The notion of algorithm

General definition:

An algorithm is the specification of a process to be carried out by a computer
Not quite an algorithm

---

5 properties of an algorithm

- Defines data to which process will be applied
- Every elementary step taken from a set of well-specified actions
- Describes ordering(s) of execution of these steps
- 2 and 3 based on precisely defined conventions, suitable for execution by an automatic computer
- For any data, guaranteed to terminate after finite number of steps

---

Algorithm vs program

- "Algorithm" usually considered a more abstract notion, independent of platform, programming language etc.
- In practice, the distinction tends to fade:
  - Algorithms need a precise notation
  - Programming languages becoming more abstract
- However:
  - In programs, data (objects) are just as important as algorithms
  - A program typically contains many algorithms and object structures
What makes up an algorithm

Basic steps:
- Feature call $x.f(a)$
- Assignment
- ...

Sequencing of these basic steps:

**CONTROL STRUCTURES**

Control structures

Definition: program construct that describes the scheduling of basic actions

Three fundamental control structures:
- Sequence
- Loop
- Conditional

They are the "Control structures of Structured Programming"

Control structures as problem-solving techniques

**Sequence**: "To achieve C from A, first achieve an intermediate goal B from A, then achieve C from B"

**Loop**: solve the problem on successive approximations of its input set

**Conditional**: solve the problem separately on two or more subsets of its input set
The sequence (or Compound)

\[ \text{instruction}_1, \text{instruction}_2, \ldots, \text{instruction}_n \]

Semicolon as optional separator

\[ \text{instruction}_1; \text{instruction}_2; \ldots; \text{instruction}_n \]

Not quite an algorithm

![Image of a recipe with times and temperatures]
Correctness of a **Compound**

- **Precondition of** instruction\(_i\) **must hold initially**
- **Postcondition of each** instruction\(_i\) **must imply precondition of each** instruction\(_{i+1}\)
- **Final effect is postcondition of** instruction\(_n\)

---

**Conditional instruction**

```
if
  Condition -- Boolean_expression
then
  Instructions -- Compound
else
  Other_instructions -- Compound
end
```

---

**Computing the greater of two numbers**

```
if
  a > b
then
  max := a
else
  max := b
end
```
As a function

```plaintext
maximum(a, b: INTEGER): INTEGER
-- The higher of a and b
do
    if a > b
        then
            Result := a
        else
            Result := b
    end
end
```

The conditional as problem-solving technique

- **PROBLEM SPACE**
- **Region 2**
- **Region 1**
- **Use technique 1**
- **Use technique 2**

Basic form

```plaintext
if Condition then
    Instructions
else
    Other_instructions
end
```
A variant of the conditional

```plaintext
if Condition then
    Instructions
end
```

(Means the same as

```plaintext
if Condition then
    Instructions
else
    end
```

with an empty "else" clause)

Nesting

```plaintext
if Condition, then
    Instructions_1
else
    if Condition, then
        Instructions_2
else
    if Condition, then
        Instructions_3
else
    ... 
end
end
```
Nested structure

Comb-like structure

Comb-like conditional

```plaintext
if Condition, then
    Instructions_1
elseif Condition_2 then
    Instructions_2
elseif Condition_3 then
    Instructions_3
elseif ...
else
    Instructions_0
end
```
On the menu for today

- Loops and their invariants
- See what it takes to ensure that a loop terminates
- Look at the general problem of loop termination
- Examine lower-level control structures: "Goto" and flowcharts; see rationale for the "control structures of Structured Programming"
- Prove the undecidability of the Halting Problem
### Loop, full form

```
from  Initialization    -- Compound
      invariant
      Invariant_expression   -- Boolean_expression
      variant
      Variant_expression     -- Integer_expression
until Exit_condition    -- Boolean_expression
loop
      Body                  -- Compound
end
```

### Another loop syntax

```
```

### Loop, full form

```
from  Initialization    -- Compound
      invariant
      Invariant_expression   -- Boolean_expression
      variant
      Variant_expression     -- Integer_expression
until Exit_condition    -- Boolean_expression
loop
      Body                  -- Compound
end
```
Looping over stations of a line

```
from fancy.start until fancy.after loop
  -- "Do something with fancy.item"
  fancy.forth
end
```

Previously and in the textbook:
```
fancy_line
```

Operations on a list

```
before item after
1 start index forth back count
(The cursor)
```

Commands
Queries
(boolean)
Looping over stations of a line

from fancy.start until fancy.after loop
   -- "Do something with fancy.item"
   fancy.forth
end

Displaying station names

from fancy.start until fancy.after loop
   -- Display name of next station:
   Console.show(fancy.item)
   fancy.forth
end

Computing the "maximum" of station names

from fancy.start ; Result := ""
until fancy.after loop
   Result := greater(Result, fancy.item.name)
   fancy.forth
end
Assignment

Result := "XYZ"

-- Change the value of Result to "XYZ"

Computing the “maximum” of station names

from
  fancy.start; Result := ""
until
  fancy.after
loop
  Result := greater (Result, fancy.item.name)
  fancy.forth
end

The greater of two strings, alphabetically, e.g.
greater ("ABC", "AD") = "AD"

In a function

highest_name: STRING is
  -- Alphabetically greatest station name of line
  do
    from
      fancy.start; Result := ""
    until
      fancy.after
    loop
      Result := greater (Result, fancy.item.name)
      fancy.forth
    end
  end
Postcondition?

highest_name: STRING is
-- Alphabetically greatest station name of line
do from fancy.start; Result := ""
until fancy.after
loop
  Result := greater (Result, fancy.item.name)
  fancy.forth
end
ensure Result /= Void and then not Result.empty
end

Loop as approximation strategy

Result = name_1
Result = Max (name_1, name_2)
Result = Max (name_1, name_2, ..., name_i)
Result = Max (name_1, name_2, ..., name_i, ..., name_n)

Computing the "maximum" of station names

from
  fancy.start; Result := ""
until fancy.after
loop
  Result := greater (Result, fancy.item.name)
  fancy.forth
end
The loop invariant

from
fancy.start : Result := ""
invariant
fancy.index <= 1
fancy.index := fancy.count + 1
-- Result is the alphabetically highest of the
-- names of previous stations
until
fancy.after
loop
Result := greater (Result, fancy.item.name)
---
fancy.forth
end

Loop invariant

(Do not confuse with class invariant)

Property that is:

- Satisfied after initialization (from clause)
- Preserved by every loop iteration (loop clause) when executed with the exit condition (until clause) not satisfied

The loop invariant

from
fancy.start : Result := ""
invariant
fancy.index <= 1
fancy.index := fancy.count + 1
-- Result is the alphabetically highest of the
-- names of previous stations
until
fancy.after
loop
Result := greater (Result, fancy.item.name)
---
fancy.forth
end
The loop invariant (better)

from fancy.start; Result := ""

invariant
  index := 1
  index <= count + 1
  -- If there are any previous stations,
  -- Result is the alphabetically highest of their names
until fancy.after
  loop
    Result := greater (Result, fancy.item.name)
  fancy.forth
end

Loop as approximation strategy

Result = name1

Result = Max (name1, name2)

Result = Max (name1, name2, ..., namei)

Result = Max (name1, name2, ..., namei, ..., name1)

In a function

highest_name: STRING is
  -- Alphabetically greatest station name of line
  do
    from fancy.start; Result := ""
    until fancy.after
    loop
      Result := greater (Result, fancy.item.name)
    fancy.forth
  end
end
Postcondition?

```plaintext
highest_name: STRING is
  -- Alphabetically greatest station name of line
do
  from fancy.start ; Result := ""
  until fancy.after
  loop
    Result := greater (Result, fancy.item.name)
    fancy.forth
  end
ensure
  Result /= Void and then not Result.empty
end
```

Loop as approximation strategy

```
Result = name_1,
Result = Max (name_1, name_2)
Result = Max (name_1, name_2, ..., name_i)
Result = Max (name_1, name_2, ..., name_i, ..., name_n)
```

The loop invariant

```
from fancy.start ; Result := ""

invariant
  fancy.index >= 1
  fancy.index := fancy.count + 1
  -- Result is the alphabetically highest of the
  -- names of previous stations
until fancy.after
loop
  Result := greater (Result, fancy.item.name)
  fancy.forth
end
```
Loop invariant

(Do not confuse with class invariant)

Property that is:

- Satisfied after initialization (from clause)
- Preserved by every loop iteration (loop clause) when executed with the exit condition (until clause) not satisfied

The loop invariant

```
from fancy.start ; Result := ""

invariant
fancy.index := 1
fancy.index <= fancy.count + 1
-- Result is highest of previous station names

until fancy.after
loop
Result := greater (Result, fancy.item.name)

fancy.forth
end
```

The loop invariant (better)

```
from fancy.start ; Result := ""

invariant
index := 1
index <= count + 1
-- If there are any previous stations, Result is the alphabetically highest of their names

until fancy.after
loop
Result := greater (Result, fancy.item.name)

fancy.forth
end
```
The effect of the loop

from fancy.start; Result := ""
invariant
index := 1
index := count + 1
loopResult := greatest (Result, fancy.item.name)

fancy.forth
end

Invariant satisfied after initialization

Invariant satisfied after each iteration

Exit condition satisfied at end

At end: Invariant and exit condition
• All stations visited (fancy.after)
• Result is highest of their names

Quiz: what’s the invariant?

xxx(a, b: INTEGER, INTEGER) is
require
a > 0; b > 0
local m, n : INTEGER
do from
invariant

variant

until m = n
loop
if m > n then
m := m − n
else
n := n − m
end
end
Result := m
end
Quiz: what’s the invariant?

```plaintext
euclid(a, b: INTEGER): INTEGER is
  -- Greatest common divisor of a and b
  require
  a > 0 ; b > 0
  local m, n: INTEGER do
    from
    invariant m = a, n = b
    variant ??????
    until m = n loop
      if m > n then
        m := m − n
      else
        n := n − m
      end
    end
  end
  Result := m
end
```

What we have seen

- The notion of algorithm
  - Basic properties
  - Difference with "program"
- The notion of control structure
- Correctness of an instruction
- Control structure: sequence
- Control structure: conditional
- Nesting, and how to avoid it
End of lecture 8