# Day 2

# 单调栈与单调队列

# 单调栈

满足单调性的栈结构。与单调队列相比，其只在一端进行进出。

将一个元素插入单调栈时，为了维护栈的单调性，需要在保证将该元素插入到栈顶后整个栈满足单调性的前提下弹出最少的元素。

# 单调队列

与单调栈不同的是，单调队列可以在队列的两边进行push和pop操作（可借助STL中的deque，即双端队列来完成）。

# 适用于单调栈解决的问题

由于单调栈是只在一端操作的数据结构，因而需要保证所求解的问题是当前解由原有的解不断优化而成，而不需要处理部分解无效的情况（如查询连续k个区间最大最小值就是一个不断更新不断舍弃的过程）。

# 适用于单调队列解决的问题

问题的构成解是需要在两端操作的（如查询连续k个区间最大最小值）

# 例题1

*M* 海运公司最近要对旗下仓库的货物进出情况进行统计。目前他们所拥有的唯一记录就是一个记录集装箱进出情况的日志。该日志记录了两类操作：第一类操作为集装箱入库操作，以及该次入库的集装箱重量；第二类操作为集装箱的出库操作。这些记录都严格按时间顺序排列。集装箱入库和出库的规则为先进后出，即每次出库操作出库的集装箱为当前在仓库里所有集装箱中最晚入库的集装箱。

出于分析目的，分析人员在日志中随机插入了若干第三类操作——查询操作。分析日志时，每遇到一次查询操作，都要报告出当前仓库中最大集装箱的重量。

# 例题2

给定一行$n$个非负整数$a[1]\ldots a[n]$。现在你可以选择其中若干个数，但不能有超过$k$个连续的数字被选择。你的任务是使得选出的数字的和最大。

可转化为求解连续$k$个数中最小的一个数，（可用单调队列求解），而后用动态规划去选择去掉哪些数。

# 例题3

$n$ 个人正在排队进入一个音乐会。人们等得很无聊，于是他们开始转来转去，想在队伍里寻找自己的熟人。

队列中任意两个人 $a$ 和 $b$，如果他们是相邻或他们之间没有人比 $a$ 或 $b$ 高，那么他们是可以互相看得见的。

写一个程序计算出有多少对人可以互相看见。

# 二分查找法

# 二分查找法

二分法的核心其实集中在 mid 和 answer 的大小关系上,不管是 min（下界）、max（上界） 或是 your_ans（当前值） 的变动都围，绕着 mid 和 answer 的关系展开。

二分法的要求:

1.答案在一个固定区间内;

2.可能查找一个符合条件的值不是很容易，但是要求能比较容易地判断某个值是否是符合条件的;

3.可行解对于区间满足一定的单调性。

# 二分查找法（普通版本）

将在a的区间$[x,y)$内寻找v。
（此部分的区间数据按从小到大排列。）

```
int bsearch(int x, int y, int v)
{
    while (x<y)// 注意：不要写成“x<=y”，否则会死循环！
    {
        int mid=x+(y-x)/2;
        if (v==a[mid])
            return mid;
        else if (v<a[mid])
            y=mid;
        else
            x=mid+1; // 注意：不要忘记加1！
    }
    return -1;// 找不到
}
```

# 二分查找法（求上界）

寻找a的区间$[x,y)$小于等于v的最后一个数
（此部分的区间数据按从小到大排列。）

```
int upper_bound(int *a, int x, int y, int v)
{
    while (x < y)
    {
        int mid = x + (y - x) / 2;
        if (v >= a[mid])
            x = mid;
        else
            y = mid - 1;
    }
    return y;
}
```

# 二分查找法（求下界）

寻找a的区间$[x,y)$中大于等于v的第一个数
（此部分的区间数据按从小到大排列。）

```
int lower_bound(int *a, int x, int y, int v)
{
    while (x < y)
    {
        int mid = x + (y - x) / 2;
        if (v <= a[mid])
            y = mid;
        else
            x = mid + 1;
    }
    return x;
}
```

# 三分法

对于在某区间内先单调增后单调减的函数，可以通过三分法以任意精度确定最大值位置

注意：对于实数的情况，不要写while(r – l < eps)，要写固定循环次数

$$mid = (l + r)/2$$

$$mid2 = (mid + r)/2$$

| | 作用 | 条件 |
| --- | --- | --- |
| 一般的二分法 | 求解一个函数的零点 | 函数在这个区间是单调函数 |
| 三分法 | 求解一个函数的极大（极小）值点 | 函数在这个区间是凸（凹）函数 |

# 筛法

# Eratosthenes筛法求素数表

```
bool visited[N]; // 如果被筛，设为true
int primes[N], n=0; // 素数表

memset(visited,0,sizeof(visited));
for (i=2;i<N;i++)
    if (!visited[i])
    {
        primes[n++]=i;
        for (j=i+i;j<N;j+=i) visited[j]=true;
    }
```

# 线性筛法

Eratosthenes 筛法利用的原理是 任意整数 $x$ 的倍数 $2x, 3x, \ldots$ 等都不是质数 。

但是即便如此也会有重复标记的现象，例如12既会被2又会被3标记，在标记2的倍数时，12=6*2，在标记3的倍数时，12=4*3 ，根本原因是没有找到唯一产生12的方式。

线性筛核心原理：每个合数必有一个最大因子（不包括它本身），用这个因子把合数筛掉。

对于一个确定的整数 $i$，$i$ 是一个合数 $t$ 的最大因数，必然有一个$p$，满足：

$$t = i * p \ (p \leq i, p\text{是质数})$$

我们只需要把所有小于 $i$ 的质数 $p$ 都挨个乘一次拿到所有合数就好了。

但这样仍然会有重复标记：

不能让 $p$ 大于 $i$ 的最小质因数（设为$x$），否则 $i$ 将不再是 $t = i * p$ 的最大因数

```cpp
#include <cstring>

int visited[N]; // 如果i是素数，对应设为i
int primes[N], n=0; // 素数表

memset(visited,0,sizeof(visited));
for(int i = 2; i <= N; i++){
    if(!visited[i]){
        visited[i] = i;
        primes[n] = i;
        n++;
    }
    for(int j = 0; j < n; j++){
        if(primes[j] > visited[i]
                || primes[j] > N/i) break;
        visited[primes[j] * i] = primes[j];
    }
}
```

```cpp
#include <cstring>
bool visited[N]; // 如果i是素数，对应设为true
int primes[N], n=0; // 素数表

memset(visited,true,sizeof(visited));
for(int i = 2; i <= N; i++){
    if(visited[i])
        primes[n++] = i;
    for(int j = 0; j < n; j++){
        if(primes[j] > N/i) break;
        visited[primes[j] * i] = false;
        if(i % primes[j] == 0) break;
    }
}
```

# 差分与前缀和

# 差分

令$b_i = a_i - a_{i-1}$，即相邻两数的差。

易得对这个序列进行一次前缀和就得到了原来的a序列。

# 例题

**题目描述**

HJ养了很多花（99999999999999999999999999999999999999盆），并且喜欢把它们排成一排，编号0~99999999999999999999999999999999999998,每天HJ都会给他的花浇水，但是他很奇怪，他会浇n(1 <= n <= 2 * $10^5$)次水，每次都会选择一个区间[l, r]，(0 <= l <= r <= $10^6$),表示对区间[l, r]的花都浇一次水。现在问你，通过这些操作之后，被浇了i(1 <= i <= n)次水的花的盆数。

**输入描述:**

输入：第一行一个n，表示HJ的操作次数，接下来的n行，表示每一次选择的浇水区间。

**输出描述:**

输出：输出n个数字Cnt1，Cnt2……Cntn，（用空格隔开）Cnti表示被浇了i次水的花的盆数。

**示例1**

输入 复制

```
3
0 3
1 3
3 8
```

```cpp
#include <bits/stdc++.h>
#define manx 1000005
using namespace std;
int a[manx];//a[i]=a[i]-a[i-1]
int sum[manx],n;
int main(){
    ios::sync_with_stdio(false);
    cin >>n;
    int minn=0x3f3f3f3f,mann = 0;
    for(int i =0;i<n;i++){
        int l,r;
        cin >>l>>r;
        minn = min(minn,l);
        mann = max(mann,r);
        a[l]++;
        a[r+1]--;
    }
    for(int i=max(0,minn);i<=mann;i++){
        a[i] += a[i-1];
        sum[a[i]]++;
    }
    for(int i =1;i<=n;i++)
        cout <<sum[i]<<" ";
    return 0;
}
```

# 前缀和

对数组的一个连续区间求和，使用枚举的时间为$O(n)$。然而，如果需要对$m$个区间求和，枚举法的时间复杂度就为$O(mn)$，这对于某些问题来说是无法承受的。所以要采用一种方法来加快求和的速度。

# 区间求和

一个数组有$n$个数字，然后有$m$个询问，每个询问的内容都是：求第$i$项到第$j$项中所有元素的和（$0 < n \leq 1000000$，$0 < m \leq 1000000$，$0 < i < j \leq n$）。

显然$m$个询问是一定要回答的，所以我们只能减小每次回答询问的时间。

用$S[i]$表示前$i$个数字的和。

$S[i]$的递推公式：$S[i] = S[i\text{-}1] + a[i]$，边读入数据边求出$S$。

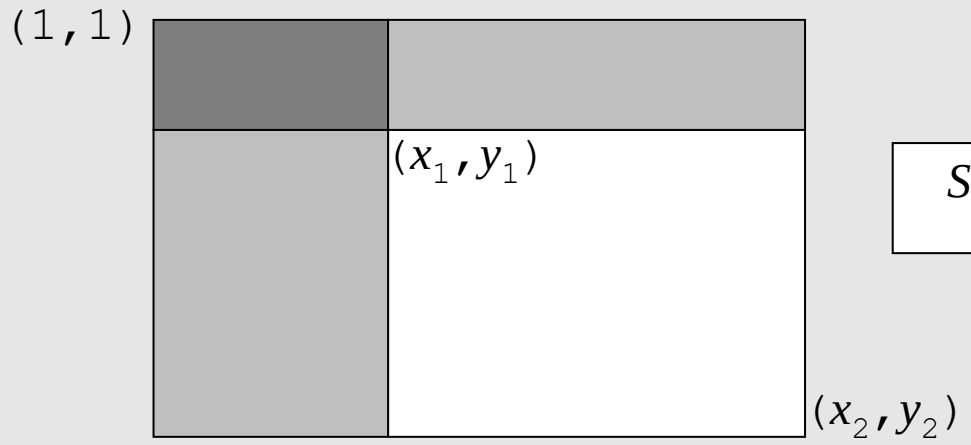
第$i$项到第$j$项中所有元素的和 = $S[j] - S[i\text{-}1]$。用$O(1)$的时间就算出来了。

# 最大子矩阵和

给出一个$m \times n$的矩阵。求一个子矩阵，使子矩阵中每个数加到一起的总和最大。

最容易想到和实现的方法是一个六次方时间的算法：枚举子矩阵的左上角、右下角，然后求和。

和区间求和一样，矩阵也可以用类似的方法求和：设sum[i][j]表示从(1,1)到$(i,j)$的所有元素之和，在读取数据时将其计算好，那么左上角为$(x_1,y_1)$，右下角为$(x_2,y_2)$的子矩阵和$S$可以这样求




$$S=\text{sum}(x_2,y_2)-\text{sum}(x_2,y_1)-\text{sum}(x_1,y_2)+\text{sum}(x_1,y_1)$$

这样，求和的时间就由二次方降到了O(1)。总时间由六次方降到了四次方。

对上面的算式进行变形，得$S=[\text{sum}(x_2,y_2)-\text{sum}(x_2,y_1)]-[\text{sum}(x_1,y_2)-\text{sum}(x_1,y_1)]$。仔细观察，可以发现，每个中括号内只有三个变量。两个中括号内都有$y_1$、$y_2$，假设它们的值确定了，就相当于$S$是一行之内的区间和。

因此，问题就转化为枚举$y_1$和$y_2$，并将子矩形压缩成一行，求其最大子序列和。

（此外最大子序列和可用动态规划[day3的内容]求出：设$f(i)$为某一“行”上的最大子序列和，则$f(i)=\max\{f(i-1)+a[i],\ a[i]\}=\max\{f(i-1),0\}+a[i]$。实际上$a[i]$是$\text{sum}(i,y_2)-\text{sum}(i-y_1)$。

这样总时间由四次方降到了三次方。）

# 数论（基础）

同余的性质

$$(a+b)\ mod\ m=(a\ mod\ m+b\ mod\ m)\ mod\ m$$

$$(a-b)mod\ m=(a\ mod\ m-b\ mod\ m+m)mod\ m$$

$$(ab)\ mod\ m=(a\ mod\ m)\times(b\ mod\ m)\ mod\ m$$

# 求最大公约数、最小公倍数

# 最大公约数

即欧几里得算法（辗转相除法）

```
int gcd(int a, int b)
{ return (b == 0) ? a : gcd(b, a % b); }

int gcd(int a, int b)
{
    int t = a % b;
    while (t)
    {
        a = b;
        b = t;
        t = a % b;
    }
    return b;
}
```

# 最小公倍数

$$lcm(a,b) = \frac{a}{\gcd(a,b)} \times b$$

```
int lcm(int a, int b)
{ return a / gcd(a, b) * b; }
```

# 扩展欧几里得算法（解不定方程）

$ax+by=c$

第一步：如果$c \ mod \ \gcd(a,b) \neq 0$，那么方程无解。

第二步：方程可变为$a'x+b'y=c'$，其中

$$a' = \frac{a}{\gcd(a,b)}, b' = \frac{b}{\gcd(a,b)}, \ c' = \frac{c}{\gcd(a,b)}$$

第三步：求解（见代码）

第四步：构造通解：假设 $x_0$， $y_0$ 是一组解，那么通解为$\begin{cases} x = cx_0 + cbk \\ x = cy_0 - cak' \end{cases}, k \in Z$。

```
void exgcd(int a, int b, int c, int &x, int &y)
{
    if (a == 0)
    {
        x = 0;
        y = c / b;
    }
    else
    {
        exgcd(b % a, a, c, x, y);
        y = x;
        x = (c - b * y) / a;
    }
}
```

# 极端数据的优化

数据的特殊情况（如只有一组数据，达到临界值的数据，或是恰好满足题设的数据）

判断测试点的情况：在进行猜想的基础上，并通过测试点提交测试代码，通过TLE等报错来判断测试点的特点

在有条件的情况下，可以写对拍来验证（4C不需要）

# 程序查错（调试办法）

# 常见的错误类型

1.思路错误：

如果算法错误，你就只好认倒霉了——重新写代码。

如果是功能缺陷，只需修补一下。不过，修补之后要认真调试，防止产生新错误。

2.语法错误：只有这种错误可以用编译器查出来。

3.书写错误：把j错写成i，把“==”错写成“=”，浮点数直接判断相等……这种错误很容易犯，并且不容易查出来。所以，写代码时要细心。

4.忘记初始化：sum、max、min忘初始化，或者用不应该的数初始化。以下的初始化都是错误的：min=INF、max=0、max=4294967295（太大以至于容易溢出）

5.中间值溢出：lcm = a*b/gcd(a,b); 如果a和b都比较大，很容易发生溢出。

6.同名变量导致混乱：最简单的办法是保证每个变量都不重名，忽略大小写之后仍然不重名。

7.内存泄漏（段错误）：数组没开足够大的空间、用了野指针

8.格式错误：细节处理问题

| 问题种类 | 审查项 |
| --- | --- |
| 程序的版式 | 空行、空格能否帮助你理解代码？ |
| | 你的代码缩进正确吗？ |
| | ++、--、+=、-=.....是否单独存在？ |
| 变量 | 你能说出每个变量的作用吗？ |
| | 在你的变量中有重名的吗？如果有，马上改成不一样的。 |
| | 在进行比较、计算、输出之前，你的变量是否已经初始化？ |
| | max、min的初始值是否正确？（max应该是0，min应该是一个大数） |
| | 变量的精度够吗？ |
| 表达式 | 你是否在复杂表达式中加了括号？ |
| | 你是否写了数学表达式（如1<a<=n）？赶快改过来吧。 |
| | 逻辑判断是否存在“安全隐患”（如浮点数直接判断相等）？ |
| | 运算过程中，会发生溢出吗？ |
| 函数 | 你知道每个参数的含义吗？顺序合理吗？ |
| | 你能否把正常结果和错误区分开？ |
| | 如果你的函数返回一个指针或引用，你是否天真地返回了函数内部的变量？ |
| 数组 | 你知道数组编号的下限吗？是0，还是1？ |
| | 数组够大吗？你知不知道数组大小应该比最大数据规模大一些，以防出错？ |
| 宏定义 | 能否把对应代码换成内联函数？ |
| | 是否在表达式外面加了括号？“参数”本身是否有括号？ |
| | “参数”中是否修改了变量的值（如使用++、--、+=）有的话赶快移走。 |
| 判断与循环 | 条件正确吗？ |
| | 你是否用了一个很复杂的判断？尽量把它们分解成简单的条件。 |
| | 不等号的方向正确吗？ |
| | 不等号中是否包含相等？你有没有把“<”写成“<=”，或写成别的？ |
| | 你是否把“==”写成“=”？ |
| | 你是否在for循环内部修改了循环变量？（不要改） |
| | 符合题目规定的格式吗？ |
| | 数据规模超过几千时，你还在使用流来输入输出数据吗？ |
| 其他 | memset的第二个参数是多少？是0或-1吗？（不可以是其他值） |

| 问题种类 | 审查项 |
| --- | --- |
| 程序崩溃 | 数组下标是否出现了负数，或者超过了上限？ |
| | 是不是递归的深度太大？ |
| | 在函数内部是否有大数组？如果有，请把它挪到外面。 |
| | 是不是变量忘初始化了？或者是变量名打错了？ |
| | 循环条件中的字母对吗？不等号的方向正确吗？ |
| | 指针是否初始化？ |
| | 对指针使用“*”运算符时，它是否指向NULL，或一个未知的地址？ |
| 程序卡死 | 是不是算法效率太低，导致程序一直在慢慢地计算？ |
| | 是不是有死循环？循环终止条件打错了？字母错了？多写或少写了等号？ |
| | 如果使用链表（图），是不是因为链表（图）中出现了环（包括自环）？ |
| 出现奇怪的输出 | 变量初始化了吗？ |
| | 变量名打错了吗？ |
| | 是否在运算过程中发生了溢出？ |
| | 变量精度够大吗？ |
| | 数组够大吗？（定义a[100]，结果正好101个元素） |
| | 如果使用字符串函数，字符串末尾是不是'\0'？ |
| | 是否把“==”错打成“=”，或者在不等号中错误地包含等号？ |
| | 是否尝试指针或动态内存分配，结果出现了问题？ |

（1）栈区（stack）— 由编译器自动分配释放，存放函数的参数值，局部变量的值等。
（2）堆区（heap） — 由程序员分配和释放，若程序员不释放，程序结束时可能由OS回收。
（3）全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量、未初始化的静态变量在相邻的另一块区域。
（4）文字常量区 — 常量字符串就是放在这里的。
（5）程序代码区 — 存放函数体的二进制代码。

gdb中显示程序收到了SIGSEGV信号——段错误。这太让人沮丧了！眼看本章就要结束了，怎么又遇到一个段错误？别急，让我们慢慢分析。我保证，这是本章最后的难点。

你有没有想过，编译后产生的可执行文件里都保存着些什么内容？答案是和操作系统相关。例如，UNIX/Linux用的ELF格式，DOS下用的是COFF格式，而Windows用的是PE文件格式（由COFF扩充而来）。这些格式不尽相同，但都有一个共同的概念——段。

“段”（segmentation）是指二进制文件内的区域，所有某种特定类型信息被保存在里面。

在可执行文件中，正文段（Text Segment）用于储存指令，数据段（Data Segment）用于储存已初始化的全局变量，BSS段（BSS Segment）用于储存未赋值的全局变量所需的空间。

是不是少了点什么？调用栈在哪里？它并不储存在可执行文件中，而是在运行时创建。

调用栈所在的段称为堆栈段（Stack Segment）。和其他段一样，堆栈段也有自己的大小，不能被越界访问，否则就会出现段错误（Segmentation Fault）。

这样，前面的错误就不难理解了：每次递归调用都需要往调用栈里增加一个栈帧，久而久之就越界了。这种情况叫做栈溢出（Stack Overflow）。

那么栈空间究竟有多大呢？这和操作系统相关。在Linux中，栈大小是由系统命令ulimit指定的，例如，ulimit -a显示当前栈大小，而ulimit -s 32768将把栈大小指定为32MB。但在Windows中，栈大小是储存在可执行文件中的。使用gcc可以这样指定可执行文件的栈大小：gcc -Wl,--stack=16777216，这样栈大小就变为16MB。

聪明的读者，现在你能理解为什么在介绍数组时，建议“把较大的数组放在main函数外”了吗？别忘了，局部变量也是放在堆栈段的。栈溢出不一定是递归调用太多，也可能是局部变量太大。只要总大小超过了允许的范围，就会产生栈溢出。

# 查错办法

1.静态查错：不要运行程序。静下心来，慢慢地用你的思路、框图和伪代码检查代码，看是否有打错的或者漏打的内容。

一般要先查局部，后查整体。调试前先静态查错。

2.样例测试：如果示例的结果都不对，就应该考虑算法的正确性，并检查代码是否写错。

3.构造小数据：根据程序功能设计几个数据，检查程序是否计算出正确结果。

4.构造极端数据

5.对于部分模块，可以采用单步运行的方法。不过，这需要耐心。

6.修改代码之后还需要检查、调试，想想错误是否改正确了，是否改彻底了（浏览一下整个程序，看是否有类似错误）。

# DEVCPP的调试功能

1.一般的IDE都会有以下几种调试功能：

- 断点（Ctrl+F5）：程序会在执行到这一步时暂停
- 单步进入（Shift+F7）、下一步（F7）：它们的区别是在遇到过程或函数时，前者会深入一个函数的内部，而后者会直接计算函数值。
- 运行到光标处（Shift+F4）
- Watch（查看，添加查看为F4，查看变量为Ctrl+W）
- 查看调用栈（在屏幕下方）

2.DEV-C++的调试功能很差，如果不用IDE的调试功能，也可以采用以下方法调试：

- 把变量的值输出到屏幕上。
- 设定输出时，程序的输出要方便查看，并且不能太多，这样才能便于你调试。
- 暂停屏幕：system("pause");或while(1);
- 注意，对stdin使用输入重定向后，system("pause")不能再暂停屏幕。

# 符号：DEBUG

调试代码在不用时，不应该直接删除，而是应该注释掉，以免给重新使用带来麻烦。不过，忘记把调试代码删除，爆0的几率会非常大。所以可以按照以下方法做：

① 把调试代码放到#ifdef块中，即

```
#ifdef DEBUG
    加入你的调试代码
#endif
```

② 在IDE的编译选项中加一个参数：-DDEBUG

# g++和gdb

g++是编译器，gdb是调试工具。在DEV-C++中可以到“工具”下的“编译选项”中设置g++的开关。DEV-C++支持直接操作gdb，但是很不好用。

在Linux中，可以使用“终端”直接调用g++和gdb。在Windows中，需要对PATH环境变量进行修改，加入g++和gdb所在目录，才能在“命令提示符”中直接调用它们。下面假设你的工作目录是程序所在目录，可以直接调用g++和gdb，并且已经打开了“终端”或“命令提示符”。

如果手动编译程序，输入“g++ test.cpp -o test.exe”（Linux下不需要“.exe”，下同）。

以下是g++常用的编译参数：

- -o：指定编译之后的程序名。如果不输入，程序就叫“a.exe”。
- -Wall：输出警告。
- -DDEBUG：编译时定义一个叫“DEBUG”的符号。
- -O1、-O2、-O3：优化。从1到3速度由慢到快。但“-O3”容易误解程序员意思。
- -g：产生调试符号。加入“-g”之后就可以用gdb直接调试程序。
- -lm：自动链接数学库（<math.h>）

| 命令 | 全称 | 作用 | 实例和说明 |
| --- | --- | --- | --- |
| l | list | 查看源代码。 | l 15：显示第15行及附近代码。<br>l main：显示main()附近10行。<br>如果不带参数，将继续显示上次代码的后10行。 |
| b<br>cl | break<br>clear | 设置断点。<br>取消断点。 | b 24或cl 24：在第24行设置/取消断点。<br>b main或cl main：在main()入口处设置/取消断点。 |
| r | run | 运行程序。 | 一直运行，直到遇到断点或程序结束。 |
| c | continue | 继续运行。 | 中断之后使用。 |
| k | kill | 杀死正在调试的进程。 | 停止调试。 |
| u | until | 让程序运行到指定位置。 | u 9：运行到第九行，然后暂停。<br>u search：运行到search()的入口，然后暂停。 |
| disp | display | 相当于IDE里的“Watch”。 | disp x：每次程序暂停，自动输出x的值。<br>disp x+1：自动输出表达式“x+1”的值。<br>dis disp或en disp：禁用/启用所有Watch。 |
| n<br>s | next<br>step | 单步运行。 | 单步执行。它们区别是一旦遇到函数，“s”要进入函数内部而“n”直接计算函数的值然后继续。 |
| P | print | 计算表达式。 | p a：输出变量a的值。<br>p 1+1：计算“1＋1”的值。 |
|  | call | 执行一条C++代码，如果有返回值就输出到屏幕上 | call j=2：改变变量j的值<br>call print()：调用print() |
| i | info | 显示信息。 | i b：显示所有断点<br>i lo（info locals）：显示所有局部变量。<br>i disp：显示所有“Watch”。 |
| d | delete | 删除。 | d disp：删除所有“Watch”。<br>d breakpoints：删除所有断点。 |
| bt | backtrace | 查看调用栈。 |  |
| q | quit | 退出gdb。 |  |
| h | help | 获得帮助信息。 | 全部是英语。 |
|  | whatis | 查看变量类型 | whatis n：查看n的类型 |
| 直接回车 |  | 执行上一条命令。 | 在输入n或s后一路回车，相当于打了一路n或s。 |

以下是和断点有关的命令。使用之前，要知道断点的编号（可用“i b”查看）：

| 命令 | 全称 | 作用 | 实例和说明 |
| --- | --- | --- | --- |
| ig | ignore | 让断点在前n次到达时都不停下来。 | ig 3 12：让编号为3的断点在前12次到达时都不停下来。 |
| cond | condition | 给断点加一个条件。 | cond 2 i>3：2号断点只有在i>3时才起作用。 |
| comm | commands | 在某个断点处停下来后执行一段gdb命令。 | comm 4：在断点4停下来后执行一段命令。输入这条命令后，就输入要执行的内容。 |
| wa | watch | 当变量或表达式的值发生改变时停下来。 | wa i：当i的值发生改变时停下来。 |
| aw | awatch | 变量被读写时都会停下来。 | aw i：当i被读写时都会停下来。 |
| rw | rwatch | 当变量被读的时候停下来。 | rw i：当i被读时停下来。 |

此外需要注意，某些函数可能无法使用，如调用sort()时提示“No symbol "sort" in current context.”。你可以自己写一个形参和它一致的mysort()，让mysort()去调用sort()。

# 对拍

# 在Windows下

下面假设程序为test.exe，输入文件为test?.in，输出文件为test?.out（问号代表0~9的数字），标准答案为test?.ans。

下面每组都是批处理文件的内容。这些内容应该被保存成bat格式的文件，而不是cpp。

注意，rem后面的内容都是注释。

1. 单个输入文件

```
@echo off
copy test?.in test.in >nul
time<nul
test.exe
time<nul
fc test.out test?.ans
pause
```

2. 一组输入文件

```
@echo off

rem 如果没有指定测试点，就进行总测试
if "%1"=="" goto loop

copy test%1.in test.in >nul
echo 数据点%1:
time<nul
test.exe
time<nul
fc test.out test%1.ans
pause
goto end

:loop
rem 将每个测试点都测试一下
for %%i in (0 1 2 3 4 5 6 7 8 9) do call %0 %%i

:end
del test.in
del test.out
```

# 在Linux下

下面假设待测程序文件名为test，标准答案生成器的文件名为std，输入数据生成器的文件名为gen。

下面是批处理文件的内容。批处理文件的扩展名为sh，这些内容应该被设置为“可执行的”（假设脚本为test.sh，则需要执行“chmod +x test.sh”），然后用“./test.sh”运行。

```
#!/bin/bash
seed=1          #随机种子，用系统时间只能每秒一次，太慢，所以直接从1开始循环
while true;do
echo "Seed: $seed"
./gen <<< "$seed" #生成数据，生成器从标准输入读入随机种子
./std
mv test.out std.out #产生标准答案
./test
if ! diff test.out std.out; then break; fi #发现不同时在屏幕上显示不同之处，
并中止对拍
seed=seed+1
done
```

# 快读

```
inline int read()
{
    int x=0,y=1;char c=getchar();
    while (c<'0'||c>'9') {if (c=='-') y=-1;c=getchar();}
    while (c>='0'&&c<='9’)
           x= (x << 3) + (x << 1) + c-'0',c=getchar();
    return x*y;//乘起来输出
}
```