


Eötvös Loránd University
Faculty of Informatics

PROGRAMMING

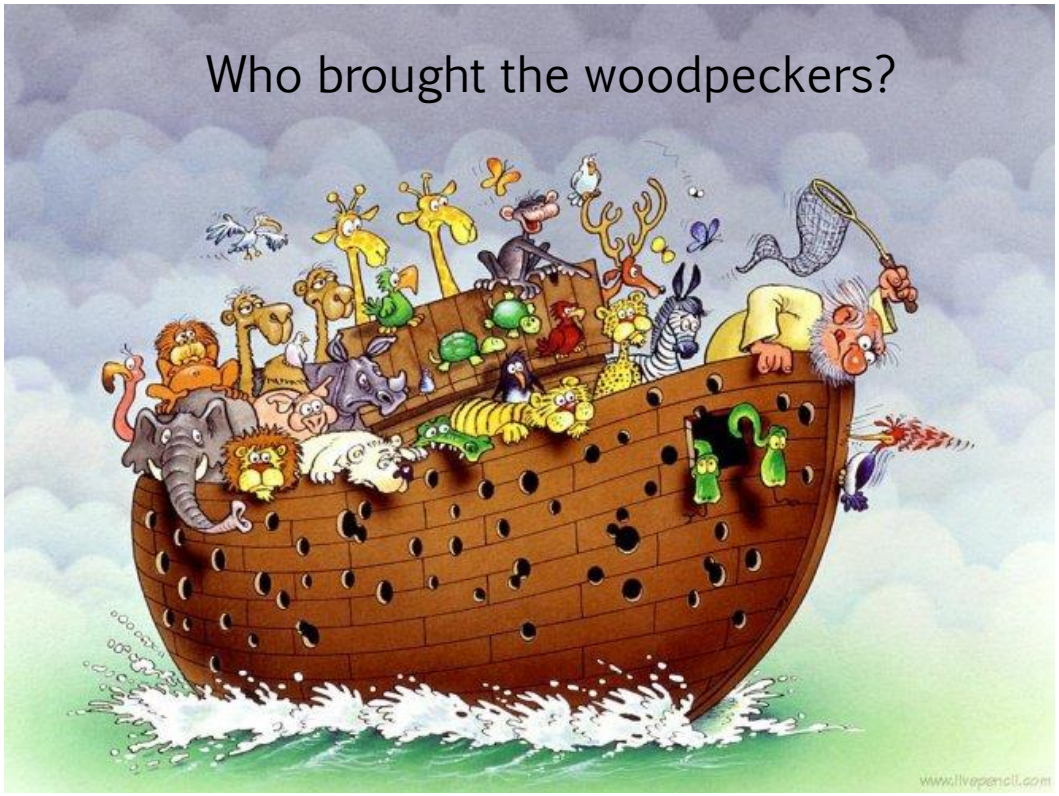
2019/20 1st semester

Lecture 7

ZS. PLUHÁR, H. TORMA

Today's lecture in pictures and sayings

„To err is human"



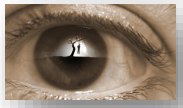
Today's lecture in pictures and sayings

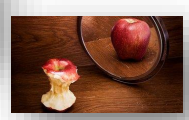
„Why do you look at the speck of sawdust in your brother's eye and pay no attention to the plank in your own eye?"

Content

- Testing
  - concepts + principles + methods
  - technique: run with data file – C++
- Searching for bugs
  - principles
  - tools
  - methods
- Correcting bugs
- Documentation

+ minimums of word processing





Testing

Aim: to reveal errors

Concepts:

- **Test case** = input + output
- **Test** = a set of test cases
- **Good test case**: it's likely that it reveals an undiscovered error
- **Ideal test:** reveals all errors
- **Reliable test**: it's likely that it reveals all errors

Instead of „error" it would be better to say „problem", as testing is not just about finding errors, but it is also for examining effectiveness.

Testing principles

- Testing for valid (accepted) and invalid (wrong) input
- One should maximally use the information from a given test case (when you choose the following test cases).
- Just others (than the author) can test the program well
- A great number of errors is in a small part of the program.
- A test case that cannot be repeated is wrong. (This is not an attribute of testing, but of the program that is tested.)


Testing methods

- Testing methods
  - Static testing: we examine the text of the program, without running the program
  - Dynamic testing: we run the program with different inputs, and we examine the results
- The result of testing:
  - we have found some bugs;
  - we haven't found any bugs – yet.
- The main question of testing?
  - How long should we test?


STATIC testing

Checking the code:

› algorithm ↔ code correspondence
  – to detect coding errors

› explaining algorithm+code to someone
  – to detect algorithmic+coding errors

Syntactic checking:

› in the case of a compiler, it's automatic
  – however, not always strict enough

› in the case of an interpreter, it means a lot of running – so actually it's actually dynamic

STATIC testing

- **Semantic checking**, searching for contradictions:
  - unused variable/value

```
i=1;
for (i=2;…) { … }
```

  - „suspicious” variable use

```
i=…; //it might come much sooner
      for (int i=2;…){ … i … }
… i …
```

  - „identical” transformation

```
i=1*i+0; //n1? K0? O?
```

STATIC testing

- **Semantic checking**, searching for contradictions (cont):
  - uninitialized variable

```
int n;    //data decription of specification
int k[n];//used in code without thinking
```

  - expression with undefined value(?)

```
int n; ← global n
…
int fv()
{
    for (int n=0;n<9;++n) ← local n
    {
        … n … ; ← using local
    }
    return …n…; ← the formula
                  containing the global n
}
```

STATIC testing

› **Semantic checking**, searching for contradictions (cont):
  – function without return value

syntactic error in C++ language:

```
int fn()
{
    ...
    return; ← ... error: return-statement with no value, in function returning 'int' |
}
```

... but this one causes only a warning:

```
int fn()
{
    ...
} ← ... warning: control reaches end of non-void function |
```

STATIC testing

- **Semantic checking**, searching for contradictions (cont):
  - equivalently true/false condition

```
( N>1 || N<100 ) / ( N<1 && N>maxN )
```

frequent error during coding of input checking

  - infinite count loop

```
for (int i=0;i<N;++i)
{
    ...
    --i; ← no problem in compile time
}
```

It might have remained there when rewritten from a while loop.

STATIC testing

- **Semantic checking**, searching for contradictions (cont):
  - Imprecise loop organisation

```
for (int i=0;i<N;++i)
{
    ...
    ++i; ← no problem at compilation}
```

It might have remained there when rewritten from a **while** loop.

  - An expression containing constant value variables

```
y=sin(x)*cos(x)-sin(2*x)/2
```

STATIC testing

- **Semantic checking**, searching for contradictions (cont):
  - Infinite conditional loop (in loop with `i<N` condition, neither `i`, nor `N` are changing or are changing synchronously)

```
i=1;
while (i<=N)
{
    ...
    i=+1; ← wanted to write i+=1?
}
```

STATIC testing

- **Semantic checking**, searching for contradictions (cont):
  - Infinite conditional loop (in loop with `i<N` condition, neither `i`, nor `N` are changing or are changing synchronously)

```
i=1;
while (i<=N)
{
    ...
    i=i++; ← wanted to write i++ or i=i+1?
}
```

DYNAMIC testing

Testing methods:

- **Black box methods** (←no complete input – cannot be tested for all possible inputs): test cases are chosen optimally based on the program specifcation (without peering into its internal structures or workings)
- **White box methods** (←no complete route – not possible to test all executable routes): test cases are chosen optimally based on the structure of the program
- **Gray box methods** – combination of Wb and Bb methods – with limited knowledge of the internals of the system.

DYNAMIC testing – black box methods

- **Equivalence class partitioning:**
  - input (or output) data is divided into disjunctive classes for which we expect the program to work the same way, then we choose one test case for each class
- **Boundary value analysis:**
  - choosing test cases from the boundaries of the equivalence classes (for input and output classes, too)
- **Cause-effect graph:**
  - the combination of input conditions (cause) and output conditions (effect) of equivalence classes (a directed graph that maps a set of causes to a set of effects)

Equivalence class partitioning - classification

- If the input condition defines a set of values (domain), then the valid equivalence class should be the set of acceptable input values, and the invalid classes should be the ranges below and above the boundaries.
  - Eg. if the data is about grades (values between 1 and 5), then we have these equivalence classes: $\{1 \leq i \leq 5\}$, $\{i < 1\}$ and $\{i > 5\}$.
- If the input condition defines the count of inputs, then we should classify similarly.
  - Eg. if we need to read in at most 6 characters, then the valid class is: reading in 0-6 characters, the invalid class is: reading in more than 6 characters (less than 0 can not occur).

Equivalence class partitioning - classification

- If the input condition says that the input data should have a certain attribute, then we need two have to equivalence classes: one valid and one invalid.
- If we assume that the program works with items of an equivalence class in different ways, then we should break this class into other classes.
- We should use the same principles for the output equivalence classes, as well.

Equivalence class partitioning – test cases

Test cases are defined by following these 2 principles:

- Until we have not covered all the **valid** equivalence classes, we should create test cases that cover the **most possible** classes.
- For each **invalid** class, we should write a test case. In the case of more errors, it could happen that the wrong data are overlapping, and because of an error, a second error does not appear.

Comment: each of them should invoke an error signal in the program

Boundary value analysis

- If the input condition defines a set of values (domain), then we should write test cases for the bottom and top boundaries of the valid range, and for a value close to the boundary from the invalid range.
  - Eg.: if the input range is the (0,1) open interval, then we should try the program with the following values: 0, 1, 0.01, 0.99
- If the input condition defines the cardinality of values, then we should work similarly.
  - Eg.: if we need to sort 1-128 names, then we should try the program with 0, 1, 128, and 129 names.

Cause-effect graph

Cause-effect analysis:

- the combination of input conditions (cause) and output conditions (effect) of equivalence classes – so an examination of the relationship between inputs and outputs
- Let's create a directed graph that maps a set of causes to a set of effects, with OR, AND and NOT relationships.
- The testing plan is crawling through the graph entirely.

DYNAMIC testing – black box methods

Task: Let's give a non-1 and non-N divisor of natural number N

› Equivalence classes (based on input):

valid data

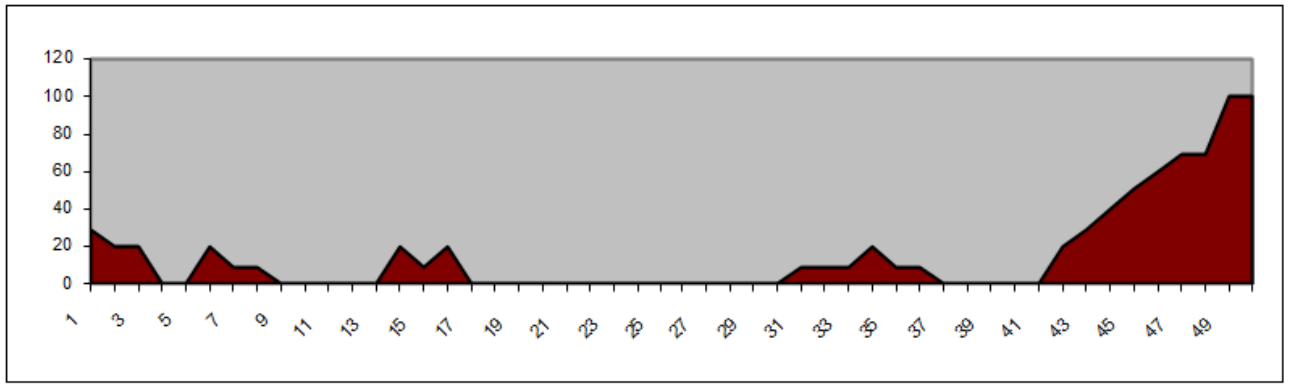
1. N is prime: 3
2. N has one real divisor: 25=5*5
3. N has more, different real divisors: 77=7*11
4. N is even: 2, 4=2*2, 6=2*3 $\subset 1. \cup 2. \cup 3.$

invalid data

5. N is anything that is not a natural number

DYNAMIC testing – black box methods

Task: We flew from Europe to North-America. During the flight, we measured the sea level below us (≥0) at some points. We got 0 where we flew above the sea, and >0 where we flew above land. Let's give the widest island!



DYNAMIC testing – black box methods

Task: We flew from Europe to North-America. During the flight, we measured the sea level below us (≥0) at some points. We got 0 where we flew above the sea, and >0 where we flew above land. Let's give the widest island!

Specification:

Input: $N \in \mathbb{N}$, $Hi_{1..N} \in \mathbb{N}^N$

Output: $Exists \in \mathbb{L}$, $B, E \in \mathbb{N}$

Precondition: $Hi_1 > 0$ and $Hi_N > 0$ and

$\forall i (1<i<N)$: $Hi_i \geq 0$ and $\exists i (1<i<N)$: $Hi_i = 0$

$[\rightarrow N \geq 3]$

Invalid equivalence classes:

- $N < 3$
- $Hi_1 \leq 0$
- $Hi_N \leq 0$
- $\exists i (1<i<N)$: $Hi_i < 0$
- $\forall i (1<i<N)$: $Hi_i > 0$

DYNAMIC testing – black box methods

Task: We flew from Europe to North-America. During the flight, we measured the sea level below us (≥0) at some points. We got 0 where we flew above the sea, and >0 where we flew above land. Let's give the widest island!

Valid equivalence classes (based on the output):

- No island
- Island exists
  - Only one island
  - More islands
    - Width is the same
    - Width is different
      - The first is the widest
      - The last is the widest
      - An inner island is the widest

DYNAMIC testing – black box methods

Task: We flew from Europe to North-America. During the flight, we measured the sea level below us (≥0) at some points. We got 0 where we flew above the sea, and >0 where we flew above land. Let's give the widest island!

Boundaries:

- Europe's width: 1, not 1;
- North America's width: 1, not 1;
- The widest island has width 1, not 1;
- The widest island is separated from its (left and right) neighbour with a sea whose width is: 1, not 1.

How many test cases are there? The first two means 4 cases. The third doubles this. The fourth makes it 4-times its value.

- **32 test cases!**

DYNAMIC testing – black box methods

Task: We flew from Europe to North-America. During the flight, we measured the sea level below us (≥0) at some points. We got 0 where we flew above the sea, and >0 where we flew above land. Let's give the widest island!

Do we find **all** the errors using this?

- Let's assume that we first find the last point of Europe and the first point of North America, and then we search for islands between these two points.
- If we fail to use the correct value for the first point of North America (for example the variable will be set to N/2 or N-10), then what guarantee do we have that previous test cases discover this?
- What if our program gave Europe or North America as the widest island?

DYNAMIC testing – gray box methods

Analysis by boundaries of sequences

› The first element will be processed
› The last element will be processed
› An inner element will be processed

Analysis by size of sequences

› Using empty sequence
› Using sequence with 1 element
› (Using sequence with 2 elements)
› Using sequence with more elements

DYNAMIC testing – gray box methods

Summation (sequence calculation)

› A sequence with 2 different elements

› Load testing, overloading analysis

Counting

› A sequence with 2 elements – both with the certain attribute

› A sequence with 0, 1, 2 or more elements with the certain attribute

Selection

› Select the first element

› Select not the first element

DYNAMIC testing – gray box methods

Searching

› The searched element is the first

› The searched element is the last

› There exists an inner element with the given attribute

› Doesn't exist inner element with the given attribute

Maximum selection

› A sequence with 2 elements – the first is bigger

› A sequence with 2 elements – the 2nd is bigger

› A sequence where an inner element is the biggest

› A sequence with more elements with maximum value

DYNAMIC testing – white box methods

Methods

- We choose a testing strategy based on the structure of the program,
- based on the strategy, we assign test predicates to the given paths,
- the test predicates define equivalence classes, and we choose one test case for each of them

DYNAMIC testing – white box methods

Strategies of trials:

- Statement coverage: This technique requires every possible **statement** in the code to be tested at least once during the testing process.
- Decision/branch coverage: This technique checks every possible path (if-else and other **conditional** loops) of a software application.
- Path coverage: tests all the paths of the program. This is a comprehensive technique which ensures that all the paths of the program are traversed at least once.

Generating test cases

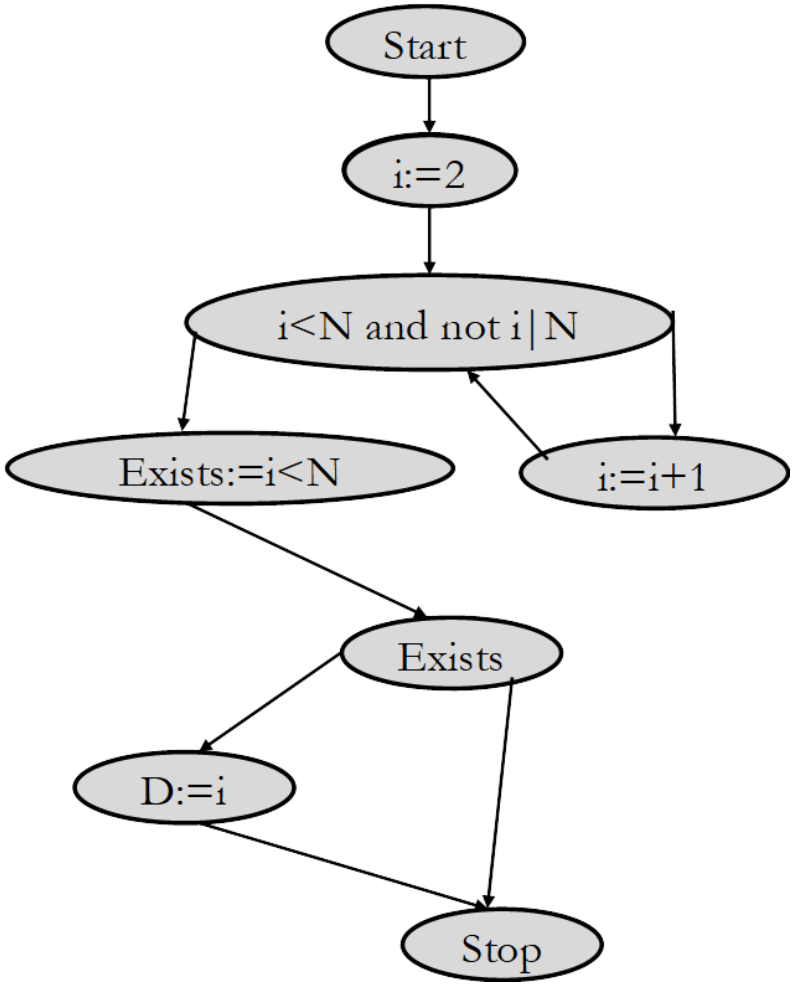
- A base path is a path of the program graph that:
  - Starts from the start point, and ends at the evaluation of the first branch condition or loop condition.
  - Starts from a branch or loop condition, and goes until the next condition or loop condition.
  - Starts from a branch or loop condition and lasts until the end of the program, if there are no other conditionals along the path.

DYNAMIC testing – white box methods

Program graph

Algorithm

i:=2
i<N **and not** i|N
i:=i+1
Exists:=i<N
Exists / T / F
D:=i | -



DYNAMIC testing – white box methods

- **Test paths** are the paths on the program graph that begin at the starting point of the graph, and stop at the end, and contain each edge only once.
- A **test predicate** means the conditions applied for the input which assure that using them the program will go through only one test path.

The first step in generating test cases is defining a minimum number of test paths that cover the program graph based on the chosen testing strategy.

DYNAMIC testing – white box methods

Generating the test predicate: This needs the symbolic execution of the program.

› Let's start from the precondition.

› Go until the first if condition or loop condition, and transform the formula based on the operations between.

› We join the corresponding condition of the test path with an **and** relationship to the test predicate.

› Then we should continue the symbolic execution to the end of the program.

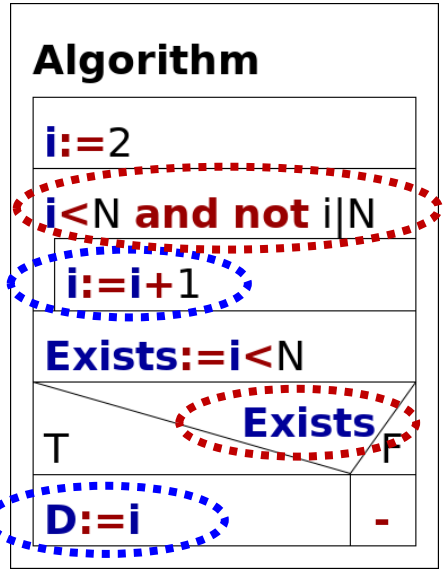
DYNAMIC testing – white box methods

Task: The non-1, non-N divisor of an N natural number

Statement coverage:

› `i:=i+1` to be executed: `N=3`

› `D:=i` to be executed: (←`Exists=True`) `N=4`



Decision coverage:

› Loop condition true: `N=3`

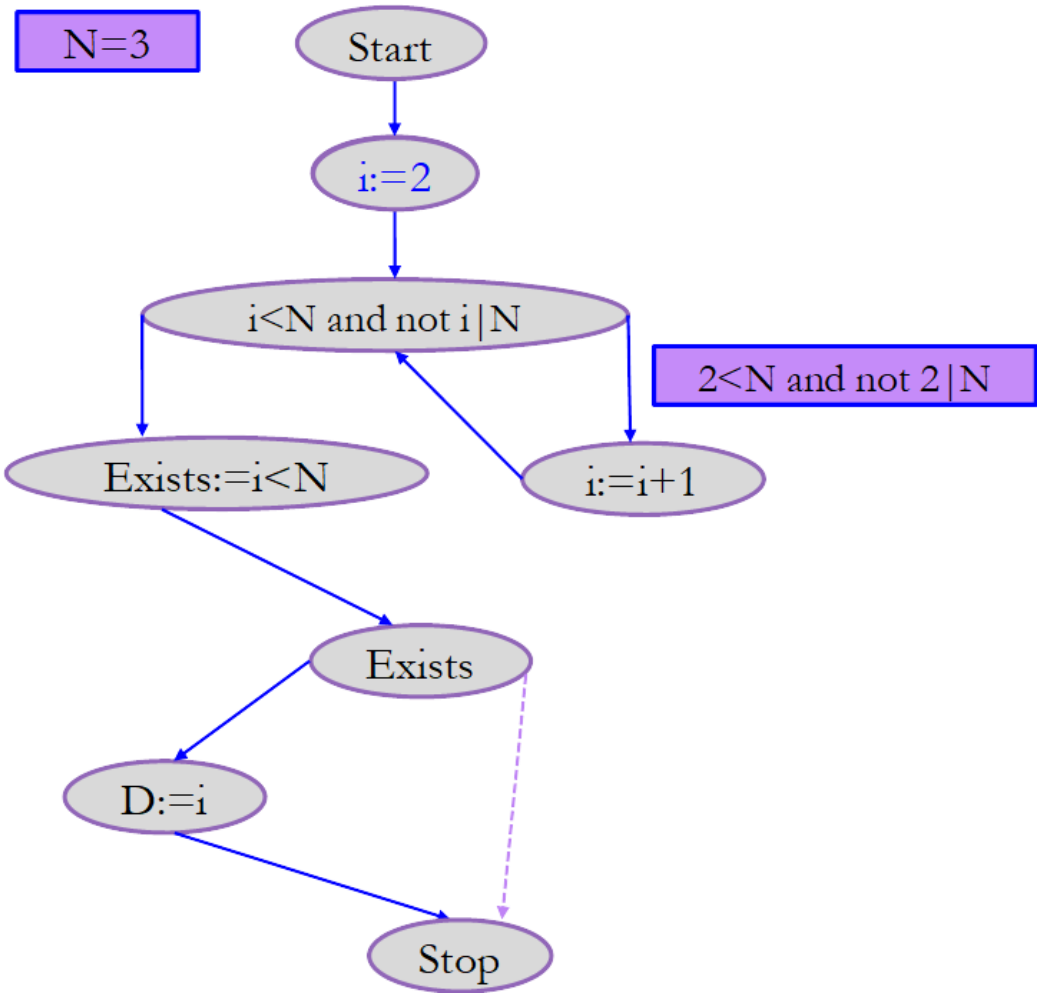
› Loop condition false: `N=2` (won't enter)

› Branch condition true: (↔`Exists=True`) `N=4`

› Branch condition false: (↔`Exists=False`) `N=2`

DYNAMIC testing – white box methods

Test path$_1$



Algorithm

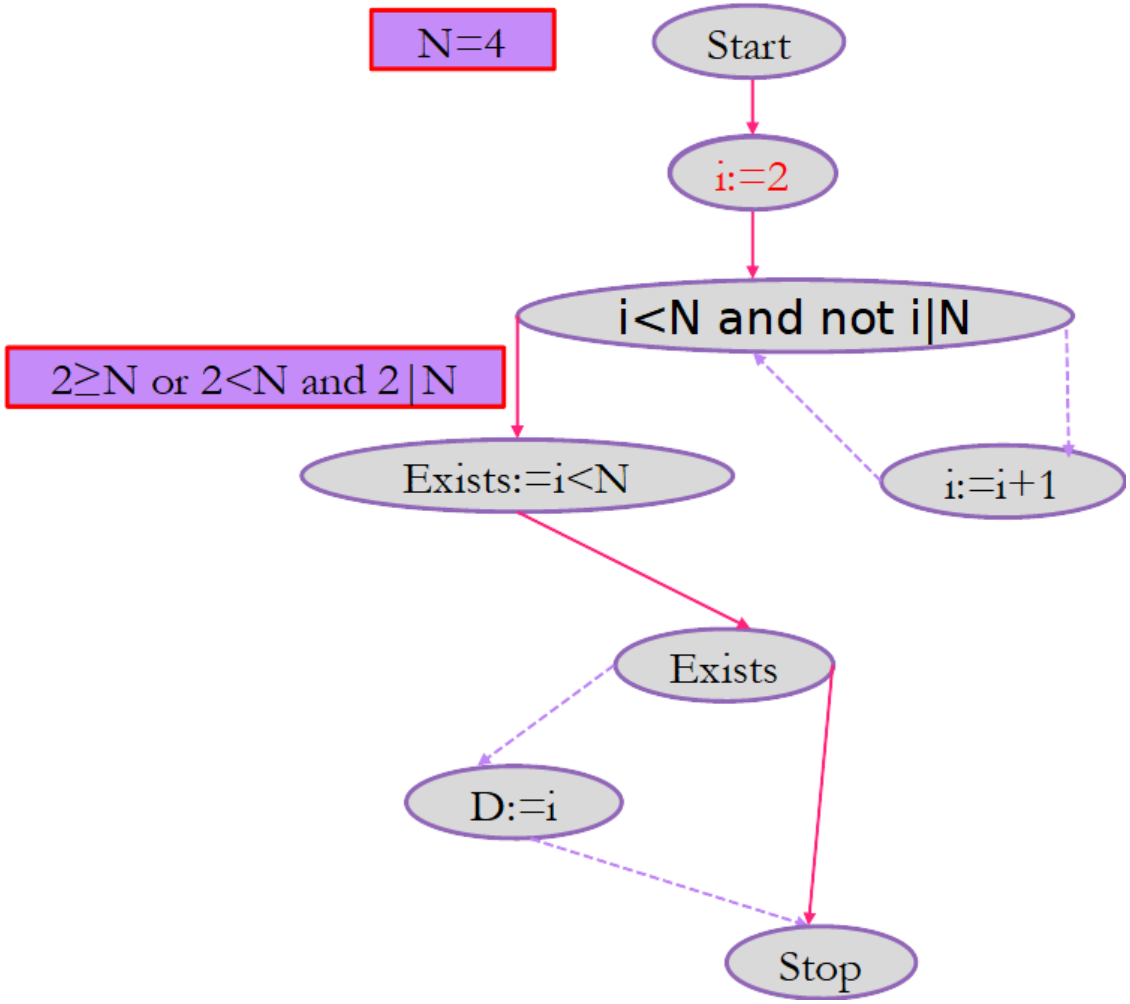
i:=2	
i<N **and not** i|N	
i:=i+1	
Exists:=i<N	
T **Exists** F	
D:=i	**-**

DYNAMIC testing – white box methods

Test path$_2$

Algorithm

i:=2
i<N **and not** i|N
i:=i+1
Exists:=i<N
Exists / T / F
D:=i | -



Special tests

- **Security testing**: the security mechanisms of an information system that protect data and maintain functionality as intended
- **Efficiency testing**: the efficiency of the code, memory usage, ...
- **Functional testing:** is a quality assurance (QA) process (describes what the system does)
- **Stress testing:** put a greater emphasis on robustness, availability, and error handling under a heavy load, rather than on what would be considered correct behavior under normal circumstances
- **Volumen testing:** refers to testing a software application with a certain amount of data (database size, size of an interface file)

Automatation of testing

Generation of test:

- manual
- automatic (generator program)
  - following a rule (regular)
  - randomly

Running of test:

- manual
- automatic (batch file, input and output files, automatic evaluation)


Running with data file (C++)

Principle:

› The standard input/output could be **redirected** to a file. Then the program will use a file for input and output. Thus the file should be the **same structure as console input/output**.

„Technique”:

› You have to put the name of the files as parameters after the compiled code when running.

```
prog.exe <inputfile >outputfile
```

Profit: Comfortable,

easy to administer testing

If there is an output file, the questions appear in that.

`prog.exe >>outputfile`

adding to output file!

Running with data file (C++)

Demo:

› Let's create some input data file (structure should match the input requirements)!

› Let's run the program the following way:

```
prog.exe <1.in >1.out
prog.exe <2.in >2.out
```

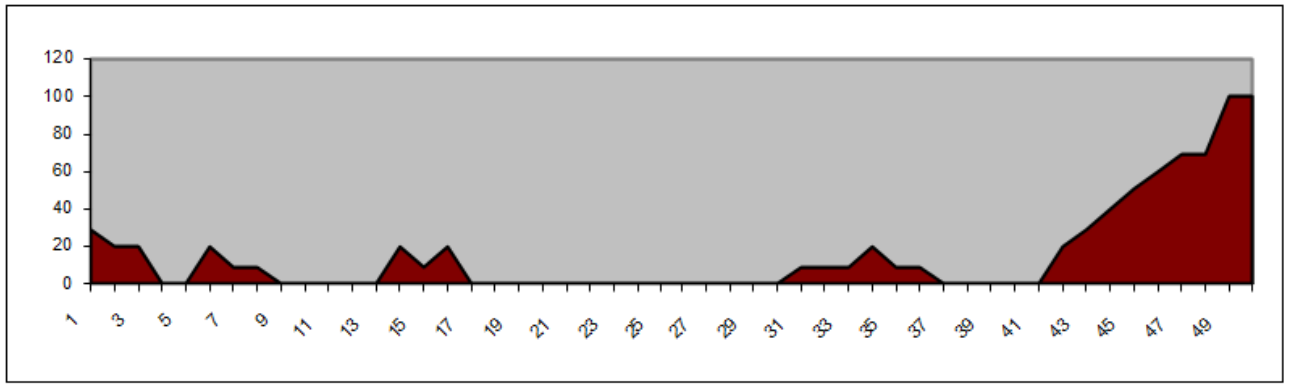
› …

› Check the content of the output files: are they as we expected them to be?

› Comment: we could make testing even easier if we use a batch file.

Generation of test

Task: We flew from Europe to North-America. During the flight, we measured the sea level below us (≥ 0) at some points. We got 0 where we flew above the sea, and >0 where we flew above land. Let's give the widest island!



Generation of test

Task: We flew from Europe to North-America. During the flight, we measured the sea level below us (≥0) at some points. We got 0 where we flew above the sea, and >0 where we flew above land. Let's give the widest island!

Specification:

Input: $N \in \mathbb{N}$, $Hi_{1..N} \in \mathbb{N}^N$

Output: $Exists \in \mathbb{L}$, $B, E \in \mathbb{N}$

…

Test cases:

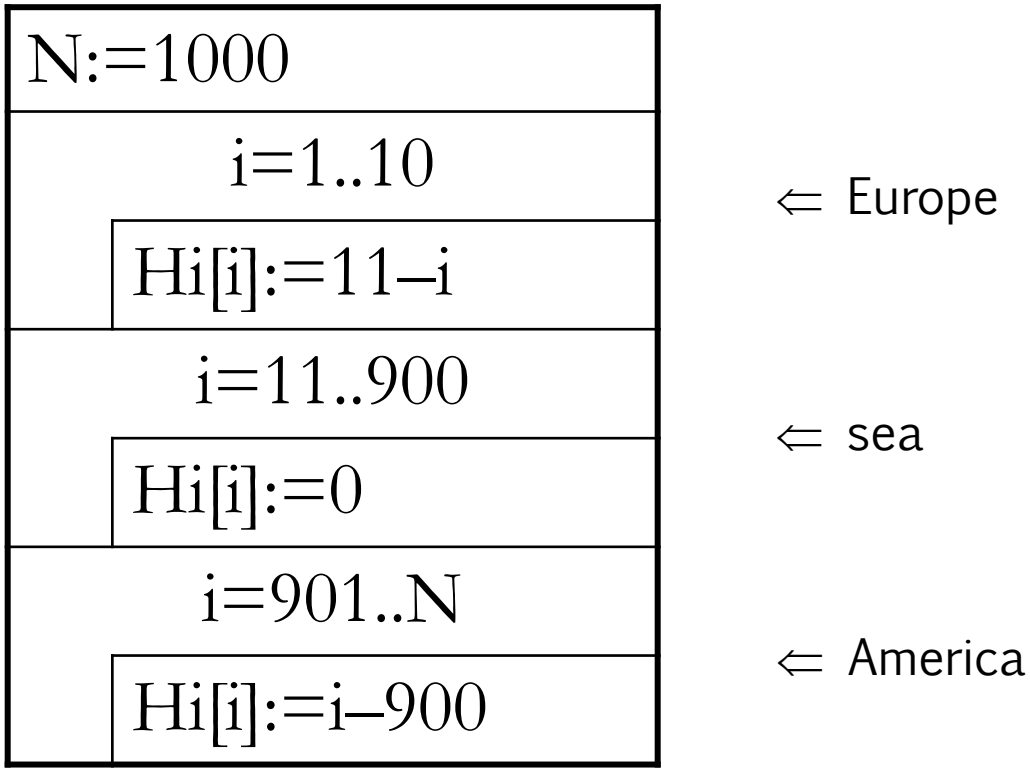
› Small tests – according to testing strategy:

N=3, Hi=(1,0,1) → no island
N=5, Hi =(1,0,1,0,1) → one „small" island
N=7, Hi =(1,0,1,0,1,0,1) → more „small" islands
N=7, Hi =(1,0,1,1,1,0,1) → wider island

› How can we prepare big tests?

Regular tests

We can generate regular tests with loops:



Random tests (basics – random numbers)

Random numbers are generated by an algorithm starting from a start value

$$x_0 \rightarrow f(x_0)=x_1 \rightarrow f(x_1)=x_2 \rightarrow \ldots$$

To be „random” – we need a proper function and a good start value.

- **Start value**: (eg.) from the clock of the computer.
- **Function** (linear congruence method):

```
f(x) = (A*x+B) mod M,
```

where A, B and M are inner constants of the function.

Random tests (basics – C++)

`rand()` give a random number between `0` and a max value (`RAND_MAX`)

`srand(number)` set a start value

- `Random(a..b)∈{a,…,b}` `v=rand() % (b-a+1)+a`
- `Random(N)∈{1,…,N}` `v=rand() % N+1`
- `RandomNumber∈[0,1)⊂ℝ` `v=rand()/(RAND_MAX+1.0)`

Roll the dice example:

```
#include <time.h>
…
srand(time(NULL));
i=rand() % 6 +1;
```

Random tests

Using random numbers to generate random tests:

```
N:=1000
M:=Random(9)
        i=1..M
            Hi[i]:=Random(4..10)
        i=M+1..900
            randomnumber<0.5
            Hi[i]:=0        Hi[i]:=1
        i=901..N
            Hi[i]:=Random(2..8)
```

⇐ Europe

⇐ sea and islands

⇐ America

Debugging

Error types during testing:

› Output is wrong,

› Runtime error occurred,

› No result

› Partial result,

› Writes out not expected elements,

› Writes too much (too many times),

› Program won't stop running,

› ...

Debugging

Aim: to find the **cause** and **location** of detected problems

Principles:

› Thorough **consideration** before using tools

› A detected error could cause problems in **other parts** of the program, as well

› The count of errors and their severity increases in a **non-linear way** (bigger) to the size of the program

› It's equally important to find *why* the program **does not do** things we *would expect*, or *why* **does** things that we would *not expect* it to do

› Only **correct** an error, if you **have found** it!

Debugging

Tools:

- Writing out variables, memory (conditional compilation)
- Setting breakpoints
- Execution step by step
- Tracing data (how changes the value)
- Tracing states (eg. preconditions of parameters, loop variables)
- Postmortem tracking: back from bug
- Special tests (eg. index boundary: `.at(.)`↔`[.]`)

Debugging – C++

Debugging tools:

› Displaying the location and cause of error
- automatically – runtime error
- manually – with standard macros (`__LINE__`, `__func__`, `assert`). Eg.

```
#include <iostream>
#include <cassert>  //for assert
using namespace std;
int main()
{
    int a,b;
    cerr<<"Function: "<<__func__;//name of current function
    cerr<<", line: "<<__LINE__<<endl;//current #, now: 8
    assert(a==0&&b==0);//checking an expectation (a==0&&b==0);
                       //if not met, stop and error message    …
```

Debugging methods

Aim:

› What part of the input work incorrectly?

› Where is the statement in the program that causes the problem?

Method types:

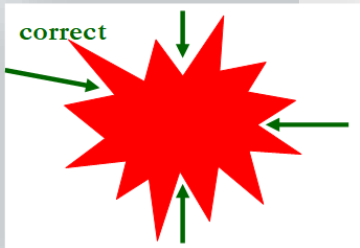
1. Induction (extending the group of incorrects)
2. Deduction (narrowing the group of incorrects)
3. Debugging backwards from the error
4. Debugging with the help of testing (we need a test case that reveals the location of the known error)

Debugging methods – example for induction

Task: Writing an N number between 1 and 99 with letters

- Test cases: N=8⇒OK, N=17⇒OK, N=30⇒incorrect
- Let's try to generalize from the incorrect cases. Let's assume that all the numbers starting with 30 are wrong!
- If we saw this is the case (with testing), then let's generalize, and assume eg. that all above 30 are wrong!
- If we cannot generalize any further, then we know what to look for in the program.
- If we could not generalize at all, then let's try another way: are all the numbers ending in 0 incorrect?
- ...

Debugging methods – example for deduction



Task: Writing an N number between 1 and 99 with letters

- Test cases: N=8⇒OK, N=17⇒OK, N=30⇒incorrect
- Let's assume that it's incorrect for all non-OK cases.
- Try to narrow based on the wrong test cases: let's assume it's good for less than 20
- If we found that this is true (with testing), then narrow it: is is OK for all that are bigger than 39?
- If you cannot narrow it any further, then we have found what we have to look for in the program.
- If not, then try to narrow it another way: let's assume that it's good for all the numbers ending in 0
- ...

Correcting errors

Aim: to correct the detected errors/bugs

Principles:

› You should correct the bug, **not the symptoms**.

› Correcting the bug might cause a problem in **another part of the program** (the correction is wrong, or the original masked another error).

› After correction, you should **test again (repetitive)**!

› The probability of a good correction is **inversely proportional** to the size of the program.

› Correcting an error might lead back to the **designing (planning) phase** (the aim of programming methodology that it does not need to go back).

Documentation

Types:

› Information about the program

› User documentation

› Developer documentation

› …

User documentation - content

- Task description (*summary* and detailed)
- Runtime environment (operating system, *hw/sw requirements*)
- Description of usage (*installation*, questions + possible answers,...)
- Input data, results, *functionalities*
- Samples – sample runs
- Error messages, and causes of most frequent errors

Developer documentation - content

- Task description, specification, *requirement analysis*
- Developer environment (operating system+compiler, ...)
- Data description (representation of task parameters)
- Algorithms, decisions (eg. PoA usage), *other alternatives, reasons*, explanations
- code, *implementation standards and decisions*, test cases
- *Effectiveness metrics*
- Developmental options
- Author(s)