CPgen

1.0.0

Generated by Doxygen 1.10.0

Chapter 1

CPgen

1.0.1 Quick Start

Seeing the awkward situation happened in the 2023 ICPC Asia Xiaan Regional Contest, The author wrote this.

CPgen stands for Competitive Programming Data Generator.It is a project aimed to be a useful generator that can safely and conveniently used in Competitive Programming(OI, ICPC, etc.).

CPgen is hoped to be a library that help problem makers to save their time on data-making.

Here is an example of a generator using CPgen:

```
#include "../generator/generator.h"

int main(int argc, char** argv) {
    registerGen(argc, argv, 1);
    int n = rnd.next(1, 1000), m = rnd.next(1, 1000);
    println(n, m);
    Array<int> arr; arr.basic_gen(m, 1, 10 * sqrt(n)).print();
}
```

This example generates an array with size of m, whose elements are integers from 1 to $10\sqrt{n}$, and print n, m and the array to the standard output.

You should note that CPgen is based on testlib. So you are required to have `testlib.h` in your working directory (or in your environment path).

To get the latest version of CPgen, download its source code via `repo`.

1.0.2 Usages

When looking through the docs of CPgen, you should note that the first place in std::vector is ignored by CPgen. That's to say, every vector used in CPgen is 1-indexed. And, `std::vector<_Tp>().size()` means `*this.size() - 1` in the implement.

Read Docs for further information.

1.0.3 FAQs

- I generate exactly the same data while I run it many times. Why is that?

It indeed testlib fault. To ensure the generator can generator exactly the same data somehow, the seed of rnd(the RNG of testlib) is calculated by the command you type when you ran it. Try to run it with different args. For example, try:

```
»> ./foo CPgen
»> ./foo Piggy424008
»> ./foo cplusplus
```

Then it's expected to generate some different data.

Chapter 2

CPgen

2.1 Quick Start

Seeing the awkward situation happened in the 2023 ICPC Asia Xiaan Regional Contest, The author wrote this.

CPgen stands for Competitive Programming Data Generator.It is a project aimed to be a useful generator that can safely and conveniently used in Competitive Programming(OI, ICPC, etc.).

CPgen is hoped to be a library that help problem makers to save their time on data-making.

Here is an example of a generator using CPgen:

```
#include "../generator/generator.h"

int main(int argc, char** argv) {
    registerGen(argc, argv, 1);
    int n = rnd.next(1, 1000), m = rnd.next(1, 1000);
    println(n, m);
    Array<int> arr; arr.basic_gen(m, 1, 10 * sqrt(n)).print();
}
```

This example generates an array with size of m, whose elements are integers from 1 to $10\sqrt{n}$, and print n, m and the array to the standard output.

You should note that CPgen is based on testlib. So you are required to have `testlib.h` in your working directory (or in your environment path).

To get the latest version of CPgen, download its source code via `repo`.

2.2 Usages

When looking through the docs of CPgen, you should note that the first place in std::vector is ignored by CPgen. That's to say, every vector used in CPgen is 1-indexed. And, `std::vector<_Tp>().size()` means `*this.size() - 1` in the implement.

Read Docs for further information.

2.3 FAQs

- I generate exactly the same data while I run it many times. Why is that?

It indeed testlib fault. To ensure the generator can generator exactly the same data somehow, the seed of rnd(the RNG of testlib) is calculated by the command you type when you ran it. Try to run it with different args. For example, try:

```
»> ./foo CPgen
»> ./foo Piggy424008
»> ./foo cplusplus
```

Then it's expected to generate some different data.

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Class Documentation

6.1 _random Class Reference

Expansion of random_t.

```
#include <generator.h>
```

Public Member Functions

- template<typename _Tp >
  std::vector< _Tp > shuffle (std::vector< _Tp > array, int l=1, int r=-1)

  Shuffle the array in-place, indexes from l to r.
- template<typename _Tp >
  _Tp get_prime (_Tp l, _Tp r)

  get a prime $p \in [l, r]$.

6.1.1 Detailed Description

Expansion of random_t.

6.1.2 Member Function Documentation

6.1.2.1 get_prime()

```
template<typename _Tp >
_Tp _random::get_prime (
            _Tp l,
            _Tp r )  [inline]
```

get a prime $p \in [l, r]$.

Parameters

array	Any std::vector<_Tp>.
l	The left bound of the section.
r	The right bound of the section.

Returns

The generated prime.

Exceptions

When	it failed to gen a prime after 5 times of iteration, it throws an error `I suspected that there's no prime from {l} to {r}.` and quit the program.

6.1.2.2 shuffle()

```
template<typename _Tp >
std::vector< _Tp > _random::shuffle (
            std::vector< _Tp > array,
            int l = 1,
            int r = -1 )  [inline]
```

Shuffle the array in-place, indexes from l to r.

Parameters

array	Any std::vector<_Tp>.
l	The left bound that should be shuffled. Default as 1.
r	The right bound that should be shuffled. Default as `array.size()`.

Returns

Return the result.

The documentation for this class was generated from the following file:

- generator.h

6.2 Array< _Tp > Class Template Reference

Class that used to generate an array.

`#include <generator.h>`

Public Types

- using **_Sequence** = std::vector<_Tp>
- using **_Self** = Array<_Tp>

Public Member Functions

- _Tp & operator[ ] (int idx)

  return the reference of the size-th element in this array.
- auto begin ()

  return the reference of the first element in this array.
- auto end ()

  return the reference of the last element in this array.
- void init (int size)

  init the whole array with the size of `size`.
- void print (char sep=' ', char end='\n')

  Output the current array.
- _Tp sum ()

  Get the sum of the elements.
- _Self basic_gen (int size, _Tp wl, _Tp wr)

  Generate an array with size `size`, while its elements are values in [wl, wr].
- void sort ()

  Sort the current array.
- void shuffle ()

  Shuffle the current array.
- void reverse ()

  Reverse the current array.
- _Self to_diffrence ()

  Turn this array into the Diffrence array of it.
- _Self binary_gen (int size)

  Generate an array with size `size`, while its elements are 0 or 1.
- _Self ascending_array (int size, _Tp wl, _Tp wr)

  Generate an array with size `size`, while its elements are not decreasing.
- _Self decending_array (int size, _Tp wl, _Tp wr)

  Generate an array with size `size`, while its elements are not increasing.
- _Self constant_sum (int size, _Tp sum, bool AcceptZero=true, bool AcceptNegative=true)

  Generate an array with size `size`, while the sum of its elements are a constant.
- _Self perturbe ()

  Perturbe the current array, keeping the sum of the elements still.
- _Self permutation (int size)

  Generate a permutation of 1 to size.
- _Self generate_function (int size, int(*GenerateFunction)(int), int begin=1)

  Generate an array with the i-th element is f(i + begin).
- _Self generate_iterate_function (int size, int(*IterateFunction)(int), int begin=1)

  Generate an array with the i-th element is f(a_{i-1}).

Public Attributes

- int **n**

  size of the array that generated.
- _Sequence **array**

  The container of the elements.

6.2.1 Detailed Description

template<typename _Tp>
class Array< _Tp >

Class that used to generate an array.

6.2.2 Member Function Documentation

6.2.2.1 ascending_array()

```
template<typename _Tp >
_Self Array< _Tp >::ascending_array (
            int size,
            _Tp wl,
            _Tp wr )  [inline]
```

Generate an array with size `size`, while its elements are not decreasing.

Parameters

size	how large this array should be.
wl	the sub of the elements.
wr	the sup of the elements.

Returns

The array itself.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.2 basic_gen()

```
template<typename _Tp >
_Self Array< _Tp >::basic_gen (
            int size,
            _Tp wl,
            _Tp wr )  [inline]
```

Generate an array with size `size`, while its elements are values in [wl, wr].

Parameters

size	how large this array should be.
wl	the sub of the elements.
wr	the sup of the elements.

Returns

The array itself.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.3 begin()

```
template<typename _Tp >
auto Array< _Tp >::begin ( )  [inline]
```

return the reference of the first element in this array.

Parameters

no	params.

Returns

The reference of the element.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.4 binary_gen()

```
template<typename _Tp >
_Self Array< _Tp >::binary_gen (
            int size )  [inline]
```

Generate an array with size `size`, while its elements are 0 or 1.

Parameters

size	how large this array should be.

Returns

The array itself.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.5 constant_sum()

```
template<typename _Tp >
_Self Array< _Tp >::constant_sum (
            int size,
            _Tp sum,
            bool AcceptZero = true,
            bool AcceptNegative = true )  [inline]
```

Generate an array with size `size`, while the sum of its elements are a constant.

Parameters

size	how large this array should be.
sum	the sum of the elements.
AcceptZero	if the array can contain zero or not.
AcceptNegative	if the array can contain negative values or not.

Returns

The array itself.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.6 decending_array()

```
template<typename _Tp >
_Self Array< _Tp >::decending_array (
            int size,
            _Tp wl,
            _Tp wr )  [inline]
```

Generate an array with size `size`, while its elements are not increasing.

Parameters

size	how large this array should be.
wl	the sub of the elements.
wr	the sup of the elements.

Returns

The array itself.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.7 end()

```
template<typename _Tp >
auto Array< _Tp >::end ( )  [inline]
```

return the reference of the last element in this array.

Parameters

no	params.

Returns

The reference of the element.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.8 generate_function()

```
template<typename _Tp >
_Self Array< _Tp >::generate_function (
            int size,
            int(*)(int) GenerateFunction,
            int begin = 1 )  [inline]
```

Generate an array with the i-th element is f(i + begin).

Parameters

size	the size of the array.
GenerateFunction	the GenerateFunction of the array.
begin	the begin point of the array.

Returns

The array itself.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.9 generate_iterate_function()

```
template<typename _Tp >
_Self Array< _Tp >::generate_iterate_function (
```

```
            int size,
            int(*)(int) IterateFunction,
            int begin = 1 )  [inline]
```

Generate an array with the i-th element is f(a_{i-1}).

Parameters

size	the size of the array.
GenerateFunction	the GenerateFunction of the array.
begin	the begin value of the array.

Returns

The array itself.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.10 init()

```
template<typename _Tp >
void Array< _Tp >::init (
            int size )  [inline]
```

init the whole array with the size of `size`.

Parameters

size	how large this array should be.

Returns

no return.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.11 operator[]()

```
template<typename _Tp >
_Tp & Array< _Tp >::operator[] (
            int idx )  [inline]
```

return the reference of the size-th element in this array.

Parameters

idx	the index of the element you requested.

Returns

The reference of the element.

Exceptions

out_of_range	if idx is an invalid index.

6.2.2.12 permutation()

```
template<typename _Tp >
_Self Array< _Tp >::permutation (
            int size )  [inline]
```

Generate a permutation of 1 to size.

Parameters

size	the size of the array.

Returns

The array itself.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.13 perturbe()

```
template<typename _Tp >
_Self Array< _Tp >::perturbe ( )  [inline]
```

Perturbe the current array, keeping the sum of the elements still.

Parameters

no	params.

Returns

The array itself.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.14 print()

```
template<typename _Tp >
void Array< _Tp >::print (
            char sep = ' ',
            char end = '\n' )  [inline]
```

Output the current array.

Parameters

no	params.

Returns

The array itself.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.15 reverse()

```
template<typename _Tp >
void Array< _Tp >::reverse ( )  [inline]
```

Reverse the current array.

Parameters

no	params.

Returns

no return.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.16 shuffle()

```
template<typename _Tp >
void Array< _Tp >::shuffle ( )  [inline]
```

Shuffle the current array.

Parameters

no	params.

Returns

no return.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.17 sort()

```
template<typename _Tp >
void Array< _Tp >::sort ( )  [inline]
```

Sort the current array.

Parameters

no	params.

Returns

no return.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.18 sum()

```
template<typename _Tp >
_Tp Array< _Tp >::sum ( )  [inline]
```

Get the sum of the elements.

Parameters

no	params.

Returns

The sum of the elements.

Exceptions

lt	throws what the _Sequence throws.

6.2.2.19 to_diffrence()

```
template<typename _Tp >
_Self Array< _Tp >::to_diffrence ( )  [inline]
```

Turn this array into the Diffrence array of it.

Parameters

no	params.

Returns

The array itself.

Exceptions

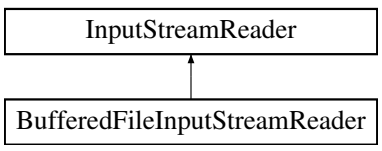
lt	throws what the _Sequence throws.

The documentation for this class was generated from the following file:

- generator.h

6.3 BufferedFileInputStreamReader Class Reference

Inheritance diagram for BufferedFileInputStreamReader:



Public Member Functions

- **BufferedFileInputStreamReader** (std::FILE *file, const std::string &name)
- void setTestCase (int)
- std::vector< int > getReadChars ()
- int curChar ()
- int nextChar ()
- void skipChar ()
- void unreadChar (int c)
- std::string getName ()
- int getLine ()
- bool eof ()
- void close ()

6.3.1 Member Function Documentation

6.3.1.1 close()

```
void BufferedFileInputStreamReader::close ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.3.1.2 curChar()

```
int BufferedFileInputStreamReader::curChar ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.3.1.3 eof()

```
bool BufferedFileInputStreamReader::eof ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.3.1.4 getLine()

```
int BufferedFileInputStreamReader::getLine ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.3.1.5 getName()

```
std::string BufferedFileInputStreamReader::getName ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.3.1.6 getReadChars()

```
std::vector< int > BufferedFileInputStreamReader::getReadChars ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.3.1.7 nextChar()

```
int BufferedFileInputStreamReader::nextChar ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.3.1.8 setTestCase()

```
void BufferedFileInputStreamReader::setTestCase (
            int  )  [inline], [virtual]
```

Implements InputStreamReader.

6.3.1.9 skipChar()

```
void BufferedFileInputStreamReader::skipChar ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.3.1.10 unreadChar()

```
void BufferedFileInputStreamReader::unreadChar (
            int c )  [inline], [virtual]
```

Implements InputStreamReader.

The documentation for this class was generated from the following file:

- testlib.h

6.4 Checker Class Reference

Public Member Functions

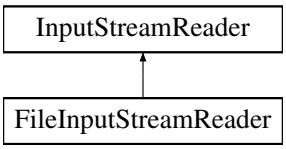
- void **initialize** ()
- std::string **testset** () const
- std::string **group** () const
- void **setTestset** (const char *const testset)
- void **setGroup** (const char *const group)

The documentation for this class was generated from the following file:

- testlib.h

6.5 FileInputStreamReader Class Reference

Inheritance diagram for FileInputStreamReader:



Public Member Functions

- **FileInputStreamReader** (std::FILE *file, const std::string &name)
- void setTestCase (int testCase)
- std::vector< int > getReadChars ()
- int curChar ()
- int nextChar ()
- void skipChar ()
- void unreadChar (int c)
- std::string getName ()
- int getLine ()
- bool eof ()
- void close ()

6.5.1 Member Function Documentation

6.5.1.1 close()

```
void FileInputStreamReader::close ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.5.1.2 curChar()

```
int FileInputStreamReader::curChar ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.5.1.3 eof()

```
bool FileInputStreamReader::eof ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.5.1.4 getLine()

```
int FileInputStreamReader::getLine ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.5.1.5 getName()

```
std::string FileInputStreamReader::getName ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.5.1.6 getReadChars()

```
std::vector< int > FileInputStreamReader::getReadChars ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.5.1.7 nextChar()

```
int FileInputStreamReader::nextChar ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.5.1.8 setTestCase()

```
void FileInputStreamReader::setTestCase (
            int testCase )  [inline], [virtual]
```

Implements InputStreamReader.

6.5.1.9 skipChar()

```
void FileInputStreamReader::skipChar ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.5.1.10 unreadChar()

```
void FileInputStreamReader::unreadChar (
            int c )  [inline], [virtual]
```

Implements InputStreamReader.

The documentation for this class was generated from the following file:

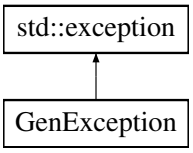
- testlib.h

6.6 GenException Struct Reference

The exception throwed when errors occured.

`#include <generator.h>`

Inheritance diagram for GenException:



Public Member Functions

- **GenException** (std::string msg)
- **GenException** (const char *msg)
- const char * **what** () const throw ()

Public Attributes

- std::string **_msg**

6.6.1 Detailed Description

The exception throwed when errors occured.

The documentation for this struct was generated from the following file:

- generator.h

6.7 Geometry< PointType > Class Template Reference

Public Types

- using **_Tp** = Point<PointType>

Public Member Functions

- void **init** ()
- void **randomize_points** (int size, _Tp leftbottom, _Tp rightup)
- void **make_raw_convex_shell** (int size)

Public Attributes

- int **n**
- std::set< _Tp > **points**

The documentation for this class was generated from the following file:

- generator.h

6.8 Graph Class Reference

Public Types

- using **_Self** = Graph

Public Member Functions

- **Graph** (Tree tr, bool direction=0)
- _Self add (Graph rhs)

  Add a graph to the current graph.
- Graph **operator+** (Graph rhs)
- _Self **operator+=** (Graph rhs)
- void init (int size, bool directed_graph)

  init the whole graph with the size of `size`*, and direct* `directed_graph`*.*
- bool exists (int u, int v)

  To check out if the edge exists or not.
- _Self randomly_gen (int size, int edges_count, bool directed_graph=false)

  Generate a graph completely random.
- _Self DAG (int size, int edges_count, bool ensure_connected=true)

  Generate a DAG.
- _Self forest (int size, int cnt=-1)

  Generate a forest.
- void hack_spfa (int size, int edges_count)

  Generate a graph, on which spfa works so slow.

Public Attributes

- int **n**
- int **m**
- bool **directed**
- std::set< pii > **edges**

6.8.1 Member Function Documentation

6.8.1.1 add()

```
_Self Graph::add (
            Graph rhs )  [inline]
```

Add a graph to the current graph.

Parameters

rhs	the graph to be added.

Returns

The graph itself.

Exceptions

lt	throws what the _Sequence throws.

6.8.1.2 DAG()

```
_Self Graph::DAG (
            int size,
            int edges_count,
            bool ensure_connected = true )  [inline]
```

Generate a DAG.

Parameters

size	how large this graph should be.
edges_count	the count of the edges.
directed_graph	is this graph directed or not.

Returns

no return.

Exceptions

lt	throws what the std::set throws.

6.8.1.3 exists()

```
bool Graph::exists (
            int u,
            int v )  [inline]
```

To check out if the edge exists or not.

Parameters

u,v	the point number of the edge that is being checked.

Returns

if the edge exists or not.

Exceptions

lt	throws what the std::set throws.

6.8.1.4 forest()

```
_Self Graph::forest (
            int size,
            int cnt = -1 )  [inline]
```

Generate a forest.

Parameters

size	how large this graph should be.
cnt	the count of the trees.

Returns

no return.

Exceptions

lt	throws what the std::set throws.

6.8.1.5 hack_spfa()

```
void Graph::hack_spfa (
            int size,
            int edges_count )  [inline]
```

Generate a graph, on which spfa works so slow.

Parameters

size	how large this graph should be.
size	how large this graph should be.
edges_count	the count of the edges.

Returns

no return.

Exceptions

lt	throws what the std::set throws.

6.8.1.6 init()

```
void Graph::init (
            int size,
            bool directed_graph )  [inline]
```

init the whole graph with the size of `size`, and direct `directed_graph`.

Parameters

size	how large this graph should be.
directed_graph	is this graph directed or not.

Returns

no return.

Exceptions

lt	throws what the std::set throws.

6.8.1.7 randomly_gen()

```
_Self Graph::randomly_gen (
            int size,
            int edges_count,
            bool directed_graph = false )  [inline]
```

Generate a graph completely random.

Parameters

size	how large this graph should be.
edges_count	the count of the edges.
directed_graph	is this graph directed or not.

Returns

no return.

Exceptions

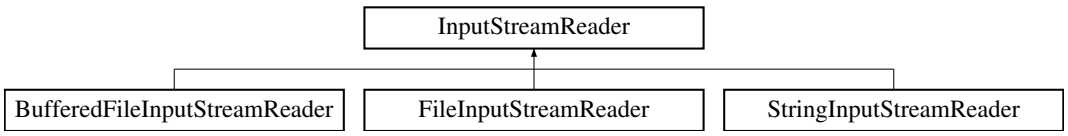
lt	throws what the std::set throws.

The documentation for this class was generated from the following file:

- generator.h

6.9 InputStreamReader Class Reference

Inheritance diagram for InputStreamReader:



Public Member Functions

- virtual void **setTestCase** (int testCase)=0
- virtual std::vector< int > **getReadChars** ()=0
- virtual int **curChar** ()=0
- virtual int **nextChar** ()=0
- virtual void **skipChar** ()=0
- virtual void **unreadChar** (int c)=0
- virtual std::string **getName** ()=0
- virtual bool **eof** ()=0
- virtual void **close** ()=0
- virtual int **getLine** ()=0

The documentation for this class was generated from the following file:

- testlib.h

6.10 InStream Struct Reference

Public Member Functions

- **InStream** (const InStream &baseStream, std::string content)
- void **init** (std::string fileName, TMode mode)
- void **init** (std::FILE *f, TMode mode)
- void **setTestCase** (int testCase)
- std::vector< int > **getReadChars** ()
- void **skipBlanks** ()
- char **curChar** ()
- void **skipChar** ()
- char **nextChar** ()
- char **readChar** ()
- char **readChar** (char c)
- char **readSpace** ()
- void **unreadChar** (char c)

- void **reset** (std::FILE *file=NULL)
- bool **eof** ()
- bool **seekEof** ()
- bool **eoln** ()
- bool **seekEoln** ()
- void **nextLine** ()
- std::string **readWord** ()
- std::string **readToken** ()
- std::string **readWord** (const std::string &ptrn, const std::string &variableName="")
- std::string **readWord** (const pattern &p, const std::string &variableName="")
- std::vector< std::string > **readWords** (int size, const std::string &ptrn, const std::string &variablesName="", int indexBase=1)
- std::vector< std::string > **readWords** (int size, const pattern &p, const std::string &variablesName="", int indexBase=1)
- std::vector< std::string > **readWords** (int size, int indexBase=1)
- std::string **readToken** (const std::string &ptrn, const std::string &variableName="")
- std::string **readToken** (const pattern &p, const std::string &variableName="")
- std::vector< std::string > **readTokens** (int size, const std::string &ptrn, const std::string &variablesName="", int indexBase=1)
- std::vector< std::string > **readTokens** (int size, const pattern &p, const std::string &variablesName="", int indexBase=1)
- std::vector< std::string > **readTokens** (int size, int indexBase=1)
- void **readWordTo** (std::string &result)
- void **readWordTo** (std::string &result, const pattern &p, const std::string &variableName="")
- void **readWordTo** (std::string &result, const std::string &ptrn, const std::string &variableName="")
- void **readTokenTo** (std::string &result)
- void **readTokenTo** (std::string &result, const pattern &p, const std::string &variableName="")
- void **readTokenTo** (std::string &result, const std::string &ptrn, const std::string &variableName="")
- long long **readLong** ()
- unsigned long long **readUnsignedLong** ()
- int **readInteger** ()
- int **readInt** ()
- long long **readLong** (long long minv, long long maxv, const std::string &variableName="")
- std::vector< long long > **readLongs** (int size, long long minv, long long maxv, const std::string &variables↩ Name="", int indexBase=1)
- std::vector< long long > **readLongs** (int size, int indexBase=1)
- unsigned long long **readUnsignedLong** (unsigned long long minv, unsigned long long maxv, const std::string &variableName="")
- std::vector< unsigned long long > **readUnsignedLongs** (int size, unsigned long long minv, unsigned long long maxv, const std::string &variablesName="", int indexBase=1)
- std::vector< unsigned long long > **readUnsignedLongs** (int size, int indexBase=1)
- unsigned long long **readLong** (unsigned long long minv, unsigned long long maxv, const std::string &variableName="")
- std::vector< unsigned long long > **readLongs** (int size, unsigned long long minv, unsigned long long maxv, const std::string &variablesName="", int indexBase=1)
- int **readInteger** (int minv, int maxv, const std::string &variableName="")
- int **readInt** (int minv, int maxv, const std::string &variableName="")
- std::vector< int > **readIntegers** (int size, int minv, int maxv, const std::string &variablesName="", int index↩ Base=1)
- std::vector< int > **readIntegers** (int size, int indexBase=1)
- std::vector< int > **readInts** (int size, int minv, int maxv, const std::string &variablesName="", int indexBase=1)
- std::vector< int > **readInts** (int size, int indexBase=1)
- double **readReal** ()
- double **readDouble** ()
- double **readReal** (double minv, double maxv, const std::string &variableName="")

- std::vector< double > **readReals** (int size, double minv, double maxv, const std::string &variablesName="", int indexBase=1)
- std::vector< double > **readReals** (int size, int indexBase=1)
- double **readDouble** (double minv, double maxv, const std::string &variableName="")
- std::vector< double > **readDoubles** (int size, double minv, double maxv, const std::string &variablesName="", int indexBase=1)
- std::vector< double > **readDoubles** (int size, int indexBase=1)
- double **readStrictReal** (double minv, double maxv, int minAfterPointDigitCount, int maxAfterPointDigitCount, const std::string &variableName="")
- std::vector< double > **readStrictReals** (int size, double minv, double maxv, int minAfterPointDigitCount, int maxAfterPointDigitCount, const std::string &variablesName="", int indexBase=1)
- double **readStrictDouble** (double minv, double maxv, int minAfterPointDigitCount, int maxAfterPointDigit↩ Count, const std::string &variableName="")
- std::vector< double > **readStrictDoubles** (int size, double minv, double maxv, int minAfterPointDigitCount, int maxAfterPointDigitCount, const std::string &variablesName="", int indexBase=1)
- std::string **readString** ()
- std::vector< std::string > **readStrings** (int size, int indexBase=1)
- void **readStringTo** (std::string &result)
- std::string **readString** (const pattern &p, const std::string &variableName="")
- std::string **readString** (const std::string &ptrn, const std::string &variableName="")
- std::vector< std::string > **readStrings** (int size, const pattern &p, const std::string &variableName="", int indexBase=1)
- std::vector< std::string > **readStrings** (int size, const std::string &ptrn, const std::string &variableName="", int indexBase=1)
- void **readStringTo** (std::string &result, const pattern &p, const std::string &variableName="")
- void **readStringTo** (std::string &result, const std::string &ptrn, const std::string &variableName="")
- std::string **readLine** ()
- std::vector< std::string > **readLines** (int size, int indexBase=1)
- void **readLineTo** (std::string &result)
- std::string **readLine** (const pattern &p, const std::string &variableName="")
- std::string **readLine** (const std::string &ptrn, const std::string &variableName="")
- std::vector< std::string > **readLines** (int size, const pattern &p, const std::string &variableName="", int indexBase=1)
- std::vector< std::string > **readLines** (int size, const std::string &ptrn, const std::string &variableName="", int indexBase=1)
- void **readLineTo** (std::string &result, const pattern &p, const std::string &variableName="")
- void **readLineTo** (std::string &result, const std::string &ptrn, const std::string &variableName="")
- void **readEoln** ()
- void **readEof** ()
- NORETURN void **quit** (TResult result, const char *msg)
- NORETURN void **quitf** (TResult result, const char *msg,...)
- void **quitif** (bool condition, TResult result, const char *msg,...)
- NORETURN void **quits** (TResult result, std::string msg)
- void **ensuref** (bool cond, const char *format,...)
- void **__testlib_ensure** (bool cond, std::string message)
- void **close** ()
- void **xmlSafeWrite** (std::FILE *file, const char *msg)
- void **skipBom** ()

Static Public Member Functions

- static void **textColor** (WORD color)
- static void **quitscr** (WORD color, const char *msg)
- static void **quitscrS** (WORD color, std::string msg)

Public Attributes

- InputStreamReader * **reader**
- int **lastLine**
- std::string **name**
- TMode **mode**
- bool **opened**
- bool **stdfile**
- bool **strict**
- int **wordReserveSize**
- std::string **_tmpReadToken**
- int **readManyIteration**
- size_t **maxFileSize**
- size_t **maxTokenLength**
- size_t **maxMessageLength**

Static Public Attributes

- static const int **NO_INDEX** = INT_MAX
- static const char **OPEN_BRACKET** = char(11)
- static const char **CLOSE_BRACKET** = char(17)
- static const WORD **LightGray** = 0x07
- static const WORD **LightRed** = 0x0c
- static const WORD **LightCyan** = 0x0b
- static const WORD **LightGreen** = 0x0a
- static const WORD **LightYellow** = 0x0e

The documentation for this struct was generated from the following file:

- testlib.h

6.11 pattern Class Reference

Public Member Functions

- **pattern** (std::string s)
- std::string **next** (random_t &rnd) const
- bool **matches** (const std::string &s) const
- std::string **src** () const

The documentation for this class was generated from the following file:

- testlib.h

6.12 Point< PointType > Struct Template Reference

Public Member Functions

- bool **operator==** (const Point &rhs) const

Public Attributes

- PointType **x**
- PointType **y**

The documentation for this struct was generated from the following file:

- generator.h

6.13 random_t Class Reference

Public Member Functions

- void **setSeed** (int argc, char ∗argv[ ])
- void **setSeed** (long long _seed)
- std::string **next** (const std::string &ptrn)
- int **next** (int n)
- unsigned int **next** (unsigned int n)
- long long **next** (long long n)
- unsigned long long **next** (unsigned long long n)
- long **next** (long n)
- unsigned long **next** (unsigned long n)
- int **next** (int from, int to)
- unsigned int **next** (unsigned int from, unsigned int to)
- long long **next** (long long from, long long to)
- unsigned long long **next** (unsigned long long from, unsigned long long to)
- long **next** (long from, long to)
- unsigned long **next** (unsigned long from, unsigned long to)
- double **next** ()
- double **next** (double n)
- double **next** (double from, double to)
- template<typename Container >
  Container::value_type **any** (const Container &c)
- template<typename Iter >
  Iter::value_type **any** (const Iter &begin, const Iter &end)
- std::string **next** (const char ∗format,...)
- int **wnext** (int n, int type)
- long long **wnext** (long long n, int type)
- double **wnext** (double n, int type)
- double **wnext** (int type)
- unsigned int **wnext** (unsigned int n, int type)
- unsigned long long **wnext** (unsigned long long n, int type)
- long **wnext** (long n, int type)
- unsigned long **wnext** (unsigned long n, int type)
- int **wnext** (int from, int to, int type)
- int **wnext** (unsigned int from, unsigned int to, int type)
- long long **wnext** (long long from, long long to, int type)
- unsigned long long **wnext** (unsigned long long from, unsigned long long to, int type)
- long **wnext** (long from, long to, int type)
- unsigned long **wnext** (unsigned long from, unsigned long to, int type)
- double **wnext** (double from, double to, int type)

- template<typename Container >
  Container::value_type **wany** (const Container &c, int type)
- template<typename Iter >
  Iter::value_type **wany** (const Iter &begin, const Iter &end, int type)
- template<typename _Tp , typename E >
  std::vector< E > **perm** (_Tp size, E first)
- template<typename _Tp >
  std::vector< _Tp > **perm** (_Tp size)
- template<typename _Tp >
  std::vector< _Tp > **distinct** (int size, _Tp from, _Tp to)
- template<typename _Tp >
  std::vector< _Tp > **distinct** (int size, _Tp upper)
- template<typename _Tp >
  std::vector< _Tp > **partition** (int size, _Tp sum, _Tp min_part)
- template<typename _Tp >
  std::vector< _Tp > **partition** (int size, _Tp sum)

Static Public Attributes

- static int **version** = -1

The documentation for this class was generated from the following file:

- testlib.h

6.14 String Class Reference

Public Member Functions

- String **operator+** (String s)
- String **operator+=** (String s)
- void **print** ()
- char & **operator[ ]** (int idx)
- template<typename... Args>
  std::string **gen** (const char *pattern, Args... t)
- std::string **lower** (int size)
- std::string **latin** (int size)
- std::string **latin_number** (int size)
- std::string **numbers_only** (int size, bool leading_zero=false)
- std::string **repeat** (int size)
- std::string **gen_multi** (std::string(*func)(int), int(*size)(), int times, std::string sep=" ")
- std::string **random_insert** (int size, char rep)

Public Attributes

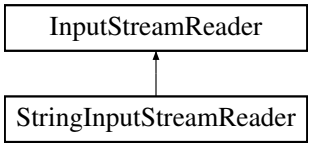
- std::string **str**

The documentation for this class was generated from the following file:

- generator.h

6.15 StringInputStreamReader Class Reference

Inheritance diagram for StringInputStreamReader:



Public Member Functions

- **StringInputStreamReader** (const std::string &content)
- void setTestCase (int)
- std::vector< int > getReadChars ()
- int curChar ()
- int nextChar ()
- void skipChar ()
- void unreadChar (int c)
- std::string getName ()
- int getLine ()
- bool eof ()
- void close ()

6.15.1 Member Function Documentation

6.15.1.1 close()

```
void StringInputStreamReader::close ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.15.1.2 curChar()

```
int StringInputStreamReader::curChar ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.15.1.3 eof()

```
bool StringInputStreamReader::eof ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.15.1.4 getLine()

```
int StringInputStreamReader::getLine ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.15.1.5 getName()

```
std::string StringInputStreamReader::getName ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.15.1.6 getReadChars()

```
std::vector< int > StringInputStreamReader::getReadChars ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.15.1.7 nextChar()

```
int StringInputStreamReader::nextChar ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.15.1.8 setTestCase()

```
void StringInputStreamReader::setTestCase (
            int  )  [inline], [virtual]
```

Implements InputStreamReader.

6.15.1.9 skipChar()

```
void StringInputStreamReader::skipChar ( )  [inline], [virtual]
```

Implements InputStreamReader.

6.15.1.10 unreadChar()

```
void StringInputStreamReader::unreadChar (
            int c )  [inline], [virtual]
```

Implements InputStreamReader.

The documentation for this class was generated from the following file:

- testlib.h

6.16 TestlibFinalizeGuard Struct Reference

Public Attributes

- int **quitCount**
- int **readEofCount**

Static Public Attributes

- static bool **alive** = true
- static bool **registered** = false

The documentation for this struct was generated from the following file:

- testlib.h

6.17 Tree Class Reference

Class that used to generate a tree.

`#include <generator.h>`

Public Types

- using **_Self** = Tree

Public Member Functions

- void init (int size)

  Initiate Tree object with size `size`.
- _Self sqrt_height_tree (int size)

  Generate a tree with an expected height of $O(\sqrt{n})$.
- _Self log_height_tree (int size)

  Generate a tree with an expected height of $O(\log n)$.
- _Self chain (int size)

  Generate a tree that is a chain.
- _Self flower (int size)

  Generate a tree that is a flower.
- _Self n_deg_tree (int size)

  Generate a tree with an expected max_deg of $O(n)$.
- _Self chain_and_flower (int size, double chain_percent=0.3, double flower_percent=0.3)

  Generate a tree with chain size is about chain_percent $*$ *size and flower size is about flower_percent* $*$ *size.*
- _Self random_shaped_tree (int size)

  Generate a tree with random shape, that is to say, randomly chose from the methods above.
- _Self print (int shuffled, std::vector< int > weights=std::vector< int >{})

  Output the generated tree to stdout. NOTE that n will not be printed.
- _Self **print_fa** (char sep=' ', char end='\n')
- std::vector< int > get_leaves ()

  Get the leave nodes of the current tree.

Public Attributes

- int **n**

  The size of the tree.
- std::vector< int > **fa**

  `fa[i]` is the no. of node `i`'s father.
- std::vector< int > leaves
- bool **weighted** = false

  *Denoting if the **edges** are weighted or not.*

6.17.1 Detailed Description

Class that used to generate a tree.

6.17.2 Member Function Documentation

6.17.2.1 chain()

```
_Self Tree::chain (
            int size )  [inline]
```

Generate a tree that is a chain.

Parameters

size	The count of the nodes that will be generated.

Returns

The graph itself.

Exceptions

out_of_range	if *size* is an invalid node count, e.g. -1.

6.17.2.2 chain_and_flower()

```
_Self Tree::chain_and_flower (
            int size,
            double chain_percent = 0.3,
            double flower_percent = 0.3 )  [inline]
```

Generate a tree with chain size is about chain_percent $*$ size and flower size is about flower_percent $*$ size.

Parameters

size	The count of the nodes that will be generated
chain_percent	the percent of the chain size.
flower_percent	the percent of the flower size.

Returns

The graph itself.

Exceptions

out_of_range	if *size* is an invalid node count, e.g. -1, or chain_percent + flower_percent $>$ 1.

6.17.2.3 flower()

```
_Self Tree::flower (
            int size )  [inline]
```

Generate a tree that is a flower.

Parameters

size	The count of the nodes that will be generated.

Returns

The graph itself.

Exceptions

out_of_range	if *size* is an invalid node count, e.g. -1.

6.17.2.4 get_leaves()

```
std::vector< int > Tree::get_leaves ( )  [inline]
```

Get the leave nodes of the current tree.

Parameters

no	params.

Returns

The leaves.

Exceptions

lt	throws what `std::vector<int>` throws.

6.17.2.5 init()

```
void Tree::init (
            int size )  [inline]
```

Initiate Tree object with size `size`.

Parameters

size	The count of the nodes that will be generated.

Returns

The graph itself.

Exceptions

out_of_range	if *size* is an invalid node count, e.g. -1.

6.17.2.6 log_height_tree()

```
_Self Tree::log_height_tree (
            int size )  [inline]
```

Generate a tree with an expected height of $O(\log n)$.

Parameters

size	The count of the nodes that will be generated.

Returns

The graph itself.

Exceptions

out_of_range	if *size* is an invalid node count, e.g. -1.

6.17.2.7 n_deg_tree()

```
_Self Tree::n_deg_tree (
            int size )  [inline]
```

Generate a tree with an expected max_deg of $O(n)$.

Parameters

size	The count of the nodes that will be generated.

Returns

The graph itself.

Exceptions

out_of_range	if *size* is an invalid node count, e.g. -1.

6.17.2.8 print()

```
_Self Tree::print (
            int shuffled,
            std::vector< int > weights = std::vector<int>{} )  [inline]
```

Output the generated tree to stdout. NOTE that n will not be printed.

Parameters

weights	the weights of the edges. Input weights[i] as the weight of the edge [fa[i], i].
shuffled	if I should print it in random order.

Returns

The graph itself.

Exceptions

out_of_range	if *size* is an invalid node count, e.g. -1.

6.17.2.9 random_shaped_tree()

```
_Self Tree::random_shaped_tree (
            int size )  [inline]
```

Generate a tree with random shape, that is to say, randomly chose from the methods above.

Parameters

size	The count of the nodes that will be generated.

Returns

The graph itself.

Exceptions

out_of_range	if *size* is an invalid node count, e.g. -1.

6.17.2.10 sqrt_height_tree()

```
_Self Tree::sqrt_height_tree (
            int size )  [inline]
```

Generate a tree with an expected height of $O(\sqrt{n})$.

Parameters

size	The count of the nodes that will be generated.

Returns

The graph itself.

Exceptions

out_of_range	if *size* is an invalid node count, e.g. -1.

6.17.3 Member Data Documentation

6.17.3.1 leaves

```
std::vector<int> Tree::leaves
```

The no. of leaves. NOTE that it would be empty UNLESS you call `get_leaves()` method.

The documentation for this class was generated from the following file:

- generator.h

6.18 Validator Class Reference

Public Member Functions

- void **initialize** ()
- std::string **testset** () const

- std::string **group** () const
- std::string **testOverviewLogFileName** () const
- std::string **testMarkupFileName** () const
- int **testCase** () const
- std::string **testCaseFileName** () const
- void **setTestset** (const char *const testset)
- void **setGroup** (const char *const group)
- void **setTestOverviewLogFileName** (const char *const testOverviewLogFileName)
- void **setTestMarkupFileName** (const char *const testMarkupFileName)
- void **setTestCase** (int testCase)
- void **setTestCaseFileName** (const char *const testCaseFileName)
- std::string **prepVariableName** (const std::string &variableName)
- bool **ignoreMinBound** (const std::string &variableName)
- bool **ignoreMaxBound** (const std::string &variableName)
- void **addBoundsHit** (const std::string &variableName, ValidatorBoundsHit boundsHit)
- std::string **getBoundsHitLog** ()
- std::string **getFeaturesLog** ()
- void **writeTestOverviewLog** ()
- void **writeTestMarkup** ()
- void **writeTestCase** ()
- void **addFeature** (const std::string &feature)
- void **feature** (const std::string &feature)

The documentation for this class was generated from the following file:

- testlib.h

6.19 ValidatorBoundsHit Struct Reference

Public Member Functions

- **ValidatorBoundsHit** (bool minHit=false, bool maxHit=false)
- ValidatorBoundsHit **merge** (const ValidatorBoundsHit &validatorBoundsHit, bool ignoreMinBound, bool ignoreMaxBound)

Public Attributes

- bool **minHit**
- bool **maxHit**

Static Public Attributes

- static const double **EPS** = 1E-12

The documentation for this struct was generated from the following file:

- testlib.h

Chapter 7

File Documentation

7.1 generator.h

```
00001
00046 #include <cassert>
00047 #include <numeric>
00048 #include <string>
00049 #include <vector>
00050 #include "testlib.h"
00051
00052 using pii = std::pair<int, int>;
00053 using i64_ll = long long;
00054 using i128_ll = __int128_t;
00055
00056 double eps = 1e-12;
00057
00061 struct GenException : public std::exception {
00062     std::string _msg;
00063     GenException(std::string msg) { _msg = msg; }
00064     GenException(const char* msg) { _msg = msg; }
00065     const char* what() const throw() { return _msg.data(); }
00066 };
00072 template <typename... Args>
00073 inline void Quit(Args... params) {
00074     ((std::cout << params << ' '), ...);
00075     std::cout << "\n";
00076     exit(1);
00077 }
00078
00084 inline i64_ll qpow(i64_ll a, i64_ll b, i64_ll mod) {
00085     assert(b >= 0);
00086     i64_ll ans = 1;
00087     while (b) {
00088         if (b & 1)
00089             ans = 1ll * ans * a % mod;
00090         a = 1ll * a * a % mod;
00091         b >>= 1;
00092     }
00093     return ans;
00094 }
00095
00101 inline bool is_prime(i64_ll n) {
00102     if (n < 3 || n % 2 == 0)
00103         return n == 2;
00104     i64_ll u = n - 1, t = 0;
00105     while (u % 2 == 0)
00106         u /= 2, ++t;
00107     i64_ll ud[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
00108     for (i64_ll a : ud) {
00109         i128_ll v = qpow(a, u, n);
00110         if (v == 1 || v == n - 1 || v == 0)
00111             continue;
00112         for (int j = 1; j <= t; j++) {
00113             v = v * v % n;
00114             if (v == n - 1 && j != t) {
00115                 v = 1;
00116                 break;
00117             }
00118             if (v == 1)
00119                 return 0;
00120         }
```

```
00121         if (v != 1)
00122             return 0;
00123     }
00124     return 1;
00125 }
00126
00130 class _random {
00131    public:
00140     template <typename _Tp>
00141     inline std::vector<_Tp> shuffle(std::vector<_Tp> array,
00142                                     int l = 1,
00143                                     int r = -1) {
00144         if (!~r)
00145             r = array.size() - 1;
00146         for (int i = l + 1; i <= r; i++)
00147             std::swap(array.at(i), array.at(rnd.next(l, i - 1)));
00148         return array;
00149     }
00160     template <typename _Tp>
00161     inline _Tp get_prime(_Tp l, _Tp r) {
00162         int times = 5;
00163         while (times-- > 0) {
00164             _Tp base = rnd.next(l, r);
00165             while (!is_prime(base) && base <= r)
00166                 base++;
00167             if (base == r)
00168                 continue;
00169             return base;
00170         }
00171         Quit(format("I suspected that there's no prime from %lld to %lld.", l,
00172                     r));
00173     }
00174 } _rnd;
00175
00184 template <typename T>
00185 inline void print(std::vector<T> vec, char sep = ' ', char end = '\n') {
00186     for (auto&& i : vec) {
00187         std::cout « i « sep;
00188     }
00189     std::cout « end;
00190 }
00191
00192 #define warn printf
00193
00197 class Tree {
00198    public:
00199     int n;
00200     std::vector<int> fa;
00201     std::vector<int> leaves;
00203     bool weighted = false;
00204     using _Self = Tree;
00211     inline void init(int size) {
00212         fa.clear(), leaves.clear();
00213         if (size < 1)
00214             throw GenException(
00215                 format("Invalid 'n' has been passed in `init`: %d", size));
00216         n = size;
00217         fa.resize(size + 1);
00218     }
00225     inline _Self sqrt_height_tree(int size) {
00226         init(size);
00227         std::vector<int> p(size - 1);
00228         for (int i = 1; i <= size - 2; i++)
00229             p.at(i) = rnd.next(1, size);
00230         std::vector<int> d(size + 1);
00231         for (int i = 1; i <= size - 2; ++i)
00232             d.at(p.at(i))++;
00233         p.at(size - 1) = size;
00234         for (int i = 1, j = 1; i < size; ++i, ++j) {
00235             while (d.at(j))
00236                 ++j;
00237             fa.at(j) = p.at(i);
00238             while (i < n && !--d.at(p.at(i)) && p.at(i) < j)
00239                 fa.at(p.at(i)) = p.at(i + 1), ++i;
00240         }
00241         fa.at(n) = fa.at(1), fa.at(1) = 0;
00242         for (int i = 2; i <= size; i++)
00243             if (fa.at(i) == 1)
00244                 fa.at(i) = size;
00245         return *this;
00246     }
00253     inline _Self log_height_tree(int size) {
00254         init(size);
00255         for (int i = 2; i <= size; i++)
00256             fa.at(i) = rnd.next(1, i - 1);
00257         return *this;
00258     }
```

```
00265     inline _Self chain(int size) {
00266         init(size);
00267         for (int i = 2; i <= size; i++)
00268             fa.at(i) = i - 1;
00269         return *this;
00270     }
00277     inline _Self flower(int size) {
00278         init(size);
00279         for (int i = 2; i <= size; i++)
00280             fa.at(i) = 1;
00281         return *this;
00282     }
00289     inline _Self n_deg_tree(int size) {
00290         init(size);
00291         int flowers_count = rnd.next(1, 10);
00292         std::vector<int> is_flower(size + 1);
00293         for (int i = 1; i <= size; i++)
00294             is_flower.at(i) = 0;
00295         std::vector<int> nodes;
00296         for (int i = 1; i <= flowers_count; i++) {
00297             int node = rnd.next(1, size);
00298             if (is_flower.at(node) == 1) {
00299                 i--;
00300                 continue;
00301             }
00302             is_flower.at(node) = 1;
00303             nodes.push_back(node);
00304         }
00305         for (int i = 2; i <= size; i++)
00306             if (is_flower.at(i))
00307                 fa.at(i) = 1;
00308             else
00309                 fa.at(i) = rnd.any(nodes);
00310         return *this;
00311     }
00322     inline _Self chain_and_flower(int size,
00323                                   double chain_percent = 0.3,
00324                                   double flower_percent = 0.3) {
00325         ensure(chain_percent + flower_percent <= 1);
00326         init(size);
00327         int i = 2;
00328         for (; i < size * chain_percent; i++)
00329             fa.at(i) = i - 1;
00330         int tmp = i - 1;
00331         for (; i < size * (chain_percent + flower_percent); i++)
00332             fa.at(i) = tmp;
00333         for (; i <= size; i++)
00334             fa.at(i) = rnd.next(1, i - 1);
00335         return *this;
00336     }
00344     inline _Self random_shaped_tree(int size) {
00345         int idx = rnd.next(6);
00346         if (idx == 0)
00347             sqrt_height_tree(size);
00348         if (idx == 1)
00349             log_height_tree(size);
00350         if (idx == 2)
00351             chain(size);
00352         if (idx == 3)
00353             flower(size);
00354         if (idx == 4)
00355             n_deg_tree(size);
00356         if (idx == 5) {
00357             double cp = rnd.wnext(1.0, 2), fp = rnd.next(1.0 - cp);
00358             chain_and_flower(size, cp, fp);
00359         }
00360         return *this;
00361     }
00371     inline _Self print(int shuffled,
00372                        std::vector<int> weights = std::vector<int>{}) {
00373         bool output_weight = true;
00374         if (weights.size() == 0)
00375             output_weight = false;
00376         else if (int(weights.size()) != n + 1)
00377             Quit("Invalid weights.size(): ", weights.size());
00378         std::vector<int> order(n + 1);
00379         std::iota(order.begin(), order.end(), 0);
00380         if (shuffled)
00381             order = _rnd.shuffle(order);
00382         for (int i = 2; i <= n; i++)
00383             if (output_weight)
00384                 println(order.at(i), fa.at(order.at(i)), weights.at(i));
00385             else
00386                 println(order.at(i), fa.at(order.at(i)));
00387         return *this;
00388     }
00389     inline _Self print_fa(char sep = ' ', char end = '\n') {
```

```
00390         for (int i = 2; i <= n; i++)
00391             std::cout « fa[i] « sep;
00392         std::cout « end;
00393         return *this;
00394     }
00401     inline std::vector<int> get_leaves() {
00402         std::vector<int> is_leave(n + 1, 1);
00403         for (int i = 1; i <= n; i++)
00404             is_leave.at(fa.at(i)) = 0;
00405         leaves.clear();
00406         for (int i = 1; i <= n; i++)
00407             if (is_leave.at(i))
00408                 leaves.push_back(i);
00409         return leaves;
00410     }
00411 };
00412
00416 template <typename _Tp>
00417 class Array {
00418    public:
00419     using _Sequence = std::vector<_Tp>;
00420     using _Self = Array<_Tp>;
00421     int n;
00422     _Sequence array;
00429     inline _Tp& operator[](int idx) {
00430         ensure(idx <= n);
00431         return array.at(idx);
00432     }
00439     inline auto begin() { return array.begin(); }
00446     inline auto end() { return array.end(); }
00453     inline void init(int size) {
00454         if (size < 1)
00455             throw GenException(
00456                 format("Invalid 'n' has been passed in `init`: %d", size));
00457         array.clear();
00458         n = size, array.resize(n + 1);
00459     }
00466     inline void print(char sep = ' ', char end = '\n') {
00467         for (int i = 1; i <= n; i++)
00468             std::cout « array.at(i) « sep;
00469         std::cout « end;
00470     }
00477     inline _Tp sum() {
00478         _Tp current_sum = 0;
00479         for (auto i : array)
00480             current_sum += i;
00481         return current_sum;
00482     }
00492     inline _Self basic_gen(int size, _Tp wl, _Tp wr) {
00493         init(size);
00494         for (int i = 1; i <= size; i++)
00495             array.at(i) = rnd.next(wl, wr);
00496         return *this;
00497     }
00504     inline void sort() { std::sort(array.begin(), array.end()); }
00511     inline void shuffle() { _rnd.shuffle(array); }
00518     inline void reverse() { std::reverse(array.begin(), array.end()); }
00525     inline _Self to_diffrence() {
00526         for (int i = n; i >= 1; i--)
00527             array.at(i) -= array.at(i - 1);
00528         return *this;
00529     }
00537     inline _Self binary_gen(int size) { return basic_gen(size, 0, 1); }
00547     inline _Self ascending_array(int size, _Tp wl, _Tp wr) {
00548         init(size);
00549         basic_gen(size, wl, wr);
00550         (*this).sort(), (*this).shuffle();
00551         return *this;
00552     }
00562     inline _Self decending_array(int size, _Tp wl, _Tp wr) {
00563         ascending_array(size, wl, wr);
00564         reverse();
00565         return *this;
00566     }
00577     inline _Self constant_sum(int size,
00578                               _Tp sum,
00579                               bool AcceptZero = true,
00580                               bool AcceptNegative = true) {
00581         init(size);
00582         if (AcceptZero)
00583             basic_gen(size - 1, 0, sum), array.at(size) = sum;
00584         else
00585             sum > n ? basic_gen(size - 1, 0, sum - n)
00586                     : basic_gen(size - 1, size - n, 0),
00587                 array.at(size) = sum - n;
00588         (AcceptNegative ? void(nullptr) : sort()), to_diffrence();
00589         if (!AcceptZero) {
```

```
00590             for (int i = 1; i <= size; i++)
00591                 array.at(i)++;
00592         }
00593         return *this;
00594     }
00602     inline _Self perturbe() {
00603         Array<_Tp> tmp_arr;
00604         tmp_arr.constant_sum(n, 0);
00605         for (int i = 1; i <= n; i++)
00606             array.at(i) += tmp_arr.at(i);
00607         return *this;
00608     }
00615     inline _Self permutation(int size) {
00616         init(size);
00617         array = rnd.perm(size, 1);
00618         return *this;
00619     }
00628     inline _Self generate_function(int size,
00629                                    int (*GenerateFunction)(int),
00630                                    int begin = 1) {
00631         init(size);
00632         for (int i = 0; i < size; i++)
00633             array.at(i) = GenerateFunction(i + begin);
00634         return *this;
00635     }
00644     inline _Self generate_iterate_function(int size,
00645                                            int (*IterateFunction)(int),
00646                                            int begin = 1) {
00647         init(size);
00648         array.at(0) = begin;
00649         for (int i = 1; i < size; i++)
00650             array.at(i) = IterateFunction(array.at(i - 1));
00651         return *this;
00652     }
00653 };
00654
00655 class Graph {
00656    public:
00657     using _Self = Graph;
00658     int n, m;
00659     bool directed;
00660     std::set<pii> edges;
00661
00662     Graph() {}
00663     Graph(Tree tr, bool direction = 0) {
00664         n = tr.n, m = n - 1;
00665         if (direction) {
00666             for (int i = 2; i <= n; i++)
00667                 edges.insert({i, tr.fa.at(i)});
00668         } else {
00669             for (int i = 2; i <= n; i++)
00670                 edges.insert({tr.fa.at(i), i});
00671         }
00672     }
00679     inline _Self add(Graph rhs) {
00680         int offset = n;
00681         n += rhs.n, m += rhs.m;
00682         for (pii edge : rhs.edges)
00683             edges.insert({offset + edge.first, offset + edge.second});
00684         return *this;
00685     }
00686     inline Graph operator+(Graph rhs) {
00687         Graph g = *this;
00688         return g.add(rhs);
00689     }
00690     inline _Self operator+=(Graph rhs) { return add(rhs); }
00699     void init(int size, bool directed_graph) {
00700         if (size < 1)
00701             throw GenException(
00702                 format("Invalid 'n' has been passed in `init`: %d", size));
00703         n = size;
00704         edges.clear();
00705         directed = directed_graph;
00706     }
00713     inline bool exists(int u, int v) {
00714         if (edges.count({u, v}))
00715             return true;
00716         if (!directed)
00717             return edges.count({v, u});
00718     }
00727     inline _Self randomly_gen(int size,
00728                               int edges_count,
00729                               bool directed_graph = false) {
00730         m = edges_count;
00731         init(size, directed_graph);
00732         for (int i = 1; i <= edges_count; i++) {
00733             int u = rnd.next(1, size), v = rnd.next(1, size);
```

```
00734             if (!exists(u, v))
00735                 edges.insert({u, v});
00736             else
00737                 i--;
00738         }
00739         return *this;
00740     }
00749     inline _Self DAG(int size, int edges_count, bool ensure_connected = true) {
00750         m = edges_count;
00751         std::vector<int> a(size + 1);
00752         std::iota(a.begin(), a.end(), 0);
00753         _rnd.shuffle(a);
00754         if (ensure_connected) {
00755             ensure(edges_count >= size - 1);
00756             Tree tree;
00757             tree.random_shaped_tree(size);
00758             auto Sort = [](pii a) {
00759                 return a.first > a.second ? std::make_pair(a.second, a.first)
00760                                           : a;
00761             };
00762             for (int i = 2; i <= size; i++)
00763                 edges.insert(Sort({a[tree.fa.at(i)], a[i]}));
00764             edges_count -= (size - 1);
00765         }
00766         for (int i = 1; i <= edges_count; i++) {
00767             int u = rnd.next(size) + 1, v = u + rnd.next(size - u) + 1;
00768             if (u == v || exists(a[u], a[v]))
00769                 i--;
00770             else
00771                 edges.insert({a[u], a[v]});
00772         }
00773         return *this;
00774     }
00783     inline _Self forest(int size, int cnt = -1) {
00784         cnt = ~cnt ? cnt : rnd.next(1, std::min(std::max(n / 1000, 10), size));
00785         Tree tr;
00786         tr.random_shaped_tree(size);
00787         *this = tr;
00788         Array<int> arr;
00789         arr.permutation(size).shuffle();
00790         for (int i = 1; i <= cnt; i++)
00791             edges.erase({tr.fa.at(arr[i]), arr[i]});
00792         return *this;
00793     }
00802     inline void hack_spfa(int size, int edges_count) {
00803         int sz = sqrt(size);
00804         m = edges_count;
00805         for (int i = 1; i <= sz; i++) {
00806             for (int j = 1; j <= sz; j++) {
00807                 if (i != 1 && edges_count)
00808                     edges.insert({(i - 1) * sz + j, i * sz + j}), edges_count--;
00809                 if (j != 1 && edges_count)
00810                     edges.insert({i * sz + j - 1, i * sz + j}), edges_count--;
00811             }
00812         }
00813         for (int i = 1; i <= edges_count; i++) {
00814             int u = rnd.next(size) + 1, v = u + rnd.next(n - u + 1) + 1;
00815             if (u == v || exists(u, v))
00816                 i--;
00817             else
00818                 edges.insert({u, v});
00819         }
00820     }
00821 };
00822
00823 class String {
00824    public:
00825     std::string str;
00826     inline String operator+(String s) { return String{str + s.str}; }
00827     inline String operator+=(String s) { return *this = String{str + s.str}; }
00828     inline void print() { println(str); }
00829     inline char& operator[](int idx) { return str[idx - 1]; }
00830     template <typename... Args>
00831     inline std::string gen(const char* pattern, Args... t) {
00832         return str = rnd.next(format(pattern, t...));
00833     }
00834     inline std::string lower(int size) { return gen("[a-z]{%d}", size); }
00835     inline std::string latin(int size) { return gen("[a-zA-Z]{%d}", size); }
00836     inline std::string latin_number(int size) {
00837         return gen("[a-zA-Z0-9]{%d}", size);
00838     }
00839     inline std::string numbers_only(int size, bool leading_zero = false) {
00840         if (leading_zero)
00841             gen("[0-9]{%d}", size);
00842         else
00843             gen("[1-9][0-9]{%d}", size - 1);
00844         return str;
```

```
00845     }
00846     inline std::string repeat(int size) {
00847         std::string res;
00848         for (int i = 1; i <= size; i++)
00849             res += str;
00850         return str = res;
00851     }
00852     inline std::string gen_multi(std::string (*func)(int),
00853                                  int (*size)(),
00854                                  int times,
00855                                  std::string sep = " ") {
00856         std::string res;
00857         for (int i = 1; i <= times; i++)
00858             res += func(size()), i != times ? res += sep : sep;
00859         return str = res;
00860     }
00861     inline std::string random_insert(int size, char rep) {
00862         Array<int> array;
00863         array.ascending_array(size, 0, str.length() - 1);
00864         for (int i : array)
00865             str[i] = rep;
00866         return str;
00867     }
00868 };
00869
00870 template <typename PointType>
00871 struct Point {
00872     PointType x, y;
00873     bool operator==(const Point& rhs) const {
00874         return (x - rhs.x) <= 15 * eps && (y - rhs.y) <= 15 * eps;
00875     }
00876 };
00877
00878 template <typename PointType>
00879 class Geometry {
00880    public:
00881     using _Tp = Point<PointType>;
00882     int n;
00883     std::set<_Tp> points;
00884     inline void init() { points.clear(); }
00885     inline void randomize_points(int size, _Tp leftbottom, _Tp rightup) {
00886         init();
00887         for (int i = 1; i <= size; i++) {
00888             PointType x = rnd.next(leftbottom.x, rightup.x),
00889                       y = rnd.next(leftbottom.y, rightup.y);
00890             if (points.find({x, y}) != points.end())
00891                 i--;
00892             else
00893                 points.insert({x, y});
00894         }
00895     }
00896     inline void make_raw_convex_shell(int size) {
00897         if (size > 100) {
00898             if (size > 10000)
00899                 warn(
00900                     "You are trying to generate a convex shell with size %d, "
00901                     "which is a big one that its coordinate may be over "
00902                     "2^{31}-1 that occurs signed-integer-overflow.",
00903                     size);
00904             else
00905                 warn(
00906                     "You are trying to generate a convex shell with size %d, "
00907                     "which is a big one that its angle may too close to pi.",
00908                     size);
00909         }
00910     }
00911 };
```

7.2 testlib.h

```
00001 /*
00002  * It is strictly recommended to include "testlib.h" before any other include
00003  * in your code. In this case testlib overrides compiler specific "random()".
00004  *
00005  * If you can't compile your code and compiler outputs something about
00006  * ambiguous call of "random_shuffle", "rand" or "srand" it means that
00007  * you shouldn't use them. Use "shuffle", and "rnd.next()" instead of them
00008  * because these calls produce stable result for any C++ compiler. Read
00009  * sample generator sources for clarification.
00010  *
00011  * Please read the documentation for class "random_t" and use "rnd" instance in
00012  * generators. Probably, these sample calls will be useful for you:
00013  *              rnd.next(); rnd.next(100); rnd.next(1, 2);
```

```
00014  *              rnd.next(3.14); rnd.next("[a-z]{1,100}").
00015  *
00016  * Also read about wnext() to generate off-center random distribution.
00017  *
00018  * See https://github.com/MikeMirzayanov/testlib/ to get latest version or bug tracker.
00019  */
00020
00021 #ifndef _TESTLIB_H_
00022 #define _TESTLIB_H_
00023
00024 /*
00025  * Copyright (c) 2005-2023
00026  */
00027
00028 #define VERSION "0.9.41"
00029
00030 /*
00031  * Mike Mirzayanov
00032  *
00033  * This material is provided "as is", with absolutely no warranty expressed
00034  * or implied. Any use is at your own risk.
00035  *
00036  * Permission to use or copy this software for any purpose is hereby granted
00037  * without fee, provided the above notices are retained on all copies.
00038  * Permission to modify the code and to distribute modified code is granted,
00039  * provided the above notices are retained, and a notice that the code was
00040  * modified is included with the above copyright notice.
00041  *
00042  */
00043
00044 /* NOTE: This file contains testlib library for C++.
00045  *
00046  *   Check, using testlib running format:
00047  *     check.exe <Input_File> <Output_File> <Answer_File> [<Result_File> [-appes]],
00048  *   If result file is specified it will contain results.
00049  *
00050  *   Validator, using testlib running format:
00051  *     validator.exe < input.txt,
00052  *   It will return non-zero exit code and writes message to standard output.
00053  *
00054  *   Generator, using testlib running format:
00055  *     gen.exe [parameter-1] [parameter-2] [... paramerter-n]
00056  *   You can write generated test(s) into standard output or into the file(s).
00057  *
00058  *   Interactor, using testlib running format:
00059  *     interactor.exe <Input_File> <Output_File> [<Answer_File> [<Result_File> [-appes]]],
00060  *   Reads test from inf (mapped to args[1]), writes result to tout (mapped to argv[2],
00061  *   can be judged by checker later), reads program output from ouf (mapped to stdin),
00062  *   writes output to program via stdout (use cout, printf, etc).
00063  */
00064
00065 const char *latestFeatures[] = {
00066         "Use setAppesModeEncoding to change xml encoding from windows-1251 to other",
00067         "rnd.any/wany use distance/advance instead of -/+: now they support sets/multisets",
00068         "Use syntax `int t = inf.readInt(1, 3, \"~t\");` to skip the lower bound check. Tildes can be
      used on either side or both: ~t, t~, ~t~",
00069         "Supported EJUDGE support in registerTestlibCmd",
00070         "Supported '--testMarkupFileName fn' and '--testCase tc/--testCaseFileName fn' for
      validators",
00071         "Added opt defaults via opt<T>(key/index, default_val); check unused opts when using has_opt
      or default opt (turn off this check with suppressEnsureNoUnusedOpt()).",
00072         "For checker added --group and --testset command line params (like for validator), use
      checker.group() or checker.testset() to get values",
00073         "Added quitpi(points_info, message) function to return with _points exit code 7 and given
      points_info",
00074         "rnd.partition(size, sum[, min_part=1]) returns random (unsorted) partition which is a
      representation of the given `sum` as a sum of `size` positive integers (or >=min_part if specified)",
00075         "rnd.distinct(size, n) and rnd.distinct(size, from, to)",
00076         "opt<bool>(\"some_missing_key\") returns false now",
00077         "has_opt(key)",
00078         "Abort validator on validator.testset()/validator.group() if registered without using command
      line",
00079         "Print integer range violations in a human readable way like `violates the range [1, 10^9]`",
00080         "Opts supported: use them like n = opt<int>(\"n\"), in a command line you can use an
      exponential notation",
00081         "Reformatted",
00082         "Use setTestCase(i) or unsetTestCase() to support test cases (you can use it in any type of
      program: generator, interactor, validator or checker)",
00083         "Fixed issue #87: readStrictDouble accepts \"-0.00\"",
00084         "Fixed issue #83: added InStream::quitif(condition, ...)",
00085         "Fixed issue #79: fixed missed guard against repeated header include",
00086         "Fixed issue #80: fixed UB in case of huge quitf message",
00087         "Fixed issue #84: added readXs(size, indexBase = 1)",
00088         "Fixed stringstream repeated usage issue",
00089         "Fixed compilation in g++ (for std=c++03)",
00090         "Batch of println functions (support collections, iterator ranges)",
00091         "Introduced rnd.perm(size, first = 0) to generate a `first`-indexed permutation",
```

```
00092         "Allow any whitespace in readInts-like functions for non-validators",
00093         "Ignore 4+ command line arguments ifdef EJUDGE",
00094         "Speed up of vtos",
00095         "Show line number in validators in case of incorrect format",
00096         "Truncate huge checker/validator/interactor message",
00097         "Fixed issue with readTokenTo of very long tokens, now aborts with _pe/_fail depending of a
      stream type",
00098         "Introduced InStream::ensure/ensuref checking a condition, returns wa/fail depending of a
      stream type",
00099         "Fixed compilation in VS 2015+",
00100         "Introduced space-separated read functions: readWords/readTokens, multilines read functions:
      readStrings/readLines",
00101         "Introduced space-separated read functions:
      readInts/readIntegers/readLongs/readUnsignedLongs/readDoubles/readReals/readStrictDoubles/readStrictReals",
00102         "Introduced split/tokenize functions to separate string by given char",
00103         "Introduced InStream::readUnsignedLong and InStream::readLong with unsigned long long
      parameters",
00104         "Supported --testOverviewLogFileName for validator: bounds hits + features",
00105         "Fixed UB (sequence points) in random_t",
00106         "POINTS_EXIT_CODE returned back to 7 (instead of 0)",
00107         "Removed disable buffers for interactive problems, because it works unexpectedly in wine",
00108         "InStream over string: constructor of InStream from base InStream to inherit policies and
      std::string",
00109         "Added expectedButFound quit function, examples: expectedButFound(_wa, 10, 20),
      expectedButFound(_fail, ja, pa, \"[n=%d,m=%d]\", n, m)",
00110         "Fixed incorrect interval parsing in patterns",
00111         "Use registerGen(argc, argv, 1) to develop new generator, use registerGen(argc, argv, 0) to
      compile old generators (originally created for testlib under 0.8.7)",
00112         "Introduced disableFinalizeGuard() to switch off finalization checkings",
00113         "Use join() functions to format a range of items as a single string (separated by spaces or
      other separators)",
00114         "Use -DENABLE_UNEXPECTED_EOF to enable special exit code (by default, 8) in case of unexpected
      eof. It is good idea to use it in interactors",
00115         "Use -DUSE_RND_AS_BEFORE_087 to compile in compatibility mode with random behavior of versions
      before 0.8.7",
00116         "Fixed bug with nan in stringToDouble",
00117         "Fixed issue around overloads for size_t on x64",
00118         "Added attribute 'points' to the XML output in case of result=_points",
00119         "Exit codes can be customized via macros, e.g. -DPE_EXIT_CODE=14",
00120         "Introduced InStream function readWordTo/readTokenTo/readStringTo/readLineTo for faster
      reading",
00121         "Introduced global functions: format(), englishEnding(), upperCase(), lowerCase(),
      compress()",
00122         "Manual buffer in InStreams, some IO speed improvements",
00123         "Introduced quitif(bool, const char* pattern, ...) which delegates to quitf() in case of first
      argument is true",
00124         "Introduced guard against missed quitf() in checker or readEof() in validators",
00125         "Supported readStrictReal/readStrictDouble - to use in validators to check strictly float
      numbers",
00126         "Supported registerInteraction(argc, argv)",
00127         "Print checker message to the stderr instead of stdout",
00128         "Supported TResult _points to output calculated score, use quitp(...) functions",
00129         "Fixed to be compilable on Mac",
00130         "PC_BASE_EXIT_CODE=50 in case of defined TESTSYS",
00131         "Fixed issues 19-21, added __attribute__ format printf",
00132         "Some bug fixes",
00133         "ouf.readInt(1, 100) and similar calls return WA",
00134         "Modified random_t to avoid integer overflow",
00135         "Truncated checker output [patch by Stepan Gatilov]",
00136         "Renamed class random -> class random_t",
00137         "Supported name parameter for read-and-validation methods, like readInt(1, 2, \"n\")",
00138         "Fixed bug in readDouble()",
00139         "Improved ensuref(), fixed nextLine to work in case of EOF, added startTest()",
00140         "Supported \"partially correct\", example: quitf(_pc(13), \"result=%d\", result)",
00141         "Added shuffle(begin, end), use it instead of random_shuffle(begin, end)",
00142         "Added readLine(const string& ptrn), fixed the logic of readLine() in the validation mode",
00143         "Package extended with samples of generators and validators",
00144         "Written the documentation for classes and public methods in testlib.h",
00145         "Implemented random routine to support generators, use registerGen() to switch it on",
00146         "Implemented strict mode to validate tests, use registerValidation() to switch it on",
00147         "Now ncmp.cpp and wcmp.cpp are return WA if answer is suffix or prefix of the output",
00148         "Added InStream::readLong() and removed InStream::readLongint()",
00149         "Now no footer added to each report by default (use directive FOOTER to switch on)",
00150         "Now every checker has a name, use setName(const char* format, ...) to set it",
00151         "Now it is compatible with TTS (by Kittens Computing)",
00152         "Added \'ensure(condition, message = \"\")\' feature, it works like assert()",
00153         "Fixed compatibility with MS C++ 7.1",
00154         "Added footer with exit code information",
00155         "Added compatibility with EJUDGE (compile with EJUDGE directive)",
00156         "Added compatibility with Contester (compile with CONTESTER directive)"
00157 };
00158
00159 #ifdef _MSC_VER
00160 #define _CRT_SECURE_NO_DEPRECATE
00161 #define _CRT_SECURE_NO_WARNINGS
00162 #define _CRT_NO_VA_START_VALIDATION
00163 #endif
```

```
00164
00165 /* Overrides random() for Borland C++. */
00166 #define random __random_deprecated
00167 #include <stdlib.h>
00168 #include <cstdlib>
00169 #include <climits>
00170 #include <algorithm>
00171 #undef random
00172
00173 #include <cstdio>
00174 #include <cctype>
00175 #include <string>
00176 #include <vector>
00177 #include <map>
00178 #include <set>
00179 #include <cmath>
00180 #include <iterator>
00181 #include <iostream>
00182 #include <sstream>
00183 #include <fstream>
00184 #include <cstring>
00185 #include <limits>
00186 #include <stdarg.h>
00187 #include <fcntl.h>
00188 #include <functional>
00189 #include <cstdint>
00190
00191 #ifdef TESTLIB_THROW_EXIT_EXCEPTION_INSTEAD_OF_EXIT
00192 #   include <exception>
00193 #endif
00194
00195 #if (_WIN32 || __WIN32__ || __WIN32 || _WIN64 || __WIN64__ || __WIN64 || WINNT || __WINNT || __WINNT__
     || __CYGWIN__)
00196 #   if !defined(_MSC_VER) || _MSC_VER > 1400
00197 #       define NOMINMAX 1
00198 #       include <windows.h>
00199 #   else
00200 #       define WORD unsigned short
00201 #       include <unistd.h>
00202 #   endif
00203 #   include <io.h>
00204 #   define ON_WINDOWS
00205 #   if defined(_MSC_VER) && _MSC_VER > 1400
00206 #       pragma warning( disable : 4127 )
00207 #       pragma warning( disable : 4146 )
00208 #       pragma warning( disable : 4458 )
00209 #   endif
00210 #else
00211 #   define WORD unsigned short
00212 #   include <unistd.h>
00213 #endif
00214
00215 #if defined(FOR_WINDOWS) && defined(FOR_LINUX)
00216 #error Only one target system is allowed
00217 #endif
00218
00219 #ifndef LLONG_MIN
00220 #define LLONG_MIN   (-9223372036854775807LL - 1)
00221 #endif
00222
00223 #ifndef ULLONG_MAX
00224 #define ULLONG_MAX   (18446744073709551615)
00225 #endif
00226
00227 #define LF ((char)10)
00228 #define CR ((char)13)
00229 #define TAB ((char)9)
00230 #define SPACE ((char)' ')
00231 #define EOFC (255)
00232
00233 #ifndef OK_EXIT_CODE
00234 #   ifdef CONTESTER
00235 #       define OK_EXIT_CODE 0xAC
00236 #   else
00237 #       define OK_EXIT_CODE 0
00238 #   endif
00239 #endif
00240
00241 #ifndef WA_EXIT_CODE
00242 #   ifdef EJUDGE
00243 #       define WA_EXIT_CODE 5
00244 #   elif defined(CONTESTER)
00245 #       define WA_EXIT_CODE 0xAB
00246 #   else
00247 #       define WA_EXIT_CODE 1
00248 #   endif
00249 #endif
```

```
00250
00251 #ifndef PE_EXIT_CODE
00252 #   ifdef EJUDGE
00253 #       define PE_EXIT_CODE 4
00254 #   elif defined(CONTESTER)
00255 #       define PE_EXIT_CODE 0xAA
00256 #   else
00257 #       define PE_EXIT_CODE 2
00258 #   endif
00259 #endif
00260
00261 #ifndef FAIL_EXIT_CODE
00262 #   ifdef EJUDGE
00263 #       define FAIL_EXIT_CODE 6
00264 #   elif defined(CONTESTER)
00265 #       define FAIL_EXIT_CODE 0xA3
00266 #   else
00267 #       define FAIL_EXIT_CODE 3
00268 #   endif
00269 #endif
00270
00271 #ifndef DIRT_EXIT_CODE
00272 #   ifdef EJUDGE
00273 #       define DIRT_EXIT_CODE 6
00274 #   else
00275 #       define DIRT_EXIT_CODE 4
00276 #   endif
00277 #endif
00278
00279 #ifndef POINTS_EXIT_CODE
00280 #   define POINTS_EXIT_CODE 7
00281 #endif
00282
00283 #ifndef UNEXPECTED_EOF_EXIT_CODE
00284 #   define UNEXPECTED_EOF_EXIT_CODE 8
00285 #endif
00286
00287 #ifndef PC_BASE_EXIT_CODE
00288 #   ifdef TESTSYS
00289 #       define PC_BASE_EXIT_CODE 50
00290 #   else
00291 #       define PC_BASE_EXIT_CODE 0
00292 #   endif
00293 #endif
00294
00295 #ifdef __GNUC__
00296 #    define __TESTLIB_STATIC_ASSERT(condition) typedef void* __testlib_static_assert_type[(condition)
      ? 1 : -1] __attribute__((unused))
00297 #else
00298 #    define __TESTLIB_STATIC_ASSERT(condition) typedef void* __testlib_static_assert_type[(condition)
      ? 1 : -1]
00299 #endif
00300
00301 #ifdef ON_WINDOWS
00302 #define I64 "%I64d"
00303 #define U64 "%I64u"
00304 #else
00305 #define I64 "%lld"
00306 #define U64 "%llu"
00307 #endif
00308
00309 #ifdef _MSC_VER
00310 #   define NORETURN __declspec(noreturn)
00311 #elif defined __GNUC__
00312 #   define NORETURN __attribute__ ((noreturn))
00313 #else
00314 #   define NORETURN
00315 #endif
00316
00317 static char __testlib_format_buffer[16777216];
00318 static int __testlib_format_buffer_usage_count = 0;
00319
00320 #define FMT_TO_RESULT(fmt, cstr, result)  std::string result;                                    \
00321             if (__testlib_format_buffer_usage_count != 0)                                        \
00322                 __testlib_fail("FMT_TO_RESULT::__testlib_format_buffer_usage_count != 0"); \
00323             __testlib_format_buffer_usage_count++;                                               \
00324             va_list ap;                                                                          \
00325             va_start(ap, fmt);                                                                   \
00326             vsnprintf(__testlib_format_buffer, sizeof(__testlib_format_buffer), cstr, ap); \
00327             va_end(ap);                                                                          \
00328             __testlib_format_buffer[sizeof(__testlib_format_buffer) - 1] = 0;                \
00329             result = std::string(__testlib_format_buffer);                                   \
00330             __testlib_format_buffer_usage_count--;                                           \
00331
00332 const long long __TESTLIB_LONGLONG_MAX = 9223372036854775807LL;
00333 const int __TESTLIB_MAX_TEST_CASE = 1073741823;
00334
```

```
00335 int __testlib_exitCode;
00336
00337 bool __testlib_hasTestCase;
00338 int __testlib_testCase = -1;
00339
00340 void setTestCase(int testCase);
00341
00342 void unsetTestCase() {
00343     __testlib_hasTestCase = false;
00344     __testlib_testCase = -1;
00345 }
00346
00347 NORETURN static void __testlib_fail(const std::string &message);
00348
00349 template<typename _Tp>
00350 #ifdef __GNUC__
00351 __attribute__((const))
00352 #endif
00353 static inline _Tp __testlib_abs(const _Tp &x) {
00354     return x > 0 ? x : -x;
00355 }
00356
00357 template<typename _Tp>
00358 #ifdef __GNUC__
00359 __attribute__((const))
00360 #endif
00361 static inline _Tp __testlib_min(const _Tp &a, const _Tp &b) {
00362     return a < b ? a : b;
00363 }
00364
00365 template<typename _Tp>
00366 #ifdef __GNUC__
00367 __attribute__((const))
00368 #endif
00369 static inline _Tp __testlib_max(const _Tp &a, const _Tp &b) {
00370     return a > b ? a : b;
00371 }
00372
00373 template<typename _Tp>
00374 #ifdef __GNUC__
00375 __attribute__((const))
00376 #endif
00377 static inline _Tp __testlib_crop(_Tp value, _Tp a, _Tp b) {
00378     return __testlib_min(__testlib_max(value, a), --b);
00379 }
00380
00381 #ifdef __GNUC__
00382 __attribute__((const))
00383 #endif
00384 static inline double __testlib_crop(double value, double a, double b) {
00385     value = __testlib_min(__testlib_max(value, a), b);
00386     if (value >= b)
00387         value = std::nexttoward(b, a);
00388     return value;
00389 }
00390
00391 static bool __testlib_prelimIsNaN(double r) {
00392     volatile double ra = r;
00393 #ifndef __BORLANDC__
00394     return ((ra != ra) == true) && ((ra == ra) == false) && ((1.0 > ra) == false) && ((1.0 < ra) ==
    false);
00395 #else
00396     return std::_isnan(ra);
00397 #endif
00398 }
00399
00400 #ifdef __GNUC__
00401 __attribute__((const))
00402 #endif
00403 static std::string removeDoubleTrailingZeroes(std::string value) {
00404     while (!value.empty() && value[value.length() - 1] == '0' && value.find('.') != std::string::npos)
00405         value = value.substr(0, value.length() - 1);
00406     if (!value.empty() && value[value.length() - 1] == '.')
00407         return value + '0';
00408     else
00409         return value;
00410 }
00411
00412 #ifdef __GNUC__
00413 __attribute__((const))
00414 #endif
00415 inline std::string upperCase(std::string s) {
00416     for (size_t i = 0; i < s.length(); i++)
00417         if ('a' <= s[i] && s[i] <= 'z')
00418             s[i] = char(s[i] - 'a' + 'A');
00419     return s;
00420 }
```

```
00421
00422 #ifdef __GNUC__
00423 __attribute__((const))
00424 #endif
00425 inline std::string lowerCase(std::string s) {
00426     for (size_t i = 0; i < s.length(); i++)
00427         if ('A' <= s[i] && s[i] <= 'Z')
00428             s[i] = char(s[i] - 'A' + 'a');
00429     return s;
00430 }
00431
00432 #ifdef __GNUC__
00433 __attribute__ ((format (printf, 1, 2)))
00434 #endif
00435 std::string format(const char *fmt, ...) {
00436     FMT_TO_RESULT(fmt, fmt, result);
00437     return result;
00438 }
00439
00440 std::string format(const std::string fmt, ...) {
00441     FMT_TO_RESULT(fmt, fmt.c_str(), result);
00442     return result;
00443 }
00444
00445 #ifdef __GNUC__
00446 __attribute__((const))
00447 #endif
00448 static std::string __testlib_part(const std::string &s);
00449
00450 static bool __testlib_isNaN(double r) {
00451     __TESTLIB_STATIC_ASSERT(sizeof(double) == sizeof(long long));
00452     volatile double ra = r;
00453     long long llr1, llr2;
00454     std::memcpy((void *) &llr1, (void *) &ra, sizeof(double));
00455     ra = -ra;
00456     std::memcpy((void *) &llr2, (void *) &ra, sizeof(double));
00457     long long llnan = 0xFFF8000000000000LL;
00458     return __testlib_prelimIsNaN(r) || llnan == llr1 || llnan == llr2;
00459 }
00460
00461 static double __testlib_nan() {
00462     __TESTLIB_STATIC_ASSERT(sizeof(double) == sizeof(long long));
00463 #ifndef NAN
00464     long long llnan = 0xFFF8000000000000LL;
00465     double nan;
00466     std::memcpy(&nan, &llnan, sizeof(double));
00467     return nan;
00468 #else
00469     return NAN;
00470 #endif
00471 }
00472
00473 static bool __testlib_isInfinite(double r) {
00474     volatile double ra = r;
00475     return (ra > 1E300 || ra < -1E300);
00476 }
00477
00478 #ifdef __GNUC__
00479 __attribute__((const))
00480 #endif
00481 inline bool doubleCompare(double expected, double result, double MAX_DOUBLE_ERROR) {
00482     MAX_DOUBLE_ERROR += 1E-15;
00483     if (__testlib_isNaN(expected)) {
00484         return __testlib_isNaN(result);
00485     } else if (__testlib_isInfinite(expected)) {
00486         if (expected > 0) {
00487             return result > 0 && __testlib_isInfinite(result);
00488         } else {
00489             return result < 0 && __testlib_isInfinite(result);
00490         }
00491     } else if (__testlib_isNaN(result) || __testlib_isInfinite(result)) {
00492         return false;
00493     } else if (__testlib_abs(result - expected) <= MAX_DOUBLE_ERROR) {
00494         return true;
00495     } else {
00496         double minv = __testlib_min(expected * (1.0 - MAX_DOUBLE_ERROR),
00497                                     expected * (1.0 + MAX_DOUBLE_ERROR));
00498         double maxv = __testlib_max(expected * (1.0 - MAX_DOUBLE_ERROR),
00499                                     expected * (1.0 + MAX_DOUBLE_ERROR));
00500         return result >= minv && result <= maxv;
00501     }
00502 }
00503
00504 #ifdef __GNUC__
00505 __attribute__((const))
00506 #endif
00507 inline double doubleDelta(double expected, double result) {
```

```
00508     double absolute = __testlib_abs(result - expected);
00509
00510     if (__testlib_abs(expected) > 1E-9) {
00511         double relative = __testlib_abs(absolute / expected);
00512         return __testlib_min(absolute, relative);
00513     } else
00514         return absolute;
00515 }
00516
00518 static void __testlib_set_binary(std::FILE *file) {
00519     if (NULL != file) {
00520 #ifdef ON_WINDOWS
00521 #   ifdef _O_BINARY
00522         if (stdin == file)
00523 #       ifdef STDIN_FILENO
00524                 return void(_setmode(STDIN_FILENO, _O_BINARY));
00525 #       else
00526                 return void(_setmode(_fileno(stdin), _O_BINARY));
00527 #       endif
00528         if (stdout == file)
00529 #       ifdef STDOUT_FILENO
00530                 return void(_setmode(STDOUT_FILENO, _O_BINARY));
00531 #       else
00532                 return void(_setmode(_fileno(stdout), _O_BINARY));
00533 #       endif
00534         if (stderr == file)
00535 #       ifdef STDERR_FILENO
00536                 return void(_setmode(STDERR_FILENO, _O_BINARY));
00537 #       else
00538                 return void(_setmode(_fileno(stderr), _O_BINARY));
00539 #       endif
00540 #   elif O_BINARY
00541         if (stdin == file)
00542 #       ifdef STDIN_FILENO
00543                 return void(setmode(STDIN_FILENO, O_BINARY));
00544 #       else
00545                 return void(setmode(fileno(stdin), O_BINARY));
00546 #       endif
00547         if (stdout == file)
00548 #       ifdef STDOUT_FILENO
00549                 return void(setmode(STDOUT_FILENO, O_BINARY));
00550 #       else
00551                 return void(setmode(fileno(stdout), O_BINARY));
00552 #       endif
00553         if (stderr == file)
00554 #       ifdef STDERR_FILENO
00555                 return void(setmode(STDERR_FILENO, O_BINARY));
00556 #       else
00557                 return void(setmode(fileno(stderr), O_BINARY));
00558 #       endif
00559 #   endif
00560 #endif
00561     }
00562 }
00563
00564 #if __cplusplus > 199711L || defined(_MSC_VER)
00565 template<typename _Tp>
00566 #ifdef __GNUC__
00567 __attribute__((const))
00568 #endif
00569 static std::string vtos(const _Tp &t, std::true_type) {
00570     if (t == 0)
00571         return "0";
00572     else {
00573         _Tp n(t);
00574         bool negative = n < 0;
00575         std::string s;
00576         while (n != 0) {
00577             _Tp digit = n % 10;
00578             if (digit < 0)
00579                 digit = -digit;
00580             s += char('0' + digit);
00581             n /= 10;
00582         }
00583         std::reverse(s.begin(), s.end());
00584         return negative ? "-" + s : s;
00585     }
00586 }
00587
00588 template<typename _Tp>
00589 static std::string vtos(const _Tp &t, std::false_type) {
00590     std::string s;
00591     static std::stringstream ss;
00592     ss.str(std::string());
00593     ss.clear();
00594     ss « t;
00595     ss » s;
```

```
00596     return s;
00597 }
00598
00599 template<typename _Tp>
00600 static std::string vtos(const _Tp &t) {
00601     return vtos(t, std::is_integral<_Tp>());
00602 }
00603
00604 /* signed case. */
00605 template<typename _Tp>
00606 static std::string toHumanReadableString(const _Tp &n, std::false_type) {
00607     if (n == 0)
00608         return vtos(n);
00609     int trailingZeroCount = 0;
00610     _Tp n_ = n;
00611     while (n_ % 10 == 0)
00612         n_ /= 10, trailingZeroCount++;
00613     if (trailingZeroCount >= 7) {
00614         if (n_ == 1)
00615             return "10^" + vtos(trailingZeroCount);
00616         else if (n_ == -1)
00617             return "-10^" + vtos(trailingZeroCount);
00618         else
00619             return vtos(n_) + "*10^" + vtos(trailingZeroCount);
00620     } else
00621         return vtos(n);
00622 }
00623
00624 /* unsigned case. */
00625 template<typename _Tp>
00626 static std::string toHumanReadableString(const _Tp &n, std::true_type) {
00627     if (n == 0)
00628         return vtos(n);
00629     int trailingZeroCount = 0;
00630     _Tp n_ = n;
00631     while (n_ % 10 == 0)
00632         n_ /= 10, trailingZeroCount++;
00633     if (trailingZeroCount >= 7) {
00634         if (n_ == 1)
00635             return "10^" + vtos(trailingZeroCount);
00636         else
00637             return vtos(n_) + "*10^" + vtos(trailingZeroCount);
00638     } else
00639         return vtos(n);
00640 }
00641
00642 template<typename _Tp>
00643 static std::string toHumanReadableString(const _Tp &n) {
00644     return toHumanReadableString(n, std::is_unsigned<_Tp>());
00645 }
00646 #else
00647 template<typename T>
00648 static std::string vtos(const T& t)
00649 {
00650     std::string s;
00651     static std::stringstream ss;
00652     ss.str(std::string());
00653     ss.clear();
00654     ss « t;
00655     ss » s;
00656     return s;
00657 }
00658
00659 template<typename T>
00660 static std::string toHumanReadableString(const T &n) {
00661     return vtos(n);
00662 }
00663 #endif
00664
00665 template<typename _Tp>
00666 static std::string toString(const _Tp &t) {
00667     return vtos(t);
00668 }
00669
00670 #if __cplusplus > 199711L || defined(_MSC_VER)
00671 /* opts */
00672 void prepareOpts(int argc, char* argv[]);
00673 #endif
00674
00675 /*
00676  * Very simple regex-like pattern.
00677  * It used for two purposes: validation and generation.
00678  *
00679  * For example, pattern("[a-z]{1,5}").next(rnd) will return
00680  * random string from lowercase latin letters with length
00681  * from 1 to 5. It is easier to call rnd.next("[a-z]{1,5}")
00682  * for the same effect.
```

```
00683  *
00684  * Another samples:
00685  * "mike|john" will generate (match) "mike" or "john";
00686  * "-?[1-9][0-9]{0,3}" will generate (match) non-zero integers from -9999 to 9999;
00687  * "id-([ac]|b{2})" will generate (match) "id-a", "id-bb", "id-c";
00688  * "[^0-9]*" will match sequences (empty or non-empty) without digits, you can't
00689  * use it for generations.
00690  *
00691  * You can't use pattern for generation if it contains meta-symbol '*'. Also it
00692  * is not recommended to use it for char-sets with meta-symbol '^' like [^a-z].
00693  *
00694  * For matching very simple greedy algorithm is used. For example, pattern
00695  * "[0-9]?1" will not match "1", because of greedy nature of matching.
00696  * Alternations (meta-symbols "|") are processed with brute-force algorithm, so
00697  * do not use many alternations in one expression.
00698  *
00699  * If you want to use one expression many times it is better to compile it into
00700  * a single pattern like "pattern p("[a-z]+")". Later you can use
00701  * "p.matches(std::string s)" or "p.next(random_t& rd)" to check matching or generate
00702  * new string by pattern.
00703  *
00704  * Simpler way to read token and check it for pattern matching is "inf.readToken("[a-z]+")".
00705  *
00706  * All spaces are ignored in regex, unless escaped with \. For example, ouf.readLine("NO SOLUTION")
00707  * will expect "NOSOLUTION", the correct call should be ouf.readLine("NO\\ SOLUTION") or
00708  * ouf.readLine(R"(NO\ SOLUTION)") if you prefer raw string literals from C++11.
00709  */
00710 class random_t;
00711
00712 class pattern {
00713 public:
00714     /* Create pattern instance by string. */
00715     pattern(std::string s);
00716
00717     /* Generate new string by pattern and given random_t. */
00718     std::string next(random_t &rnd) const;
00719
00720     /* Checks if given string match the pattern. */
00721     bool matches(const std::string &s) const;
00722
00723     /* Returns source string of the pattern. */
00724     std::string src() const;
00725
00726 private:
00727     bool matches(const std::string &s, size_t pos) const;
00728
00729     std::string s;
00730     std::vector<pattern> children;
00731     std::vector<char> chars;
00732     int from;
00733     int to;
00734 };
00735
00736 /*
00737  * Use random_t instances to generate random values. It is preferred
00738  * way to use randoms instead of rand() function or self-written
00739  * randoms.
00740  *
00741  * Testlib defines global variable "rnd" of random_t class.
00742  * Use registerGen(argc, argv, 1) to setup random_t seed be command
00743  * line (to use latest random generator version).
00744  *
00745  * Random generates uniformly distributed values if another strategy is
00746  * not specified explicitly.
00747  */
00748 class random_t {
00749 private:
00750     unsigned long long seed;
00751     static const unsigned long long multiplier;
00752     static const unsigned long long addend;
00753     static const unsigned long long mask;
00754     static const int lim;
00755
00756     long long nextBits(int bits) {
00757         if (bits <= 48) {
00758             seed = (seed * multiplier + addend) & mask;
00759             return (long long) (seed >> (48 - bits));
00760         } else {
00761             if (bits > 63)
00762                 __testlib_fail("random_t::nextBits(int bits): n must be less than 64");
00763
00764             int lowerBitCount = (random_t::version == 0 ? 31 : 32);
00765
00766             long long left = (nextBits(31) << 32);
00767             long long right = nextBits(lowerBitCount);
00768
00769             return left ^ right;
```

```
00770         }
00771     }
00772
00773 public:
00774     static int version;
00775
00776     /* New random_t with fixed seed. */
00777     random_t()
00778             : seed(3905348978240129619LL) {
00779     }
00780
00781     /* Sets seed by command line. */
00782     void setSeed(int argc, char *argv[]) {
00783         random_t p;
00784
00785         seed = 3905348978240129619LL;
00786         for (int i = 1; i < argc; i++) {
00787             std::size_t le = std::strlen(argv[i]);
00788             for (std::size_t j = 0; j < le; j++)
00789                 seed = seed * multiplier + (unsigned int) (argv[i][j]) + addend;
00790             seed += multiplier / addend;
00791         }
00792
00793         seed = seed & mask;
00794     }
00795
00796     /* Sets seed by given value. */
00797     void setSeed(long long _seed) {
00798         seed = (unsigned long long) _seed;
00799         seed = (seed ^ multiplier) & mask;
00800     }
00801
00802 #ifndef __BORLANDC__
00803
00804     /* Random string value by given pattern (see pattern documentation). */
00805     std::string next(const std::string &ptrn) {
00806         pattern p(ptrn);
00807         return p.next(*this);
00808     }
00809
00810 #else
00811     /* Random string value by given pattern (see pattern documentation). */
00812     std::string next(std::string ptrn)
00813     {
00814         pattern p(ptrn);
00815         return p.next(*this);
00816     }
00817 #endif
00818
00819     /* Random value in range [0, n-1]. */
00820     int next(int n) {
00821         if (n <= 0)
00822             __testlib_fail("random_t::next(int n): n must be positive");
00823
00824         if ((n & -n) == n)  // n is a power of 2
00825             return (int) ((n * (long long) nextBits(31)) >> 31);
00826
00827         const long long limit = INT_MAX / n * n;
00828
00829         long long bits;
00830         do {
00831             bits = nextBits(31);
00832         } while (bits >= limit);
00833
00834         return int(bits % n);
00835     }
00836
00837     /* Random value in range [0, n-1]. */
00838     unsigned int next(unsigned int n) {
00839         if (n >= INT_MAX)
00840             __testlib_fail("random_t::next(unsigned int n): n must be less INT_MAX");
00841         return (unsigned int) next(int(n));
00842     }
00843
00844     /* Random value in range [0, n-1]. */
00845     long long next(long long n) {
00846         if (n <= 0)
00847             __testlib_fail("random_t::next(long long n): n must be positive");
00848
00849         const long long limit = __TESTLIB_LONGLONG_MAX / n * n;
00850
00851         long long bits;
00852         do {
00853             bits = nextBits(63);
00854         } while (bits >= limit);
00855
00856         return bits % n;
```

```
00857     }
00858
00859     /* Random value in range [0, n-1]. */
00860     unsigned long long next(unsigned long long n) {
00861         if (n >= (unsigned long long) (__TESTLIB_LONGLONG_MAX))
00862             __testlib_fail("random_t::next(unsigned long long n): n must be less LONGLONG_MAX");
00863         return (unsigned long long) next((long long) (n));
00864     }
00865
00866     /* Random value in range [0, n-1]. */
00867     long next(long n) {
00868         return (long) next((long long) (n));
00869     }
00870
00871     /* Random value in range [0, n-1]. */
00872     unsigned long next(unsigned long n) {
00873         if (n >= (unsigned long) (LONG_MAX))
00874             __testlib_fail("random_t::next(unsigned long n): n must be less LONG_MAX");
00875         return (unsigned long) next((unsigned long long) (n));
00876     }
00877
00878     /* Returns random value in range [from,to]. */
00879     int next(int from, int to) {
00880         return int(next((long long) to - from + 1) + from);
00881     }
00882
00883     /* Returns random value in range [from,to]. */
00884     unsigned int next(unsigned int from, unsigned int to) {
00885         return (unsigned int) (next((long long) to - from + 1) + from);
00886     }
00887
00888     /* Returns random value in range [from,to]. */
00889     long long next(long long from, long long to) {
00890         return next(to - from + 1) + from;
00891     }
00892
00893     /* Returns random value in range [from,to]. */
00894     unsigned long long next(unsigned long long from, unsigned long long to) {
00895         if (from > to)
00896             __testlib_fail("random_t::next(unsigned long long from, unsigned long long to): from can't
      not exceed to");
00897         return next(to - from + 1) + from;
00898     }
00899
00900     /* Returns random value in range [from,to]. */
00901     long next(long from, long to) {
00902         return next(to - from + 1) + from;
00903     }
00904
00905     /* Returns random value in range [from,to]. */
00906     unsigned long next(unsigned long from, unsigned long to) {
00907         if (from > to)
00908             __testlib_fail("random_t::next(unsigned long from, unsigned long to): from can't not
      exceed to");
00909         return next(to - from + 1) + from;
00910     }
00911
00912     /* Random double value in range [0, 1). */
00913     double next() {
00914         long long left = ((long long) (nextBits(26)) « 27);
00915         long long right = nextBits(27);
00916         return __testlib_crop((double) (left + right) / (double) (1LL « 53), 0.0, 1.0);
00917     }
00918
00919     /* Random double value in range [0, n). */
00920     double next(double n) {
00921         if (n <= 0.0)
00922             __testlib_fail("random_t::next(double): n should be positive");
00923         return __testlib_crop(n * next(), 0.0, n);
00924     }
00925
00926     /* Random double value in range [from, to). */
00927     double next(double from, double to) {
00928         if (from >= to)
00929             __testlib_fail("random_t::next(double from, double to): from should be strictly less than
      to");
00930         return next(to - from) + from;
00931     }
00932
00933     /* Returns random element from container. */
00934     template<typename Container>
00935     typename Container::value_type any(const Container &c) {
00936         int size = int(c.size());
00937         if (size <= 0)
00938             __testlib_fail("random_t::any(const Container& c): c.size() must be positive");
00939         typename Container::const_iterator it = c.begin();
00940         std::advance(it, next(size));
```

```
00941         return *it;
00942     }
00943
00944     /* Returns random element from iterator range. */
00945     template<typename Iter>
00946     typename Iter::value_type any(const Iter &begin, const Iter &end) {
00947         int size = static_cast<int>(std::distance(begin, end));
00948         if (size <= 0)
00949             __testlib_fail("random_t::any(const Iter& begin, const Iter& end): range must have
      positive length");
00950         Iter it = begin;
00951         std::advance(it, next(size));
00952         return *it;
00953     }
00954
00955     /* Random string value by given pattern (see pattern documentation). */
00956 #ifdef __GNUC__
00957     __attribute__ ((format (printf, 2, 3)))
00958 #endif
00959     std::string next(const char *format, ...) {
00960         FMT_TO_RESULT(format, format, ptrn);
00961         return next(ptrn);
00962     }
00963
00964     /*
00965      * Weighted next. If type == 0 than it is usual "next()".
00966      *
00967      * If type = 1, than it returns "max(next(), next())"
00968      * (the number of "max" functions equals to "type").
00969      *
00970      * If type < 0, than "max" function replaces with "min".
00971      */
00972     int wnext(int n, int type) {
00973         if (n <= 0)
00974             __testlib_fail("random_t::wnext(int n, int type): n must be positive");
00975
00976         if (abs(type) < random_t::lim) {
00977             int result = next(n);
00978
00979             for (int i = 0; i < +type; i++)
00980                 result = __testlib_max(result, next(n));
00981
00982             for (int i = 0; i < -type; i++)
00983                 result = __testlib_min(result, next(n));
00984
00985             return result;
00986         } else {
00987             double p;
00988
00989             if (type > 0)
00990                 p = std::pow(next() + 0.0, 1.0 / (type + 1));
00991             else
00992                 p = 1 - std::pow(next() + 0.0, 1.0 / (-type + 1));
00993
00994             return __testlib_crop((int) (double(n) * p), 0, n);
00995         }
00996     }
00997
00998     /* See wnext(int, int). It uses the same algorithms. */
00999     long long wnext(long long n, int type) {
01000         if (n <= 0)
01001             __testlib_fail("random_t::wnext(long long n, int type): n must be positive");
01002
01003         if (abs(type) < random_t::lim) {
01004             long long result = next(n);
01005
01006             for (int i = 0; i < +type; i++)
01007                 result = __testlib_max(result, next(n));
01008
01009             for (int i = 0; i < -type; i++)
01010                 result = __testlib_min(result, next(n));
01011
01012             return result;
01013         } else {
01014             double p;
01015
01016             if (type > 0)
01017                 p = std::pow(next() + 0.0, 1.0 / (type + 1));
01018             else
01019                 p = 1 - std::pow(next() + 0.0, 1.0 / (-type + 1));
01020
01021             return __testlib_crop((long long) (double(n) * p), 0LL, n);
01022         }
01023     }
01024
01025     /* Returns value in [0, n). See wnext(int, int). It uses the same algorithms. */
01026     double wnext(double n, int type) {
```

```
01027         if (n <= 0)
01028             __testlib_fail("random_t::wnext(double n, int type): n must be positive");
01029
01030         if (abs(type) < random_t::lim) {
01031             double result = next();
01032
01033             for (int i = 0; i < +type; i++)
01034                 result = __testlib_max(result, next());
01035
01036             for (int i = 0; i < -type; i++)
01037                 result = __testlib_min(result, next());
01038
01039             return n * result;
01040         } else {
01041             double p;
01042
01043             if (type > 0)
01044                 p = std::pow(next() + 0.0, 1.0 / (type + 1));
01045             else
01046                 p = 1 - std::pow(next() + 0.0, 1.0 / (-type + 1));
01047
01048             return __testlib_crop(n * p, 0.0, n);
01049         }
01050     }
01051
01052     /* Returns value in [0, 1). See wnext(int, int). It uses the same algorithms. */
01053     double wnext(int type) {
01054         return wnext(1.0, type);
01055     }
01056
01057     /* See wnext(int, int). It uses the same algorithms. */
01058     unsigned int wnext(unsigned int n, int type) {
01059         if (n >= INT_MAX)
01060             __testlib_fail("random_t::wnext(unsigned int n, int type): n must be less INT_MAX");
01061         return (unsigned int) wnext(int(n), type);
01062     }
01063
01064     /* See wnext(int, int). It uses the same algorithms. */
01065     unsigned long long wnext(unsigned long long n, int type) {
01066         if (n >= (unsigned long long) (__TESTLIB_LONGLONG_MAX))
01067             __testlib_fail("random_t::wnext(unsigned long long n, int type): n must be less
      LONGLONG_MAX");
01068
01069         return (unsigned long long) wnext((long long) (n), type);
01070     }
01071
01072     /* See wnext(int, int). It uses the same algorithms. */
01073     long wnext(long n, int type) {
01074         return (long) wnext((long long) (n), type);
01075     }
01076
01077     /* See wnext(int, int). It uses the same algorithms. */
01078     unsigned long wnext(unsigned long n, int type) {
01079         if (n >= (unsigned long) (LONG_MAX))
01080             __testlib_fail("random_t::wnext(unsigned long n, int type): n must be less LONG_MAX");
01081
01082         return (unsigned long) wnext((unsigned long long) (n), type);
01083     }
01084
01085     /* Returns weighted random value in range [from, to]. */
01086     int wnext(int from, int to, int type) {
01087         if (from > to)
01088             __testlib_fail("random_t::wnext(int from, int to, int type): from can't not exceed to");
01089         return wnext(to - from + 1, type) + from;
01090     }
01091
01092     /* Returns weighted random value in range [from, to]. */
01093     int wnext(unsigned int from, unsigned int to, int type) {
01094         if (from > to)
01095             __testlib_fail("random_t::wnext(unsigned int from, unsigned int to, int type): from can't
      not exceed to");
01096         return int(wnext(to - from + 1, type) + from);
01097     }
01098
01099     /* Returns weighted random value in range [from, to]. */
01100     long long wnext(long long from, long long to, int type) {
01101         if (from > to)
01102             __testlib_fail("random_t::wnext(long long from, long long to, int type): from can't not
      exceed to");
01103         return wnext(to - from + 1, type) + from;
01104     }
01105
01106     /* Returns weighted random value in range [from, to]. */
01107     unsigned long long wnext(unsigned long long from, unsigned long long to, int type) {
01108         if (from > to)
01109             __testlib_fail(
01110                     "random_t::wnext(unsigned long long from, unsigned long long to, int type): from
```

```
      can't not exceed to");
01111         return wnext(to - from + 1, type) + from;
01112     }
01113
01114     /* Returns weighted random value in range [from, to]. */
01115     long wnext(long from, long to, int type) {
01116         if (from > to)
01117             __testlib_fail("random_t::wnext(long from, long to, int type): from can't not exceed to");
01118         return wnext(to - from + 1, type) + from;
01119     }
01120
01121     /* Returns weighted random value in range [from, to]. */
01122     unsigned long wnext(unsigned long from, unsigned long to, int type) {
01123         if (from > to)
01124             __testlib_fail("random_t::wnext(unsigned long from, unsigned long to, int type): from
      can't not exceed to");
01125         return wnext(to - from + 1, type) + from;
01126     }
01127
01128     /* Returns weighted random double value in range [from, to). */
01129     double wnext(double from, double to, int type) {
01130         if (from >= to)
01131             __testlib_fail("random_t::wnext(double from, double to, int type): from should be strictly
      less than to");
01132         return wnext(to - from, type) + from;
01133     }
01134
01135     /* Returns weighted random element from container. */
01136     template<typename Container>
01137     typename Container::value_type wany(const Container &c, int type) {
01138         int size = int(c.size());
01139         if (size <= 0)
01140             __testlib_fail("random_t::wany(const Container& c, int type): c.size() must be positive");
01141         typename Container::const_iterator it = c.begin();
01142         std::advance(it, wnext(size, type));
01143         return *it;
01144     }
01145
01146     /* Returns weighted random element from iterator range. */
01147     template<typename Iter>
01148     typename Iter::value_type wany(const Iter &begin, const Iter &end, int type) {
01149         int size = static_cast<int>(std::distance(begin, end));
01150         if (size <= 0)
01151             __testlib_fail(
01152                     "random_t::any(const Iter& begin, const Iter& end, int type): range must have
      positive length");
01153         Iter it = begin;
01154         std::advance(it, wnext(size, type));
01155         return *it;
01156     }
01157
01158     /* Returns random permutation of the given size (values are between `first` and `first`+size-1)*/
01159     template<typename _Tp, typename E>
01160     std::vector<E> perm(_Tp size, E first) {
01161         if (size < 0)
01162             __testlib_fail("random_t::perm(T size, E first = 0): size must non-negative");
01163         else if (size == 0)
01164             return std::vector<E>();
01165         std::vector<E> p(size);
01166         E current = first;
01167         for (_Tp i = 0; i < size; i++)
01168             p[i] = current++;
01169         if (size > 1)
01170             for (_Tp i = 1; i < size; i++)
01171                 std::swap(p[i], p[next(i + 1)]);
01172         return p;
01173     }
01174
01175     /* Returns random permutation of the given size (values are between 0 and size-1)*/
01176     template<typename _Tp>
01177     std::vector<_Tp> perm(_Tp size) {
01178         return perm(size, _Tp(0));
01179     }
01180
01181     /* Returns `size` unordered (unsorted) distinct numbers between `from` and `to`. */
01182     template<typename _Tp>
01183     std::vector<_Tp> distinct(int size, _Tp from, _Tp to) {
01184         std::vector<_Tp> result;
01185         if (size == 0)
01186             return result;
01187
01188         if (from > to)
01189             __testlib_fail("random_t::distinct expected from <= to");
01190
01191         if (size < 0)
01192             __testlib_fail("random_t::distinct expected size >= 0");
01193
```

```
01194         uint64_t n = to - from + 1;
01195         if (uint64_t(size) > n)
01196             __testlib_fail("random_t::distinct expected size <= to - from + 1");
01197 
01198         double expected = 0.0;
01199         for (int i = 1; i <= size; i++)
01200             expected += double(n) / double(n - i + 1);
01201 
01202         if (expected < double(n)) {
01203             std::set<_Tp> vals;
01204             while (int(vals.size()) < size) {
01205                 _Tp x = _Tp(next(from, to));
01206                 if (vals.insert(x).second)
01207                     result.push_back(x);
01208             }
01209         } else {
01210             if (n > 1000000000)
01211                 __testlib_fail("random_t::distinct here expected to - from + 1 <= 1000000000");
01212             std::vector<_Tp> p(perm(int(n), from));
01213             result.insert(result.end(), p.begin(), p.begin() + size);
01214         }
01215 
01216         return result;
01217     }
01218 
01219     /* Returns `size' unordered (unsorted) distinct numbers between `0' and `upper'-1. */
01220     template<typename _Tp>
01221     std::vector<_Tp> distinct(int size, _Tp upper) {
01222         if (size < 0)
01223             __testlib_fail("random_t::distinct expected size >= 0");
01224         if (size == 0)
01225             return std::vector<_Tp>();
01226 
01227         if (upper <= 0)
01228             __testlib_fail("random_t::distinct expected upper > 0");
01229         if (size > upper)
01230             __testlib_fail("random_t::distinct expected size <= upper");
01231 
01232         return distinct(size, _Tp(0), upper - 1);
01233     }
01234 
01235     /* Returns random (unsorted) partition which is a representation of sum as a sum of integers not
      less than min_part. */
01236     template<typename _Tp>
01237     std::vector<_Tp> partition(int size, _Tp sum, _Tp min_part) {
01238         if (size < 0)
01239             __testlib_fail("random_t::partition: size < 0");
01240         if (size == 0 && sum != 0)
01241             __testlib_fail("random_t::partition: size == 0 && sum != 0");
01242         if (min_part * size > sum)
01243             __testlib_fail("random_t::partition: min_part * size > sum");
01244         if (size == 0 && sum == 0)
01245             return std::vector<_Tp>();
01246 
01247         _Tp sum_ = sum;
01248         sum -= min_part * size;
01249 
01250         std::vector<_Tp> septums(size);
01251         std::vector<_Tp> d = distinct(size - 1, _Tp(1), _Tp(sum + size - 1));
01252         for (int i = 0; i + 1 < size; i++)
01253             septums[i + 1] = d[i];
01254         sort(septums.begin(), septums.end());
01255 
01256         std::vector<_Tp> result(size);
01257         for (int i = 0; i + 1 < size; i++)
01258             result[i] = septums[i + 1] - septums[i] - 1;
01259         result[size - 1] = sum + size - 1 - septums.back();
01260 
01261         for (std::size_t i = 0; i < result.size(); i++)
01262             result[i] += min_part;
01263 
01264         _Tp result_sum = 0;
01265         for (std::size_t i = 0; i < result.size(); i++)
01266             result_sum += result[i];
01267         if (result_sum != sum_)
01268             __testlib_fail("random_t::partition: partition sum is expected to be the given sum");
01269 
01270         if (*std::min_element(result.begin(), result.end()) < min_part)
01271             __testlib_fail("random_t::partition: partition min is expected to be no less than the
      given min_part");
01272 
01273         if (int(result.size()) != size || result.size() != (size_t) size)
01274             __testlib_fail("random_t::partition: partition size is expected to be equal to the given
      size");
01275 
01276         return result;
01277     }
```

```
01278
01279     /* Returns random (unsorted) partition which is a representation of sum as a sum of positive
    integers. */
01280     template<typename _Tp>
01281     std::vector<_Tp> partition(int size, _Tp sum) {
01282         return partition(size, sum, _Tp(1));
01283     }
01284 };
01285
01286 const int random_t::lim = 25;
01287 const unsigned long long random_t::multiplier = 0x5DEECE66DLL;
01288 const unsigned long long random_t::addend = 0xBLL;
01289 const unsigned long long random_t::mask = (1LL << 48) - 1;
01290 int random_t::version = -1;
01291
01292 /* Pattern implementation */
01293 bool pattern::matches(const std::string &s) const {
01294     return matches(s, 0);
01295 }
01296
01297 static bool __pattern_isSlash(const std::string &s, size_t pos) {
01298     return s[pos] == '\\';
01299 }
01300
01301 #ifdef __GNUC__
01302 __attribute__((pure))
01303 #endif
01304 static bool __pattern_isCommandChar(const std::string &s, size_t pos, char value) {
01305     if (pos >= s.length())
01306         return false;
01307
01308     int slashes = 0;
01309
01310     int before = int(pos) - 1;
01311     while (before >= 0 && s[before] == '\\')
01312         before--, slashes++;
01313
01314     return slashes % 2 == 0 && s[pos] == value;
01315 }
01316
01317 static char __pattern_getChar(const std::string &s, size_t &pos) {
01318     if (__pattern_isSlash(s, pos))
01319         pos += 2;
01320     else
01321         pos++;
01322
01323     return s[pos - 1];
01324 }
01325
01326 #ifdef __GNUC__
01327 __attribute__((pure))
01328 #endif
01329 static int __pattern_greedyMatch(const std::string &s, size_t pos, const std::vector<char> chars) {
01330     int result = 0;
01331
01332     while (pos < s.length()) {
01333         char c = s[pos++];
01334         if (!std::binary_search(chars.begin(), chars.end(), c))
01335             break;
01336         else
01337             result++;
01338     }
01339
01340     return result;
01341 }
01342
01343 std::string pattern::src() const {
01344     return s;
01345 }
01346
01347 bool pattern::matches(const std::string &s, size_t pos) const {
01348     std::string result;
01349
01350     if (to > 0) {
01351         int size = __pattern_greedyMatch(s, pos, chars);
01352         if (size < from)
01353             return false;
01354         if (size > to)
01355             size = to;
01356         pos += size;
01357     }
01358
01359     if (children.size() > 0) {
01360         for (size_t child = 0; child < children.size(); child++)
01361             if (children[child].matches(s, pos))
01362                 return true;
01363         return false;
```

```
01364     } else
01365         return pos == s.length();
01366 }
01367
01368 std::string pattern::next(random_t &rnd) const {
01369     std::string result;
01370     result.reserve(20);
01371
01372     if (to == INT_MAX)
01373         __testlib_fail("pattern::next(random_t& rnd): can't process character '*' for generation");
01374
01375     if (to > 0) {
01376         int count = rnd.next(to - from + 1) + from;
01377         for (int i = 0; i < count; i++)
01378             result += chars[rnd.next(int(chars.size()))];
01379     }
01380
01381     if (children.size() > 0) {
01382         int child = rnd.next(int(children.size()));
01383         result += children[child].next(rnd);
01384     }
01385
01386     return result;
01387 }
01388
01389 static void __pattern_scanCounts(const std::string &s, size_t &pos, int &from, int &to) {
01390     if (pos >= s.length()) {
01391         from = to = 1;
01392         return;
01393     }
01394
01395     if (__pattern_isCommandChar(s, pos, '{')) {
01396         std::vector<std::string> parts;
01397         std::string part;
01398
01399         pos++;
01400
01401         while (pos < s.length() && !__pattern_isCommandChar(s, pos, '}')) {
01402             if (__pattern_isCommandChar(s, pos, ','))
01403                 parts.push_back(part), part = "", pos++;
01404             else
01405                 part += __pattern_getChar(s, pos);
01406         }
01407
01408         if (part != "")
01409             parts.push_back(part);
01410
01411         if (!__pattern_isCommandChar(s, pos, '}'))
01412             __testlib_fail("pattern: Illegal pattern (or part) \"" + s + "\"");
01413
01414         pos++;
01415
01416         if (parts.size() < 1 || parts.size() > 2)
01417             __testlib_fail("pattern: Illegal pattern (or part) \"" + s + "\"");
01418
01419         std::vector<int> numbers;
01420
01421         for (size_t i = 0; i < parts.size(); i++) {
01422             if (parts[i].length() == 0)
01423                 __testlib_fail("pattern: Illegal pattern (or part) \"" + s + "\"");
01424             int number;
01425             if (std::sscanf(parts[i].c_str(), "%d", &number) != 1)
01426                 __testlib_fail("pattern: Illegal pattern (or part) \"" + s + "\"");
01427             numbers.push_back(number);
01428         }
01429
01430         if (numbers.size() == 1)
01431             from = to = numbers[0];
01432         else
01433             from = numbers[0], to = numbers[1];
01434
01435         if (from > to)
01436             __testlib_fail("pattern: Illegal pattern (or part) \"" + s + "\"");
01437     } else {
01438         if (__pattern_isCommandChar(s, pos, '?')) {
01439             from = 0, to = 1, pos++;
01440             return;
01441         }
01442
01443         if (__pattern_isCommandChar(s, pos, '*')) {
01444             from = 0, to = INT_MAX, pos++;
01445             return;
01446         }
01447
01448         if (__pattern_isCommandChar(s, pos, '+')) {
01449             from = 1, to = INT_MAX, pos++;
01450             return;
```

```
01451         }
01452
01453         from = to = 1;
01454     }
01455 }
01456
01457 static std::vector<char> __pattern_scanCharSet(const std::string &s, size_t &pos) {
01458     if (pos >= s.length())
01459         __testlib_fail("pattern: Illegal pattern (or part) \"" + s + "\"");
01460
01461     std::vector<char> result;
01462
01463     if (__pattern_isCommandChar(s, pos, '[')) {
01464         pos++;
01465         bool negative = __pattern_isCommandChar(s, pos, '^');
01466         if (negative)
01467             pos++;
01468
01469         char prev = 0;
01470
01471         while (pos < s.length() && !__pattern_isCommandChar(s, pos, ']')) {
01472             if (__pattern_isCommandChar(s, pos, '-') && prev != 0) {
01473                 pos++;
01474
01475                 if (pos + 1 == s.length() || __pattern_isCommandChar(s, pos, ']')) {
01476                     result.push_back(prev);
01477                     prev = '-';
01478                     continue;
01479                 }
01480
01481                 char next = __pattern_getChar(s, pos);
01482                 if (prev > next)
01483                     __testlib_fail("pattern: Illegal pattern (or part) \"" + s + "\"");
01484
01485                 for (char c = prev; c != next; c++)
01486                     result.push_back(c);
01487                 result.push_back(next);
01488
01489                 prev = 0;
01490             } else {
01491                 if (prev != 0)
01492                     result.push_back(prev);
01493                 prev = __pattern_getChar(s, pos);
01494             }
01495         }
01496
01497         if (prev != 0)
01498             result.push_back(prev);
01499
01500         if (!__pattern_isCommandChar(s, pos, ']'))
01501             __testlib_fail("pattern: Illegal pattern (or part) \"" + s + "\"");
01502
01503         pos++;
01504
01505         if (negative) {
01506             std::sort(result.begin(), result.end());
01507             std::vector<char> actuals;
01508             for (int code = 0; code < 255; code++) {
01509                 char c = char(code);
01510                 if (!std::binary_search(result.begin(), result.end(), c))
01511                     actuals.push_back(c);
01512             }
01513             result = actuals;
01514         }
01515
01516         std::sort(result.begin(), result.end());
01517     } else
01518         result.push_back(__pattern_getChar(s, pos));
01519
01520     return result;
01521 }
01522
01523 pattern::pattern(std::string s) : s(s), from(0), to(0) {
01524     std::string t;
01525     for (size_t i = 0; i < s.length(); i++)
01526         if (!__pattern_isCommandChar(s, i, ' '))
01527             t += s[i];
01528     s = t;
01529
01530     int opened = 0;
01531     int firstClose = -1;
01532     std::vector<int> seps;
01533
01534     for (size_t i = 0; i < s.length(); i++) {
01535         if (__pattern_isCommandChar(s, i, '(')) {
01536             opened++;
01537             continue;
```

```
01538         }
01539
01540         if (__pattern_isCommandChar(s, i, ')')) {
01541             opened--;
01542             if (opened == 0 && firstClose == -1)
01543                 firstClose = int(i);
01544             continue;
01545         }
01546
01547         if (opened < 0)
01548             __testlib_fail("pattern: Illegal pattern (or part) \"" + s + "\"");
01549
01550         if (__pattern_isCommandChar(s, i, '|') && opened == 0)
01551             seps.push_back(int(i));
01552     }
01553
01554     if (opened != 0)
01555         __testlib_fail("pattern: Illegal pattern (or part) \"" + s + "\"");
01556
01557     if (seps.size() == 0 && firstClose + 1 == (int) s.length()
01558         && __pattern_isCommandChar(s, 0, '(') && __pattern_isCommandChar(s, s.length() - 1, ')')) {
01559         children.push_back(pattern(s.substr(1, s.length() - 2)));
01560     } else {
01561         if (seps.size() > 0) {
01562             seps.push_back(int(s.length()));
01563             int last = 0;
01564
01565             for (size_t i = 0; i < seps.size(); i++) {
01566                 children.push_back(pattern(s.substr(last, seps[i] - last)));
01567                 last = seps[i] + 1;
01568             }
01569         } else {
01570             size_t pos = 0;
01571             chars = __pattern_scanCharSet(s, pos);
01572             __pattern_scanCounts(s, pos, from, to);
01573             if (pos < s.length())
01574                 children.push_back(pattern(s.substr(pos)));
01575         }
01576     }
01577 }
01578
01579 /* End of pattern implementation */
01580
01581 template<typename C>
01582 inline bool isEof(C c) {
01583     return c == EOFC;
01584 }
01585
01586 template<typename C>
01587 inline bool isEoln(C c) {
01588     return (c == LF || c == CR);
01589 }
01590
01591 template<typename C>
01592 inline bool isBlanks(C c) {
01593     return (c == LF || c == CR || c == SPACE || c == TAB);
01594 }
01595
01596 inline std::string trim(const std::string &s) {
01597     if (s.empty())
01598         return s;
01599
01600     int left = 0;
01601     while (left < int(s.length()) && isBlanks(s[left]))
01602         left++;
01603     if (left >= int(s.length()))
01604         return "";
01605
01606     int right = int(s.length()) - 1;
01607     while (right >= 0 && isBlanks(s[right]))
01608         right--;
01609     if (right < 0)
01610         return "";
01611
01612     return s.substr(left, right - left + 1);
01613 }
01614
01615 enum TMode {
01616     _input, _output, _answer
01617 };
01618
01619 /* Outcomes 6-15 are reserved for future use. */
01620 enum TResult {
01621     _ok = 0,
01622     _wa = 1,
01623     _pe = 2,
01624     _fail = 3,
```

```
01625     _dirt = 4,
01626     _points = 5,
01627     _unexpected_eof = 8,
01628     _partially = 16
01629 };
01630
01631 enum TTestlibMode {
01632     _unknown, _checker, _validator, _generator, _interactor, _scorer
01633 };
01634
01635 #define _pc(exitCode) (TResult(_partially + (exitCode)))
01636
01637 /* Outcomes 6-15 are reserved for future use. */
01638 const std::string outcomes[] = {
01639         "accepted",
01640         "wrong-answer",
01641         "presentation-error",
01642         "fail",
01643         "fail",
01644 #ifndef PCMS2
01645         "points",
01646 #else
01647         "relative-scoring",
01648 #endif
01649         "reserved",
01650         "reserved",
01651         "unexpected-eof",
01652         "reserved",
01653         "reserved",
01654         "reserved",
01655         "reserved",
01656         "reserved",
01657         "reserved",
01658         "reserved",
01659         "partially-correct"
01660 };
01661
01662 class InputStreamReader {
01663 public:
01664     virtual void setTestCase(int testCase) = 0;
01665
01666     virtual std::vector<int> getReadChars() = 0;
01667
01668     virtual int curChar() = 0;
01669
01670     virtual int nextChar() = 0;
01671
01672     virtual void skipChar() = 0;
01673
01674     virtual void unreadChar(int c) = 0;
01675
01676     virtual std::string getName() = 0;
01677
01678     virtual bool eof() = 0;
01679
01680     virtual void close() = 0;
01681
01682     virtual int getLine() = 0;
01683
01684     virtual ~InputStreamReader() = 0;
01685 };
01686
01687 InputStreamReader::~InputStreamReader() {
01688     // No operations.
01689 }
01690
01691 class StringInputStreamReader : public InputStreamReader {
01692 private:
01693     std::string s;
01694     size_t pos;
01695
01696 public:
01697     StringInputStreamReader(const std::string &content) : s(content), pos(0) {
01698         // No operations.
01699     }
01700
01701     void setTestCase(int) {
01702         __testlib_fail("setTestCase not implemented in StringInputStreamReader");
01703     }
01704
01705     std::vector<int> getReadChars() {
01706         __testlib_fail("getReadChars not implemented in StringInputStreamReader");
01707     }
01708
01709     int curChar() {
01710         if (pos >= s.length())
01711             return EOFC;
```

```
01712         else
01713             return s[pos];
01714     }
01715
01716     int nextChar() {
01717         if (pos >= s.length()) {
01718             pos++;
01719             return EOFC;
01720         } else
01721             return s[pos++];
01722     }
01723
01724     void skipChar() {
01725         pos++;
01726     }
01727
01728     void unreadChar(int c) {
01729         if (pos == 0)
01730             __testlib_fail("StringInputStreamReader::unreadChar(int): pos == 0.");
01731         pos--;
01732         if (pos < s.length())
01733             s[pos] = char(c);
01734     }
01735
01736     std::string getName() {
01737         return __testlib_part(s);
01738     }
01739
01740     int getLine() {
01741         return -1;
01742     }
01743
01744     bool eof() {
01745         return pos >= s.length();
01746     }
01747
01748     void close() {
01749         // No operations.
01750     }
01751 };
01752
01753 class FileInputStreamReader : public InputStreamReader {
01754 private:
01755     std::FILE *file;
01756     std::string name;
01757     int line;
01758     std::vector<int> undoChars;
01759     std::vector<int> readChars;
01760     std::vector<int> undoReadChars;
01761
01762     inline int postprocessGetc(int getcResult) {
01763         if (getcResult != EOF)
01764             return getcResult;
01765         else
01766             return EOFC;
01767     }
01768
01769     int getc(FILE *file) {
01770         int c;
01771         int rc;
01772
01773         if (undoChars.empty()) {
01774             c = rc = ::getc(file);
01775         } else {
01776             c = undoChars.back();
01777             undoChars.pop_back();
01778             rc = undoReadChars.back();
01779             undoReadChars.pop_back();
01780         }
01781
01782         if (c == LF)
01783             line++;
01784
01785         readChars.push_back(rc);
01786         return c;
01787     }
01788
01789     int ungetc(int c/*, FILE* file*/) {
01790         if (!readChars.empty()) {
01791             undoReadChars.push_back(readChars.back());
01792             readChars.pop_back();
01793         }
01794         if (c == LF)
01795             line--;
01796         undoChars.push_back(c);
01797         return c;
01798     }
```

```
01799
01800 public:
01801     FileInputStreamReader(std::FILE *file, const std::string &name) : file(file), name(name), line(1)
      {
01802         // No operations.
01803     }
01804
01805     void setTestCase(int testCase) {
01806         if (testCase < 0 || testCase > __TESTLIB_MAX_TEST_CASE)
01807             __testlib_fail(format("testCase expected fit in [1,%d], but %d doesn't",
      __TESTLIB_MAX_TEST_CASE, testCase));
01808         readChars.push_back(testCase + 256);
01809     }
01810
01811     std::vector<int> getReadChars() {
01812         return readChars;
01813     }
01814
01815     int curChar() {
01816         if (feof(file))
01817             return EOFC;
01818         else {
01819             int c = getc(file);
01820             ungetc(c/*, file*/);
01821             return postprocessGetc(c);
01822         }
01823     }
01824
01825     int nextChar() {
01826         if (feof(file))
01827             return EOFC;
01828         else
01829             return postprocessGetc(getc(file));
01830     }
01831
01832     void skipChar() {
01833         getc(file);
01834     }
01835
01836     void unreadChar(int c) {
01837         ungetc(c/*, file*/);
01838     }
01839
01840     std::string getName() {
01841         return name;
01842     }
01843
01844     int getLine() {
01845         return line;
01846     }
01847
01848     bool eof() {
01849         if (NULL == file || feof(file))
01850             return true;
01851         else {
01852             int c = nextChar();
01853             if (c == EOFC || (c == EOF && feof(file)))
01854                 return true;
01855             unreadChar(c);
01856             return false;
01857         }
01858     }
01859
01860     void close() {
01861         if (NULL != file) {
01862             fclose(file);
01863             file = NULL;
01864         }
01865     }
01866 };
01867
01868 class BufferedFileInputStreamReader : public InputStreamReader {
01869 private:
01870     static const size_t BUFFER_SIZE;
01871     static const size_t MAX_UNREAD_COUNT;
01872
01873     std::FILE *file;
01874     std::string name;
01875     int line;
01876
01877     char *buffer;
01878     bool *isEof;
01879     int bufferPos;
01880     size_t bufferSize;
01881
01882     bool refill() {
01883         if (NULL == file)
```

```
01884             __testlib_fail("BufferedFileInputStreamReader: file == NULL (" + getName() + ")");
01885
01886         if (bufferPos >= int(bufferSize)) {
01887             size_t readSize = fread(
01888                     buffer + MAX_UNREAD_COUNT,
01889                     1,
01890                     BUFFER_SIZE - MAX_UNREAD_COUNT,
01891                     file
01892             );
01893
01894             if (readSize < BUFFER_SIZE - MAX_UNREAD_COUNT
01895                 && ferror(file))
01896                 __testlib_fail("BufferedFileInputStreamReader: unable to read (" + getName() + ")");
01897
01898             bufferSize = MAX_UNREAD_COUNT + readSize;
01899             bufferPos = int(MAX_UNREAD_COUNT);
01900             std::memset(isEof + MAX_UNREAD_COUNT, 0, sizeof(isEof[0]) * readSize);
01901
01902             return readSize > 0;
01903         } else
01904             return true;
01905     }
01906
01907     char increment() {
01908         char c;
01909         if ((c = buffer[bufferPos++]) == LF)
01910             line++;
01911         return c;
01912     }
01913
01914 public:
01915     BufferedFileInputStreamReader(std::FILE *file, const std::string &name) : file(file), name(name),
      line(1) {
01916         buffer = new char[BUFFER_SIZE];
01917         isEof = new bool[BUFFER_SIZE];
01918         bufferSize = MAX_UNREAD_COUNT;
01919         bufferPos = int(MAX_UNREAD_COUNT);
01920     }
01921
01922     ~BufferedFileInputStreamReader() {
01923         if (NULL != buffer) {
01924             delete[] buffer;
01925             buffer = NULL;
01926         }
01927         if (NULL != isEof) {
01928             delete[] isEof;
01929             isEof = NULL;
01930         }
01931     }
01932
01933     void setTestCase(int) {
01934         __testlib_fail("setTestCase not implemented in BufferedFileInputStreamReader");
01935     }
01936
01937     std::vector<int> getReadChars() {
01938         __testlib_fail("getReadChars not implemented in BufferedFileInputStreamReader");
01939     }
01940
01941     int curChar() {
01942         if (!refill())
01943             return EOFC;
01944
01945         return isEof[bufferPos] ? EOFC : buffer[bufferPos];
01946     }
01947
01948     int nextChar() {
01949         if (!refill())
01950             return EOFC;
01951
01952         return isEof[bufferPos] ? EOFC : increment();
01953     }
01954
01955     void skipChar() {
01956         increment();
01957     }
01958
01959     void unreadChar(int c) {
01960         bufferPos--;
01961         if (bufferPos < 0)
01962             __testlib_fail("BufferedFileInputStreamReader::unreadChar(int): bufferPos < 0");
01963         isEof[bufferPos] = (c == EOFC);
01964         buffer[bufferPos] = char(c);
01965         if (c == LF)
01966             line--;
01967     }
01968
01969     std::string getName() {
```

```
01970         return name;
01971     }
01972
01973     int getLine() {
01974         return line;
01975     }
01976
01977     bool eof() {
01978         return !refill() || EOFC == curChar();
01979     }
01980
01981     void close() {
01982         if (NULL != file) {
01983             fclose(file);
01984             file = NULL;
01985         }
01986     }
01987 };
01988
01989 const size_t BufferedFileInputStreamReader::BUFFER_SIZE = 2000000;
01990 const size_t BufferedFileInputStreamReader::MAX_UNREAD_COUNT =
      BufferedFileInputStreamReader::BUFFER_SIZE / 2;
01991
01992 /*
01993  * Streams to be used for reading data in checkers or validators.
01994  * Each read*() method moves pointer to the next character after the
01995  * read value.
01996  */
01997 struct InStream {
01998     /* Do not use them. */
01999     InStream();
02000
02001     ~InStream();
02002
02003     /* Wrap std::string with InStream. */
02004     InStream(const InStream &baseStream, std::string content);
02005
02006     InputStreamReader *reader;
02007     int lastLine;
02008
02009     std::string name;
02010     TMode mode;
02011     bool opened;
02012     bool stdfile;
02013     bool strict;
02014
02015     int wordReserveSize;
02016     std::string _tmpReadToken;
02017
02018     int readManyIteration;
02019     size_t maxFileSize;
02020     size_t maxTokenLength;
02021     size_t maxMessageLength;
02022
02023     void init(std::string fileName, TMode mode);
02024
02025     void init(std::FILE *f, TMode mode);
02026
02027     void setTestCase(int testCase);
02028     std::vector<int> getReadChars();
02029
02030     /* Moves stream pointer to the first non-white-space character or EOF. */
02031     void skipBlanks();
02032
02033     /* Returns current character in the stream. Doesn't remove it from stream. */
02034     char curChar();
02035
02036     /* Moves stream pointer one character forward. */
02037     void skipChar();
02038
02039     /* Returns current character and moves pointer one character forward. */
02040     char nextChar();
02041
02042     /* Returns current character and moves pointer one character forward. */
02043     char readChar();
02044
02045     /* As "readChar()" but ensures that the result is equal to given parameter. */
02046     char readChar(char c);
02047
02048     /* As "readChar()" but ensures that the result is equal to the space (code=32). */
02049     char readSpace();
02050
02051     /* Puts back the character into the stream. */
02052     void unreadChar(char c);
02053
02054     /* Reopens stream, you should not use it. */
02055     void reset(std::FILE *file = NULL);
```

```
02056
02057     /* Checks that current position is EOF. If not it doesn’t move stream pointer. */
02058     bool eof();
02059
02060     /* Moves pointer to the first non-white-space character and calls "eof()". */
02061     bool seekEof();
02062
02063     /*
02064      * Checks that current position contains EOLN.
02065      * If not it doesn’t move stream pointer.
02066      * In strict mode expects "#13#10" for windows or "#10" for other platforms.
02067      */
02068     bool eoln();
02069
02070     /* Moves pointer to the first non-space and non-tab character and calls "eoln()". */
02071     bool seekEoln();
02072
02073     /* Moves stream pointer to the first character of the next line (if exists). */
02074     void nextLine();
02075
02076     /*
02077      * Reads new token. Ignores white-spaces into the non-strict mode
02078      * (strict mode is used in validators usually).
02079      */
02080     std::string readWord();
02081
02082     /* The same as "readWord()", it is preferred to use "readToken()". */
02083     std::string readToken();
02084
02085     /* The same as "readWord()", but ensures that token matches to given pattern. */
02086     std::string readWord(const std::string &ptrn, const std::string &variableName = "");
02087
02088     std::string readWord(const pattern &p, const std::string &variableName = "");
02089
02090     std::vector<std::string>
02091     readWords(int size, const std::string &ptrn, const std::string &variablesName = "", int indexBase
      = 1);
02092
02093     std::vector<std::string>
02094     readWords(int size, const pattern &p, const std::string &variablesName = "", int indexBase = 1);
02095
02096     std::vector<std::string> readWords(int size, int indexBase = 1);
02097
02098     /* The same as "readToken()", but ensures that token matches to given pattern. */
02099     std::string readToken(const std::string &ptrn, const std::string &variableName = "");
02100
02101     std::string readToken(const pattern &p, const std::string &variableName = "");
02102
02103     std::vector<std::string>
02104     readTokens(int size, const std::string &ptrn, const std::string &variablesName = "", int indexBase
      = 1);
02105
02106     std::vector<std::string>
02107     readTokens(int size, const pattern &p, const std::string &variablesName = "", int indexBase = 1);
02108
02109     std::vector<std::string> readTokens(int size, int indexBase = 1);
02110
02111     void readWordTo(std::string &result);
02112
02113     void readWordTo(std::string &result, const pattern &p, const std::string &variableName = "");
02114
02115     void readWordTo(std::string &result, const std::string &ptrn, const std::string &variableName =
      "");
02116
02117     void readTokenTo(std::string &result);
02118
02119     void readTokenTo(std::string &result, const pattern &p, const std::string &variableName = "");
02120
02121     void readTokenTo(std::string &result, const std::string &ptrn, const std::string &variableName =
      "");
02122
02123     /*
02124      * Reads new long long value. Ignores white-spaces into the non-strict mode
02125      * (strict mode is used in validators usually).
02126      */
02127     long long readLong();
02128
02129     unsigned long long readUnsignedLong();
02130
02131     /*
02132      * Reads new int. Ignores white-spaces into the non-strict mode
02133      * (strict mode is used in validators usually).
02134      */
02135     int readInteger();
02136
02137     /*
02138      * Reads new int. Ignores white-spaces into the non-strict mode
```

```
02139      * (strict mode is used in validators usually).
02140      */
02141     int readInt();
02142
02143     /* As "readLong()" but ensures that value in the range [minv,maxv]. */
02144     long long readLong(long long minv, long long maxv, const std::string &variableName = "");
02145
02146     /* Reads space-separated sequence of long longs. */
02147     std::vector<long long>
02148     readLongs(int size, long long minv, long long maxv, const std::string &variablesName = "", int
      indexBase = 1);
02149
02150     /* Reads space-separated sequence of long longs. */
02151     std::vector<long long> readLongs(int size, int indexBase = 1);
02152
02153     unsigned long long
02154     readUnsignedLong(unsigned long long minv, unsigned long long maxv, const std::string &variableName
      = "");
02155
02156     std::vector<unsigned long long>
02157     readUnsignedLongs(int size, unsigned long long minv, unsigned long long maxv, const std::string
      &variablesName = "",
02158                       int indexBase = 1);
02159
02160     std::vector<unsigned long long> readUnsignedLongs(int size, int indexBase = 1);
02161
02162     unsigned long long readLong(unsigned long long minv, unsigned long long maxv, const std::string
      &variableName = "");
02163
02164     std::vector<unsigned long long>
02165     readLongs(int size, unsigned long long minv, unsigned long long maxv, const std::string
      &variablesName = "",
02166               int indexBase = 1);
02167
02168     /* As "readInteger()" but ensures that value in the range [minv,maxv]. */
02169     int readInteger(int minv, int maxv, const std::string &variableName = "");
02170
02171     /* As "readInt()" but ensures that value in the range [minv,maxv]. */
02172     int readInt(int minv, int maxv, const std::string &variableName = "");
02173
02174     /* Reads space-separated sequence of integers. */
02175     std::vector<int>
02176     readIntegers(int size, int minv, int maxv, const std::string &variablesName = "", int indexBase =
      1);
02177
02178     /* Reads space-separated sequence of integers. */
02179     std::vector<int> readIntegers(int size, int indexBase = 1);
02180
02181     /* Reads space-separated sequence of integers. */
02182     std::vector<int> readInts(int size, int minv, int maxv, const std::string &variablesName = "", int
      indexBase = 1);
02183
02184     /* Reads space-separated sequence of integers. */
02185     std::vector<int> readInts(int size, int indexBase = 1);
02186
02187     /*
02188      * Reads new double. Ignores white-spaces into the non-strict mode
02189      * (strict mode is used in validators usually).
02190      */
02191     double readReal();
02192
02193     /*
02194      * Reads new double. Ignores white-spaces into the non-strict mode
02195      * (strict mode is used in validators usually).
02196      */
02197     double readDouble();
02198
02199     /* As "readReal()" but ensures that value in the range [minv,maxv]. */
02200     double readReal(double minv, double maxv, const std::string &variableName = "");
02201
02202     std::vector<double>
02203     readReals(int size, double minv, double maxv, const std::string &variablesName = "", int indexBase
      = 1);
02204
02205     std::vector<double> readReals(int size, int indexBase = 1);
02206
02207     /* As "readDouble()" but ensures that value in the range [minv,maxv]. */
02208     double readDouble(double minv, double maxv, const std::string &variableName = "");
02209
02210     std::vector<double>
02211     readDoubles(int size, double minv, double maxv, const std::string &variablesName = "", int
      indexBase = 1);
02212
02213     std::vector<double> readDoubles(int size, int indexBase = 1);
02214
02215     /*
02216      * As "readReal()" but ensures that value in the range [minv,maxv] and
```

```
02217      * number of digit after the decimal point is in range
       [minAfterPointDigitCount,maxAfterPointDigitCount]
02218      * and number is in the form "[-]digit(s)[.digit(s)]".
02219      */
02220     double readStrictReal(double minv, double maxv,
02221                           int minAfterPointDigitCount, int maxAfterPointDigitCount,
02222                           const std::string &variableName = "");
02223
02224     std::vector<double> readStrictReals(int size, double minv, double maxv,
02225                                         int minAfterPointDigitCount, int maxAfterPointDigitCount,
02226                                         const std::string &variablesName = "", int indexBase = 1);
02227
02228     /*
02229      * As "readDouble()" but ensures that value in the range [minv,maxv] and
02230      * number of digit after the decimal point is in range
       [minAfterPointDigitCount,maxAfterPointDigitCount]
02231      * and number is in the form "[-]digit(s)[.digit(s)]".
02232      */
02233     double readStrictDouble(double minv, double maxv,
02234                             int minAfterPointDigitCount, int maxAfterPointDigitCount,
02235                             const std::string &variableName = "");
02236
02237     std::vector<double> readStrictDoubles(int size, double minv, double maxv,
02238                                           int minAfterPointDigitCount, int maxAfterPointDigitCount,
02239                                           const std::string &variablesName = "", int indexBase = 1);
02240
02241     /* As readLine(). */
02242     std::string readString();
02243
02244     /* Read many lines. */
02245     std::vector<std::string> readStrings(int size, int indexBase = 1);
02246
02247     /* See readLine(). */
02248     void readStringTo(std::string &result);
02249
02250     /* The same as "readLine()/readString()", but ensures that line matches to the given pattern. */
02251     std::string readString(const pattern &p, const std::string &variableName = "");
02252
02253     /* The same as "readLine()/readString()", but ensures that line matches to the given pattern. */
02254     std::string readString(const std::string &ptrn, const std::string &variableName = "");
02255
02256     /* Read many lines. */
02257     std::vector<std::string>
02258     readStrings(int size, const pattern &p, const std::string &variableName = "", int indexBase = 1);
02259
02260     /* Read many lines. */
02261     std::vector<std::string>
02262     readStrings(int size, const std::string &ptrn, const std::string &variableName = "", int indexBase
       = 1);
02263
02264     /* The same as "readLine()/readString()", but ensures that line matches to the given pattern. */
02265     void readStringTo(std::string &result, const pattern &p, const std::string &variableName = "");
02266
02267     /* The same as "readLine()/readString()", but ensures that line matches to the given pattern. */
02268     void readStringTo(std::string &result, const std::string &ptrn, const std::string &variableName =
       "");
02269
02270     /*
02271      * Reads line from the current position to EOLN or EOF. Moves stream pointer to
02272      * the first character of the new line (if possible).
02273      */
02274     std::string readLine();
02275
02276     /* Read many lines. */
02277     std::vector<std::string> readLines(int size, int indexBase = 1);
02278
02279     /* See readLine(). */
02280     void readLineTo(std::string &result);
02281
02282     /* The same as "readLine()", but ensures that line matches to the given pattern. */
02283     std::string readLine(const pattern &p, const std::string &variableName = "");
02284
02285     /* The same as "readLine()", but ensures that line matches to the given pattern. */
02286     std::string readLine(const std::string &ptrn, const std::string &variableName = "");
02287
02288     /* Read many lines. */
02289     std::vector<std::string>
02290     readLines(int size, const pattern &p, const std::string &variableName = "", int indexBase = 1);
02291
02292     /* Read many lines. */
02293     std::vector<std::string>
02294     readLines(int size, const std::string &ptrn, const std::string &variableName = "", int indexBase =
       1);
02295
02296     /* The same as "readLine()", but ensures that line matches to the given pattern. */
02297     void readLineTo(std::string &result, const pattern &p, const std::string &variableName = "");
02298
```

```
02299     /* The same as "readLine()", but ensures that line matches to the given pattern. */
02300     void readLineTo(std::string &result, const std::string &ptrn, const std::string &variableName =
    "");
02301
02302     /* Reads EOLN or fails. Use it in validators. Calls "eoln()" method internally. */
02303     void readEoln();
02304
02305     /* Reads EOF or fails. Use it in validators. Calls "eof()" method internally. */
02306     void readEof();
02307
02308     /*
02309      * Quit-functions aborts program with <result> and <message>:
02310      * input/answer streams replace any result to FAIL.
02311      */
02312     NORETURN void quit(TResult result, const char *msg);
02313     /*
02314      * Quit-functions aborts program with <result> and <message>:
02315      * input/answer streams replace any result to FAIL.
02316      */
02317     NORETURN void quitf(TResult result, const char *msg, ...);
02318
02319     /*
02320      * Quit-functions aborts program with <result> and <message>:
02321      * input/answer streams replace any result to FAIL.
02322      */
02323     void quitif(bool condition, TResult result, const char *msg, ...);
02324     /*
02325      * Quit-functions aborts program with <result> and <message>:
02326      * input/answer streams replace any result to FAIL.
02327      */
02328     NORETURN void quits(TResult result, std::string msg);
02329
02330     /*
02331      * Checks condition and aborts a program if condition is false.
02332      * Returns _wa for ouf and _fail on any other streams.
02333      */
02334 #ifdef __GNUC__
02335     __attribute__ ((format (printf, 3, 4)))
02336 #endif
02337     void ensuref(bool cond, const char *format, ...);
02338
02339     void __testlib_ensure(bool cond, std::string message);
02340
02341     void close();
02342
02343     const static int NO_INDEX = INT_MAX;
02344     const static char OPEN_BRACKET = char(11);
02345     const static char CLOSE_BRACKET = char(17);
02346
02347     const static WORD LightGray = 0x07;
02348     const static WORD LightRed = 0x0c;
02349     const static WORD LightCyan = 0x0b;
02350     const static WORD LightGreen = 0x0a;
02351     const static WORD LightYellow = 0x0e;
02352
02353     static void textColor(WORD color);
02354
02355     static void quitscr(WORD color, const char *msg);
02356
02357     static void quitscrS(WORD color, std::string msg);
02358
02359     void xmlSafeWrite(std::FILE *file, const char *msg);
02360
02361     /* Skips UTF-8 Byte Order Mark. */
02362     void skipBom();
02363
02364 private:
02365     InStream(const InStream &);
02366
02367     InStream &operator=(const InStream &);
02368 };
02369
02370 InStream inf;
02371 InStream ouf;
02372 InStream ans;
02373 bool appesMode;
02374 std::string appesModeEncoding = "windows-1251";
02375 std::string resultName;
02376 std::string checkerName = "untitled checker";
02377 random_t rnd;
02378 TTestlibMode testlibMode = _unknown;
02379 double __testlib_points = std::numeric_limits<float>::infinity();
02380
02381 struct ValidatorBoundsHit {
02382     static const double EPS;
02383     bool minHit;
02384     bool maxHit;
```

```
02385
02386     ValidatorBoundsHit(bool minHit = false, bool maxHit = false) : minHit(minHit), maxHit(maxHit) {
02387     };
02388
02389     ValidatorBoundsHit merge(const ValidatorBoundsHit &validatorBoundsHit, bool ignoreMinBound, bool
      ignoreMaxBound) {
02390         return ValidatorBoundsHit(
02391                 __testlib_max(minHit, validatorBoundsHit.minHit) || ignoreMinBound,
02392                 __testlib_max(maxHit, validatorBoundsHit.maxHit) || ignoreMaxBound
02393         );
02394     }
02395 };
02396
02397 const double ValidatorBoundsHit::EPS = 1E-12;
02398
02399 class Validator {
02400 private:
02401     const static std::string TEST_MARKUP_HEADER;
02402     const static std::string TEST_CASE_OPEN_TAG;
02403     const static std::string TEST_CASE_CLOSE_TAG;
02404
02405     bool _initialized;
02406     std::string _testset;
02407     std::string _group;
02408
02409     std::string _testOverviewLogFileName;
02410     std::string _testMarkupFileName;
02411     int _testCase = -1;
02412     std::string _testCaseFileName;
02413
02414     std::map<std::string, ValidatorBoundsHit> _boundsHitByVariableName;
02415     std::set<std::string> _features;
02416     std::set<std::string> _hitFeatures;
02417
02418     bool isVariableNameBoundsAnalyzable(const std::string &variableName) {
02419         for (size_t i = 0; i < variableName.length(); i++)
02420             if ((variableName[i] >= '0' && variableName[i] <= '9') || variableName[i] < ' ')
02421                 return false;
02422         return true;
02423     }
02424
02425     bool isFeatureNameAnalyzable(const std::string &featureName) {
02426         for (size_t i = 0; i < featureName.length(); i++)
02427             if (featureName[i] < ' ')
02428                 return false;
02429         return true;
02430     }
02431
02432 public:
02433     Validator() : _initialized(false), _testset("tests"), _group() {
02434     }
02435
02436     void initialize() {
02437         _initialized = true;
02438     }
02439
02440     std::string testset() const {
02441         if (!_initialized)
02442             __testlib_fail("Validator should be initialized with registerValidation(argc, argv)
      instead of registerValidation() to support validator.testset()");
02443         return _testset;
02444     }
02445
02446     std::string group() const {
02447         if (!_initialized)
02448             __testlib_fail("Validator should be initialized with registerValidation(argc, argv)
      instead of registerValidation() to support validator.group()");
02449         return _group;
02450     }
02451
02452     std::string testOverviewLogFileName() const {
02453         return _testOverviewLogFileName;
02454     }
02455
02456     std::string testMarkupFileName() const {
02457         return _testMarkupFileName;
02458     }
02459
02460     int testCase() const {
02461         return _testCase;
02462     }
02463
02464     std::string testCaseFileName() const {
02465         return _testCaseFileName;
02466     }
02467
02468     void setTestset(const char *const testset) {
```

```
02469         _testset = testset;
02470     }
02471
02472     void setGroup(const char *const group) {
02473         _group = group;
02474     }
02475
02476     void setTestOverviewLogFileName(const char *const testOverviewLogFileName) {
02477         _testOverviewLogFileName = testOverviewLogFileName;
02478     }
02479
02480     void setTestMarkupFileName(const char *const testMarkupFileName) {
02481         _testMarkupFileName = testMarkupFileName;
02482     }
02483
02484     void setTestCase(int testCase) {
02485         _testCase = testCase;
02486     }
02487
02488     void setTestCaseFileName(const char *const testCaseFileName) {
02489         _testCaseFileName = testCaseFileName;
02490     }
02491
02492     std::string prepVariableName(const std::string &variableName) {
02493         if (variableName.length() >= 2 && variableName != "~~") {
02494             if (variableName[0] == '~' && variableName.back() != '~')
02495                 return variableName.substr(1);
02496             if (variableName[0] != '~' && variableName.back() == '~')
02497                 return variableName.substr(0, variableName.length() - 1);
02498             if (variableName[0] == '~' && variableName.back() == '~')
02499                 return variableName.substr(1, variableName.length() - 2);
02500         }
02501         return variableName;
02502     }
02503
02504     bool ignoreMinBound(const std::string &variableName) {
02505         return variableName.length() >= 2 && variableName != "~~" && variableName[0] == '~';
02506     }
02507
02508     bool ignoreMaxBound(const std::string &variableName) {
02509         return variableName.length() >= 2 && variableName != "~~" && variableName.back() == '~';
02510     }
02511
02512     void addBoundsHit(const std::string &variableName, ValidatorBoundsHit boundsHit) {
02513         if (isVariableNameBoundsAnalyzable(variableName)) {
02514             std::string preparedVariableName = prepVariableName(variableName);
02515             _boundsHitByVariableName[preparedVariableName] =
    boundsHit.merge(_boundsHitByVariableName[preparedVariableName],
02516                 ignoreMinBound(variableName), ignoreMaxBound(variableName));
02517         }
02518     }
02519
02520     std::string getBoundsHitLog() {
02521         std::string result;
02522         for (std::map<std::string, ValidatorBoundsHit>::iterator i = _boundsHitByVariableName.begin();
02523              i != _boundsHitByVariableName.end();
02524              i++) {
02525             result += "\"" + i->first + "\":";
02526             if (i->second.minHit)
02527                 result += " min-value-hit";
02528             if (i->second.maxHit)
02529                 result += " max-value-hit";
02530             result += "\n";
02531         }
02532         return result;
02533     }
02534
02535     std::string getFeaturesLog() {
02536         std::string result;
02537         for (std::set<std::string>::iterator i = _features.begin();
02538              i != _features.end();
02539              i++) {
02540             result += "feature \"" + *i + "\":";
02541             if (_hitFeatures.count(*i))
02542                 result += " hit";
02543             result += "\n";
02544         }
02545         return result;
02546     }
02547
02548     void writeTestOverviewLog() {
02549         if (!_testOverviewLogFileName.empty()) {
02550             std::string fileName(_testOverviewLogFileName);
02551             _testOverviewLogFileName = "";
02552
02553             FILE* f;
02554             bool standard_file = false;
```

```
02555             if (fileName == "stdout")
02556                 f = stdout, standard_file = true;
02557             else if (fileName == "stderr")
02558                 f = stderr, standard_file = true;
02559             else {
02560                 f = fopen(fileName.c_str(), "wb");
02561                 if (NULL == f)
02562                     __testlib_fail("Validator::writeTestOverviewLog: can't write test overview log to
    (" + fileName + ")");
02563             }
02564             fprintf(f, "%s%s", getBoundsHitLog().c_str(), getFeaturesLog().c_str());
02565             std::fflush(f);
02566             if (!standard_file)
02567                 if (std::fclose(f))
02568                     __testlib_fail("Validator::writeTestOverviewLog: can't close test overview log
    file (" + fileName + ")");
02569         }
02570     }
02571
02572     void writeTestMarkup() {
02573         if (!_testMarkupFileName.empty()) {
02574             std::vector<int> readChars = inf.getReadChars();
02575             if (!readChars.empty()) {
02576                 std::string markup(TEST_MARKUP_HEADER);
02577                 for (size_t i = 0; i < readChars.size(); i++) {
02578                     int c = readChars[i];
02579                     if (i + 1 == readChars.size() && c == -1)
02580                         continue;
02581                     if (c <= 256) {
02582                         char cc = char(c);
02583                         if (cc == '\\' || cc == '!')
02584                             markup += '\\';
02585                         markup += cc;
02586                     } else {
02587                         markup += TEST_CASE_OPEN_TAG;
02588                         markup += toString(c - 256);
02589                         markup += TEST_CASE_CLOSE_TAG;
02590                     }
02591                 }
02592                 FILE* f;
02593                 bool standard_file = false;
02594                 if (_testMarkupFileName == "stdout")
02595                     f = stdout, standard_file = true;
02596                 else if (_testMarkupFileName == "stderr")
02597                     f = stderr, standard_file = true;
02598                 else {
02599                     f = fopen(_testMarkupFileName.c_str(), "wb");
02600                     if (NULL == f)
02601                         __testlib_fail("Validator::writeTestMarkup: can't write test markup to (" +
    _testMarkupFileName + ")");
02602                 }
02603                 std::fprintf(f, "%s", markup.c_str());
02604                 std::fflush(f);
02605                 if (!standard_file)
02606                     if (std::fclose(f))
02607                         __testlib_fail("Validator::writeTestMarkup: can't close test markup file (" +
    _testCaseFileName + ")");
02608             }
02609         }
02610     }
02611
02612     void writeTestCase() {
02613         if (_testCase > 0) {
02614             std::vector<int> readChars = inf.getReadChars();
02615             if (!readChars.empty()) {
02616                 std::string content, testCaseContent;
02617                 bool matchedTestCase = false;
02618                 for (size_t i = 0; i < readChars.size(); i++) {
02619                     int c = readChars[i];
02620                     if (i + 1 == readChars.size() && c == -1)
02621                         continue;
02622                     if (c <= 256)
02623                         content += char(c);
02624                     else {
02625                         if (matchedTestCase) {
02626                             testCaseContent = content;
02627                             matchedTestCase = false;
02628                         }
02629                         content = "";
02630                         int testCase = c - 256;
02631                         if (testCase == _testCase)
02632                             matchedTestCase = true;
02633                     }
02634                 }
02635                 if (matchedTestCase)
02636                     testCaseContent = content;
02637
```

```
02638                 if (!testCaseContent.empty()) {
02639                     FILE* f;
02640                     bool standard_file = false;
02641                     if (_testCaseFileName.empty() || _testCaseFileName == "stdout")
02642                         f = stdout, standard_file = true;
02643                     else if (_testCaseFileName == "stderr")
02644                         f = stderr, standard_file = true;
02645                     else {
02646                         f = fopen(_testCaseFileName.c_str(), "wb");
02647                         if (NULL == f)
02648                             __testlib_fail("Validator::writeTestCase: can't write test case to (" +
      _testCaseFileName + ")");
02649                     }
02650                     std::fprintf(f, "%s", testCaseContent.c_str());
02651                     std::fflush(f);
02652                     if (!standard_file)
02653                         if (std::fclose(f))
02654                             __testlib_fail("Validator::writeTestCase: can't close test case file (" +
      _testCaseFileName + ")");
02655                 }
02656             }
02657         }
02658     }
02659
02660     void addFeature(const std::string &feature) {
02661         if (_features.count(feature))
02662             __testlib_fail("Feature " + feature + " registered twice.");
02663         if (!isFeatureNameAnalyzable(feature))
02664             __testlib_fail("Feature name '" + feature + "' contains restricted characters.");
02665
02666         _features.insert(feature);
02667     }
02668
02669     void feature(const std::string &feature) {
02670         if (!isFeatureNameAnalyzable(feature))
02671             __testlib_fail("Feature name '" + feature + "' contains restricted characters.");
02672
02673         if (!_features.count(feature))
02674             __testlib_fail("Feature " + feature + " didn't registered via addFeature(feature).");
02675
02676         _hitFeatures.insert(feature);
02677     }
02678 } validator;
02679
02680 const std::string Validator::TEST_MARKUP_HEADER = "MU\xF3\x01";
02681 const std::string Validator::TEST_CASE_OPEN_TAG = "!c";
02682 const std::string Validator::TEST_CASE_CLOSE_TAG = ";";
02683
02684 struct TestlibFinalizeGuard {
02685     static bool alive;
02686     static bool registered;
02687
02688     int quitCount, readEofCount;
02689
02690     TestlibFinalizeGuard() : quitCount(0), readEofCount(0) {
02691         // No operations.
02692     }
02693
02694     ~TestlibFinalizeGuard() {
02695         bool _alive = alive;
02696         alive = false;
02697
02698         if (_alive) {
02699             if (testlibMode == _checker && quitCount == 0)
02700                 __testlib_fail("Checker must end with quit or quitf call.");
02701
02702             if (testlibMode == _validator && readEofCount == 0 && quitCount == 0)
02703                 __testlib_fail("Validator must end with readEof call.");
02704
02705             /* opts */
02706             autoEnsureNoUnusedOpts();
02707
02708             if (!registered)
02709                 __testlib_fail("Call register-function in the first line of the main
      (registerTestlibCmd or other similar)");
02710         }
02711
02712         if (__testlib_exitCode == 0) {
02713             validator.writeTestOverviewLog();
02714             validator.writeTestMarkup();
02715             validator.writeTestCase();
02716         }
02717     }
02718
02719 private:
02720     /* opts */
02721     void autoEnsureNoUnusedOpts();
```

```
02722 };
02723
02724 bool TestlibFinalizeGuard::alive = true;
02725 bool TestlibFinalizeGuard::registered = false;
02726 extern TestlibFinalizeGuard testlibFinalizeGuard;
02727
02728 /*
02729  * Call it to disable checks on finalization.
02730  */
02731 void disableFinalizeGuard() {
02732     TestlibFinalizeGuard::alive = false;
02733 }
02734
02735 /* Interactor streams.
02736  */
02737 std::fstream tout;
02738
02739 /* implementation
02740  */
02741
02742 InStream::InStream() {
02743     reader = NULL;
02744     lastLine = -1;
02745     opened = false;
02746     name = "";
02747     mode = _input;
02748     strict = false;
02749     stdfile = false;
02750     wordReserveSize = 4;
02751     readManyIteration = NO_INDEX;
02752     maxFileSize = 128 * 1024 * 1024; // 128MB.
02753     maxTokenLength = 32 * 1024 * 1024; // 32MB.
02754     maxMessageLength = 32000;
02755 }
02756
02757 InStream::InStream(const InStream &baseStream, std::string content) {
02758     reader = new StringInputStreamReader(content);
02759     lastLine = -1;
02760     opened = true;
02761     strict = baseStream.strict;
02762     stdfile = false;
02763     mode = baseStream.mode;
02764     name = "based on " + baseStream.name;
02765     readManyIteration = NO_INDEX;
02766     maxFileSize = 128 * 1024 * 1024; // 128MB.
02767     maxTokenLength = 32 * 1024 * 1024; // 32MB.
02768     maxMessageLength = 32000;
02769 }
02770
02771 InStream::~InStream() {
02772     if (NULL != reader) {
02773         reader->close();
02774         delete reader;
02775         reader = NULL;
02776     }
02777 }
02778
02779 void InStream::setTestCase(int testCase) {
02780     if (testlibMode != _validator || mode != _input || !stdfile || this != &inf)
02781         __testlib_fail("InStream::setTestCase can be used only for inf in validator-mode."
02782             " Actually, prefer setTestCase function instead of InStream member");
02783     reader->setTestCase(testCase);
02784 }
02785
02786 std::vector<int> InStream::getReadChars() {
02787     if (testlibMode != _validator || mode != _input || !stdfile || this != &inf)
02788         __testlib_fail("InStream::getReadChars can be used only for inf in validator-mode.");
02789     return reader == NULL ? std::vector<int>() : reader->getReadChars();
02790 }
02791
02792 void setTestCase(int testCase) {
02793     static bool first_run = true;
02794     static bool zero_based = false;
02795
02796     if (first_run && testCase == 0)
02797         zero_based = true;
02798
02799     if (zero_based)
02800         testCase++;
02801
02802     __testlib_hasTestCase = true;
02803     __testlib_testCase = testCase;
02804
02805     if (testlibMode == _validator)
02806         inf.setTestCase(testCase);
02807
02808     first_run = false;
```

```
02809 }
02810
02811 #ifdef __GNUC__
02812 __attribute__((const))
02813 #endif
02814 int resultExitCode(TResult r) {
02815     if (r == _ok)
02816         return OK_EXIT_CODE;
02817     if (r == _wa)
02818         return WA_EXIT_CODE;
02819     if (r == _pe)
02820         return PE_EXIT_CODE;
02821     if (r == _fail)
02822         return FAIL_EXIT_CODE;
02823     if (r == _dirt)
02824         return DIRT_EXIT_CODE;
02825     if (r == _points)
02826         return POINTS_EXIT_CODE;
02827     if (r == _unexpected_eof)
02828 #ifdef ENABLE_UNEXPECTED_EOF
02829         return UNEXPECTED_EOF_EXIT_CODE;
02830 #else
02831         return PE_EXIT_CODE;
02832 #endif
02833     if (r >= _partially)
02834         return PC_BASE_EXIT_CODE + (r - _partially);
02835     return FAIL_EXIT_CODE;
02836 }
02837
02838 void InStream::textColor(
02839 #if !(defined(ON_WINDOWS) && (!defined(_MSC_VER) || _MSC_VER > 1400)) && defined(__GNUC__)
02840         __attribute__((unused))
02841 #endif
02842         WORD color
02843 ) {
02844 #if defined(ON_WINDOWS) && (!defined(_MSC_VER) || _MSC_VER > 1400)
02845     HANDLE handle = GetStdHandle(STD_OUTPUT_HANDLE);
02846     SetConsoleTextAttribute(handle, color);
02847 #endif
02848 #if !defined(ON_WINDOWS) && defined(__GNUC__)
02849     if (isatty(2))
02850     {
02851         switch (color)
02852         {
02853         case LightRed:
02854             fprintf(stderr, "\033[1;31m");
02855             break;
02856         case LightCyan:
02857             fprintf(stderr, "\033[1;36m");
02858             break;
02859         case LightGreen:
02860             fprintf(stderr, "\033[1;32m");
02861             break;
02862         case LightYellow:
02863             fprintf(stderr, "\033[1;33m");
02864             break;
02865         case LightGray:
02866         default:
02867             fprintf(stderr, "\033[0m");
02868         }
02869     }
02870 #endif
02871 }
02872
02873 #ifdef TESTLIB_THROW_EXIT_EXCEPTION_INSTEAD_OF_EXIT
02874 class exit_exception: public std::exception {
02875 private:
02876     int exitCode;
02877 public:
02878     exit_exception(int exitCode): exitCode(exitCode) {}
02879     int getExitCode() { return exitCode; }
02880 };
02881 #endif
02882
02883 NORETURN void halt(int exitCode) {
02884 #ifdef FOOTER
02885     InStream::textColor(InStream::LightGray);
02886     std::fprintf(stderr, "Checker: \"%s\"\n", checkerName.c_str());
02887     std::fprintf(stderr, "Exit code: %d\n", exitCode);
02888     InStream::textColor(InStream::LightGray);
02889 #endif
02890     __testlib_exitCode = exitCode;
02891 #ifdef TESTLIB_THROW_EXIT_EXCEPTION_INSTEAD_OF_EXIT
02892     throw exit_exception(exitCode);
02893 #endif
02894     std::exit(exitCode);
02895 }
```

```
02896
02897 static bool __testlib_shouldCheckDirt(TResult result) {
02898     return result == _ok || result == _points || result >= _partially;
02899 }
02900
02901 static std::string __testlib_appendMessage(const std::string &message, const std::string &extra) {
02902     int openPos = -1, closePos = -1;
02903     for (size_t i = 0; i < message.length(); i++) {
02904         if (message[i] == InStream::OPEN_BRACKET) {
02905             if (openPos == -1)
02906                 openPos = int(i);
02907             else
02908                 openPos = INT_MAX;
02909         }
02910         if (message[i] == InStream::CLOSE_BRACKET) {
02911             if (closePos == -1)
02912                 closePos = int(i);
02913             else
02914                 closePos = INT_MAX;
02915         }
02916     }
02917     if (openPos != -1 && openPos != INT_MAX
02918         && closePos != -1 && closePos != INT_MAX
02919         && openPos < closePos) {
02920         size_t index = message.find(extra, openPos);
02921         if (index == std::string::npos || int(index) >= closePos) {
02922             std::string result(message);
02923             result.insert(closePos, ", " + extra);
02924             return result;
02925         }
02926         return message;
02927     }
02928
02929     return message + " " + InStream::OPEN_BRACKET + extra + InStream::CLOSE_BRACKET;
02930 }
02931
02932 static std::string __testlib_toPrintableMessage(const std::string &message) {
02933     int openPos = -1, closePos = -1;
02934     for (size_t i = 0; i < message.length(); i++) {
02935         if (message[i] == InStream::OPEN_BRACKET) {
02936             if (openPos == -1)
02937                 openPos = int(i);
02938             else
02939                 openPos = INT_MAX;
02940         }
02941         if (message[i] == InStream::CLOSE_BRACKET) {
02942             if (closePos == -1)
02943                 closePos = int(i);
02944             else
02945                 closePos = INT_MAX;
02946         }
02947     }
02948     if (openPos != -1 && openPos != INT_MAX
02949         && closePos != -1 && closePos != INT_MAX
02950         && openPos < closePos) {
02951         std::string result(message);
02952         result[openPos] = '(';
02953         result[closePos] = ')';
02954         return result;
02955     }
02956
02957     return message;
02958 }
02959
02960 NORETURN void InStream::quit(TResult result, const char *msg) {
02961     if (TestlibFinalizeGuard::alive)
02962         testlibFinalizeGuard.quitCount++;
02963
02964     std::string message(msg);
02965     message = trim(message);
02966
02967     if (__testlib_hasTestCase) {
02968         if (result != _ok)
02969             message = __testlib_appendMessage(message, "test case " + vtos(__testlib_testCase));
02970         else {
02971             if (__testlib_testCase == 1)
02972                 message = __testlib_appendMessage(message, vtos(__testlib_testCase) + " test case");
02973             else
02974                 message = __testlib_appendMessage(message, vtos(__testlib_testCase) + " test cases");
02975         }
02976     }
02977
02978     // You can change maxMessageLength.
02979     // Example: 'inf.maxMessageLength = 1024 * 1024;'.
02980     if (message.length() > maxMessageLength) {
02981         std::string warn = "message length exceeds " + vtos(maxMessageLength)
02982                            + ", the message is truncated: ";
```

```
02983         message = warn + message.substr(0, maxMessageLength - warn.length());
02984     }
02985
02986 #ifndef ENABLE_UNEXPECTED_EOF
02987     if (result == _unexpected_eof)
02988         result = _pe;
02989 #endif
02990
02991     if (testlibMode == _scorer && result != _fail)
02992         quits(_fail, "Scorer should return points only. Don't use a quit function.");
02993
02994     if (mode != _output && result != _fail) {
02995         if (mode == _input && testlibMode == _validator && lastLine != -1)
02996             quits(_fail, __testlib_appendMessage(__testlib_appendMessage(message, name), "line " +
      vtos(lastLine)));
02997         else
02998             quits(_fail, __testlib_appendMessage(message, name));
02999     }
03000
03001     std::FILE *resultFile;
03002     std::string errorName;
03003
03004     if (__testlib_shouldCheckDirt(result)) {
03005         if (testlibMode != _interactor && !ouf.seekEof())
03006             quit(_dirt, "Extra information in the output file");
03007     }
03008
03009     int pctype = result - _partially;
03010     bool isPartial = false;
03011
03012     switch (result) {
03013         case _ok:
03014             errorName = "ok ";
03015             quitscrS(LightGreen, errorName);
03016             break;
03017         case _wa:
03018             errorName = "wrong answer ";
03019             quitscrS(LightRed, errorName);
03020             break;
03021         case _pe:
03022             errorName = "wrong output format ";
03023             quitscrS(LightRed, errorName);
03024             break;
03025         case _fail:
03026             errorName = "FAIL ";
03027             quitscrS(LightRed, errorName);
03028             break;
03029         case _dirt:
03030             errorName = "wrong output format ";
03031             quitscrS(LightCyan, errorName);
03032             result = _pe;
03033             break;
03034         case _points:
03035             errorName = "points ";
03036             quitscrS(LightYellow, errorName);
03037             break;
03038         case _unexpected_eof:
03039             errorName = "unexpected eof ";
03040             quitscrS(LightCyan, errorName);
03041             break;
03042         default:
03043             if (result >= _partially) {
03044                 errorName = format("partially correct (%d) ", pctype);
03045                 isPartial = true;
03046                 quitscrS(LightYellow, errorName);
03047             } else
03048                 quit(_fail, "What is the code ??? ");
03049     }
03050
03051     if (resultName != "") {
03052         resultFile = std::fopen(resultName.c_str(), "w");
03053         if (resultFile == NULL) {
03054             resultName = "";
03055             quit(_fail, "Can not write to the result file");
03056         }
03057         if (appesMode) {
03058             std::fprintf(resultFile, "<?xml version=\"1.0\" encoding=\"%s\"?>",
      appesModeEncoding.c_str());
03059             if (isPartial)
03060                 std::fprintf(resultFile, "<result outcome = \"%s\" pctype = \"%d\">",
03061                              outcomes[(int) _partially].c_str(), pctype);
03062             else {
03063                 if (result != _points)
03064                     std::fprintf(resultFile, "<result outcome = \"%s\">", outcomes[(int)
      result].c_str());
03065                 else {
03066                     if (__testlib_points == std::numeric_limits<float>::infinity())
```

```
03067                       quit(_fail, "Expected points, but infinity found");
03068                   std::string stringPoints = removeDoubleTrailingZeroes(format("%.10f",
    __testlib_points));
03069                   std::fprintf(resultFile, "<result outcome = \"%s\" points = \"%s\">",
03070                                outcomes[(int) result].c_str(), stringPoints.c_str());
03071               }
03072           }
03073           xmlSafeWrite(resultFile, __testlib_toPrintableMessage(message).c_str());
03074           std::fprintf(resultFile, "</result>\n");
03075       } else
03076           std::fprintf(resultFile, "%s", __testlib_toPrintableMessage(message).c_str());
03077       if (NULL == resultFile || fclose(resultFile) != 0) {
03078           resultName = "";
03079           quit(_fail, "Can not write to the result file");
03080       }
03081   }
03082
03083   quitscr(LightGray, __testlib_toPrintableMessage(message).c_str());
03084   std::fprintf(stderr, "\n");
03085
03086   inf.close();
03087   ouf.close();
03088   ans.close();
03089   if (tout.is_open())
03090       tout.close();
03091
03092   textColor(LightGray);
03093
03094   if (resultName != "")
03095       std::fprintf(stderr, "See file to check exit message\n");
03096
03097   halt(resultExitCode(result));
03098 }
03099
03100 #ifdef __GNUC__
03101 __attribute__ ((format (printf, 3, 4)))
03102 #endif
03103 NORETURN void InStream::quitf(TResult result, const char *msg, ...) {
03104     FMT_TO_RESULT(msg, msg, message);
03105     InStream::quit(result, message.c_str());
03106 }
03107
03108 #ifdef __GNUC__
03109 __attribute__ ((format (printf, 4, 5)))
03110 #endif
03111 void InStream::quitif(bool condition, TResult result, const char *msg, ...) {
03112     if (condition) {
03113         FMT_TO_RESULT(msg, msg, message);
03114         InStream::quit(result, message.c_str());
03115     }
03116 }
03117
03118 NORETURN void InStream::quits(TResult result, std::string msg) {
03119     InStream::quit(result, msg.c_str());
03120 }
03121
03122 void InStream::xmlSafeWrite(std::FILE *file, const char *msg) {
03123     size_t lmsg = strlen(msg);
03124     for (size_t i = 0; i < lmsg; i++) {
03125         if (msg[i] == '&') {
03126             std::fprintf(file, "%s", "&amp;");
03127             continue;
03128         }
03129         if (msg[i] == '<') {
03130             std::fprintf(file, "%s", "&lt;");
03131             continue;
03132         }
03133         if (msg[i] == '>') {
03134             std::fprintf(file, "%s", "&gt;");
03135             continue;
03136         }
03137         if (msg[i] == '"') {
03138             std::fprintf(file, "%s", "&quot;");
03139             continue;
03140         }
03141         if (0 <= msg[i] && msg[i] <= 31) {
03142             std::fprintf(file, "%c", '.');
03143             continue;
03144         }
03145         std::fprintf(file, "%c", msg[i]);
03146     }
03147 }
03148
03149 void InStream::quitscrS(WORD color, std::string msg) {
03150     quitscr(color, msg.c_str());
03151 }
03152
```

```
03153 void InStream::quitscr(WORD color, const char *msg) {
03154     if (resultName == "") {
03155         textColor(color);
03156         std::fprintf(stderr, "%s", msg);
03157         textColor(LightGray);
03158     }
03159 }
03160
03161 void InStream::reset(std::FILE *file) {
03162     if (opened && stdfile)
03163         quit(_fail, "Can't reset standard handle");
03164
03165     if (opened)
03166         close();
03167
03168     if (!stdfile && NULL == file)
03169         if (NULL == (file = std::fopen(name.c_str(), "rb"))) {
03170             if (mode == _output)
03171                 quits(_pe, std::string("Output file not found: \"") + name + "\"");
03172
03173             if (mode == _answer)
03174                 quits(_fail, std::string("Answer file not found: \"") + name + "\"");
03175         }
03176
03177     if (NULL != file) {
03178         opened = true;
03179         __testlib_set_binary(file);
03180
03181         if (stdfile)
03182             reader = new FileInputStreamReader(file, name);
03183         else
03184             reader = new BufferedFileInputStreamReader(file, name);
03185     } else {
03186         opened = false;
03187         reader = NULL;
03188     }
03189 }
03190
03191 void InStream::init(std::string fileName, TMode mode) {
03192     opened = false;
03193     name = fileName;
03194     stdfile = false;
03195     this->mode = mode;
03196
03197     std::ifstream stream;
03198     stream.open(fileName.c_str(), std::ios::in);
03199     if (stream.is_open()) {
03200         std::streampos start = stream.tellg();
03201         stream.seekg(0, std::ios::end);
03202         std::streampos end = stream.tellg();
03203         size_t fileSize = size_t(end - start);
03204         stream.close();
03205
03206         // You can change maxFileSize.
03207         // Example: 'inf.maxFileSize = 256 * 1024 * 1024;'.
03208         if (fileSize > maxFileSize)
03209             quitf(_pe, "File size exceeds %d bytes, size is %d", int(maxFileSize), int(fileSize));
03210     }
03211
03212     reset();
03213 }
03214
03215 void InStream::init(std::FILE *f, TMode mode) {
03216     opened = false;
03217     name = "untitled";
03218     this->mode = mode;
03219
03220     if (f == stdin)
03221         name = "stdin", stdfile = true;
03222     if (f == stdout)
03223         name = "stdout", stdfile = true;
03224     if (f == stderr)
03225         name = "stderr", stdfile = true;
03226
03227     reset(f);
03228 }
03229
03230 void InStream::skipBom() {
03231     const std::string utf8Bom = "\xEF\xBB\xBF";
03232     size_t index = 0;
03233     while (index < utf8Bom.size() && curChar() == utf8Bom[index]) {
03234         index++;
03235         skipChar();
03236     }
03237     if (index < utf8Bom.size()) {
03238         while (index != 0) {
03239             unreadChar(utf8Bom[index - 1]);
```

```
03240             index--;
03241         }
03242     }
03243 }
03244
03245 char InStream::curChar() {
03246     return char(reader->curChar());
03247 }
03248
03249 char InStream::nextChar() {
03250     return char(reader->nextChar());
03251 }
03252
03253 char InStream::readChar() {
03254     return nextChar();
03255 }
03256
03257 char InStream::readChar(char c) {
03258     lastLine = reader->getLine();
03259     char found = readChar();
03260     if (c != found) {
03261         if (!isEoln(found))
03262             quit(_pe, ("Unexpected character '" + std::string(1, found) + "', but '" + std::string(1,
    c) +
03263                        "' expected").c_str());
03264         else
03265             quit(_pe, ("Unexpected character " + ("#" + vtos(int(found))) + ", but '" + std::string(1,
    c) +
03266                        "' expected").c_str());
03267     }
03268     return found;
03269 }
03270
03271 char InStream::readSpace() {
03272     return readChar(' ');
03273 }
03274
03275 void InStream::unreadChar(char c) {
03276     reader->unreadChar(c);
03277 }
03278
03279 void InStream::skipChar() {
03280     reader->skipChar();
03281 }
03282
03283 void InStream::skipBlanks() {
03284     while (isBlanks(reader->curChar()))
03285         reader->skipChar();
03286 }
03287
03288 std::string InStream::readWord() {
03289     readWordTo(_tmpReadToken);
03290     return _tmpReadToken;
03291 }
03292
03293 void InStream::readWordTo(std::string &result) {
03294     if (!strict)
03295         skipBlanks();
03296
03297     lastLine = reader->getLine();
03298     int cur = reader->nextChar();
03299
03300     if (cur == EOFC)
03301         quit(_unexpected_eof, "Unexpected end of file - token expected");
03302
03303     if (isBlanks(cur))
03304         quit(_pe, "Unexpected white-space - token expected");
03305
03306     result.clear();
03307
03308     while (!(isBlanks(cur) || cur == EOFC)) {
03309         result += char(cur);
03310
03311         // You can change maxTokenLength.
03312         // Example: 'inf.maxTokenLength = 128 * 1024 * 1024;'.
03313         if (result.length() > maxTokenLength)
03314             quitf(_pe, "Length of token exceeds %d, token is '%s...'", int(maxTokenLength),
03315                   __testlib_part(result).c_str());
03316
03317         cur = reader->nextChar();
03318     }
03319
03320     reader->unreadChar(cur);
03321
03322     if (result.length() == 0)
03323         quit(_unexpected_eof, "Unexpected end of file or white-space - token expected");
03324 }
```

```
03325
03326 std::string InStream::readToken() {
03327     return readWord();
03328 }
03329
03330 void InStream::readTokenTo(std::string &result) {
03331     readWordTo(result);
03332 }
03333
03334 #ifdef __GNUC__
03335 __attribute__((const))
03336 #endif
03337 static std::string __testlib_part(const std::string &s) {
03338     std::string t;
03339     for (size_t i = 0; i < s.length(); i++)
03340         if (s[i] != '\0')
03341             t += s[i];
03342         else
03343             t += '~';
03344     if (t.length() <= 64)
03345         return t;
03346     else
03347         return t.substr(0, 30) + "..." + t.substr(s.length() - 31, 31);
03348 }
03349
03350 #define __testlib_readMany(readMany, readOne, typeName, space)                  \
03351     if (size < 0)                                                               \
03352         quit(_fail, #readMany ": size should be non-negative.");                \
03353     if (size > 100000000)                                                       \
03354         quit(_fail, #readMany ": size should be at most 100000000.");           \
03355                                                                                 \
03356     std::vector<typeName> result(size);                                         \
03357     readManyIteration = indexBase;                                              \
03358                                                                                 \
03359     for (int i = 0; i < size; i++)                                              \
03360     {                                                                           \
03361         result[i] = readOne;                                                    \
03362         readManyIteration++;                                                    \
03363         if (strict && space && i + 1 < size)                                            \
03364             readSpace();                                                        \
03365     }                                                                           \
03366                                                                                 \
03367     readManyIteration = NO_INDEX;                                               \
03368     return result;                                                              \
03369
03370
03371 std::string InStream::readWord(const pattern &p, const std::string &variableName) {
03372     readWordTo(_tmpReadToken);
03373     if (!p.matches(_tmpReadToken)) {
03374         if (readManyIteration == NO_INDEX) {
03375             if (variableName.empty())
03376                 quit(_wa,
03377                      ("Token \"" + __testlib_part(_tmpReadToken) + "\" doesn't correspond to pattern
      \"" + p.src() +
03378                       "\"").c_str());
03379             else
03380                 quit(_wa, ("Token parameter [name=" + variableName + "] equals to \"" +
      __testlib_part(_tmpReadToken) +
03381                            "\", doesn't correspond to pattern \"" + p.src() + "\"").c_str());
03382         } else {
03383             if (variableName.empty())
03384                 quit(_wa, ("Token element [index=" + vtos(readManyIteration) + "] equals to \"" +
03385                            __testlib_part(_tmpReadToken) + "\" doesn't correspond to pattern \"" +
      p.src() +
03386                            "\"").c_str());
03387             else
03388                 quit(_wa, ("Token element " + variableName + "[" + vtos(readManyIteration) + "] equals
      to \"" +
03389                            __testlib_part(_tmpReadToken) + "\", doesn't correspond to pattern \"" +
      p.src() +
03390                            "\"").c_str());
03391         }
03392     }
03393     return _tmpReadToken;
03394 }
03395
03396 std::vector<std::string>
03397 InStream::readWords(int size, const pattern &p, const std::string &variablesName, int indexBase) {
03398     __testlib_readMany(readWords, readWord(p, variablesName), std::string, true);
03399 }
03400
03401 std::vector<std::string> InStream::readWords(int size, int indexBase) {
03402     __testlib_readMany(readWords, readWord(), std::string, true);
03403 }
03404
03405 std::string InStream::readWord(const std::string &ptrn, const std::string &variableName) {
03406     return readWord(pattern(ptrn), variableName);
```

```
03407 }
03408
03409 std::vector<std::string>
03410 InStream::readWords(int size, const std::string &ptrn, const std::string &variablesName, int
      indexBase) {
03411     pattern p(ptrn);
03412     __testlib_readMany(readWords, readWord(p, variablesName), std::string, true);
03413 }
03414
03415 std::string InStream::readToken(const pattern &p, const std::string &variableName) {
03416     return readWord(p, variableName);
03417 }
03418
03419 std::vector<std::string>
03420 InStream::readTokens(int size, const pattern &p, const std::string &variablesName, int indexBase) {
03421     __testlib_readMany(readTokens, readToken(p, variablesName), std::string, true);
03422 }
03423
03424 std::vector<std::string> InStream::readTokens(int size, int indexBase) {
03425     __testlib_readMany(readTokens, readToken(), std::string, true);
03426 }
03427
03428 std::string InStream::readToken(const std::string &ptrn, const std::string &variableName) {
03429     return readWord(ptrn, variableName);
03430 }
03431
03432 std::vector<std::string>
03433 InStream::readTokens(int size, const std::string &ptrn, const std::string &variablesName, int
      indexBase) {
03434     pattern p(ptrn);
03435     __testlib_readMany(readTokens, readWord(p, variablesName), std::string, true);
03436 }
03437
03438 void InStream::readWordTo(std::string &result, const pattern &p, const std::string &variableName) {
03439     readWordTo(result);
03440     if (!p.matches(result)) {
03441         if (variableName.empty())
03442             quit(_wa, ("Token \"" + __testlib_part(result) + "\" doesn't correspond to pattern \"" +
      p.src() +
03443                        "\"").c_str());
03444         else
03445             quit(_wa, ("Token parameter [name=" + variableName + "] equals to \"" +
      __testlib_part(result) +
03446                        "\", doesn't correspond to pattern \"" + p.src() + "\"").c_str());
03447     }
03448 }
03449
03450 void InStream::readWordTo(std::string &result, const std::string &ptrn, const std::string
      &variableName) {
03451     return readWordTo(result, pattern(ptrn), variableName);
03452 }
03453
03454 void InStream::readTokenTo(std::string &result, const pattern &p, const std::string &variableName) {
03455     return readWordTo(result, p, variableName);
03456 }
03457
03458 void InStream::readTokenTo(std::string &result, const std::string &ptrn, const std::string
      &variableName) {
03459     return readWordTo(result, ptrn, variableName);
03460 }
03461
03462 #ifdef __GNUC__
03463 __attribute__((pure))
03464 #endif
03465 static inline bool equals(long long integer, const char *s) {
03466     if (integer == LLONG_MIN)
03467         return strcmp(s, "-9223372036854775808") == 0;
03468
03469     if (integer == 0LL)
03470         return strcmp(s, "0") == 0;
03471
03472     size_t length = strlen(s);
03473
03474     if (length == 0)
03475         return false;
03476
03477     if (integer < 0 && s[0] != '-')
03478         return false;
03479
03480     if (integer < 0)
03481         s++, length--, integer = -integer;
03482
03483     if (length == 0)
03484         return false;
03485
03486     while (integer > 0) {
03487         int digit = int(integer % 10);
```

```
03488
03489         if (s[length - 1] != '0' + digit)
03490             return false;
03491
03492         length--;
03493         integer /= 10;
03494     }
03495
03496     return length == 0;
03497 }
03498
03499 #ifdef __GNUC__
03500 __attribute__((pure))
03501 #endif
03502 static inline bool equals(unsigned long long integer, const char *s) {
03503     if (integer == ULLONG_MAX)
03504         return strcmp(s, "18446744073709551615") == 0;
03505
03506     if (integer == 0ULL)
03507         return strcmp(s, "0") == 0;
03508
03509     size_t length = strlen(s);
03510
03511     if (length == 0)
03512         return false;
03513
03514     while (integer > 0) {
03515         int digit = int(integer % 10);
03516
03517         if (s[length - 1] != '0' + digit)
03518             return false;
03519
03520         length--;
03521         integer /= 10;
03522     }
03523
03524     return length == 0;
03525 }
03526
03527 static inline double stringToDouble(InStream &in, const char *buffer) {
03528     double result;
03529
03530     size_t length = strlen(buffer);
03531
03532     int minusCount = 0;
03533     int plusCount = 0;
03534     int decimalPointCount = 0;
03535     int digitCount = 0;
03536     int eCount = 0;
03537
03538     for (size_t i = 0; i < length; i++) {
03539         if (('0' <= buffer[i] && buffer[i] <= '9') || buffer[i] == '.'
03540             || buffer[i] == 'e' || buffer[i] == 'E'
03541             || buffer[i] == '-' || buffer[i] == '+') {
03542             if ('0' <= buffer[i] && buffer[i] <= '9')
03543                 digitCount++;
03544             if (buffer[i] == 'e' || buffer[i] == 'E')
03545                 eCount++;
03546             if (buffer[i] == '-')
03547                 minusCount++;
03548             if (buffer[i] == '+')
03549                 plusCount++;
03550             if (buffer[i] == '.')
03551                 decimalPointCount++;
03552         } else
03553             in.quit(_pe, ("Expected double, but \"" + __testlib_part(buffer) + "\" found").c_str());
03554     }
03555
03556     // If for sure is not a number in standard notation or in e-notation.
03557     if (digitCount == 0 || minusCount > 2 || plusCount > 2 || decimalPointCount > 1 || eCount > 1)
03558         in.quit(_pe, ("Expected double, but \"" + __testlib_part(buffer) + "\" found").c_str());
03559
03560     char *suffix = new char[length + 1];
03561     std::memset(suffix, 0, length + 1);
03562     int scanned = std::sscanf(buffer, "%lf%s", &result, suffix);
03563     bool empty = strlen(suffix) == 0;
03564     delete[] suffix;
03565
03566     if (scanned == 1 || (scanned == 2 && empty)) {
03567         if (__testlib_isNaN(result))
03568             in.quit(_pe, ("Expected double, but \"" + __testlib_part(buffer) + "\" found").c_str());
03569         return result;
03570     } else
03571         in.quit(_pe, ("Expected double, but \"" + __testlib_part(buffer) + "\" found").c_str());
03572 }
03573
03574 static inline double stringToDouble(InStream &in, const std::string& buffer) {
```

```
03575     for (size_t i = 0; i < buffer.length(); i++)
03576         if (buffer[i] == '\0')
03577             in.quit(_pe, ("Expected double, but \"" + __testlib_part(buffer) + "\" found (it contains
      \\0)").c_str());
03578     return stringToDouble(in, buffer.c_str());
03579 }
03580
03581 static inline double stringToStrictDouble(InStream &in, const char *buffer,
03582         int minAfterPointDigitCount, int maxAfterPointDigitCount) {
03583     if (minAfterPointDigitCount < 0)
03584         in.quit(_fail, "stringToStrictDouble: minAfterPointDigitCount should be non-negative.");
03585
03586     if (minAfterPointDigitCount > maxAfterPointDigitCount)
03587         in.quit(_fail,
03588                 "stringToStrictDouble: minAfterPointDigitCount should be less or equal to
      maxAfterPointDigitCount.");
03589
03590     double result;
03591
03592     size_t length = strlen(buffer);
03593
03594     if (length == 0 || length > 1000)
03595         in.quit(_pe, ("Expected strict double, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03596
03597     if (buffer[0] != '-' && (buffer[0] < '0' || buffer[0] > '9'))
03598         in.quit(_pe, ("Expected strict double, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03599
03600     int pointPos = -1;
03601     for (size_t i = 1; i + 1 < length; i++) {
03602         if (buffer[i] == '.') {
03603             if (pointPos > -1)
03604                 in.quit(_pe, ("Expected strict double, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03605             pointPos = int(i);
03606         }
03607         if (buffer[i] != '.' && (buffer[i] < '0' || buffer[i] > '9'))
03608             in.quit(_pe, ("Expected strict double, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03609     }
03610
03611     if (buffer[length - 1] < '0' || buffer[length - 1] > '9')
03612         in.quit(_pe, ("Expected strict double, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03613
03614     int afterDigitsCount = (pointPos == -1 ? 0 : int(length) - pointPos - 1);
03615     if (afterDigitsCount < minAfterPointDigitCount || afterDigitsCount > maxAfterPointDigitCount)
03616         in.quit(_pe, ("Expected strict double with number of digits after point in range ["
03617                       + vtos(minAfterPointDigitCount)
03618                       + ","
03619                       + vtos(maxAfterPointDigitCount)
03620                       + "], but \"" + __testlib_part(buffer) + "\" found").c_str()
03621         );
03622
03623     int firstDigitPos = -1;
03624     for (size_t i = 0; i < length; i++)
03625         if (buffer[i] >= '0' && buffer[i] <= '9') {
03626             firstDigitPos = int(i);
03627             break;
03628         }
03629
03630     if (firstDigitPos > 1 || firstDigitPos == -1)
03631         in.quit(_pe, ("Expected strict double, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03632
03633     if (buffer[firstDigitPos] == '0' && firstDigitPos + 1 < int(length)
03634         && buffer[firstDigitPos + 1] >= '0' && buffer[firstDigitPos + 1] <= '9')
03635         in.quit(_pe, ("Expected strict double, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03636
03637     char *suffix = new char[length + 1];
03638     std::memset(suffix, 0, length + 1);
03639     int scanned = std::sscanf(buffer, "%lf%s", &result, suffix);
03640     bool empty = strlen(suffix) == 0;
03641     delete[] suffix;
03642
03643     if (scanned == 1 || (scanned == 2 && empty)) {
03644         if (__testlib_isNaN(result) || __testlib_isInfinite(result))
03645             in.quit(_pe, ("Expected double, but \"" + __testlib_part(buffer) + "\" found").c_str());
03646         if (buffer[0] == '-' && result >= 0)
03647             in.quit(_pe, ("Redundant minus in \"" + __testlib_part(buffer) + "\" found").c_str());
03648         return result;
03649     } else
03650         in.quit(_pe, ("Expected double, but \"" + __testlib_part(buffer) + "\" found").c_str());
03651 }
03652
```

```
03653 static inline double stringToStrictDouble(InStream &in, const std::string& buffer,
03654         int minAfterPointDigitCount, int maxAfterPointDigitCount) {
03655     for (size_t i = 0; i < buffer.length(); i++)
03656         if (buffer[i] == '\0')
03657             in.quit(_pe, ("Expected double, but \"" + __testlib_part(buffer) + "\" found (it contains
      \\0)").c_str());
03658     return stringToStrictDouble(in, buffer.c_str(), minAfterPointDigitCount, maxAfterPointDigitCount);
03659 }
03660
03661 static inline long long stringToLongLong(InStream &in, const char *buffer) {
03662     size_t length = strlen(buffer);
03663     if (length == 0 || length > 20)
03664         in.quit(_pe, ("Expected integer, but \"" + __testlib_part(buffer) + "\" found").c_str());
03665
03666     bool has_minus = (length > 1 && buffer[0] == '-');
03667     int zeroes = 0;
03668     bool processingZeroes = true;
03669
03670     for (int i = (has_minus ? 1 : 0); i < int(length); i++) {
03671         if (buffer[i] == '0' && processingZeroes)
03672             zeroes++;
03673         else
03674             processingZeroes = false;
03675
03676         if (buffer[i] < '0' || buffer[i] > '9')
03677             in.quit(_pe, ("Expected integer, but \"" + __testlib_part(buffer) + "\" found").c_str());
03678     }
03679
03680     long long int result;
03681     try {
03682         result = std::stoll(buffer);
03683     } catch (const std::exception&) {
03684         in.quit(_pe, ("Expected integer, but \"" + __testlib_part(buffer) + "\" found").c_str());
03685     } catch (...) {
03686         in.quit(_pe, ("Expected integer, but \"" + __testlib_part(buffer) + "\" found").c_str());
03687     }
03688
03689     if ((zeroes > 0 && (result != 0 || has_minus)) || zeroes > 1)
03690         in.quit(_pe, ("Expected integer, but \"" + __testlib_part(buffer) + "\" found").c_str());
03691
03692     return result;
03693 }
03694
03695 static inline long long stringToLongLong(InStream &in, const std::string& buffer) {
03696     for (size_t i = 0; i < buffer.length(); i++)
03697         if (buffer[i] == '\0')
03698             in.quit(_pe, ("Expected integer, but \"" + __testlib_part(buffer) + "\" found (it contains
      \\0)").c_str());
03699     return stringToLongLong(in, buffer.c_str());
03700 }
03701
03702 static inline unsigned long long stringToUnsignedLongLong(InStream &in, const char *buffer) {
03703     size_t length = strlen(buffer);
03704
03705     if (length == 0 || length > 20)
03706         in.quit(_pe, ("Expected unsigned integer, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03707     if (length > 1 && buffer[0] == '0')
03708         in.quit(_pe, ("Expected unsigned integer, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03709
03710     for (int i = 0; i < int(length); i++) {
03711         if (buffer[i] < '0' || buffer[i] > '9')
03712             in.quit(_pe, ("Expected unsigned integer, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03713     }
03714
03715     unsigned long long result;
03716     try {
03717         result = std::stoull(buffer);
03718     } catch (const std::exception&) {
03719         in.quit(_pe, ("Expected unsigned integer, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03720     } catch (...) {
03721         in.quit(_pe, ("Expected unsigned integer, but \"" + __testlib_part(buffer) + "\"
      found").c_str());
03722     }
03723
03724     return result;
03725 }
03726
03727 static inline long long stringToUnsignedLongLong(InStream &in, const std::string& buffer) {
03728     for (size_t i = 0; i < buffer.length(); i++)
03729         if (buffer[i] == '\0')
03730             in.quit(_pe, ("Expected unsigned integer, but \"" + __testlib_part(buffer) + "\" found (it
      contains \\0)").c_str());
03731     return stringToUnsignedLongLong(in, buffer.c_str());
```

```
03732 }
03733
03734 int InStream::readInteger() {
03735     if (!strict && seekEof())
03736         quit(_unexpected_eof, "Unexpected end of file - int32 expected");
03737
03738     readWordTo(_tmpReadToken);
03739
03740     long long value = stringToLongLong(*this, _tmpReadToken);
03741     if (value < INT_MIN || value > INT_MAX)
03742         quit(_pe, ("Expected int32, but \"" + __testlib_part(_tmpReadToken) + "\" found").c_str());
03743
03744     return int(value);
03745 }
03746
03747 long long InStream::readLong() {
03748     if (!strict && seekEof())
03749         quit(_unexpected_eof, "Unexpected end of file - int64 expected");
03750
03751     readWordTo(_tmpReadToken);
03752
03753     return stringToLongLong(*this, _tmpReadToken);
03754 }
03755
03756 unsigned long long InStream::readUnsignedLong() {
03757     if (!strict && seekEof())
03758         quit(_unexpected_eof, "Unexpected end of file - int64 expected");
03759
03760     readWordTo(_tmpReadToken);
03761
03762     return stringToUnsignedLongLong(*this, _tmpReadToken);
03763 }
03764
03765 long long InStream::readLong(long long minv, long long maxv, const std::string &variableName) {
03766     long long result = readLong();
03767
03768     if (result < minv || result > maxv) {
03769         if (readManyIteration == NO_INDEX) {
03770             if (variableName.empty())
03771                 quit(_wa, ("Integer " + vtos(result) + " violates the range [" +
      toHumanReadableString(minv) + ", " + toHumanReadableString(maxv) +
03772                            "]").c_str());
03773             else
03774                 quit(_wa, ("Integer parameter [name=" + std::string(variableName) + "] equals to " +
      vtos(result) +
03775                            ", violates the range [" + toHumanReadableString(minv) + ", " +
      toHumanReadableString(maxv) + "]").c_str());
03776         } else {
03777             if (variableName.empty())
03778                 quit(_wa, ("Integer element [index=" + vtos(readManyIteration) + "] equals to " +
      vtos(result) +
03779                            ", violates the range [" + toHumanReadableString(minv) + ", " +
      toHumanReadableString(maxv) + "]").c_str());
03780             else
03781                 quit(_wa,
03782                      ("Integer element " + std::string(variableName) + "[" + vtos(readManyIteration) +
      "] equals to " +
03783                       vtos(result) + ", violates the range [" + toHumanReadableString(minv) + ", " +
      toHumanReadableString(maxv) + "]").c_str());
03784         }
03785     }
03786
03787     if (strict && !variableName.empty())
03788         validator.addBoundsHit(variableName, ValidatorBoundsHit(minv == result, maxv == result));
03789
03790     return result;
03791 }
03792
03793 std::vector<long long>
03794 InStream::readLongs(int size, long long minv, long long maxv, const std::string &variablesName, int
      indexBase) {
03795     __testlib_readMany(readLongs, readLong(minv, maxv, variablesName), long long, true)
03796 }
03797
03798 std::vector<long long> InStream::readLongs(int size, int indexBase) {
03799     __testlib_readMany(readLongs, readLong(), long long, true)
03800 }
03801
03802 unsigned long long
03803 InStream::readUnsignedLong(unsigned long long minv, unsigned long long maxv, const std::string
      &variableName) {
03804     unsigned long long result = readUnsignedLong();
03805
03806     if (result < minv || result > maxv) {
03807         if (readManyIteration == NO_INDEX) {
03808             if (variableName.empty())
03809                 quit(_wa,
```

```
03810                     ("Unsigned integer " + vtos(result) + " violates the range [" +
      toHumanReadableString(minv) + ", " + toHumanReadableString(maxv) +
03811                      "]").c_str());
03812             else
03813                 quit(_wa,
03814                     ("Unsigned integer parameter [name=" + std::string(variableName) + "] equals to "
      + vtos(result) +
03815                      ", violates the range [" + toHumanReadableString(minv) + ", " +
      toHumanReadableString(maxv) + "]").c_str());
03816         } else {
03817             if (variableName.empty())
03818                 quit(_wa,
03819                     ("Unsigned integer element [index=" + vtos(readManyIteration) + "] equals to " +
      vtos(result) +
03820                      ", violates the range [" + toHumanReadableString(minv) + ", " +
      toHumanReadableString(maxv) + "]").c_str());
03821             else
03822                 quit(_wa, ("Unsigned integer element " + std::string(variableName) + "[" +
      vtos(readManyIteration) +
03823                            "] equals to " + vtos(result) + ", violates the range [" +
      toHumanReadableString(minv) + ", " + toHumanReadableString(maxv) +
03824                            "]").c_str());
03825         }
03826     }
03827
03828     if (strict && !variableName.empty())
03829         validator.addBoundsHit(variableName, ValidatorBoundsHit(minv == result, maxv == result));
03830
03831     return result;
03832 }
03833
03834 std::vector<unsigned long long> InStream::readUnsignedLongs(int size, unsigned long long minv,
      unsigned long long maxv,
03835                                                             const std::string &variablesName, int
      indexBase) {
03836     __testlib_readMany(readUnsignedLongs, readUnsignedLong(minv, maxv, variablesName), unsigned long
      long, true)
03837 }
03838
03839 std::vector<unsigned long long> InStream::readUnsignedLongs(int size, int indexBase) {
03840     __testlib_readMany(readUnsignedLongs, readUnsignedLong(), unsigned long long, true)
03841 }
03842
03843 unsigned long long
03844 InStream::readLong(unsigned long long minv, unsigned long long maxv, const std::string &variableName)
      {
03845     return readUnsignedLong(minv, maxv, variableName);
03846 }
03847
03848 int InStream::readInt() {
03849     return readInteger();
03850 }
03851
03852 int InStream::readInt(int minv, int maxv, const std::string &variableName) {
03853     int result = readInt();
03854
03855     if (result < minv || result > maxv) {
03856         if (readManyIteration == NO_INDEX) {
03857             if (variableName.empty())
03858                 quit(_wa, ("Integer " + vtos(result) + " violates the range [" +
      toHumanReadableString(minv) + ", " + toHumanReadableString(maxv) +
03859                            "]").c_str());
03860             else
03861                 quit(_wa, ("Integer parameter [name=" + std::string(variableName) + "] equals to " +
      vtos(result) +
03862                            ", violates the range [" + toHumanReadableString(minv) + ", " +
      toHumanReadableString(maxv) + "]").c_str());
03863         } else {
03864             if (variableName.empty())
03865                 quit(_wa, ("Integer element [index=" + vtos(readManyIteration) + "] equals to " +
      vtos(result) +
03866                            ", violates the range [" + toHumanReadableString(minv) + ", " +
      toHumanReadableString(maxv) + "]").c_str());
03867             else
03868                 quit(_wa,
03869                     ("Integer element " + std::string(variableName) + "[" + vtos(readManyIteration) +
      "] equals to " +
03870                      vtos(result) + ", violates the range [" + toHumanReadableString(minv) + ", " +
      toHumanReadableString(maxv) + "]").c_str());
03871         }
03872     }
03873
03874     if (strict && !variableName.empty())
03875         validator.addBoundsHit(variableName, ValidatorBoundsHit(minv == result, maxv == result));
03876
03877     return result;
03878 }
```

```
03879
03880 int InStream::readInteger(int minv, int maxv, const std::string &variableName) {
03881     return readInt(minv, maxv, variableName);
03882 }
03883
03884 std::vector<int> InStream::readInts(int size, int minv, int maxv, const std::string &variablesName,
      int indexBase) {
03885     __testlib_readMany(readInts, readInt(minv, maxv, variablesName), int, true)
03886 }
03887
03888 std::vector<int> InStream::readInts(int size, int indexBase) {
03889     __testlib_readMany(readInts, readInt(), int, true)
03890 }
03891
03892 std::vector<int> InStream::readIntegers(int size, int minv, int maxv, const std::string
      &variablesName, int indexBase) {
03893     __testlib_readMany(readIntegers, readInt(minv, maxv, variablesName), int, true)
03894 }
03895
03896 std::vector<int> InStream::readIntegers(int size, int indexBase) {
03897     __testlib_readMany(readIntegers, readInt(), int, true)
03898 }
03899
03900 double InStream::readReal() {
03901     if (!strict && seekEof())
03902         quit(_unexpected_eof, "Unexpected end of file - double expected");
03903
03904     return stringToDouble(*this, readWord());
03905 }
03906
03907 double InStream::readDouble() {
03908     return readReal();
03909 }
03910
03911 double InStream::readReal(double minv, double maxv, const std::string &variableName) {
03912     double result = readReal();
03913
03914     if (result < minv || result > maxv) {
03915         if (readManyIteration == NO_INDEX) {
03916             if (variableName.empty())
03917                 quit(_wa, ("Double " + vtos(result) + " violates the range [" + vtos(minv) + ", " +
      vtos(maxv) +
03918                            "]").c_str());
03919             else
03920                 quit(_wa, ("Double parameter [name=" + std::string(variableName) + "] equals to " +
      vtos(result) +
03921                            ", violates the range [" + vtos(minv) + ", " + vtos(maxv) + "]").c_str());
03922         } else {
03923             if (variableName.empty())
03924                 quit(_wa, ("Double element [index=" + vtos(readManyIteration) + "] equals to " +
      vtos(result) +
03925                            ", violates the range [" + vtos(minv) + ", " + vtos(maxv) + "]").c_str());
03926             else
03927                 quit(_wa,
03928                      ("Double element " + std::string(variableName) + "[" + vtos(readManyIteration) +
      "] equals to " +
03929                       vtos(result) + ", violates the range [" + vtos(minv) + ", " + vtos(maxv) +
      "]").c_str());
03930         }
03931     }
03932
03933     if (strict && !variableName.empty())
03934         validator.addBoundsHit(variableName, ValidatorBoundsHit(
03935                 doubleDelta(minv, result) < ValidatorBoundsHit::EPS,
03936                 doubleDelta(maxv, result) < ValidatorBoundsHit::EPS
03937         ));
03938
03939     return result;
03940 }
03941
03942 std::vector<double>
03943 InStream::readReals(int size, double minv, double maxv, const std::string &variablesName, int
      indexBase) {
03944     __testlib_readMany(readReals, readReal(minv, maxv, variablesName), double, true)
03945 }
03946
03947 std::vector<double> InStream::readReals(int size, int indexBase) {
03948     __testlib_readMany(readReals, readReal(), double, true)
03949 }
03950
03951 double InStream::readDouble(double minv, double maxv, const std::string &variableName) {
03952     return readReal(minv, maxv, variableName);
03953 }
03954
03955 std::vector<double>
03956 InStream::readDoubles(int size, double minv, double maxv, const std::string &variablesName, int
      indexBase) {
```

```
03957     __testlib_readMany(readDoubles, readDouble(minv, maxv, variablesName), double, true)
03958 }
03959
03960 std::vector<double> InStream::readDoubles(int size, int indexBase) {
03961     __testlib_readMany(readDoubles, readDouble(), double, true)
03962 }
03963
03964 double InStream::readStrictReal(double minv, double maxv,
03965                                 int minAfterPointDigitCount, int maxAfterPointDigitCount,
03966                                 const std::string &variableName) {
03967     if (!strict && seekEof())
03968         quit(_unexpected_eof, "Unexpected end of file - strict double expected");
03969
03970     double result = stringToStrictDouble(*this, readWord(), minAfterPointDigitCount,
      maxAfterPointDigitCount);
03971
03972     if (result < minv || result > maxv) {
03973         if (readManyIteration == NO_INDEX) {
03974             if (variableName.empty())
03975                 quit(_wa, ("Strict double " + vtos(result) + " violates the range [" + vtos(minv) + ",
      " + vtos(maxv) +
03976                         "]").c_str());
03977             else
03978                 quit(_wa,
03979                      ("Strict double parameter [name=" + std::string(variableName) + "] equals to " +
      vtos(result) +
03980                       ", violates the range [" + vtos(minv) + ", " + vtos(maxv) + "]").c_str());
03981         } else {
03982             if (variableName.empty())
03983                 quit(_wa, ("Strict double element [index=" + vtos(readManyIteration) + "] equals to "
      + vtos(result) +
03984                         ", violates the range [" + vtos(minv) + ", " + vtos(maxv) + "]").c_str());
03985             else
03986                 quit(_wa, ("Strict double element " + std::string(variableName) + "[" +
      vtos(readManyIteration) +
03987                         "] equals to " + vtos(result) + ", violates the range [" + vtos(minv) + ",
      " + vtos(maxv) +
03988                         "]").c_str());
03989         }
03990     }
03991
03992     if (strict && !variableName.empty())
03993         validator.addBoundsHit(variableName, ValidatorBoundsHit(
03994                 doubleDelta(minv, result) < ValidatorBoundsHit::EPS,
03995                 doubleDelta(maxv, result) < ValidatorBoundsHit::EPS
03996         ));
03997
03998     return result;
03999 }
04000
04001 std::vector<double> InStream::readStrictReals(int size, double minv, double maxv,
04002                                               int minAfterPointDigitCount, int
      maxAfterPointDigitCount,
04003                                               const std::string &variablesName, int indexBase) {
04004     __testlib_readMany(readStrictReals,
04005                        readStrictReal(minv, maxv, minAfterPointDigitCount, maxAfterPointDigitCount,
      variablesName),
04006                        double, true)
04007 }
04008
04009 double InStream::readStrictDouble(double minv, double maxv,
04010                                   int minAfterPointDigitCount, int maxAfterPointDigitCount,
04011                                   const std::string &variableName) {
04012     return readStrictReal(minv, maxv,
04013                           minAfterPointDigitCount, maxAfterPointDigitCount,
04014                           variableName);
04015 }
04016
04017 std::vector<double> InStream::readStrictDoubles(int size, double minv, double maxv,
04018                                                 int minAfterPointDigitCount, int
      maxAfterPointDigitCount,
04019                                                 const std::string &variablesName, int indexBase) {
04020     __testlib_readMany(readStrictDoubles,
04021                        readStrictDouble(minv, maxv, minAfterPointDigitCount, maxAfterPointDigitCount,
      variablesName),
04022                        double, true)
04023 }
04024
04025 bool InStream::eof() {
04026     if (!strict && NULL == reader)
04027         return true;
04028
04029     return reader->eof();
04030 }
04031
04032 bool InStream::seekEof() {
04033     if (!strict && NULL == reader)
```

```
04034         return true;
04035     skipBlanks();
04036     return eof();
04037 }
04038
04039 bool InStream::eoln() {
04040     if (!strict && NULL == reader)
04041         return true;
04042
04043     int c = reader->nextChar();
04044
04045     if (!strict) {
04046         if (c == EOFC)
04047             return true;
04048
04049         if (c == CR) {
04050             c = reader->nextChar();
04051
04052             if (c != LF) {
04053                 reader->unreadChar(c);
04054                 reader->unreadChar(CR);
04055                 return false;
04056             } else
04057                 return true;
04058         }
04059
04060         if (c == LF)
04061             return true;
04062
04063         reader->unreadChar(c);
04064         return false;
04065     } else {
04066         bool returnCr = false;
04067
04068 #if (defined(ON_WINDOWS) && !defined(FOR_LINUX)) || defined(FOR_WINDOWS)
04069         if (c != CR) {
04070             reader->unreadChar(c);
04071             return false;
04072         } else {
04073             if (!returnCr)
04074                 returnCr = true;
04075             c = reader->nextChar();
04076         }
04077 #endif
04078         if (c != LF) {
04079             reader->unreadChar(c);
04080             if (returnCr)
04081                 reader->unreadChar(CR);
04082             return false;
04083         }
04084
04085         return true;
04086     }
04087 }
04088
04089 void InStream::readEoln() {
04090     lastLine = reader->getLine();
04091     if (!eoln())
04092         quit(_pe, "Expected EOLN");
04093 }
04094
04095 void InStream::readEof() {
04096     lastLine = reader->getLine();
04097     if (!eof())
04098         quit(_pe, "Expected EOF");
04099
04100     if (TestlibFinalizeGuard::alive && this == &inf)
04101         testlibFinalizeGuard.readEofCount++;
04102 }
04103
04104 bool InStream::seekEoln() {
04105     if (!strict && NULL == reader)
04106         return true;
04107
04108     int cur;
04109     do {
04110         cur = reader->nextChar();
04111     } while (cur == SPACE || cur == TAB);
04112
04113     reader->unreadChar(cur);
04114     return eoln();
04115 }
04116
04117 void InStream::nextLine() {
04118     readLine();
04119 }
04120
```

```
04121 void InStream::readStringTo(std::string &result) {
04122     if (NULL == reader)
04123         quit(_pe, "Expected line");
04124 
04125     result.clear();
04126 
04127     for (;;) {
04128         int cur = reader->curChar();
04129 
04130         if (cur == LF || cur == EOFC)
04131             break;
04132 
04133         if (cur == CR) {
04134             cur = reader->nextChar();
04135             if (reader->curChar() == LF) {
04136                 reader->unreadChar(cur);
04137                 break;
04138             }
04139         }
04140 
04141         lastLine = reader->getLine();
04142         result += char(reader->nextChar());
04143     }
04144 
04145     if (strict)
04146         readEoln();
04147     else
04148         eoln();
04149 }
04150 
04151 std::string InStream::readString() {
04152     readStringTo(_tmpReadToken);
04153     return _tmpReadToken;
04154 }
04155 
04156 std::vector<std::string> InStream::readStrings(int size, int indexBase) {
04157     __testlib_readMany(readStrings, readString(), std::string, false)
04158 }
04159 
04160 void InStream::readStringTo(std::string &result, const pattern &p, const std::string &variableName) {
04161     readStringTo(result);
04162     if (!p.matches(result)) {
04163         if (readManyIteration == NO_INDEX) {
04164             if (variableName.empty())
04165                 quit(_wa, ("Line \"" + __testlib_part(result) + "\" doesn't correspond to pattern \""
      + p.src() +
04166                            "\"").c_str());
04167             else
04168                 quit(_wa, ("Line [name=" + variableName + "] equals to \"" + __testlib_part(result) +
04169                            "\", doesn't correspond to pattern \"" + p.src() + "\"").c_str());
04170         } else {
04171             if (variableName.empty())
04172                 quit(_wa,
04173                      ("Line element [index=" + vtos(readManyIteration) + "] equals to \"" +
      __testlib_part(result) +
04174                       "\" doesn't correspond to pattern \"" + p.src() + "\"").c_str());
04175             else
04176                 quit(_wa,
04177                      ("Line element " + std::string(variableName) + "[" + vtos(readManyIteration) + "]
      equals to \"" +
04178                       __testlib_part(result) + "\", doesn't correspond to pattern \"" + p.src() +
      "\"").c_str());
04179         }
04180     }
04181 }
04182 
04183 void InStream::readStringTo(std::string &result, const std::string &ptrn, const std::string
      &variableName) {
04184     readStringTo(result, pattern(ptrn), variableName);
04185 }
04186 
04187 std::string InStream::readString(const pattern &p, const std::string &variableName) {
04188     readStringTo(_tmpReadToken, p, variableName);
04189     return _tmpReadToken;
04190 }
04191 
04192 std::vector<std::string>
04193 InStream::readStrings(int size, const pattern &p, const std::string &variablesName, int indexBase) {
04194     __testlib_readMany(readStrings, readString(p, variablesName), std::string, false)
04195 }
04196 
04197 std::string InStream::readString(const std::string &ptrn, const std::string &variableName) {
04198     readStringTo(_tmpReadToken, ptrn, variableName);
04199     return _tmpReadToken;
04200 }
04201 
04202 std::vector<std::string>
```

```
04203 InStream::readStrings(int size, const std::string &ptrn, const std::string &variablesName, int
      indexBase) {
04204     pattern p(ptrn);
04205     __testlib_readMany(readStrings, readString(p, variablesName), std::string, false)
04206 }
04207
04208 void InStream::readLineTo(std::string &result) {
04209     readStringTo(result);
04210 }
04211
04212 std::string InStream::readLine() {
04213     return readString();
04214 }
04215
04216 std::vector<std::string> InStream::readLines(int size, int indexBase) {
04217     __testlib_readMany(readLines, readString(), std::string, false)
04218 }
04219
04220 void InStream::readLineTo(std::string &result, const pattern &p, const std::string &variableName) {
04221     readStringTo(result, p, variableName);
04222 }
04223
04224 void InStream::readLineTo(std::string &result, const std::string &ptrn, const std::string
      &variableName) {
04225     readStringTo(result, ptrn, variableName);
04226 }
04227
04228 std::string InStream::readLine(const pattern &p, const std::string &variableName) {
04229     return readString(p, variableName);
04230 }
04231
04232 std::vector<std::string>
04233 InStream::readLines(int size, const pattern &p, const std::string &variablesName, int indexBase) {
04234     __testlib_readMany(readLines, readString(p, variablesName), std::string, false)
04235 }
04236
04237 std::string InStream::readLine(const std::string &ptrn, const std::string &variableName) {
04238     return readString(ptrn, variableName);
04239 }
04240
04241 std::vector<std::string>
04242 InStream::readLines(int size, const std::string &ptrn, const std::string &variablesName, int
      indexBase) {
04243     pattern p(ptrn);
04244     __testlib_readMany(readLines, readString(p, variablesName), std::string, false)
04245 }
04246
04247 #ifdef __GNUC__
04248 __attribute__ ((format (printf, 3, 4)))
04249 #endif
04250 void InStream::ensuref(bool cond, const char *format, ...) {
04251     if (!cond) {
04252         FMT_TO_RESULT(format, format, message);
04253         this->__testlib_ensure(cond, message);
04254     }
04255 }
04256
04257 void InStream::__testlib_ensure(bool cond, std::string message) {
04258     if (!cond)
04259         this->quit(_wa, message.c_str());
04260 }
04261
04262 void InStream::close() {
04263     if (NULL != reader) {
04264         reader->close();
04265         delete reader;
04266         reader = NULL;
04267     }
04268
04269     opened = false;
04270 }
04271
04272 NORETURN void quit(TResult result, const std::string &msg) {
04273     ouf.quit(result, msg.c_str());
04274 }
04275
04276 NORETURN void quit(TResult result, const char *msg) {
04277     ouf.quit(result, msg);
04278 }
04279
04280 NORETURN void __testlib_quitp(double points, const char *message) {
04281     __testlib_points = points;
04282     std::string stringPoints = removeDoubleTrailingZeroes(format("%.10f", points));
04283
04284     std::string quitMessage;
04285     if (NULL == message || 0 == strlen(message))
04286         quitMessage = stringPoints;
```

```
04287     else
04288         quitMessage = stringPoints + " " + message;
04289
04290     quit(_points, quitMessage.c_str());
04291 }
04292
04293 NORETURN void __testlib_quitp(int points, const char *message) {
04294     __testlib_points = points;
04295     std::string stringPoints = format("%d", points);
04296
04297     std::string quitMessage;
04298     if (NULL == message || 0 == strlen(message))
04299         quitMessage = stringPoints;
04300     else
04301         quitMessage = stringPoints + " " + message;
04302
04303     quit(_points, quitMessage.c_str());
04304 }
04305
04306 NORETURN void quitp(float points, const std::string &message = "") {
04307     __testlib_quitp(double(points), message.c_str());
04308 }
04309
04310 NORETURN void quitp(double points, const std::string &message = "") {
04311     __testlib_quitp(points, message.c_str());
04312 }
04313
04314 NORETURN void quitp(long double points, const std::string &message = "") {
04315     __testlib_quitp(double(points), message.c_str());
04316 }
04317
04318 NORETURN void quitp(int points, const std::string &message = "") {
04319     __testlib_quitp(points, message.c_str());
04320 }
04321
04322 NORETURN void quitpi(const std::string &points_info, const std::string &message = "") {
04323     if (points_info.find(' ') != std::string::npos)
04324         quit(_fail, "Parameter 'points_info' can't contain spaces");
04325     if (message.empty())
04326         quit(_points, ("points_info=" + points_info).c_str());
04327     else
04328         quit(_points, ("points_info=" + points_info + " " + message).c_str());
04329 }
04330
04331 template<typename F>
04332 #ifdef __GNUC__
04333 __attribute__ ((format (printf, 2, 3)))
04334 #endif
04335 NORETURN void quitp(F points, const char *format, ...) {
04336     FMT_TO_RESULT(format, format, message);
04337     quitp(points, message);
04338 }
04339
04340 #ifdef __GNUC__
04341 __attribute__ ((format (printf, 2, 3)))
04342 #endif
04343 NORETURN void quitf(TResult result, const char *format, ...) {
04344     FMT_TO_RESULT(format, format, message);
04345     quit(result, message);
04346 }
04347
04348 #ifdef __GNUC__
04349 __attribute__ ((format (printf, 3, 4)))
04350 #endif
04351 void quitif(bool condition, TResult result, const char *format, ...) {
04352     if (condition) {
04353         FMT_TO_RESULT(format, format, message);
04354         quit(result, message);
04355     }
04356 }
04357
04358 NORETURN void __testlib_help() {
04359     InStream::textColor(InStream::LightCyan);
04360     std::fprintf(stderr, "TESTLIB %s, https://github.com/MikeMirzayanov/testlib/ ", VERSION);
04361     std::fprintf(stderr, "by Mike Mirzayanov, copyright(c) 2005-2020\n");
04362     std::fprintf(stderr, "Checker name: \"%s\"\n", checkerName.c_str());
04363     InStream::textColor(InStream::LightGray);
04364
04365     std::fprintf(stderr, "\n");
04366     std::fprintf(stderr, "Latest features: \n");
04367     for (size_t i = 0; i < sizeof(latestFeatures) / sizeof(char *); i++) {
04368         std::fprintf(stderr, "*) %s\n", latestFeatures[i]);
04369     }
04370     std::fprintf(stderr, "\n");
04371
04372     std::fprintf(stderr, "Program must be run with the following arguments: \n");
04373     std::fprintf(stderr, "    [--testset testset] [--group group] <input-file> <output-file>
```

```
    <answer-file> [<report-file> [<-appes>]]\n\n");
04374
04375     __testlib_exitCode = FAIL_EXIT_CODE;
04376     std::exit(FAIL_EXIT_CODE);
04377 }
04378
04379 static void __testlib_ensuresPreconditions() {
04380     // testlib assumes: sizeof(int) = 4.
04381     __TESTLIB_STATIC_ASSERT(sizeof(int) == 4);
04382
04383     // testlib assumes: INT_MAX == 2147483647.
04384     __TESTLIB_STATIC_ASSERT(INT_MAX == 2147483647);
04385
04386     // testlib assumes: sizeof(long long) = 8.
04387     __TESTLIB_STATIC_ASSERT(sizeof(long long) == 8);
04388
04389     // testlib assumes: sizeof(double) = 8.
04390     __TESTLIB_STATIC_ASSERT(sizeof(double) == 8);
04391
04392     // testlib assumes: no -ffast-math.
04393     if (!__testlib_isNaN(+__testlib_nan()))
04394         quit(_fail, "Function __testlib_isNaN is not working correctly: possible reason is
    '-ffast-math'");
04395     if (!__testlib_isNaN(-__testlib_nan()))
04396         quit(_fail, "Function __testlib_isNaN is not working correctly: possible reason is
    '-ffast-math'");
04397 }
04398
04399 std::string __testlib_testset;
04400
04401 std::string getTestset() {
04402     return __testlib_testset;
04403 }
04404
04405 std::string __testlib_group;
04406
04407 std::string getGroup() {
04408     return __testlib_group;
04409 }
04410
04411 static void __testlib_set_testset_and_group(int argc, char* argv[]) {
04412     for (int i = 1; i < argc; i++) {
04413         if (!strcmp("--testset", argv[i])) {
04414             if (i + 1 < argc && strlen(argv[i + 1]) > 0)
04415                 __testlib_testset = argv[++i];
04416             else
04417                 quit(_fail, std::string("Expected non-empty testset after --testset command line
    parameter"));
04418         } else if (!strcmp("--group", argv[i])) {
04419             if (i + 1 < argc)
04420                 __testlib_group = argv[++i];
04421             else
04422                 quit(_fail, std::string("Expected group after --group command line parameter"));
04423         }
04424     }
04425 }
04426
04427 void registerGen(int argc, char *argv[], int randomGeneratorVersion) {
04428     if (randomGeneratorVersion < 0 || randomGeneratorVersion > 1)
04429         quitf(_fail, "Random generator version is expected to be 0 or 1.");
04430     random_t::version = randomGeneratorVersion;
04431
04432     __testlib_ensuresPreconditions();
04433     TestlibFinalizeGuard::registered = true;
04434
04435     testlibMode = _generator;
04436     __testlib_set_binary(stdin);
04437     rnd.setSeed(argc, argv);
04438
04439 #if __cplusplus > 199711L || defined(_MSC_VER)
04440     prepareOpts(argc, argv);
04441 #endif
04442 }
04443
04444 #ifdef USE_RND_AS_BEFORE_087
04445 void registerGen(int argc, char* argv[])
04446 {
04447     registerGen(argc, argv, 0);
04448 }
04449 #else
04450 #ifdef __GNUC__
04451 #if (__GNUC__ > 4) || ((__GNUC__ == 4) && (__GNUC_MINOR__ > 4))
04452 __attribute__ ((deprecated("Use registerGen(argc, argv, 0) or registerGen(argc, argv, 1)."
04453 " The third parameter stands for the random generator version."
04454 " If you are trying to compile old generator use macro -DUSE_RND_AS_BEFORE_087 or registerGen(argc,
    argv, 0)."
04455 " Version 1 has been released on Spring, 2013. Use it to write new generators.")))
```

```
04456 #else
04457 __attribute__ ((deprecated))
04458 #endif
04459 #endif
04460 #ifdef _MSC_VER
04461 __declspec(deprecated("Use registerGen(argc, argv, 0) or registerGen(argc, argv, 1)."
04462         " The third parameter stands for the random generator version."
04463         " If you are trying to compile old generator use macro -DUSE_RND_AS_BEFORE_087 or
      registerGen(argc, argv, 0)."
04464         " Version 1 has been released on Spring, 2013. Use it to write new generators."))
04465 #endif
04466 void registerGen(int argc, char *argv[]) {
04467     std::fprintf(stderr, "Use registerGen(argc, argv, 0) or registerGen(argc, argv, 1)."
04468                          " The third parameter stands for the random generator version."
04469                          " If you are trying to compile old generator use macro
      -DUSE_RND_AS_BEFORE_087 or registerGen(argc, argv, 0)."
04470                          " Version 1 has been released on Spring, 2013. Use it to write new
      generators.\n\n");
04471     registerGen(argc, argv, 0);
04472 }
04473 #endif
04474
04475 void setAppesModeEncoding(std::string appesModeEncoding) {
04476     static const char* const ENCODINGS[] = {"ascii", "utf-7", "utf-8", "utf-16", "utf-16le",
      "utf-16be", "utf-32", "utf-32le", "utf-32be", "iso-8859-1",
04477 "iso-8859-2", "iso-8859-3", "iso-8859-4", "iso-8859-5", "iso-8859-6", "iso-8859-7", "iso-8859-8",
      "iso-8859-9", "iso-8859-10", "iso-8859-11",
04478 "iso-8859-13", "iso-8859-14", "iso-8859-15", "iso-8859-16", "windows-1250", "windows-1251",
      "windows-1252", "windows-1253", "windows-1254", "windows-1255",
04479 "windows-1256", "windows-1257", "windows-1258", "gb2312", "gbk", "gb18030", "big5", "shift-jis",
      "euc-jp", "euc-kr",
04480 "euc-cn", "euc-tw", "koi8-r", "koi8-u", "tis-620", "ibm437", "ibm850", "ibm852", "ibm855", "ibm857",
04481 "ibm860", "ibm861", "ibm862", "ibm863", "ibm865", "ibm866", "ibm869", "macroman", "maccentraleurope",
      "maciceland",
04482 "maccroatian", "macromania", "maccyrillic", "macukraine", "macgreek", "macturkish", "machebrew",
      "macarabic", "macthai", "hz-gb-2312",
04483 "iso-2022-jp", "iso-2022-kr", "iso-2022-cn", "armscii-8", "tscii", "iscii", "viscii", "geostd8",
      "cp949", "cp874",
04484 "cp1006", "cp775", "cp858", "cp737", "cp853", "cp856", "cp922", "cp1046", "cp1125", "cp1131",
04485 "ptcp154", "koi8-t", "koi8-ru", "mulelao-1", "cp1133", "iso-ir-166", "tcvn", "iso-ir-14", "iso-ir-87",
      "iso-ir-159"};
04486
04487     appesModeEncoding = lowerCase(appesModeEncoding);
04488     bool valid = false;
04489     for (size_t i = 0; i < sizeof(ENCODINGS) / sizeof(ENCODINGS[0]); i++)
04490         if (appesModeEncoding == ENCODINGS[i]) {
04491             valid = true;
04492             break;
04493         }
04494     if (!valid)
04495         quit(_fail, "Unexpected encoding for setAppesModeEncoding(encoding)");
04496     ::appesModeEncoding = appesModeEncoding;
04497 }
04498
04499 void registerInteraction(int argc, char *argv[]) {
04500     __testlib_ensuresPreconditions();
04501     __testlib_set_testset_and_group(argc, argv);
04502     TestlibFinalizeGuard::registered = true;
04503
04504     testlibMode = _interactor;
04505     __testlib_set_binary(stdin);
04506
04507     if (argc > 1 && !strcmp("--help", argv[1]))
04508         __testlib_help();
04509
04510     if (argc < 3 || argc > 6) {
04511         quit(_fail, std::string("Program must be run with the following arguments: ") +
04512                     std::string("<input-file> <output-file> [<answer-file> [<report-file>
      [<-appes>]]]") +
04513                     "\nUse \"--help\" to get help information");
04514     }
04515
04516     if (argc <= 4) {
04517         resultName = "";
04518         appesMode = false;
04519     }
04520
04521 #ifndef EJUDGE
04522     if (argc == 5) {
04523         resultName = argv[4];
04524         appesMode = false;
04525     }
04526
04527     if (argc == 6) {
04528         if (strcmp("-APPES", argv[5]) && strcmp("-appes", argv[5])) {
04529             quit(_fail, std::string("Program must be run with the following arguments: ") +
04530                         "<input-file> <output-file> <answer-file> [<report-file> [<-appes>]]");
```

```
04531         } else {
04532             resultName = argv[4];
04533             appesMode = true;
04534         }
04535     }
04536 #endif
04537
04538     inf.init(argv[1], _input);
04539
04540     tout.open(argv[2], std::ios_base::out);
04541     if (tout.fail() || !tout.is_open())
04542         quit(_fail, std::string("Can not write to the test-output-file '") + argv[2] +
      std::string("'"));
04543
04544     ouf.init(stdin, _output);
04545
04546     if (argc >= 4)
04547         ans.init(argv[3], _answer);
04548     else
04549         ans.name = "unopened answer stream";
04550 }
04551
04552 void registerValidation() {
04553     __testlib_ensuresPreconditions();
04554     TestlibFinalizeGuard::registered = true;
04555
04556     testlibMode = _validator;
04557
04558     __testlib_set_binary(stdin);
04559     __testlib_set_binary(stdout);
04560     __testlib_set_binary(stderr);
04561
04562     inf.init(stdin, _input);
04563     inf.strict = true;
04564 }
04565
04566 void registerValidation(int argc, char *argv[]) {
04567     registerValidation();
04568     __testlib_set_testset_and_group(argc, argv);
04569
04570     validator.initialize();
04571     TestlibFinalizeGuard::registered = true;
04572
04573     std::string comment = "Validator must be run with the following arguments:"
04574                           " [--testset testset]"
04575                           " [--group group]"
04576                           " [--testOverviewLogFileName fileName]"
04577                           " [--testMarkupFileName fileName]"
04578                           " [--testCase testCase]"
04579                           " [--testCaseFileName fileName]"
04580                           ;
04581
04582     for (int i = 1; i < argc; i++) {
04583         if (!strcmp("--testset", argv[i])) {
04584             if (i + 1 < argc && strlen(argv[i + 1]) > 0)
04585                 validator.setTestset(argv[++i]);
04586             else
04587                 quit(_fail, comment);
04588         }
04589         if (!strcmp("--group", argv[i])) {
04590             if (i + 1 < argc)
04591                 validator.setGroup(argv[++i]);
04592             else
04593                 quit(_fail, comment);
04594         }
04595         if (!strcmp("--testOverviewLogFileName", argv[i])) {
04596             if (i + 1 < argc)
04597                 validator.setTestOverviewLogFileName(argv[++i]);
04598             else
04599                 quit(_fail, comment);
04600         }
04601         if (!strcmp("--testMarkupFileName", argv[i])) {
04602             if (i + 1 < argc)
04603                 validator.setTestMarkupFileName(argv[++i]);
04604             else
04605                 quit(_fail, comment);
04606         }
04607         if (!strcmp("--testCase", argv[i])) {
04608             if (i + 1 < argc) {
04609                 long long testCase = stringToLongLong(inf, argv[++i]);
04610                 if (testCase < 1 || testCase >= __TESTLIB_MAX_TEST_CASE)
04611                     quit(_fail, format("Argument testCase should be between 1 and %d, but ",
      __TESTLIB_MAX_TEST_CASE)
04612                         + toString(testCase) + " found");
04613                 validator.setTestCase(int(testCase));
04614             } else
04615                 quit(_fail, comment);
```

```
04616         }
04617         if (!strcmp("--testCaseFileName", argv[i])) {
04618             if (i + 1 < argc) {
04619                 validator.setTestCaseFileName(argv[++i]);
04620             } else
04621                 quit(_fail, comment);
04622         }
04623     }
04624 }
04625
04626 void addFeature(const std::string &feature) {
04627     if (testlibMode != _validator)
04628         quit(_fail, "Features are supported in validators only.");
04629     validator.addFeature(feature);
04630 }
04631
04632 void feature(const std::string &feature) {
04633     if (testlibMode != _validator)
04634         quit(_fail, "Features are supported in validators only.");
04635     validator.feature(feature);
04636 }
04637
04638 class Checker {
04639 private:
04640     bool _initialized;
04641     std::string _testset;
04642     std::string _group;
04643
04644 public:
04645     Checker() : _initialized(false), _testset("tests"), _group() {
04646     }
04647
04648     void initialize() {
04649         _initialized = true;
04650     }
04651
04652     std::string testset() const {
04653         if (!_initialized)
04654             __testlib_fail("Checker should be initialized with registerTestlibCmd(argc, argv) instead
      of registerTestlibCmd() to support checker.testset()");
04655         return _testset;
04656     }
04657
04658     std::string group() const {
04659         if (!_initialized)
04660             __testlib_fail("Checker should be initialized with registerTestlibCmd(argc, argv) instead
      of registerTestlibCmd() to support checker.group()");
04661         return _group;
04662     }
04663
04664     void setTestset(const char *const testset) {
04665         _testset = testset;
04666     }
04667
04668     void setGroup(const char *const group) {
04669         _group = group;
04670     }
04671 } checker;
04672
04673 void registerTestlibCmd(int argc, char *argv[]) {
04674     __testlib_ensuresPreconditions();
04675     __testlib_set_testset_and_group(argc, argv);
04676     TestlibFinalizeGuard::registered = true;
04677
04678     testlibMode = _checker;
04679     __testlib_set_binary(stdin);
04680
04681     std::vector<std::string> args(1, argv[0]);
04682     checker.initialize();
04683
04684     for (int i = 1; i < argc; i++) {
04685         if (!strcmp("--testset", argv[i])) {
04686             if (i + 1 < argc && strlen(argv[i + 1]) > 0)
04687                 checker.setTestset(argv[++i]);
04688             else
04689                 quit(_fail, std::string("Expected testset after --testset command line parameter"));
04690         } else if (!strcmp("--group", argv[i])) {
04691             if (i + 1 < argc)
04692                 checker.setGroup(argv[++i]);
04693             else
04694                 quit(_fail, std::string("Expected group after --group command line parameter"));
04695         } else
04696             args.push_back(argv[i]);
04697     }
04698
04699     argc = int(args.size());
04700     if (argc > 1 && "--help" == args[1])
```

```
04701         __testlib_help();
04702
04703     if (argc < 4 || argc > 6) {
04704         quit(_fail, std::string("Program must be run with the following arguments: ") +
04705                     std::string("[--testset testset] [--group group] <input-file> <output-file>
      <answer-file> [<report-file> [<-appes>]]") +
04706                     "\nUse \"--help\" to get help information");
04707     }
04708
04709     if (argc == 4) {
04710         resultName = "";
04711         appesMode = false;
04712     }
04713
04714 #ifndef EJUDGE
04715     if (argc == 5) {
04716         resultName = args[4];
04717         appesMode = false;
04718     }
04719
04720     if (argc == 6) {
04721         if ("-APPES" != args[5] && "-appes" != args[5]) {
04722             quit(_fail, std::string("Program must be run with the following arguments: ") +
04723                         "<input-file> <output-file> <answer-file> [<report-file> [<-appes>]]");
04724         } else {
04725             resultName = args[4];
04726             appesMode = true;
04727         }
04728     }
04729 #endif
04730
04731     inf.init(args[1], _input);
04732     ouf.init(args[2], _output);
04733     ouf.skipBom();
04734     ans.init(args[3], _answer);
04735 }
04736
04737 void registerTestlib(int argc, ...) {
04738     if (argc < 3 || argc > 5)
04739         quit(_fail, std::string("Program must be run with the following arguments: ") +
04740                     "<input-file> <output-file> <answer-file> [<report-file> [<-appes>]]");
04741
04742     char **argv = new char *[argc + 1];
04743
04744     va_list ap;
04745     va_start(ap, argc);
04746     argv[0] = NULL;
04747     for (int i = 0; i < argc; i++) {
04748         argv[i + 1] = va_arg(ap, char*);
04749     }
04750     va_end(ap);
04751
04752     registerTestlibCmd(argc + 1, argv);
04753     delete[] argv;
04754 }
04755
04756 static inline void __testlib_ensure(bool cond, const std::string &msg) {
04757     if (!cond)
04758         quit(_fail, msg.c_str());
04759 }
04760
04761 #ifdef __GNUC__
04762 __attribute__((unused))
04763 #endif
04764 static inline void __testlib_ensure(bool cond, const char *msg) {
04765     if (!cond)
04766         quit(_fail, msg);
04767 }
04768
04769 #define ensure(cond) __testlib_ensure(cond, "Condition failed: \"" #cond "\"")
04770 #define STRINGIZE_DETAIL(x) #x
04771 #define STRINGIZE(x) STRINGIZE_DETAIL(x)
04772 #define ensure_ext(cond) __testlib_ensure(cond, "Line " STRINGIZE(__LINE__) ": Condition failed: \""
      #cond "\"")
04773
04774 #ifdef __GNUC__
04775 __attribute__ ((format (printf, 2, 3)))
04776 #endif
04777 inline void ensuref(bool cond, const char *format, ...) {
04778     if (!cond) {
04779         FMT_TO_RESULT(format, format, message);
04780         __testlib_ensure(cond, message);
04781     }
04782 }
04783
04784 NORETURN static void __testlib_fail(const std::string &message) {
04785     quitf(_fail, "%s", message.c_str());
```

```
04786 }
04787
04788 #ifdef __GNUC__
04789 __attribute__ ((format (printf, 1, 2)))
04790 #endif
04791 void setName(const char *format, ...) {
04792     FMT_TO_RESULT(format, format, name);
04793     checkerName = name;
04794 }
04795
04796 /*
04797  * Do not use random_shuffle, because it will produce different result
04798  * for different C++ compilers.
04799  *
04800  * This implementation uses testlib random_t to produce random numbers, so
04801  * it is stable.
04802  */
04803 template<typename _RandomAccessIter>
04804 void shuffle(_RandomAccessIter __first, _RandomAccessIter __last) {
04805     if (__first == __last) return;
04806     for (_RandomAccessIter __i = __first + 1; __i != __last; ++__i)
04807         std::iter_swap(__i, __first + rnd.next(int(__i - __first) + 1));
04808 }
04809
04810
04811 template<typename _RandomAccessIter>
04812 #if defined(__GNUC__) && !defined(__clang__)
04813 __attribute__ ((error("Don't use random_shuffle(), use shuffle() instead")))
04814 #endif
04815 void random_shuffle(_RandomAccessIter, _RandomAccessIter) {
04816     quitf(_fail, "Don't use random_shuffle(), use shuffle() instead");
04817 }
04818
04819 #ifdef __GLIBC__
04820 #  define RAND_THROW_STATEMENT throw()
04821 #else
04822 #  define RAND_THROW_STATEMENT
04823 #endif
04824
04825 #if defined(__GNUC__) && !defined(__clang__)
04826
04827 __attribute__ ((error("Don't use rand(), use rnd.next() instead")))
04828 #endif
04829 #ifdef _MSC_VER
04830 #   pragma warning( disable : 4273 )
04831 #endif
04832 int rand() RAND_THROW_STATEMENT
04833 {
04834     quitf(_fail, "Don't use rand(), use rnd.next() instead");
04835
04836     /* This line never runs. */
04837     //throw "Don't use rand(), use rnd.next() instead";
04838 }
04839
04840 #if defined(__GNUC__) && !defined(__clang__)
04841
04842 __attribute__ ((error("Don't use srand(), you should use "
04843 "'registerGen(argc, argv, 1);' to initialize generator seed "
04844 "by hash code of the command line params. The third parameter "
04845 "is randomGeneratorVersion (currently the latest is 1).")))
04846 #endif
04847 #ifdef _MSC_VER
04848 #   pragma warning( disable : 4273 )
04849 #endif
04850 void srand(unsigned int seed) RAND_THROW_STATEMENT
04851 {
04852     quitf(_fail, "Don't use srand(), you should use "
04853                  "'registerGen(argc, argv, 1);' to initialize generator seed "
04854                  "by hash code of the command line params. The third parameter "
04855                  "is randomGeneratorVersion (currently the latest is 1) [ignored seed=%u].", seed);
04856 }
04857
04858 void startTest(int test) {
04859     const std::string testFileName = vtos(test);
04860     if (NULL == freopen(testFileName.c_str(), "wt", stdout))
04861         __testlib_fail("Unable to write file '" + testFileName + "'");
04862 }
04863
04864 #ifdef __GNUC__
04865 __attribute__((const))
04866 #endif
04867 inline std::string compress(const std::string &s) {
04868     return __testlib_part(s);
04869 }
04870
04871 #ifdef __GNUC__
04872 __attribute__((const))
```

```
04873 #endif
04874 inline std::string englishEnding(int x) {
04875     x %= 100;
04876     if (x / 10 == 1)
04877         return "th";
04878     if (x % 10 == 1)
04879         return "st";
04880     if (x % 10 == 2)
04881         return "nd";
04882     if (x % 10 == 3)
04883         return "rd";
04884     return "th";
04885 }
04886
04887 template<typename _ForwardIterator, typename _Separator>
04888 #ifdef __GNUC__
04889 __attribute__((const))
04890 #endif
04891 std::string join(_ForwardIterator first, _ForwardIterator last, _Separator separator) {
04892     std::stringstream ss;
04893     bool repeated = false;
04894     for (_ForwardIterator i = first; i != last; i++) {
04895         if (repeated)
04896             ss « separator;
04897         else
04898             repeated = true;
04899         ss « *i;
04900     }
04901     return ss.str();
04902 }
04903
04904 template<typename _ForwardIterator>
04905 #ifdef __GNUC__
04906 __attribute__((const))
04907 #endif
04908 std::string join(_ForwardIterator first, _ForwardIterator last) {
04909     return join(first, last, ' ');
04910 }
04911
04912 template<typename _Collection, typename _Separator>
04913 #ifdef __GNUC__
04914 __attribute__((const))
04915 #endif
04916 std::string join(const _Collection &collection, _Separator separator) {
04917     return join(collection.begin(), collection.end(), separator);
04918 }
04919
04920 template<typename _Collection>
04921 #ifdef __GNUC__
04922 __attribute__((const))
04923 #endif
04924 std::string join(const _Collection &collection) {
04925     return join(collection, ' ');
04926 }
04927
04932 #ifdef __GNUC__
04933 __attribute__((const))
04934 #endif
04935 std::vector<std::string> split(const std::string &s, char separator) {
04936     std::vector<std::string> result;
04937     std::string item;
04938     for (size_t i = 0; i < s.length(); i++)
04939         if (s[i] == separator) {
04940             result.push_back(item);
04941             item = "";
04942         } else
04943             item += s[i];
04944     result.push_back(item);
04945     return result;
04946 }
04947
04952 #ifdef __GNUC__
04953 __attribute__((const))
04954 #endif
04955 std::vector<std::string> split(const std::string &s, const std::string &separators) {
04956     if (separators.empty())
04957         return std::vector<std::string>(1, s);
04958
04959     std::vector<bool> isSeparator(256);
04960     for (size_t i = 0; i < separators.size(); i++)
04961         isSeparator[(unsigned char) (separators[i])] = true;
04962
04963     std::vector<std::string> result;
04964     std::string item;
04965     for (size_t i = 0; i < s.length(); i++)
04966         if (isSeparator[(unsigned char) (s[i])]) {
04967             result.push_back(item);
```

```
04968             item = "";
04969         } else
04970             item += s[i];
04971     result.push_back(item);
04972     return result;
04973 }
04974
04978 #ifdef __GNUC__
04979 __attribute__((const))
04980 #endif
04981 std::vector<std::string> tokenize(const std::string &s, char separator) {
04982     std::vector<std::string> result;
04983     std::string item;
04984     for (size_t i = 0; i < s.length(); i++)
04985         if (s[i] == separator) {
04986             if (!item.empty())
04987                 result.push_back(item);
04988             item = "";
04989         } else
04990             item += s[i];
04991     if (!item.empty())
04992         result.push_back(item);
04993     return result;
04994 }
04995
04999 #ifdef __GNUC__
05000 __attribute__((const))
05001 #endif
05002 std::vector<std::string> tokenize(const std::string &s, const std::string &separators) {
05003     if (separators.empty())
05004         return std::vector<std::string>(1, s);
05005
05006     std::vector<bool> isSeparator(256);
05007     for (size_t i = 0; i < separators.size(); i++)
05008         isSeparator[(unsigned char) (separators[i])] = true;
05009
05010     std::vector<std::string> result;
05011     std::string item;
05012     for (size_t i = 0; i < s.length(); i++)
05013         if (isSeparator[(unsigned char) (s[i])]) {
05014             if (!item.empty())
05015                 result.push_back(item);
05016             item = "";
05017         } else
05018             item += s[i];
05019
05020     if (!item.empty())
05021         result.push_back(item);
05022
05023     return result;
05024 }
05025
05026 NORETURN void __testlib_expectedButFound(TResult result, std::string expected, std::string found,
      const char *prepend) {
05027     std::string message;
05028     if (strlen(prepend) != 0)
05029         message = format("%s: expected '%s', but found '%s'",
05030                          compress(prepend).c_str(), compress(expected).c_str(),
      compress(found).c_str());
05031     else
05032         message = format("expected '%s', but found '%s'",
05033                          compress(expected).c_str(), compress(found).c_str());
05034     quit(result, message);
05035 }
05036
05037 NORETURN void __testlib_expectedButFound(TResult result, double expected, double found, const char
      *prepend) {
05038     std::string expectedString = removeDoubleTrailingZeroes(format("%.12f", expected));
05039     std::string foundString = removeDoubleTrailingZeroes(format("%.12f", found));
05040     __testlib_expectedButFound(result, expectedString, foundString, prepend);
05041 }
05042
05043 template<typename _Tp>
05044 #ifdef __GNUC__
05045 __attribute__ ((format (printf, 4, 5)))
05046 #endif
05047 NORETURN void expectedButFound(TResult result, _Tp expected, _Tp found, const char *prependFormat =
      "", ...) {
05048     FMT_TO_RESULT(prependFormat, prependFormat, prepend);
05049     std::string expectedString = vtos(expected);
05050     std::string foundString = vtos(found);
05051     __testlib_expectedButFound(result, expectedString, foundString, prepend.c_str());
05052 }
05053
05054 template<>
05055 #ifdef __GNUC__
05056 __attribute__ ((format (printf, 4, 5)))
```

```
05057 #endif
05058 NORETURN void
05059 expectedButFound<std::string>(TResult result, std::string expected, std::string found, const char
      *prependFormat, ...) {
05060     FMT_TO_RESULT(prependFormat, prependFormat, prepend);
05061     __testlib_expectedButFound(result, expected, found, prepend.c_str());
05062 }
05063
05064 template<>
05065 #ifdef __GNUC__
05066 __attribute__ ((format (printf, 4, 5)))
05067 #endif
05068 NORETURN void expectedButFound<double>(TResult result, double expected, double found, const char
      *prependFormat, ...) {
05069     FMT_TO_RESULT(prependFormat, prependFormat, prepend);
05070     std::string expectedString = removeDoubleTrailingZeroes(format("%.12f", expected));
05071     std::string foundString = removeDoubleTrailingZeroes(format("%.12f", found));
05072     __testlib_expectedButFound(result, expectedString, foundString, prepend.c_str());
05073 }
05074
05075 template<>
05076 #ifdef __GNUC__
05077 __attribute__ ((format (printf, 4, 5)))
05078 #endif
05079 NORETURN void
05080 expectedButFound<const char *>(TResult result, const char *expected, const char *found, const char
      *prependFormat,
05081                                ...) {
05082     FMT_TO_RESULT(prependFormat, prependFormat, prepend);
05083     __testlib_expectedButFound(result, std::string(expected), std::string(found), prepend.c_str());
05084 }
05085
05086 template<>
05087 #ifdef __GNUC__
05088 __attribute__ ((format (printf, 4, 5)))
05089 #endif
05090 NORETURN void expectedButFound<float>(TResult result, float expected, float found, const char
      *prependFormat, ...) {
05091     FMT_TO_RESULT(prependFormat, prependFormat, prepend);
05092     __testlib_expectedButFound(result, double(expected), double(found), prepend.c_str());
05093 }
05094
05095 template<>
05096 #ifdef __GNUC__
05097 __attribute__ ((format (printf, 4, 5)))
05098 #endif
05099 NORETURN void
05100 expectedButFound<long double>(TResult result, long double expected, long double found, const char
      *prependFormat, ...) {
05101     FMT_TO_RESULT(prependFormat, prependFormat, prepend);
05102     __testlib_expectedButFound(result, double(expected), double(found), prepend.c_str());
05103 }
05104
05105 #if __cplusplus > 199711L || defined(_MSC_VER)
05106 template<typename _Tp>
05107 struct is_iterable {
05108     template<typename U>
05109     static char test(typename U::iterator *x);
05110
05111     template<typename U>
05112     static long test(U *x);
05113
05114     static const bool value = sizeof(test<_Tp>(0)) == 1;
05115 };
05116
05117 template<bool B, class _Tp = void>
05118 struct __testlib_enable_if {
05119 };
05120
05121 template<class _Tp>
05122 struct __testlib_enable_if<true, _Tp> {
05123     typedef _Tp type;
05124 };
05125
05126 template<typename _Tp>
05127 typename __testlib_enable_if<!is_iterable<_Tp>::value, void>::type __testlib_print_one(const _Tp &t) {
05128     std::cout << t;
05129 }
05130
05131 template<typename _Tp>
05132 typename __testlib_enable_if<is_iterable<_Tp>::value, void>::type __testlib_print_one(const _Tp &t) {
05133     bool first = true;
05134     for (typename _Tp::const_iterator i = t.begin(); i != t.end(); i++) {
05135         if (first)
05136             first = false;
05137         else
05138             std::cout << " ";
```

```
05139         std::cout « *i;
05140     }
05141 }
05142
05143 template<>
05144 typename __testlib_enable_if<is_iterable<std::string>::value, void>::type
05145 __testlib_print_one<std::string>(const std::string &t) {
05146     std::cout « t;
05147 }
05148
05149 template<typename A, typename B>
05150 void __println_range(A begin, B end) {
05151     bool first = true;
05152     for (B i = B(begin); i != end; i++) {
05153         if (first)
05154             first = false;
05155         else
05156             std::cout « " ";
05157         __testlib_print_one(*i);
05158     }
05159     std::cout « std::endl;
05160 }
05161
05162 template<class _Tp, class Enable = void>
05163 struct is_iterator {
05164     static _Tp makeT();
05165
05166     typedef void *twoptrs[2];
05167
05168     static twoptrs &test(...);
05169
05170     template<class R>
05171     static typename R::iterator_category *test(R);
05172
05173     template<class R>
05174     static void *test(R *);
05175
05176     static const bool value = sizeof(test(makeT())) == sizeof(void *);
05177 };
05178
05179 template<class _Tp>
05180 struct is_iterator<_Tp, typename __testlib_enable_if<std::is_array<_Tp>::value>::type> {
05181     static const bool value = false;
05182 };
05183
05184 template<typename A, typename B>
05185 typename __testlib_enable_if<!is_iterator<B>::value, void>::type println(const A &a, const B &b) {
05186     __testlib_print_one(a);
05187     std::cout « " ";
05188     __testlib_print_one(b);
05189     std::cout « std::endl;
05190 }
05191
05192 template<typename A, typename B>
05193 typename __testlib_enable_if<is_iterator<B>::value, void>::type println(const A &a, const B &b) {
05194     __println_range(a, b);
05195 }
05196
05197 template<typename A>
05198 void println(const A *a, const A *b) {
05199     __println_range(a, b);
05200 }
05201
05202 template<>
05203 void println<char>(const char *a, const char *b) {
05204     __testlib_print_one(a);
05205     std::cout « " ";
05206     __testlib_print_one(b);
05207     std::cout « std::endl;
05208 }
05209
05210 template<typename _Tp>
05211 void println(const _Tp &x) {
05212     __testlib_print_one(x);
05213     std::cout « std::endl;
05214 }
05215
05216 template<typename A, typename B, typename C>
05217 void println(const A &a, const B &b, const C &c) {
05218     __testlib_print_one(a);
05219     std::cout « " ";
05220     __testlib_print_one(b);
05221     std::cout « " ";
05222     __testlib_print_one(c);
05223     std::cout « std::endl;
05224 }
05225
```

```
05226 template<typename A, typename B, typename C, typename D>
05227 void println(const A &a, const B &b, const C &c, const D &d) {
05228     __testlib_print_one(a);
05229     std::cout « " ";
05230     __testlib_print_one(b);
05231     std::cout « " ";
05232     __testlib_print_one(c);
05233     std::cout « " ";
05234     __testlib_print_one(d);
05235     std::cout « std::endl;
05236 }
05237
05238 template<typename A, typename B, typename C, typename D, typename E>
05239 void println(const A &a, const B &b, const C &c, const D &d, const E &e) {
05240     __testlib_print_one(a);
05241     std::cout « " ";
05242     __testlib_print_one(b);
05243     std::cout « " ";
05244     __testlib_print_one(c);
05245     std::cout « " ";
05246     __testlib_print_one(d);
05247     std::cout « " ";
05248     __testlib_print_one(e);
05249     std::cout « std::endl;
05250 }
05251
05252 template<typename A, typename B, typename C, typename D, typename E, typename F>
05253 void println(const A &a, const B &b, const C &c, const D &d, const E &e, const F &f) {
05254     __testlib_print_one(a);
05255     std::cout « " ";
05256     __testlib_print_one(b);
05257     std::cout « " ";
05258     __testlib_print_one(c);
05259     std::cout « " ";
05260     __testlib_print_one(d);
05261     std::cout « " ";
05262     __testlib_print_one(e);
05263     std::cout « " ";
05264     __testlib_print_one(f);
05265     std::cout « std::endl;
05266 }
05267
05268 template<typename A, typename B, typename C, typename D, typename E, typename F, typename G>
05269 void println(const A &a, const B &b, const C &c, const D &d, const E &e, const F &f, const G &g) {
05270     __testlib_print_one(a);
05271     std::cout « " ";
05272     __testlib_print_one(b);
05273     std::cout « " ";
05274     __testlib_print_one(c);
05275     std::cout « " ";
05276     __testlib_print_one(d);
05277     std::cout « " ";
05278     __testlib_print_one(e);
05279     std::cout « " ";
05280     __testlib_print_one(f);
05281     std::cout « " ";
05282     __testlib_print_one(g);
05283     std::cout « std::endl;
05284 }
05285
05286 /* opts */
05287
05292 struct TestlibOpt {
05293     std::string value;
05294     bool used;
05295
05296     TestlibOpt() : value(), used(false) {}
05297 };
05298
05312 size_t getOptType(char *s) {
05313     if (!s || strlen(s) <= 1)
05314         return 0;
05315
05316     if (s[0] == '-') {
05317         if (isalpha(s[1]))
05318             return 1;
05319         else if (s[1] == '-')
05320             return isalpha(s[2]) ? 2 : 0;
05321     }
05322
05323     return 0;
05324 }
05325
05355 size_t parseOpt(size_t argc, char *argv[], size_t index, std::map<std::string, TestlibOpt> &opts) {
05356     if (index >= argc)
05357         return 0;
05358
```

```
05359     size_t type = getOptType(argv[index]), inc = 1;
05360     if (type > 0) {
05361         std::string key(argv[index] + type), val;
05362         size_t sep = key.find('=');
05363         if (sep != std::string::npos) {
05364             val = key.substr(sep + 1);
05365             key = key.substr(0, sep);
05366         } else {
05367             if (index + 1 < argc && getOptType(argv[index + 1]) == 0) {
05368                 val = argv[index + 1];
05369                 inc = 2;
05370             } else {
05371                 if (key.length() > 1 && isdigit(key[1])) {
05372                     val = key.substr(1);
05373                     key = key.substr(0, 1);
05374                 } else {
05375                     val = "true";
05376                 }
05377             }
05378         }
05379         opts[key].value = val;
05380     } else {
05381         return inc;
05382     }
05383
05384     return inc;
05385 }
05386
05390 std::vector<std::string> __testlib_argv;
05391
05395 std::map<std::string, TestlibOpt> __testlib_opts;
05396
05404 bool __testlib_ensureNoUnusedOptsFlag = false;
05405
05410 bool __testlib_ensureNoUnusedOptsSuppressed = false;
05411
05416 void prepareOpts(int argc, char *argv[]) {
05417     if (argc <= 0)
05418         __testlib_fail("Opts: expected argc>=0 but found " + toString(argc));
05419     size_t n = static_cast<size_t>(argc); // NOLINT(hicpp-use-auto,modernize-use-auto)
05420     __testlib_opts = std::map<std::string, TestlibOpt>();
05421     for (size_t index = 1; index < n; index += parseOpt(n, argv, index, __testlib_opts));
05422     __testlib_argv = std::vector<std::string>(n);
05423     for (size_t index = 0; index < n; index++)
05424         __testlib_argv[index] = argv[index];
05425 }
05426
05431 std::string __testlib_indexToArgv(int index) {
05432     if (index < 0 || index >= int(__testlib_argv.size()))
05433         __testlib_fail("Opts: index '" + toString(index) + "' is out of range [0,"
05434             + toString(__testlib_argv.size()) + ")");
05435     return __testlib_argv[size_t(index)];
05436 }
05437
05442 std::string __testlib_keyToOpts(const std::string &key) {
05443     auto it = __testlib_opts.find(key);
05444     if (it == __testlib_opts.end())
05445         __testlib_fail("Opts: unknown key '" + compress(key) + "'");
05446     it->second.used = true;
05447     return it->second.value;
05448 }
05449
05450 template<typename _Tp>
05451 _Tp optValueToIntegral(const std::string &s, bool nonnegative);
05452
05453 long double optValueToLongDouble(const std::string &s);
05454
05455 std::string parseExponentialOptValue(const std::string &s) {
05456     size_t pos = std::string::npos;
05457     for (size_t i = 0; i < s.length(); i++)
05458         if (s[i] == 'e' || s[i] == 'E') {
05459             if (pos != std::string::npos)
05460                 __testlib_fail("Opts: expected typical exponential notation but '" + compress(s) + "'
    found");
05461             pos = i;
05462         }
05463     if (pos == std::string::npos)
05464         return s;
05465     std::string e = s.substr(pos + 1);
05466     if (!e.empty() && e[0] == '+')
05467         e = e.substr(1);
05468     if (e.empty())
05469         __testlib_fail("Opts: expected typical exponential notation but '" + compress(s) + "' found");
05470     if (e.length() > 20)
05471         __testlib_fail("Opts: expected typical exponential notation but '" + compress(s) + "' found");
05472     int ne = optValueToIntegral<int>(e, false);
05473     std::string num = s.substr(0, pos);
```

```
05474     if (num.length() > 20)
05475         __testlib_fail("Opts: expected typical exponential notation but '" + compress(s) + "' found");
05476     if (!num.empty() && num[0] == '+')
05477         num = num.substr(1);
05478     optValueToLongDouble(num);
05479     bool minus = false;
05480     if (num[0] == '-') {
05481         minus = true;
05482         num = num.substr(1);
05483     }
05484     for (int i = 0; i < +ne; i++) {
05485         size_t sep = num.find('.');
05486         if (sep == std::string::npos)
05487             num += '0';
05488         else {
05489             if (sep + 1 == num.length())
05490                 num[sep] = '0';
05491             else
05492                 std::swap(num[sep], num[sep + 1]);
05493         }
05494     }
05495     for (int i = 0; i < -ne; i++) {
05496         size_t sep = num.find('.');
05497         if (sep == std::string::npos)
05498             num.insert(num.begin() + int(num.length()) - 1, '.');
05499         else {
05500             if (sep == 0)
05501                 num.insert(num.begin() + 1, '0');
05502             else
05503                 std::swap(num[sep - 1], num[sep]);
05504         }
05505     }
05506     while (!num.empty() && num[0] == '0')
05507         num = num.substr(1);
05508     while (num.find('.') != std::string::npos && num.back() == '0')
05509         num = num.substr(0, num.length() - 1);
05510     if (!num.empty() && num.back() == '.')
05511         num = num.substr(0, num.length() - 1);
05512     if ((!num.empty() && num[0] == '.') || num.empty())
05513         num.insert(num.begin(), '0');
05514     return (minus ? "-" : "") + num;
05515 }
05516
05517 template<typename _Tp>
05518 _Tp optValueToIntegral(const std::string &s_, bool nonnegative) {
05519     std::string s(parseExponentialOptValue(s_));
05520     if (s.empty())
05521         __testlib_fail("Opts: expected integer but '" + compress(s_) + "' found");
05522     _Tp value = 0;
05523     long double about = 0.0;
05524     signed char sign = +1;
05525     size_t pos = 0;
05526     if (s[pos] == '-') {
05527         if (nonnegative)
05528             __testlib_fail("Opts: expected non-negative integer but '" + compress(s_) + "' found");
05529         sign = -1;
05530         pos++;
05531     }
05532     for (size_t i = pos; i < s.length(); i++) {
05533         if (s[i] < '0' || s[i] > '9')
05534             __testlib_fail("Opts: expected integer but '" + compress(s_) + "' found");
05535         value = _Tp(value * 10 + s[i] - '0');
05536         about = about * 10 + s[i] - '0';
05537     }
05538     value *= sign;
05539     about *= sign;
05540     if (fabsl(value - about) > 0.1)
05541         __testlib_fail("Opts: integer overflow: expected integer but '" + compress(s_) + "' found");
05542     return value;
05543 }
05544
05545 long double optValueToLongDouble(const std::string &s_) {
05546     std::string s(parseExponentialOptValue(s_));
05547     if (s.empty())
05548         __testlib_fail("Opts: expected float number but '" + compress(s_) + "' found");
05549     long double value = 0.0;
05550     signed char sign = +1;
05551     size_t pos = 0;
05552     if (s[pos] == '-') {
05553         sign = -1;
05554         pos++;
05555     }
05556     bool period = false;
05557     long double mul = 1.0;
05558     for (size_t i = pos; i < s.length(); i++) {
05559         if (s[i] == '.') {
05560             if (period)
```

```
05561                 __testlib_fail("Opts: expected float number but '" + compress(s_) + "' found");
05562             else {
05563                 period = true;
05564                 continue;
05565             }
05566         }
05567         if (period)
05568             mul *= 10.0;
05569         if (s[i] < '0' || s[i] > '9')
05570             __testlib_fail("Opts: expected float number but '" + compress(s_) + "' found");
05571         if (period)
05572             value += (s[i] - '0') / mul;
05573         else
05574             value = value * 10 + s[i] - '0';
05575     }
05576     value *= sign;
05577     return value;
05578 }
05579
05587 bool has_opt(const std::string &key) {
05588     __testlib_ensureNoUnusedOptsFlag = true;
05589     return __testlib_opts.count(key) != 0;
05590 }
05591
05592 /* About the following part for opt with 2 and 3 arguments.
05593  *
05594  * To parse the argv/opts correctly for a give type (integer, floating point or
05595  * string), some meta programming must be done to determine the type of
05596  * the type, and use the correct parsing function accordingly.
05597  *
05598  * The pseudo algorithm for determining the type of T and parse it accordingly
05599  * is as follows:
05600  *
05601  * if (T is integral type) {
05602  *   if (T is unsigned) {
05603  *     parse the argv/opt as an **unsigned integer** of type T.
05604  *   } else {
05605  *     parse the argv/opt as an **signed integer** of type T.
05606  * } else {
05607  *   if (T is floating point type) {
05608  *     parse the argv/opt as an **floating point** of type T.
05609  *   } else {
05610  *     // T should be std::string
05611  *     just the raw content of the argv/opts.
05612  *   }
05613  * }
05614  *
05615  * To help with meta programming, some `opt` function with 2 or 3 arguments are
05616  * defined.
05617  *
05618  * Opt with 3 arguments:    T opt(true/false is_integral, true/false is_unsigned, index/key)
05619  *
05620  *   + The first argument is for determining whether the type T is an integral
05621  *   type. That is, the result of std::is_integral<T>() should be passed to
05622  *   this argument. When false, the type _should_ be either floating point or a
05623  *   std::string.
05624  *
05625  *   + The second argument is for determining whether the signedness of the type
05626  *   T (if it is unsigned or signed). That is, the result of
05627  *   std::is_unsigned<T>() should be passed to this argument. This argument can
05628  *   be ignored if the first one is false, because it only applies to integer.
05629  *
05630  * Opt with 2 arguments:    T opt(true/false is_floating_point, index/key)
05631  *   + The first argument is for determining whether the type T is a floating
05632  *   point type. That is, the result of std::is_floating_point<T>() should be
05633  *   passed to this argument. When false, the type _should_ be a std::string.
05634  */
05635
05636 template<typename _Tp>
05637 _Tp opt(std::false_type is_floating_point, int index);
05638
05639 template<>
05640 std::string opt(std::false_type /*is_floating_point*/, int index) {
05641     return __testlib_indexToArgv(index);
05642 }
05643
05644 template<typename _Tp>
05645 _Tp opt(std::true_type /*is_floating_point*/, int index) {
05646     return _Tp(optValueToLongDouble(__testlib_indexToArgv(index)));
05647 }
05648
05649 template<typename _Tp, typename U>
05650 _Tp opt(std::false_type /*is_integral*/, U /*is_unsigned*/, int index) {
05651     return opt<_Tp>(std::is_floating_point<_Tp>(), index);
05652 }
05653
05654 template<typename _Tp>
```

```
05655 _Tp opt(std::true_type /*is_integral*/, std::false_type /*is_unsigned*/, int index) {
05656     return optValueToIntegral<_Tp>(__testlib_indexToArgv(index), false);
05657 }
05658
05659 template<typename _Tp>
05660 _Tp opt(std::true_type /*is_integral*/, std::true_type /*is_unsigned*/, int index) {
05661     return optValueToIntegral<_Tp>(__testlib_indexToArgv(index), true);
05662 }
05663
05664 template<>
05665 bool opt(std::true_type /*is_integral*/, std::true_type /*is_unsigned*/, int index) {
05666     std::string value = __testlib_indexToArgv(index);
05667     if (value == "true" || value == "1")
05668         return true;
05669     if (value == "false" || value == "0")
05670         return false;
05671     __testlib_fail("Opts: opt by index '" + toString(index) + "': expected bool true/false or 0/1 but
      '"
05672             + compress(value) + "' found");
05673 }
05674
05678 template<typename _Tp>
05679 _Tp opt(int index) {
05680     return opt<_Tp>(std::is_integral<_Tp>(), std::is_unsigned<_Tp>(), index);
05681 }
05682
05686 std::string opt(int index) {
05687     return opt<std::string>(index);
05688 }
05689
05694 template<typename _Tp>
05695 _Tp opt(int index, const _Tp &default_value) {
05696     if (index >= int(__testlib_argv.size())) {
05697         return default_value;
05698     }
05699     return opt<_Tp>(index);
05700 }
05701
05706 std::string opt(int index, const std::string &default_value) {
05707     return opt<std::string>(index, default_value);
05708 }
05709
05710 template<typename _Tp>
05711 _Tp opt(std::false_type is_floating_point, const std::string &key);
05712
05713 template<>
05714 std::string opt(std::false_type /*is_floating_point*/, const std::string &key) {
05715     return __testlib_keyToOpts(key);
05716 }
05717
05718 template<typename _Tp>
05719 _Tp opt(std::true_type /*is_integral*/, const std::string &key) {
05720     return _Tp(optValueToLongDouble(__testlib_keyToOpts(key)));
05721 }
05722
05723 template<typename _Tp, typename U>
05724 _Tp opt(std::false_type /*is_integral*/, U, const std::string &key) {
05725     return opt<_Tp>(std::is_floating_point<_Tp>(), key);
05726 }
05727
05728 template<typename _Tp>
05729 _Tp opt(std::true_type /*is_integral*/, std::false_type /*is_unsigned*/, const std::string &key) {
05730     return optValueToIntegral<_Tp>(__testlib_keyToOpts(key), false);
05731 }
05732
05733 template<typename _Tp>
05734 _Tp opt(std::true_type /*is_integral*/, std::true_type /*is_unsigned*/, const std::string &key) {
05735     return optValueToIntegral<_Tp>(__testlib_keyToOpts(key), true);
05736 }
05737
05738 template<>
05739 bool opt(std::true_type /*is_integral*/, std::true_type /*is_unsigned*/, const std::string &key) {
05740     if (!has_opt(key))
05741         return false;
05742     std::string value = __testlib_keyToOpts(key);
05743     if (value == "true" || value == "1")
05744         return true;
05745     if (value == "false" || value == "0")
05746         return false;
05747     __testlib_fail("Opts: key '" + compress(key) + "': expected bool true/false or 0/1 but '"
05748         + compress(value) + "' found");
05749 }
05750
05754 template<typename _Tp>
05755 _Tp opt(const std::string &key) {
05756     return opt<_Tp>(std::is_integral<_Tp>(), std::is_unsigned<_Tp>(), key);
05757 }
```

```
05758
05762 std::string opt(const std::string &key) {
05763     return opt<std::string>(key);
05764 }
05765
05766 /* Scorer started. */
05767
05768 enum TestResultVerdict {
05769     SKIPPED,
05770     OK,
05771     WRONG_ANSWER,
05772     RUNTIME_ERROR,
05773     TIME_LIMIT_EXCEEDED,
05774     IDLENESS_LIMIT_EXCEEDED,
05775     MEMORY_LIMIT_EXCEEDED,
05776     COMPILATION_ERROR,
05777     CRASHED,
05778     FAILED
05779 };
05780
05781 std::string serializeVerdict(TestResultVerdict verdict) {
05782     switch (verdict) {
05783         case SKIPPED: return "SKIPPED";
05784         case OK: return "OK";
05785         case WRONG_ANSWER: return "WRONG_ANSWER";
05786         case RUNTIME_ERROR: return "RUNTIME_ERROR";
05787         case TIME_LIMIT_EXCEEDED: return "TIME_LIMIT_EXCEEDED";
05788         case IDLENESS_LIMIT_EXCEEDED: return "IDLENESS_LIMIT_EXCEEDED";
05789         case MEMORY_LIMIT_EXCEEDED: return "MEMORY_LIMIT_EXCEEDED";
05790         case COMPILATION_ERROR: return "COMPILATION_ERROR";
05791         case CRASHED: return "CRASHED";
05792         case FAILED: return "FAILED";
05793     }
05794     throw "Unexpected verdict";
05795 }
05796
05797 TestResultVerdict deserializeTestResultVerdict(std::string s) {
05798     if (s == "SKIPPED")
05799         return SKIPPED;
05800     else if (s == "OK")
05801         return OK;
05802     else if (s == "WRONG_ANSWER")
05803         return WRONG_ANSWER;
05804     else if (s == "RUNTIME_ERROR")
05805         return RUNTIME_ERROR;
05806     else if (s == "TIME_LIMIT_EXCEEDED")
05807         return TIME_LIMIT_EXCEEDED;
05808     else if (s == "IDLENESS_LIMIT_EXCEEDED")
05809         return IDLENESS_LIMIT_EXCEEDED;
05810     else if (s == "MEMORY_LIMIT_EXCEEDED")
05811         return MEMORY_LIMIT_EXCEEDED;
05812     else if (s == "COMPILATION_ERROR")
05813         return COMPILATION_ERROR;
05814     else if (s == "CRASHED")
05815         return CRASHED;
05816     else if (s == "FAILED")
05817         return FAILED;
05818     ensuref(false, "Unexpected serialized TestResultVerdict");
05819     // No return actually.
05820     return FAILED;
05821 }
05822
05823 struct TestResult {
05824     int testIndex;
05825     std::string testset;
05826     std::string group;
05827     TestResultVerdict verdict;
05828     double points;
05829     long long timeConsumed;
05830     long long memoryConsumed;
05831     std::string input;
05832     std::string output;
05833     std::string answer;
05834     int exitCode;
05835     std::string checkerComment;
05836 };
05837
05838 std::string serializePoints(double points) {
05839     if (std::isnan(points))
05840         return "";
05841     else {
05842         char c[64];
05843         snprintf(c, 64, "%.03lf", points);
05844         return c;
05845     }
05846 }
05847
```

```
05848 double deserializePoints(std::string s) {
05849     if (s.empty())
05850         return std::numeric_limits<double>::quiet_NaN();
05851     else {
05852         double result;
05853         ensuref(sscanf(s.c_str(), "%lf", &result) == 1, "Invalid serialized points");
05854         return result;
05855     }
05856 }
05857
05858 std::string escapeTestResultString(std::string s) {
05859     std::string result;
05860     for (size_t i = 0; i < s.length(); i++) {
05861         if (s[i] == '\r')
05862             continue;
05863         if (s[i] == '\n') {
05864             result += "\\n";
05865             continue;
05866         }
05867         if (s[i] == '\\' || s[i] == ';')
05868             result += '\\';
05869         result += s[i];
05870     }
05871     return result;
05872 }
05873
05874 std::string unescapeTestResultString(std::string s) {
05875     std::string result;
05876     for (size_t i = 0; i < s.length(); i++) {
05877         if (s[i] == '\\' && i + 1 < s.length()) {
05878             if (s[i + 1] == 'n') {
05879                 result += '\n';
05880                 i++;
05881                 continue;
05882             } else if (s[i + 1] == ';' || s[i + 1] == '\\') {
05883                 result += s[i + 1];
05884                 i++;
05885                 continue;
05886             }
05887         }
05888         result += s[i];
05889     }
05890     return result;
05891 }
05892
05893 std::string serializeTestResult(TestResult tr) {
05894     std::string result;
05895     result += std::to_string(tr.testIndex);
05896     result += ";";
05897     result += escapeTestResultString(tr.testset);
05898     result += ";";
05899     result += escapeTestResultString(tr.group);
05900     result += ";";
05901     result += serializeVerdict(tr.verdict);
05902     result += ";";
05903     result += serializePoints(tr.points);
05904     result += ";";
05905     result += std::to_string(tr.timeConsumed);
05906     result += ";";
05907     result += std::to_string(tr.memoryConsumed);
05908     result += ";";
05909     result += escapeTestResultString(tr.input);
05910     result += ";";
05911     result += escapeTestResultString(tr.output);
05912     result += ";";
05913     result += escapeTestResultString(tr.answer);
05914     result += ";";
05915     result += std::to_string(tr.exitCode);
05916     result += ";";
05917     result += escapeTestResultString(tr.checkerComment);
05918     return result;
05919 }
05920
05921 TestResult deserializeTestResult(std::string s) {
05922     std::vector<std::string> items;
05923     std::string t;
05924     for (size_t i = 0; i < s.length(); i++) {
05925         if (s[i] == '\\') {
05926             t += s[i];
05927             if (i + 1 < s.length())
05928                 t += s[i + 1];
05929             i++;
05930             continue;
05931         } else {
05932             if (s[i] == ';') {
05933                 items.push_back(t);
05934                 t = "";
```

```
05935             } else
05936                 t += s[i];
05937         }
05938     }
05939     items.push_back(t);
05940
05941     ensuref(items.size() == 12, "Invalid TestResult serialization: expected exactly 12 items");
05942
05943     TestResult tr;
05944     size_t pos = 0;
05945     tr.testIndex = stoi(items[pos++]);
05946     tr.testset = unescapeTestResultString(items[pos++]);
05947     tr.group = unescapeTestResultString(items[pos++]);
05948     tr.verdict = deserializeTestResultVerdict(items[pos++]);
05949     tr.points = deserializePoints(items[pos++]);
05950     tr.timeConsumed = stoll(items[pos++]);
05951     tr.memoryConsumed = stoll(items[pos++]);
05952     tr.input = unescapeTestResultString(items[pos++]);
05953     tr.output = unescapeTestResultString(items[pos++]);
05954     tr.answer = unescapeTestResultString(items[pos++]);
05955     tr.exitCode = stoi(items[pos++]);
05956     tr.checkerComment = unescapeTestResultString(items[pos++]);
05957
05958     return tr;
05959 }
05960
05961 std::vector<TestResult> readTestResults(std::string fileName) {
05962     std::ifstream stream;
05963     stream.open(fileName.c_str(), std::ios::in);
05964     ensuref(stream.is_open(), "Can't read test results file '%s'", fileName.c_str());
05965     std::vector<TestResult> result;
05966     std::string line;
05967     while (getline(stream, line))
05968         if (!line.empty())
05969             result.push_back(deserializeTestResult(line));
05970     stream.close();
05971     return result;
05972 }
05973
05974 std::function<double(std::vector<TestResult>)> __testlib_scorer;
05975
05976 struct TestlibScorerGuard {
05977     ~TestlibScorerGuard() {
05978         if (testlibMode == _scorer) {
05979             std::vector<TestResult> testResults;
05980             while (!inf.eof()) {
05981                 std::string line = inf.readLine();
05982                 if (!line.empty())
05983                     testResults.push_back(deserializeTestResult(line));
05984             }
05985             inf.readEof();
05986             printf("%.3f\n", __testlib_scorer(testResults));
05987         }
05988     }
05989 } __testlib_scorer_guard;
05990
05991 void registerScorer(int argc, char *argv[], std::function<double(std::vector<TestResult>)> scorer) {
05992     /* Suppress unused. */
05993     (void)(argc), (void)(argv);
05994
05995     __testlib_ensuresPreconditions();
05996
05997     testlibMode = _scorer;
05998     __testlib_set_binary(stdin);
05999
06000     inf.init(stdin, _input);
06001     inf.strict = false;
06002
06003     __testlib_scorer = scorer;
06004 }
06005
06006 /* Scorer ended. */
06007
06016 template<typename _Tp>
06017 _Tp opt(const std::string &key, const _Tp &default_value) {
06018     if (!has_opt(key)) {
06019         return default_value;
06020     }
06021     return opt<_Tp>(key);
06022 }
06023
06032 std::string opt(const std::string &key, const std::string &default_value) {
06033     return opt<std::string>(key, default_value);
06034 }
06035
06043 void ensureNoUnusedOpts() {
06044     for (const auto &opt: __testlib_opts) {
```

```
06045         if (!opt.second.used) {
06046             __testlib_fail(format("Opts: unused key '%s'", compress(opt.first).c_str()));
06047         }
06048     }
06049 }
06050
06051 void suppressEnsureNoUnusedOpts() {
06052     __testlib_ensureNoUnusedOptsSuppressed = true;
06053 }
06054
06055 void TestlibFinalizeGuard::autoEnsureNoUnusedOpts() {
06056     if (__testlib_ensureNoUnusedOptsFlag && !__testlib_ensureNoUnusedOptsSuppressed) {
06057         ensureNoUnusedOpts();
06058     }
06059 }
06060
06061 TestlibFinalizeGuard testlibFinalizeGuard;
06062
06063 #endif
06064 #endif
```

Index