Levenshtein distance

"Michael Jackson" to "Mendelssohn"

```
  MICHAEL
  JACKSON
    E N D S
```

Operation: S D S S S D D D D I
Distance: 0 1 2 3 4 5 6 7 8 9 10

![Map of Switzerland]
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

\[
\text{from} \quad \text{Initialization} \quad \text{-- Compound}
\]

\[
\text{until} \quad \text{Exit\_condition} \quad \text{-- Boolean\_expression}
\]

\[
\text{loop} \quad \text{Body} \quad \text{-- Compound}
\]

\[
\text{end}
\]

Loop, full form

\[
\text{from} \quad \text{Initialization} \quad \text{-- Compound}
\]

\[
\text{invariant} \quad \text{Invariant\_expression} \quad \text{-- Boolean\_expression}
\]

\[
\text{variant} \quad \text{Variant\_expression} \quad \text{-- Integer\_expression}
\]

\[
\text{until} \quad \text{Exit\_condition} \quad \text{-- Boolean\_expression}
\]

\[
\text{loop} \quad \text{Body} \quad \text{-- Compound}
\]

\[
\text{end}
\]
Another loop syntax

Loop, full form

Looping over stations of a line

Previously and in the textbook: fancy_line
Operations on a list

before
1
start
index
forth
(The cursor)

item

after

count

Commands

Queries

(boolean)

Loops 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)
end
fancy, forth
```

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)
  end
  fancy, forth
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)
  end
  fancy, forth
 endorsements
  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 = 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)

In a function

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

Postcondition?

highest_name: STRING is
  -- Alphabetically greatest station name of line
  do
    from
    until
    loop
      Result := greater (Result, fancy.item.name)
    fancy.forth
  ensure
    Result /= Void and then not Result.empty
  end
**Loop as approximation strategy**

Result = \( \text{name}_1 \)

Result = \( \text{Max}(\text{name}_1, \text{name}_2) \)

Result = \( \text{Max}(\text{name}_1, \text{name}_2, ..., \text{name}_i) \)

Result = \( \text{Max}(\text{name}_1, \text{name}_2, ..., \text{name}_i, ..., \text{name}_n) \)

---

**The loop invariant**

```plaintext
from
fancy.start: Result := ***

invariant
fancy.index := 1
fancy.index <= fancy.count + 1
\( \rightarrow \) Result is the alphabetically highest of the
\( \rightarrow \) 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
  -- Result is highest of previous station names
until fancy.after
loop
  Result := greater(Result, fancy.item.name)
  Result is highest of their names
fancy.forth
end

Invariant satisfied after initialization

Invariant satisfied after each iteration

At end: invariant and exit condition
  • All stations visited (fancy.after)
  • Result is highest of their names
Quiz: what’s the invariant?

```c
xxx(a, b: INTEGER, n: INTEGER) {
  require: a >= b > 0
  local: m, n: INTEGER
  do from
    invariant: m >= a; n <= b
    variant: ???????
    until
      loop
        m := m + n
        if m = n then
          m := m - n
        else
          n := m - n
      end
    end
  end
  Result := m
}
```

Intermezzo: Levenshtein distance

Also called “Edit distance”

Purpose: to compute the smallest set of basic operations

- Insertion
- Deletion
- Replacement

that will turn one string into another

Levenshtein distance

"Michael Jackson" to "Mendelssohn"

```
M I C H A E L  J A C K S O N
  E  N  D  S
```

Operation: SDS S SDS D D D D D D D D D D I

Distance: 0 1 2 3 4 5 6 7 8 9 10
Levenshtein distance algorithm

```plaintext
distance(source, target: STRING): INTEGER
    -- Minimum number of operations to turn source into target
local
dist: ARRAY_2[INTEGER]
i, j, new, deletion, insertion, substitution: INTEGER

begin
    create dist, make (source, count, target, count)
from i := 0 until i < source, count loop
    dist[i, 0] := i; j := j + 1
end

from j := 0 until j < target, count loop
    dist[0, j] := j; j := j + 1
end

-- (Continued)
```

Levenshtein , continued

```plaintext
from i := 1 until i < source, count loop
    from j := 1 until j < target, count loop
        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
        end
    j := j + 1
end
i := i + 1
Result := dist(source, count, target, count)
```

Loop semantics rule

The effect of a loop is the combination of:

- Its invariant
- Its exit condition
How do we know a loop terminates?

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 variant

Integer expression that must:

Be non-negative when after initialization (from)

Decrease (i.e. by at least one), while remaining non-
negative, for every iteration of the body (loop) executed
with exit condition not satisfied

The variant for our loop

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

Can EiffelStudio find out if your program will terminate?

No, it can’t 😞

No other program, for any other realistic programming language, can! 😞 😞 😞

The halting problem and undecidability

(“Entscheidungsproblem”, Alan Turing, 1936.)

It is not possible to devise an effective procedure that will find out if an arbitrary program will terminate on arbitrary input

(or, for that matter, if an arbitrary program with no input will terminate)

The halting problem in Eiffel

Assume we have a routine

```plaintext
terminates (my_program: STRING): BOOLEAN
   -- Does my_program terminate?
   do
      ... Your algorithm here ...
   end
```

The halting problem in Eiffel

Then we can write

```plaintext
file_terminates(file_name: STRING): BOOLEAN
  -- Does program in file file_name terminate?
  do
      ... Your algorithm here ...
  end
```

The halting problem in practice

Some programs do not terminate in certain cases...

That's a bug!

Yours had better terminate in all cases

Use variants

Control structures at the machine level

Unconditional branch:

```
BR label
```

Conditional branch, for example:

```
BEQ loc_a loc_b label
```
The equivalent of if-then-else

```
if a = b then Compound_1 else Compound_2 end
```

BEQ loc_a, loc_b, 111

101 ... Code for Compound_2 ...

BR 125

111 ... Code for Compound_1 ...

125 ... Code for continuation of program ...

Flowcharts

```
True
\n\n\n\nFalse
```

```
\n\n\n\n\n```

In programming languages: the Goto

```
test condition goto else_part
   Compound_1
   goto continue
else_part:    Compound_2
continue:    ... Continuation of program ...
```

```
\n\n\n\n```

```
\n\n\n\n```

```
```
"Goto considered harmful"

Dijkstra, 1968
Arbitrary goto instructions lead to messy, hard to maintain programs ("spaghetti code")

The Goto today

Almost universally decried
Still exists in some programming languages
Also hides under various disguises, e.g. break

```plaintext
loop
  ...
  if c then break end
  ...
end
```

One-entry, one-exit

- (Compound)
- (Loop)
- (Conditional)
Quiz: what’s the invariant?

```plaintext
xxx(a, b: INTEGER, INTEGER) is
   -- ?????????????????????????????
   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 := m - n
   end
   end
   Result := m
end
```

Quiz: what’s the invariant?

```plaintext
gcd(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 := m - n
   end
   end
   Result := m
end
```

Intermezzo: Levenshtein distance

Also called “Edit distance”

Purpose: to compute the smallest set of basic operations

- Insertion
- Deletion
- Replacement

that will turn one string into another
Levenshtein distance

"Michael Jackson" to "Mendelssohn"

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>I</td>
<td>H</td>
</tr>
</tbody>
</table>

Distance 0 1 2 3 4 5 6 7 8 9 10

Levenshtein distance algorithm

```plaintext
distance (source, target: STRING): INTEGER
    -- Minimum number of operations to turn source into target
local
dist: ARRAY_2[INTEGER]
i, j, new, deletion, insertion, substitution: 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)
```

Levenshtein, continued

```plaintext
from i := 1 until i < source, count loop
    from j := 1 until j < target, count loop invariant
    -- For all p, q, 1 <= p, q <= i, 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
        i := i + 1
        j := j + 1
    end
Result := dist(source, count, target, count)
```

```
Levenshtein distance algorithm

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

Levenshtein, continued

```plaintext
from i := 1 until i > source, count loop
from j := 1 until j > target, count invariant
    -- For all p \geq 1, q \geq 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
    i := i + 1
end
j := j + 1
end
Result := dist (source, count, target, count)
```

What we have seen
What we have seen

- Basic loop concepts
- Lists as machines
- Loop correctness: invariant, variant
- Termination
- Concept of undecidability
- The halting problem and its undecidability
- The Levenshtein distance (first intro)
- The Goto, and why we don’t use it
- Flowcharts
- Machine-level branch instructions

End of lecture 10