Polyman Complete User Guide

Version 1.0.0 | A comprehensive guide for creating competitive programming problems using Polyman CLI

Table of Contents

1. Introduction
    - What is Polyman?
    - Key Features
    - Prerequisites
2. Getting Started
    - Installation
    - Creating Your First Problem
    - Directory Structure
3. Configuration File Reference
    - Overview
    - Basic Properties
    - Statements
    - Solutions
    - Generators
    - Checker
    - Validator
    - Testsets
4. Writing Validators
    - Purpose
    - Basic Structure
    - Common Validation Functions
    - Validator Self-Tests
    - Do's and Don'ts
    - Advanced Example
5. Writing Checkers
    - Purpose
    - Basic Structure
    - Checker Streams
    - Checker Verdicts
    - Checker Self-Tests
    - Do's and Don'ts
    - Floating-Point Checker Example
    - Multiple Answer Checker Example
6. Writing Generators
    - Purpose
    - Basic Structure
    - Random Functions
    - Do's and Don'ts

- Advanced Example

7. Writing Solutions
   - Main Correct Solution (MA)
   - Other Solution Types
   - Do's and Don'ts
8. Test Generation
   - Generation Workflow
   - Manual Tests
   - Generated Tests
   - Test Organization
9. CLI Commands Reference
   - Problem Creation
   - Test Management
   - Validation
   - Solution Execution
   - Testing Components
   - Full Verification
10. Best Practices
    - Directory Organization
    - Configuration Management
    - Test Coverage
    - Solution Testing
    - Version Control
11. Troubleshooting
    - Compilation Errors
    - Validation Errors
    - Checker Errors
    - Solution Errors
    - Test Generation Errors
    - Memory Issues
    - Time Issues
12. FAQ
    - Do I need a custom checker?
    - Can I use Python for main solution?
    - How many tests should I include?
    - Interactive vs Regular problems?
    - How to handle floating-point?
    - Multiple problems in one directory?
    - How to debug WA solutions?
    - Validator too strict/lenient?

Introduction

What is Polyman?

Polyman is a command-line tool designed for competitive programming problem setters to create, validate, and verify problems locally before uploading to Codeforces Polygon or other platforms.

Key Features

- ✅ **Local Problem Development**: Create and test problems entirely on your machine
- ✅ **Comprehensive Validation**: Validate inputs, check outputs, and verify solutions
- ✅ **Multiple Languages**: Support for C++, Java, and Python solutions
- ✅ **Standard Checkers**: Built-in testlib checkers for common comparison types
- ✅ **Full Verification**: Complete workflow automation for problem testing

Prerequisites

- **Node.js** v14 or higher
- **C++ Compiler** (g++, clang, or MSVC)
- **Java JDK** (optional, for Java solutions)
- **Python** (optional, for Python solutions)
- **testlib.h** (automatically downloadable via Polyman)

Getting Started

Installation

```
npm install -g polyman-cli
```

Creating Your First Problem

```
# Create a new problem template
polyman new my-problem

# Navigate to the problem directory
cd my-problem

# Download testlib.h (required for validators, checkers, generators)
polyman download-testlib
```

Directory Structure

After creating a new problem, you'll have:

```
my-problem/
├── Config.json              # Main configuration file
├── testlib.h               # Testlib header (after download)
├── checker/
│   ├── chk.cpp             # Checker implementation
│   └── checker_tests.json # Checker self-tests
├── validator/
│   ├── val.cpp             # Validator implementation
```

```
│   └── validator_tests.json # Validator self-tests
├── generators/
│   └── gen.cpp             # Test generator
├── solutions/
│   ├── acc.cpp             # Main correct solution
│   ├── acc2.java           # Alternative correct solution
│   └── tle.py              # Time limit solution
├── testsets/
│   └── tests/              # Generated tests appear here
└── statements/
    ├── english/            # English problem statement
    └── russian/            # Russian problem statement
```

Configuration File Reference

Overview

`Config.json` is the central configuration file that defines all aspects of your problem.

Basic Properties

Required Fields

```json
{
  "name": "my-problem-name",
  "timeLimit": 1000,
  "memoryLimit": 256,
  "inputFile": "stdin",
  "outputFile": "stdout",
  "interactive": false
}
```

Field	Type	Description	Possible Values
`name`	string	Problem identifier	Any valid string
`timeLimit`	number	Time limit in milliseconds	100-15000 (typical: 1000-2000)
`memoryLimit`	number	Memory limit in megabytes	4-1024 (typical: 256-512)
`inputFile`	string	Input source	`"stdin"` or filename like `"input.txt"`
`outputFile`	string	Output destination	`"stdout"` or filename like `"output.txt"`
`interactive`	boolean	Interactive problem flag	`true` or `false`

⚠️ **Important Notes:**

- Time limits are in **milliseconds** (1000ms = 1 second)
- Memory limits are in **megabytes**
- Use `"stdin"` / `"stdout"` for standard I/O problems
- Set `interactive: true` only for interactive problems (Interactive Problems Are Not Supported Yet)

Optional Fields

```
{
  "description": "A brief description of the problem",
  "tags": ["implementation", "math", "greedy"],
  "tutorial": "Solution explanation and approach"
}
```

Statements

Define problem statements in multiple languages:

```
{
  "statements": {
    "english": {
      "encoding": "UTF-8",
      "name": "Problem Title",
      "legend": "./statements/english/legend.tex",
      "input": "./statements/english/input-format.tex",
      "output": "./statements/english/output-format.tex",
      "notes": "./statements/english/notes.tex"
    },
    "russian": {
      "encoding": "UTF-8",
      "name": "Название Задачи",
      "legend": "./statements/russian/legend.tex",
      "input": "./statements/russian/input-format.tex",
      "output": "./statements/russian/output-format.tex"
    }
  }
}
```

✅ Do's:

- Always use UTF-8 encoding
- Store statement files in respective language folders
- Use LaTeX format for mathematical expressions

× Don'ts:

- Don't use absolute paths for statement files
- Don't mix encodings within the same problem

Solutions

Define all solutions with their expected behavior:

```
{
  "solutions": [
    {
      "name": "main",
      "source": "./solutions/acc.cpp",
```

```json
      "tag": "MA",
      "sourceType": "cpp.g++17"
    },
    {
      "name": "wa-solution",
      "source": "./solutions/wa.cpp",
      "tag": "WA",
      "sourceType": "cpp.g++17"
    },
    {
      "name": "tle-solution",
      "source": "./solutions/tle.py",
      "tag": "TL",
      "sourceType": "python.3"
    }
  ]
}
```

Solution Tags

Tag	Meaning	Description
`MA`	Main Correct	**Required**. The reference solution (must exist)
`OK`	Correct	Alternative correct solution
`WA`	Wrong Answer	Should get Wrong Answer on some tests
`TL`	Time Limit	Should exceed time limit on some tests
`TO`	Time/OK	May TLE but is algorithmically correct
`ML`	Memory Limit	Should exceed memory limit
`RE`	Runtime Error	Should crash or have runtime errors
`PE`	Presentation Error	Wrong output format
`RJ`	Rejected	Should fail for any reason

Source Types

C++ Compilers:

- `cpp.g++11` , `cpp.g++14` , `cpp.g++17` , `cpp.g++20`
- `cpp.ms2017` , `cpp.ms2019`
- `cpp.clang++17` , `cpp.clang++20`

Java Versions:

- `java.8` , `java.11` , `java.17` , `java.21`

Python Versions:

- `python.2` , `python.3` , `python.pypy2` , `python.pypy3`

✅ **Do's:**

- Always include exactly **one** solution with tag `MA`
- Include solutions with different expected behaviors (WA, TL, etc.)
- Use appropriate sourceType for each solution
- You May Leave The Source Types Empty to Use Default Compilers

× **Don'ts:**

- Don't have multiple `MA` solutions
- Don't forget to test non-MA solutions
- Don't use Python for time-critical main solutions

Generators

Define test generators:

```
{
  "generators": [
    {
      "name": "gen-random",
      "source": "./generators/random.cpp",
      "sourceType": "cpp.g++17"
    },
    {
      "name": "gen-special",
      "source": "./generators/special.cpp",
      "sourceType": "cpp.g++17"
    }
  ]
}
```

⚠️ **Important:** Generators **must** be C++ and use testlib.h

× **Don'ts:**

- Don't use absolute paths for statement files
- Don't mix encodings within the same problem

Checker

Define output checker:

```
{
  "checker": {
    "name": "custom_checker",
    "source": "./checker/chk.cpp",
    "testsFilePath": "./checker/checker_tests.json",
    "isStandard": false
  }
}
```

For Standard Checkers:

```json
{
  "checker": {
    "name": "wcmp",
    "source": "./checker/wcmp.cpp",
    "isStandard": true
  }
}
```

Available Standard Checkers

Use `polyman list-checkers` to see all available checkers:

- **wcmp**: Compare tokens (whitespace-insensitive)
- **ncmp**: Compare numbers with absolute/relative error
- **fcmp**: Compare floating-point numbers
- **lcmp**: Compare lines exactly
- **yesno**: Compare yes/no answers
- And many more...

✅ **Do's:**

- Use standard checkers when possible (wcmp for most problems)
- Include checker tests in `checker_tests.json`
- Test your custom checker thoroughly

× **Don'ts:**

- Don't write custom checkers unless necessary

Validator

Define input validator:

```json
{
  "validator": {
    "name": "validator",
    "source": "./validator/val.cpp",
    "testsFilePath": "./validator/validator_tests.json"
  }
}
```

⚠️ **Important:** Validators **must** be C++ and use testlib.h

× **Don'ts:**

- Don't write custom checkers unless necessary

Testsets

Testsets are collections of test cases that define how your problem will be tested. Each testset can contain multiple tests organized into groups.

Understanding Testsets

What is a Testset?

- A testset is a named collection of test cases
- Common names: `"tests"` , `"pretests"` , `"system-tests"`
- Each testset generates its own folder in `testsets/<testset-name>/`
- Most problems have just one testset called `"tests"`

Multiple Testsets Example:

```
{
  "testsets": [
    {
      "name": "pretests",
      "groupsEnabled": true,
      "groups": [{"name": "samples"}],
      "generatorScript": { "commands": [...] }
    },
    {
      "name": "system-tests",
      "groupsEnabled": true,
      "groups": [{"name": "full"}],
      "generatorScript": { "commands": [...] }
    }
  ]
}
```

⚠️ **For Most Problems:** Use a single testset named `"tests"`

Understanding Groups

What are Groups?

- Groups organize tests within a testset into logical categories
- Enable better organization and targeted testing
- Can be enabled/disabled with `groupsEnabled` field

When to Use Groups:

✅ **Use Groups (`groupsEnabled: true`):**

- When you want to organize tests by type (samples, edge cases, stress tests)
- When you need to run specific categories of tests separately
- For better organization in larger problem sets

```
{
  "groupsEnabled": true,
  "groups": [
    { "name": "samples" }, // Sample tests shown in problem statement
    { "name": "small" }, // Small tests for debugging
    { "name": "main" }, // Main test cases
    { "name": "edge" }, // Edge cases and boundaries
    { "name": "stress" } // Large random tests
  ]
}
```

```
{
  "groupsEnabled": false,
  "generatorScript": {
    "commands": [
      // No "group" field needed in commands
    ]
  }
}
```

⚠️ **Important:** If `groupsEnabled: true`, you **must**:

1. Define groups in the `groups` array
2. Specify a `group` field in each command
3. Use only group names that are defined

Complete Testset Structure

```
{
  "testsets": [
    {
      "name": "tests", // Testset name (required)
      "groupsEnabled": true, // Enable/disable groups (required)
      "groups": [
        // Group definitions (required if groupsEnabled: true)
        {
          "name": "samples" // Group name (required)
        },
        {
          "name": "main"
        },
        {
          "name": "edge-cases"
        }
      ],
      "generatorScript": {
        // Test generation commands (required)
        "commands": [
          {
            "type": "manual", // Command type (required)
            "manualFile": "./tests/manual/sample1.txt",
            "group": "samples" // Group assignment (required if groupsEnabled: true)
          },
          {
            "type": "generator-single",
            "generator": "gen-random",
            "number": 10,
            "group": "main"
          },
          {
            "type": "generator-range",
```

```
              "generator": "gen-random",
              "range": [100, 1000],
              "group": "main"
            }
          ]
        }
      }
    ]
  }
```

Generator Command Types

Commands define how individual tests are created. There are three types:

---

1. Manual Tests

Manual tests use pre-written test files that you create yourself.

```
{
  "type": "manual", // Required: Command type
  "manualFile": "./tests/manual/test.txt", // Required: Path to test file
  "group": "samples" // Required if groupsEnabled: true
}
```

What Happens:

1. Polyman reads the file at `manualFile` path
2. Copies it to `testsets/<testset-name>/test<index>.txt`
3. The file is used as-is (when generating the testset, gets copied directly)

When to Use:

- Sample tests (shown in problem statement)
- Carefully crafted edge cases
- Tests that are hard to generate programmatically
- Initial tests while developing generators

Example Setup:

```
# Create manual test files
mkdir -p tests/manual
echo "5 3" > tests/manual/sample1.txt
echo "10 7" > tests/manual/sample2.txt
```

```
{
  "commands": [
    {
      "type": "manual",
      "manualFile": "./tests/manual/sample1.txt",
      "group": "samples"
    },
```

```
    {
      "type": "manual",
      "manualFile": "./tests/manual/sample2.txt",
      "group": "samples"
    }
  ]
}
```

Result:

- `testsets/tests/test1.txt` (contains: `5 3`)
- `testsets/tests/test2.txt` (contains: `10 7`)

⚠️ **Important:**

- File must exist before running `polyman generate`
- Use relative paths from `Config.json` location
- Test index is assigned sequentially if not specified

---

2. Single Generator Call

Executes a generator once with a specific parameter.

```
{
  "type": "generator-single", // Required: Command type
  "generator": "gen-random", // Required: Generator name (must be defined in generators)
  "number": 42, // Required: Parameter passed to generator
  "group": "main" // Required if groupsEnabled: true
}
```

What Happens:

1. Polyman compiles the generator (e.g., `gen-random.cpp`)
2. Runs: `./gen-random 42`
3. Captures the output
4. Saves to `testsets/<testset-name>/test<index>.txt`

When to Use:

- Specific test cases with known parameters
- Tests with particular characteristics (e.g., n=1000, n=10^5)
- When you need control over exact parameters

Example:

```
{
  "commands": [
    {
      "type": "generator-single",
      "generator": "gen-random",
      "number": 10,
      "group": "small"
```

```
    },
    {
      "type": "generator-single",
      "generator": "gen-random",
      "number": 1000,
      "group": "main"
    },
    {
      "type": "generator-single",
      "generator": "gen-random",
      "number": 100000,
      "group": "stress"
    }
  ]
}
```

Execution:

- Test 1: Runs `gen-random 10` → generates small test
- Test 2: Runs `gen-random 1000` → generates medium test
- Test 3: Runs `gen-random 100000` → generates large test

Generator Parameters: The `number` field is passed as `argv[1]` to your generator:

```cpp
int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);
    int n = atoi(argv[1]);  // Receives the "number" from config
    // Generate test with parameter n
}
```

⚠️ **Important:**

- Generator must be defined in `generators` array
- Generator must handle the parameter correctly
- Each command creates exactly one test

---

3. Range Generator Calls

Executes a generator multiple times with different parameters.

```
{
  "type": "generator-range", // Required: Command type
  "generator": "gen-random", // Required: Generator name
  "range": [1, 100], // Required: [start, end] inclusive range
  "group": "stress" // Required if groupsEnabled: true
}
```

What Happens:

1. Polyman compiles the generator
2. Runs the generator once for each number in the range
3. For range `[1, 100]`: runs `gen-random 1`, `gen-random 2`, ..., `gen-random 100`

4. Creates 100 test files

When to Use:

- Generating many similar tests with varying sizes
- Stress testing with gradual complexity increase
- Creating comprehensive test coverage

Example:

```json
{
  "commands": [
    {
      "type": "generator-range",
      "generator": "gen-random",
      "range": [1, 10],
      "group": "small"
    },
    {
      "type": "generator-range",
      "generator": "gen-random",
      "range": [100, 1000],
      "group": "main"
    }
  ]
}
```

Execution:

- Tests 1-10: Runs `gen-random 1` , `gen-random 2` , ..., `gen-random 10`
- Tests 11-910: Runs `gen-random 100` , `gen-random 101` , ..., `gen-random 1000`
- **Total:** 910 tests created

⚠️ **Important:**

- Range is **inclusive**: `[1, 100]` generates 100 tests
- Be careful with large ranges (can create many files)
- Tests are numbered sequentially across all commands

Advanced Usage:

```json
{
  "commands": [
    // Manual samples
    {
      "type": "manual",
      "manualFile": "./tests/manual/sample1.txt",
      "group": "samples"
    },
    {
      "type": "manual",
      "manualFile": "./tests/manual/sample2.txt",
      "group": "samples"
    },
```

```
    // Small tests (n from 1 to 10)
    {
      "type": "generator-range",
      "generator": "gen-random",
      "range": [1, 10],
      "group": "small"
    },
    // Specific medium tests
    {
      "type": "generator-single",
      "generator": "gen-random",
      "number": 100,
      "group": "main"
    },
    {
      "type": "generator-single",
      "generator": "gen-random",
      "number": 1000,
      "group": "main"
    },
    // Large stress tests (n from 10000 to 10100)
    {
      "type": "generator-range",
      "generator": "gen-random",
      "range": [10000, 10100],
      "group": "stress"
    }
  ]
}
```

Result:

- Tests 1-2: Manual samples
- Tests 3-12: Small generated tests (n=1 to n=10)
- Tests 13-14: Specific tests (n=100, n=1000)
- Tests 15-115: Stress tests (n=10000 to n=10100)
- **Total:** 115 tests

Test Numbering and Indexing

Automatic Numbering: Tests are numbered sequentially starting from 1:

```
{
  "commands": [
    { "type": "manual", "manualFile": "./tests/manual/sample1.txt" }, // test1.txt
    { "type": "manual", "manualFile": "./tests/manual/sample2.txt" }, // test2.txt
    { "type": "generator-single", "generator": "gen", "number": 10 }, // test3.txt
    { "type": "generator-range", "generator": "gen", "range": [1, 3] } // test4.txt,
test5.txt, test6.txt
  ]
}
```

Manual Index Assignment: You can specify the test number explicitly for manual tests:

```json
{
  "commands": [
    {
      "type": "manual",
      "manualFile": "./tests/manual/sample1.txt"
    },
    {
      "type": "manual",
      "manualFile": "./tests/manual/special.txt"
    }
  ]
}
```

⚠️ **Note:** Manual index assignment is primarily for manual tests. Generator commands use automatic numbering.

Practical Examples

Example 1: Simple Problem (No Groups)

```json
{
  "testsets": [
    {
      "name": "tests",
      "groupsEnabled": false,
      "generatorScript": {
        "commands": [
          { "type": "manual", "manualFile": "./tests/manual/sample1.txt" },
          { "type": "manual", "manualFile": "./tests/manual/sample2.txt" },
          { "type": "generator-range", "generator": "gen", "range": [1, 50] }
        ]
      }
    }
  ]
}
```

Result: 52 tests (2 manual + 50 generated)

Example 2: Standard Problem (With Groups)

```json
{
  "testsets": [
    {
      "name": "tests",
      "groupsEnabled": true,
      "groups": [{ "name": "samples" }, { "name": "main" }, { "name": "edge" }],
      "generatorScript": {
        "commands": [
```

```
          // Samples
          {
            "type": "manual",
            "manualFile": "./tests/manual/sample1.txt",
            "group": "samples"
          },
          {
            "type": "manual",
            "manualFile": "./tests/manual/sample2.txt",
            "group": "samples"
          },

          // Main tests
          {
            "type": "generator-range",
            "generator": "gen-random",
            "range": [10, 100],
            "group": "main"
          },

          // Edge cases
          {
            "type": "generator-single",
            "generator": "gen-edge",
            "number": 1,
            "group": "edge"
          },
          {
            "type": "generator-single",
            "generator": "gen-edge",
            "number": 100000,
            "group": "edge"
          }
        ]
      }
    }
  ]
}
```

Result: 95 tests

- 2 samples
- 91 main tests (n=10 to n=100)
- 2 edge tests (n=1, n=100000)

Example 3: Multiple Generators

```
{
  "generators": [
    { "name": "gen-random", "source": "./generators/random.cpp" },
    { "name": "gen-worst", "source": "./generators/worst-case.cpp" },
```

```
    { "name": "gen-special", "source": "./generators/special.cpp" }
  ],
  "testsets": [
    {
      "name": "tests",
      "groupsEnabled": true,
      "groups": [
        { "name": "samples" },
        { "name": "random" },
        { "name": "worst" },
        { "name": "special" }
      ],
      "generatorScript": {
        "commands": [
          // Samples
          {
            "type": "manual",
            "manualFile": "./tests/manual/sample1.txt",
            "group": "samples"
          },

          // Random tests
          {
            "type": "generator-range",
            "generator": "gen-random",
            "range": [1, 30],
            "group": "random"
          },

          // Worst case tests
          {
            "type": "generator-range",
            "generator": "gen-worst",
            "range": [1, 20],
            "group": "worst"
          },

          // Special cases
          {
            "type": "generator-single",
            "generator": "gen-special",
            "number": 1,
            "group": "special"
          },
          {
            "type": "generator-single",
            "generator": "gen-special",
            "number": 2,
            "group": "special"
          }
        ]
      }
```

```
    }
  ]
}
```

Result: 53 tests from 3 different generators

Important Notes and Best Practices

✅ Do's:

1. **Always include sample tests**
    - Use manual tests for samples shown in problem statement
    - Place in separate "samples" group
    - 2-3 samples are typical
2. **Use meaningful group names**

```
{"name": "samples"}      // ✅ Clear
{"name": "edge-cases"}   // ✅ Descriptive
{"name": "group1"}       // ❌ Not descriptive
```

3. **Order tests logically**
    - Start with manual samples
    - Progress from small to large
    - End with stress tests
4. **Validate generated tests**

```
polyman generate tests
polyman validate all
```

5. **Use range for many similar tests**

```
// ✅ Good: One command for 100 tests
{"type": "generator-range", "generator": "gen", "range": [1, 100]}

// ❌ Bad: 100 separate commands
{"type": "generator-single", "generator": "gen", "number": 1}
{"type": "generator-single", "generator": "gen", "number": 2}
// ... (98 more)
```

× Don'ts:

1. **Don't forget to define groups**

```
{
  "groupsEnabled": true,
  "groups": [], // ❌ Empty groups array
  "generatorScript": {
    "commands": [
```

```
      { "type": "manual", "manualFile": "./test.txt", "group": "samples" } // ❌ "samples" not defined
    ]
  }
}
```

2. **Don't use undefined groups**

```
{
  "groups": [{ "name": "samples" }],
  "generatorScript": {
    "commands": [
      { "type": "manual", "manualFile": "./test.txt", "group": "main" } // ❌ "main" not defined
    ]
  }
}
```

3. **Don't create too many tests**
   - Be mindful of range sizes
   - `[1, 10000]` creates 10,000 tests!
   - Consider if you really need that many
4. **Don't mix group modes**

```
// ❌ Bad: groupsEnabled: true but no groups in commands
{
  "groupsEnabled": true,
  "groups": [{ "name": "main" }],
  "generatorScript": {
    "commands": [
      { "type": "manual", "manualFile": "./test.txt" } // ❌ Missing "group" field
    ]
  }
}
```

5. **Don't use missing manual files**
   - Create manual test files before running generate
   - Use correct relative paths
   - Verify files exist

⚠️ **Common Mistakes:**

```
// ❌ Wrong: Generator not defined
{
  "generators": [
    {"name": "gen-random", "source": "./generators/random.cpp"}
  ],
  "testsets": [{
```

```json
    "generatorScript": {
      "commands": [
        {"type": "generator-single", "generator": "gen-wrong", "number": 10}  // ❌
"gen-wrong" doesn't exist
      ]
    }
  }]
}

// ✅ Correct: Use defined generator
{
  "generators": [
    {"name": "gen-random", "source": "./generators/random.cpp"}
  ],
  "testsets": [{
    "generatorScript": {
      "commands": [
        {"type": "generator-single", "generator": "gen-random", "number": 10}  // ✅
Matches defined generator
      ]
    }
  }]
}
```

Testing Your Testset Configuration

After configuring testsets, verify everything works:

```bash
# 1. Generate tests
polyman generate all

# 2. Check generated files
ls testsets/tests/

# 3. Validate tests
polyman validate all

# 4. Run main solution
polyman run-solution main all

# 5. Full verification
polyman verify
```

Expected Output Structure:

```
testsets/
└── tests/              # Testset name
    ├── test1.txt       # First test (manual or generated)
    ├── test2.txt       # Second test
    ├── test3.txt       # And so on...
    └── test100.txt     # Last test
```

Summary

Command Type	Purpose	Creates	Use When
manual	Use pre-written file	1 test	Sample tests, specific edge cases
generator-single	Run generator once	1 test	Specific parameter values
generator-range	Run generator multiple times	N tests	Many similar tests with different sizes

Key Points:

- **Testsets** = Collections of tests (usually just one named "tests")
- **Groups** = Categories within testsets (optional but recommended)
- **Commands** = Instructions for creating individual tests
- **Test files** = Stored in `testsets/<testset-name>/test<N>.txt`

Typical Configuration:

```
{
  "testsets": [
    {
      "name": "tests",
      "groupsEnabled": true,
      "groups": [{ "name": "samples" }, { "name": "main" }, { "name": "edge" }],
      "generatorScript": {
        "commands": [
          // 2-3 manual samples
          {
            "type": "manual",
            "manualFile": "./tests/manual/sample1.txt",
            "group": "samples"
          },

          // 30-50 main tests
          {
            "type": "generator-range",
            "generator": "gen-random",
            "range": [10, 60],
            "group": "main"
          },

          // 5-10 edge cases
          {
            "type": "generator-single",
            "generator": "gen-edge",
            "number": 1,
            "group": "edge"
          },
          {
            "type": "generator-single",
```

```json
            "generator": "gen-edge",
            "number": 100000,
            "group": "edge"
          }
        ]
      }
    }
  ]
}
```

Total: ~55-65 tests with good coverage

✅ **Do's:**

- Always include sample tests in a separate group
- Use meaningful group names
- Order tests from simple to complex

× **Don'ts:**

- Don't create too many small testsets (combine related tests)
- Don't forget to specify groups if groupsEnabled is true
- Don't use the same test number twice

Writing Validators

Purpose

Validators ensure that test inputs conform to problem constraints.

Basic Structure

```cpp
#include "testlib.h"

int main(int argc, char* argv[]) {
    registerValidation(argc, argv);

    // Read and validate input
    int n = inf.readInt(1, 100000, "n");
    inf.readSpace();
    int m = inf.readInt(1, 100000, "m");
    inf.readEoln();

    // Read array
    for (int i = 0; i < n; i++) {
        inf.readInt(1, 1000000000, "a[i]");
        if (i < n - 1)
            inf.readSpace();
    }
    inf.readEoln();

    // Ensure end of file
```

```
    inf.readEof();

    return 0;
}
```

Common Validation Functions

Function	Description	Example
`readInt(min, max, name)`	Read integer in range	`readInt(1, 1e9, "n")`
`readLong(min, max, name)`	Read long long in range	`readLong(1LL, 1e18, "x")`
`readDouble(min, max, name)`	Read double in range	`readDouble(0, 1, "p")`
`readString(name)`	Read string (non-whitespace)	`readString("s")`
`readToken(name)`	Read token	`readToken("word")`
`readLine(name)`	Read entire line	`readLine("text")`
`readSpace()`	Expect single space	`readSpace()`
`readSpaces()`	Read one or more spaces	`readSpaces()`
`readEoln()`	Expect end of line	`readEoln()`
`readEof()`	Expect end of file	`readEof()`

Validator Self-Tests

Create `validator_tests.json`:

```json
[
  {
    "index": 1,
    "input": "5 3\n1 2 3 4 5\n",
    "expectedVerdict": "VALID"
  },
  {
    "index": 2,
    "input": "0 5\n",
    "expectedVerdict": "INVALID"
  },
  {
    "index": 3,
    "input": "5 3\n1 2 3 4 5 6\n",
    "expectedVerdict": "INVALID"
  }
]
```

✅ Do's:

1. **Validate all constraints** mentioned in the problem statement

2. **Check format exactly** (spaces, newlines, EOF)
3. **Use meaningful variable names** in validation messages
4. **End with `readEof()`** to ensure no extra data
5. **Test validator** with both valid and invalid inputs
6. **Use strict validation** for interactive problems

✕ Don'ts:

1. **Don't skip validation** of any constraint
2. **Don't allow extra whitespace** unless problem allows it
3. **Don't validate output** in validator (that's checker's job)
4. **Don't use `scanf/cin`** - always use testlib functions
5. **Don't forget** to validate relationships between variables
6. **Don't allow** trailing spaces or lines unless specified

Advanced Example

```cpp
#include "testlib.h"
#include <vector>

int main(int argc, char* argv[]) {
    registerValidation(argc, argv);

    int n = inf.readInt(1, 100000, "n");
    inf.readEoln();

    // Read and validate a tree
    std::vector<int> parent(n);
    for (int i = 1; i < n; i++) {
        parent[i] = inf.readInt(1, i, "parent[i]");
        if (i < n - 1)
            inf.readSpace();
    }
    inf.readEoln();

    // Validate no cycles (tree property)
    ensure(parent[0] == 0 || n == 1);

    inf.readEof();
    return 0;
}
```

Writing Checkers

Purpose

Checkers compare contestant output with jury answer and determine verdict.

Basic Structure

```cpp
#include "testlib.h"

int main(int argc, char* argv[]) {
    registerTestlibCmd(argc, argv);

    // Read jury answer
    int jans = ans.readInt();

    // Read contestant output
    int pans = ouf.readInt();

    // Compare
    if (jans == pans) {
        quitf(_ok, "Correct answer: %d", jans);
    } else {
        quitf(_wa, "Wrong answer: expected %d, found %d", jans, pans);
    }
}
```

Checker Streams

- `inf` : Input file (test input)
- `ans` : Answer file (jury's output)
- `ouf` : Output file (contestant's output)

Checker Verdicts

```cpp
quitf(_ok, "message");              // Accepted
quitf(_wa, "message");              // Wrong Answer
quitf(_pe, "message");              // Presentation Error
quitf(_fail, "message");            // Checker failed
```

Checker Self-Tests

Create `checker_tests.json` :

```json
[
  {
    "index": 1,
    "input": "5 3\n",
    "output": "8\n",
    "answer": "8\n",
    "expectedVerdict": "OK"
  },
  {
    "index": 2,
    "input": "5 3\n",
    "output": "7\n",
    "answer": "8\n",
    "expectedVerdict": "WRONG_ANSWER"
```

```
  },
  {
    "index": 3,
    "input": "5 3\n",
    "output": "  8  \n",
    "answer": "8\n",
    "expectedVerdict": "PRESENTATION_ERROR"
  }
]
```

✅ Do's:

1. **Use standard checkers** when possible (`wcmp` for most problems)
2. **Read from correct streams** (inf, ans, ouf)
3. **Provide helpful messages** in quitf
4. **Handle edge cases** (empty output, extra whitespace)
5. **Test checker** with various outputs
6. **Be lenient with formatting** unless problem requires strict format

✗ Don'ts:

1. **Don't use `_fail`** for contestant errors (use `_wa` or `_pe`)
2. **Don't read from wrong streams**
3. **Don't crash** on invalid output (handle gracefully)
4. **Don't compare floating-point** with `==` (use epsilon)
5. **Don't forget** to test checker self-tests

Floating-Point Checker Example

```
#include "testlib.h"

int main(int argc, char* argv[]) {
    registerTestlibCmd(argc, argv);

    double jans = ans.readDouble();
    double pans = ouf.readDouble();

    const double EPS = 1e-6;

    if (abs(jans - pans) < EPS) {
        quitf(_ok, "Correct: %.6f", pans);
    } else {
        quitf(_wa, "Wrong: expected %.6f, found %.6f", jans, pans);
    }
}
```

Multiple Answer Checker Example

```
#include "testlib.h"
#include <set>
```

```cpp
int main(int argc, char* argv[]) {
    registerTestlibCmd(argc, argv);

    int n = inf.readInt();

    // Read all possible jury answers
    std::set<int> validAnswers;
    while (!ans.seekEof()) {
        validAnswers.insert(ans.readInt());
    }

    // Read contestant answer
    int pans = ouf.readInt();

    if (validAnswers.count(pans)) {
        quitf(_ok, "One of valid answers");
    } else {
        quitf(_wa, "Invalid answer: %d", pans);
    }
}
```

Writing Generators

Purpose

Generators create test inputs programmatically.

Basic Structure

```cpp
#include "testlib.h"
#include <iostream>

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    // Read parameters
    int n = atoi(argv[1]);

    // Generate test
    std::cout << n << " " << rnd.next(1, n) << std::endl;

    for (int i = 0; i < n; i++) {
        std::cout << rnd.next(1, 1000000);
        if (i < n - 1) std::cout << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Random Functions

Function	Description	Example
`rnd.next(n)`	Random int in [0, n)	`rnd.next(10)` → 0..9
`rnd.next(l, r)`	Random int in [l, r]	`rnd.next(1, 100)`
`rnd.next(l, r)`	Random long long	`rnd.next(1LL, 1e18)`
`rnd.next(s)`	Random element from string	`rnd.next("abc")`
`rnd.wnext(n, w)`	Weighted random [0, n)	`rnd.wnext(100, 3)`

✅ Do's:

1. **Use command-line arguments** for test parameters
2. **Ensure deterministic generation** (same args → same test)
3. **Generate valid inputs** according to constraints
4. **Use meaningful parameters** (n, maxValue, etc.)
5. **Test generator outputs** with validator
6. **Document generator parameters** in comments

❌ Don'ts:

1. **Don't use `rand()`** - use testlib's `rnd`
2. **Don't generate invalid tests**
3. **Don't ignore command-line arguments**
4. **Don't exceed memory/time** during generation
5. **Don't forget edge cases** (min/max values)

Advanced Example

```cpp
#include "testlib.h"
#include <iostream>
#include <vector>
#include <algorithm>

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n = atoi(argv[1]);  // Number of nodes

    // Generate random tree
    std::cout << n << std::endl;

    for (int i = 2; i <= n; i++) {
        int parent = rnd.next(1, i - 1);
        std::cout << parent;
        if (i < n) std::cout << " ";
    }
    std::cout << std::endl;
```

```
    return 0;
}
```

Writing Solutions

Main Correct Solution (MA)

Your main solution should be:

- **Correct**: Solves all possible inputs
- **Efficient**: Runs within time/memory limits
- **Clean**: Well-commented and readable

```cpp
#include <iostream>
using namespace std;

int main() {
    int n, m;
    cin >> n >> m;

    // Your algorithm here
    cout << n + m << endl;

    return 0;
}
```

Other Solution Types

Wrong Answer (WA):

```cpp
// Intentionally wrong algorithm
int result = n * m;  // Should be n + m
cout << result << endl;
```

Time Limit (TL):

```python
# Intentionally slow algorithm
n, m = map(int, input().split())
result = 0
for i in range(n * m):  # O(n*m) when O(1) is possible
    result += 1
print(result)
```

✅ Do's:

1. **Test main solution** thoroughly
2. **Verify WA solutions** actually get WA
3. **Verify TL solutions** actually TLE
4. **Use appropriate language** for each solution type

5. **Include alternative correct solutions** if possible

✕ Don'ts:

1. **Don't have bugs** in MA solution
2. **Don't make TL solution** too slow (should TLE, not timeout forever)
3. **Don't make WA solution** accidentally correct
4. **Don't forget** to test solutions against each other

Test Generation

Generation Workflow

```
# 1. Compile generators
polyman generate tests

# 2. Validate generated tests
polyman validate all

# 3. Run main solution
polyman run-solution main all
```

Manual Tests

Create manual test files in `tests/manual/` :

```
tests/manual/
├── sample1.txt
├── sample2.txt
└── edge-case.txt
```

Reference in Config.json:

```json
{
  "type": "manual",
  "manualFile": "./tests/manual/sample1.txt",
  "group": "samples"
}
```

Generated Tests

Use generator commands:

```json
{
  "type": "generator-single",
  "generator": "gen-random",
  "number": 42,
  "group": "main"
}
```

This calls: `gen-random 42`

Test Organization

Best Practice Structure:

```
{
  "groups": [
    {
      "name": "samples" // 1-3 tests
    },
    {
      "name": "small" // Small inputs for debugging
    },
    {
      "name": "main" // Main test cases
    },
    {
      "name": "edge" // Edge cases (min/max values)
    },
    {
      "name": "stress" // Large random tests
    }
  ]
}
```

CLI Commands Reference

This section explains all available Polyman commands and what happens when you run them.

Problem Creation

`polyman new <directory>`

Creates a new problem template in the specified directory.

Usage:

```
polyman new my-problem
```

What Happens:

1. Creates the directory `my-problem/`
2. Copies template structure with:
   - `Config.json` - Pre-configured problem settings
   - `checker/` - Sample checker with tests
   - `validator/` - Sample validator with tests
   - `generators/` - Sample generator
   - `solutions/` - Example solutions (C++, Java, Python)
   - `statements/` - Statement template files
   - `tests/manual/` - Directory for manual test files

3. All files are ready to customize

After Running:

```
my-problem/
├── Config.json
├── checker/
│   ├── chk.cpp
│   └── checker_tests.json
├── validator/
│   ├── val.cpp
│   └── validator_tests.json
├── generators/
│   └── gen.cpp
├── solutions/
│   ├── acc.cpp
│   ├── acc2.java
│   └── tle.py
└── statements/
    ├── english/
    └── russian/
```

`polyman download-testlib`

Downloads the latest testlib.h from GitHub.

Usage:

```
polyman download-testlib
```

What Happens:

1. Connects to https://github.com/MikeMirzayanov/testlib
2. Downloads the latest `testlib.h` file
3. Saves it to the current directory
4. Displays installation instructions for system-wide usage

Output:

- File: `testlib.h` in current directory
- Instructions for copying to system include directory (optional)

Note: Required for compiling validators, checkers, and generators.

`polyman list-checkers`

Lists all available standard checkers from testlib.

Usage:

```
polyman list-checkers
```

What Happens:

1. Reads the internal checker database
2. Displays each checker with:
   - **Name** (e.g., `wcmp` , `ncmp`)
   - **Description** (what it checks)
   - **Use case** (when to use it)

Example Output:

```
Available Standard Checkers:
  wcmp   - Compare tokens (whitespace-insensitive)
  ncmp   - Compare numbers with absolute/relative error
  fcmp   - Compare floating-point numbers
  lcmp   - Compare lines exactly
  ...
```

Test Management

`polyman generate all`

Generates all tests for all testsets defined in Config.json.

Usage:

```
polyman generate all
```

What Happens:

1. **Reads Config.json** and validates testset configuration
2. **For each testset:**
   - Creates directory `testsets/<testset-name>/`
   - **For each command in generatorScript:**
     - **If manual:** Copies file from `manualFile` path
     - **If generator-single:**
       - Compiles generator if needed
       - Runs `./generator <number>`
       - Saves output to `test<N>.txt`
     - **If generator-range:**
       - Compiles generator if needed
       - Runs generator for each number in range
       - Saves each output to `test<N>.txt`
3. **Reports:** Number of tests generated per testset

Example Output:

```
✓ Step 1: Validating configuration for test generation
✓ Step 2: Compiling generators for testset 'tests'
  Compiling gen-random.cpp...
✓ Step 3: Generating tests for testset 'tests'
  Generated test 1 (manual)
  Generated test 2 (manual)
  Generated test 3 from gen-random 1
```

```
Generated test 4 from gen-random 2
...
Generated 52 tests
```

`polyman generate <testset>`

Generates tests for a specific testset only.

Usage:

```
polyman generate tests
```

What Happens: Same as `generate all`, but only for the specified testset.

Use Case: When you have multiple testsets and want to regenerate just one.

`polyman generate <testset> <group>`

Generates tests for a specific group within a testset.

Usage:

```
polyman generate tests samples
```

What Happens:

1. Reads Config.json
2. Filters commands to only those with `"group": "samples"`
3. Generates only those tests
4. Skips tests from other groups

Use Case: Quickly regenerate just sample tests or a specific category.

`polyman generate <testset> <index>`

Generates a specific test by number.

Usage:

```
polyman generate tests 5
```

What Happens:

1. Reads Config.json
2. Finds the 5th command in the generator script
3. Executes only that command
4. Generates only `test5.txt`

Use Case: Regenerate a single test after fixing a generator bug.

Validation

`polyman validate all`

Validates all generated tests using the validator.

Usage:

```
polyman validate all
```

What Happens:

1. **Reads Config.json** and locates validator source
2. **Compiles validator** (e.g., `val.cpp`)
   - Uses C++ compiler (g++, clang, or MSVC)
   - Creates executable `val` or `val.exe`
3. **For each testset:**
   - **For each test file** in `testsets/<testset-name>/`:
     - Runs `./val < test<N>.txt`
     - Captures validator verdict (VALID/INVALID)
     - If INVALID: Shows error message
4. **Reports:**
   - ✓ Valid tests (green)
   - ✗ Invalid tests (red) with error details
   - Total: X/Y tests passed

Example Output:

```
✓ Step 1: Validating configuration for validator
✓ Step 2: Compiling validator
  Compiling val.cpp...
✓ Step 3: Validating tests for testset 'tests'
  ✓ test1.txt - VALID
  ✓ test2.txt - VALID
  ✗ test3.txt - INVALID (Integer n out of range [1, 1000])
  ...
  50/52 tests valid
```

What to Do if Tests Fail:

- Check validator constraints
- Fix generator to produce valid output
- Verify manual test files

`polyman validate <testset>`

Validates tests for a specific testset.

Usage:

```
polyman validate tests
```

What Happens: Same as `validate all`, but only for specified testset.

`polyman validate <testset> <group>`

Validates tests in a specific group.

Usage:

```
polyman validate tests samples
```

What Happens:

1. Compiles validator
2. Only validates tests that belong to the "samples" group
3. Skips other groups

How It Knows Which Tests:

- Reads Config.json to determine which test numbers belong to which group
- Only runs validator on those specific tests

`polyman validate <testset> <index>`

Validates a single test.

Usage:

```
polyman validate tests 5
```

What Happens:

1. Compiles validator
2. Runs `./val < testsets/tests/test5.txt`
3. Shows verdict (VALID/INVALID) with details

Use Case: Debug a specific failing test.

Solution Execution

`polyman run-solution <name> all`

Runs a solution on all tests in all testsets.

Usage:

```
polyman run-solution main all
```

What Happens:

1. **Reads Config.json** and finds solution by name
2. **Compiles solution:**
    - C++: Uses g++/clang
    - Java: Uses javac
    - Python: No compilation
3. **For each testset:**
    - Creates output directory: `solutions-outputs/<solution-name>/<testset>/`

- **For each test:**
  - Runs solution: `./solution < test<N>.txt > output<N>.txt`
  - Measures execution time
  - Detects crashes, TLE, MLE

4. **Runs main solution** (if not already) to generate answers
5. **Compiles and runs checker:**
   - For each test: `./checker input.txt jury_answer.txt contestant_output.txt`
   - Gets verdict: OK, WA, PE, etc.
6. **Reports:**
   - Verdict for each test
   - Execution time
   - Summary statistics

Example Output:

```
✓ Step 1: Validating configuration for solutions
✓ Step 2: Compiling solutions
  Compiling main (acc.cpp)...
✓ Step 3: Running solution 'main' on testset 'tests'
  Test 1: OK (15 ms)
  Test 2: OK (18 ms)
  Test 3: OK (142 ms)
  ...
  Summary: 52/52 tests passed
  Max time: 142 ms / 1000 ms
```

`polyman run-solution <name> <testset>`

Runs solution on specific testset.

Usage:

```
polyman run-solution main tests
```

What Happens: Same as above, but only for the specified testset.

`polyman run-solution <name> <testset> <group>`

Runs solution on specific group.

Usage:

```
polyman run-solution main tests samples
```

What Happens:

1. Compiles solution
2. Runs only on tests in the "samples" group
3. Shows results for those tests only

Use Case: Quick check on sample tests before full testing.

`polyman run-solution <name> <testset> <index>`

Runs solution on single test.

Usage:

```
polyman run-solution main tests 5
```

What Happens:

1. Compiles solution
2. Runs on test 5 only
3. Shows detailed verdict and timing

Use Case: Debug specific test failure.

Testing Components

`polyman test validator`

Tests the validator against its self-tests.

Usage:

```
polyman test validator
```

What Happens:

1. **Reads** `validator/validator_tests.json`
2. **Compiles validator**
3. **For each test case:**
    - Creates temporary input file with test input
    - Runs `./val < temp_input.txt`
    - Compares actual verdict with expected verdict
    - Reports PASS or FAIL
4. **Summary:** X/Y tests passed

Example Output:

```
✓ Compiling validator
✓ Running validator self-tests
  Test 1: ✓ PASS (expected VALID, got VALID)
  Test 2: ✓ PASS (expected INVALID, got INVALID)
  Test 3: ✗ FAIL (expected INVALID, got VALID)
  ...
  2/3 tests passed
```

What to Do if Tests Fail:

- Check validator logic
- Verify expected verdicts in validator_tests.json
- Update validator code or test expectations

`polyman test checker`

Tests the checker against its self-tests.

Usage:

```
polyman test checker
```

What Happens:

1. **Reads** `checker/checker_tests.json`
2. **Compiles checker**
3. **For each test case:**
    - Creates temporary files:
        - `input.txt` (test input)
        - `answer.txt` (jury answer)
        - `output.txt` (contestant output)
    - Runs `./checker input.txt answer.txt output.txt`
    - Compares actual verdict with expected verdict
4. **Summary:** X/Y tests passed

Example Output:

```
✓ Compiling checker
✓ Running checker self-tests
  Test 1: ✓ PASS (expected OK, got OK)
  Test 2: ✓ PASS (expected WRONG_ANSWER, got WRONG_ANSWER)
  Test 3: ✓ PASS (expected PRESENTATION_ERROR, got PRESENTATION_ERROR)
  ...
  3/3 tests passed
```

`polyman test <solution-name>`

Tests a solution against the main correct solution.

Usage:

```
polyman test wa-solution
```

What Happens:

1. **Validates** solution exists in Config.json
2. **Generates all tests** (if not already generated)
3. **Runs main solution** (tag: MA) on all tests to get correct answers
4. **Runs target solution** (e.g., wa-solution) on all tests
5. **Compares outputs** using checker
6. **Verifies expected behavior:**
    - If tag is `WA` : Expects at least one Wrong Answer
    - If tag is `TL` : Expects at least one Time Limit
    - If tag is `OK` : Expects all Accepted
    - If tag is `RE` : Expects at least one Runtime Error

    - If tag is `ML`: Expects at least one Memory Limit
7. **Reports:**
    - Whether solution behaves as expected
    - Which tests failed/passed
    - If behavior doesn't match tag

Example Output:

```
✓ Step 1: Validating configuration
✓ Step 2: Generating tests (if needed)
✓ Step 3: Running main solution
✓ Step 4: Running solution 'wa-solution'
  Test 1: OK
  Test 2: OK
  Test 3: WA (Wrong answer: expected 42, got 24)
  ...
✓ Solution behaves as expected (tag: WA, got WA on test 3)
```

Use Case: Verify that WA/TL/RE solutions actually fail as expected.

Full Verification

`polyman verify`

Runs complete problem verification workflow.

Usage:

```
polyman verify
```

What Happens:

This is the **most comprehensive command**. It runs all steps in order:

1. **Step 1: Validate Configuration**
    - Checks Config.json is valid
    - Verifies all required files exist
    - Validates testset structure
2. **Step 2: Compile Generators**
    - Compiles all generators defined in Config.json
    - Reports compilation errors if any
3. **Step 3: Generate All Tests**
    - Runs `generate all` internally
    - Creates all test files for all testsets
4. **Step 4: Compile Validator**
    - Compiles validator source code
    - Reports errors if compilation fails
5. **Step 5: Test Validator**

    - Runs validator self-tests from validator_tests.json
    - Ensures validator works correctly

6. **Step 6: Validate All Tests**
    - Runs validator on all generated tests
    - Ensures all tests are valid inputs

7. **Step 7: Compile Checker**
    - Compiles checker source code
    - Reports errors if compilation fails

8. **Step 8: Test Checker**
    - Runs checker self-tests from checker_tests.json
    - Ensures checker works correctly

9. **Step 9: Compile All Solutions**
    - Compiles every solution in Config.json
    - Reports which solutions compiled successfully

10. **Step 10: Run All Solutions**
    - Runs each solution on all tests
    - Checks outputs with the checker
    - Verifies solutions behave according to their tags

11. **Step 11: Verify Solution Behaviors**
    - Confirms MA solution passes all tests
    - Confirms WA solutions get WA on some tests
    - Confirms TL solutions get TL on some tests
    - Confirms other solution tags match behavior

12. **Final Report:**
    - ✓ All tests valid
    - ✓ All solutions behave correctly
    - ✗ Any issues found

Example Output:

```
=== Full Problem Verification ===

✓ Step 1: Validating configuration
✓ Step 2: Compiling generators
  Compiled gen-random.cpp
✓ Step 3: Generating all tests
  Generated 52 tests for testset 'tests'
✓ Step 4: Compiling validator
✓ Step 5: Testing validator
  3/3 validator tests passed
✓ Step 6: Validating all tests
  52/52 tests valid
✓ Step 7: Compiling checker
```

```
✓ Step 8: Testing checker
  3/3 checker tests passed
✓ Step 9: Compiling all solutions
  Compiled: main (C++)
  Compiled: wa-solution (C++)
  Compiled: tle-solution (Python)
✓ Step 10: Running all solutions
  main: 52/52 passed (tag: MA) ✓
  wa-solution: 45/52 passed, 7 WA (tag: WA) ✓
  tle-solution: 12/52 passed, 40 TLE (tag: TL) ✓
✓ Step 11: Verifying solution behaviors
  All solutions behave as expected

=== Verification Complete ===
✓ Problem is ready for use!
```

When to Use:

- Before submitting to Polygon
- After making major changes
- To ensure everything works together
- As a final check before contests

What to Do if It Fails:

- Read error messages carefully
- Fix the failing step
- Run `verify` again
- Repeat until all steps pass

Command Execution Summary

Command	Compiles	Generates	Validates	Runs Solutions	Checks Behavior
`new`	-	Template	-	-	-
`download-testlib`	-	-	-	-	-
`generate all`	Generators	✓	-	-	-
`validate all`	Validator	-	✓	-	-
`run-solution`	Solution, Checker	-	-	✓	-
`test validator`	Validator	-	Self-tests	-	-
`test checker`	Checker	-	Self-tests	-	-
`test <solution>`	All	✓	-	✓	✓
`verify`	All	✓	✓	✓	✓

Tips for Efficient Workflow

Development Cycle:

```
# 1. Create problem
polyman new my-problem
cd my-problem
polyman download-testlib

# 2. Write components (validator, checker, generators, solutions)
# ... edit files ...

# 3. Test individual components
polyman test validator
polyman test checker

# 4. Generate and validate tests
polyman generate all
polyman validate all

# 5. Test main solution
polyman run-solution main all

# 6. Test WA/TL solutions
polyman test wa-solution
polyman test tle-solution

# 7. Full verification before submission
polyman verify
```

Quick Iteration:

```
# After fixing a generator
polyman generate tests small    # Regenerate just one group
polyman validate tests small     # Validate just that group
polyman run-solution main tests small  # Test just that group
```

Debugging:

```
# Test single failing test
polyman generate tests 5
polyman validate tests 5
polyman run-solution main tests 5
```

Best Practices

1. Directory Organization

✅ **Good:**

```
problem/
├── Config.json
├── checker/
│   ├── chk.cpp
│   └── checker_tests.json
├── validator/
│   ├── val.cpp
│   └── validator_tests.json
├── generators/
│   ├── gen-random.cpp
│   └── gen-special.cpp
├── solutions/
│   ├── main.cpp
│   ├── wa.cpp
│   └── tle.py
└── tests/
    └── manual/
        ├── sample1.txt
        └── sample2.txt
```

2. Configuration Management

✅ **Good:**

- Use relative paths
- Include all necessary solution types
- Group tests logically
- Document generator parameters

× **Bad:**

- Absolute paths
- Missing MA solution
- All tests in one group
- Undocumented generators

3. Test Coverage

✅ **Include:**

- Sample tests (2-3)
- Small tests for debugging
- Main test cases
- Edge cases (min/max)
- Corner cases (n=1, empty, etc.)
- Stress tests

× **Avoid:**

- Only sample tests
- No edge cases
- Duplicate tests
- Tests without purpose

4. Solution Testing

```
# Always verify your workflow
polyman verify

# This catches:
# - Invalid test inputs
# - Checker errors
# - Solution mismatches
# - TLE/WA not behaving as expected
```

5. Version Control

✅ **Commit:**

- Config.json
- All source files
- Manual tests
- Self-test configurations

× **Don't commit:**

- Compiled binaries
- Generated test files
- testlib.h (download on setup)
- solutions-outputs/

Recommended `.gitignore`:

```
# Compiled files
*.exe
*.out
*.class
__pycache__/

# Generated tests
testsets/*/test*.txt

# Solution outputs
solutions-outputs/

# System files
.DS_Store
Thumbs.db
```

Troubleshooting

Compilation Errors

Problem: Validator/Checker won't compile

```
Error: testlib.h: No such file or directory
```

Solution:

```
# Download testlib.h first
polyman download-testlib

# Or copy to system include directory
sudo cp testlib.h /usr/include/
```

Problem: Generator compilation fails

```
Error: undefined reference to registerGen
```

Solution:

- Ensure you're using `#include "testlib.h"`
- Ensure testlib.h is in the same directory
- Check that you called `registerGen(argc, argv, 1)`

Validation Errors

Problem: Validator rejects valid test

```
FAIL: expected EOLN but found space
```

Solution:

- Check exact input format in validator
- Ensure proper use of `readSpace()` and `readEoln()`
- Verify no trailing spaces in test files

Problem: Generated test fails validation

```
FAIL: Integer x out of range [1, 100000]
```

Solution:

- Check generator's output range
- Verify generator parameters are correct
- Run generator manually to see output

Checker Errors

Problem: Checker crashes on contestant output

```
CRASHED: Unexpected end of file
```

Solution:

- Handle EOF gracefully in checker
- Use `seekEof()` before reading

- Catch exceptions and return `_pe` or `_wa`

Problem: Checker gives wrong verdict

```
Expected WA but got OK
```

Solution:

- Review checker logic
- Test checker with checker_tests.json
- Verify you're reading from correct streams (ans vs ouf)

Solution Errors

Problem: Main solution gets WA

```
Solution main marked as MA but got Wrong Answer
```

Solution:

- Debug main solution algorithm
- Test against sample inputs manually
- Check for edge cases (overflow, off-by-one)
- Verify input/output format matches

Problem: WA solution passes all tests

```
Solution wa-solution marked as WA but passed all tests
```

Solution:

- Make WA solution's bug more obvious
- Add specific test cases that expose the bug
- Verify the bug is actually wrong (not an alternative correct solution)

Problem: TL solution doesn't TLE

```
Solution tle-solution marked as TL but did not timeout
```

Solution:

- Make algorithm slower (higher complexity)
- Increase test input sizes
- Check time limit isn't too generous
- Use stress tests with maximum n

Test Generation Errors

Problem: Manual test file not found

```
Error: Cannot read manual file: ./tests/manual/sample1.txt
```

Solution:

- Create the directory: `mkdir -p tests/manual`
- Verify file path in Config.json is correct
- Use relative path from Config.json location

Problem: Generator not found

```
Error: Generator gen-random not defined
```

Solution:

- Add generator to Config.json generators array
- Ensure generator name matches exactly
- Compile generator first

Memory Issues

Problem: Solution exceeds memory limit during testing

```
Memory Limit Exceeded (256 MB)
```

Solution:

- If it's ML solution: Expected behavior ✓
- If it's MA solution:
  - Optimize data structures
  - Reduce memory usage
  - Check for memory leaks
  - Increase memoryLimit if appropriate

Time Issues

Problem: Compilation takes too long

```
Timeout while compiling validator
```

Solution:

- Simplify validator code
- Remove unnecessary includes
- Use faster compilation flags
- Check for infinite template recursion

FAQ

1. Do I need to write a custom checker for my problem?

Answer: No, in most cases you can use a standard checker:

- **Token comparison** (whitespace-insensitive): Use `wcmp`
- **Number comparison**: Use `ncmp` or `fcmp`
- **Line-by-line**: Use `lcmp`
- **Yes/No**: Use `yesno`

Only write a custom checker if:

- Multiple valid answers exist
- Output requires validation beyond simple comparison
- Special precision handling needed
- Output order doesn't matter

---

2. Can I use Python for my main solution?

Answer: Yes, but with caution:

✅ **Good for:**

- Problems with generous time limits (3-5 seconds)
- I/O-light problems
- String manipulation
- Math problems

× **Not recommended for:**

- Time-critical algorithms
- Heavy I/O problems
- Large data structures
- When TLE solutions are needed (Python may TLE on main solution)

Best practice: Write main solution in C++, use Python for alternative OK solutions.

---

3. How many tests should I include?

Answer: Typical problem structure:

- **Samples**: 2-3 tests (shown in statement)
- **Small**: 5-10 tests (for debugging)
- **Main**: 20-50 tests (covers all cases)
- **Edge**: 5-10 tests (boundaries, corner cases)
- **Stress**: 10-30 tests (large random)

Total: 40-100 tests for most problems

Key principle: Quality over quantity. Each test should serve a purpose.

---

4. What's the difference between interactive and regular problems?

Answer:

Regular Problem:

- Solution reads all input at start
- Produces output once
- Uses `stdin` / `stdout`

Interactive Problem:

- Solution communicates back-and-forth with interactor
- Multiple read/write cycles
- Uses flush after each output

- Set `"interactive": true` in Config.json
- Requires custom interactor program

Example interactive I/O:

```
// Solution
cout << "query 5" << endl;  // Must flush
int response;
cin >> response;
```

Note: Polyman currently has limited interactive support. Use regular problems unless necessary.

---

5. How do I handle floating-point problems?

Answer:

In Validator:

```
double x = inf.readDouble(0.0, 1e9, "x");
```

In Checker:

```
#include "testlib.h"

int main(int argc, char* argv[]) {
    registerTestlibCmd(argc, argv);

    double jans = ans.readDouble();
    double pans = ouf.readDouble();

    const double EPS = 1e-6;  // Or use relative error

    if (abs(jans - pans) < EPS) {
        quitf(_ok, "Correct: %.9f", pans);
    } else {
        quitf(_wa, "Wrong: expected %.9f, found %.9f, diff %.9f",
              jans, pans, abs(jans - pans));
    }
}
```

Or use standard checker:

- `fcmp` with absolute error: `fcmp 1e-6`
- `rcmp` with relative error: `rcmp 1e-9`

Best practice: Specify precision clearly in problem statement.

---

6. Can I test multiple problems in the same directory?

Answer: No, each problem should have its own directory:

```
problems/
├── problem-a/
│   ├── Config.json
│   ├── checker/
│   └── ...
└── problem-b/
    ├── Config.json
    ├── checker/
    └── ...
```

Each directory is independent. Use separate `Config.json` for each problem.

---

7. How do I debug why my solution is getting WA?

Steps:

1. **Run on samples manually:**

   ```
   ./solution < tests/manual/sample1.txt
   ```

2. **Run specific test:**

   ```
   polyman run-solution main tests 5
   ```

3. **Check solution output:**

   ```
   cat solutions-outputs/main/tests/output_test5.txt
   ```

4. **Compare with checker:**

   ```
   # The checker will show detailed error message
   polyman test main
   ```

5. **Add debug output** to solution (remove before submission)

6. **Create minimal failing test** and debug

---

8. What if my validator is too strict/lenient?

Too strict:

```
FAIL: Expected EOLN but found space
```

Solution:

- Use `readEoln()` only when newline is required
- Use `readSpace()` for required spaces
- Use `readSpaces()` if multiple spaces allowed
- Check problem statement format specification

Too lenient:

```
Accepted invalid input: -5 when range is [1, 100]
```

Solution:

- Add range checks: `readInt(1, 100, "n")`
- Add constraint validation
- Test validator with invalid inputs

---

9. How do I set up continuous integration for my problems?

GitHub Actions example:

```yaml
# .github/workflows/verify.yml
name: Verify Problem

on: [push, pull_request]

jobs:
  verify:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Setup Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '16'

      - name: Install Polyman
        run: npm install -g polyman-cli

      - name: Download testlib
        run: polyman download-testlib

      - name: Full Verification
        run: polyman verify
```

This automatically tests your problem on every commit.

---

10. Can I export my problem to Codeforces Polygon?

Answer: Polyman uses Polygon-compatible format. To upload:

1. **Create problem on Polygon** (manually for now)
2. **Upload files** via Polygon interface:
   - Validator, Checker, Generators
   - Solutions
   - Tests (can be generated on Polygon too)
3. **Configure** using Polygon UI

Future: Polyman will support direct Polygon API integration for automated upload.

Additional Resources

Official Documentation

- **Testlib GitHub**: https://github.com/MikeMirzayanov/testlib
- **Codeforces Polygon**: https://polygon.codeforces.com/
- **Testlib Tutorial**: https://codeforces.com/testlib

Example Problems

Check the template directory for a complete example problem.

Community

- Ask questions on Codeforces forums
- Join problem setting communities
- Share your experiences

Appendix: Complete Example

See the included template for a fully working example problem with:

- ✓ Sample validator
- ✓ Custom checker
- ✓ Test generators
- ✓ Multiple solutions (MA, OK, TL)
- ✓ Manual and generated tests
- ✓ Self-tests for validator and checker

Study this template to understand the complete workflow!

Happy Problem Setting! 🎉

If you encounter issues not covered in this guide, please report them or consult the Polyman documentation.