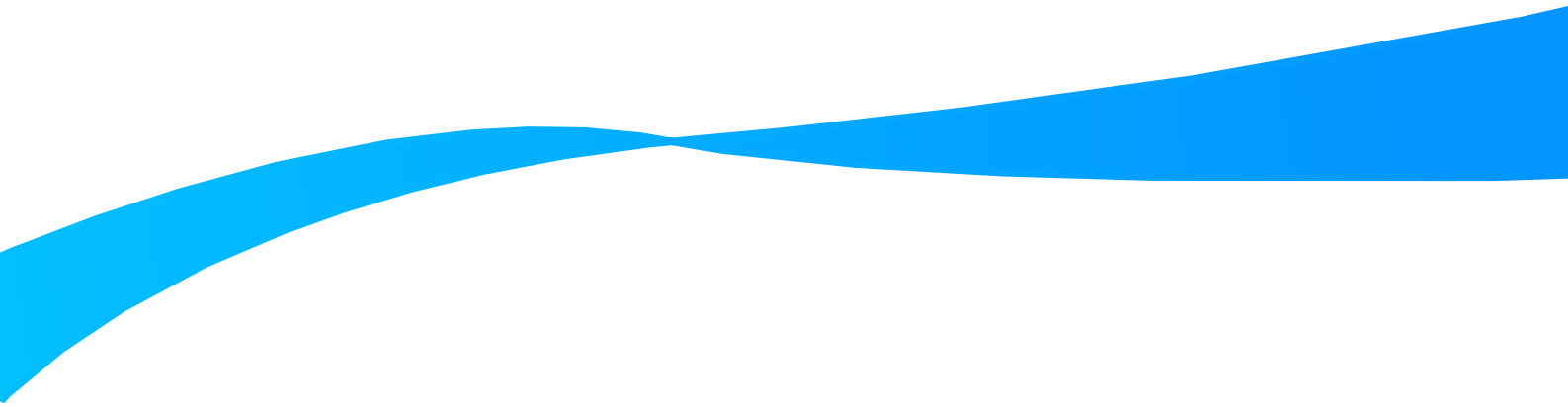

OI Coding Guidelines

riteme, Sept, 2017

目录

前言

这份简单的指南主要是关于以下两个方面：

1. OI 赛场上 C++ 语言的一些常用技巧和注意事项。
2. NOI Linux (Ubuntu) 中常用的工具。

由于大多是我个人的感受，而本人又是一个菜鸡被线踩选手，所以后面的内容大家简单看看就好。

下面有些段落前面会标上 “*” 号，表示这一部分的内容在正式赛场上不能使用，因为至少需要 C++11 的支持。

第一部分：C++ 语言

CCF 已经发了公告，不久后将会逐步取消 Pascal 和 C 语言。对于使用 Pascal 的同学，需要尽快转到 C++ 上面来。而对于使用 C 语言的同学，我觉得这是不存在的，因为 OI 中鲜有使用一些 C++ 高级特性的场景，那么 C 语言选手和 C++ 选手的主要区别就只剩是否使用 STL。

这里我将会写一点一般 OIer 容易忽视的一些 C++ 语法和技巧，希望对你们有用。

代码结构

我看过的大部分代码都是下面这个结构：

```
// 开头注释，或是一些奇怪的东西

#include <...>
#include <...>
#include <...>
// #include <bits/stdc++.h>: someone may love it. This is OK in NOI.

// 花式宏定义
// 一些简单的函数

struct XXPoint {
    // 某某 struct
};

struct XXTree {
    // 某某数据结构
};

void aaa() {
    // ...
}

inline int bbb() {
    // ...
}

// ...

int main() {
    // 开文件
    init();
    work();
    return 0;
}
```

至于到底该怎么写，这都是个人的习惯和爱好。这里只是建议每个函数不宜过长，最好不要有超过 20 行的函数，因为这种很长的函数不利于代码调试。

空格与空行

通常，很多表达式的中间我们都会打上一个空格，用来凸显运算符的位置。一个很长的表达式没有空间，都是比较难以阅读的。如下面这个表达式：

```
x1=-b+sqrt(b*b-4*a*c)/2/a;x2=-b-sqrt(b*b-4*a*c)/(2*a);// 解方程
```

普遍会写成这样：

```
x1 = -b + sqrt(b * b - 4 * a * c) / 2 / a;
x2 = -b - sqrt(b * b - 4 * a * c) / (2 * a);  // 注释与代码之间通常有 1-2 个的空格
```

相比之下，下面这个在代码中就清晰很多。当然，如果你觉得像 `4 * a * c` 这种可以写的和数学公式 $4ac$

一样紧凑，`4*a*c` 也是可以接受的。打这种空格并不需要花费什么时间，只要多试几次，自然就熟悉了。我们的目的在于希望通过这种习惯性的空格来减少自己检查错误的时间。

空行通常用于分隔两段作用有明显差异的代码块，如两个函数之间、同一个函数的不同步骤之间。这个时候写点注释也是可以的，但是上了考场，由于时间原因，大家普遍不会写注释.....

下面这段代码我是随便从 UOJ 上截的一段改的，体会一下就好。

```
sieve();

for (int i = 1; i <= h; i++)
    if (!(a[1] % pri[i])) p[++tot] = pri[i];

for (int i = 1; i <= n; i++) {
    i64 t = gcd(a[1], a[i]);

    if (t == 1) {
        printf("-1 ");
        continue;
    }

    bool flag = 1;
    for (int j = 1; j <= tot && 1LL * p[j] * p[j] <= t; j++) {
        if (!(t % p[j])){
            printf("%lld ", t / p[j]);
            flag = 0;
            break;
        }
    }

    if (flag) printf("1 ");  // t为质数
}
```

如果您是一个热衷于缩行的青少年，上面的话您可以无视.....

命名习惯

通常普遍为大家所接受的就是使用英文（或简写），而不是拼音乃至拼音简写。我见过一个比较极端的例子就是 `ksm`。乍一看不知道是个什么东西，直到看了代码才知道是 “快速幂” 的意思.....一般这种函数通常都会写做 `quick_pow` 或者是 `qpow`。当然，上面的结论是基于方便他人理解你的代码而得出的。在 OI 中，基本没人会在意你是怎么写的。所以如果你更熟悉我们的国语，使用拼音命名法也无妨。

兼顾阅读效率和编码效率，我的推荐是名称最好 2-4 个字符，超过 6 个字符是不太适合的。之所以不推荐 1 个字符，是因为容易产生命名冲突，也就意味着容易写错代码。

下面列举了一些在 OI 中常见的写法及其含义：

写法	原始单词 / 符号	注释
`aha`	Aha!	dalao HJWJBSR 最爱
`ans`	answer	
`arr`	array	
`buc`	bucket	桶排序相关
`buf`	buffer	缓存、缓冲
`ch`	child / children	通常指树上的儿子
`chk`	check	
`cls`	clear	貌似来源于一个命令
`cmp`, `comp`	compare	想必你在 `sort` 中用过

写法	原始单词 / 符号	注释
`cnt`	count	
`cur`	current	
`del`	delete	
`dist`, `dis`	distance	
`div`	divisor	因子
`div`	divide	除法
`e`, `E`	e = 2.71828...	自然对数函数的底数
`eps`, `EPS`	ε (epsilon)	通常表示精度设定（浮点数中）
`equ`	equal	
`err`	error	
`eval`	evaluate	计算
`expr`	expression	表达式
`f`, `g`, `h`, `dp`	-	一般的 DP 数组
`fa`	father	
`func`	function	函数
`g`, `G`	graph / primive root	图 / 数论中的原根
`i`, `j`, `k`	-	各种下标
`inf`, `INF`	infinity	`#define INF 0x3f3f3f3f`?
`init`	initialize	
`ins`	insert	
`interp`	interpolate	线性插值，$\vec{c} = (1-t)\vec{a} + t\vec{b}$
`inv`	inverse	逆元
`iX (X = 16, 32, 64, 128)`	signed integer	有符号整型
`jmp`	jump	
`l`, `r`, `m`	left, right, middle	常用于二分中
`lch`	left child	
`len`	length	
`ll`, `LL`	`long long` (in C++)	
`mi`, `mx`	min, max	（我个人不喜欢 `mi`）
`mid`	middle	

写法	原始单词 / 符号	注释
`mod`, `MOD`	modulo	10^9+7, 998244353, your best friends
`mul`	multiply	
`num`	number	
`nxt`	next	
`pi`, `PI`	$\pi = 3.14159...$	
`pos`	position	
`pre`	prefix	前缀
`pre`, `prev`	previous	上一个
`pri`	prime	
`ptr`	pointer	指针
`rand`, `rnd`	random	
`rch`	right child	
`ref`	reference	引用
`res`	result	
`ret`	return	
`rev`	reverse	
`S`	set	通常表示集合
`s`, `t`	source, sink (target)	源点和汇点，在网络流和最短路中常见
`seg`	segment	You must know segment trees.
`seq`	sequence	
`sqr`	square	x^2
`sqrt`	square root	$\sqrt{x}$
`stk`	stack	
`str`	string	
`suf`	suffix	后缀
`tmp`	temporary	临时的
`tot`	total	似乎 Pascal 选手喜欢用？
`u`, `v`, `w`	vertexes and weight	图论题中天天有
`upd`	update	
`uX (X = 16, 32, 64, 128)`	unsigned integer	无符号整型

写法	原始单词 / 符号	注释
val	value	

当然，每个人都会有自己的喜好，你可以参照上面来简化你的命名规范。

全局中的变量和函数

这一部分主要是提一下 inline 和 static 关键字。inline 关键字用于函数，表示提醒编译器可以对这个函数展开内联优化，简单的说就是将函数的代码直接写出来，而不是一个函数调用。这里给一个具体的例子：

```
int func(int n) {
    return n + 5;
}

int main() {
    printf("%d\n", func(10));  // 输出 15
    return 0;
}
```

那么内联展开后的结果应该等价于：

```
int main() {
    printf("%d\n", 10 + 5);  // 输出 15
    return 0;
}
```

从实际的汇编代码来看，如果不写 inline，得到的代码是这样的：

```
.LC0:  // 字符串 "%d\n" 的声明
    .string "%d\n"
    .text
    .globl main
    .type main, @function

// main 函数中
movl $10, %edi    // 填入参数 10
call func         // 执行 func
movl %eax, %esi   // 将 func 的返回结果填入 printf 的参数
movl $.LC0, %edi  // 将 "%d\n" 填入参数
movl $0, %eax     // 由于 printf 使用了变长参数列表，所以这里应该是填了一个终止标记?
call printf       // 调用 printf
```

而进行了内联优化之后，就会变成这样子：

```
movl $10, -4(%rbp)
movl -4(%rbp), %eax
addl $5, %eax  // 直接执行 10 + 5 (eax 处原是 10)
movl %eax, %esi
movl $.LC0, %edi
movl $0, %eax
call printf
```

上面的过程在不开优化的情况下是不会进行的，这里我是通过强制 GCC 执行内联才得到的结果。此外，对于递归函数，编译器是不会进行内联优化的，也就是说递归函数前面加上 inline 没有任何用处。如果一个函数很长或者很复杂，编译器也通常不会进行优化。我的推荐是如果行数不超过 7 行，就写一个 inline，否则就没有必要浪费时间。

static 可以用于函数和变量前。在函数前写个 static 纯属画蛇填足，因为 OI 中没有任何用处，但在变量前写个 static，可以使你少写一个清零。被声明为 static 的原始类型变量初始默认都是 0，而不是一个随机的值。

在函数中的静态变量相当于是一个全局变量，只是函数外部不可以访问到它。这个特性通常被拿来做计数器：

```
// 调用 5 次 cnt 将分别返回 1 2 3 4 5
int cnt() {
    static int tot;  // 初始为 0
    return ++tot;
}
```

模块化

虽然大部分题目只会涉及到一两个算法，但是 OI 中不缺乏代码量大、涉及算法和方法比较多的题目，例如需要打各种公式的“【清华集训2016】定向越野”、为了给部分分而造出的三合一“【NOI2015】小园丁与老司机”，以及 NOIP 三合一的“【WC2013】平面图”。如果代码没有良好的规划，那么在考场上写这种题目出错的概率将大大增加。

模块化通常是将没有什么关联的代码块做出适当的隔离。最需要这种方法的情况是需要使用不同的算法解决不同要求的问题的时候，通常代码会有下面的结构：

```
struct NAIVE {
    // 暴力算法

    void main() {
        // ...
    }
};

struct Solver1 {
    // 骗分算法 1

    void main() {
        // ...
    }
};

struct Solver2 {
    // 骗分算法 2

    void main() {
        // ...
    }
};

int main() {
    if (n <= 1000)
        (new NAIVE)->main();    // 使用暴力算法
    else if (something met)
        (new Solver1)->main();  // 采用算法一
    else if (something else met)
        (new Solver2)->main();  // 采用算法二
    else  // ***k the problem
        puts("orz");            // 表示对出题人崇高的敬意

    return 0;
}
```

这样做相比于所有算法都写在全局的好处在于它首先避免了算法之间的命名冲突。其次，如果一个算法将不会被使用，那么它所需要的内存也不会被申请，避免了全部开在全局而导致超出内存限制的隐患。请注意，在 NOI 的评测机中，你开了多大的数组就会计入多少的内存，而不是某些评测机，即使开了 512MB 的数组，但是如果只用了 32MB，就只会记你 32MB。

除了使用 `class`/`struct` 这种方法之外，一些人还会使用 `namespace`。虽然我个人觉得 `namespace` 在 OI 中没什么用，因为它的作用可以说只是把命名空间中的所有名字全部加了个前缀。比如 STL 中的所有东西全部被放在了一个叫做 `std` 的命名空间，所以，如果程序前面不用 `using namespace std`，那么使用 STL 中的函数就都需要加上一个 `std::`，如 `std::sort`。

class 与 struct

在 C++ 中，class 与 struct 的唯一区别在于：class 的成员默认使用 private 修饰，而 struct 中默认使用 public。换句话说，下面的这个定义：

```
class A {
 public:
    // ...
};
```

与 struct A {}; 是等价的。在 OI 中，没有必要关心成员应该被声明为 public 还是 private，所以大家普遍使用 struct。出于这个原因，下文中将不会再提及 class。

构造函数是 struct 一个很方便的特性，一般被用于给算法或数据结构提供参数，或者用于初始化。但是请注意，构造函数不是普通的函数。构造函数的调用意味着一个新的实例的创建，因此一般不会递归的调用构造函数。例如，下面这份代码：

```
struct Aha {
    int arr[1000000];

    Aha(int n) {
        if (!n)
            return;
        printf("%d\n", arr[0]);  // 防止编译器优化
        Aha(n - 1);
    }
};
```

如果执行 new Aha(1000)，将意味着会有 1000 个 Aha 会被创建，意味着 $10^6 \times 10^3 = 10^9$ 个 int 被创建，意味着你会获得 Memory Limit Exceeded 或者 Runtime Error。意思是，每一次 Aha 构造函数的递归，都是在一个新的 Aha 上进行的，而不是在同一个上。所以，像线段树这种普遍使用递归来构造的数据结构，一般会新定义一个函数来构建。

此外，构造函数在 OI 中更普遍的应用是简单数据的声明，如 Point、Vector 和 Line 之类的东西。例如在需要计算几何的代码中，通常会有：

```
struct Point {
    Point() : x(0), y(0) {}  // 默认构造函数
    Point(double _x, double _y) : x(_x), y(_y) {}
    Point(const Point &p) : x(p.x), y(p.y) {}  // 复制构造函数

    double x, y;
};
```

那么，使用 Point() 将会得到一个位于原点的点，而使用 Point(2, 3) 将会得到一个坐标在 (2, 3) 的点。

OIer 普遍不会关心内存回收的问题，除非是确实不回收会导致超内存。所以大家的代码中都鲜有析构函数的出现。一般析构函数的作用就是删除一些用 new 或者 malloc 申请出的空间，一个典型的例子就是线段树和平衡树的删除：

```
struct Node {
    Node(int l, int r)  // 构造函数
        : left(l), right(r), lch(NULL), rch(NULL) {
        dat = new int;
    }

    ~Node() {
        delete dat;
        delete lch;
        delete rch;
    }

    int left, right;
    int *dat;
```

```
    Node *lch, *rch;
};
```

假设 root 是树根，那么 delete root 将会回收整个线段树，因为 delete lch 和 delete rch 意味着左儿子和右儿子的析构函数也会被递归的调用。也许你会问当 lch 或 rch 为 NULL 的时候会怎么样，其实什么事都不会发生，因为如果 delete 一个空指针（0、NULL 或者 C++11 中的 nullptr），这个操作将会被无视。

另外一个常用的特性就是运算符重载。基本上你能想到的运算符都是可以重载的，这些都可以在 cppreference (http://en.cppreference.com/w/cpp/language/operators) 上看到。接下来只介绍几个常用的重载。

重载比较关系符，大于小于等于之类的，如果重载了小于就可以直接用 std::sort 排序而不用写 cmp：

```
#include <cmath>  // For hypot

#define EPS 1e-8

inline bool equ(double a, double b) {
    return a - EPS < b && b < a + EPS;
}

struct Point {
    double x, y;

    // 计算模长
    double len() const {
        return hypot(x, y);
    }

    bool operator<(const Point &b) const {
        return len() < b.len();  // 按模长长短排序。
    }

    // 相等和不等
    bool operator==(const Point &b) const {
        return equ(x, b.x) && equ(y, b.y);
    }

    bool operator!=(const Point &b) const {
        return !(*this == b);
    }
};
```

通常，运算符重载还可以用友元函数来写。上面的例子用友元函数来写就是这样的：

```
struct Point {
    double x, y;

    double len() const {
        return hypot(x, y);
    }

    friend bool operator<(const Point &a, const Point &b);
};

bool operator<(const Point &a, const Point &b) {
    return a.len() < b.len();
}
```

这就类似于写 cmp 这种东西了。

像向量这种东西，我们经常会需要重载加减、数乘、叉积和点积运算，这时候就可以使用运算符重载。具体的写法你们可以自己脑补。我一般把数乘和点积重载为 * 号，把叉积重载为 % 号。

此外，函数调用也可以重载（就是函数调用后的那一对括号……）。这通常用于提供给 std::map 和 std::priority_queue 之类的 cmp：

```
struct cmp {
    bool operator()(int a, int b) const {
        return a > b;
    }
};

static priority_queue<int, vector<int>, cmp> q;  // 转变为小根堆
static cmp func;  // 可以直接把 func 当函数用。
```

运算符优先级

C++ 中书写表达式的时候不可避免的就是运算符优先级的考虑，尤其是含有位运算的表达式中。了解 C++ 中的运算符优先级有利于我们在考场上节约时间和精力。一般情况下只要知道以下几个相对顺序就可以了：

1. 乘法、除法、取模 (*, /, %)
2. 加法、减法 (+, -)
3. 按位左移、右移 (<<, >>)
4. 大于、小于、不大于、不小于 (>, <, <=, >=)
5. 相等、不相等 (==, !=)
6. 按位与 (&)
7. 按位异或 (^)
8. 按位或 (|)
9. 逻辑与 (&&)
10. 逻辑或 (||)

比较容易写错的是有按位左移、右移的，要注意加法减法的优先级比按位左移右移高，因此 `1 << 3 + 7` 的结果不是 2^3+7，而是 2^{10}。此外由于比较运算符的优先级没有按位左移、右移高，所以如果比较符两边的按位平移要用括号包起来。另外，与的优先级高于或的优先级。如果考试时忘记了，那你最好还是多写几个括号以免出错。

详细的运算符优先级也可以在 cppreference (http://en.cppreference.com/w/cpp/language/operator_precedence) 上看到。

模板

这里将介绍一些简单的模板（`template`）的使用。一般是用于一些简单的函数，如 `max`：

```
template <typename T>
inline T max(const T &a, const T &b) {
    return b < a ? a : b;
}
```

我们可以直接使用 `max(1, 2)`，它将返回 `2`。实际上是编译器发现你填入的参数 `a` 和 `b` 都是 `int`，所以编译器可以推断出 `template` 中的声明 `T` 应该是 `int`。于是编译器将上面的 `max` 函数补全，然后产生了下面的代码：

```
inline int max(const int &a, const int &b) {
    return b < a ? a : b;
}
```

这是模板的本质。实际上，编译器可能有时候无法推断类型是什么，就比如下面这段快读的代码：

```
template <typename T>
inline T read() {
    T x = 0, f = 1;
    char c;
    do {
        c = getchar();
        if (c == '-')
            f = -1;
    } while (!isdigit(c));
```

```
    do {
        x = x * 10 + c - '0';
        c = getchar();
    } while (isdigit(c));

    return f * x;
}
```

此时，如果你想读入一个 int，就需要调用 read<int>()，通过尖括号内的内容告诉编译器 T 应该是什么东西。当然，一般的快读不会这么写，因为在性能要求很高的情况下返回值可能会增大常数，所以一般都是引用参数：

```
template <typename T>
inline void read(T &x) {
    x = 0;
    // 下面基本一致
}
```

这样就不用填模板参数了，直接像 read(n) 一样使用就好。

模板当然可以多放几个待定的类型，例如，我一般写的判断两个浮点数是否相同的代码是这样的：

```
template <typename T1, typename T2>
inline bool equ(const T1 a, const T2 b) {
    return fabs(a - b) < EPS;
}
```

如果 a 和 b 只声明一个 T，那么当调用 equ(1.0f, 1.0) 会出现编译错误，因为 a 是 float 而 b 是 double，T 不能同时是两个类型。

你可以为特定的类型写特化的函数。还是拿之前快读做例子，读入整型应该是那么读，但读入字符串就要换一种方式，因此我们需要这么写：

```
template <>
// read<char> 可以简写为 read，因为 T = char 可以从参数中推出
inline void read<char>(char &ch) {
    do {
        ch = getchar();
    } while (!isalpha(ch));

    char *str = &ch;
    size_t pos = 1;
    do {
        str[pos++] = getchar();
    } while (isalpha(str[pos - 1]));
    str[pos - 1] = 0;
}

// 输入 "owengetmad"
char buf[233];
read(buf[0]);  // buf = "owengetmad"
```

上面是将 T 指定为 char 时的特化，说白了就是将 T 替换掉再写一遍。之所以不特化为 char []，是因为之前读取整数的模板使用的是引用参数，而数组类型不能为引用，所以只能折中一下使用 char，对 ch 取个地址就可得到字符数组的地址。

class 和 struct 也可以使用模板，虽然这在 OI 中不常用。我所在 OI 见到的也就类似于下面这种情况：

```
template <typename T>
struct Point {
    T x, y;
};
```

如果需要特化，会写成这样：

```
template <>
struct Point<int> {
    int x, y;
};
```

* 剩下一个常用的就是变长模板参数，OI 中一般用于输出调试信息：

```
template <typename ... Args>
void DEBUG(const char *str, const Args & ... args) {
    printf("(debug) ");
    printf(str, args...);
    putchar('\n');
}
```

使用 DEBUG("BOY♂%s♂DOOR %d", "NEXT", 123) 会输出 “(debug) BOY♂NEXT♂DOOR 123”。

宏定义

宏定义的本质是文本替换，通常用于节约代码量。当需要重复打很多形式差不多的代码的时候，宏是一个非常有用的工具。例如 “【NOIP2015】斗地主“加强”版 ” 的 “爆搜 + DP” 做法，其 DP 转移几乎长得一模一样，但是数量很多，因此这里使用宏定义可以大大节约时间。

for 通常会被众多 OIer 弄成宏函数：

```
#define rep(i, l, r) for (int i = l; i <= (r); i++)
#define rrep(i, r, l) for (int i = r; i >= (l); i--)

rep(i, 1, n) rep(j, 1, m) {
    // ...
}
```

这样就不用写 for 中那重复而又枯燥无味的部分了。在维数比较多的 DP 中，写出来的代码会简洁许多。一些人也喜欢用 per 来表示倒序遍历。

* 对于 STL 容器的遍历 (vector, map 之类)，可以使用这样的语法：

```
vector<int> vec;
for (auto &v : vec) {
    // Do something with v
}
```

宏开关经常用来打开 / 关闭调试信息，如定义 NDBEUG 宏：

```
// #define NDEBUG

#ifdef NDEBUG  // 如果 NDEBUG 打开 (取消第一行的注释)
#define DEBUG(...)  // 空的调试信息输出函数
#else
#define DEBUG(msg, ...) printf("(%s #%d) " msg, __FUNCTION__, __LINE__, __VA_ARG
S__);  // 否则启用调试
#endif
```

上面的 “...” 跟之前模板中的变长参数类似，使用 __VA_ARGS__ 这个特殊的宏可以将 “...” 所指代的参数放入。__FUNCTION__ 和 __LINE__ 是另外两个特殊的宏，它们可以记录下 DEBUG 宏函数被调用的地方所处的函数名称和行号。也可以使用 __PRETTY_FUNCTION__，它会记录下所处的函数的详细声明，可以用于区分重载的函数 。另外，assert 的打开和关闭也是受 NDEBUG 影响。assert 意思是断言，即做一个假设，如果假设不成立则终止程序 （调用 abort），它包含在 cassert 头文件中。一般的用法如下：

```
#include <cassert>

int a = 1 + 1;
assert(a == 2);  // 想必 a = 2
assert(a == 3);  // a 显然不等于 3，所以这里会导致程序终止
```

善用 assert 是一个良好的习惯。在比较复杂的数据结构中，一些地方可以添加几个断言，比如在访问一个指针前 assert 一下看指针是否为空。从而在调试的时候，如果出现异常，可以很快的发现错误的位置。最后提交文件的时候在开头声明一下 NDEBUG，就可以关掉所有的调试信息和断言。

作为一个简单的文本替换，宏的弊端在于容易使人犯错。首先就是运算符优先级的问题。一个典型的例子就是开一个大小为 $2n$ 的数组。OI 中由于不是很关心浪费的一点点空间，所以有人可能会写出这样的代码：

```
#define NMAX 100000 + 5
int arr[NMAX * 2];  // arr 的大小为 100010 而不是所期望的 200010
```

这种错误是会让人非常苦恼的，因为你的程序小数据不会有事，而大数据会不停的 Runtime Error，可你却一直看不出有什么问题。还有就是之前的 rep 的定义中，我写的是 `i <= (r)` 而不是 `i <= r`，原因非常简单，如果你要使用 `rep(i, 1, a ^ b)` 后者可能会导致你无尽的循环（因为是 `i <= a ^ b`，小于等于的优先级高于异或，所以它的值是 `(i <= a) ^ b` ）。这种例子相当之多，这里就不再列举。因此，但凡使用宏，就一定要注意多打几个括号。

另外一个经典的错误，来自于一个简单的 max 宏函数：

```
#define max(a, b) ((a) > (b) ? a : b)

int query(Node *x, int l, int r) {
    if (l <= x->left && x->right <= r)
        return x->value;

    int mid = (x->left + x->right) >> 1, ans = INT_MIN;
    if (l <= mid)
        ans = max(ans, query(x->lch, l, r));
    if (r > mid)
        ans = max(ans, query(x->rch, l, r));

    return ans;
}
```

乍一看好像没什么问题，但实际上你的线段树早已背叛了你，它将给予你无尽的 Time Limit Exceeded。原因在于，上述代码中两个取 max 的地方，如果儿子给出的答案更大，那么那个 query 将会被调用第二次，导致线段树时间复杂度的退化。因此在这里我还是推荐使用内联函数而不是宏来实现这些简单的函数。

常用函数和 STL

虽然像 max、min 之类的函数在 algorithm 头文件已有定义，但是一些特别的情况下可以使用更简单的函数来减小常数。例如用于更新最值：

```
template <typename T>
inline void chkmax(T &a, const T &b) {
    if (b > a) a = b;
}

template <typename T>
inline void chkmin(T &a, const T &b) {
    if (b < a) a = b;
}
```

相比于 `a = max(a, b)` 而言，可以减少不必要的赋值，从而减小了常数。另外一个同类型的就是取模加法：

```
template <typename T>
inline void add(T &a, const T &b) {
    a += b;

    if (a >= MOD) a -= MOD;
    if (a < 0) a += MOD;
}
```

因为只是加减法，所以使用 `(a + b) % MOD` 实际上会浪费一次取模操作。顺带一提，一般的计算机上加减

乘的速度差不多，取模和除法速度与它们相比慢很多，所以经常会尝试减少除法和取模的次数来减小常数。在浮点数中，减法和除法造成的精度误差比较大。

algorithm 头文件中有很多 OI 常用的函数，下面将列举我经常用的几个。

使用 reverse 可以将数组翻转：

```
int a[] = {1, 2, 3, 4, 5};
reverse(a, a + 5);  // a = {5 4, 3, 2, 1}
```

众所周知 sort 主要使用的是快速排序算法，同时也有一个 stable_sort，使用的是归并排序。绝大部分情况下大家都会使用 sort，少数交互题可能会卡快速排序的比较次数，这时才会使用 stable_sort 来避免。

unique 的作用是将多个连续相同的元素变为一个，通常和 sort 一起使用来去重：

```
int a[] = {1, 3, 2, 1, 2, 1, 4};
sort(a, a + 7);
int cnt = unique(a, a + 7) - a;
// a = {1, 2, 3, 4, ...}, cnt = 4
```

lower_bound 和 upper_bound 则是常用的二分函数。它们只能用于有序的序列上。lower_bound 会帮你找出某个元素第一次出现的位置，而 upper_bound 会帮你找出某个元素最后一次出现的位置的下一个位置，如：

```
int a[] = {1, 2, 2, 2, 2, 3, 7};
//            ^           ^
//            lower       upper
lower = lower_bound(a, a + 7, 2);
upper = upper_bound(a, a + 7, 2);
```

inplace_merge 是用于合并两个有序序列的，用于实现归并排序。在一些情况下可能会用到。

```
int a[] = {1, 3, 5, 2, 4, 6};
inplace_merge(a, a + 3, a + 6);  // a = {1, 2, 3, 4, 5, 6}
```

STL 中的容器也常常被使用，用于减少代码量。一般常用的有这么一些：

- vector 动态数组，这个其实看个人喜好，它可以支持下标访问，并且可以快速在末尾增加、删除元素。
- deque 双端队列，可以下标访问，但常数比数组略高，可以在两头快速插入、删除。
- queue 队列，接口比较简单，用 front 查看队首，pop 弹出队首，push 加入队尾。
- stack 栈，类似于 queue，只不过是用 top 查看栈顶。
- set 和 map，这是 STL 良心为大家提供的红黑树，可以快速查找和使用与 algorithm 中类似的 lower_bound 以及 upper_bound。
- priority_queue 优先队列，也就是大根堆，接口和 stack 差不多。
- bitset 卡常神器，用于存储二进制序列，可以进行位运算，时空复杂度基本都是 $O(n/w)$。
- string 字符串，其实就是 vector<char>，一般不怎么用。
- * unordered_set 和 unordered_map，功能与 set 和 map 对应，但是是哈希表实现的，复杂度更优秀，但不支持二分查找。可以在头文件 tr1/unordered_set 和 tr1/unordered_map 中找到。

这些东西主要还是推荐大家自己去 cppreference 上去查然后试着用一下，相信你们的英文水平足够了。

第二部分：Linux 下编程环境

编辑器

写程序第一步是什么？当然是写代码。在写代码之前，你需要确定你在将来在考场上要使用什么编辑器。NOI Linux 目前提供的编辑器有这些：

- gedit
- VIM
- Emacs
- GUIDE、Anjuta、Lazarus 之流

在这里我也不做什么推荐，大家找自己喜欢的就好。我只知道正式考场上普通玩家都是使用与 "UOJ 自定义测试" 差不多的 gedit，中高端玩家普遍选择 Emacs 或 VIM。

编译器

NOI Linux 下只提供了 `g++` 来给我们编译程序。如果你的源代码叫做 `a.cpp`，那么使用：

```
g++ a.cpp
```

将会在当前目录下产生一个 `a.out` 的可执行文件，使用 `./a.out` 就可以运行它。如果你觉得 `a.out` 这个名字太 naïve，可以使用：

```
g++ a.cpp -o exec
```

将会得到 `exec` 这个可执行文件。对于交互式的题目，一般会提供一个叫做 `grader.cpp` 的文件，此时你按照要求写好程序后，应该这样编译：

```
g++ grader.cpp a.cpp
```

如果需要打开优化，可以使用：

```
g++ a.cpp -O2  // 或者 -O3
```

一套题目的第一页通常会有注明表示某些题目是否开优化。这是比较重要的信息，通常会决定你是否需要花时间来卡常数。

一般为了充分发挥编译器的作用，我会打开编译器的绝大部分警告：

```
g++ a.cpp -Wall -Wextra
```

打开之后，一些常见的却不是语法错误的小问题都会被编译器报告。如 `if (a = b)` 这种相等符号少写一个等号的错误，编译器都会提示你。

调试器

`gdb` 是最常用于调试程序的工具。尽管多数情况下输出调试可以解决问题，但是 `gdb` 可以帮你在某些情况下节约时间。`gdb` 的主要优势在于可以灵活的跟踪程序运行的状况，方便我们找出 BUG 的根源。

如果一个程序想要使用 `gdb` 调试，那么在编译的时候需要加上 `-g` 选项：

```
g++ a.cpp -g
```

然后执行 `gdb a.out` 即可进入调试。一般你会看到下面这些东西：

```
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
```

```
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
(gdb)
```

其实只有最后一句 "Reading symbols from a.out...done." 有用，它告诉我们 a.out 被成功读取。如果你不想看到上面一大堆话，那么在命令末尾加个 -q 即可。

现在来举个例子。就以下面的程序开刀，来体会一下使用 gdb 的基本流程：

```
#include <cstdio>

int fac(int n) {
    return fac(n - 1) * n;
}

int main() {
    int n;
    scanf("%d", &n);
    n = fac(n);

    for (unsigned i = n; i >= 0; i--) {
        puts("Aha!");
    }

    return 0;
}
```

编译后进入 gdb，输入 l 并回车，可以看到程序的前面几行：

```
(gdb) l
1   #include <cstdio>
2
3   int fac(int n) {
4       return fac(n - 1) * n;
5   }
6
7   int main() {
8       int n;
9       scanf("%d", &n);
10      n = fac(n);
```

这时按下回车，可以看到紧接着的后面几行：

```
(gdb)
11
12      for (unsigned i = n; i >= 0; i--) {
13          puts("Aha!");
14      }
15
16      return 0;
17  }
```

这不是因为回车就是看代码的功能，而是在 gdb 中什么也不输入就直接回车表示执行上一个你使用的命令。所以这里实际上是又执行了一遍 l 命令。你也可以使用上下方向键来翻阅你之前使用的命令。使用 l [行号] 将会把你所指定的行号附近的代码展示给你。而 l [函数名] 则会把对应函数展示。

当然我们不是拿 gdb 来看代码的……如果你现在使用 r 命令，gdb 将会正常启动这个程序。你会发现这在和

gdb 外面运行程序没什么区别，输入了一个数字之后程序就会段错误。这是因为我们还没有设置断点，gdb 不会无缘无故使程序暂停。

使用 b [函数名 / 行号] 可以在特定的函数入口或者某一行设置断点，当程序运行到这里的时候，就会暂停下来：

```
(gdb) b main
Breakpoint 1 at 0x40058c: file a.cpp, line 7.
(gdb) r
Starting program: /home/owen/aknoi/a.out

Breakpoint 1, main () at a.cpp:7
7   int main() {
```

使用 n 可以执行当前这一行代码：

```
(gdb) n
9       scanf("%d", &n);
```

这里再回车一下就会执行 scanf 了，也就会要你输入一个数字了：

```
(gdb) n
1
10      n = fac(n);
```

注意，到了第 10 如果继续使用 n，将不会进入 fac 函数。如果想进入 fac 函数一探究竟，就需要使用 s：

```
(gdb) s
fac (n=1) at a.cpp:4
4       return fac(n - 1) * n;
```

使用 p 可以计算一些表达式，如：

```
(gdb) p n * 123
$1 = 123
```

p 是 gdb 的方便所在，它可以查看程序中的变量从而知道程序当前的状况。当然，总是手动使用 p 非常麻烦，我们可以使用 disp 来将某个表达式挂在旁边随时查看：

```
(gdb) disp n
1: n = 1
(gdb) s
fac (n=0) at a.cpp:4
4       return fac(n - 1) * n;
1: n = 0
```

没错，你会发现 n 的值会在每一步指令结束后展示给你。注意到左边的那个标号 1，你可以使用 undisp 1 来取消这个显示。

当然你不会一直使用 n 和 s 让程序跑下去，有时候是希望让程序先自由的跑一跑。这里我们先给第 4 行加个断点，并使用 c 命令继续运行程序：

```
(gdb) b 4
Breakpoint 2 at 0x400571: file a.cpp, line 4.
(gdb) c
Continuing.

Breakpoint 2, fac (n=-1) at a.cpp:4
4       return fac(n - 1) * n;
```

由于在继续执行的过程中遇上了断点 2，所以程序又一次暂停。现在可以来看看函数的调用栈了：

```
(gdb) bt
```

```
#0  fac (n=-1) at a.cpp:4
#1  0x000000000040057e in fac (n=0) at a.cpp:4
#2  0x000000000040057e in fac (n=1) at a.cpp:4
#3  0x00000000004005bb in main () at a.cpp:10
```

可以看到 fac 调用了三层。我们可以使用 d 命令来删除一个断点，并且尝试继续执行：

```
(gdb) d 2
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x000000000040056e in fac (n=<error reading variable: Cannot access memory at ad
dress 0x7fffff7feffc>) at a.cpp:3
3   int fac(int n) {
```

没有了断点的阻挠，我们的程序直接段错误。输入 bt 就可以看到 fac 递归了相当多层，导致超出栈空间。找到了错误原因，就可使用 q 命令退出 gdb 了：

```
(gdb) q
A debugging session is active.

    Inferior 1 [process 23333] will be killed.

Quit anyway? (y or n) y
```

由于 fac 函数缺少边界条件导致爆栈，所以修改之后的程序是这样的：

```
#include <cstdio>

int fac(int n) {
    if (n == 0)
        return 1;
    return fac(n - 1) * n;
}

int main() {
    int n;
    scanf("%d", &n);
    n = fac(n);

    for (unsigned i = n; i >= 0; i--) {
        puts("Aha!");
    }

    return 0;
}
```

现在编译并运行程序将会无限输出 “Aha!”。再次使用 gdb 来调试，这次直接开始运行程序，并且在一堆 “Aha!” 面前沉着地按下“Ctrl + C”：

```
Aha!
Aha!
Aha!
Aha!
Aha!
^C
Program received signal SIGINT, Interrupt.
0x00007ffff7856290 in __write_nocancel () at ../sysdeps/unix/syscall-template.S:
84
84  ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb)
```

然后这个程序就暂停了。gdb 告诉你现在在一个奇怪的文件里面，其实是因为在执行 puts 时暂停了。现在

我们想回到 a.cpp 函数的第 15 行，因此可以设一个断点：

```
(gdb) b a.cpp:15
Breakpoint 2 at 0x4005d1: file a.cpp, line 15.
(gdb) c
Continuing.


Breakpoint 2, main () at a.cpp:15
15          puts("Aha!");
(gdb) p i
$1 = 4294925190
```

打印出 i 的值，发现变得非常大。这是因为 i 是无符号整型，当 i = 0 时再减一，就会变成 UINT_MAX，因而导致死循环。至此，这个程序的异常就全部查清了。以上也差不多就是 OI 中常用的命令，如果想要 gdb 为你发挥更大的作用，就需要平常多练习使用，然后才能在正式考场上熟练运用这些工具。

性能剖析工具

NOI Linux 上唯一提供的性能剖析工具就是 gprof 了。它的主要目的就是生成一份程序的运行报告，主要包括程序中每个函数的相关数据。使用方法并不复杂，大致流程如下：

首先，编译程序的时候带上 -pg 选项：

```
g++ a.cpp -pg
```

然后运行一下这个程序，OI 中通常会拿个大数据来跑一下。注意，带上这个选项后程序运行速度可能变慢了很多，这是因为此时程序运行中会收集一些信息，变慢是正常的。程序运行完毕后，使用 gprof 生成报告：

```
gprof a.out > result.out
```

result.out 就是报告了，使用任意编辑器可以打开。下面放一个我最近写的程序生成的报告的前几行：

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 17.29      0.52     0.52 99401724     0.00     0.00  pushdown(int)
 10.97      0.85     0.33 25936095     0.00     0.00  rotate(int, int)
  6.32      1.36     0.19  6808190     0.00     0.00  upans(int, int)
  6.15      1.54     0.19 38677554     0.00     0.00  update(int)
  5.32      1.70     0.16 36189660     0.00     0.00  clear(int)
  5.15      1.86     0.16 40849140     0.00     0.00  splay(int)
  4.99      2.16     0.15       43     3.49    60.87  dfs(int, int)
  4.82      2.30     0.15 13616380     0.00     0.00  query(int, long long)
  4.82      2.45     0.15 83375581     0.00     0.00  rminb(int)
  3.66      2.56     0.11                             main
  2.66      2.64     0.08 83375581     0.00     0.00  rmina(int)
  2.33      2.71     0.07        1    70.06   130.11  initialize()
```

上面可以看到每个被调查到的函数所占用的时间百分比，秒数和调用次数。排名越靠前的表明占用时间越多。从上面可以看出，pushdown 和 rotate 占用了绝大部分时间，因此如果想卡常数，首先应考虑它们，这样效果才最明显。有时候这份报告里面可能会出现一些名字很长很奇怪的函数，不要觉得是出了问题，因为它们可能是 STL 里面的函数，所以名字才会很奇怪。

gprof 一般是用在你考试时间比较充裕的情况下，给大数据结构和一些数论筛题来分析程序性能瓶颈用的。如果你想卡常数了，可以利用这份报告做一个参考。

后记

至此，这篇废话连篇的简单的指南就结束了~~~
（我自己也没想到写了这么长.....）
最后祝大家 OI 顺利，实现自己心中的理想！

23-24 Sept 2017, by riteme