INF280: Competitive programming

Competitive programming

Competitive programming

Competitive programming is about solving problems.

Competitive programming

Competitive programming is about solving problems.

Let us solve a first problem!

Multiple types of contest

- IOI
- ICPC (including SWERC)
- Top Coder
- USACO
- ...

Different parameters

- team or individual
- duration
- partial points
- ...

Typical contest

A typical contest is generally a list of problems.

Problem statement

- a short story describing the problem
- a specification of the input and output (usually on stdin/stdout)
- limits (time / RAM / etc.)
- In-out example

Solution

A solution is a source code that gives the right outputs for the given inputs using the time and memory specified.

Why follow this course?

Competitive programming develops a lot of important skills:

- Algorithmic thinking
- Programming and Debugging
- Learning to describe algorithms
- Job interview style of technical questions

It is also fun :)

Why follow this course?

Competitive programming develops a lot of important skills:

- Algorithmic thinking
- Programming and Debugging
- Learning to describe algorithms
- Job interview style of technical questions

It is also fun :)

In this course you will also:

- familiarize yourself with C++
- develop your pseudo code skills
- learn how to methodically solve problems

Organization of a typical course

$\sim$15 min question part

Answer questions you might have

$\sim$30 min test part

Test on a set of “prepared” exercises (either exercises already studied or direct applications of studied algorithms)

$\sim$45min lesson part

Learn some methods or algorithms

$\sim$1h30 coding

Solving exercise with code, to develop fast programming skills.

Grading

Graded exercises in class

Every class (except today) will have a test on computers

Final exam

The final exam will be 3h exam on a computer

Final grade

Your grade will be the half the graded exercises in class and half the final exam.

Final exam

- individual participation
- 3 hours
- around 6 problems of varying difficulty
- one programming language: C++
- no Internet but some documentation allowed

Final exam on the 26th of June afternoon!

Graded exercises in class

- individual participation
- 30 min
- 3 problems
- one programming language: C++
- no Internet but some documentation allowed
- 2 of the problems are selected from the set of exercises given in a previous class
- the last problem is an application of an exercise seen in class

Solving competitive programming problems

Solving a problem requires to

- (optional) Reading the problem quickly to understand the context
- Reading the problem very carefully
- Finding an algorithm solving the problem within the specified limits
- Writing the code
- Testing the code on examples
- Submitting your program
- (optional) Debugging

Solving competitive programming problems

Program submission

Submitting programs

- You submit the source code on a website
- The system compiles your and then evaluates your programs on unknown inputs while checking the limits
- After a few seconds or minutes the system produces a verdict

Submitting programs

- You submit the source code on a website
- The system compiles your and then evaluates your programs on unknown inputs while checking the limits
- After a few seconds or minutes the system produces a verdict

If the verdict is **Accepted** you have just solved this problem.

Other verdicts

Compilation error.

It means your program does not compile...

Time limit exceeded / Memory limit exceeded

A recent CPU can process 5×10^7 C++ loop iterations per second
Also possible: infinite loop, memory corruption...

Runtime error.

Something went very wrong: assert failure, out of bounds, segfault, division by zero, etc.

Wrong answer.

You have the wrong algorithm or a bug...

Presentation error.

Not the right output format (*e.g.* extra space, caps, etc.).

Solving competitive programming problems

Testing your program

Testing programs

Cons:

- testing takes time
- it does not guarantee the absence of bugs

Testing programs

Cons:

- testing takes time
- it does not guarantee the absence of bugs

Pros:

- refused solutions incur a 20 min penalty
- it might take a few minutes to wait on a verdict
- the verdict itself is not enough to know what is happening

Testing programs

Cons:

- testing takes time
- it does not guarantee the absence of bugs

Pros:

- refused solutions incur a 20 min penalty
- it might take a few minutes to wait on a verdict
- the verdict itself is not enough to know what is happening

You should test your program in a quick but thorough manner.

How to test?

You have limited time...

- no need to generate tests
- no need to write many tests
- adapt the amount of testing to the complexity of your program

... but you do want to test

- use the sample in and out
- write several tests with several outputs
- compute in advance the results
- try to cover as many edge cases as possible

Testing with files

In all likelihood you will test your program several times, therefore your tests should lie in files:

Testing with files

In all likelihood you will test your program several times, therefore your tests should lie in files:

```
./a.out < test01.in > test01.out # redirect in and out
```

Testing with files

In all likelihood you will test your program several times, therefore your tests should lie in files:

```
./a.out < test01.in > test01.out # redirect in and out
```

```
diff test01.out test01.ans # compare with expected result
```

This works for Unix-based systems

Testing with files

In all likelihood you will test your program several times, therefore your tests should lie in files:

```
./a.out < test01.in > test01.out # redirect in and out
```

```
diff test01.out test01.ans # compare with expected result
```

This works for Unix-based systems

```
# with input in testXY.in and output in testXY.ans
for i in *.in ; do
    echo "=== $i ===" ;
    ./a.out < $i > ${i%%in}out
    diff ${i%%in}out ${i%%in}ans
done
```

Solving competitive programming problems

Writing code

When you have the idea

Try to reformulate the idea for your solution:

- imagine explaining the idea to a peer
- look for ways to simplify the idea
  - does your idea relies on a standard algorithm?
  - if so, can you match exactly the algorithm description?
  - can you add special values to match the edge cases?

Using pseudo code

Writing pseudo-code has several benefits

- you can concentrate on the idea of the algorithm and not the implementation details
- you can check that your idea works (correct answer and complexity)
- and in a SWERC competition you free the computer

On simpler problems you can avoid writing pseudo-code or just give the big picture.

Classical programming errors

- using a non-strict comparison where a strict was required
- making a mistake in a constant (e.g. 100000 instead of 1000000)
- not allocating enough memory (e.g. int t[1000] and then accessing t[1000])
- not checking for overflow or float type that are not precise enough
- comparing two different types of things (e.g. $idCow < nbCarrots$)
- swapping xs and ys in a function call
- mixing variable and constant

Adopt good and more importantly STANDARD practices

- always use semi intervals $[a; b[$
- write large constants as product e.g. `1000 * 1000`
- constants should be defined with consts, e.g. `const int MAX_NB_COWS = 42;`
- note precisely which cells you might access in an array
- compute the maximal values for all dimensions
- always use meaningful variable names (e.g. idCow, nbCows, etc.)
- fix function parameters order, e.g. `f(x,y)` and `t[y][x]`
- store the input in global variables / arrays

Know your types!

For integer types, you can expect:

- `char`, **8 bits**, -2^7 to $2^7 - 1$
- `int`, **32 bits**, -2^{31} to $2^{31} - 1$ *not standard*
- `long long`, **64 bits**, -2^{63} to $2^{63} - 1$
- `int128`, **128 bits**, -2^{127} to $2^{127} - 1$

There are also the unsigned versions (only positive numbers).

Know your types!

For integer types, you can expect:

- `char`, **8 bits**, -2^7 to $2^7 - 1$
- `int`, **32 bits**, -2^{31} to $2^{31} - 1$ *not standard*
- `long long`, **64 bits**, -2^{63} to $2^{63} - 1$
- `int128`, **128 bits**, -2^{127} to $2^{127} - 1$

There are also the unsigned versions (only positive numbers).

For float types, we have 1 bit for the sign and:

- `float`, **23 bits** fraction, **8 bits** exponent
- `double`, **52 bits** fraction, **11 bits** exponent
- `long double`, **64 bits** fraction, **15 bits** exponent

Know your types (string)!

C strings

A string in C is an array of `char` ended by a value 0 (also written `'\0'`).

Know your types (string)!

C strings

A string in C is an array of `char` ended by a value 0 (also written `'\0'`).

C++ strings

C strings work in C++ but C++ also has a `string` object. You can use `string(myCstring)` to create a C++ string out of a C string (this will be useful for comparisons!).

Use C+ not C++

C++ is a very complete language:

- object-oriented programming
- templates
- exception handling
- lambda functions

We DON'T want those for competitive programming.

Use C+ not C++

C++ is a very complete language:

- object-oriented programming
- templates
- exception handling
- lambda functions

We DON'T want those for competitive programming.

We want C+, which is C and:

- auto, const, boolean
- references, foreach
- and all of the STL

Your first problems

Reminder on reading input

```
int d ; scanf("%d",&d);  // reads the integer d
double f ; scanf("%lf",&f); // read the double f
char t[256] ; // remember that strings are null
              // terminated when allocating space
scanf("%s",t);  // reads a s string on the input
                // until a space or a \n
scanf("%[^\n]",t);  // reads a string t on the input
                     // until a \n (i.e. does not stop
                     // at a space). DOES NOT READ THE \n
scanf("%[^\n]\n",t);// reads a line, t ends with \0 not \n
scanf("%d %lf\n",&d,&f); // read an int followed by a
     // double and eats the final \n (important if you
     //  want to read a string after)
```

Note that scanf returns the numbers of items read

Reminder on writing output

```
printf("%d\n",42);  // prints 42 and a new line symbol
printf("%s","Hello !");  // prints "Hello !" but
                   // no new line

printf("%lf",42.5);  // prints 42.5
printf("%.2lf",42.5); // prints 42.50
                      // (.2 = 2 digits precision after .)

printf("%02d",2); // prints 02
                  // (%2d means at least 2 digits)
printf("%02d",42); // prints 42
printf("%02d",123); // prints 123 (at least 2 digits)
```

Today’s exercises

Today's exercises

The exercises are simpler in term of algorithm but:

- the input is hard to read
- double-check the types you use