Technical Slide

1. Lesson 1: Testing

   Video 1.1: Testing, sample tests, min/max tests

   Video 1.2: Custom cases and testing workflow

   Video 1.3: Stress-testing

Testing

- Run your program locally on some inputs

Testing

- Run your program locally on some inputs
- Incorrect attempts are penalized

Testing

- Run your program locally on some inputs
- Incorrect attempts are penalized
- You need a test for debug

Testing

- Run your program locally on some inputs
- Incorrect attempts are penalized
- You need a test for debug
- In this lesson:
    - Common types of test cases
    - Testing workflow
    - Stress-testing

What to check

- **Correctness:** compare your output with the correct answer

What to check

- **Correctness:** compare your output with the correct answer
- Need to know the answer — get it manually or otherwise

What to check

- **Correctness:** compare your output with the correct answer
- Need to know the answer — get it manually or otherwise
- **Reliability:** make sure that your program doesn't crash

What to check

- **Correctness:** compare your output with the correct answer
- Need to know the answer — get it manually or otherwise
- **Reliability:** make sure that your program doesn't crash
- Asserts help — check invariants without correct answer

What to check

- **Correctness:** compare your output with the correct answer
- Need to know the answer — get it manually or otherwise
- **Reliability:** make sure that your program doesn't crash
- Asserts help — check invariants without correct answer
- **Limits:** check working time and memory on large inputs

What to check

- **Correctness:** compare your output with the correct answer
- Need to know the answer — get it manually or otherwise
- **Reliability:** make sure that your program doesn't crash
- Asserts help — check invariants without correct answer
- **Limits:** check working time and memory on large inputs
- Locally — detailed information on perfomance

Sample tests

- Always are given, with the answer

Sample tests

- Always are given, with the answer
- Test your understanding of the statement

Sample tests

- Always are given, with the answer
- Test your understanding of the statement
- You could've gotten it wrong

Sample tests

- Always are given, with the answer
- Test your understanding of the statement
- You could've gotten it wrong
- Test your solution before implementing

Sample tests

- Always are given, with the answer
- Test your understanding of the statement
- You could've gotten it wrong
- Test your solution before implementing
- Save time by realizing you're wrong earlier

Sample tests

- Always are given, with the answer
- Test your understanding of the statement
- You could've gotten it wrong
- Test your solution before implementing
- Save time by realizing you're wrong earlier
- Samples check general correctness and sometimes special cases

Sample tests

- Always are given, with the answer
- Test your understanding of the statement
- You could've gotten it wrong
- Test your solution before implementing
- Save time by realizing you're wrong earlier
- Samples check general correctness and sometimes special cases
- Do not rely on samples only!

Minimal test

- Test of minimal size/minimal input values

Minimal test

- Test of minimal size/minimal input values
- Given: integer N ($1 \le N \le 10^6$), then a sequence of N nonnegative integers, each not greater than 10^9

Minimal test

- Test of minimal size/minimal input values
- Given: integer N $(1 \leq N \leq 10^6)$, then a sequence of N nonnegative integers, each not greater than 10^9

```
1
0
```

Minimal test

- Test of minimal size/minimal input values
- Given: integer N $(1 \leq N \leq 10^6)$, then a sequence of N nonnegative integers, each not greater than 10^9

  ```
  1
  0
  ```

- Often is "special"

Minimal test

- Test of minimal size/minimal input values
- Given: integer N ($1 \leq N \leq 10^6$), then a sequence of N nonnegative integers, each not greater than 10^9

```
1
0
```

- Often is “special”
- Easy to construct

Minimal test

- Test of minimal size/minimal input values
- Given: integer N ($1 \le N \le 10^6$), then a sequence of N nonnegative integers, each not greater than 10^9

  ```
  1
  0
  ```

- Often is “special”
- Easy to construct
- Something else could be minimized, e.g. answer size

Maximal test

- Maximal size/maximal input values

Maximal test

- Maximal size/maximal input values
- Given: integer N ($1 \leq N \leq 10^6$), then a sequence of N nonnegative integers, each not greater than 10^9

  1000000

  1000000000 1000000000 1000000000 ...

Maximal test

- Maximal size/maximal input values
- Given: integer N ($1 \leq N \leq 10^6$), then a sequence of N nonnegative integers, each not greater than 10^9

  1000000

  1000000000 1000000000 1000000000 ...

- Hard to compute the answer

Maximal test

- Maximal size/maximal input values
- Given: integer N ($1 \leq N \leq 10^6$), then a sequence of N nonnegative integers, each not greater than 10^9

  ```
  1000000
  1000000000 1000000000 1000000000 ...
  ```

- Hard to compute the answer
- Checks crashes (e.g. array sizes)

Maximal test

- Maximal size/maximal input values
- Given: integer N ($1 \leq N \leq 10^6$), then a sequence of N nonnegative integers, each not greater than 10^9

  1000000

  1000000000 1000000000 1000000000 . . .

- Hard to compute the answer
- Checks crashes (e.g. array sizes)
- TL/ML — but max time not always on any max size test

Maximal test

- Maximal size/maximal input values
- Given: integer N ($1 \le N \le 10^6$), then a sequence of N nonnegative integers, each not greater than 10^9

  1000000

  1000000000 1000000000 1000000000 ...

- Hard to compute the answer
- Checks crashes (e.g. array sizes)
- TL/ML — but max time not always on any max size test
- Integer overflow — if negative answer when should be nonnegative

How to obtain max test

How to obtain max test

- Generate by another program

```cpp
int n = 1000000;
cout << n << '\n';
for (int i = 0; i < n; ++i) {
    cout << int(1e9) << '␣';
}
```

How to obtain max test

- Generate by another program

```cpp
int n = 1000000;
cout << n << '\n';
for (int i = 0; i < n; ++i) {
    cout << int(1e9) << '␣';
}
```

- Plug in inside your code

```cpp
int n;
//cin >> n;
n = 1000000;
for (int i = 0; i < n; ++i) {
    //cin >> a[i];
    a[i] = int(1e9);
}
```

- Better to have special function for reading data, to replace it as a whole

```cpp
void readInput() {
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cin >> a[i];
    }
}
void setInput() {
    n = 1000000;
    for (int i = 0; i < n; ++i) {
        a[i] = int(1e9);
    }
}
int main() {
    //readInput();
    setInput();
}
```

Technical Slide

1. Lesson 1: Testing
   - Video 1.1: Testing, sample tests, min/max tests
   - Video 1.2: Custom cases and testing workflow
   - Video 1.3: Stress-testing

Specific problem types

- String problems

  `aaaaaa`

  `abcdef`

Specific problem types

- String problems
  `aaaaaa`
  `abcdef`
- Problems about divisibility — prime numbers, numbers with many divisors
  2, 3, 11, 31, 997, $10^9 + 7$ are prime
  48 has 10 divisors, 931 170 240 has 1344

Specific problem types

- String problems
  `aaaaaa`
  `abcdef`
- Problems about divisibility — prime numbers, numbers with many divisors
  2, 3, 11, 31, 997, $10^9 + 7$ are prime
  48 has 10 divisors, 931 170 240 has 1344
- Graphs, geometry, . . .

Program structure

- Test all branches in your code

```
if (condition) {
    ...
} else {
    ...
}
```

Include test with `condition` true, and `condition` false

Program structure

- Test all branches in your code

```
if (condition) {
    ...
} else {
    ...
}
```

  Include test with `condition` true, and `condition` false

- Different answer types (YES/NO, -1 for there is no answer, etc)

Program structure

- Test all branches in your code

```
if (condition) {
    ...
} else {
    ...
}
```

  Include test with `condition` true, and `condition` false

- Different answer types (YES/NO, -1 for there is no answer, etc)
- Test different parts separately, each right after it's finished

Custom tests

- Make “interesting” tests — but note that they are not necessarily interesting for your solution

Custom tests

- Make “interesting” tests — but note that they are not necessarily interesting for your solution
- Test different run patterns, special cases, pathological cases — depends on the solution and its proof

Custom tests

- Make “interesting” tests — but note that they are not necessarily interesting for your solution
- Test different run patterns, special cases, pathological cases — depends on the solution and its proof
- Combine all of the above

Testing stages

Testing stages

1. Before submission — to not waste attempts

Testing stages

1. Before submission — to not waste attempts
2. After submission — to find a test case for debugging

How long to test before submitting

- Time is limited, so there is always a trade-off between “test well” and “test fast”

How long to test before submitting

- Time is limited, so there is always a trade-off between “test well” and “test fast”
- Depends on the rules

How long to test before submitting

- Time is limited, so there is always a trade-off between “test well” and “test fast”
- Depends on the rules
- Depends on complexity and how sure you are in your solution

How long to test before submitting

- Time is limited, so there is always a trade-off between “test well” and “test fast”
- Depends on the rules
- Depends on complexity and how sure you are in your solution
- Always check on samples — that your program works at all, and that the format is correct

How long to test before submitting

- Time is limited, so there is always a trade-off between “test well” and “test fast”
- Depends on the rules
- Depends on complexity and how sure you are in your solution
- Always check on samples — that your program works at all, and that the format is correct
- Nearly always test on cases other than samples

Testing workflow

Testing workflow

- After fixing a bug, start testing from the beginning

Testing workflow

- After fixing a bug, start testing from the beginning
- Save all tests you’ve come up with — so you don’t need to invent them again

Testing workflow

- After fixing a bug, start testing from the beginning
- Save all tests you've come up with — so you don't need to invent them again
- Check on all your tests on one run

Testing workflow

- After fixing a bug, start testing from the beginning
- Save all tests you've come up with — so you don't need to invent them again
- Check on all your tests on one run
  - All tests are saved in one file one after another, and your program solves input cases repeatedly until the end of file, not just one test

Testing workflow

- After fixing a bug, start testing from the beginning
- Save all tests you've come up with — so you don't need to invent them again
- Check on all your tests on one run
    - All tests are saved in one file one after another, and your program solves input cases repeatedly until the end of file, not just one test
    - Tests are saved with special extension (e.g. 01.in, 02.in, ...), and you have a script to run your program on all files with it (like *.in)

Testing workflow

- After fixing a bug, start testing from the beginning
- Save all tests you've come up with — so you don't need to invent them again
- Check on all your tests on one run
    - All tests are saved in one file one after another, and your program solves input cases repeatedly until the end of file, not just one test
    - Tests are saved with special extension (e.g. 01.in, 02.in, ...), and you have a script to run your program on all files with it (like *.in)
    - Use some unit-testing software to manage tests, like JUnit

Technical Slide

1. Lesson 1: Testing
   - Video 1.1: Testing, sample tests, min/max tests
   - Video 1.2: Custom cases and testing workflow
   - Video 1.3: Stress-testing

Stress-testing

- You can make the computer invent tests for you!

Stress-testing

- You can make the computer invent tests for you!
- Write a *generator* program, which outputs some random input

Stress-testing

- You can make the computer invent tests for you!
- Write a *generator* program, which outputs some random input
- Repeatedly:
  - Generate a random input
  - Run your solution on it
  - Check if the output is correct
  - If not — stop and output the test case

Stress-testing

- You can make the computer invent tests for you!
- Write a *generator* program, which outputs some random input
- Repeatedly:
  - Generate a random input
  - Run your solution on it
  - Check if the output is correct
  - If not — stop and output the test case
- Fully automated, thousands tests per second!

How to check correctness

How to check correctness

- No need if only interested in crashes, so utilize asserts

How to check correctness

- No need if only interested in crashes, so utilize asserts
- *Trivial* solution — as simple as possible

How to check correctness

- No need if only interested in crashes, so utilize asserts
- *Trivial* solution — as simple as possible
- It's correct, and maybe slow — so small tests

How to check correctness

- No need if only interested in crashes, so utilize asserts
- *Trivial* solution — as simple as possible
- It's correct, and maybe slow — so small tests
- Obtain the correct answer via the trivial solution

How to check correctness

- No need if only interested in crashes, so utilize asserts
- *Trivial* solution — as simple as possible
- It's correct, and maybe slow — so small tests
- Obtain the correct answer via the trivial solution
- Tests outputs for equality or use custom *checker*

How to check correctness

- No need if only interested in crashes, so utilize asserts
- *Trivial* solution — as simple as possible
- It's correct, and maybe slow — so small tests
- Obtain the correct answer via the trivial solution
- Tests outputs for equality or use custom *checker*
- In total — a small version of a testing system

Generators

Generators

- Generate small tests — faster (esp. for trivial solution), easier debuging

Generators

- Generate small tests — faster (esp. for trivial solution), easier debuging
- Make parameters to easily tweak test size

Generators

- Generate small tests — faster (esp. for trivial solution), easier debuging
- Make parameters to easily tweak test size
- Example:
    - Fails only on `aaaaa`, `zzz`, …

Generators

- Generate small tests — faster (esp. for trivial solution), easier debuging
- Make parameters to easily tweak test size
- Example:
  - Fails only on `aaaaa`, `zzz`, ...
  - Random 'a'–'z' strings of length 10: probability of $26^{-9} \simeq 2 \cdot 10^{-13}$

Generators

- Generate small tests — faster (esp. for trivial solution), easier debuging
- Make parameters to easily tweak test size
- Example:
    - Fails only on aaaaa, zzz, ...
    - Random 'a'–'z' strings of length 10: probability of $26^{-9} \simeq 2 \cdot 10^{-13}$
    - Only 'a', 'b', 'c' and length 5: $3^{-4} \simeq 0.01$

Generators

- Generate small tests — faster (esp. for trivial solution), easier debuging
- Make parameters to easily tweak test size
- Example:
  - Fails only on `aaaaa`, `zzz`, ...
  - Random 'a'–'z' strings of length 10: probability of $26^{-9} \simeq 2 \cdot 10^{-13}$
  - Only 'a', 'b', 'c' and length 5: $3^{-4} \simeq 0.01$
- Do not lose generality

  Strings of 'a' far less interesting than strings of 'a' and 'b'

Generators

- Generate small tests — faster (esp. for trivial solution), easier debuging
- Make parameters to easily tweak test size
- Example:
  - Fails only on `aaaaa`, `zzz`, ...
  - Random 'a'–'z' strings of length 10: probability of $26^{-9} \simeq 2 \cdot 10^{-13}$
  - Only 'a', 'b', 'c' and length 5: $3^{-4} \simeq 0.01$
- Do not lose generality
  Strings of 'a' far less interesting than strings of 'a' and 'b'
- Correctly initialize random to get different tests

Stress-test for crashes

```
for (( test=1; ; test++ ))
do
    echo Test $test
    ./generate > in
    ./solution < in > out
    if [ $? -ne 0 ]
    then
        echo Runtime error
        break
    fi
done
```

Terminates on error, so error test is in the `in` file afterwards

Stress-test for correctness

```
for (( test=1; ; test++ ))
do
    echo Test $test
    ./generate > in
    ./solution < in > out
    ./solution_trivial < in > ans
    diff out ans
    if [ $? -ne 0 ]
    then
        echo Wrong answer
        break
    fi
done
```

Stress-testing workflow

Stress-testing workflow

- Stress-test after manual testing

Stress-testing workflow

- Stress-test after manual testing
- No point if generator/trivial solution/checker is too complex

Stress-testing workflow

- Stress-test after manual testing
- No point if generator/trivial solution/checker is too complex
- Start with very small test sizes

Stress-testing workflow

- Stress-test after manual testing
- No point if generator/trivial solution/checker is too complex
- Start with very small test sizes
- Couple of minutes running is usually enough

Stress-testing workflow

- Stress-test after manual testing
- No point if generator/trivial solution/checker is too complex
- Start with very small test sizes
- Couple of minutes running is usually enough
- While running do something else useful

Stress-testing workflow

- Stress-test after manual testing
- No point if generator/trivial solution/checker is too complex
- Start with very small test sizes
- Couple of minutes running is usually enough
- While running do something else useful
- If nothing is found, generate larger tests
  Or rethink the generator

Summary

Summary

- Test your solution before and after submitting

Summary

- Test your solution before and after submitting
- Start with samples

Summary

- Test your solution before and after submitting
- Start with samples
- “Interesting” manual cases — min/max, problem type specific, and anything you could imagine

Summary

- Test your solution before and after submitting
- Start with samples
- "Interesting" manual cases — min/max, problem type specific, and anything you could imagine
- Test different parts separately

Summary

- Test your solution before and after submitting
- Start with samples
- “Interesting” manual cases — min/max, problem type specific, and anything you could imagine
- Test different parts separately
- If everything else fails, run a stress-test

Summary

- Test your solution before and after submitting
- Start with samples
- "Interesting" manual cases — min/max, problem type specific, and anything you could imagine
- Test different parts separately
- If everything else fails, run a stress-test
- Watch out for the generator
  Generate small tests