INF280: Competitive programming

Competitive programming

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.

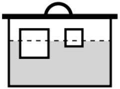
Barrel Example

Baltic OI '03 P1 - Barrel

Time limit: 2.0s **Memory limit:** 64M

Baltic Olympiad in Informatics: 2003 Day 1, Problem 1

Some amount of water is poured into a barrel, then a number of cubes of different size and density are put into water. Finally, a lid is put onto the barrel and pushed down until it touches the edges of the barrel.



Write a program to compute the resulting water level in the barrel.

It can be assumed that:

- the density of water is 1.0,
- the influence of air can be neglected,
- the cubes fit completely into the barrel,
- the cubes do not rotate and do not touch each other.

Input Specification

The first line contains three real numbers - the bottom area of the barrel S $(0 < S \leq 1\,000)$, the height of the barrel H $(0 < H \leq 1\,000)$, and the volume of the water V $(0 < V \leq S \cdot H)$. The next line contains the number of cubes N $(0 < N \leq 1\,000)$. It is followed by N lines, each containing two real numbers describing the cube - the length of a side of the cube L $(0 < L \leq 1\,000)$, and the density of the cube D $(0 < D \leq 10)$.

Output Specification

The first and only line of the output must contain one real number - the resulting water level. The output must not differ from the correct value by more than 10^{-4}.

Sample Input

```
100 10 500
1
1 0.5
```

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

1h30 lesson part

Learn some methods or algorithms and solve some exercise without code, to develop algorithmic thinking.

1h30 coding

Solving exercise with code, to develop fast programming skills.

Grading

Graded exercises in class

There will be 2 to 4 graded exercises without computers.

Final exam

The final exam will be on a computer in a SWERC-like contest.

Final grade

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

ICPC - SWERC

International Collegiate Programming Contest (ICPC)

ICPC is:

- Collegiate: team-based contests with teams of 3 students from the same university
- International: teams from all over the world

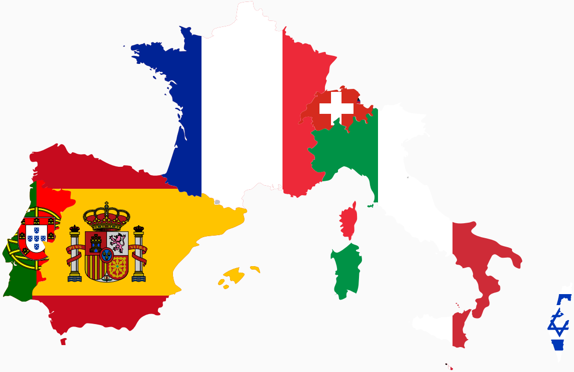
International Collegiate Programming Contest (ICPC)

ICPC is:

- Collegiate: team-based contests with teams of 3 students from the same university
- International: teams from all over the world

The ICPC contest happens over multiple phases:

- University level (to select teams)
- South Western European Regional Contest (SWERC)
- European level (new contest!)
- World finals



Previous editions

- **2017-2021** At Télécom
- **2022-2023** In Milan
- **2023-????** In Jussieu

SWERC-ICPC specificities

- 3 members per team (Telecom sends 2 or 3 teams every year)
- Contest is 5 hours
- One computer per team
- no Internet but some documentation allowed
- a few programming languages (C++, Java, Python3)

See also:

- `https://swerc.eu/2023/regulations/`
- `http://icpc.global/worldfinals/rules`
- `http://icpc.global/worldfinals/programming-environment`

SWERC-ICPC ranking

In ICPC contests, teams are ranked using the following order:

- first we rank by numbers of problems solved,
- then we rank by “total time” to solved those problems,
- the time to solve a problem is computed as $X + 20F$ where
  - X is the number of minutes from the beginning of the contest to the accepted submission
  - F is the number of failed attempts

Usually a SWERC contest comprises around 12 problems.

Final exam

- individual participation for a SWERC-like contest
- 3 hours
- around 6 problems
- one programming language, C++
- no Internet but some documentation allowed

Final exam on the 20th of June afternoon!

Solving SWERC-like 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 SWERC-like 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

1s on a recent CPU is around 5×10^7 loop iterations in C++
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.

Your output does not have the right format (i.e. extra space, caps,

Solving SWERC-like problems

Testing your program

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 SWERC-like 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?

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 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:

- `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 is also the unsigned version (only positive numbers).

Know your types!

For integer types:

- `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 is also the unsigned version (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.

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

Solving SWERC-like problems

Reading and solving a problem

A seven step method

These seven steps will help you find solutions:

- Reformulate / summarize
- Listing dimensions
- Finding good visual representations
- Do examples by hand and represent the solution visually
- Finding a naive algorithm
- Simplify the problem
- Change the point of view

Remember this method!

Step 1: Reformulate and summarize

After carefully reading the problem you should be able to:

- summarize the problem in one (or very few) sentences in the form of a question leaving out all numerical constraints listed in the dimensions (step 1)
- list the parameters or dimensions of the problem (step 2)

Do not hesitate to read the problem multiple times. *Usually* there are no bugs in the subjects, if you don't understand something you have probably missed something.

Reformulation

Reformulation

We have a big rectangle split into many rectangles of known lengths and widths. We know the width of the big rectangle, what is its length?

Step 2: Listing all the dimensions of the problem

In a problem you are often given values in a dimension (it might be the age of a cow, the number of boxes, the number of lanes in a road, etc.).

You should list precisely all of the dimensions

And for each note the minimal/maximal values and whether the order is important.

Different types of dimensions

We can distinguish between input, output or implicit dimensions.

Step 2: Listing all the dimensions of the problem

In a problem you are often given values in a dimension (it might be the age of a cow, the number of boxes, the number of lanes in a road, etc.).

You should list precisely all of the dimensions

And for each note the minimal/maximal values and whether the order is important.

Different types of dimensions

We can distinguish between input, output or implicit dimensions.

Use this step to consider what types might work!

Let us consider Shattered Cake, what dimensions do you find?

Input dimensions

- $1 \leq N \leq 5 \times 10^6$
- $1 \leq W \leq 10^4$
- $1 \leq w_i, l_i \leq 10^4$

Output dimensions

- $1 \leq L \leq 10^4$

Implicit dimensions

- $1 \leq A \leq 10^8$

Let us consider Barrel, what are the dimensions?

Step 3: Finding good visual representations

You can now find a visual representation. Usually there is a pair of dimensions that offer a good representation...

Barrel example

Step 4: Craft examples and solve them by hand

Solving examples by hand has many benefits:

- it provides you with examples to test your program
- your mind is lazy and might find an “algorithm” if you create examples that are complex enough
- you can use your examples with the visual representation to understand some properties of the problem

Step 5: Finding a possibly naive solution

Do you have any algorithm that finds the solution regardless of the time and memory constraints?

You can try to put all the dimensions in the input.

Step 6: Simplifying the problem (dim-DSR)

You can use the list of dimensions to simplify the problem. For each dimension you can try to:

Delete (D)

What happens if remove completely the dimension?

Point in the 2D plane are now points on a 1D line

Set (S)

What happens if set all values in the dimension to a specific value?

All cows are 1 year old

Reduce (R)

What happens if we restrain the amount of possible values?

x is 0 or 1 instead of 0 to 100

Step 6: Simplifying the problem (rules)

If the problem contains constraints that the solution has to follow what happens if you simplify or remove the constraints?

This is a less mechanical way to solve problems...

...but it sometimes makes sense

Step 6 bis: Ranking simplifications

Useless simplifications

Some simplifications simplify the problem too much or lead to a problem that does not really make sense.

Promising simplification

A good simplification keeps the idea of the original problem. If a simplification is just a particular case of the original problem, it makes sense to (temporarily) forget about the original problem and trying to solve this simplification.

Beware: some "simplifications" actually make it harder to find the solution!

Ranking simplifications

Once you have listed all the simplifications you can think of, rank them from most promising to most useless.

Step 6 ter: Using simplifications

Once you have solved a simplified version of the problem you can try to generalize it by:

- using it on the original problem (or a small modification of it)
- using it to solve a part of the problem
- repeating the solution for each possible value
- if two dimensions play the same role you can try applying the simplification in one dimension and then the other
- generalizing the idea that lead to this solution

It is also often a good idea to look at what happens visually in your solution to the simplified version.

Step 7: Adopting a different approach

Usually most problems are disguised but can be solved with a standard algorithm. Instead of trying to find the algorithm for a problem you can list all the classical algorithms and try to use them to solve the algorithm...

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)
```

Let us solve a very simple problem

Our first problem

Problem statement

The input contains on the first line 3 space-separated integers a, b and c. The next two lines contain two strings s_t and s_f with at most 100 characters. Print s_t when $a + b = c$ and s_f otherwise.

Our first problem

Problem statement

The input contains on the first line 3 space-separated integers a, b and c. The next two lines contain two strings s_t and s_f with at most 100 characters. Print s_t when $a + b = c$ and s_f otherwise.

Solution

The problem statement is almost the algorithm :)

C++ solution

```
#include <cstdio>
int main () {
  int a,b,c ;
  char s1[101], s2[101];
  scanf("%d %d %d\n", &a, &b, &c);
  scanf("%[^\n]\n",s1);
  scanf("%[^\n]\n",s2);
  if(a+b==c) {
    printf("%s\n",s1);
  } else {
    printf("%s\n",s2);
  }
  return 0;
}
```

Let us solve this on domjudge

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