Implementation Techniques
and other stuff for practical contests

Michal misof Forišek

Department of Theoretical Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University
Bratislava, Slovakia

February 2012
Talk overview

A collection of tips and tricks:

- editing
- compiling
- testing
- debugging
- implementation
- ... and more
Choose your tools

Programming contests are not only about solving problems.

Hardest part: statement $\rightarrow$ solution idea.

Your goal: spend as much time as possible on the hardest part
In other words: spend as little time as possible on everything else.

What helps: good tools, a good strategy, lots of practice

Language choice for contests: C++ is the winner
Editor

Does the editor matter?
All editors are more or less the same when you write code. The difference appears once you need to edit it.

Essentials
- syntax highlighting
- automatic indentation

Bonuses
- quick and simple searching, replacing, indentation, etc.
- interaction with the compiler
- vim does all of this and more – run vimtutor to get a taste
Compiler: Use warnings!

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

int compute() {
    int a, b;
    cin >> a;
    if (a==0) {
        cout << "zero" << endl;
        return b;
    } else {
        cout << "non-zero" << endl;
    }
}

int main() {
    if (compute()) cout << "success" << endl;
}
```
Compiler: Use warnings!

Compiler output without warnings

(That is, absolutely none!)

Compiler output with g++ -Wall -Wextra warnings.cc

```
warnings.cc:7: warning: suggest parentheses around assignment used as truth value
warnings.cc:13: warning: control reaches end of non-void function
warnings.cc:9: warning: ‘b’ may be used uninitialized in this function
```
Compiler: Use warnings!

Compiler output without warnings

(That is, absolutely none!)

Compiler output with `g++ -Wall -Wextra warnings.cc`

```
warnings.cc:7: warning: suggest parentheses around assignment used as truth value
warnings.cc:13: warning: control reaches end of non-void function
warnings.cc:9: warning: ‘b’ may be used uninitialized in this function
```
The bash shell is your friend

**Input/output redirection**

```
./my_program < task.in > task.my_out
```

**Input straight from the command line**

```
./my_program <<< "5 1 2 3 4 5"
```

Very useful e.g. when writing generators

**Check whether your output is correct**

```
diff task.my_out task.correct
```

(Learn to read diff’s output
or use “diff -y” to see both files side by side.)
The bash shell is your friend

For-cycles, variables, wildcards

```bash
for i in a b c ; do echo $i ; done
for i in *.in ; do echo $i ; done
for i in *.in ; do ./my_program < $i ; done
```

Sequences

```bash
seq $start [$end [$step]]
```

For example:

- `seq 47` prints 1 to 47
- `seq 1 12 3` prints 1 4 7 10

Expressions

```bash
echo $(( 4 + ( 7 * 1 ) ))
```
The bash shell is your friend

A complete script testall.sh

#!/bin/bash
for infile in *.in ; do
    echo $infile

    name='basename $infile .in'
    outfile=$name.out
    myfile=$name.my

    time ./my_program < $infile > $myfile
diff -q $myfile $outfile
done

Make it executable
chmod a+x testall.sh
Simple word frequencies

cat $f | sed -e 's/ /\n/g' | sort | uniq -c | sort -n

Output:

...  
941 and  
991 to  
1555 of  
1995 the

Primes

seq 2 100 | factor | grep -v '[^-:]' | sed -e s/::.*//
Debugging 1: the real deal

Knowing a debugger may be an advantage.
Compile with the -g switch

A simple gdb session

```
$ g++ error.cc -g -o error
$ ./error
Floating point exception
$ gdb ./error
(gdb) run
Starting program: /home/misof/error
Program received signal SIGFPE, Arithmetic exception.
0x0804841a in boo () at error.cc:2
2 void boo() { --x; y/=x; } 
(gdb) print x
$1 = 0
```
Debugging 2: valgrind

valgrind: a smart tool, available at IOI, learn to use it

```cpp
#include <cstdlib>

void get_memory() {
    int *a = new int[100];
    int *b = (int*)calloc(100, sizeof(int));
}

int main() { for (int i=0; i<100; ++i) get_memory(); } 
```

valgrind ./memory_leak (truncated)

```
==21639== HEAP SUMMARY:
==21639== in use at exit: 80,000 bytes in 200 blocks
==21639== total heap usage: 200 allocs, 0 frees, 80,000 bytes allocated
==21639==
==21639== LEAK SUMMARY:
==21639== definitely lost: 80,000 bytes in 200 blocks
```
Debugging 2: valgrind

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> A(10,0);
    for (int i=4;i<=10;++i) A[i]=i;
    std::cout << A[12] << "\n";
}
```

`valgrind ./out_of_range (truncated)`

```
==22347== Invalid write of size 4
==22347== at 0x8048789: main (out_of_bounds.cc:6)
==22347==
==22347== Invalid read of size 4
==22347== at 0x804878B: main (out_of_bounds.cc:7)
```
Debugging 2: valgrind

```c
int x=2, y=47;
void boo() { --x; y/=x; }
void foo() { boo(); boo(); }
int main() { foo(); }
```

valgrind ./error (truncated)

```
==23331== Process terminating with default action of signal 8 (SIGFPE)
==23331== Integer divide by zero at address 0x62DA948F
==23331== at 0x804841A: boo() (error.cc:2)
==23331== by 0x804842B: foo() (error.cc:3)
==23331== by 0x804843A: main (error.cc:4)
```
assertions = checks that the data is still sane

**Assertions in C++**

```cpp
#include <cassert>
...
int x = foo();
assert((x>=0) && (x<N));
```

... and the code is executed

```cpp
assert: assert.cc:8: int main():
Assertion ‘(x>=0) && (x<N)’ failed.
```

asserts cost you **nothing**: just add “#define NDEBUG” before “#include”s to disable them.
Debugging 3: asserts

Assertions in FreePascal

```pascal
{C+}
var x : longint;
...
x := foo();
assert( (x>=0) and (x<N) );
```
Never delete debug outputs – just make them inactive!

**Debug outputs using the preprocessor**

```c
x := foo();
#ifndef NDEBUG
    cerr << "x: " << x << endl;
#endif
```

**A handy macro**

```c
#ifdef NDEBUG
    #define DEBUG(x)
#else
    #define DEBUG(x) cerr << #x << "": " << (x) << endl;
#endif
```
Avoid Copy&Paste like the Plague

Copy and Paste

- one of the most frequent bug sources
- produces long code:
  - hard to read, hard to modify
- if you introduce a bug, it’s impossible to find
- almost never necessary!

How to avoid it?

- implement each functionality once, and once only
- one option: wrap it in a function
- another option: replace it with a loop
Copy&Paste case study: Maze exploration

navigating a 4-connected maze

```c
int dr[] = {-1, 0, 1, 0};
int dc[] = {0, 1, 0, -1};

// generate all 4 cells reachable from (r,c):
for (int dir=0; dir<4; ++dir) {
    int nr = r + dr[dir];
    int nc = c + dc[dir];
    ...
}

// Note: (dir+1) % 4 is the next direction clockwise
```

knight moves?

```c
int dr[] = {-2, -2, -1, -1, 1, 1, 2, 2};
int dc[] = {-1, 1, -2, 2, -2, 2, -1, 1};
for (int dir=0; dir<8; ++dir) ...
```
Copy&Paste case study: Maze exploration

navigating a 4-connected maze

```c
int dr[] = {-1, 0, 1, 0};
int dc[] = {0, 1, 0, -1};

// generate all 4 cells reachable from (r,c):
for (int dir=0; dir<4; ++dir) {
    int nr = r + dr[dir];
    int nc = c + dc[dir];
    ...
}

// Note: (dir+1) % 4 is the next direction clockwise
```

knight moves?

```c
int dr[] = {-2, -2, -1, -1, 1, 1, 2, 2};
int dc[] = {-1, 1, -2, 2, -2, 2, -1, 1};
for (int dir=0; dir<8; ++dir) ...
```
Sentinels

Special cases are bad:
– you are forced to write more code
– you may make more bugs

An useful technique: sentinels

idea: add new data with extremal values
result: each original item is processed in the same way

Example #1

- data: a sorted array
- goal: find the number of unique elements
- sentinels: add “∞” at the end
- gain: one for-cycle with no special cases
Sentinels

Example #2
- data: a sorted array
- goal: binary searching for many $x$s
- sentinels:
  - add a $\text{–}\infty$ value at the beginning,
  - add a $\infty$ at the end
- gain: easier binary search: $x$ is always inside

Example #3
- data: halfplanes
- goal: compute their intersection
- sentinels: start with a huge bounding box
- gain: no infinity as a special case
Sentinels

Example #4

- data: a bitmap of a maze
- goal: exploration
- sentinels: add a row/column of walls at each side
- gain: no need for checks like

  ```
  if ((r>=0) && (r<R) && (c>=0) && (c<C)) ...
  ```

```
########
..#... #..#...
#...#. ##...#.#
..#.#. ---> #..#.#.
.##..# #.##..##
..##.. #..##..#
```

```
########
```
Binary search

Binary search is easy:

```cpp
int binary_search(const vector<int> &array, int value) {
    // initialize pointers to the first and last element
    int start = 0, end = array.size()-1;
    // check whether value falls outside of the array
    if (value < array[start]) return -1;
    if (value > array[end]) return -1;
    // while we have multiple choices, halve the interval
    while (start != end) {
        int middle = (start+end)/2;
        if (array[middle] < value) start = middle; else end = middle;
    }
    if (array[start] == value) return start; else return -1;
}
```
int binary_search(const vector<int>& array, int value) {
    int start = 0, end = array.size() - 1;
    if (value < array[start]) return -1;
    if (value > array[end]) return -1;
    while (start != end) {
        int middle = (start + end) / 2;
        if (array[middle] < value) start = middle; else end = middle;
    }
    if (array[start] == value) return start; else return -1;
}

Does not even work for values actually present!
Example: array[]={0,10,20,30,40}, value=30
(start, end): (0, 4) → (2, 4) → (2, 3) → (2, 3) → ···
Half-open intervals

The previous example

bug type: ±1 errors
how to avoid: always see a clear invariant
one helpful technique: half-open intervals

What's a half-open interval?

\([a, b) = \{x \mid a \leq x < b\}\)
Read: \(a\) is the first number inside, \(b\) the first one outside

Useful to learn: used e.g. in STL, in Python
in general, they lead to code with few ±1s
Half-open intervals

**Basic properties**

Length: $b - a$ (also the number of integers in range)

Natural representation of an empty range: $[a, a)$.

For any $c$ such that $a < c < b$ we can split interval $[a, b)$ into $[a, c)$ and $[c, b)$.

**Example: binary search**

- In the beginning:
  make sure that $\text{array}[a] \leq \text{value} < \text{array}[b]$.
- When to terminate:
  as soon as $b - a = 1$: now $a$ is the only candidate left
- How to proceed if $b - a > 1$:
  split $[a, b)$ into $[a, c)$ and $[c, b)$ for $c = (a + b) \div 2$
Half-open intervals

**Basic properties**

Length: \( b - a \)  
(also the number of integers in range)

Natural representation of an empty range: \([a, a)\).  

For any \( c \) such that \( a < c < b \) we can split interval \([a, b)\) into \([a, c)\) and \([c, b)\).

**Example: binary search**

- In the beginning:  
  make sure that \( \text{array}[a] \leq \text{value} < \text{array}[b] \).

- When to terminate:  
  as soon as \( b - a = 1 \): now \( a \) is the only candidate left

- How to proceed if \( b - a > 1 \):  
  split \([a, b)\) into \([a, c)\) and \([c, b)\) for \( c = (a + b) \div 2 \)
Half-open intervals

Fixed binary search

```cpp
int binary_search(const vector<int> &array, int value) {
    // ensure the precondition
    if (value < array[0]) return -1;
    // set the bounds
    int a = 0, b = array.size();

    // do the search
    while (b-a > 1) {
        int c = (a+b)/2;
        if (array[c] <= value) a=c; else b=c;
    }
    if (array[a] == value) return a; else return -1;
}
```

Note: we divided the array into a “good” and a “bad” part.
Half-open intervals

Prefix sums: the problem
You have: an unsorted array \(A[0..N-1]\) of numbers
You want: quickly determine sum of any segment

Prefix sums: idea of the solution
\((A[i] + \cdots + A[j]) = (A[0] + \cdots + A[j]) - (A[0] + \cdots + A[i-1])\)

Prefix sums: the solution
Definition: Let \(S[i] = A[0] + \cdots + A[i-1]\).
Note: \(S[i]\) is the sum of elements of \(A\) with indices in \([0, i)\).
Computation in \(O(N)\): \(S[0] = 0\) and \(S[k+1] = S[k] + A[k]\).
Sum of segment with indices in \([a, b)\): simply \(S[b] - S[a]\).
Half-open intervals

**Prefix sums: the problem**

You have: an unsorted array $A[0..N - 1]$ of numbers
You want: quickly determine sum of any segment

**Prefix sums: idea of the solution**

$$(A[i] + \cdots + A[j]) = (A[0] + \cdots + A[j]) - (A[0] + \cdots + A[i − 1])$$

**Prefix sums: the solution**


Note: $S[i]$ is the sum of elements of $A$ with indices in $[0, i)$.


Sum of segment with indices in $[a, b)$: simply $S[b] − S[a]$. 
Half-open intervals

**Prefix sums: the problem**

You have: an unsorted array \( A[0..N - 1] \) of numbers  
You want: quickly determine sum of any segment

**Prefix sums: idea of the solution**

\[
(A[i] + \cdots + A[j]) = (A[0] + \cdots + A[j]) - (A[0] + \cdots + A[i - 1])
\]

**Prefix sums: the solution**

Definition: Let \( S[i] = A[0] + \cdots + A[i - 1] \).

Note: \( S[i] \) is the sum of elements of \( A \) with indices in \([0, i)\).

Computation in \( O(N) \):

\[
S[0] = 0 \text{ and } S[k + 1] = S[k] + A[k].
\]

Sum of segment with indices in \([a, b)\):

simply \( S[b] - S[a] \).
Contest strategy

Write a brute force solution!

- scores points!
- usually easy to implement (bitsets, next_perm)
- use it to test your faster solution (if any)
- combine both to be sure
- if enough time, write a generator as well

Optimizations?

- never prematurely!
- **never** overwrite, always back up a working version
- always compare both versions