



西安电子科技大学
XIDIAN UNIVERSITY

程序设计竞赛实训基地
Programming Contest Training Base



程序调试

Debugging

席若尧

西安电子科技大学空间科学与技术学院

2020 年 6 月 16 日



- ▶ 本课程相关资料课后会下发/公开。
- ▶ 由于主讲人比较菜，如果讲错了请及时指出。



- ▶ 空间院 2019 级博士 (硕士毕不了业只能继续苟着的屑)
- ▶ 曾获 2015 年省赛冠军 (陕西省)
- ▶ 邀请赛 1 铜 2 银, 区域赛 3 铜 3 银 (没有金牌的退役狗)
- ▶ Codeforces: [loujunjie](#) (走狗屎运上了分就不打 Div. 1 了)



西安电子科技大学
XIDIAN UNIVERSITY

程序设计竞赛实训基地
Programming Contest Training Base



第 I 部分

引言





在比赛和训练中，往往出现以下现象：

70777	bytedance_Inever	A	时间超限	7612	3000	C++	1060 B	2017-04-14 14:43:25
70771	bytedance_Inever	A	时间超限	7876	3000	C++	1060 B	2017-04-14 14:40:15
70708	bytedance_Inever	A	运行错误	7848	111	C++	1420 B	2017-04-14 14:18:45
70706	bytedance_Inever	A	时间超限	7456	3000	C++	1416 B	2017-04-14 14:18:15
70693	bytedance_Inever	A	时间超限	7456	3000	C++	1259 B	2017-04-14 14:13:27
70680	bytedance_Inever	A	时间超限	11276	3000	C++	1075 B	2017-04-14 14:07:33
70639	bytedance_Inever	A	时间超限	36780	3000	C++	856 B	2017-04-14 13:54:20
70617	bytedance_Inever	A	内存超限	131140	238	C++	808 B	2017-04-14 13:46:03
70587	bytedance_Inever	A	时间超限	25932	3000	C++	773 B	2017-04-14 13:38:18
70578	bytedance_Inever	A	时间超限	5896	3000	C++	786 B	2017-04-14 13:34:18
70570	bytedance_Inever	A	时间超限	5896	3000	C++	786 B	2017-04-14 13:32:57
69581	bytedance_Inever	A	时间超限	7876	3000	C++	833 B	2017-04-13 17:26:52
69573	bytedance_Inever	A	运行错误	4840	79	C++	810 B	2017-04-13 17:25:43
69559	bytedance_Inever	A	时间超限	8008	3000	C++	790 B	2017-04-13 17:23:43



在比赛和训练中，往往出现以下现象：

Dec/31/2014 09:38 Wrong answer on test 7 [main tests] → [9330872](#)
Dec/31/2014 09:42 Wrong answer on test 7 [main tests] → [9330898](#)
Dec/31/2014 09:48 Wrong answer on test 1 [main tests] → [9330944](#)
Dec/31/2014 09:49 Wrong answer on test 7 [main tests] → [9330949](#)
Dec/31/2014 09:50 Wrong answer on test 7 [main tests] → [9330955](#)
Dec/31/2014 09:51 Wrong answer on test 7 [main tests] → [9330963](#)
Dec/31/2014 09:54 Wrong answer on test 7 [main tests] → [9330996](#)
Dec/31/2014 09:56 Wrong answer on test 7 [main tests] → [9331015](#)
Dec/31/2014 09:57 Wrong answer on test 7 [main tests] → [9331020](#)
Dec/31/2014 10:02 Wrong answer on test 7 [main tests] → [9331058](#)
Dec/31/2014 10:12 Wrong answer on test 7 [main tests] → [9331135](#)
Dec/31/2014 10:19 Wrong answer on test 7 [main tests] → [9331172](#)
Dec/31/2014 10:35 Wrong answer on test 7 [main tests] → [9331310](#)
Dec/31/2014 18:36 Wrong answer on test 7 [main tests] → [9334018](#)
Jan/01/2015 12:54 Accepted [main tests] → [9337856](#)



wrong answer



- ▶ 写程序
 - ▶ 软件开发中一半以上时间用于测试和调试
- ▶ 做电路
- ▶ 写论文
- ▶ 物理学 (?)
- ▶ 做课件 (例如本课件!)



根据中国航天科技集团公司 Q/QJA 10-2002 标准，质量问题技术归零要做到：

- ▶ 定位准确
- ▶ 机理清楚
- ▶ 问题复现
- ▶ 措施有效
- ▶ 举一反三

除了技术归零以外还有管理归零：过程清楚、责任明确、措施落实、严肃处理、完善规章



西安电子科技大学
XIDIAN UNIVERSITY

程序设计竞赛实训基地
Programming Contest Training Base

Bug 分类

- ▶ 语法错误?
- ▶ 语义错误?



- ▶ 语法错误 -> CE

例如: `(1+x;`

- ▶ 语义错误

- ▶ 违反约束条件 -> CE

例如: `int a; char a;`

- ▶ 违反其他语义规则 -> **未定义的**

例如: `int x[2]; x + 3;`

- ▶ “纯粹”的逻辑错误 -> WA, TLE, MLE, OLE

例如: 算法假了



某些人总是无视语言标准尝试使用各种未定义行为，然后到处说“这程序在我的电脑上能工作，为什么交上去就错？”对此，Roger Miller 讽刺道：

- ▶ 有人曾经告诉我，在打篮球的时候，你不能带球走。我找了个篮球试了一下，发现走得很好。那人显然根本不懂篮球。



某些人总是无视语言标准尝试使用各种未定义行为，然后到处说“这程序在我的电脑上能工作，为什么交上去就错？”对此，Roger Miller 讽刺道：

- ▶ 有人曾经告诉我，在打篮球的时候，你不能带球走。我找了个篮球试了一下，发现走得很好。那人显然根本不懂篮球。





西安电子科技大学
XIDIAN UNIVERSITY

程序设计竞赛实训基地
Programming Contest Training Base



第 II 部分

评测系统返回的错误信息



- ▶ 指你的程序无法编译成可执行文件
- ▶ 一般 OJ 都提供了查看编译器输出消息的方法，照着改即可
- ▶ 正式比赛要求选手机器和评测机完全一致，所以几乎不会出现
- ▶ 正式比赛 CE 不算罚时



西安电子科技大学
XIDIAN UNIVERSITY

程序设计竞赛实训基地
Programming Contest Training Base

时间超限
TIMELIMIT

► 你的程序跑得太慢了



- ▶ 你的程序没有以返回值 0 正常退出
- ▶ 一般是因为出现了未定义行为
- ▶ 也可能是不小心返回了一个非 0 值，比如压行压成了 `exit(printf("%d\n", x));`
- ▶ 现场赛使用 DOMJudge，内存超限也报告为运行错误



- ▶ 你输出了太多东西，超过了题目的限制
- ▶ 可能是陷入了带输出的死循环
- ▶ 也可能是忘了删除调试输出:)
- ▶ 对于某些题目只是 WA 的一种表现形式
 - ▶ 例如：如果有解输出一个 $[0, 9]$ 中的整数，如果无解输出 "a very very long error message"，那么如果把很多有解的情况判断成了无解就可能输出超限



- ▶ 你的程序输出和标准答案不一致，或者被 SPJ 判断为错误
- ▶ 不同评测系统对于空白字符 (特别是行末空格和文末回车) 的处理不一致，如果空白字符有区别可能返回 AC、WA、PE 等不同结果。
- ▶ DOMJudge 没有 PE，无 SPJ 的情况下会忽略行末空格和文末回车，但对于空白字符的其他差异会直接返回 WA。建议写输出时遵循一般的规范，每行都以 "\n" 结束，行末不要有多余的空格
- ▶ 如果完全没有输出，一些评测系统 (包括 DOMJudge) 会返回 "NO-OUTPUT"



西安电子科技大学
XIDIAN UNIVERSITY

程序设计竞赛实训基地
Programming Contest Training Base



第 III 部分

程序设计竞赛中常见的编程错误





- ▶ 函数调用的过程中会把一些数据（返回地址，局部变量，部分参数）放入系统的栈空间中，调用结束后再弹出。
- ▶ 为了更灵活地为堆和共享库分配内存，Linux 默认将栈空间限制在 8MB。
- ▶ 如果栈的大小越过 8MB，内核会发送 SIGSEGV 信号杀死进程。
- ▶ Windows 默认栈空间限制为 1MB。



- ▶ Runtime Error
- ▶ 段错误
- ▶ segmentation fault (core dumped)



- ▶ 绝对不要写死递归
- ▶ 一些评测系统会调整系统栈空间限制，因此可以大胆使用栈
 - ▶ DOMJudge 的当前版本不限制栈空间，建议区域赛热身赛进行测试
 - ▶ Codeforces 调整到了 256MB
 - ▶ 然而大多数在线评测系统都没有调整
- ▶ 可以将本机调成不限制栈空间：`ulimit -s unlimited`
 - ▶ 只对当前终端开出来的进程有效
 - ▶ 这会导致死递归难以排查，甚至卡死系统，可以选择折中地把栈调大一点：
`ulimit -s 131072`（单位 KB）
- ▶ 对于递归，本机测试开与评测时相同的优化（例如 `-O2`），防止误判（将其他错误当成栈溢出，或者相反）
- ▶ 不要将巨大的数组或结构体声明为自动局部变量或按值传递的参数
 - ▶ 可以开全局或者静态变量，通过指针/引用传递
- ▶ 使用非递归写法，或者显式开栈模拟递归



暗黑魔法，使用时自负风险。

```
1  #include <cstdlib>
2  char _s_t_a_c_k_[128 << 20];
3  int main()
4  {
5      __asm volatile (
6  #ifdef __x86_64__
7          "movq %0, %%rsp"
8  #else
9          "movl %0, %%esp"
10 #endif
11          :: "r" (_s_t_a_c_k_+(128<<20)):
12          );
13
14      exit(0);
15  }
```




以下属于未定义行为：

- ▶ 带符号整数算术运算溢出
- ▶ 移位位数超过整数位数
- ▶ 使用 scanf 读取数字时，输入超过格式化字符串指定的表示范围
- ▶ 浮点数转化为整数时，其值超过了整数类型能表示的范围

以下行为可能导致出人意料的结果：

- ▶ 无符号整数算术运算溢出（丢弃高位）
- ▶ 使用 cin 读取数字时，输入超过变量的表示范围（读入错误的值，并设定 failbit）
- ▶ 整数类型互相转化时高位被丢弃
- ▶ 浮点数转化为整数时小数部分被截断



以下属于未定义行为：

- ▶ 带符号整数算术运算溢出
- ▶ 移位位数超过整数位数
- ▶ 使用 scanf 读取数字时，输入超过格式化字符串指定的表示范围
- ▶ 浮点数转化为整数时，其值超过了整数类型能表示的范围
 - ▶ 欧洲的 Ariane 5 型运载火箭首飞即因为代码中 64 位浮点值向 16 位整数转换时发生此类错误而坠毁

以下行为可能导致出人意料的结果：

- ▶ 无符号整数算术运算溢出（丢弃高位）
- ▶ 使用 cin 读取数字时，输入超过变量的表示范围（读入错误的值，并设定 failbit）
- ▶ 整数类型互相转化时高位被丢弃
- ▶ 浮点数转化为整数时小数部分被截断



- ▶ 大部分会 WA，可能会 RE 甚至 TLE。
- ▶ 在测试时你的程序可能莫名其妙输出负数。

```
1 | int a, b, ans = 0;  
2 | scanf("%d%d", &a, &b);  
3 | for (int i = a; i <= b; i++)  
4 |     ans += foo(i);  
5 | printf("%d\n", ans);  
6 | return 0;
```



- ▶ 开始编码前预先考虑好输入、中间结果、最终结果的可能范围
- ▶ 该取模的取
- ▶ 该开 64 位和 128 位整数的开
- ▶ 该转型的转：111 * a * b
- ▶ 该换语言的换
- ▶ 该写高精度的写
- ▶ 打开编译器相关警告选项
- ▶ 编造数据测试是否有溢出
- ▶ 如果测试过程中发现溢出，打开运行时检查工具
- ▶ 如果确实需要溢出，使用无符号整数
- ▶ 反对盲目蛮干



- ▶ 未初始化的指针
 - ▶ 和一般未初始化值一样，使用就是未定义行为
- ▶ 空指针
 - ▶ 只能用作哨兵值 (sentinel)
- ▶ 指向数组最后一个元素“之后一个元素”的指针
 - ▶ 可以构造出来，例如 `sort(a, a+n);`
 - ▶ 可以用于偏移运算，例如 `int *p = a+n; a[-2];`
 - ▶ 不能解引用，否则引发未定义行为
- ▶ 越出数组界限的指针
 - ▶ 对不指向数组中元素的指针进行偏移运算是未定义的
 - ▶ 对指向数组中元素的指针进行偏移运算，越出数组范围（结果不指向同一数组中的元素，或该数组最后一个元素“之后的一个元素”），行为是未定义的



- ▶ RE、WA、TLE、MLE 都有可能
- ▶ 可能严重干扰调试器



- ▶ 数组开到足够大
- ▶ 对指针和数组下标进行必要检查
 - ▶ `for (; j < n && a[j] < b[i]; j++;) foo(j);`
- ▶ 不要乱用指针
- ▶ 如果本地测试出现问题，怀疑无效指针，可以打开相关运行时检查



- ▶ 和指针一样，迭代器也不能越界
- ▶ 此外，在一些（可能意想不到的）情况下，迭代器会失效变成非法的



我们希望删掉一个 `std::set` 里面所有的偶数：

```
1 #include <set>
2 #include <cstdio>
3 using namespace std;
4
5 int main()
6 {
7     set<int> S = {4, -1, 5, 8, -2, -5};
8     for (int x: S)
9         if (x % 2 == 0)
10             S.erase(x);
11     for (int x: S)
12         printf("%d\n", x);
13 }
```



```
shell$ g++ set_invalidate.cc  
shell$ ./a.out  
Segmentation fault (core dumped)
```



根据语言标准，循环

```
for (int x: S)
    if (x % 2 == 0)
        S.erase(x);
```

等价于

```
for (auto __begin = S.begin(),
      __end = S.end();
      __begin != __end;
      ++__begin) {
    int x = *__begin;
    if (x % 2 == 0)
        S.erase(x);
}
```



根据语言标准，循环

```
for (int x: S)
    if (x % 2 == 0)
        S.erase(x);
```

等价于

```
for (auto __begin = S.begin(),
      __end = S.end();
      __begin != __end;
      ++__begin) {
    int x = *__begin;
    if (x % 2 == 0)
        S.erase(x);
}
```

- ▶ 删掉一个偶数以后，指向它的迭代器 `__begin` 就失效了，然后再使用这个迭代器就会发生未定义行为



先自增，再删除：

```
for (auto it = S.begin(); it != S.end(); )  
    if (*it % 2 == 0)  
        S.erase(it++);  
    else  
        it++;
```



你可能认为不删除就没问题了，然而……

```
1 #include <vector>
2 #include <cstdio>
3 using namespace std;
4
5 int main()
6 {
7     vector<int> v = {1,2,3};
8     for (int i: v)
9         if (i > 0)
10             v.push_back(-i);
11     for (int i: v)
12         printf("%d\n", i);
13 }
```



```
shell$ g++ vec_invalidate.cc
shell$ ./a.out
1
2
3
-1
-7802896
```

► 什么鬼？



```
shell$ g++ vec_invalidate.cc
shell$ ./a.out
1
2
3
-1
-7802896
```

- ▶ 什么鬼？
- ▶ 这是因为 vector 在插入元素时，可能需要重新分配一段更大的内存，并搬移原有元素，从而导致之前的迭代器失效



```
shell$ g++ vec_invalidate.cc
shell$ ./a.out
1
2
3
-1
-7802896
```

- ▶ 什么鬼？
- ▶ 这是因为 vector 在插入元素时，可能需要重新分配一段更大的内存，并搬移原有元素，从而导致之前的迭代器失效
- ▶ 改用下标就行了



- ▶ 不要滥用迭代器
- ▶ 使用 range-based for 循环时，最好不要对被迭代的容器进行插入或删除操作
- ▶ 在 set、map、multiset、multimap、list 等进行删除操作时，可以使用 `c.erase(it++)` 的写法
 - ▶ 对于 C++11 以上，还支持 `it = c.erase(it)` 的写法
- ▶ 如果怀疑使用了无效迭代器，可以打开 C++ 标准库的运行时检查



- ▶ 算法假了
- ▶ 算法是真的，但某个细节没考虑，导致高次时间复杂度
- ▶ 常数太大



- ▶ 算法假了
 - ▶ 典型代表：暴力字符串匹配、keduoli 树
- ▶ 算法是真的，但某个细节没考虑，导致高次时间复杂度
 - ▶ 典型代表：for(int i = 0; i < strlen(s); i++)
 - ▶ memset
- ▶ 常数太大
 - ▶ 典型代表：endl、valarray



- ▶ 做好时间复杂度分析
 - ▶ 几乎一定要分析最坏情况
 - ▶ “期望时间复杂度”几乎没有用
- ▶ 使用工具寻找可能存在的高次复杂度和大常数
- ▶ 卡常数



众所周知，浮点数的精度是有限的。

- ▶ float: 23 位
- ▶ double: 52 位
- ▶ long double: 可能是 52 位或 64 位



- ▶ float 这种东西就别用了吧……
- ▶ 如果要输出的东西本身就是一个浮点值（如长度、面积），那么可以放心使用 double，但要注意控制精度，防止出现数值稳定性问题
 - ▶ 一般会有 SPJ
 - ▶ 减少中间步骤
 - ▶ 防止出现奇异值
 - ▶ 如果没有 SPJ 的话需要调一调 eps ……
 - ▶ 比如 1.285 四舍五入保留小数点后两位，不加 eps 输出会暴毙
- ▶ 如果有比较逻辑，要输出 YES/NO 或方案数，尽量避免浮点数
 - ▶ $x \geq y$?
 - ▶ 32 位机器上“薛定谔的精度”
 - ▶ 避免除法
 - ▶ 如果一定要用除法，写分数类



- ▶ floating point exception (core dumped)
- ▶ 和浮点数并没有什么关系……
- ▶ 意味着出现了除以 0 或者模 0
- ▶ 直接用调试器看在哪里除了 0 就行了



- ▶ 提供给 sort 或者 set 的比较函数是假的？
- ▶ 在没排序的数组上执行 lower_bound 等二分查找函数？
- ▶ 可以打开 C++ 运行库的运行时检查，来寻找可能的这种错误



西安电子科技大学
XIDIAN UNIVERSITY

程序设计竞赛实训基地
Programming Contest Training Base



第 IV 部分

调试技巧





建议使用的警告选项：

- ▶ -Wall -Wextra
- ▶ -Wshadow：防止局部变量不小心遮盖其他变量
- ▶ -Wformat=2：防止 printf/ scanf 写错
- ▶ -Wconversion：防止意外的类型转换

某些情况下有用的警告选项：

- ▶ -Wstack-usage=1：看栈空间使用情况



```
1 struct mat
2 {
3     int a[4][4];
4     mat operator*(const mat &rhs) const
5     {
6         mat m;
7         for (int i = 0; i < 4; i++)
8             for (int j = 0; j < 4; j++) {
9                 m.a[i][j] = 0;
10                for (int k = 0; k < 4; k++)
11                    m.a[i][j] += a[i][k] * m.a[k][j];
12            }
13    }
14};
```



编译器立刻说出我忘了写 `return` 语句，还打错个变量名：

```
shell$ g++ -c -Wall -Wextra matrix_bug.cc
matrix_bug.cc: In member function 'mat mat::operator*(const mat&)
const':
matrix_bug.cc:13:2: warning: no return statement in function
      returning non-void [-Wreturn-type]
    13 |     }
        |     ^
matrix_bug.cc:4:27: warning: unused parameter 'rhs' [-Wunused-
parameter]
     4 |     mat operator*(const mat &rhs) const
        |                      ~~~~~~^~~
```



```
1  #include <iostream>
2  using namespace std;
3
4  const int M = 998244353;
5  int pow_mod(int a, int b); // impl skipped
6
7  int main()
8  {
9      long long a, b;
10     cin >> a >> b;
11     pow_mod(a, b);
12 }
```



```
shell$ g++ -c -Wall -Wextra -Wconversion pow_mod_bug.cc
pow_mod_bug.cc: In function 'int main()':
pow_mod_bug.cc:11:10: warning: conversion from 'long long int' to
    'int' may change value [-Wconversion]
    11 |     pow_mod(a, b);
        |               ^
pow_mod_bug.cc:11:13: warning: conversion from 'long long int' to
    'int' may change value [-Wconversion]
    11 |     pow_mod(a, b);
        |               ^
```

可以看出是抄快速幂板子忘了改参数类型。



编译警告虽然很有用，但由于停机问题是不可解的，不可能在编译期找出所有错误，这就需要使用运行期检查。下面介绍一些有用的，可以在区域赛使用的运行期检查工具：

- ▶ Undefined Behavior Sanitizer
- ▶ Address Sanitizer
- ▶ Libstdc++ Debug Mode

像 Valgrind 这种东西，现场赛不能用，就不说了（其实是我不会）



使用编译选项 `-fsanitize=undefined -g` 开启，用于寻找未定义行为。我们用一个初学者经常写出来的程序演示一下：

```
1 #include <iostream>
2 using namespace std;
3
4 const int M = 998244353;
5
6 int main()
7 {
8     int a, b;
9     cin >> a >> b;
10    long long c = a * b % M;
11    cout << c << '\n';
12    return 0;
13 }
```



```
shell$ g++ overflow.cc -fsanitize=undefined -g
shell$ echo "100000 100000" | ./a.out
overflow.cc:10:18: runtime error: signed integer overflow: 100000
    * 100000 cannot be represented in type 'int'
411821055
```



UBSan 虽然也能检测到一些越界访问的情况（毕竟这种情况也属于未定义行为），但对于稍微复杂一些的越界（涉及动态内存分配或者一些库函数）就无能为力了。例如：

```
1 #include <stdio>
2 using namespace std;
3
4 int main()
5 {
6     char buf[5];
7     scanf("%s", buf);
8     printf("hello, %s\n", buf);
9 }
```



这个程序就算开了 UBSan 也能正常运行，甚至在越界不多时，仍输出正确答案：

```
shell$ g++ scanf_bound.cc -fsanitize=undefined  
shell$ echo wang9897 | ./a.out  
hello, wang9897
```

然而交上去就自闭了……



这时就需要更专业的 Address Sanitizer 了，我们使用 `-fsanitize=address -g` 启用它：

```
shell$ g++ scanf_bound.cc -fsanitize=address -g
shell$ echo wang9897 | ./a.out
=====
==29446==ERROR: AddressSanitizer: stack-buffer-overflow on address
    0x7ffffba02d8d5 at pc 0x7fde4a9ef6dd bp 0x7ffffba02d760 sp 0
    x7ffffba02cf10
```

.....

```
#3 0x401228 in main /home/xry111/git-repos/2019-summer-lecture
-debug/code/scanf_bound.cc:7
```



编译时使用 `-D_GLIBCXX_DEBUG` 打开调试模式。此时 `libstdc++` 会插入两项运行时检查：

- ▶ 迭代器安全性检查
- ▶ 算法前提条件检查



我们用前面那个错误使用 vector 的迭代器的程序演示一下：

```
1 #include <vector>
2 #include <cstdio>
3 using namespace std;
4
5 int main()
6 {
7     vector<int> v = {1,2,3};
8     for (int i: v)
9         if (i > 0)
10             v.push_back(-i);
11     for (int i: v)
12         printf("%d\n", i);
13 }
```



```
shell$ g++ vec_invalidate.cc -D_GLIBCXX_DEBUG
shell$ ./a.out
/usr/include/c++/10.1.0/debug/safe_iterator.h:328:
In function:
    __gnu_debug::_Safe_iterator<_Iterator, _Sequence, _Category>&
    __gnu_debug::_Safe_iterator<_Iterator, _Sequence,
    _Category>::operator++() [with _Iterator =
    __gnu_cxx::__normal_iterator<int*, std::__cxx1998::vector<int,
    std::allocator<int> > >; _Sequence = std::__debug::vector<int
        >;
    _Category = std::forward_iterator_tag]
Error: attempt to increment a singular iterator.
```




如果忘了排序就二分查找：

```
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int arr[] = { 3, 1, 2 };
8     cout << binary_search(arr, arr+3, 2) << endl;
9 }
```



```
shell$ g++ bsearch_buggy.cc -D_GLIBCXX_DEBUG
shell$ ./a.out
/usr/include/c++/10.1.0/bits/stl_algo.h:2269:
In function:
    bool std::binary_search(_FIter, _FIter, const _Tp&) [with
        _FIter = int*;
        _Tp = int]
Error: elements in iterator range [__first, __last) are not
    partitioned by
the value __val.
```



如果某题要求时间复杂度 $\mathcal{O}(n \log n)$ ，你写好以后交上去 WA 了，又找不到错，可以写个 $\mathcal{O}(n^2)$ 的暴力，然后用 $n = 1000$ 左右的随机数据去检验。

- ▶ 前提是你暴力能写对
- ▶ 要小心，有时候随机数据并不能拍出所有 bug
- ▶ 如果你有别人已经 AC 的程序也可以用来拍



初始化随机数生成器的种子：

```
int x;  
scanf("%d", &x);  
srand(x);
```

然后用 rand() 取模去生成随机数据。

- ▶ 如果不初始化种子，每次都会用一样的种子，然后得到一样的数据。



```
1 #!/bin/sh
2
3 for ((i=0;;i++)) do
4     echo $i | ./rand > data-$i.in
5     ./brute < data-$i.in > data-$i.ans
6     if ! ./solution < data-$i.in > data-$i.out; then
7         echo "runtime error on test $i"
8         break
9     fi
10    if diff data-$i.ans data-$i.out; then
11        rm data-$i.{in,out,ans}
12    else
13        echo "wrong answer on test $i"
14        break
15    fi
16    echo "test $i ok"
17 done
```



如果我们用上面的方法发现了错误，但不知道为什么，就可能需要分析程序的执行过程……



调试宏：

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 #define DBG(x) \
5 (void)(cerr << "L" << __LINE__ \
6         << ": " << #x << " = " \
7         << (x) << endl)
8
9 int main()
10 {
11     int something = 233;
12     DBG(something);
13 }
```

输出的效果：L12: something = 233



<cassert> 提供了 assert 宏，可以用来确保前提条件的成立。

- ▶ 例如，BSGS 算法要求 a 和 M 互质，我们可以在自己的 BSGS 模板里面加上 `assert(gcd(a, M) == 1);`
- ▶ 如果使用 BSGS 时不满足这个条件，程序就会报错退出
- ▶ 同时，它也可以在敲模板的时候提醒你 BSGS 只能用于这种情况
- ▶ 提交代码时可以在程序开头加 `#define NDEBUG`，关闭所有断言，以避免不必要的运行时间



建议直接使用 GDB 的命令行，Code::Blocks 的调试非常难用。调试时需要打开编译选项 -g，建议禁用优化。GDB 的常用命令有：

- ▶ b (breakpoint) 行号/函数名
- ▶ r (run) [< 输入文件名]
- ▶ n (next)
- ▶ s (step)
- ▶ c (continue)
- ▶ p (print) 表达式
- ▶ d (disp) 表达式
- ▶ cond (condition) 断点编号 表达式
- ▶ bt (backtrace)
- ▶ fr (frame) 栈帧编号



- ▶ `gcov/-fctest-coverage -fprofile-arcs`: 代码覆盖率检测, 可以看代码中每一行被执行的次数
- ▶ `gprof/-pg`: 代码剖析, 可以看函数执行时间占总时间的百分比



BAPC 2018 E 题的一份 TLE 代码。我们先造一个大数据试一下：

```
shell$ g++ bapc2018e.cc -O2
shell$ time ./a.out < bapc2018e-data.txt
965105033

real    0m0.635s
user    0m0.634s
sys     0m0.001s
```

可以看到跑得很慢。



```
shell$ g++ bapc2018e.cc -O2 -fno-inline -pg
```

```
shell$ ./a.out < bapc2018e-data.txt
```

```
965105033
```

```
shell$ gprof
```

```
Flat profile:
```

```
Each sample counts as 0.01 seconds.
```

%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
97.49	0.74	0.74	1999	0.37	0.37	pow_mod(int,
						int)



```
shell$ g++ bapc2018e.cc -O2 -ftest-coverage -fprofile-arcs
shell$ rm *.gcda -f
shell$ ./a.out < bapc2018e-data.txt
965105033
shell$ gcov bapc2018e.cc > /dev/null
shell$ head bapc2018e.cc.gcov -n20
-:      0:Source:bapc2018e.cc
-:      0:Graph:bapc2018e.gcno
-:      0:Data:bapc2018e.gcda
-:      0:Runs:1
-:      1:#include <bits/stdc++.h>
-:      2:using namespace std;
-:      3:
-:      4:const int M = 1'0000'0000'9;
-:      5:
124124000: 6:int pow_mod(int a, int x)
```



```
-:      7:{
124124000:  8:      if (!x)
-:      9:          return 1;
120120000: 10:      long long t = pow_mod(a, x>>1);
120120000: 11:      t = t * t % M;
120120000: 12:      if (x&1)
64062000: 13:          t = t * a % M;
120120000: 14:      return t;
-:      15:}
-:      16:
```



- ▶ 两种方法都能发现快速幂使用了过多时间
- ▶ gprof 输出的是时间，但只能精确到函数
- ▶ gcov 精确到行，但只能输出调用次数



- ▶ 找别人帮忙调程序几乎一定会破坏友谊
- ▶ 如果一定要找，把代码的可读性弄得好一点……
- ▶ 小黄鸭调试法



西安电子科技大学
XIDIAN UNIVERSITY

程序设计竞赛实训基地
Programming Contest Training Base



感谢观看！