Contest Tips and Tricks

Maxim Buzdalov

ITMO UNIVERSITY

October 28, 2015

The aim of the talk

- Programming competitions are about making things efficiently
  - Obvious when talking about programs and algorithms
  - Also true about what you do at the contest
    - reading statements
    - writing code
    - reading code
    - finding bugs

Outline

Coding styles and conventions

Correct types for your variables

Correct use of language

Similarity

Use of standard library

Use of IDE

Coding styles and conventions

Correct types for your variables

Correct use of language

Similarity

Use of standard library

Use of IDE

Coding style and conventions

- Determine and use the conventions that will be used in your team
  - You can use any reasonable convention, but the same for all the team
- If all team members write the code similarly, you understand your teammate's code faster
- Examples:
  - Java Coding Conventions
  - Kernigan & Ritchie's style
- Modify them if the changes make code better

Coding style and conventions

- Piece of code from team X

```
if(i*j+j*k+i*k<ans) ans=i*j+j*k+k*i, ai=i, aj=j, ak=k;
```

- Another piece of code from the same team

```
while (lb + 1 < rb) {
    int mid = (lb + rb) >> 1;
    int p = mid * mid / 2;
    int q = mid * mid - p;
    if (p <= b && q <= w) lb = mid;
    else rb = mid;
}
```

- Blind guess: different authors

Coding style and conventions

- Four ways of using braces in one file!

```
int nbEtages, start, endPos, up, down;
int tMin[2000*1000];

bool valideVide(int p)
{ return p > 0 && p <= nbEtages && tMin[p] == -1; }

int main() {
    ios_base::sync_with_stdio(false);
    freopen("elevator.in", "r", stdin);
    freopen("elevator.out", "w", stdout);

    ...

    while (!actuals.empty())
    {
        ...
    }

    if (tMin[endPos] == -1)
        cout << "use the stairs";
    else
        cout << tMin[endPos];
}
```

Coding styles and conventions

Correct types for your variables

Correct use of language

Similarity

Use of standard library

Use of IDE

Most used integers in C++

- `int`: 32-bit signed integer
  - $[-2^{31} \ldots 2^{31} - 1]$
  - roughly $[-2 \cdot 10^9 \ldots 2 \cdot 10^9]$
  - `long` in 32-bit mode
- `long long`: 64-bit signed integer
  - $[-2^{63} \ldots 2^{63} - 1]$
  - roughly $[-9 \cdot 10^{18} \ldots 9 \cdot 10^{18}]$
  - `long` in 64-bit mode

How to choose integers?

- Don't use `int` blindly!
- Estimate the possible value:

significantly less than $2 \cdot 10^9$	$\rightarrow$	`int`
may be close to $2 \cdot 10^9$	$\rightarrow$	think twice!
significantly less than $9 \cdot 10^{18}$	$\rightarrow$	`long long`
may be close to $9 \cdot 10^{18}$	$\rightarrow$	think twice!
bigger	$\rightarrow$	big integers

An ultimate solution to overflows?

```
long long _();
int main() { return _(); }
#define int long long
#define main _
```

An ultimate solution to overflows?

```
long long _();
int main() { return _(); }
#define int long long
#define main _
```

- OK, but beware of memory limits!
- When would it hit you?
  - Code with large int32 arrays and intermediate int64 computations

How to find where overflows are?

- Symptom: Wrong Answer (or sometimes Runtime Error) on big test numbers
- Where?

How to find where overflows are?

- Symptom: Wrong Answer (or sometimes Runtime Error) on big test numbers
- Where? A hard but a good way:
  - Print the code and sit somewhere
  - Annotate each integer variable with an interval of possible values
    - Beware! Non-constant variables may have different intervals in different times
    - Maybe an interval is a function of iteration number etc.
  - For each operation:
    - Add/subtract/multiply/divide intervals
    - If the variable domain does not cover the interval for the result, either prove formally it cannot happen, or you have a bug there

Integer with positive infinity

- A useful abstraction for solving some problems
  - i.e. the Bellman-Ford algorithm
- The maximum value (like $2^{31} - 1$ for `int`) is said to be ∞
- Need to implement basic operations (example: addition)
  - $a + \infty = \infty$
  - $\infty + a = \infty$
- Make sure that $a + b$ does not overflow

Integer with positive infinity

- A useful abstraction for solving some problems
  - i.e. the Bellman-Ford algorithm
- The maximum value (like $2^{31} - 1$ for `int`) is said to be ∞
- Need to implement basic operations (example: addition)
  - $a + \infty = \infty$
  - $\infty + a = \infty$
- Make sure that $a + b$ does not overflow
- May have $-\infty$ as well

Coding styles and conventions

Correct types for your variables

Correct use of language

Similarity

Use of standard library

Use of IDE

Correct use of language

- Constant declarations are your friends!

```
char M[1002][1002];
lli A[1002][1002];
bool E[1002][1002];
```

Correct use of language

- Constant declarations are your friends!

```
const int SIZE = 1002;

char M[SIZE][SIZE];
lli A[SIZE][SIZE];
bool E[SIZE][SIZE];
```

Correct use of language

- This relates to macros as well!

```
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32) ||
     defined(__WIN32__) || defined(WIN64) || defined(_WIN64) ||
     defined(__WIN64) || defined(__WIN64__)
  #define debug(a) ;
  #define LL "%I64d"
#else
  #define debug(a) cerr << #a << " = " << (a) << endl;
  #define LL "%lld"
#endif

...

#if defined(WIN32) || defined(_WIN32) || defined(__WIN32) ||
     defined(__WIN32__) ||  defined(WIN64) || defined(_WIN64) ||
     defined(__WIN64) || defined(__WIN64__)
  freopen(filename ".in", "r", stdin);
  freopen(filename ".out", "w", stdout);
#endif
```

Correct use of language

- This relates to macros as well!

```
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32) ||
    defined(__WIN32__) || defined(WIN64) || defined(_WIN64) ||
    defined(__WIN64) || defined(__WIN64__)
  #define ON_WINDOWS
#endif

#ifdef ON_WINDOWS
  #define debug(a) ;
  #define LL "%I64d"
#else
  #define debug(a) cerr << #a << " = " << (a) << endl;
  #define LL "%lld"
#endif

...

#ifdef ON_WINDOWS
  freopen(filename ".in", "r", stdin);
  freopen(filename ".out", "w", stdout);
#endif
```

No code reuse → bad

```
if((temp.first-1 >= 0)&&(field[temp.first-1][temp.second] != '#')
    &&(not(mark[temp.first-1][temp.second])))
{
    temptoadd.first = temp.first-1;
    temptoadd.second = temp.second;
    tovisit.push(temptoadd);
}
if((temp.first+1 < n)&&(field[temp.first+1][temp.second] != '#')
    &&(not(mark[temp.first+1][temp.second])))
{
    temptoadd.first = temp.first+1;
    temptoadd.second = temp.second;
    tovisit.push(temptoadd);
}
if((temp.second-1 >= 0)&&(field[temp.first][temp.second-1] != '#'
    )&&(not(mark[temp.first][temp.second-1])))
{
    temptoadd.first = temp.first;
    temptoadd.second = temp.second-1;
    tovisit.push(temptoadd);
}
if((temp.second+1 < n)&&(field[temp.first][temp.second+1] != '#')
    &&(not(mark[temp.first][temp.second+1])))
{
    temptoadd.first = temp.first;
    temptoadd.second = temp.second+1;
    tovisit.push(temptoadd);
}
```

Code reuse → good

```
int dx[] = {1, 0, -1, 0};
int dy[] = {0, 1, 0, -1};

for (int d = 0; d < 4; ++d)
{
    int nx = temp.first + dx[d];
    int ny = temp.second + dy[d];
    if (nx >= 0 && nx < n && ny >= 0 && ny < n &&
        field[nx][ny] != '#' && !mark[nx][ny])
    {
        temptoadd.first = nx;
        temptoadd.second = ny;
        tovisit.push(temptoadd);
    }
}
```

Coding styles and conventions

Correct types for your variables

Correct use of language

Similarity

Use of standard library

Use of IDE

Similar things should be similar

- Similar pieces of code written in different ways may hide bugs
- Consistency in the code is good:
  - you understand the code faster
  - bugs often introduce inconsistency
    - and you can spot them with higher probability
- Develop a style which helps spotting bugs

Example: segment intersection test

```
boolean intersect(Point src1, Point trg1,
                  Point src2, Point trg2) {
    if (max(src1.x, trg1.x) < min(src2.x, trg2.x) ||
        max(src1.y, trg1.y) < min(src2.y, trg2.y) ||
        max(src2.x, trg2.x) < min(src1.x, trg1.x) ||
        max(src2.y, trg2.y) < min(src1.y, trg1.y)) {
        return false;
    }
    int vmul00 = src2.sub(src1).vmul(trg1.sub(src1));
    int vmul01 = src2.sub(src1).vmul(trg2.sub(src1));
    int vmul10 = trg2.sub(trg1).vmul(src1.sub(trg1));
    int vmul11 = trg2.sub(trg1).vmul(src2.sub(trg1));

    return signum(vmul00) * signum(vmul01) <= 0 &&
           signum(vmul10) * signum(vmul11) <= 0;
}
```

Coding styles and conventions

Correct types for your variables

Correct use of language

Similarity

Use of standard library

Use of IDE

Know and love your standard library

- Don't code a thing from scratch if the library has it
- The library code is efficient (often)
- ...and has no bugs (always)

Know and love your standard library

Spot the bug. My teammates didn't for two hours!

```
for (int i = 0; i < n; ++i) {
    for (int j = 1; j < n; ++j) {
        if (a[i - 1] > a[i]) {
            int tmp = a[i];
            a[i] = a[i - 1];
            a[i - 1] = tmp;
            tmp = b[i];
            b[i] = b[i - 1];
            b[i] = tmp;
        }
    }
}
```

Know and love your standard library

Dijkstra algorithm with heap on priority queue

- Runs in $O(E \log E)$ instead of $O(E \log V)$, almost no change
- Faster than implementation using ordered sets
- No need to implement a heap with decreaseKey

Know and love your standard library

```java
class Record implements Comparable<Record> {
    final int vertex;
    final int distance;
    Record(int vertex, int distance) {
        this.vertex = vertex;
        this.distance = distance;
    }
    public int compareTo(Record that) {
        return Integer.compare(distance, that.distance);
    }
}

PriorityQueue<Record> q = new PriorityQueue<>();
q.add(new Record(start, 0));
while (!q.isEmpty()) {
    Record curr = q.remove();
    if (dist[curr.vertex] == curr.distance) {
        for (Edge e : graph[curr.vertex]) {
            int nd = curr.distance + e.length;
            if (dist[e.target] > nd) {
                dist[e.target] = nd;
                q.add(new Record(e.target, nd));
            }
        }
    }
}
```

Coding styles and conventions

Correct types for your variables

Correct use of language

Similarity

Use of standard library

Use of IDE

Use of IDE

- Code completion
  - can speed up coding if used appropriately
- Error highlighting and background compilation
  - can save your time, especially in last few minutes
- Static and dynamic analysis
  - can help to find bugs in the code while typing
- Navigation
  - find the variable or function declaration faster
- Refactoring
  - helps to keep code clear when solving large technical problems

Example: static analysis helps

```
Queue<Integer> qx = new ArrayDeque<>();
Queue<Integer> qy = new ArrayDeque<>();
qx.add(0);
qy.add(0);
while (!qx.isEmpty()) {
    int x = qx.remove();
    int y = qx.remove(); // the bug is here!
    for (int d = 0; d < 4; ++d) {
        int nx = x + dx[d];
        int ny = y + dy[d];
        if (!used[nx][ny] && !field[nx][ny]) {
            qx.add(nx);
            qy.add(ny);
            used[nx][ny] = true;
        }
    }
}
```

Warning: The contents of collection are updated, but never queried