsquigarrow 25$

Problem: given x and n, evaluate geometric series

$$1 + x + x^2 + x^3 + \cdots + x^n = \sum_{i=0}^{n} x^i = \ ?$$

Example: what is $1 + 2 + 4 + 8$? $(x = 2, n = 3)$

To find a closed expression, we first give the sum a name:

$$(*) \quad S := 1 + x + x^2 + x^3 + \cdots + x^n = \sum_{i=0}^{n} x^i$$

Multiply both sides in $(*)$ by x

$$(**) \quad S \cdot x = x + x^2 + x^3 + \cdots + x^{n+1}$$

subtract $(*)$ from $(**)$ – all but two x terms cancel each other out:

$$S \cdot x - S = x^{n+1} - 1$$

$$\Rightarrow \quad S(x-1) = x^{n+1} - 1$$

$$(x \neq 1) \Rightarrow \quad S = \frac{x^{n+1} - 1}{x - 1}$$

If $x = 1$ then $S = n + 1$.

To summarize:

Theorem: For $x \neq 1$, $\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$,

and $n + 1$ if $x = 1$

Applications:

$$\sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n$$

$$= \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1,$$

e.g. $\sum_{i=0}^{3} 2^i = 1 + 2 + 4 + 8 = 16 - 1 = 15.$

$$\sum_{i=0}^{n} \frac{1}{3^i} = 1 + \frac{1}{3} + \frac{1}{9} + \cdots + \frac{1}{3^n}$$

$$= \frac{\frac{1}{3^{n+1}} - 1}{\frac{1}{3} - 1} = \frac{3}{2} - \frac{1}{2 \cdot 3^n}$$

This means that for increasing n the sum approaches $3/2$ from below, because $\frac{1}{2 \cdot 3^n}$ goes to 0 for $n \to \infty$.

Common Proof Techniques

In general, a proof is a sequence of inference steps that starts with a set of true statements such as axioms, assumptions, and previously proven theorems to establish the truth of a new theorem. Here we review three important proof techniques.

Direct Proofs

Direct proofs are formed by implication chains of the form $A \Rightarrow B \Rightarrow \cdots \Rightarrow T$ meaning that if statement A holds then statement T also holds (Note: in case A is false, we can't conclude anything about T).

Example:

Claim: The square of every even natural number is even.

Proof: By definition we know x even $\Leftrightarrow$ $\exists k \in \mathbb{N} : x = 2 \cdot k$. E.g. $16 = 2 \cdot 8$, so 16 is even.

[Here we use the existential quantifier $\exists$. The right-hand side reads "there exists a natural number k such that $x = 2k$]

Then, x even

$\stackrel{\text{def.}}{\Rightarrow} \exists k \in \mathbb{N} : x = 2 \cdot k$

$\stackrel{\text{square}}{\Rightarrow} \exists k \in \mathbb{N} : x^2 = (2k)^2 = 4k^2 = 2(2k^2)$

$\Rightarrow \exists k' \in \mathbb{N} : x^2 = 2 \cdot k'$ (choose $k' = 2k^2$ in previous line)

$\stackrel{\text{def.}}{\Rightarrow} x^2$ even □

Proof by Contraposition

Elementary logic tells us that $(A \Rightarrow B)$ is equivalent to $(\neg B \Rightarrow \neg A)$.

E.g. "It rains" $\Rightarrow$ "the road is wet" is equivalent to:

"The road is not wet" $\Rightarrow$ "It doesn't rain"

Thus, rather than showing $A \Rightarrow B \Rightarrow \cdots \Rightarrow T$ starting with a true statement A to establish the truth of T, we can also prove it by assuming $\neg T$ and showing $\neg T \Rightarrow \cdots \neg B \Rightarrow \neg A$.

Proof by Contradiction

Suppose we want to show that statement T holds.

If by assuming that T is false, we arrive at a contradiction by following a valid implication chain, we can conclude that T is actually true. Example:

Assume $\neg T$ ... Therefore Q ... Therefore $\neg Q$. Hence, $Q \wedge \neg Q$, a contradiction. Thus, $\neg T$ can't be true, and so T holds.

Example:

Claim: There are infinitely many prime numbers, i.e. natural numbers ≥ 2 that are only divisible by 1 and themselves $(2, 3, 5, 7, 11, 13 \ldots)$

Proof: Assume there are only finitely many prime numbers, say $p_1, p_2, \ldots, p_n$. Let

$$p = p_1 \cdot p_2 \cdots p_n + 1.$$

Then p is bigger than any p_i. Thus, p is not a prime number, because it is not on the list. On the other hand, p is not divisible by any p_i, because the remainder is always 1. Because all non-prime numbers can be decomposed into a product of primes, either p is a prime, or there are prime numbers which p can be decomposed into which are not on the list. In either case, this leads to a contradiction. Therefore, there are infinitely many prime numbers. □

Note: For increasing n, $p_1 p_2 \ldots p_n + 1$ does not generate all prime numbers, because it doesn't even always produce prime numbers! Smallest counterexample:

$$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 + 1 = 30031 = 59 \cdot 509$$

Mathematical Induction Proofs

The goal here is to prove that statements of the following form hold:

$$\forall n \in \mathbb{N} : P(n) \quad (*)$$

(read "for all natural numbers n, $P(n)$ is true.")

Examples:

$\forall n \in \mathbb{N} : even(n)$ obviously false, as 5 is not even. This a **counterexample** disproving the $\forall$ statement.

$$\forall n \in \mathbb{N} : \sum_{i=0}^{n} 2i \;=\; n(n+1) \quad \text{true}$$

One way to show statement $(*)$ is to prove that $P(0)$ holds (called the induction base (I.B.)) and then to show that for all n : $P(n)$ implies $P(n+1)$, i.e. we prove that if the induction hypothesis (I.H.) $P(n)$ holds, then $P(n+1)$ also holds (this is called the induction step (I.S.)). Both properties allow us to build an implication chain that proves that $P(n)$ holds for all n:

$$P(0) \Rightarrow P(1) \Rightarrow P(2) \Rightarrow \cdots \Rightarrow P(n)$$

We know that $P(0)$ is true, therefore $P(n)$ holds for all n.

A variation of this technique is to assume in the induction step that $P(0), \ldots, P(n)$ hold in order to prove $P(n+1)$ also holds. Both forms of mathematical induction are equivalent.

Induction Proof Example:

Claim: $\forall n \in \mathbb{N} : \underbrace{\sum_{i=0}^{n} 2i \;=\; n(n+1)}_{P(n)}$

Proof: By mathematical induction.

Induction Base (I.B.) $n = 0$

$\sum_{i=0}^{0} i \;=\; 0 = 0(0+1)$. So, $P(0)$ holds.

Induction Step (I.S.): from n to $n+1$

Suppose $P(n)$ holds, i.e. $\sum_{i=0}^{n} 2i \;=\; n(n+1)$ (*)

Knowing this, we want to show that $P(n+1)$ also holds, i.e.

$$\sum_{i=0}^{n+1} 2i \;=\; (n+1)(n+2)$$

(here we have replaced all occurrences of n in the definition of $P(n)$ by $n+1$)

Split up the sum: $\sum_{i=0}^{n+1} 2i \;=\; (\sum_{i=0}^{n} 2i) + 2(n+1)$ and plug in (*):

$$\sum_{i=0}^{n+1} 2i \;=\; n(n+1) + 2(n+1) \;=\; (n+1)(n+2)$$

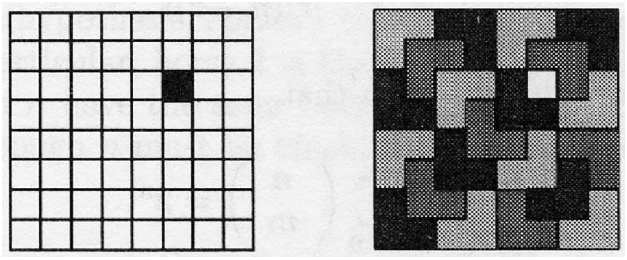
Thus $P(n+1)$ holds, and therefore $P(n)$ is true for all n. □

Another Example

This is an L-shaped tromino: 

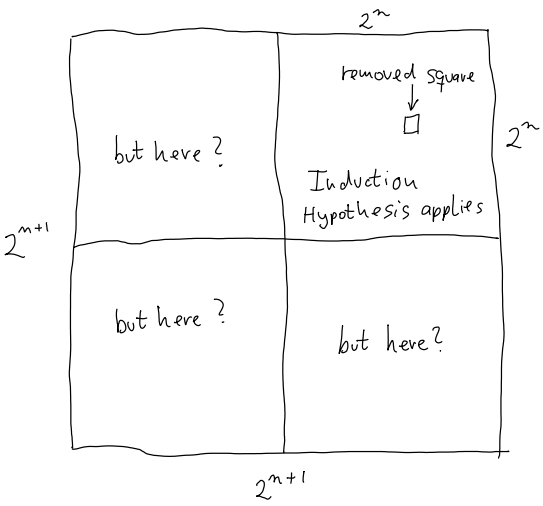
Question: Can chess boards of size 2^n by 2^n with one square removed be tiled with L-shaped trominos for all $n \geq 1$?

8 by 8 case and tiling:



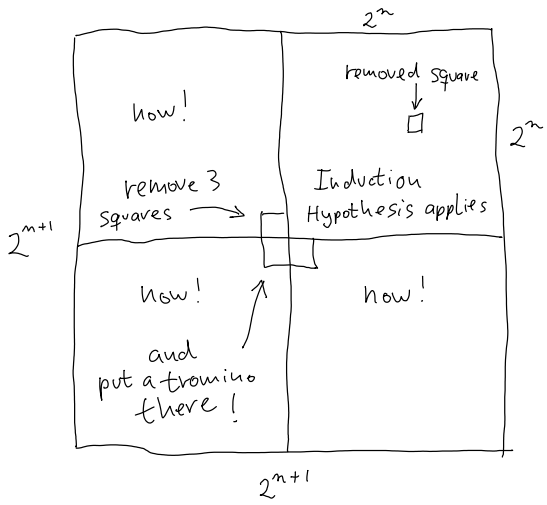
Try induction. The induction base $n = 1$ (2 by 2 board with one hole) is easy: tromino + hole.

Induction step: suppose all 2^n by 2^n boards with one removed square can be tiled. Show, that then also all 2^{n+1} by 2^{n+1} boards with one removed square can be tiled with trominos.



Doesn't seem to work because only one of the four quadrants has one square removed, and so the induction hypothesis can't be applied to the other three ... but

we could remove the squares in the other three quadrants that are closest to the center. Then the I.H. applies to these quadrants as well, i.e. we can tile them. What's left is to tile the 3 squares in the middle, which is easy.



So the induction step works, and therefore the claim is true for all n.

Lecture 4

Proving Program Correctness

Proving the correctness of programs is essential, because without it we can't be sure whether the program follows the output specification for infinitely many possible inputs.

We call a program correct if and only if

- it halts on every valid input, and
- the computed result meets the program's output specification.

Proving correctness of programs that use loops or recursion can be tricky. Here we review the Hoar logic approach to proving the correctness of while loops by considering pre- and post-conditions and loop invariants. If you have never had to prove programs correct before, visit the Hoar logic page on wikipedia.

Example:

```
// assume n > 0
// input: array of n numbers
// output: sum of all numbers
function sum(A[0..n-1])
sum <- 0
i <- 0
while i < n do
  sum <- sum + A[i]  // (*)
  i <- i + 1
end
return sum
```

Step 1: The program terminates for all inputs.

We note that i is initialized with 0 and then is incremented by 1 in each loop iteration. Therefore, after exactly n iterations it will reach n and the loop terminates. Therefore, the program stops for all inputs n.

Step 2: The program computes the correct value.

We can prove this claim by finding a suitable **loop invariant** φ that is true before the loop body is entered and continues to hold after the loop body has been executed:

$$\varphi \equiv (\texttt{sum} = \sum_{\texttt{j}=0}^{\texttt{i}-1} \texttt{A}[\texttt{j}] \ \wedge \ \texttt{i} \leq \texttt{n})$$

Claim 1. φ holds when entering the loop body for the first time [base case or initialization, usually easy to check].

Claim 2. if φ holds upon entering the loop body, then φ still holds after execution of the body. [maintenance or induction step]

Proof of claim 1 and 2: φ holds prior to entering the while-loop, because

$$\texttt{sum} = \sum_{\texttt{j}=0}^{0-1} \texttt{A}[\texttt{j}] = 0 \text{ and } 0 \leq \texttt{n}$$

Now suppose φ holds just before executing line (*):

$$\texttt{sum} = \sum_{\texttt{j}=0}^{\texttt{i}-1} \texttt{A}[\texttt{j}] \ \wedge \ \texttt{i} \leq \texttt{n}$$

We also know that $\texttt{i} < \texttt{n}$ because of the loop condition. Let sum0 be the value of sum at this time. Then after executing line (*)

$$\texttt{sum} = \texttt{sum0} + \texttt{A}[\texttt{i}] = (\sum_{\texttt{j}=0}^{\texttt{i}-1} \texttt{A}[\texttt{j}]) + \texttt{A}[\texttt{i}] = \sum_{\texttt{j}=0}^{\texttt{i}} \texttt{A}[\texttt{j}]$$

Now i gets incremented. Because $\texttt{i} < \texttt{n}$ prior to this step, we have

$$\texttt{sum} = \sum_{\texttt{j}=0}^{\texttt{i}-1} \texttt{A}[\texttt{j}] \ \wedge \ \texttt{i} \leq \texttt{n},$$

i.e. φ continues to hold after executing the loop body. □

Claim 3. φ and the loop exit condition implies that the correct value is computed.

Proof: when the loop exits, we know $\texttt{i} \geq \texttt{n}$. Together with φ this means $\texttt{i} = \texttt{n}$ and therefore

$$\texttt{sum} = \sum_{\texttt{j}=0}^{\texttt{n}-1} \texttt{A}[\texttt{j}]$$

which we wanted to prove. □