

Algorithm Correctness

We say that an algorithm is **correct** if for every input data that satisfy some condition, which is called the *precondition* of the algorithm, the output data satisfy a certain predefined condition, which is called the *postcondition* of the algorithm. We give an example:

```
DIV(A, B)
1. R := A / B
2. return R
```

For this algorithm we have :

Precondition : A, B are real numbers and $B \neq 0$.

Postcondition : $R = A / B$.

The proof of correctness of an algorithm has two parts:

1. Partial correctness. If the algorithm will terminate then it will give the right result (the result will satisfy the postcondition).
2. Termination. Proof that the algorithm terminates.

0. From Algorithms to Algebraic Relations

In order to prove the correctness we have to use mathematical relations and so we have to "translate" the algorithm to mathematical relations. Consider the following algorithm.

```
AL1(A, B)
1. S := A
2. S := S + B
3. S := S / 2
4. return S
```

We want to prove a relation of the result to the input data. Let s be the numerical value of the variable **S**. Similarly let a , b be the values of the variables **A** and **B**. We interpret the commands of the algorithm to algebraic relations (equalities):

1. $s = a$
2. $s = a + b$
3. $s = (a + b)/2$

We conclude that the algorithm returns the average of A and B.

Remark: The symbol $:=$ in the algorithm is an **assignment** while the symbol $=$ in the Algebra means an **equality**, therefore we cannot write $s = s + b$.

1. Dealing with if-else structures

The following algorithm contains an `if - else` structure.

```

ABS(A)
1. if (A < 0)
2.   R := -A
3. else
4.   R := A
5. return A

```

We will denote a, r the values of A, R respectively. The formal way to interpret the previous algorithm into algebraic relations is :

- 2. $a < 0 \rightarrow r = -a$
- 4. $a \geq 0 \rightarrow r = a$

Both cases can be rewritten as

- 2. $a < 0 \rightarrow r = |a|$
- 4. $a \geq 0 \rightarrow r = |a|$

and using the rule of inference:

$$\frac{\begin{array}{l} (p \wedge \text{condition}) \rightarrow q \\ (p \wedge \neg\text{condition}) \rightarrow q \end{array}}{p \rightarrow q}$$

we have

$$(\text{true})\{\text{ABS}\}(r = |a|).$$

For the rest of the notes we will use an informal way to deal with such structures.

2. Dealing with Loops

```

AL2(A[], M)
1. S := 0
2. I := 0
3. while (I < M)
4.   S := S + A[I]
5.   I := I + 1
6. return S

```

In this case we cannot say: let s be the numerical value of the variable S , because we have to determine in which iteration S has this value. In order to deal with this problem we use an index for the value s , to indicate the number of iteration, so we say:

Let s_n be the numerical value of the variable **S** at line 3, which is the beginning (-end) of the loop, after the n iteration. We denote s_0 the value before the loop. Similarly let i_n the value of the variable **I** after the n iteration. Observe that we don't change the values of $A[i]$, so we don't need indexes for these variables. Let $a[i]$ be the value of $A[i]$.

We interpret the commands of the previous algorithm to algebraic relations (equalities):

1. $s_0 = 0$
2. $i_0 = 0$
4. $s_n = s_{n-1} + a[i_{n-1}]$
5. $i_n = i_{n-1} + 1$

Obviously the variable **I** counts the number of iterations that have occurred.

Claim 1. $i_n = n$

Proof. By induction:

Base case: For $n = 0$ we have $i_0 = 0$ from line 2.

Inductive step. Suppose that the claim is true for $n = k$. Then from line 5,

$$i_{k+1} = i_k + 1 = k + 1.$$

□

A similar proof shows that s is the sum of the values $a[i]$. We assume (*precondition*) that $M > 0$.

Claim 2. $s_n = \sum_{j=0}^{n-1} a[j]$ for all $n \geq 1$.

Proof. By induction:

Base case: For $n = 1$ we have $s_1 = s_0 + a[0] = 0 + a[0] = a[0]$.

Inductive step. Suppose that the claim is true for $n = k$. Then from line 6,

$$s_{k+1} = s_k + a[i_k].$$

From claim 1 we know that $i_k = k$, so

$$\begin{aligned} s_{k+1} &= s_k + a[i_k] \\ &= s_k + a[k] \\ &= \sum_{j=0}^{k-1} a[j] + a[k] \\ &= \sum_{j=0}^k a[j] \end{aligned}$$

and the proof is complete. □

The previous two claims are proved to be true for every iteration. Assertions like these, that are true at the beginning and end of each iteration, independently of the number of iteration, are called **Loop Invariants**.

Problem 3. *Compute the loop invariants for I, S in the following algorithm:*

```

AL3(A[], M)
1. S := 0
2. I := 0
3. while (I < M)
4.   S := S + A[I]
5.   I := I + 2
6. return S

```

What is the return value?

3. Further Discussion on Loop Invariants

Consider the following algorithm.

```

AL4(A, M)
1. B := A
2. E := M
3. R := 1
4. while (E > 0)
5.   if (E is odd)
6.     R := R * B
7.     E := E - 1
8.   else
9.     B := B * B
10.    E := E / 2
11. return R

```

For this algorithm we have the precondition that A is real and M is a positive int. Let b_n, e_n, r_n be the values of the variables B, E, R at line 4 after the n iteration. Then we have

Claim 4. $r_n \cdot b_n^{e_n} = a^m$ and $e_n \geq 0$.

Proof. By induction:

Base case: For $n = 0$ we have $b_0 = a$, $e_0 = m$, $r_0 = 1$ and obviously

$$r_0 \cdot b_0^{e_0} = a^m.$$

Inductive step. Suppose that the claim is true for $n = k$. We have two cases:

Case 1. e_k is odd. Then

$$r_{k+1} = r_k \cdot b_k,$$

$$e_{k+1} = e_k - 1$$

and

$$b_{k+1} = b_k.$$

So

$$\begin{aligned} r_{k+1} \cdot b_{k+1}^{e_{k+1}} &= r_k \cdot b_k \cdot b_k^{e_k-1} \\ &= r_k \cdot b_k \cdot b_k^{e_k-1} \\ &= r_k \cdot b_k^{e_k} \\ &= a^m. \end{aligned}$$

Case 2. e_k is even. Then

$$r_{k+1} = r_k,$$

$$e_{k+1} = e_k/2$$

and

$$b_{k+1} = b_k^2.$$

So

$$\begin{aligned} r_{k+1} \cdot b_{k+1}^{e_{k+1}} &= r_k \cdot (b_k^2)^{\frac{e_k}{2}} \\ &= r_k \cdot b_k^{e_k} \\ &= a^m. \end{aligned}$$

In both cases the inductive step holds and the claim is proved. \square

4. Correctness for recursive functions

FIBB(n)

1. if (n < 3)
2. return n-1
3. return FIBB(n-1)+FIBB(n-2)

The precondition for this algorithm is that n is a positive integer. Here we will denote $f(n)$ the return value of the algorithm for an input value n . Then clearly

$$f(1) = 0, \quad f(2) = 1$$

and

$$f(n) = f(n-1) + f(n-2), \quad \text{for } n \geq 3.$$

Problem 5. *Prove using (strong) induction that the previous algorithm returns the n Fibonacci number.*

Problem 6. *Write a non recursive algorithm that gives the same result as the previous. Compute the loop invariants and give a correctness proof. What is the complexity of the new algorithm as opposed to the (exponential) complexity of the recursive algorithm?*

5. Proving the Termination of an Algorithm

Proposition 7. *A strictly increasing (decreasing) sequence of integers has no upper (lower) bound.*

Proposition 8. *A sequence of integers a_n for which we have $a_{n+1} - a_n \geq e > 0$ has no upper bound. Similarly if we have $a_{n+1} - a_n \leq e < 0$ has no lower bound.*

Let us examine the algorithm of section 3 and give a proof of termination. The proof will be focused on the termination of the loop, since the other part of the code obviously terminates. For the value e_{n+1} of the variable E we have

$$\begin{aligned} e_{n+1} &= e_n - 1, \text{ if } e_n \text{ is odd,} \\ e_{n+1} &= e_n/2, \text{ if } e_n \text{ is even.} \end{aligned}$$

In both cases we have

$$e_{n+1} < e_n.$$

According to the previous proposition, the sequence $\{e_n\}$, as defined above, has no lower bound and so there exist some value $e_k \leq 0$. We pick k to be the first such index that satisfies the inequality. For this k the condition of the loop $e_k > 0$ does not hold and thus the loop terminates.

Lets compute now what is the value e_k . From the previous conditions we have

$$\begin{aligned} e_k &\leq 0 \\ e_{k-1} &> 0 \end{aligned}$$

Now if $e_{k-1} > 1$ the either $e_k = e_{k-1} - 1 > 0$ or $e_k = e_{k-1}/2 > 0$. Both cases lead to a contradiction, so $e_{k-1} = 1$ and $e_k = e_{k-1} - 1 = 0$. From the loop invariant $r_n \cdot b_n^{e_n} = a^m$ we have

$$r_k \cdot b_k^{e_k} = a^m \Rightarrow r_k b_k^0 = r_k = a^m$$

So the return value of the algorithm is the m power of a . This completes the proof of correctness for this algorithm.

6. Problems

Problem 9. Prove that the next algorithm is correct for the following conditions:

Precondition : $K < M \wedge X \in A[K, \dots, M]$.

Postcondition : $X \notin A[K, \dots, I - 1] \wedge X \in A[I, \dots, M]$.

```

AL6(A[], K, M, X)
1. I := K
2. while (X != A[I])
3.     I := I + 1
4. return I

```

Problem 10. What is the return value of the following algorithm? Give a proof of correctness.

```

COMB(N, K)
1. if (K = 1)
2.     return N
3. if (N = K)
4.     return 1
5. return COMB(N-1, K-1)+COMB(N-1, K)

```

Problem 11. The following algorithm computes the value of a polynomial of the form:

$$a_0 + a_1x + \dots + a_dx^d.$$

Find the loop invariants and give a proof of correctness.

```

HORNER(A[], D, X)
1. total := 0;
2. I := D
2. while (I >= 0)
3.     total := A[I] + (total * X)
4.     I := I - 1
5. return total

```

Problem 12. *What are the loop invariants for the following algorithm? Give a proof of termination.*

```

    binary_search( to_find, elements[])
1.  lowpoint = 1
2.  highpoint = size of list of elements
3.  while lowpoint < highpoint
4.      pivot = lbound(( lowpoint+ highpoint)/ 2)
5.      if (to_find > elements[ pivot])
6.          lowpoint = pivot + 1
7.      else
8.          highpoint = pivot
9.  if to_find = elements[lowpoint] then
10.     return lowpoint
11. else
12.     return NOT FOUND

```

Problem 13. *What is the return value of the following algorithm when the initial parameter is the root node of a tree? Give a proof.*

```

    tree_depth_rec(node)
1.  if (node == null)
2.      return 0;
3.  subtree_max_depth = 0
3.  while (node.has_childs())
4.      next_node = node.next_child()
5.      subtree_depth = tree_depth_rec(node.left_child)
6.      if (subtree_depth > subtree_max_depth)
7.          subtree_max_depth = subtree_depth
8.  return (subtree_max_depth + 1)

```