ETH Zürich
Chair of Software Engineering

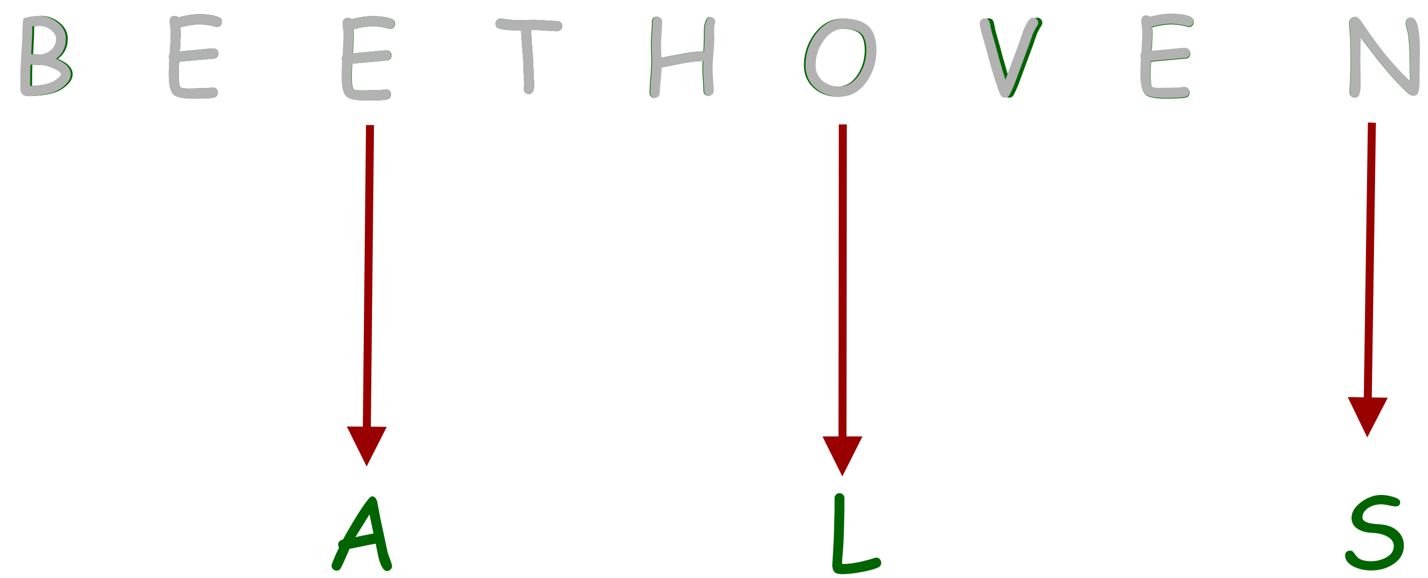
Software Verification

Bertrand Meyer

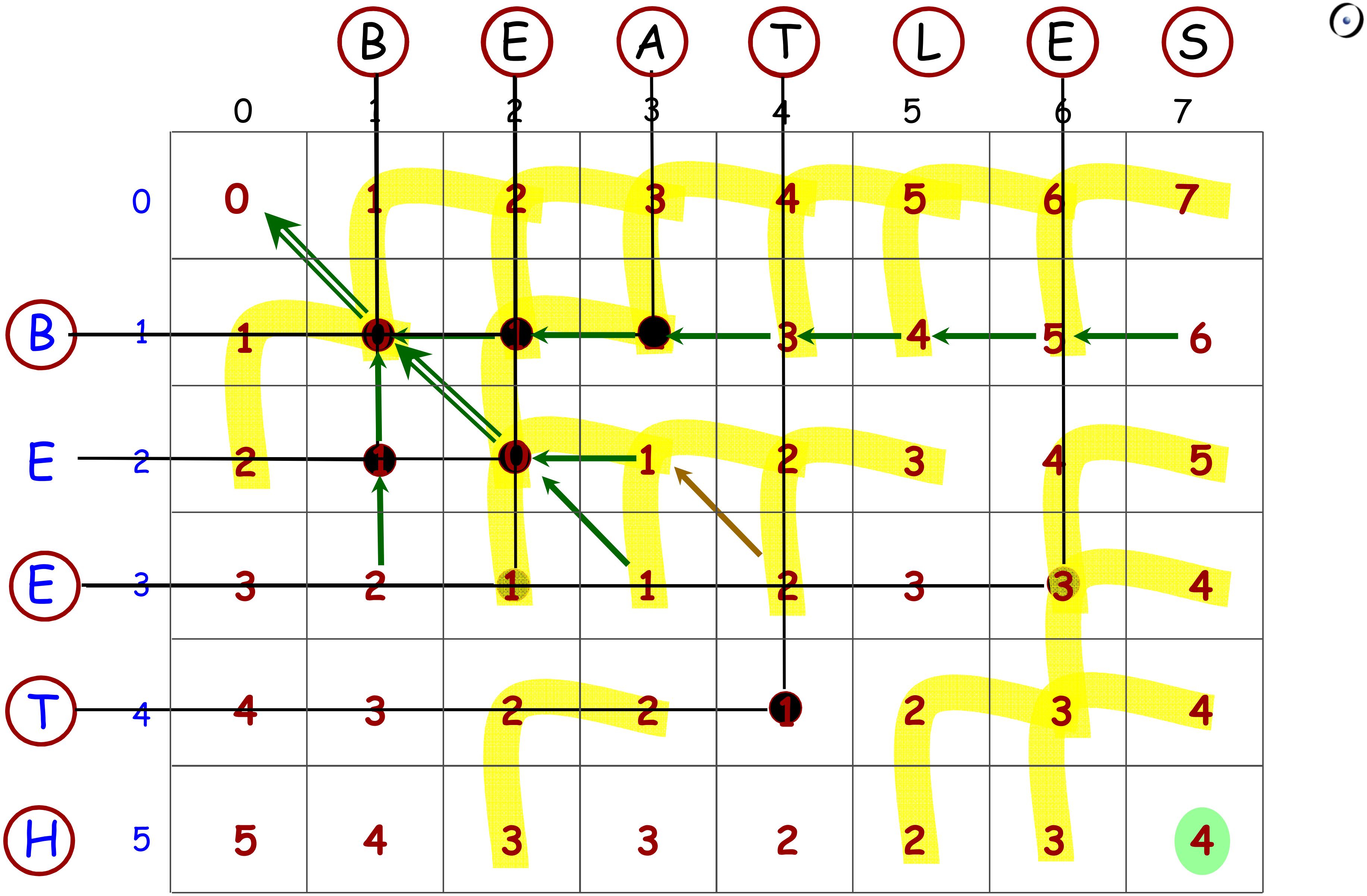
Lecture 2: Axiomatic semantics

Levenshtein distance

"Beethoven" to "Beatles"



Operation	–	–	R	–	D	R	D	–	R
Distance	0	0	1	1	2	3	4	4	5

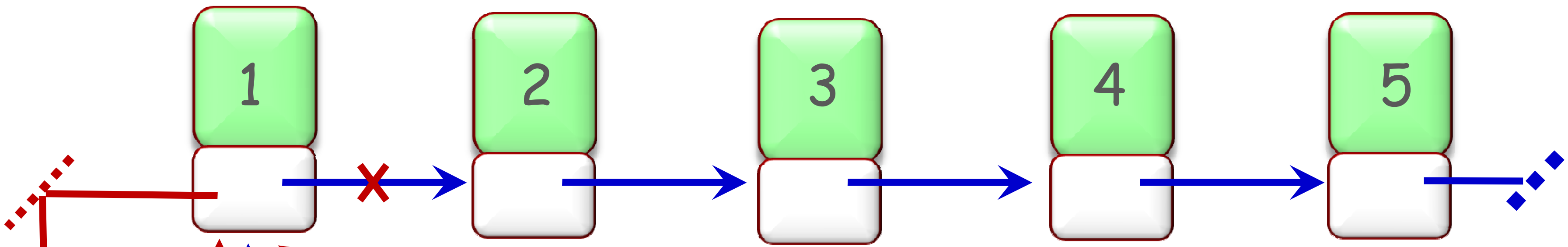

B E A T L E S
0 1 2 3 4 5 6 7
B E E T H
0 1 2 3 4 5
0 1 2 3 4 5 6 7
1 3 4 5 6
2 0 1 2 3 4 5
3 2 1 1 2 3 3 4
4 3 2 2 1 2 3 4
5 4 3 3 2 2 3 4

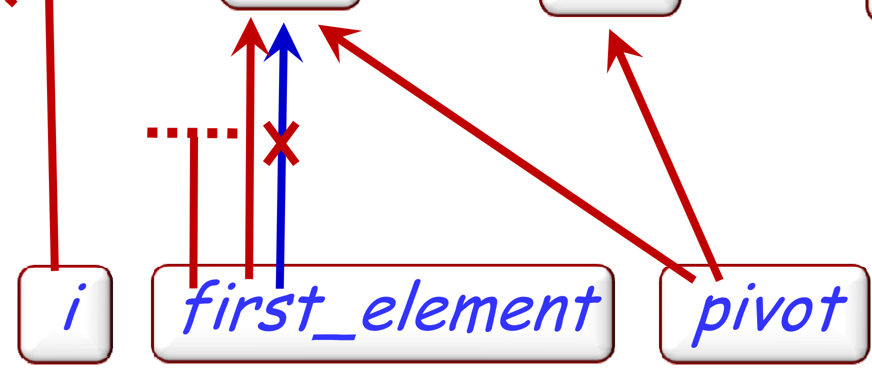
Levenshtein distance algorithm

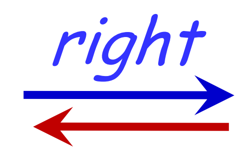
```
distance (source, target: STRING): INTEGER
        -- Minimum number of operations to turn source into target
    local
        dist: ARRAY_2 [INTEGER]
        i, j, del, ins, subst : INTEGER
    do
        create dist.make (source.count, target.count)
        from i := 0 until i > source.count loop
                dist [i, 0] := i ; i := i + 1
        end

        from j := 0 until j > target.count loop
                dist [0, j ] := j ; j := j + 1
        end
        -- (Continued)
```

```
  from i := 1 until i > source.count loop
    from j := 1 until j > target.count invariant
      ???
    loop
      if source [i ] = target [ j ] then
           dist [i, j ] := dist [ i -1, j -1]
      else
           deletion := dist [i -1, j ]
           insertion := dist [i , j - 1]
           substitution := dist [i - 1, j - 1]
           dist [i, j ] := minimum (deletion, insertion, substitution) + 1
      end
      j := j + 1
    end
    i := i + 1
  end
  Result := dist (source.count, target.count)
end
```

Reversing a list

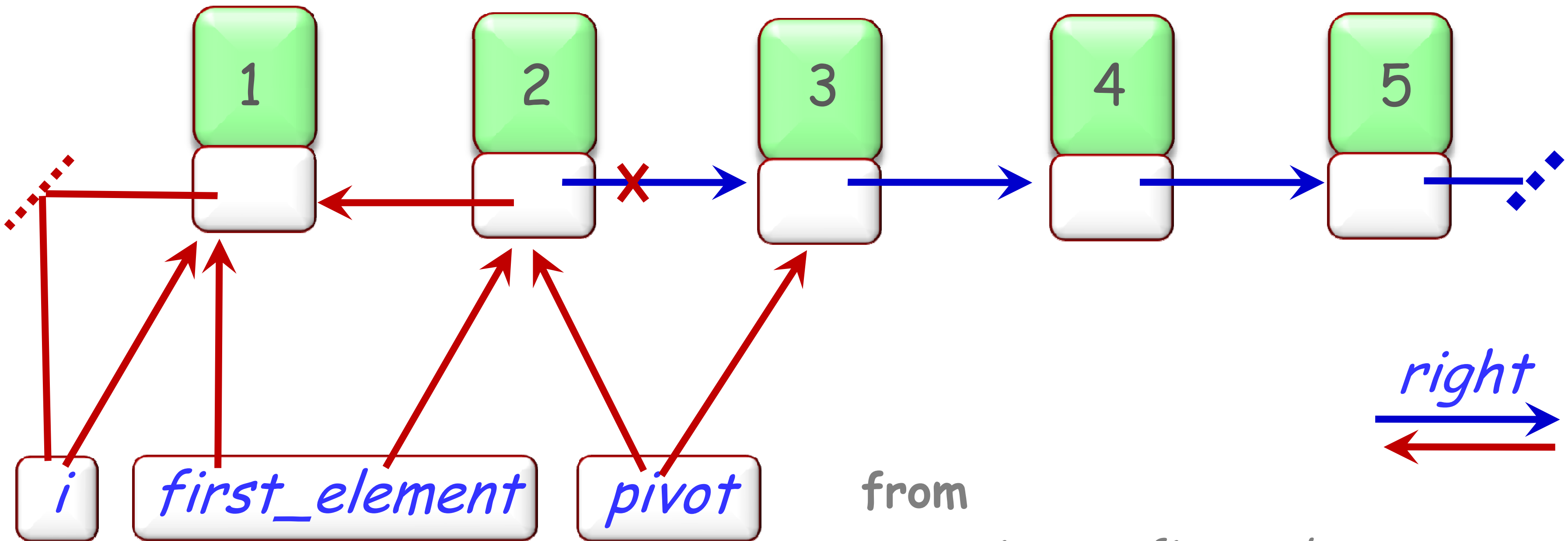






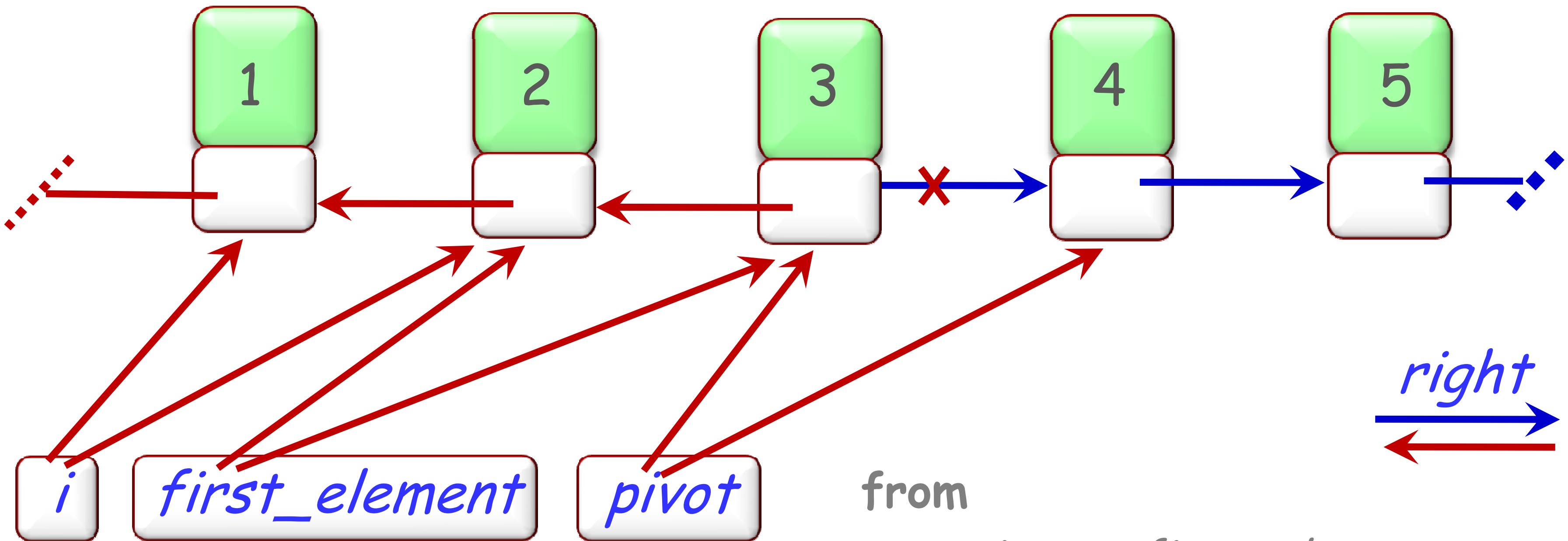

```
from
    pivot := first_element
    first_element := Void
until pivot = Void loop
    i := first_element
    first_element := pivot
    pivot := pivot.right
    first_element.put_right (i)
end
```

Reversing a list



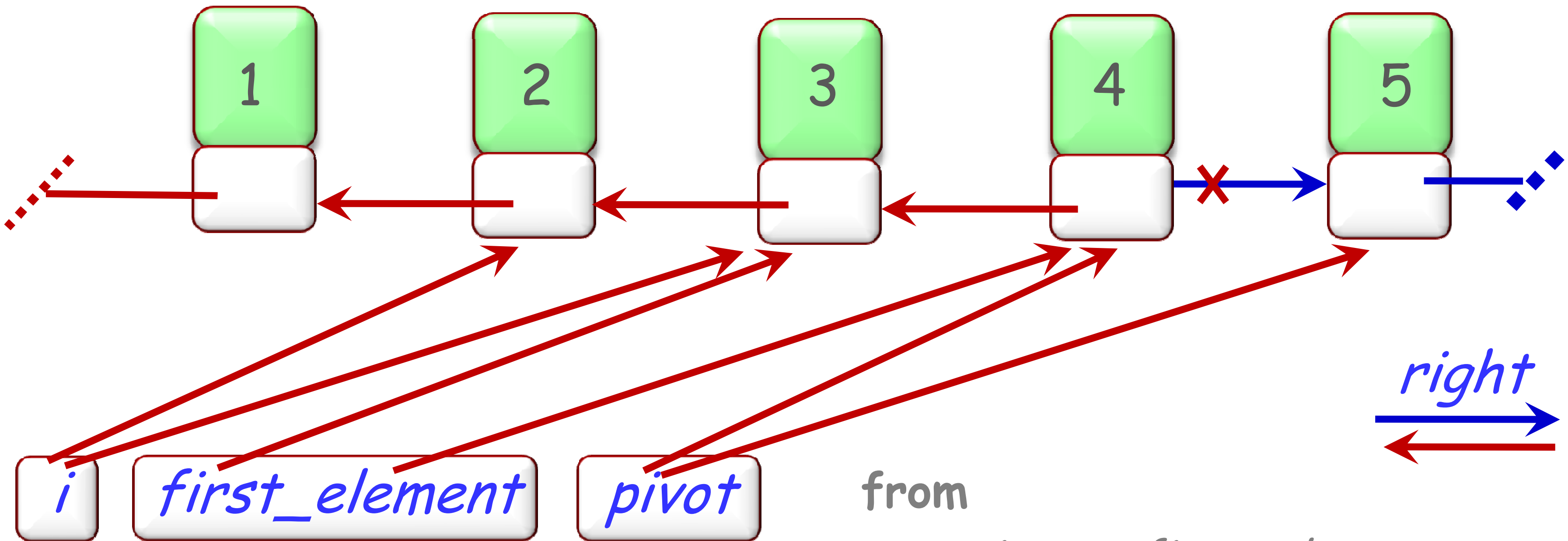

```
from
    pivot := first_element
    first_element := Void
until pivot = Void loop
    i := first_element
    first_element := pivot
    pivot := pivot.right
    first_element.put_right (i)
end
```

Reversing a list



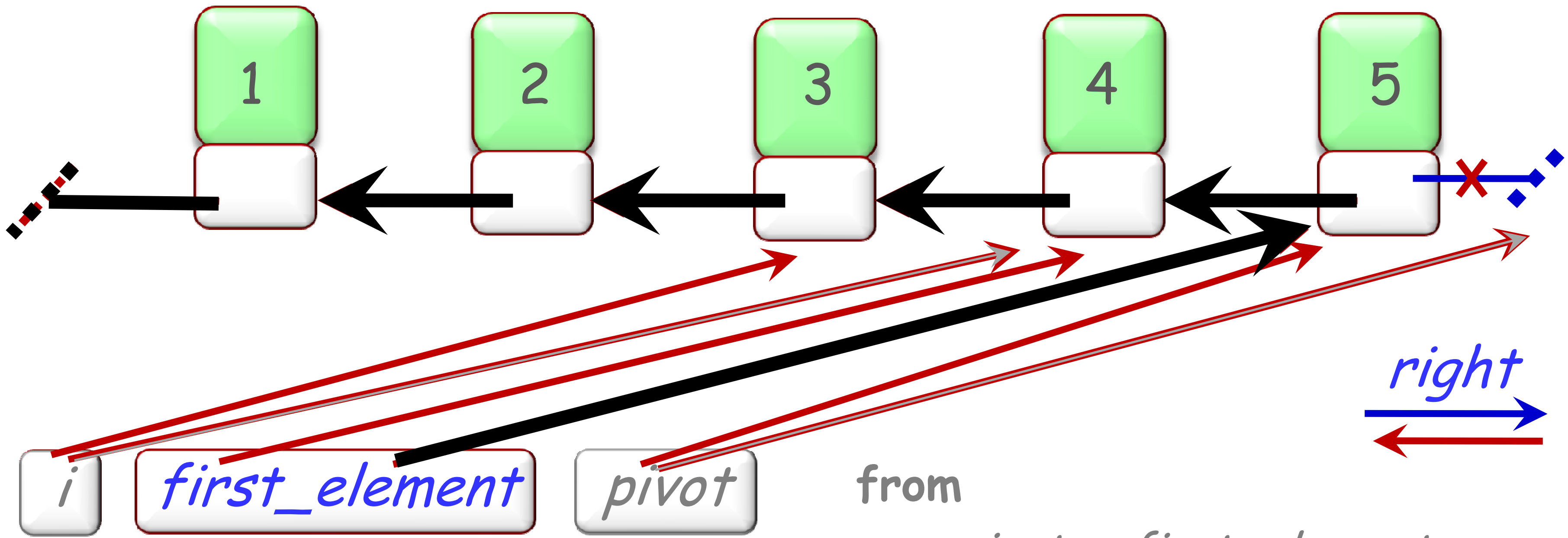

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



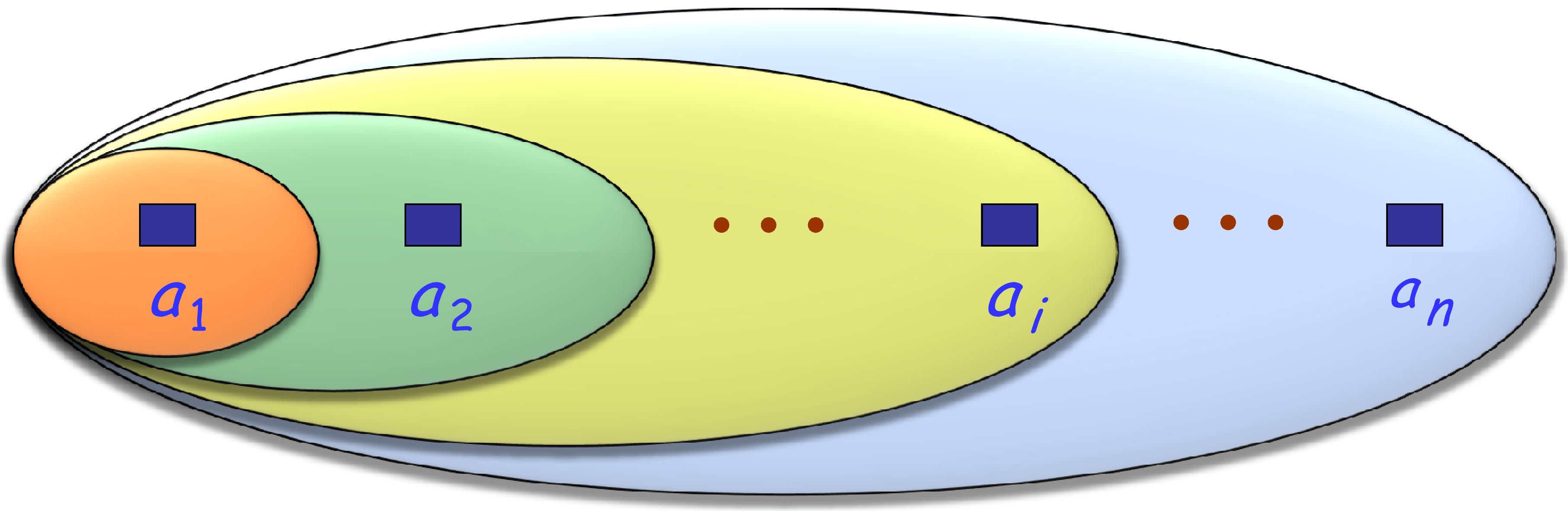

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list




```
from
    pivot := first_element
    first_element := Void
until pivot = Void loop
    i := first_element
    first_element := pivot
    pivot := pivot.right
    first_element.put_right (i)
end
```

Loop as approximation strategy



Result = a_1 = Max (a_1 .. a_1)

Result = Max (a_2 .. a_2)

Result = Max (a_1 .. a_i)

The loop invariant

Result = Max (a_1 .. a_n)

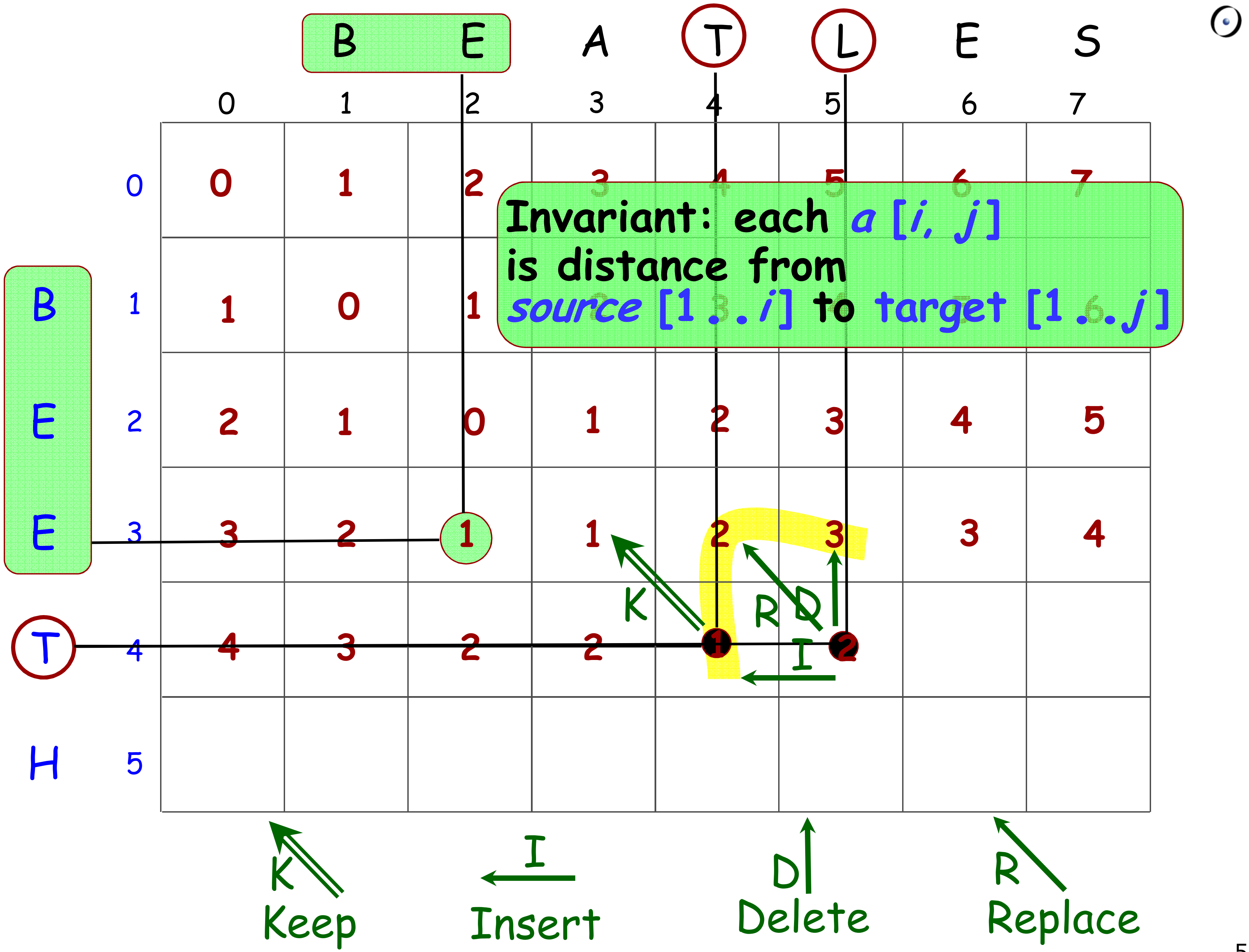
Loop body:

```
i := i + 1
Result := max (Result , a [i ])
```

Loops as problem-solving strategy

A loop invariant is a property that:

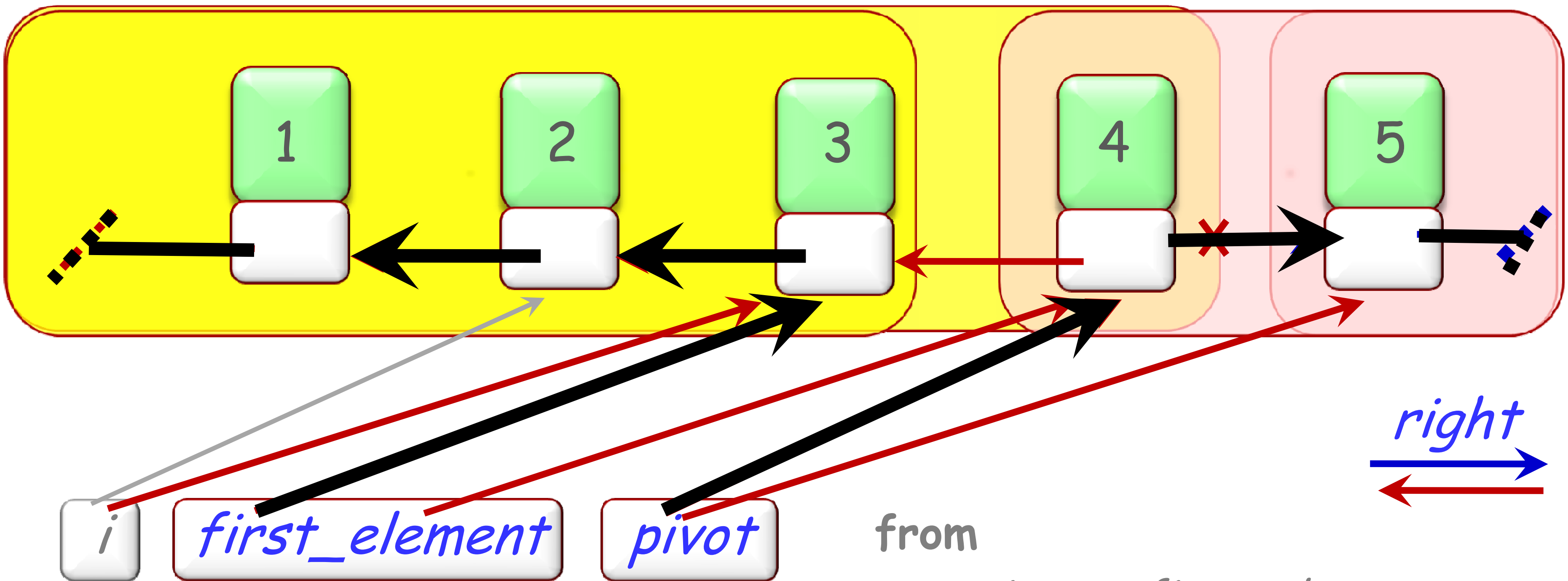
- Is easy to **establish initially**
  (even to cover a trivial part of the data)
- Is easy to **extend** to cover a bigger part
- If covering all data, gives the **desired result**!


B E A T L E S
0 1 2 3 4 5 6 7
B E E T H
Invariant: each a [i, j]
is distance from
source [1 .. i] to target [1 .. j]
K R D I
K Keep
I Insert
D Delete
R Replace

Levenshtein loop

```
from i := 1 until i > source.count loop
    from j := 1 until j > target.count invariant
        -- For all p : 1 .. i, q : 1 .. j –1, we can turn source [1 .. p ]
        -- into target [1 .. q ] in dist [p, q ] operations
    loop
        if source [i ] = target [ j ] then
            new := dist [ i -1, j -1]
    else
            deletion := dist [i -1, j ]
            insertion := dist [i , j - 1]
            substitution := dist [i - 1, j - 1]
            new := deletion.min (insertion.min (substitution)) + 1
        end
        dist [i, j ] := new
        j := j + 1
    end
    i := i + 1
end
Result := dist (source.count, target.count)
```

Reversing a list




```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Invariant: from *first_element* following right, initial items in inverse order; from *pivot*, rest of items in original order

Routines (1)

For:

f (x: T) **do** Body **end**

$$\frac{\{P\}\ Body\ \{Q\}}{\{P\ [a/x]\}\ \ f\ (a)\ \ \{Q\ [a/x]\}}$$

Routines (2)

For:

f (x: T) **do** Body **end**

$$\frac{(\forall a \mid \{P[a/x]\}\ f(a)\ \{Q[a/x]\})\ \textbf{implies}\ \{P\}\ Body\ \{Q\}}{\{P[a/x]\}\ f(a)\ \{Q[a/x]\}}$$

Hoare (1971)

The solution to the infinite regress is simple and dramatic: to permit the use of the desired conclusion as a hypothesis in the proof of the body itself. Thus we are permitted to prove that the procedure body possesses a property, on the assumption that every recursive call possesses that property, and then to assert categorically that every call, recursive or otherwise, has that property. This assumption of what we want to prove before embarking on the proof explains well the aura of magic which attends a programmer's first introduction to recursive programming.

Procedures and Parameters: An Axiomatic Approach, in E. Engeler (ed.), *Symposium on Semantics of Algorithmic Languages*, Lecture Notes in Mathematics 188, pp. 102-16 (1971).

Functions

The preceding rule applies to procedures (routines with no results)

Extension to functions?