Как писать код

Олимпиадные оптимизации кода

В этом блоке вы сможете найти список приёмов, которые помогают в олимпиадном программировании. Эти приёмы могут или ускорять программу, или упрощать её написание. Итак, поехали!

Ускорение кода

Поскольку очень важно, чтобы программа уложилась в ограничения по времени, то код ускоряют, отключая поддержку функций ввода-вывода из C (после этого нужно пользоваться только `cin` и `cout`). Ускоряется программа вот таким кодом (его помещают в самое начало `main`):

```
ios_base::sync_with_stdio(false);
cin.tie(nullptr);
cout.tie(nullptr);
```

Можно все выражения в скобках заменить на `0`, и это тоже будет работать.

`#include`

В олимпиадном движении не модно запоминать название файлов, в которых содержатся встроенные функции. Поэтому, всегда пишите, как в примере ниже (к тому же, от этого не страдает скорость выполнения программы).

```
#include <bits/stdc++.h>
```

Вместо, того, чтобы заниматься чем-то таким:

```
#include <iostream>
#include <cmath>
#include <vector>
#include <set>
#include <algorithm>
// И ещё много-много ...
```

Подробнее вы можете об этом прочитать в блоке первая программа.

`#define`

Макросы позволяют сокращать код и их нужно писать под себя, но вот примеры полезных макросов:

```
#define FOR_(i, s, n) for(int i = s; i < n; ++i)
#define FOR(i, n) FOR_(i, 0, n)
#define FORR(x, a) for(auto &x : a)
#define IN(x) FORR(y, x) cin >> y;
#define OUT(a) FORR(x, a) cout << x << " "; cout << "\n";
#define OUTM(m) FORR(x, m) cout << x.first << " : " << x.second << "\n";
#define FAST ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define F first
#define S second
```

Подробнее вы можете об этом прочитать в блоке макросы.

`using`

С помощью слова `using` можно сокращать названия типов данных. Например:

```
using ll = long long;
using pi = pair<int, int>;
using vi = vector<int, int>;
```

Подробнее вы можете об этом прочитать в блоке макросы.

`namespace`

Аналогично никто не хочет писать что-то длинное, например такое:

```
std::sort
std::cin
std::cout
std::min
// И ещё много-много ...
```

Поэтому делают оптимизацию ниже.

```
using namespace std;
```

Подробнее вы можете об этом прочитать в блоке первая программа.

`double`

Если вы когда пробовали запускать код ниже, то знаете, что он выведет `0`:

```
cout << 1 / 3 + 1 / 3 + 1 / 3;
```

И, как говорилось в блоке операторы, чтобы программа вывела `1`, нужно во делить не целое число, а использовать тип `double`. Но у `double` не обязательно указывать нули после точки, поэтому можно писать так:

```
cout << 1. / 3 + 1. / 3 + 1. / 3;
```

И такой код уже выведет `1`.

`INF`

Часто в алгоритмах нам будет нужна константа обозначающая бесконечность (`INF`, она уже встречалась в разделе макросы). Значение `INF` подбирается в зависимости от ограничений в задаче, а именно `INF` должен быть на столько большим, чтобы точно не являться ответом к задаче. Объявлять `INF` можно как хочется (хоть через макросы, хоть не как константу), например, так:

```
const int INF = 2e9 + 1024;                    // где-то в верху файла, чтобы везде пользоваться
```

Также работать с бесконечностью нужно аккуратно, чтобы не получилось `INF + 5`, `2 * INF`, или, ещё хуже, `INF * INF` (так как операции с бесконечностью могут не влезть в тип данных, он переполнится, и может получиться число, меньшее бесконечности).

Файловый ввод-вывод

Иногда бывает нужно работать с файлами (читать входные данные или записывать выходные). Если делать это совсем честно, то в C++ нужно писать вот так:

```
#include <fstream>                          // для ifstream и ofstream
ifstream fin("in.txt");                     // чтение из файла
ofstream fout("out.txt");                   // запись в файл
int n;
fin >> n;                                   // считали число
fout << n;                                  // записали число
```

Но писать дополнительные строчки с кодом никто не любит, поэтому можно *«перенаправить»* встроенные потоки для записи / чтения (`cin` и `cout`):

```
freopen("in.txt", "r", stdin);              // перенаправили cin
freopen("out.txt", "w", stdout);            // перенаправили cout
int n;
cin >> n;                                   // считали число
cout << n;                                  // записали число
```

`#pragma`

В очень редких случаях может быть полезно подсказать компилятору, как оптимизировать код. В таком случае в самом начале файла нужно добавить `#pragma`, какие именно добавлять – вопрос сложный, так как для полного понимания нужно хорошо знать устройство компьютера. Можно использовать например такие:

```
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
```

Дебаг

Случается, что код написали, а он почему-то не работает. В таких случаях приходится его в каком-то виде дебажить. Упомяну про два очень известных способа искать ошибки в своём коде, и один не дружелюбный, зато максимально гибкий.

Первый известный способ – добавить так называемые *«debug print»* – вывод переменных, информации о запуске программы и т.п. в консоль. У такого способа понятны минусы – во-первых нужно заранее угадать в каких местах нужно делать выводы, чтобы понять что сломалось; во-вторых, чтобы сделать выводы в нужных местах, придётся пересобрать программу; и в третьих, нет функционала прервать программу в какой-то точке выполнения, можно только дождаться её завершения. В целом для прерываний в произвольном месте можно добавить *«debug input»*. Но если вы всё же решили идти по этому пути, то в самом начале коде можно добавить строку:

```
#define DEBUG
```

И потом каждый вывод окружать макросами:

```
#ifdef DEBUG
cout << "Hello world";
#endif
```

Или же можно сделать чуть более хитрую конструкцию из макросов:

```
#ifdef DEBUG
#define DEBUG_OUT(x) cout << x;
#else
#define DEBUG_OUT(x)
#endif
```

И каждый раз делать вывод как:

```
DEBUG_OUT("Hello world")
```

Цель всех этих макросов в том, чтобы можно было отключить все ненужные выводы закомментировав всего одну строку (которая представлена в самом первом кусочке кода).

Теперь немного скажу про второй известный способ – всякие дебагеры, встроенные в IDE. Сам я ими никогда не пользовался, но потенциально, если научиться, то ошибки могут искаться быстрее чем прошлым методом. В любом случае, это точно более масштабируемое и переносимое решение.

И теперь немного о грустном. В реальной жизни не всегда существует IDE и графический интерфейс. Поэтому приходится пользоваться консольным дебагером *«gdb»*. По нему, конечно же, всего не расскажешь в короткой заметке, но вот маленький список его команд:

Команда	Альтернатива	Эффект
`g++ main.cpp -o ./solve -g`		Компиляция, дебагеру нужен флаг `-g`.
`gdb ./solve`		Запустить дебагер
`b main`	break	Создать точку останова на функции `main`
`r`	run	Запустить до первой точки останова
`p var`	print	Вывести переменную `var` (можно контейнеры)
`c`	continue	Продолжить до следующей точки останова
`s`	step	Следующая инструкция, входит в функции
`n`	next	Следующая инструкция, не входит в функции
`q`	quit	Выключить дебагер