Efficient implementation and debugging

The beOI Instructors

November 8, 2018

OLYMPIADE BELGE D'INFORMATIQUE
BELGISCHE INFORMATICA-OLYMPIADE

Why programming tricks?

- During contests, you will have to solve a lot of subtasks when you cannot solve the full problems.
- You have to be able to do this efficiently because you will need as much time as possible to solve other problems/subtasks.
- Here are a few tricks to help you solve problems efficiently.

Bit-level operators

As you probably know, computers represent numbers in base two, that is with `0`s and `1`s. Some C++ operators are designed to make use of this. There are shift operators to quickly multiply or divide by powers of 2 and bitwise operators to act on the bits of a number separately.

Bitshit operators

There are two bitshift operators.

- `a<<b` Shift *a*, *b* bits to the left. This means that *a* is basically divided by 2^b and then rounded down.
- `a>>b` Shift *a*, *b* bits to the right. This means that *a* is basically multiplied by 2^b.

Warning

Type limits apply here. Also, if *b* is bigger than the number of bits of the type, these two operations are not defined anymore, which means they could return an arbitrary result. Be very careful if you are using variables as shift parameters.

Bitwise operators

There are four bitwise operators.

- `~a` Bitwise NOT. 1 if the input is 0. 0 if the input is 1.
- `a|b` Bitwise OR. 1 if at least one of the inputs is 1. 0 if both inputs are 0.
- `a^b` Bitwise XOR. 1 if both inputs are not equal. 0 if both inputs are equal.
- `a&b` Bitwise AND. 1 if both inputs are 1. 0 if at least one of the inputs is 1.

Bit-level operators

Warning

Bit-level operators have a very low priority compared to other operators, which means you should put them inside parentheses or even create separate functions for them. For example, `a&1 == 0` is actually equivalent to `a&(1 == 0)`, which is always `false`.

There is a difference between `&|` and `&&||` for `bool`s. The former always evaluate both operands while the latter *short circuit*: they only evaluate the right operand if necessary. This allows you to write `if(i < n && arr1[i] < arr2[i])`, for example.

Iterate over all subsets

Use a for loop to iterate over all 2^n subsets.

```
for(int i = 0; i < (1 << n); ++i) {
  /* ... */
  if((i&(1 << j)) > 0) { // check if j is part of subset ‘i‘

  }
  /* ... */
 }
```

How do we exclude the empty subset? ($\{\}$)

We start the loop at 1.

Iterate over all permutations

Use `next_permutation()` to iterate over all $n!$ permutations.

```
vector<int> perm(n);
for(int i = 0; i < n; ++i) perm[i] = i;
do {
  /* Use permutation */
 } while(next_permutation(perm.begin(), perm.end());
```

How do we iterate over all combinations (pick k elements among n)?

Use permutations of k `1`s and $n-k$ `0`s.

Debugging

Why use a debugger (gdb)?

- It allows you to locate where the program segfaults.
- It allows you to print any global and local variable, in any function call.
- It allows you to set breakpoints to inspect variables at any point in time.
- Much more...

Learn to use it now!

Debugging

How to setup a debugger?

- Install `gdb` if you don't have it already. Do it now!
- Add a `-g` flag to your compilation command. For example, `g++ -std=c++14 -g main.cpp`.
- Open `gdb`

Gdb commands

Common gdb commands

`file <name>`	Load file
`r`	Run program
`^C`	Terminate program
`q`	Quit gdb
`print <x>`	Print variable
`bt`	Print backtrace
`up`	Move one frame (one function call) up
`down`	Move one frame (one function call) down

Download `examplecode.cpp` and try these functions now.

Effective debugging

Great, now you know how to inspect your program? What to do if your program doesn't behave as expected, be it WA on a sample or on the server.

- Step 1: Always, always find a test case that breaks your code. As long as you didn't prove that your code isn't correct by building a killer test case, you cannot check that the bug disappeared. I like to name the test cases `x.in` and `x.out` where `x` is the test case number.
- Step 2: Fix the bug by using gdb to find where the code fails.
- Step 3: Test all the test cases you have so far to make sure you did not reintroduce a previous bug.
- Repeat until you get AC.

Effective debugging

What should you do if you cannot find a killer test case or if the test cases are so big that you have no way to compute the result by hand?

- Read the problem statement once again.
- Write a bruteforce to test the code (maybe you already have one from a previous subtask).
- If you don't have the time to write a bruteforce, try to find invariants in the input and change the input accordingly. For example, the order of the elements for the knapsack problem is irrelevant, so you could try permutating the elements in the input.[1]

[1] Credits goes to errichto.

Editor choice

- Learn to use Vim now and use it for competitive programming.
- Vim is not an IDE, but it's one of the best text editors.
- Vim has incredibly short keybindings for a lot of editing operations.
- The learning curve is extremely steep, but it's totally worth it. Try not to use arrow keys. *Not even in insert mode.*
- (Bonus) When you will need an IDE and not a text editor, you will be ready to use Spacemacs!

Templates

Use a template. Write it once at the beginning of the contest and reuse it for every single problem. Now, you may be wondering what uses there are for a template.

Killer features

- Macros help you prevent stupid errors like `for(int i = 0; i < n; ++j)` while you code.
- Macros allow you to type less code to get the same result. Compare `V<V<int>>` and `vector<vector<int>>`.
- Macros provide a consistent way to print debug output, combined with a global switch to disable them.

Example template

```
#include <bits/stdc++.h>
#define int long long
#define P(x) do {if(debug) cout << x << endl;} while(false)
#define H(x) P(#x << ": " << x)
#define FR(i, a, b) for(int i = (a); i < (b); ++i)
#define F(i, n) FR(i, 0, n)
#define DR(i, a, b) for(int i = (b); i --> (a);)
#define D(i, n) DR(i, 0, n)
#define S(s) (int)(s).size()
#define ALL(x) (x).begin(), (x).end()
#define MI(x, a) (x) = min(x, a)
#define MA(x, a) (x) = max(x, a)
#define V vector
#define pb push_back
#define mt make_tuple
using namespace std;
const bool debug = 1;

signed main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);

  return 0;
}
```

Aliases

Always use an alias to compile. An alias is a shortcut to some command. You'll probably want to use `-fsanitize=address,undefined` to quickly discover issues with your code, but then you'll want another alias for gdb specifically. Put the following in your `.bashrc`:

```
alias g='g++ -std=c++14 -Wall -Wextra -Wshadow -Wfatal-errors -O2
 -fsanitize=undefined,address'
alias gd='g++ -std=c++14 -Wall -Wextra -Wshadow -Wfatal-errors -g'
```

Usage: `g filename.cpp`.