NOIP 复习资料

（C++版）

主　编　　葫芦岛市一高中　李思洋

完成日期　　2012 年 8 月 27 日

前 言

有一天，我整理了 NOIP 的笔记，并收集了一些经典算法。不过我感觉到笔记比较凌乱，并且有很多需要修改和补充的内容，于是我又搜集一些资料，包括一些经典习题，在几个月的时间内编写出了《NOIP 复习资料》。

由于急于在假期之前打印出来并分发给同校同学（**我们学校既没有竞赛班，又没有懂竞赛的老师。我们大家都是自学党**），《NOIP 复习资料》有很多的错误，还有一些想收录而未收录的内容。

在“减负”的背景下，暑期放了四十多天的假。于是我又有机会认真地修订《NOIP 复习资料》。

我编写资料的目的有两个：总结我学过（**包括没学会**）的算法、数据结构等知识；与同学共享 NOIP 知识，同时使我和大家的 RP++。

大家要清醒地认识到，《NOIP 复习资料》页数多，是因为程序代码占了很大篇幅。这里的内容只是信息学的皮毛。对于我们来说，未来学习的路还很漫长。

基本假设

作为自学党，大家应该具有以下知识和能力：

① 能够熟练地运用 C++语言编写程序（**或熟练地把 C++语言“翻译”成 Pascal 语言**）；

② 能够阅读代码，理解代码含义，并尝试运用；
③ 对各种算法和数据结构有一定了解，熟悉相关的概念；
④ 学习了高中数学的算法、数列、计数原理，对初等数论有一些了解；
⑤ 有较强的自学能力。

代码约定

N、M、MAX、INF 是事先定义好的常数（**不会在代码中再次定义，除非代码是完整的程序**）。N、M、MAX 针对数据规模而言，比实际最大数据规模大；INF 针对取值而言，是一个非常大，但又与 int 的最大值有一定差距的数，如 100000000。

对于不同程序，数组下标的下限也是不同的，有的程序是 0，有的程序是 1。阅读程序时要注意。

阅读顺序和方法

没听说过 NOIP，或对 NOIP 不甚了解的同学，应该先阅读附录 E，以加强对竞赛的了解。

如果不能顺利通过初赛，你就应该先补习初赛知识。这本《NOIP 复习资料》总结的是复赛知识。

如果没有学过 C++语言，应该先选择一本 C++语言教材。一般情况下，看到“面向对象编程”一章的前一页就足够了（NOIP **不用“面向对象编程”，更不用摆弄窗口对话框**）。

附录 G 介绍了一些书籍和网站。你应该选择一本书，认真地学习。再选择一个网站，作为练习的题库。

第一单元对竞赛中常用的操作和简单的算法分析进行了总结，算作对 C++语言的巩固。同时，阅读这一单元之后，你应该选择一个合适的 C++代码编辑器。

第二到第六单元介绍了竞赛常用的算法。阅读每一章时，应该先阅读“小结”——名曰“小结”，实际上是“导读”。

这五个单元除了经典习题，还有某些思想和算法的具体实现方法。这些信息可能在明处，也可能在暗处，阅读时要注意挖掘和体会。如果有时间，应该在不看解析和代码的前提下独立完成这些题。

第七单元是第六单元的一个部分，由于它的内容来自《背包九讲》，所以单独放在一个单元。

从第八单元开始，到第十三单元，基本上就没有习题了。换句话说，该“背课文”了。

第八单元介绍了常用的排序算法。你可以有选择地学习，但一定要掌握“STL 算法”和“快速排序”。

第九单元介绍了基本数据结构，你一定要掌握第九单元前五小节的内容（**本单元也有应该优先阅读的“小结”**）。有余力的话，第六小节的并查集也应该掌握。

第十单元介绍了与查找、检索有关的数据结构和算法。你也可以有选择地学习。

第十一单元与数学有关。数学对于信息学来说具有举足轻重的地位。标有“！”的应该背下来，至于其他内容，如果出题，你应该能把它解决。

第十二单元仍与数学有关。

第十三单元是图论。学习时要先阅读“小结”，把概念弄清楚。之后要掌握图的实现方法。接下来要掌握一些经典图论算法：Kruskal 算法、Dijkstra 算法、SPFA、Floyd 算法、拓扑排序。

附录 F 总结了 2004 年以来 NOIP 考察的知识点，可以作为选择性学习的参考。

在学习算法和数据结构的同时，应该阅读和学习附录 A。

如果你还有余力，你应该学习第十四单元。第十四单元的内容不是必须要掌握的，但是一旦学会，可以发挥 C++语言的优势，降低编程复杂度。

临近竞赛时，应该阅读附录 B 和附录 C，以增加经验，减少失误。

面临的问题

1. 这是复赛复习资料——需要有人能用心总结、整理初赛的知识，就像这份资料一样。
2. 潜在的问题还是相当多的，只是时间不够长，问题尚未暴露。
3. 部分代码缺少解说，或解说混乱。
4. 个人语文水平较差，《资料》也是如此。
5. 没有对应的 Pascal 语言版本。
   如果有人能为 P 党写一个 Pascal 版的 STL，他的 RP 一定会爆增！
6. 希望有人能用LaTeX整理《资料》，并以自由文档形式发布。

最后，欢迎大家以交流、分享和提高为目的修改、复制、分发本《资料》，同时欢迎大家将《资料》翻译成 Pascal 语言版供更多 OIer 阅读！

谢谢大家的支持！

葫芦岛市一高中　李思洋

2012 年 8 月 27 日

目 录

标题上的符号：

1. !：表示读者应该熟练掌握这些内容，并且在竞赛时能很快地写出来。换句话说就是应该背下来。
2. *：表示内容在 NOIP 中很少涉及，或者不完全适合 NOIP 的难度。
3. #：表示代码存在未更正的错误，或算法本身存在缺陷。

第一单元　C++语言基础

1.1　程序结构

(1) 程序框架

- 注释：注释有两种，一种是“//”，另一种是“/* … */”。“//”必须单独放置一行，或代码所在行的后面；而“/*”、“*/”成对存在，可以插入到代码的任意位置。
- 引用头文件：在代码开头写“#include <头文件名>”。如果想引用自己的头文件，需要把尖括号（**表示只从系统目录搜索头文件**）换成双引号（**表示先从 cpp 所在文件夹搜索，然后再到系统文件夹搜索**）。
- 命名空间：很多 C++的东西都要引用 std 命名空间，所以代码中会有“using namespace std;”。
- main()：所有程序都要从 main()开始。
  在所有的算法竞赛中，main()的返回值必须是 0，否则视为程序异常结束，得分为 0 分。
- 语句和语句块：
  1. 语句：一般情况下，一条语句要用一个分号“;”结束。为了美观和可读性，可以把一条语句扩展成几行，也可以把多个语句写到同一行上。
  2. 语句块：用“{”和“}”包围的代码是语句块。无论里面有多少代码，在原则上，语句块所在的整体都视为一条语句。

(2) 选择结构

1. if 语句：if 表示判断。如果条件为真，就执行接在 if 后的语句（**语句块**），否则执行 else 后的语句（**语句块**）。如果没有 else，就直接跳过。if 有以下几种格式：

```
if (条件)           // 如果条件成立，就执行if后面的语句或语句块。
    语句或语句块

if (条件)           // 如果条件成立，就执行if后面的A，否则执行B。
    语句或语句块A
else
    语句或语句块B

if (条件1)          // 实际上，这是if语句内的if语句，即if的嵌套。所以else和if中间要有空格。
    语句或语句块A
else if (条件2)
    语句或语句块B
……
else
    语句或语句块N
```

2. switch 语句：switch 表示选择。它根据条件的不同取值来执行不同的语句。格式如下：

```
switch (表达式)
{
case 值1:
    代码段A
```

```
    break;
case 值2:
    代码段B
    break;
……
default:
    代码段N
    break;
};
```

如果表达式的值是值 1，就执行代码段 A；如果表达式的值是值 2，就执行代码段 B……否则执行代码段 N。

注意：

- default 一部分可以省略。
- 如果不使用 break，那么紧随其后的 case 部分代码也会被执行，直到遇到 break 或 switch 语句结束为止！
- switch 结尾要有一个分号。

3. if、switch 都可以嵌套使用。

【问题描述】输入一个日期，判断它所在年份是否为闰年，并输出所在月份的天数。闰年的判断方法：四年一闰，百年不闰，四百年又闰。

```
int year,month,day;
bool b=false;
cin>>year>>month>>day;
// 判断是否为闰年
if (n%400==0)
    b=true;
else if (n%100!=0 && n%4==0)
    b=true;

if (b)
    cout<<y<<"是闰年。"<<endl;
else
    cout<<y<<"不是闰年。"<<endl;

// 判断所在月份的天数
switch (month)
{
case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    cout<<"这个月有31天。"<<endl;
    break;
case 4: case 6: case 9: case 11:
    cout<<"这个月有30天。"<<endl;
    break;
case 2:
    cout<<"这个月有"<<(b ? 29 : 28)<<"天。"<<endl;
    break;
};
```

（3）循环结构

1．while 语句：如果条件成立，就继续循环，直到条件不成立为止。格式如下：

```
while (条件)
    循环体（语句或语句块）
```

2．do…while 语句：如果条件成立，就继续循环，直到条件不成立为止。它与 while 的最大区别在于，do…while 循环中的语句会被执行至少一次，而 while 中的语句可能一次都没有被执行。格式如下：

```
do
{
    循环体
}
while (条件);                          // 注意分号
```

4．for 语句：for 分四部分，有初始条件、继续循环的条件、状态转移的条件和循环体。格式如下：

```
for (初始条件; 继续循环的条件; 状态转移的条件)
    循环体
```

转换成 while 循环，即：

```
初始条件
while (继续循环的条件)
{
    循环体
    状态转移
}
```

for 后三个条件不是必需的，可以省略不写，但分号必须保留。

5．在循环语句内部使用 break，可以跳出循环；使用 continue，可以忽略它后面的代码，马上进入下一轮循环。

注意，这两个语句只对它所在的一层循环有效。

6．写 for 循环时，一定要注意：

- 不要把计数器的字母打错，尤其是在复制/粘贴一段代码的时候。
- 根据算法要明确不等号是“<”、“>”，还是“<=”、“>=”。
- 逆序循环时，不要把自减“--”写成自增“++”！

【问题描述】输入 n，输出 $n!$（$n!=1\times2\times3\times4\times\cdots\cdots\times n$）。结果保证小于 long long 的范围。当输入值为负数时结束程序。

```
int n;
long long r=1;
cin>>n;
while (n>-1)
{
    r=1;
    for (int i=1; i<=n; i++)
        r*=i;
    cout<<n<<"! = "<<r<<endl;
    cin>>n;
}
```

(4) goto 语句

goto 语句用于无条件跳转。要想跳转，首先要定义标签（**在代码开头的一段标识符，后面紧跟冒号**），然后才能 goto 那个标签。

很多教程不提倡使用无条件跳转，因为它破坏了程序结构，还容易给代码阅读者带来麻烦。不过，这不代表 goto 没有使用价值。goto 的一个用途是跳出多层循环：

```
for (int i=0; i<9; i++)
    for (int j=0; j<9; j++)
        for (int k=0; k<9; k++)
        {
            if (满足某种条件) goto __exited;
            ……
        }
__exited:
```

(5) C 与 C++的区别

C++语言是以 C 语言为基础开发出来的，C 语言的大多数内容被保留了下来。在信息学竞赛领域，很多情况下 C 和 C++可以互相转化，甚至不用对代码进行任何修改。

下面是信息学竞赛领域中 C 和 C++的重要区别：

- C++支持用流输入输出，而 C 只能用 scanf 和 printf——再见了，%d!
- C++非常支持面向对象编程，而 C 已经“out”了。
  《资料》中的“高精度算法”就只能用 C++完成，因为在 struct 内定义了成员函数。
  C++可以用更强大的 string 类处理字符串，而不必担心发生某些低级错误。
- C++有强大的 STL，而 C 没有（**有一个小小的 qsort 和 bsearch 算是补偿了**）。
  STL 是很多人从 C 转到 C++的重要原因。
- C 的头文件名仍然可以用在 C++中，不过可能会收到警报——应该去掉“.h”，前面再加一个“c”。
  如<stdio.h>应该改成<cstdio>。
- C 程序运行速度稍优于 C++。不过也没快多少。

总之，C 能做的一切事情，C++也能做；C++能做的一切事情，C 不一定能做。

1.2　数据类型

(1) 基本数据类型

名称	占用空间	别名	数据范围
int	4	signed, signed int, long, long int	-2,147,483,648～2,147,483,647
unsigned int[①]	4	unsigned, unsigned long,unsigned long int	0～4,294,967,295
char	1	char	-128～127
unsigned char	1	unsigned char	0～255
short[②]	2	short int,	-32,768～32,767

① 一般都使用有符号整数，除非范围不够大，或者你确定你的减法运算不会出现“小数减大数”。
② 一般来说，使用 int、long long 更保险一些，除非内存不够用。

		signed short int	
unsigned short	2	unsigned short int	0～65,535
long long[①]	8	signed long long	-9,223,372,036,854,775,808～9,223,372,036,854,775,807[②]
unsigned long long	8		0～18,446,744,073,709,551,615
bool	1		true 或 false
char	1		-128～127
signed char	1		-128～127
unsigned char	1		0～255
float	4		3.4E +/- 38 (7 位有效数字)
double	8	long double	1.7E +/- 308 (15 位有效数字)

(2) 变量与常量

1. 定义变量："变量类型 标识符"，如"int i;"定义了一个名字为 i 的整型变量。
   注意，此时 i 并未初始化，所以 i 的值是不确定的。
2. 定义常量："const 变量类型 标识符=初始值"，如：const int N=90;
3. 合法的标识符：
   - 标识符不能和关键字（**在 IDE 中会变色的词语**）相同。
   - 标识符只能包括字母、数字和下划线"_"，并且开头只能是字母或下划线。
   - 标识符必须先定义后使用。
   - 在同一作用域内，标识符不能重复定义（**即使在不同作用域内也不应重复，否则容易产生歧义**）。
   - C++区分大小写。所以 A 和 a 是两个不同的标识符。

(3) 数组

1. 定义一个一维数组：int a[10];
   这个数组一共 10 个元素，下标分别为 0～9。访问某个元素时，直接用 a 加方括号，如 a[5]。
2. 定义一个二维数组：int b[5][3];
   这个数组一共 5×3＝15 个元素，分别是 b[0][0]、b[0][1]、b[0][2]、b[1][0]……b[4][2]。
   访问某个元素时要用两个方括号，如 b[2][1]。
   多维数组的定义和使用方法与此类似。
3. 数组名和元素的寻址：以上面的 a、b 为例
   - 数组名是一个指针，指向整个数组第一个元素所在的地址。如 a 就是&a[0]、b 就是&b[0][0]。
   - 多维数组的本质是数组的数组，所以 b[0]实际上是 b[0][0]、b[0][1]……的数组名，b[0]就是&b[0][0]。
   - 在内存中，数组中每个元素都是紧挨着的，所以可以直接进行指针的运算。如 a+3 就是&a[3]，**(b+1)就是 b[1][0]，*(*(b+3)+2)就是 b[3][2]。
   - 在竞赛中要尽可能回避这些功能。
4. 字符串：
   - 字符串实际上是 char 的数组。
   - 字符串最后一位必须是'\0'，否则会在进行输出、使用字符串函数时发生意外。

---

① **不要使用"__int64"！**它是 Visual C++特有的关键字。

② 假如 a 是 long long 类型，把超过 2^{31} 的值赋给它时要使用字面值 LL（ULL）：a=123456789012345**LL**。

- 数组，包括字符串，不可以整体地赋值和比较。如果需要，应使用 memcpy 和 memcmp（**字符串是** strcpy **和** strcmp）。

5. C++中数组的下标只能从 0 开始（**当然可以闲置不用**），并且 **int a[10]中 a 的最后一个元素是 a[9]，不是 a[10]！**
6. **C++不检查数组下标是否越界**！如果下标越界，程序很有可能会崩溃！

(4) 指针

1. 取地址运算符和取值运算符：
   - 取地址运算符“&”：返回变量所在的地址。一般用于变量。（**而数组名本身就是指针，无需**“&”）
   - 取值运算符“*”：返回地址对应的值，或用于改变指针所指内存空间的值。只能用于指针。
2. 指针的意义：保存另一个变量的内存地址。
3. 定义指针：int *p;
   定义多个指针时，每个字母的前面都要有“*”。
   注意，如果 p 没有被初始化，它就会指向一个未知的内存空间，而错误地操作内存会导致程序崩溃！
4. 指针使用实例：

```
int a = 0, b = 1; int c[] = {1,2,3,4,5,6,7,8,9,10};
int *p;                      // 定义一个指针
p=&a;                        // 让p指向a
(*p)=3;                      // 相当于a=3
(*p)=b;                      // 相当于a=b，此时a等于1
// p=b;                      // 非法操作，左边是int *，右边是int，类型不匹配。
p=&b;                        // 让p指向b，从此p和a没关系了
p=c+6;                       // 让p指向c[6]，p和b又没关系了
cout<<*p;                    // 输出p指向的变量的值，即c[6]
p++;                         // 现在p指向了c[7];
p=NULL;                      // 表示p没有指向任何变量
cout<<*p;                    // 由于NULL（0）是一段无意义的地址，所以程序极有可能崩溃。
```

为了不在竞赛中把自己搞晕，请回避指针，对其敬而远之。

(5) 引用

通俗地讲，引用是某个变量的别名。下面定义了一个叫 p 的引用，它实际上是 f[0][2]。无论是改变 p 的值，还是改变 f[0][2]的值，结果都是一样的。

```
int &p = f[0][2];
```

使用引用的好处是，在函数的形参中定义引用类型，可以直接修改变量的值，而不用考虑“&”和“*”运算符。像上面一行代码一样，如果频繁调用 f[0][2]，也可以用引用节省篇幅，提高代码可读性。

引用与指针不同。引用被创建的同时也必须被初始化，并且必须与合法的存储单元关联。一旦引用被初始化，就不能改变引用的关系。而指针可以不立刻初始化，也可以改变所指向的内存空间。

(6) 结构体

- 结构体用 struct 定义。例如下面代码定义了一个叫 pack 的结构体，它有两个成员，一个叫 value，另一个叫 weight。

```
struct pack
{
    int value, weight;
};
```

- 变量可以定义成上面的 pack 类型：pack p;　　// 不必写成 struct pack p;
- 访问 pack 的成员时，用“.”运算符（**指针变量用**“->”）：p.value、(&p)->value
- C++中结构体可以像类一样建立自己的构造函数、成员函数，也可以重载运算符。
- 对于 pack 这个结构体，它的内部不允许再有 pack 类型的成员，但是可以有 pack 类型的指针。

1.3 运算符

(1) 运算符的优先级

运算符	结合方式
::	无
.（对象成员） ->（指针） []（数组下标） ()（函数调用）	从左向右
++ -- (*typecast*)（强制类型转换） sizeof ~ ! +（一元） -（一元） *（取值运算符） &（取地址运算符） new delete	从右向左
.* ->*	从左向右
* / %（取余数）	从左向右
+ -	从左向右
<<（左移） >>（右移）	从左向右
< <= > >=	从左向右
==（判断相等） !=（判断不等）	从左向右
&（按位与）	从左向右
^（异或）	从左向右
|（按位或）	从左向右
&&（条件与）	从左向右
||（条件或）	从左向右
?:（条件运算符）	从右向左
= *= /= %= += -= &= ^= >>= <<=	从右向左
,	从左向右

(2) 常用运算符的作用

1. 算术运算符：+、-、*、/、%分别表示加、减、乘、除、取余。

   两个整数做除法时，如果除不开，程序将把小数部分直接截断（**不是四舍五入**）。即：整数/整数＝整数，浮点数/浮点数＝浮点数。

   学习过其他语言的同学要注意，C++中没有乘方运算符，“^”也不是乘方运算符。

2. 比较运算符：

   - >、>=、<、<=、==（**相等**）、!=（**不等**）用来判断不等关系，返回值是 true 或 false。

     小心，千万不要把“==”写成“=”！
   - **永远不要对浮点数使用==和!=运算符**！正确的做法是：

     ```
     const double eps = 0.000001;       // 自己控制精度
     ……
     if (d>=2-eps && d<=2+eps) ……       // 相当于 if (d==2)
     ```
   - 不应该判断一个变量的值是否等于 true。安全的做法是判断它是否不等于 false。

3. 位运算：
   - &、|、^分别表示按位与、按位或、按位异或，即两个操作数上每一个二进制位都要进行运算。
   - ~表示按位求反。
   - <<、>>表示二进制位的移动。当某一位数字移动到二进制位的外面之后，它就直接“消失”了。a<<n 相当于 $a\times 2^n$，a>>n 相当于 $a\div 2^n$。

4. 逻辑运算符：
   - &&、||、^分别表示与、或、异或。!表示非。
   - ?:是条件运算，它是 C++唯一一个三目运算符。它的格式如下：A ? B : C。

     如果 A 不为 false，那么返回 B，否则返回 C。

     可以将这个运算符理解为一个简化版的 if。
   - ||、&&、?:是短路运算符[①]。不要在这三种运算符内调用函数或赋值，以免发生难以查出的错误！

5. 比较运算符、位移运算符、逻辑运算符、条件运算符的优先级较低，所以使用时要多加括号，以防出错。

6. 自增（++）、自减（--）运算符：
   - 增量分别是 1 和-1。
   - 这两种运算符只能用于数值类型的变量，不能用于非数值类型变量、常量、表达式和函数调用上。
   - 前缀++、--和后缀++、--的作用效果不同：

     ```
     int i=0, j=8, k=5;
     j = j + (++i);       // i 先自增，变成 1，然后再和 j 相加。执行之后 i=1，j=9。
     k = k + (i++);       // i 先和 k 相加，使 k=6。然后 i 再自增。执行之后 i=2，k=6。
     ```
   - 前缀运算符返回引用类型，后缀运算符返回数值类型。
   - 为了避免错误，不要让++、--和其他能够改变变量的操作在同一行出现！

7. 赋值运算符：
   - 在 C++中赋值是一种运算符。所以你会看到 i=j=0、d[x=y]、return c=i+j 之类的代码。
   - +=、-=、*=、……可以简化书写。例如 a*=2+3 相当于 a=a*(2+3)。

(3) 真值表

p	q	p && q (p & q)	p || q (p | q)	p ^ q	!p (~p)
true (1)	true (1)	true (1)	true (1)	false (0)	false (0)

[①] 例如计算“a && b”，如果 a 为 false，那么实际上结果就是 false——不管 b 是什么，程序都不再计算了。

true (1)	false (0)	false (0)	true (1)	true (1)	false (0)
false (0)	true (1)	false (0)	true (1)	true (1)	true (1)
false (0)	false (0)	false (0)	false (0)	false (0)	true (1)

(4) 类型强制转换

用一对小括号把数据类型包上，则它右侧的变量或常量的**值**（变量本身不变）就变成了对应的类型。如：

```
int i=2;
float c=6.4/(float)i;                    // 把i的值变成float类型。
```

两个操作数进行四则运算，如果想保留小数位，那么两个操作数应该都是浮点数。上面的代码就是这样。

1.4　函数

(1) 定义和使用函数

1. 定义和调用函数：下面定义了一个函数，返回值是 double 类型的，其中有两个参数 i、j，分别是 int 和 float 类型的。

```
double foo(int j, float j)
{
    ……
}
```

   - 如果函数不需要返回任何值，可定义为 void 类型。
   - 函数的定义必须在函数调用的前面。只有在前面添加了函数定义，才能把具体实现放到调用的后面：
     `double foo(int, float);          // 放到调用之前`
2. 返回值：return 值;
   - 函数是 void 类型的，那么 return 后面除了分号，什么都不跟。
   - 调用之后，函数立刻结束。
   - 不可以直接对函数名赋值（**学过 Pascal 或 Basic 语言的同学要特别注意**）。
3. 如果你的函数需要返回指针或引用，你必须注意：不要引用函数内部的变量！因为函数一结束，函数内部的变量就烟消云散，不复存在了。正确做法是引用静态变量（static）或全局变量。
4. 内联函数（inline）：当一个函数内部只有寥寥几句时，如“华氏度变摄氏度”，可以考虑将其定义成内联函数，通知编译器省略函数入栈等操作，直接展开函数内容，以加快运行速度。

```
inline int FtoC(int f) { return (f-32)/9*5; }
```

(2) 传递实参

1. 按值传递：例如 int foo(int n)，在调用 foo 时，程序会把参数复制一份给 n。这样，对 n 的任何修改都不会反映到调用 foo 的参数上面。
   对于按值传递数组，一定要慎重。因为复制数组的元素要浪费很多时间。
2. 传递指针：例如 int foo(int *n)。对 n 的修改会反映到调用 foo 的参数上面。
   - 修改 n 的值时要注意，必须用取值运算符，否则改变的是 n 指向的内存空间[①]。
   - 此外，这种方法可以用于传递数组——调用时只需把数组名作为参数。这时不需要取值运算符。
3. 传递引用：例如 int foo(int &n)。

优点是既可以直接修改调用 foo 的参数，又不会带来指针的麻烦（**无需取值运算符**）。缺点是不能传入常

[①] 使用 const 可防止 n 指向的内存空间发生改变：int foo(**const** int *n)。这时再写 n=5 之类的语句会被报错。

数或表达式。

1.5　输入和输出！

(1) 使用标准输入/输出

头文件：<cstdio>

变量约定：FILE *fin, *fout;——fin、fout 分别代表输入文件和输出文件。把它们换成 stdin 和 stdout，就是从屏幕输入和从屏幕输出。“1.5 字符串操作”也使用了同样的变量。

1. **输出字符串或变量的值**：printf("格式字符串", ……);

   或 fprintf(fout, "格式字符串", ……);

   格式字符：“%”后连接一个字母。

字符	含义	字符	含义
d	整数[①]	e, E	用科学记数法表示的浮点数
u	无符号整数	f	浮点数
o	八进制整数	c	字符
x, X	十六进制整数（**小写、大写**）	s	字符串（**字符数组**）

   常见的修饰符：

   - %5d：5 位数，右对齐。不足 5 位用空格补齐，超过 5 位按实际位数输出。
   - %-5d：5 位数，左对齐。不足 5 位用空格补齐，超过 5 位按实际位数输出。
   - %05d：5 位数，右对齐。不足 5 位用'0'补齐，超过 5 位按实际位数输出。
   - %+d：无论是正数还是负数，都要把符号输出。
   - %.2f：保留 2 位小数。如果小数部分超过 2 位就四舍五入，否则用 0 补全。

1. **输入到变量**

   - **读取不含空白的内容**：scanf("格式字符串", &……);

     或 fscanf(fin, "格式字符串", &……);

     ① 格式字符和 printf 基本一致。

     ② **不要忘记“&”！printf 传的是值，scanf 传的是地址！**

     ③ scanf 和 fscanf 的返回值是：成功输入的变量个数。fscanf 返回 EOF，表示文件结束。

     ④ scanf 和 fscanf 忽略 TAB、空格、回车。遇到这些字符它们就停止读取。

   - **读取单个字符**：fgetc(fin);

     首先要判断它是否为 EOF（文件结束）。如果不是，就可以用强制类型转换变成 char。

     读取到行末时，要注意对换行符的处理。

   - Windows、Linux、Mac 的回车字符是不同的。Linux 是'\n'，Mac 是'\r'，Windows 下是两个字符——'\r'和'\n'。

(2) 使用流输入/输出

头文件：<iostream>

1. **输入到变量**：cin>>n;
2. **输出到屏幕上**：cout<<a;

   可以连续输入、输出，如 cin>>n>>m;　cout<<a<<','<<b<<endl;
3. **换行**：cout<<endl;
4. **格式化输出**：

[①] 在 Windows 下调试时，用“%I64d”输出 long long 类型的值。交卷时，**由于用 Linux 测试，要改成“%lld”**。

头文件：<iomanip>

- **右对齐，长度为 n，不足的部分用空格补齐：**

  ```
  cout.width(n);
  cout.fill(' ');              // 如果想用“0”补齐，就可以把空格换成“0”
  cout<<a;
  ```

  前两行代码，每次输出之前都要调用。

- **输出成其他进位制数：**

  8： cout<<oct<<a;

  16： cout<<hex<<a;

  其他进位制需要自己转换。

5. **注意，数据规模很大时，流的输入输出速度会变得很慢，甚至数据还没读完就已经超时了。**

   在进行输入输出之前加入这样一条语句：ios::sync_with_stdio(false);

   调用之后，用 cin、cout 输入输出的速度就和 scanf、printf 的速度一样了。

1.6 其他常用操作！

本资料常用的头文件：<iostream>、<cstdlib>、<cstring>、<fstream>以及<algorithm>、<stack>、<queue>、<vector>等。

C++的流、容器、算法等都需要引用 std 命名空间。所以需要在#include 后面、你的代码前面加上一句：using namespace std;

(1) 库函数

1. **数组的整体操作：**

   头文件：<cstring>

   - 将 a[]初始化：memset(a, 0, sizeof(a));

     第二个参数应该传入 0、-1 或 0x7F。传入 0 或-1 时，a[]中每个元素的值都是 0 或-1；如果传入 0x7F 时，那么 a[]中每个元素的值都是 0x7F7F7F7F（不是 0x7F！），可认为是“无穷大”。

   - 将 a[]整体复制到 b[]中：memcpy(b, a, sizeof(a));
   - 判断 a[]和 b[]是否等价：memcmp(a, b, sizeof(a)); // 返回 0 表示等价

2. **字符操作：**

   头文件：<cctype>

   - tolower(c)、toupper(c)：将 c 转化为小写或大写。
   - isdight(c)、isalpha(c)、isupper(c)、islower(c)、isgraph(c)、isalnum(c)：分别判断 c 是否为十进制数字、英文字母、大写英文字母、小写英文字母、非空格、字母或数字。

3. **最大值/最小值：**

   头文件：<algorithm>

   max(a,b)返回 a 和 b 中的最小值，min(a,b)返回 a 和 b 中的最大值。

   其实我们可以自己写：

4. **交换变量的值：**swap(a,b)

   头文件：<algorithm>

   其实我们可以自己写：inline void swap(int &a, int &b) { int t=a; a=b; b=t; }

5. **执行 DOS 命令或其他程序：**system("命令行");

   - 头文件：<cstdlib>
   - 暂停屏幕：system("pause");
   - 竞赛交卷或 OJ 提交代码之前必须删除 system，否则会被视为作弊（**如果是 tyvj 甚至都无法提交**）。
   - 如果使用输入重定向，那么命令提示符不会接受任何键盘输入——直接用文件内容代替键盘了。

6. **立刻退出程序：**exit(0);

这种方法常用于深度优先搜索。执行后，程序立刻停止并返回 0，所以在调用前应该输出计算结果。

头文件：<cstdlib>

7. **计时**：double a = (double)clock() / (double)CLOCKS_PER_SEC;

上面的 a 对应一个时刻。而将两个时刻相减，就是时间间隔。可用这种方法卡时。

头文件：<ctime>

8. **断言**：assert(条件)

- 条件为假时，程序立刻崩溃。
- 头文件：<cassert>
- 如果定义了 NDEBUG 符号，那么它将不会起任何作用。
- 断言和错误处理不同：例如出现“人数为负数”的情况，如果责任在于用户，那么应该提示错误并重新输入，而不是用断言；如果发生在计算过程，应该用断言来使程序崩溃，以帮助改正代码中的错误。换句话说，错误处理防的是用户的错误，断言防的是代码的错误。

9. **快速排序**：qsort(首项的指针，待排序元素个数，每个元素所占字节，比较函数)

- 头文件：<cstdlib>
- 这是留给 C 党的快速排序，它比 STL 的排序算法啰嗦一些。
- 比较函数返回 int 类型，用于对两个元素的比较。原型如下：

  int compare(const void *i, const void *j);

  如果*i＜*j，则应返回一个小于 0 的数；如果*i==*j 则应返回 0，否则返回一个大于 0 的数。

10. **随机数发生器**：

- 头文件：<cstdlib>
- 产生随机数：
  ① 0~32767 的随机数：rand()
  ② 粗略地控制范围：rand()%范围
  注意，这种方法产生的随机数的分布是不均匀的。
  ③ 精确地控制范围：(double)rand()/RAND_MAX*范围
  ④ 控制在[a, b)之间：a + (int) ((double)rand()/RAND_MAX*(b-a))
- 初始化随机数种子：
  ① srand(数字)：初始化随机数种子。
  ② 注意，这条语句在程序开头使用，并且最多用一次。同一程序、同一平台，srand 中的参数相等，用 rand()产生的随机数序列相同。
  ③ 使随机数更加随机：引用<ctime>，然后这样初始化随机数种子，srand(time(NULL))。
  不要在循环中使用这条语句（例如批量产生随机数据），因为 time 只精确到秒。

11. **数学函数**：

- 头文件：<cmath>
- abs(x)：求 x 的绝对值（该函数同时包含于<cstdlib>）。
- sin、cos、tan、asin、acos、atan：三角函数，角的单位为弧度。
  可用 atan(1)*4 表示 π。
- sinh、cosh、tanh、asinh、acosh、atanh：双曲函数
- sqrt：求平方根
- ceil(x)、floor(x)：分别返回大于等于 x 的最小整数、小于等于 x 的最大整数。注意，参数和返回值都是浮点数类型。
- exp(x)、log(x)、log10：分别求 e^x、$\ln x$、$\lg x$

  （**顺便提一句，指数可以把加法问题转化为乘法问题，对数可以把乘法问题转化为加法问题。**）

- pow(a,b)：计算 a^b。由于精度问题，你仍然需要学会快速幂。
- fmod(a,b)：计算 a 除以 b 的余数。当然，这是浮点数的版本。

(2) 宏定义

宏定义是 C 语言的产物。在 C++中，它真的 out 了。

1. **第一种用法——配合条件编译：**#define DEBUG

定义一个叫 DEBUG 的标识符。它应该与#ifdef 或#ifndef 配合使用。举例如下：

```
#define DEBUG
#ifdef DEBUG
   void print(int v) { cout << v << endl;}
#else
   void print(int) {}
#endif
```

如果符号 DEBUG 存在，那么编译器会编译上面的、能输出数值的 print，否则编译器编译下面的、什么事情都不做的 print。

把上面的#ifdef 换成#ifndef，那么编译的代码正好上面所说的相反。

2. **第二种用法——表达式：**

```
#define N     5000
```

编译时，编译器会用类似于“查找和替换”的方法，把代码中的 N 换成 5000。如果需要换成表达式，应该用括号把它们包围。例如：

```
#define a        1+2
#define b        (1+2)
c=a*2; d=b*2;
```

编译时上面一行会变成“c=1+2*2; d=(1+2)*1;”，显然它们的值是不同的。

所以，用 enum 和 const 代替它是明智之举。

此外，要注意表达式末尾不能有分号（**除非你需要**）。

3. **第三种用法——简易“函数”：**

```
#define FtoC(a) (((a)-32)/9*5)
```

这类似于一个函数。不过，由于编译器只是进行简单替换，所以为了安全，a、b 应该用括号包围，整个表达式也应该用括号包围。

这种“函数”用法和普通函数一样，且速度更快。然而，它很容易出现难以查出的错误。所以，请用内联函数（inline）代替宏定义。

注意，**不要在“参数”中改变变量的值！**

4. **第四种用法——简化一段代码：**

```
#define move(dx, dy)    if (isfull(dir)) return;                    \
                        if (map(x+dx, y+dy)=='0')                   \
                        {                                           \
                           push(dir,x+dx,y+dy,head[dir], dep);      \
                           check(dir);                              \
                        }
```

不要忘记每行后面的“\”，它相当于换行符。这次 move 简化了一大段代码。同样，内联函数也可以。

1.7　字符串操作！

头文件：<cstring>。printf 和 scanf 在<cstdio>中，cin 和 cout 在头文件<iostream>中且位于 std 命名空间内。

下面假设待处理的字符串为 str 和 str2，即：char str[MAX], str2[MAX];

牢记，**字符串的最后一个字符一定是'\0'**。如果字符串内没有'\0'，进行以下操作（**输入除外**）时可能

会造成意外事故。

1. **输出字符串 str：**
   - cout<<str;
   - printf("%s",str); // 输出到文件：fprintf(fout, "%s", str);
2. **输入字符串 str：**
   - scanf("%s", str); // 输出到文件：fscanf(fin, "%s", str);
   - cin>>str;
     以上两种方法在输入时会忽略空格、回车、TAB 等字符，并且在一个或多个非空格字符后面输入空格时，会终止输入。
   - fgets(str, MAX, fin);

     每调用一次，就会读取一行的内容（**即不断读取，直到遇到回车停止**）。
3. **求字符串 str 的长度：**strlen(str) // 这个长度不包括末尾的'\0'。
4. **把字符串 str2 连接到字符串 str 的末尾：**strcat(str, str2)
   - str 的空间必须足够大，能够容纳连接之后的结果。
   - 连接的结果直接保存到 str 里。函数返回值为&str[0]。
   - strncat(str, str2, n)是把 str2 的前 n 个字符连接到 str 的末尾。
5. **把字符串 str2 复制到字符串 str 中：**strcpy(str, str2)
6. **比较 str 和 str2 的大小：**strcmp(str, str2)
   如果 str＞str2，返回 1；如果 str==str2，返回 0；如果 str＜str2，返回-1。
7. **在 str 中寻找一个字符 c：**strchr(str, c)
   返回值是一个指针，表示 c 在 str 中的位置。用 strchr 的返回值减 str，就是具体的索引位置。
8. **在 str 中寻找 str2：**strstr(str, str2)
   - 返回值是一个指针，表示 str2 在 str 中的位置。用 strstr 的返回值减 str，就是具体的索引位置。
   - 此问题可以用 KMP 算法解决。KMP 算法很复杂，在 NOIP 范围内用途不大。
9. **从 str 中获取数据：**sscanf(str, "%d", &i);
   格式化字符串：sprintf(str, "%d", i);
   它们和 fscanf、fprintf 非常像，用法也类似。可以通过这两个函数进行数值与字符串之间的转换。

1.8 文件操作!

正式竞赛时，数据都从扩展名为“in”的文件读入，并且需要你把结果输出到扩展名为“out”的文件中；在 OJ（Online Judge，在线测评器）中则不需要文件操作。具体情况要仔细查看题目说明，以免发生悲剧。

(1) 输入/输出重定向

头文件：<fstream>或<cstdio>

- 方法：只需在操作文件之前添加以下两行代码。

```
freopen("XXXXX.in","r",stdin);
freopen("XXXXX.out","w",stdout);
```

- 调用两次 freopen 后，scanf、printf、cin、cout 的用法完全不变，只是操作对象由屏幕变成了指定的文件。
- 使用输入重定向之后，“命令提示符”窗口将不再接受任何键盘输入（**调用 system 时也是如此**），直到程序退出。这时不能再用 system("pause")暂停屏幕。

(2) 文件流

头文件：<fstream>

流的速度比较慢，在输入/输出大量数据的时候，要使用其他文件操作方法。

- 方法：定义两个全局变量。

```
ifstream fin("XXXXX.in");
ofstream fout("XXXXX.out");
```

- fin、fout 和 cin、cout 一样，也用“<<”、“>>”运算符输入/输出，如：fin>>a; fout<<b;

  当然，也可以通过#define 把 fin、fout 变成 cin、cout：

```
#define cin fin
#define cout fout
```

(3) FILE 指针

头文件：<cstdio>或<fstream>

- 方法：定义两个指针。

```
FILE *fin, *fout;
……
int main()
{
    fin = fopen("XXXXX.in", "r");
    fout = fopen("XXXXX.out", "w");
    ......
    fclose(fin); fclose(fout);     // 在某些情况下，忘记关闭文件会被认为是没有产生文件。
    return 0;
}
```

- 进行输入/输出操作时，要注意**函数名的前面有 f**，即 fprintf、fscanf、fgets，并且这些函数的第一个参数不是格式字符串，而是 fin 或 fout，如 fprintf(**fout,** "%d", ans)。
- 想改成从屏幕上输入/输出时，不用对代码动手术，只需把含 fopen 和 fclose 的代码注释掉，并改成：fin=stdin; fout=stdout;

1.9　简单的算法分析和优化

(1) 复杂度

为了描述一个算法的优劣，我们引入算法时间复杂度和空间复杂度的概念。

时间复杂度：一个算法主要运算的次数，用大 O 表示。通常表示时间复杂度时，我们只保留数量级最大的项，并忽略该项的系数。

例如某算法，赋值做了 $3n^3+n^2+8$ 次，则认为它的时间复杂度为 $O(n^3)$；另一算法的主要运算是比较，做了 $4\times2^n+2n^4+700$ 次，则认为它的时间复杂度为 $O(2^n)$。

当然，如果有多个字母对算法的运行时间产生很大影响，就把它们都写进表达式。如对 $m\times n$ 的数组遍历的时间复杂度可以写作 $O(mn)$。

空间复杂度：一个算法主要占用的内存空间，也用大 O 表示。

在实际应用时，空间的占用是需要特别注意的问题。太大的数组经常是开不出来的，即使开出来了，遍历的时间消耗也是惊人的。

(2) 常用算法的时空复杂度

1s 运算次数约为 5,000,000[①]。也就是说，如果把 n 代入复杂度的表达式，得数接近或大于 5,000,000，那么会有超时的危险。

常见的数量级大小：$O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)<O(2^n)<O(n!)$

数量级	能承受的大致规模	常见算法
$O(1)$	任意	直接输出结果
$O(\log n)$	任意	二分查找、快速幂
$O(n)$	以百万计（**五六百万**）	贪心算法、扫描和遍历
$O(n\log n)$	以十万计（**三四十万**）	带有分治思想的算法，如二分法
$O(n^2)$	以千计数（**两千**）	枚举、动态规划
$O(n^3)$	不到两百	动态规划
$O(2^n)$	24	搜索
$O(n!)$	10	产生全排列
$O(n^n)$	8	暴力法破解密码

$O(1)$叫常数时间；$O(n)$、$O(n^2)$、$O(n^3)$、$O(n^4)$……叫做多项式时间；$O(2^n)$、$O(3^n)$……叫做指数时间。

(3) 简单的优化方法

1．时间的简单优化方法

时间上的优化在于少做运算、做耗时短的运算等。有几个规律需要注意：

- 整型运算耗时远低于实型运算耗时。
- 位运算速度极快。
- 逻辑运算比四则运算快。
- +、-、*运算都比较快（-、*比+慢一点点，可以忽略不计）。
- /运算比+、-、*慢得多（甚至慢几十倍）。
- 取余%和除法运算速度相当。
- 调用函数要比直接计算慢（因为要入栈和出栈）。

这些规律我们可以从宏观上把握。事实上，究竟做了几步运算、几次赋值、变量在内存还是缓存等多数由编译器、系统决定。但是，少做运算（尤其在循环体、递归体中）一定能很大程度节省时间。

2．空间的简单优化方法

空间上的优化主要在于减小数组大小、降低数组维数等。常用的节省内存的方法有：

压缩储存——见 192 页“A.7　状态压缩*”。

覆盖旧数据——例如滚动数组（见 66 页“(5) 使用滚动数组”）。

要注意的是，对空间的优化即使不改变复杂度，只是改变 n 的系数也是极有意义的。空间复杂度有时对时间也有影响，要想方设法进行优化。

[①] 不同资料给出的数值是不同的，不过这不要紧。在 NOIP 中，只要你的算法正确，就不会在运行时间上“打擦边球”。

3．优化的原则

- 尽量不让程序做已做过的事。
- 不让程序做显然没有必要的事。
- 不要解决无用的子问题。
- 不要对结果进行无意义的引用。

1.10　代码编辑器

为了学习信息学，你需要一台带有 C++编译器的计算机。

为了方便编程，你还需要 IDE（集成开发环境）或具有编程特性的代码编辑器，否则你需要自己从命令行编译程序。

常见的 IDE 和编辑器如下：

名称	适用操作系统	代码编辑功能	编译器	调试功能	单文件编译
DEV-C++	Windows	一般	自备	差	√
Code::Blocks	Linux、Windows	好	×	好	√（**调试除外**）
Anjuta C/C++ IDE	Linux	好	×	好	×
GUIDE	Windows、Linux	一般	×	一般	√
Emacs（Linux **自带**）	Linux、Windows	好	×	好	√
Vim（Linux **自带**）	Linux、Windows	好	×	gdb**	g++***
Eclipse****	Windows、Linux	好	×	好	×
记事本＋命令提示符	Windows	×	×	gdb	g++
gedit＋终端	Linux	差	×	gdb	g++
Visual C++	Windows	好	自备*	好	×
Qt Creator	Windows、Linux	好	自备	好	×

* 不是 GCC，而是微软自己的编译器。

** gdb 调试功能的好坏取决于你使用的熟练程度。

*** 会用就能编译。

**** 需要安装 CDT 插件才能编写 C++的程序。

Windows 不预备 C++编译器，你需要自己下载并安装，下载地址 www.mingw.org。Linux 自备 GCC，无需再安装编译器。

第二单元　基础算法

2.1　经典枚举问题

(1) 韩信点兵

【问题描述】正整数 x 除以 3 余 1，除以 5 余 2，除以 7 余 3。问符合条件的最小 x。

```
// 当x比较大时，更合适的做法是解同余式组，具体做法见145页。
int x;
for (x=1; ;x++)
    if (x%3==1  &&  x%5==2  &&  x%7==3) break;
cout<<x;
```

(2) 百鸡问题

【问题描述】1 只公鸡 5 文钱，1 只母鸡 3 文钱，3 只小鸡 1 文钱。如果正好花 100 文钱，可以买几只公鸡、几只母鸡、几只小鸡？

```
for (int x=0; x<=20; x++)                    // 公鸡
    for (int y=0; y<=33; y++)                // 母鸡
    {
        int z=3*(100-5*x-3*y);               // 小鸡
        if (z>=0) cout<<x<<" "<<y<<" "<<z<<endl;
    }
```

(3) 求水仙花数

【问题描述】请输出所有的水仙花数。水仙花数是一个三位数，每一位数字的立方相加，得到的总和是它本身，如 $143=1^3+4^3+3^3$。

```
for (int i=1; i<=9; i++)
    for (int j=0; j<=9; j++)
        for (int k=0; k<=9; k++)
        {
            int p=i*100+j*10+k, q=i*i*i + j*j*j + k*k*k;
            if (p==q) cout<<p<<endl;
        }
```

(4) 砝码称重

【问题描述】给你若干个 1g、2g、5g、10g、20g、50g 的砝码，数量分别为 a[1]、a[2]……a[6]，问用这些砝码可以称量出多少种重量（**重量大于** 0，**小于等于** 1000）。

```
int count=0, weight[1001], a[7];
memset(weight, 0, sizeof(weight));
```

```
for (int x1=0; x1<=a[1]; x1++)
    for (int x2=0; x2<=a[2]; x2++)
        for (int x3=0; x3<=a[3]; x3++)
            for (int x4=0; x4<=a[4]; x4++)
                for (int x5=0; x5<=a[5]; x5++)
                    for (int x6=0; x6<=a[6]; x6++)
                    {
                        int w=1*x1+2*x2+5*x3+10*x4+20*x5+50*x6;
                        weight[w]++;
                    }

for (int i=1; i<=1000; i++)
    if (weight[i]>0) count++;
cout<<count;
```

上面的代码看起来似乎很“笨拙”，但实际上它已经能顺利地解决问题了。

2.2　火柴棒等式[①]

【问题简述】给你 n（$n \leqslant 24$）根火柴棍，你可以拼出多少个形如“$A+B=C$”的等式？等式中的 A、B、C 是用火柴棍拼出的整数（**若该数非零，则最高位不能是** 0），数字的形状和电子表上的一样。

注意：

1．加号与等号各自需要两根火柴棍。

2．如果 $A \neq B$，则 $A+B=C$ 与 $B+A=C$ 视为不同的等式（A，B，$C \geqslant 0$）。

3．n 根火柴棍必须全部用上。

【分析】

直接枚举 A 和 B（**事实证明只到 3 位数**）。为了加快速度，事先把 222 以内各个数所用的火柴数算出来。

```
#include <iostream>
using namespace std;

int matches[223], n;
void getmatches();          // 把火柴棍事先算好
int count=0;

int main()
{
    getmatches();

    cin>>n;
    for (int i=0;i<=111;i++)
        for (int j=0;j<=111;j++)
        {
            int k=i+j;
            if(matches[i]+matches[j]+matches[k]+4==n)
```

[①] 题目来源：NOIP2008 第二题

```
            count++;
        }
    cout<<count;
    return 0;
}

void getmatches()
{
    matches[0]=6;
    matches[1]=2;
    matches[2]=5;
    ……                          // 事先编一个程序来产生这一部分代码。
    matches[222]=15;
}
```

2.3　梵塔问题

【问题描述】已知有三根针分别用 1、2、3 表示。在一号针中从小到大放 n 个盘子，现要求把所有的盘子从 1 针全部移到 3 针。移动规则是：使用 2 针作为过渡针，每次只移动一块盘子，且每根针上不能出现大盘压小盘，找出移动次数最小的方案。

【分析】

这是一个经典的递归问题。

递归的思路：如果想把 n 个盘子放到 3 针上，就应该先把 n-1 个盘子放到 2 针上。然后把 1 针最底部的盘子移动到 3 针，再把 2 针上的 n-1 个盘子移动到 3 针上。

```
void move(int n, int a, int b, int c)          // a是盘子来源，b是暂存区、c是目标
{
    if (n==1)
        cout<<a<<"->"<<c<<endl;                // 只有一根针时直接移动
    else
    {
        move(n-1,a,c,b);                       // 先把n-1个盘子移动到#2上
        cout<<a<<"->"<<c<<endl;
        move(n-1,b,a,c);                       // 把n-1个盘子移动到#3上
    }
}
```

调用：move(n,1,2,3);

时间复杂度：O(2^n)。n 层梵塔至少要移动 2^n-1 步。

2.4　斐波那契数列

斐波那契数列为 1，1，2，3，5，8，……

很明显，斐波那契数的递推公式是 $f(n)=\begin{cases}1 & (n\leqslant 2)\\ f(n-2)+f(n-1) & (n>2)\end{cases}$

(1) 递归

```
int f(int n)
{
```

```
    if (n<=2)
        return 1;
    else
        return f(n-2)+f(n-1);
}
```

(2) 记忆化搜索

当 n 很大时，递归的运行速度会明显变慢。根本原因在于递归算法进行了大量的重复运算。所以可以开一个数组，记录每个被计算过的函数值。每次调用 f 时，如果函数值被计算过，就直接调用计算好的结果。

```
int F[MAX] = {0};                          // 初始化为0表明没有计算过
int f(int n)
{
    if (F[n]!=0) return F[n];              // 直接调用记录的值
    if (n<=2)
        return F[n]=1;
    else
        return F[n]=f(n-2)+f(n-1);         // 计算，同时记录计算结果
}
```

(3) for 循环

尽管记忆化快了很多，但仍然不是最好的。本问题递推顺序明显，用一个 for 循环就可以解决问题了。

```
int t1, t2, r;                  // int f[MAX] = {0, 1, 1};
t1=t2=r=1;
for (int i=3; i<=n; i++)
{
    r=t1+t2;                    // f[i]=f[i-2]+f[i-1];
    t1=t2, t2=r;                // 使用f后这一行代码就没有意义了。
}
// cout<<r;                     // cout<<f[n];
```

2.5　常见的递推关系！

1. **前序和**：设 S_i 表示数列中首项[①]到第 i 项的和。
   - S_i 的递推公式：$S_i=S_{i-1}+a_i$
   - 第 i 项到第 j 项中所有元素的和：$S=S_j-S_{i-1}$
2. **等差数列**
   - 递推公式：设首项为 a_0，公差为 d，则 $a_n=a_{n-1}+d$
   - 通项公式[②]：$a_n=a_0+nd$
   - 前 n 项和：$S_n=\frac{1}{2}(a_0+a_n)(n+1)=(n+1)a_0+\frac{1}{2}dn(n+1)$
3. **等比数列**
   - 递推公式：设首项为 a_0，公比为 q，则 $a_n=qa_{n-1}$
   - 通项公式：$a_n=a_0q^n$

[①] 在本节若无特殊说明，n 都是从 0 开始的，当然“n 条直线”、“n 个物品”是例外。
[②] 这里的通项公式、求和公式和高中数学的略有不同，原因是首项的下标不同。

- 前 n 项和：$S_n=\frac{a_0(1-q^{n+1})}{1-q}$

4．**斐波那契（Fibonacci）数列**

- 递推公式：$f_n=\begin{cases}1 & n=0\text{或}n=1\\ f_{n-1}+f_{n-2} & n\geqslant 2\end{cases}$
- 前 n 项和：$S_n=f_{n+2}-1$
- 通项公式（**不能用于编程！**）：$f_n=\frac{\sqrt{5}}{5}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right]$
- 实例：
  ① 楼梯有 n 阶台阶，上楼可以一步上 1 阶，也可以一步上 2 阶，计算共有多少种不同的走法。
  ② 有一对雌雄兔，每两个月就繁殖雌雄各一对兔子.问 n 个月后共有多少对兔子？
  ③ 有 $n*2$ 的一个长方形方格，用一个 1*2 的骨牌铺满方格。求铺法总数。

5．**第二类 Stirling 数**

- $s(n,k)$ 表示含 n 个元素的集合划分为 k 个集合的情况数。
- **递推公式**：$s(n,k)=s(n-1,k-1)+k\cdot s(n-1,k)$，$1\leqslant k<n$

6．**错位排列**

- 示例：在书架上放有编号为 1，2，…，n 的 n 本书。现将 n 本书全部取下然后再放回去，当放回去时要求每本书都不能放在原来的位置上。求满足以上条件的放法共有多少种？
- 错位排列数列为 0，1，2，9，44，265，…
- 第一种递推公式：$d_n=\begin{cases}0 & n=1\\ 1 & n=2\\ (n-1)(d_{n-2}+d_{n-1}) & n\geqslant 3\end{cases}$
- 第二种递推公式：$d_n=\begin{cases}0 & n=1\\ nd_{n-1}+(-1)^{n-2} & n\geqslant 2\end{cases}$
- 通项公式：$d_n=n!\left[\frac{1}{0!}-\frac{1}{1!}+\frac{1}{2!}-\frac{1}{3!}+\frac{1}{4!}-\cdots+(-1)^n\frac{1}{n!}\right]$

7．**分平面的最大区域数**

- n 条直线分平面的最大区域数的序列为：2，4，7，11，…
  递推公式：$f_n=f_{n-1}+n$
  通项公式：$f_n=n(n+1)/2+1$
- n 条折线分平面的最大区域数的序列为：2，7，16，29，…
  递推公式：$f_n=f_{n-1}+4n-3$
  通项公式：$f_n=(n-1)(2n-1)+2n$
- n 条封闭曲线（**如一般位置上的圆**）分平面的最大区域数的序列为：2，4，8，14，…

  递推公式：$f_n=f_{n-1}+2(n-1)$
  通项公式：$f_n=n^2-n+2$

8．**组合数**

- 通项公式：$C_n^m=\frac{A_n^m}{m!}=\frac{n(n-1)(n-2)\cdots\cdots(n-m+1)}{m!}=\frac{n!}{m!(n-m)!}$
- 第一种递推公式：$C_n^m=C_{n-1}^{m-1}+C_{n-1}^m$　（边界条件$C_n^0=C_n^n=1$）
- 第二种递推公式：$C_n^m=\frac{n-m+1}{m}C_n^{m-1}$　（边界条件$C_n^0=1$，必须先乘后除）

  优化：$m>\frac{n}{2}$时，可利用$C_n^m=C_n^{n-m}$来减少计算次数。

9．**卡塔兰（Catalan）数列**

- 序列：1，2，5，14，42，132，429，1430，4862，16796…

- 通项公式：$h_n=\frac{C_{2n}^{n}}{n+1}$
- 第一种递推公式（**基于公式意义**）：$h_n=\begin{cases}1 & n=1\\ h_1h_{n-1}+h_2h_{n-2}+\ldots+h_{n-1}h_1 & n\geqslant 2\end{cases}$
- 第二种递推公式（**基于组合数性质**）：$h_n=\begin{cases}1 & n=1\\ \frac{2(2n-1)}{n+1}h_{n-1} & n\geqslant 2\end{cases}$
- 实例：

① 有 $2n$ 个人排成一行进入剧场。入场费 5 元。其中只有 n 个人有一张 5 元钞票，另外 n 人只有 10 元钞票，剧院无其它钞票，问有多少中方法使得只要有 10 元的人买票，售票处就有 5 元的钞票找零？

② 一位大城市的律师在她住所以北 n 个街区和以东 n 个街区处工作。每天她走 $2n$ 个街区去上班。如果他从不穿越（**但可以碰到**）从家到办公室的对角线，那么有多少条可能的道路？

③ 在圆上选择 $2n$ 个点，将这些点成对连接起来使得所得到的 n 条线段不相交的方法数？

④ n 个结点可构造多少个不同的二叉树？

⑤ 一个栈（**无穷大**）的进栈序列为 1，2，3，…n，有多少个不同的出栈序列？

⑥ 将一个凸多边形区域分成三角形区域的方法数？

⑦ 一个乘法算式 $P=a_1a_2a_3\cdots a_n$，在保证表达式合法的前提下（**某个数不会被括号括两次**，**如**“((a))”**是错误的**），有多少种添加括号的方法？

2.6　选择客栈[①]

【问题描述】

丽江河边有 n 家（$2\leqslant n\leqslant 200,000$）很有特色的客栈，客栈按照其位置顺序从 1 到 n 编号。每家客栈都按照某一种色调进行装饰（总共 k 种，用整数 0~k-1 表示，且 $0<k\leqslant 50$），且每家客栈都设有一家咖啡店，每家咖啡店均有各自的最低消费。

两位游客一起去丽江旅游，他们喜欢相同的色调，又想尝试两个不同的客栈，因此决定分别住在色调相同的两家客栈中。晚上，他们打算选择一家咖啡店喝咖啡，要求咖啡店位于两人住的两家客栈之间（**包括他们住的客栈**），且咖啡店的最低消费不超过 p（$0\leqslant p$，最低消费$\leqslant 100$）。

他们想知道总共有多少种选择住宿的方案，保证晚上可以找到一家最低消费不超过 p 元的咖啡店小聚。

【分析】

首先考虑 30%（$n\leqslant 100$）的数据。很明显，本题需要用枚举法解决。

枚举法的思路很明显：枚举区间 $[i, j]$（$i\neq j$，且 i、j 色调相同），再判断区间内是否有最低消费不超过 p 的咖啡店。时间复杂度为 $O(n^3)$。

现在开始优化算法。可以看到，判断区间内咖啡店的时间为 $O(n)$，有减少的余地。令 $c[i]$ 表示 $[1, i]$ 之间最低消费不超过 p 的咖啡店个数[②]，那么可以看出，如果 $c[j]-c[i]>0$，说明 i、j 之间有符合要求的咖啡店。这样判断区间的时间就降到了 $O(1)$。

有这一步做基础，为了使思路清晰，就可以把不同颜色的客栈分开了。

在处理 j 时，我们要统计 $1\sim j-1$ 与 j 的解的个数。处理 $j+1$ 时又要处理 $1\sim j-1$，造成了重复计算。

[①] 题目来源：NOIP2011 day1 第二题

[②] 事实上不是咖啡店个数。如果 i、j 并列出现“√×”的情况，那么 $c[j]-c[i]$ 也应该大于 0。

设 sum[j]为[1，j]到[i，j]中解的个数。如果j＋1 无法消费，那么由于j 的存在，有 sum[j+1]＝sum[j]；如果j＋1 可以消费，那么j＋1 可以和 1 到j 中任何一个客栈组合，那么 sum[j+1]＝j。

最后将每种颜色的所有 sum 相加，就可以在 O(n)的时间内得出答案了。

```
#include <iostream>
#include <cstring>
using namespace std;

int n,k,p;
int c[51][200001], top[51];
long long total=0;

int main()
{
   memset(top,-1,sizeof(top));
   memset(c,0,sizeof(c));

   ios::sync_with_stdio(false);
   cin>>n>>k>>p;

   int cnt=0;
   bool P=false;
   for (int i=0; i<n; i++)
   {
      int x,y;
      cin>>x>>y;

      int t=++top[x];
      if (y<=p || P)
         cnt++;
      c[x][t]=cnt;

      P = y<=p;
   }

   for (int color=0; color<k; color++)
   {
      int prev=c[color][0];
      long long sum=0;

      for (int i=0; i<=top[color]; i++)
      {
         if (c[color][i]-prev>0) sum=i;
         total+=sum;
         prev=c[color][i];
      }
   }

   cout<<total<<endl;
```

```
    return 0;
}
```

2.7　2^k进制数[①]

【问题描述】

设 r 是个 2^k 进制数（$1 \leqslant k \leqslant 9$），并满足以下条件：

(1) r 至少是个 2 位的 2^k 进制数。

(2) 作为 2^k 进制数，除最后一位外，r 的每一位严格小于它右边相邻的一位。

(3) 将 r 转换为 2 进制数 q 后，则 q 的总位数不超过 w（$k < w \leqslant 30000$）。

输入 k 和 W，请满足上述条件的不同 r 的数量。

【分析】

本题不可能直接枚举——答案不超过 200 位，说明用到了高精度算法！

先不考虑 w。对于一个 m 位数来讲，m 个数字的顺序是确定的。所以只需直接从 n（$n=2^k-1$）个数中取 m 个数，组成 m 位数。所以符合条件的 m 位数有C_m^n个。

现在考虑 w 存在的情况。这里有一个明显的事实：2^k 进制数，除首位外，每一位转化为二进制后都是 k 个二进制位。现在最大的问题就是 2^k 进制数的首位。

二进制不超过 w 位的数，在 2^k 进制下就是 $a=[w/k]+1$ 位（w 能被 k 整除时，认为首位有 0 个数字）。此时首位剩下 $b=w\%k$ 个二进制位，即最大可以取 2^b-1。接下来只需针对 2^k 进制下位数不足 a 和恰好为 a 两种情况分别进行枚举。

所以，符合条件的 2^k 进制数个数为 $f(k,w)=\begin{cases}\sum_{i=2}^{n} C_n^i & w \geqslant kn \\ \sum_{i=2}^{a-1} C_n^i + \sum_{i=1}^{2^b-1} C_{n-i}^a & k < w < kn\end{cases}$

其中 $n=2^k-1$，$a=[w/k]+1$，$b=w \bmod k$。

通过排列组合，我们间接地枚举了所有符合条件的解。

2.8　Healthy Holsteins[②]

【问题描述】

农民约翰以他有世界上最健康的奶牛为骄傲。他知道每种饲料的各种维生素的含量，以及一头牛每天需要最少的维生素的量。请你帮助约翰喂养这些奶牛，使得它们能够保持健康，并且消耗的饲料种类最少。

【输入格式】

第一行：一个数 V（$1 \leqslant V \leqslant 25$），表示维生素的种类数。

第二行：V 个整数，表示一头牛一天需要的维生素量。一个整数对应一种维生素。

第三行：一个数 G（$1 \leqslant G \leqslant 15$），表示饲料的种类数。

第四行：G 个整数，表示每种饲料的各种维生素的含量。

【输出格式】

输出共一行。第一个数是需要的最少的饲料种类数。从第二个数开始，输出所需的饲料种类序号。请输出字典序最小的解。

【分析】

[①] 题目来源：NOIP2006 第四题

[②] 题目来源：USACO Training Gateway

本题最多只有 15 种饲料。如果尝试所有取法，也只有 $2^{15}=32768$ 种状态，因此完全可以枚举所有方案。

考虑一种饲料，它要么用来喂牛，要么扔到一边去。用来喂牛可以用“1”表示，放到一边用“0”表示。于是，G 种饲料的状态可以转化成一个 G 位的二进制数。这样，枚举 0 到 2^G-1 的数，就枚举了所有状态。

当然这只是一种思路。如果不能熟练使用位运算，这种做法的编程复杂度会比 DFS 高很多。

【代码】

```
#include <iostream>
using namespace std;

int v, req[26];
int g, vitamin[16][26];
int Min=100, minstat=0;        // Min表示需要最少饲料数，minstat表示目前的最优状态

int count(int stat)            // 统计stat中“1”的个数
{
    int r=0;
    while (stat)
    {
        r+=stat & 1;
        stat>>=1;
    }
    return r;
}

int main()
{
    cin>>v;
    for (int i=1; i<=v; i++) cin>>req[i];

    cin>>g;
    for (int i=1; i<=g; i++)
        for (int j=1; j<=v; j++)
            cin>>vitamin[i][j];

    for (int stat=0; stat < (1<<g); stat++)                // 尝试所有组合
    {
        int r[26] = {0};

        for (int i=1; i<=g; i++)                           // 计算组合中每种维生素的量
            if (stat & (1<<(i-1)))
                for (int j=1; j<=v; j++) r[j]+=vitamin[i][j];

        bool no=false;
        for (int i=1; i<=v; i++) if (r[i]<req[i]) no=true; // 判断维生素是否达标
        if (no) continue;

        int c=count(stat);
        if (c<Min && c>0) Min=c, minstat=stat;
```

```
    }

    cout<<Min<<" ";
    for (int i=1; i<=g; i++) if (minstat & (1<<(i-1))) cout<<i<<" ";
    return 0;
}
```

2.9　小结

本单元总结了三种基本算法：枚举、递归和递推。

1．枚举

所谓枚举，就是将所有的可能一一遍历出来，在遍历中逐个检查答案的正确性，从而确定最终结果。因此，枚举法适合解决“是否有解”和“有多少个解”的问题。搜索的本质就是枚举。

应用枚举法搜索答案之前，要确定枚举的对象和状态的表示方式。

常见的枚举方法有三种：第一种是使用 for 循环，第二种是利用排列组合，第三种是顺序化（**比如把状态与二进制数对应**）。

由于枚举法产生了大量的无用解，所以规模稍大的时候要进行优化。优化时要从两方面入手，一是减少状态量，二是降低单个状态的考察代价。优化过程要从以下几个方面考虑：枚举对象的选取、枚举方法的确定、采用局部枚举或引进其他算法。

（**在这里附上次短路的求法：先求出最短路，枚举最短路上的每一条边，对于每一条边都删除然后重新求最短路。几个新的“最短路”的最小值就是次短路。**）

2．递归

递归，即自己调用自己。递归法将较复杂的大问题分解成较简单的小问题，然后一层一层地得出最终答案。

递归是实现许多重要算法的基础。深度优先搜索、记忆化搜索、分治算法和输出动态规划的中间过程等都需要用递归实现。

有一些数据结构也是递归定义的，如二叉树。使用此类数据结构时，也要递归地思考问题。

写递归要从两方面入手：分解问题的方式、边界条件。

注意，没有边界条件会引发无限递归，从而导致程序崩溃！

3．递推

递推指利用数列中前面一项或几项，通过递推公式来求出当前项的解。

递推与递归不同。一般情况下，递推用循环语句完成，而递归必须定义函数。递推也可以转化为递归，即记忆化搜索，这常用于递推顺序不明显的情况。

解决递推问题要从以下几方面入手：状态表示、递推关系、边界条件。

按照计算顺序，递推可以分为顺退和逆推。对于某些问题，递推顺序甚至直接影响编码的复杂度。

动态规划必须使用递推或记忆化搜索来解决。动态规划的状态转移方程一定是递推式。但动态规划与纯粹的递推不同，前者有明显的阶段，而后者的数学性更强。

我们已经在数学课上学习了数列的通项公式和递推公式。高中数学喜欢通项公式，而信息学更喜欢递推公式。下次遇到类似问题，就去寻找递推式吧。

4．递归和递推的比较

	递归	递推

适合解决的问题	1．问题本身是递归定义的，或者问题所涉及到的数据结构是递归定义的，或者是问题的解决方法是递归形式的。 2．需要利用分治思想解决的问题。 3．回溯、深度优先搜索 4．输出动态规划的中间过程。	1．能够用递推式计算的数学题。 2．动态规划（**必须使用递推或记忆化搜索**）
特点	结构清晰、可读性强 目的性强	速度较快、比较灵活 思路不易想到
编码方式	在函数中调用自己	迭代（**使用** `for` **循环等**）
替代方法	问题的性质不同，改写的方法也不同。 ① 有的问题可以根据程序本身的特点，使用栈来模拟递归调用。 ② 有的问题可以改写成递推或迭代算法。	在拓扑关系不明显时，可以采用记忆化搜索。

第三单元　搜索

3.1　N 皇后问题

【问题描述】在国际象棋中，皇后可以攻击与她在同一条水平线、垂直线和对角线的棋子。现在有一张 $N \times N$ 的国际象棋棋盘，在上面放置 N 个皇后。有多少种使皇后不能互相攻击的方案？（$N \leqslant 13$）

(1) 深度优先搜索（DFS）

逐列（行）搜索。每测试一个位置，就检查它是否与其他皇后冲突，如果冲突，回溯[①]。

每放置一个皇后，就要记录——所在列、“/”对角线和“\”对角线都不能放皇后了。

```
#include <iostream>
#include <cstring>
using namespace std;

bool column[20],cross1[50],cross2[50];
int pos[20];
int n, sum=0;

void dfs(int row)
{
    if (row==n)
    {
        sum++;
        return;
    }

    for (int i=0;i<n;i++)
        if (!(column[i] || cross1[row-i+n] || cross2[row+i])) // 判断是否可以放置皇后
        {
            // 对皇后已经控制的列和对角线做标记
            column[i]=cross1[row-i+n]=cross2[row+i]=true;
            pos[row]=i;

            dfs(row+1);

            // 解除标记，这样才能在新位置上放置皇后。
            column[i]=cross1[row-i+n]=cross2[row+i]=false;
        }
}

int main()
{
    cin>>n;
    memset(column,0,sizeof(column));
```

[①] 一般情况下，深度优先搜索要使用回溯法。

```
    memset(cross1,0,sizeof(cross1));
    memset(cross2,0,sizeof(cross2));

    dfs(0);

    cout<<sum<<endl;
    return 0;
}
```

(2) 状态压缩*

把状态放在一个 4 字节的 int 整型变量中，并且使用位运算——这样，速度会比朴素算法的快很多。

```
#include <iostream>
using namespace std;

// sum用来记录皇后放置成功的不同布局数；upperlim用来标记所有列都已经放置好了皇后。
int n, sum = 0, upperlim = 1;

// 搜索算法，从最右边的列开始。
void test(long row, long ld, long rd)
{
    if (row != upperlim)
    {
        /*
          row，ld，rd进行“或”运算，求得所有可以放置皇后的列,对应位为0，
            然后再取反后“与”上全1的数，来求得当前所有可以放置皇后的位置，对应列改为1。
            也就是求取当前哪些列可以放置皇后。
        */
        long pos = upperlim & ~(row | ld | rd);
        while (pos) // 0 -- 皇后没有地方可放，回溯。
        {
            /*
              拷贝pos最右边为1的bit，其余bit置0。
              也就是取得可以放皇后的最右边的列。
            */
            long p = pos & -pos;

            /*
              将pos最右边为1的bit清零。
              也就是为获取下一次的最右可用列使用做准备，
              程序将来会回溯到这个位置继续试探。
            */
            pos -= p;

            /*
              row + p，将当前列置1，表示记录这次皇后放置的列。
              (ld + p) << 1，标记当前皇后左边相邻的列不允许下一个皇后放置。
              (ld + p) >> 1，标记当前皇后右边相邻的列不允许下一个皇后放置。
```

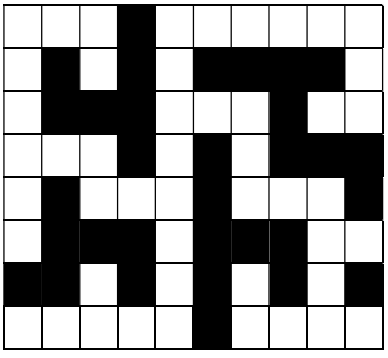
```
            */
            /*
              此处的移位操作实际上是记录对角线上的限制，只是因为问题都化归
              到一行网格上来解决，所以表示为列的限制就可以了。显然，随着移位
              在每次选择列之前进行，原来N×N网格中某个已放置的皇后针对其对角线
              上产生的限制都被记录下来了。
            */
            test(row + p, (ld + p) << 1, (rd + p) >> 1);
        }
    }
    else
        sum++;          // row的所有位都为1，即找到了一个成功的布局，回溯。
}

int main()
{
    cin>>n;

    // N个皇后只需N位存储，N列中某列有皇后则对应bit置1。
    upperlim = (upperlim << n) - 1;

    test(0, 0, 0);
    cout<<sum;
    return 0;
}
```

3.2　走迷宫

【问题描述】从文件 map.txt 中读入一个 m 行 n 列的迷宫，其中 0 代表可以通过，1 代表不可通过（**墙**）。

请给出一条从(1,1)出发到(m,n)的路径。如果不能到达，输出“Failed”。例如：

```
10 8
0001000000
0101011110
0111000100
0001010111
0100010001
0111011100
1101010101
0000010000
```



(1) 预处理

为了防止在搜索时发生下标越界，我们需要做一个小处理：在迷宫的外圈圈上一层围墙，即

```
111111111111
100010000001
101010111101
```

```
101110001001
100010101111
101000100011
101110111001
111010101011
100000100001
111111111111
```

以下是一些定义：

```
char map[110][110];             // 先行后列。
bool visited[110][110];
point prev[110][110];           // 记录路径的方法：从A结点扩展到B结点时，将prev[B]设为A。
struct point {int r,c;};
int n,m;

void flag(int r, int c)         // 追踪结点，在经过的路径上画箭头。
{
    point &t=prev[r][c];
    if (!(r==1&&c==1)) flag(t.r,t.c);

    if (t.r==r && t.c<c) map[r][c]='v';
    else if (t.r==r && t.c>c) map[r][c]='^';
    else if (t.r>r && t.c==c) map[r][c]='<';
    else if (t.r<r && t.c==c) map[r][c]='>';
}
```

(2) 深度优先搜索（DFS）

搜索方法叫做“一条道走到黑”。

```
#define isok(a,b)    (Map[a][b]!='1')&&(visited[a][b]==false)
#define expand(a,b) {prev[a][b]=p;DFS(a,b);}

void DFS(int r,int c)
{
    if (r==m&&c==n)
    {
        flag(m,n);
        for (int i=0;i<=m+1;i++) cout<<map[i]<<endl;
        exit(0);                    // 该语句会使程序立即退出。
    }

    visited[r][c]=true;

    point p; p.r=r; p.c=c;
    if (isok(r-1,c)) expand(r-1,c)
    if (isok(r,c-1)) expand(r,c-1)
    if (isok(r+1,c)) expand(r+1,c)
    if (isok(r,c+1)) expand(r,c+1)
```

```
}
```

(3) 广度优先搜索（BFS）

搜索过程像墨水的扩散。

```
queue<point> q;

#define isok(a,b)    (map[a][b]!='1')&&(visited[a][b]==false)    // 判断能否扩展
#define expand(a,b) {prev[a][b]=p;t.r=a;t.c=b;q.push(t);}        // 扩展结点

void BFS()
{
    point t,p; p.r=p.c=1;
    q.push(p);

    while (!q.empty())
    {
        p=q.front(); q.pop();

        int &r=p.r, &c=p.c;
        visited[r][c]=true;

        if (r==m&&c==n)
        {
            flag(m,n);
            for (int i=0;i<=m+1;i++) cout<<map[i]<<endl;
            return;
        }

        if (isok(r-1,c)) expand(r-1,c)
        if (isok(r,c-1)) expand(r,c-1)
        if (isok(r+1,c)) expand(r+1,c)
        if (isok(r,c+1)) expand(r,c+1)
    }
    cout<<"No Way."<<endl;
}
```

正常形态的迷宫由于道路“狭窄”，能扩展的结点很少，所以 DFS、BFS、双向搜索和启发式搜索都能承受。但是，如果换成畸形迷宫（比如 100×100 的“0”），DFS 和 BFS 将无法忍受。

3.3　8 数码问题

【问题描述】用最少的步数实现（空格用 0 表示）：

2	8	3
1	6	4
7		5

→

1	2	3
4	5	6
7	8	

(1) BFS[①]

```
typedef int state[9];
const int N=1000000;
state st[N],goal;
int dist[N],parent[N];

const int dx[]={-1,1,0,0}, dy[]={0,0,-1,1};

void print(int p)
{
    if (parent[p]!=-1) print(parent[p]);

    // 输出st[p]
    state &s=st[p];
    cout<<s[0]<<" "<<s[1]<<" "<<s[2]<<endl;
    cout<<s[3]<<" "<<s[4]<<" "<<s[5]<<endl;
    cout<<s[6]<<" "<<s[7]<<" "<<s[8]<<endl;
    cout<<endl;
}

int BFS()                               // 返回目标状态在st[]中的下标
{
    init_lookup_table();                // 初始化查找表（判重用）
    int front=1, rear=2;
    parent[front]=-1;
    while (front<rear)
    {
        state &s=st[front];
        if (memcmp(goal, s, sizeof(s))==0) return front;
        int z;
        for (z=0; z<9; z++) if (s[z]==0) break;          // 找“零”
        int x=z/3, y=z%3;                                // “0”的行列编号
        for (int d=0; d<4; d++)
        {
            int newx=x+dx[d];
            int newy=y+dy[d];
            int newz=newx*3+newy;
            if (newx>=0 && newx<3 && newy>=0 && newy<3)
            {
                state &t=st[rear];
                memcpy(&t, &s, sizeof(s));               // 扩展结点
                t[newz]=s[z];
                t[z]=s[newz];
                dist[rear]=dist[front]+1;
                parent[rear]=front;
                if (try_to_insert(rear)) rear++;
```

[①] 以下代码直接摘自刘汝佳的《算法竞赛入门经典》。

```
            }
        }
        front++;
    }
    return 0;
}
```

(2) 顺序化*

下面的代码把 0～8 的全排列和 0～362879（9!－1）的整数一一对应起来。

```
bool visited[362880];
int fact[9];
void init_lookup_table()
{
    fact[0]=1;
    for (int i=1;i<9;i++) fact[i]=fact[i-1]*i;
}
bool try_to_insert(int s)
{
    int code=0;                 // 把st[s]映射到整数code
    state &p = st[s];
    for (int i=0; i<9; i++)
    {
        int cnt=0;
        for (int j=i+1; j<9; j++)
            if (p[j]<p[i])
                code+=fact[8-i]*cnt;
    }
    if (visited[code])
        return false;
    else
        return visited[code]=true;
}
```

(3) 使用哈希表判重

上面的编码法时间效率很高，但是毕竟不是通用的方法。如果结点数非常大，编码也是无法承受的。

```
const int MAXSIZE = 1000003;
int head[MAXSIZE], next[MAXSIZE];
void init_lookup_table() {memset(head,0,sizeof(head));}
int h(state &s)
{
    // 散列函数：把每个格子里数连接成一个九位数，并将它除以MAXSIZE的余数作为哈希函数值。
    int v=0;
    for (int i=0; i<9; i++) v=v*10+s[i];
    return v%MAXSIZE;
}
bool try_to_insert(int s)
```

```
{
    int h=hash(st[s]);
    int u=head[h];
    while (u)
    {
        if (memcmp(st[u],st[s],sizeof(st[s]))==0) return false;
        u=next[u];
    }
    next[s]=head[h];
    head[h]=s;
    return true;
}
```

3.4　埃及分数

【问题描述】在古埃及，人们使用单位分数的和（**形如** 1/a，**其中** a **是自然数**）表示一切有理数。

如：2/3＝1/2＋1/6，但不允许 2/3＝1/3＋1/3，因为加数中有相同的。

对于一个分数 a/b，表示方法有很多种，但是哪种最好呢？首先，加数少的比加数多的好；其次，加数个数相同的，最小的分数越大越好。如：

19/45＝1/3＋1/12＋1/180

19/45＝1/3＋1/15＋1/45

19/45＝1/3＋1/18＋1/30

19/45＝1/4＋1/6＋1/180

19/45＝1/5＋1/6＋1/18

最好的是最后一种，因为 1/18 比 1/180，1/45，1/30，1/180 都大。

给出 a，b（$0<a<b<1000$），编程计算最好的表达方式（**输入** a，b，**按从小到大顺序输出所有分母**）。

【分析】

1. **迭代加深搜索**：本题由于搜索层数不明，用深搜极易陷入死胡同，用广搜空间又吃不消，这时迭代加深搜索就成了考虑的对象。
2. **有序化**：枚举的对象为分母，$\frac{a}{b}=\frac{1}{a_1}+\frac{1}{a_2}+\frac{1}{a_3}+\cdots+\frac{1}{a_n}$，输出又要求有序，所以，不妨设 $a_1<a_2<\cdots<a_n$。
3. **定界法**：设限定的搜索层数为 depth，当前搜到第 k 层，当前正要枚举分母 a_k，还需枚举总和为 x/y 的分数。answer[d]表示当前最优解中的第 d 个分母，如果还没有得到解则表示正无穷。则必然有：
   $\max\{y/x, a_{k-1}\}+1 \leqslant a_k \leqslant \min\{(D-C+1)*y/x, \text{Maxlongint}^{①}/x, \text{answer[depth]}-1\}$
   枚举的初值容易得出，但终值的确定则要用到我们一开始对分母有序性的假设了。值得注意的是，直接限界避免了搜索过程中屡次使用可行性剪枝，在一定程度上可以提高了程序的运行速度。

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
const int MAXDEPTH=10, maxlongint=2147483647;
int depth;
```

① maxlongint 是 Pascal 语言中的常数，它等于 2147483647L，即 int 类型可以取到的最大值。

```
bool found;
int answer[MAXDEPTH], d[MAXDEPTH];

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

void search(int a, int b, int k)
{
    int i,m,s,t;

    if (k==depth+1)
        return;
    else if (b%a==0 && b/a>d[k-1])
    {
        d[k]=b/a;
        if (!found || d[k]<answer[k])
            memcpy(answer,d,sizeof(d));       // 更新最优解
        found = true;
        return;
    }

    // 确定上下界
    s = max(b/a, d[k-1]) + 1;

    t=(depth-k+1)*b/a;
    t=min(t, maxlongint/b);
    if (found) t=min(t,answer[depth]-1);

    for (i=s; i<=t; i++)
    {
        // 从a/b中减去那个埃及分数
        d[k]=i;
        m=gcd(i*a-b, b*i);
        search((i*a-b)/m, b*i/m, k+1);
    }
}

int main()
{
```

```
    int a,b;
    int i;

    found = false;
    d[0] = 1;
    cin>>a>>b;

    for (depth=1; depth<=MAXDEPTH; depth++)
    {
        search(a,b,1);
        if (found)
        {
            for(i=1;i<=depth;i++)
                cout<<answer[i]<<" ";
            cout<<endl;
            break;
        }
    }
    return 0;
}
```

3.5　Mayan 游戏[①]

由于题目很长，这里不再描述。大家可以“百度一下”，或在百度文库中查 NOIP2011 试题。

【分析】

问题的性质加上 3s 的运行时间告诉我们：这个世界需要暴力。

一个块向右交换，与它右边的块向左交换是等价的，但前者的字典序更小，所以不必考虑后者。

此外，如果一种颜色的数量在 1～2 之间，就可以直接剪枝。

与其说本题在考察搜索，不如说本题在考察选手的编程习惯和心理素质。很多人没有清晰明确的思路和良好的编码习惯，结果“投降”，输出了“-1”，或者写了数 KB 代码却没有得到多少分。所以大家要养成好的编程习惯，包括变量的命名、代码缩进、注释等，同时要发扬敢于使用暴力的精神。

【代码】

下面代码没有任何剪枝，但已经测试通过了。

```
#include <iostream>
#include <cstring>
using namespace std;

inline void swap(int &a, int &b) { int t=a; a=b; b=t; }

// 程序中使用横向棋盘，5行7列，水平移动变竖直移动，竖直移动变水平运动。
int n;
int map[7][6][9];
int a[6],b[6],c[6];             // 每一步的变化，a是棋盘倒放之后的行，b是列
```

[①] 题目来源：NOIP2011 day1 第三题。

```
// 方块下落
void fall(int depth)
{
    for (int i=1; i<=5; i++)
    {
        int *m=map[depth][i];

        // 在第i行中，寻找第一个0，然后再寻找非0，接下来移动。不过不一定正确……
        int j=1,k=0;

        while (m[j]) j++;
        if (j==8) continue;

        k=j+1;
        while (m[k]==0 && k<8) k++;

        while (k<8) swap(m[j++],m[k++]);
    }
}

// 检查是否可以消除方块。如果可以，就消除方块。
bool mark[6][8];                // 消除方块时用
bool check(int depth)
{
    bool changed=false;
    memset(mark,0,sizeof(mark));

    // 检查是否可以消除
    // 水平方向
    for (int i=1; i<=5; i++)
        for (int delta=7; delta>=3; delta--)     // 连通的方块数
            for (int j=1; j<=8-delta; j++)
            {
                int a=map[depth][i][j];
                for (int k=j+1; k<=j+delta-1; k++)
                    if (map[depth][i][k]!=a)
                    {
                        a=0;
                        break;
                    }

                if (a) for (int k=j; k<=j+delta-1; k++) mark[i][k]=true;
            }

    // 竖直方向
    for (int j=1; j<=7; j++)
        for (int delta=5; delta>=3; delta--)
            for (int i=1; i<=6-delta; i++)
```

```
            {
                int a=map[depth][i][j];
                for (int k=i+1; k<=i+delta-1; k++)
                    if (map[depth][k][j]!=a)
                    {
                        a=0;
                        break;
                    }

                if (a) for (int k=i; k<=i+delta-1; k++) mark[k][j]=true;
            }

    // 消除
    for (int i=1; i<=5; i++)
        for (int j=1; j<=7; j++)
            if (mark[i][j]) map[depth][i][j]=0, changed=true;

    return changed;
}

// 进行搜索，如果有解则返回true，否则返回false。
bool DFS(int depth)
{
    if (depth>n)
    {
        for (int i=1; i<=5; i++)
            if (map[n+1][i][1]) return false;
        return true;
    }

    for (int i=1; i<=5; i++)
        for (int j=1; j<=7; j++)
        {
            /*
             移动一种方块。分四种情况：
             ① 方块本身就是0，那么直接跳过；
             ② 在左边界，只能向右移动；
             ③ 在右边界，只能向左移动。如果左边有方块，那么可以忽略这一步，
                因为那个方块右移和这个方块左移效果相同，但字典序更小。
                也就是说，在右边界时，如果左边为空，则向左移动，否则不移动。
             ④ 不在边界，如果左边为空，就既向左又向右（特殊考虑，因为要多一段代码）；
                如果左边不为空，那么不管右边是什么都向右移动。
            */
            if (map[depth][i][j]==0) continue;

            a[depth]=i;
            b[depth]=j;
            int &dir=c[depth];                          // 移动方向
```

```
            if (i<5)                                        // 第②种情况＋第④种情况的第一部分
                dir=1;
            else if (i==5 && map[depth][4][j]==0)    // 第③种情况
                dir=-1;
            else
                continue;

            memcpy(map[depth+1], map[depth], sizeof(map[depth]));
            swap(map[depth+1][i][j], map[depth+1][i+dir][j]);

            // 下落与消除方块
            do
            {
                fall(depth+1);
            } while (check(depth+1));

            // 继续搜索
            if (DFS(depth+1)) return true;

            if (i>1 && i<5 && map[depth][i-1][j]==0) // 第④种情况的第二部分
            {
                a[depth]=i;
                b[depth]=j;
                dir=-1;
                memcpy(map[depth+1], map[depth], sizeof(map[depth]));
                swap(map[depth+1][i][j], map[depth+1][i+dir][j]);

                // 和上面一样
                do
                {
                    fall(depth+1);
                } while (check(depth+1));
                if (DFS(depth+1)) return true;
            }
        }

    return false;
}

int main()
{
    memset(map,0,sizeof(map));

    cin>>n;
    for (int i=1; i<=5; i++)
    {
        int temp, j=1;
```

```
        do
        {
            cin>>temp;
            map[1][i][j++]=temp;
        } while (temp);
    }

    if (DFS(1))
        for (int i=1; i<=n; i++)
            cout<<(a[i]-1)<<" "<<(b[i]-1)<<" "<<c[i]<<endl;
    else
        cout<<"-1"<<endl;

    return 0;
}
```

3.6　预处理和优化

(1) 预处理

1．数学性分析，找出一定规律，减少搜索范围。

【问题描述】在 IOI98[①]的节日宴会上,我们有 N（10≤N≤100）盏彩色灯，他们分别从 1 到 N 被标上号码。这些灯都连接到四个按钮。

按钮 1：当按下此按钮，将改变所有的灯，使本来亮着的灯熄灭，本来是关着的灯被点亮。

按钮 2：当按下此按钮，将改变所有奇数号的灯。

按钮 3：当按下此按钮，将改变所有偶数号的灯。

按钮 4：当按下此按钮,将改变所有序号是 $3k+1(k\geq 0)$的灯，例如 1，4，7……

此外一个计数器 C 记录按钮被按下的次数。当宴会开始，所有的灯都亮着，此时计数器 C 为 0。

你将得到计数器 C（0≤C≤10000）上的数值和经过若干操作后所有灯的状态。写一个程序去找出所有灯最后可能的与所给出信息相符的状态，并且没有重复。

【分析】

灯的状态多达 2^{100} 个，似乎无法枚举。

为了找到按钮之间的联系和控制规律，我们不妨列个表说明一下（“√”**表示能被按钮控制**）：

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	…
#1	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	…
#2	√		√		√		√		√		√		√		√		√		√		…
#3		√		√		√		√		√		√		√		√		√		√	…
#4	√			√			√			√			√			√			√		…

我们发现，控制情况是“循环”的，每 6 个灯为一个周期。1、7、13、19……，2、8、14、20……灯的状态是相同的。所以只需枚举前 6 个灯的状态，至于后面的灯，直接对 6 取模即可。

2．保存一些运算结果。如记忆化搜索。

3．优先在更小的搜索树中搜索，减少搜索量。

[①] 这是 IOI98 的一道题，名为 Party Lamps（**派对灯**）。

【问题描述】[①]输出 a 到 b（$5 \leqslant a \leqslant b \leqslant 100000000$）之间既是质数，又是回文数的数。

【分析】

有两种判断方法：一种是先找质数后找回文数，另一种方法是先找回文数然后找质数。

应用素数定理可以估算，100000000 以内约有 540 万个素数，所以显然不能先找质数。

即使是先找回文数，也不能全找。

仔细分析，由于质数的末位不能是 2、4、5、6、8、0（**数字 5 本身要特殊考虑**），所以只需搜尾数为 1、3、7、9 的回文数。

如果数学基础牢固，你还会发现，由偶数个数字组成的回文数可以被 11 整除。于是搜索树又小了许多。

4．有序化处理。

常见的做法是把输入数据排序或分类。有序化搜索往往可以简化问题，帮助分析问题的本质。

（2）通用优化方法

1．DFS

- **剪枝**：可行性剪枝、最优化剪枝
- **记忆化**：这种方法常用于动态规划的动态转移方程中。
- 状态压缩

2．BFS

- **用散列表来判重**：为了避免重复扩展结点，可以使用散列表存储已经扩展的状态，并在扩展结点之前进行判重。如果散列表中找到了对应的状态，就不要再入队了。对于图的遍历，判重可以很大程度地提升速度，减少空间的浪费。

  对于有些问题，布尔数组、二叉排序树等也能起到相同的作用。
- **双向搜索**：由于代码比较复杂，这种搜索方法在竞赛时不推荐使用。
- **启发式搜索**：如果能够构造出好的估价函数，就可以大大减少搜索量。
- 状态压缩

（3）剪枝

剪枝主要有两种，一种是可行性剪枝，另一种是最优化剪枝。它们都可以帮助程序不做“无用功”。

举一个可行性剪枝的例子：在 N 皇后问题中，放置第 i 个皇后时，要进行判断。如果这个皇后和已有的皇后发生冲突，回溯。

如果不进行剪枝，那么就应该继续放置这个皇后，继续搜索。继续搜索的结果是可想而知的。

举一个最优化剪枝的例子：在解决某类最值问题时，如果搜索到一个较优的方案，可以把这个方案记录下来。接下来再进行搜索时，每扩展一个结点，就用估价函数判断一下是否能达到已记录的最优值。如果不能，果断剪枝。

合理的估价函数是最优化剪枝的关键。当然，解最小值问题时可以不用估价函数。

[①] 题目名称为 Prime Palindromes（回文质数），题目来源：USACO Training Gateway。

3.7　代码模板

(1) DFS（递归实现）！

```
void DFS(int depth)
{
    if (depth==n)        // 深度超过范围，说明找到了一个解。
    {
        // 找到了一个解，对这个解进行处理，如：输出、解的数量加1、更新目前搜索到的最优值等
        return;
    }

    // 扩展结点，如
    for (int i=0; i<n; i++)
    {
        // 处理结点
        …

        // 继续搜索
        DFS(depth+1);

        // 部分问题需要恢复状态，如N皇后问题
        …
    }
}
```

非递归版本见 99 页“(2) DFS 和栈”。稳妥的做法是先写递归版本，如果有空闲时间和改写的可能，再把递归版本改写成非递归版本。

(2) BFS！

BFS 占用存储空间比较大，尤其是在状态比较复杂的时候。

树的 BFS 不用判重，因为不会重复。但对于图来说，如果不判重，时间和空间会造成极大的浪费。

```
queue <int> q;                          // 存储状态
// bool try_to_insert(int state);       // 结点入队前判断状态是否重复，以免重复搜索
                                        // 使用散列表可以提高查找的效率。
// void init_lookup_table();

void BFS()
{
    // init_lookup_table();             // 判重之前的初始化
    q.push(…);                          // 初始状态入队
    while (!q.empty())
    {
        int s = q.front(); q.pop();     // 获取状态

        // 处理结点
        if (s达到某种条件)
        {
            // 输出，或做些什么
```

```
            …
            return;
        }

        // 扩展状态
        for (i=0;i<n;i++)
        {
            int s;
            // if (try_to_insert(s)) q.push(s);
            q.push(s);
        }
    }
    // 如果运行到这里，说明无解。
}
```

(3) 迭代加深搜索

```
void search(int depth)              // depth表示深度
{
    if (得到了合适的解)
    {
        // 已经得到了合适的解，接下来输出或解的数量加1
        return;
    }

    if (depth == 0) return;     // 无解

    // 扩展结点，如
    for (int i=0; i<n; i++)
    {
        // 处理结点
        …

        // 继续搜索
        search(depth-1, …);

        // 部分问题需要恢复状态，如N皇后问题
        …
    }
}

const int maxdepth = 10;            // 限定的最大搜索深度
void IDS()
{
    for (int i=1; i<=maxdepth; i++)
        search(i, …);
}
```

3.8　搜索题的一些调试技巧[①]

如果决定用 DFS 做一道题，那么：

- 如果能套用模板或经典的 DFS 模型，就直接套用。
- 如果不能套用，就先写爆搜，并命名为#0 版本。
- **对于每一个版本，调试正确之后，再进行剪枝，并且每次只加一个剪枝，调试正确后再继续。**
- **每次加剪枝都应该开一个新的版本，不能直接修改原版本。**
  如果时间来不及了，就按照从强到弱或从好写到难写的顺序来加剪枝。
- 在动态查错之前应该进行一次静态查错。
- 调试时，如果出现 RE（运行时错误）或死循环，就在出错的状态处停止，并检查其所有的祖先状态（显然这需要记下每次的决策），看看这些决策是否合法。
- 如果程序正常结束，但结果错误，应该从以下三个角度入手：
  1．明明有解，输出无解：可以把正解的各个决策一步一步代入，看看程序里面是在哪一步出了问题（明明合法的决策它木有作出），从而方便检查；
  2．输出不合法的解：可以把形成这个不合法解的决策一步一步代入，看看是哪一步出了问题；
  3．输出合法但非最优的解：必然是某些最优性剪枝错误或者最优性判断出了问题，可以直接到这些地方去检查，实在不行也可以把最优解代入检查。
- 有时也可以用分段检查的方法，即把代码中的各个过程或者片段截取下来，将几组小的输入代入，看看执行之后，所有在该片段里改变了值的全局变量是否正确，进而发现这里的错误。
- **千万不要一下子输出全部的状态！这样不仅不容易查出结果，还把容易把自己绕晕。**
  事实上，只有在状态总数不太多，且相对关系明了、顺序整齐的时候（比如一般的 DP 等），可以一下子输出全部状态来检查，否则就不能这么做。

对于 BFS，其实它比 DFS 容易调试，因为所有的状态都储存在队列里。因此，可以在状态结点中记下每个点的前趋，然后，哪里出了问题，就直接把它的前趋状态全部输出。

另外，BFS 一般没有“剪枝”这一说（除了判重），且决策过程一般是比较整齐的，因此更容易调试。

3.9　小结

搜索，就要用搜索算法来产生所有可能的答案，并从中筛选出正确的答案。

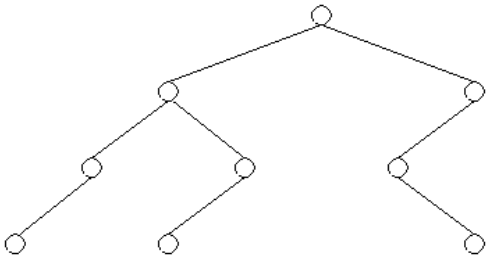
搜索法常用于求可行解或可行解的个数。在竞赛中，搜索法还是“救命稻草”——大多数问题都可以使用搜索来谋取部分分数。

搜索的方式有很多种，常见的有：枚举搜索、深度优先搜索、广度优先搜索、迭代加深搜索。

解搜索题时要从以下几方面入手：确定状态、确定结点扩展方式、选用合适的搜索方式、优化。

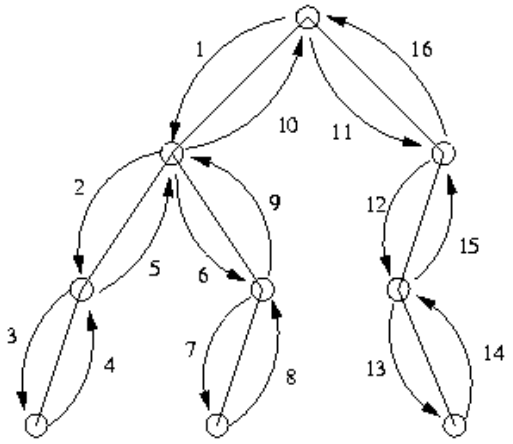
1．深度优先搜索和广度优先搜索

将搜索时产生的状态组成一棵树，就成了解答树。深度优先搜索和广度优先搜索的区别在于解答树的遍历方式。下图是一棵解答树：

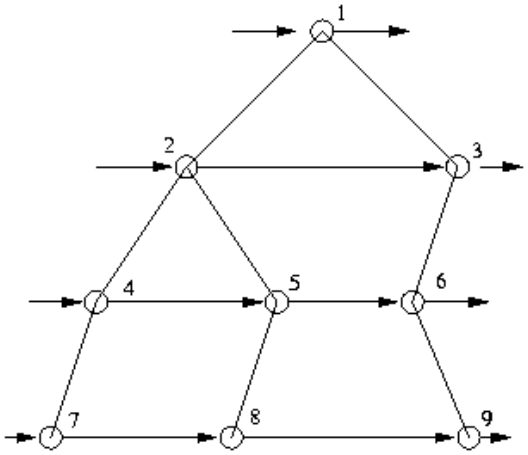


深度优先搜索用逐步加层的方法搜索并且适当时回溯，遍历过程如下图所示：

[①] 选自 Mato_No1 的博客，有改动。博客的网址为 http://www.cppblog.com/MatoNo1。



广度优先搜索得名于它的实现方式：每次都先将搜索树某一层的所有结点全部访问完毕后再访问下一层，再利用先前的那颗搜索树，广度优先搜索以如下顺序遍历：



首先访问根结点，而后是搜索树第一层的所有结点，之后第二层、第三层……以此类推。

此外，在使用广度优先搜索时，应小心处理空间问题。

下面是两种搜索算法的比较：

	DFS	BFS
优势	1．比较适合回溯类搜索 2．参数传递、状态修改和恢复都比较方便 3．自顶向下地处理问题 4．记忆化搜索容易实现 5．能很快到达解答树的底端	1．解决“**最少步数**”、“**深度最小**”问题 2．问题的解靠近解答树的根结点 3．启发式搜索在BFS中更容易实现 4．能立刻停止搜索
缺点	1．使用递归算法容易导致栈溢出 2．有时不容易输出方案 3．不易立即结束搜索	1．空间一般比DFS大 2．状态重复的排除有时耗时多

2．迭代加深搜索

广度优先搜索可以用迭代加深搜索代替。迭代加深搜索实质是限定下界的深度优先搜索，即首先允许深度优先搜索搜索 k 层搜索树，若没有发现可行解，再将 k+1 代入后再进行一次以上步骤，直到搜索到可行解。这个“模仿广度优先搜索”搜索法比起广搜是牺牲了时间，但节约了空间。

在需要做 BFS，但没有足够的空间，时间却很充裕的情况下，可以考虑迭代加深搜索。

迭代加深搜索的时间不会超过等效 BFS 占用时间的两倍，当数据规模增大时，二者的差距还会逐渐减小，而迭代加深搜索的空间占用和 DFS 一样小。

3．搜索的优化

没有经过优化的搜索叫“爆搜”或“裸搜”。爆搜能承受的规模不大，原因有二：一是搜索时产生了无用的结点，二是重复计算。

优化时从以下几方面入手：确定更合适的搜索顺序、剪枝（**可行性剪枝、最优化剪枝**）、降低考察和扩展结点的代价，必要时可采用高级的搜索方法，或使用状态压缩。

此外，写搜索程序时应该先从朴素算法入手，逐步优化，以降低编程复杂度、减少不必要的失误。

第四单元　贪心算法

以下 9 个问题都给出了相应贪心策略，但是为什么这些策略是正确的呢？请自己证明（**提示：反证法**）。

4.1　装载问题

(1) 简单的装载问题

【问题描述】有 n 件物品和一个容量为 C 的背包。第 i 件物品的重量是 $w[i]$。求解将哪些物品装入背包可物品数量最多。

【贪心策略】将物品重量从小到大进行排序，优先挑重量轻的装入背包。

(2) 部分背包问题

【问题描述】有 n 件物品和一个容量为 C 的背包。第 i 件物品的重量是 $w[i]$，价值是 $v[i]$。**每个物体可以取走一部分，价值和重量按比例计算。**求解将哪些物品装入背包可使价值总和最大。

【贪心策略】将背包按照单价（**价值/重量的比值**）排序。从物美价廉（**单价尽可能大**）的物体开始挑起，直到背包装满或没有物体可拿。

(3) 乘船问题

【问题描述】有 n 个人，第 i 个人的重量是 w_i。每艘船的最大载重量都是 C，且最多能乘两个人。用最少的船装尽可能多的人。

【贪心策略】让最轻的人和能与他同船的最重的人乘一条船。如果办不到，就一人一条船。

4.2　区间问题

(1) 选择不相交区间

【问题描述】数轴上有 n 个开区间 (a_i, b_i)。选择尽量多的区间，使这些区间两两没有公共点。

【贪心策略】按 b_i 从小到大的顺序排序，然后一定选择第一个区间。接下来把所有与第一个区间相交的区间排除在外。这样在排序后再扫描一遍即可。

(2) 区间选点问题

【问题描述】数轴上有 n 个闭区间 $[a_i, b_i]$。取尽量少的点，使得每个区间内都至少有一个点（**不同区间内含有的点可以是同一个**）。

【贪心策略】把所有区间按 b_i 从小到大排序（b_i **相同时**，a **从大到小排序**），然后一定取第一个区间的最后一

个点。

(3) 区间覆盖问题

【问题描述】数轴上有 n 个闭区间 $[a_i, b_i]$。选择尽量少的区间来覆盖指定线段 $[s, t]$。

【贪心策略】

1. 预处理：扔掉不能覆盖 $[s, t]$ 的区间。
2. 把各区间按 a 从小到大排序。如果区间 1 的起点不是 s，无解，否则选择起点在 s 的最长区间。
3. 选择此区间 $[a_i, b_i]$ 后，问题转化了覆盖 $[b_i, t]$，于是返回①，直到 $[s, t]$ 被完全覆盖为止。

4.3 删数问题

【问题描述】给出一个 N 位的十进制高精度数，要求从中删掉 S 个数字（**其余数字相对位置不得改变**），使剩余数字组成的数最小。

【贪心策略】

1. 每次找出最靠前的这样的一对数字——两个数字紧邻，且前面的数字大于后面的数字。
   删除这对数字中靠前的一个。
2. 重复步骤 1，直至删去了 S 个数字或找不到这样的一对数。
3. 若还未删够 S 个数字，则舍弃末尾的部分数字，取前 $N-S$ 个。

4.4 工序问题

【问题描述】n 件物品，每件需依次在 A、B 机床上加工。已知第 i 件在 A、B 所需加工时间分别为 A_i、B_i，设计一加工顺序，使所需加工总时间最短。

【贪心策略】

1. 设置集合 F、M、S：先加工 F 中的，再加工 M 中的，最后加工 S 中的。
2. 对第 i 件，若 $A_i > B_i$，则归入 S；若 $A_i = B_i$，则归入 M。否则归入 F（“**拉开时间差**”）。
3. 对 F 中的元素按 A_i 从小到大排列，S 中的按 B_i 从大到小排列。

4.5 种树问题

【问题描述】一条街道分为 n 个区域（**按 1～n 编号**），每个都可种一棵树。有 m 户居民，每户会要求在区域 i～j 区间内种至少一棵树。现求一个能满足所有要求且种树最少的方案。

【贪心策略】

1. 对于要求，以区间右端（**升序**）为首要关键字，左端（**升序**）为次要关键字排序。
2. 按排好的序依次考察这些要求，若未满足，则在其最右端的区域种树，这时可能会满足多个要求。

4.6　马的哈密尔顿链

【问题描述】在一个 8×8 的国际象棋棋盘上，马（“马走日”）的初始位置 (x, y)。怎么走可以不重复地走过每一个格子？这样输出结果：如果马在第 i 步落在了格子 (s, t) 上，则在对应位置输出 i。

【贪心策略】每次往出度最小的点上跳。

```
#include <iostream>
#include <iomanip>
#include <cstring>
using namespace std;

// 两个数组存储对应的偏移量
const int stepRow[8] = {-1,-2,-2,-1,1,2,2,1};
const int stepLine[8] = {-2,-1,1,2,2,1,-1,-2};

int board[8][8];

// 求(i,j)的出口数，各个出口对应的号存在a[]中。
// s表示顺序选择法的开始
int exitn(int i,int j,int s,int *a)
{
    int i1,j1,k,count;

    for (count=k=0;k<8;k++)
    {
        i1 = i + step8[(s + k)% 8];
        j1 = j + step8[(s + k)% 8];
        if (i1>=0 && i1<8 && j1>=0 && j1<8 && board[i1][j1]==0)
            a[count++] = (s + k)% 8;
    }
    return count;
}

// 判断选择下个出口，s 是顺序选择法的开始序号
int next(int i,int j,int s)
{
    int m, kk, a[8], b[8], temp;
    if ((m=exitn(i,j,s,a))==0) return -1;              // 没有出口

    for (int min=9,k=0; k<m; k++)
    {
        // 逐个考虑取下一步最少的出口的出口
        temp = exitn(i+step8[a[k]], j+step8[a[k]], s, b);
        if (temp<min) min = temp, kk = a[k];
    }
    return kk;
}
```

```
int main()
{
    int i,j,step,no,start=0;

    cin>>sx>>sy;
    do
    {
        memset(board,0,sizeof(board));
        board[sx][sy] = 1;
        i = sx, j = sy;

        for (step=2; step<=64; step++)
        {
            no = next(i,j,start)
            if (no == -1) break;
            i += step8[no], j += step8[no];
            board[i][j] = step;
        }

        if (step>64 || no==-1) break;
        start++;
    } while(step<=64);

    if (no != -1)                          // 输出结果
        for (i=0; i<8; i++)
        {
            for (j=0; j<8; j++) cout<<setw(4)<<board[i][j];
            cout<<endl;
        }
    return 0;
}
```

4.7　三值的排序[①]

【问题描述】

排序是一种很频繁的计算任务。现在考虑最多只有三值的排序问题。一个实际的例子是，当我们给某项竞赛的优胜者按金银铜牌序的时候。

在这个任务中可能的值只有三种：1，2 和 3。我们用交换的方法把他排成升序的。

写一个程序计算出把给定的一个由 1、2、3 组成的数字序列排成升序所需的最少交换次数。

【分析】

为了使交换次数最小，我们想到了以下贪心策略：

① 能不交换就不交换；

② 如果能只用一次交换就完成归位，就不用两次交换。

由于排序之后是 11……1122……2233……33 的形式，我们不妨按照排序之后的结果对原数据分区。令

[①] 题目来源：IOI96。但数据规模比原题大。

$w(i,j)$ 表示数字 i 在 j 区的数量。例如 122 312 13 中 $w(2,1)=2$。

按照①的说法，在一区的 1、二区的 2、三区的 3 就不应该再被交换了。

按照②的说法，在一区的 2 应该和二区的 1 进行交换，1 和 3、2 和 3 类似。

设这一次交换的步数为 A，则

$$A=\min\{w(1,2)+w(2,1)\}+\min\{w(1,3)+w(3,1)\}+\min\{w(2,3)+w(3,2)\}$$

接下来已经不能一步恢复两个数字了，就像 312 一样。这时只有先让一个数字归位，然后再交换另外两个。这样，每三个数字需要用两步完成。

设这一次交换的步数为 B，则

$$B=(S-2A)\div 3\times 2$$

其中 S 表示需交换的数字的总个数，即 $S=w(1,2)+w(2,1)+w(2,3)+w(3,2)+w(1,3)+w(3,1)$。

最后将 A 和 B 相加，即最终结果。

4.8　田忌赛马[①]

【问题描述】大家都知道“田忌赛马”的故事。现在，田忌再一次和齐王赛马。他们各派出 N 匹马（$N\leqslant 2000$）。每场比赛，输的一方将要给赢的一方 200 两黄金，如果是平局的话，双方都不必拿出钱。

每匹马的速度值是固定而且已知的，而齐王出马也不管田忌的出马顺序。请问田忌该如何安排自己的马去对抗齐王的马，才能赢最多的钱？

【分析】[②]

题目本身已经告诉我们怎样用二分图最佳匹配来解决这个问题——把田忌的马放左边，把齐王的马放右边，田忌的马 A 和齐王的 B 之间，如果田忌的马胜，则连一条权为 200 的边；如果平局，则连一条权为 0 的边；如果输，则连一条权为－200 的边。

但是题目告诉我们没有必要这样做，我们也无法这样做（**复杂度很高，无法承受 N = 2000 的规模**）。

我们不妨用贪心思想来分析一下问题。因为田忌掌握有比赛的“主动权”，他总是根据齐王所出的马来分配自己的马，所以这里不妨认为齐王的出马顺序是按马的速度从高到低出的。由这样的假设，我们归纳出如下贪心策略：

1．如果田忌剩下的马中最强的马都赢不了齐王剩下的最强的马，那么应该用最差的一匹马去输给齐王最强的马。

2．如果田忌剩下的马中最强的马可以赢齐王剩下的最强的马，那就用这匹马去赢齐王剩下的最强的马。

3．如果田忌剩下的马中最强的马和齐王剩下的最强的马打平，可以选择打平或者用最差的马输掉比赛。

我们发现，第三个贪心策略出现了一个分支：打平或输掉。如果穷举所有的情况，算法的复杂度将比求二分图最佳匹配还要高；如果一概而论的选择让最强的马去打平比赛或者是让最差的马去输掉比赛，则存在反例。

虽然因为第三个贪心出现了分支，我们不能直接的按照这种方法来设计出一个完全贪心的方法，但是通过上述的三种贪心策略，我们可以发现，如果齐王的马是按速度排序之后，从高到低被派出的话，田忌一定是将他马按速度排序之后，从两头取马去和齐王的马比赛。有了这个信息之后，动态规划的模型也就出来了！

设 $f(i,j)$ 表示田忌从“头”取了 i 匹较强的马，从“尾”取了 j 匹较弱的马进行比赛所能够得到的最大盈利，则状态转移方程为：$f(i,j)=\max\{f(i,j-1)+g[n-(j-1)],f(i-1,j)+g(i)\}$

其中 $g(x)$ 表示田忌的第 x 匹马和齐王的第 $i+j$ 匹马（**此时正是第 $i+j$ 场比赛**）分别按照由强到弱的顺序排序之后，田忌所能取得的盈利，胜为 200，输为－200，平为 0。

[①] 题目来源：ACM2004 上海赛区，有删改。

[②] 选自黄劲松《贪婪的动态规划》，有改动。

4.9　小结

贪心算法指从问题的初始状态出发，通过若干次的贪心选择而得出最优值（**或较优解**）的一种解题方法。贪心思想的本质是每次都形成局部最优解，换一种方法说，就是每次都处理出一个最好的方案。

除了本单元的问题，还有一些常见问题也是用贪心算法求解的：建立哈夫曼树、Prim 算法、Kruskal 算法、拓扑排序等。

如果有贪心性质存在，那么一定要采用贪心算法。因为它容易编写，容易调试，速度极快，并且节约空间。几乎可以说，它是所有算法中最好的。

然而，贪心算法并不总是正确的，因为并不是每次局部最优解都会与整体最优解之间有联系，往往靠贪心生成的解不是最优解。一般情况下，构造出贪心策略后要进行证明。

贪心还是一种思想。运用贪心思想，主要是为了分析出问题的一些本质，或者分析出低效算法的一些冗余。合理地运用贪心思想，可以帮助运用其他算法解决问题。

第五单元　分治算法

5.1　一元三次方程求解[①]

【问题描述】有形如 $ax^3+bx^2+cx+d=0$ 这样的一个一元三次方程。给出该方程中各项的系数（a，b，c，d 均为实数），并约定该方程存在三个不同实根（根的范围在-100 至 100 之间），且根与根之差的绝对值≥1。要求由小到大依次在同一行输出这三个实根（根与根之间留有空格），并精确到小数点后 4 位。

【分析】

记方程为 $f(x)=0$，若存在 2 个数 x_1 和 x_2，且 $x_1<x_2$，$f(x_1)\cdot f(x_2)<0$，则在 (x_1, x_2) 之间一定有一个根。

如果保留小数点后 2 位，则可以从-100.00 到 100.00 进行枚举，步长 0.01，然后取使 $f(x)$ 最接近 0 的三个数。

当已知区间 (a,b) 内有一个根时，可用二分法求根。若区间 (a,b) 内有根，则必有 $f(a)\cdot f(b)<0$。重复执行如下的过程：

令 $m=(a+b)\div 2$，

(1) 若 $a+0.0001>b$ 或 $f(m)=0$，则可确定根为 m，并退出过程；

(2) 若 $f(a)\cdot f(m)<0$，则根在区间 (a,m) 中，故对区间重复该过程；

(3) 若 $f(a)\cdot f(m)>0$，则必然有 $f(m)\cdot f(b)<0$，根在 (m,b) 中，对此区间重复该过程。

执行完毕，就可以得到精确到 0.0001 的根。

所以，根据“根与根之差的绝对值≥1”，我们先对区间[-100,-99]、[-99,-98]……[99,100]进行枚举，确定这些区间内是否有解，然后对有解的区间使用上面的二分法。这样就能快速地求出所有的解了。

5.2　快速幂

用朴素算法求 a^n，时间复杂度为 $O(n)$。能不能更快一些？

以 a^{10} 为例。我们知道，求一个数的平方是很快的，因为不用循环。那么 a^{10} 能不能转化为谁的平方？没问题，a^{10} 是 a^5 的平方，也就是说，如果 a^5 求出来了，那么接下来只需对这个结果平方就能得出结果，并且少做了 4 次乘法（**求平方本身需要一次乘法**）！

a^5 能不能用类似的方法求出来？按照刚才的思想，指数应该能被 2 整除。我们可以先求 a^4，然后再乘一个 a 就是 a^5。

很明显，a^4 是 a^2 的平方，而 a^2 可以直接求出来。于是我们最终只做了 4 次乘法。

总结刚才的思路，则有

$$a^n=\begin{cases}a^{n/2}\times a^{n/2} & n\text{是偶数}\\ a^{[n/2]}\times a^{[n/2]}\times a & n\text{是奇数}\end{cases}$$

时间复杂度为 $O(\log n)$。

5.3　排序

(1) 快速排序

快速排序的思路如下：如果序列中只有一个数字，则认为已经排好序了。

在待排序序列中任取一个数（一般取第一个），作为“标志”。扫描序列中的其他数，将比“标志”小的数放到它的左面，将比“标志”大的数放到它的右面。完成后，“标志”就把序列分成了两部分，一部分的数都比它小，另一部分的数都比它大。

[①] 题目来源：NOIP2001 第一题。原题要求保留小数点后两位。

接下来，对被划分出的两部分分别进行快速排序就可以了。（**因为递归结束后，序列已经有序，所以不需要对结果进行合并了。**）

（2）归并排序

归并排序的思路如下：如果序列中只有一个数字，则认为已经排好序了。

划分阶段——将序列分成长度相等或接近的两部分。

递归——对划分后的两部分序列分别进行归并排序。

合并——递归地排序之后，两边的序列都已经有序，接下来需要将这两个有序序列合并。

那么，如何将两个有序序列合并？

首先需要一段临时空间来保存结果，我们不妨叫它“队列”。

接下来在两个序列的开头放置两个“指针”。每次都对“指针”所指的值进行比较，把最小值放到“队列”中，并将那个最小值对应的“指针”向右移动。重复以上步骤，直到所有的数都被放入“队列”。

最后，将“队列”内的数重新放回待排序的序列中就可以了。

（3）求逆序对个数

【问题描述】对于一个序列 a_1，a_2，a_3，…，a_n，如果存在 $i<j$，使 $a_i>a_j$，那么 a_i，a_j 就是一个逆序对。现在给你一个有 N 个元素的序列，请求出序列内逆序对的个数。$N\leqslant 100000$。

【分析】

可以考虑将序列分成两部分，分别求出子序列的逆序对个数。

横跨两序列的逆序对个数怎么统计？用枚举法行吗？不行，那样做就失去分治的意义了。

如果两区间有序，统计就很容易了，所以我们边统计边排序。由于子序列内逆序对个数的统计是在排序之前完成的，排序又不影响两序列之间的逆序对个数，所以排序不会影响最终结果。

```
int cnt=0;                          // 逆序对个数
int a[100002], c[100002];
void MergeSort(int l, int r)    // r＝右边界索引＋1
{
    int mid, i, j, tmp;
    if (r>l+1)
    {
        mid = (l+r)/2;
        MergeSort(l, mid);
        MergeSort(mid, r);
        tmp = l;
        for (i=l,j=mid; i<mid && j<r; )
        {
            if (a[i]>a[j])
            {
                c[tmp++] = a[j++];
                cnt += mid-i;          // 使用排序，就可以方便地数跨“分界”的逆序对个数了
            }
            else
                c[tmp++] = a[i++];
```

```
        }
        if (j<r) for (; j<r; j++) c[tmp++] = a[j];
        else for (; i<mid; i++) c[tmp++] = a[i];
        for (i=l; i<r; i++) a[i]=c[i];
    }
}
```

5.4　最长非降子序列

【问题描述】一个序列 a_1，a_2，a_3，…，a_n 共 N 个元素。现在要求从序列找到一个长度最长、且前面一项不大于它后面任何一项的子序列。只需算出这个序列的长度。N≤100000。

【分析】

用动态规划求解，则状态转移方程为：$f(i)$＝max{$f(j)$}＋1（a_j＞a_i 且 i＞j）。时间复杂度为 O(n^2)，n 很大时会超时。超时的原因是大部分时间都耗在了寻找 max{$f(j)$}上面。

根据这一点，我们可以对算法进行改进。假设数组 C[]已经是符合条件的最长非降子序列，C[i]表示序列中第 i 小的元素，那么我们就可以利用二分查找来确定 f[j]。时间复杂度由 O(n^2)降到了 O($n$log$n$)。

```
int a[N], C[N], f[N], n;
int LIS()                                   // 最长非降子序列
{
    int min, max;                           // x的左右指针
    for (int i=1; i<=n; i++) C[i]=INF;
    for (int i=1; i<=n; i++)
    {
        min=1, max=i;
        while (min<max)
        {
            int mid=(min+max)/2;
            if (C[mid]<a[i]) min=mid+1; else max=mid;
        }
        f[i]=min;
        C[min]=a[i];
    }
    return f[n];
}
```

5.5　循环赛日程表问题

【问题描述】设有 2^n（n≤6）个球队进行单循环比赛，计划在 2^n－1 天内完成，每个队每天进行一场比赛。设计一个比赛的安排，使在 2^n－1 天内每个队都与不同的对手比赛。例如 n＝2 时的比赛安排为：

日期 / 客场 / 主场	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

【分析】

此题很难直接给出结果，我们先将问题进行分解。

设 $m=2^n$，并将规模减半。假如 $n=3$（即 $m=8$），8 个球队的比赛，减半后变成 4 个球队的比赛（$m=4$）。4 个球队的比赛的安排方式还不是很明显，再减半到两个球队的比赛（$m=2$）。

两个球队的比赛安排方式很简单，只要让两个球队直接进行一场比赛即可：1-2

日期/客场/主场	1
1	2
2	1

分析两个球队的比赛的情况不难发现，表格主体是一个对称的方阵，我们把这个方阵分成 4 部分（即左上、右上、左下、右下），右上部分可由左上部分加 1（即加 $m/2$）得到，而左上与右下部分、右上与左下部分分别相等。因此我们也可以把这个方阵看作是由 $m=1$ 的方阵所生成的。同理，可得 $m=4$ 的方阵：

日期/客场/主场	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

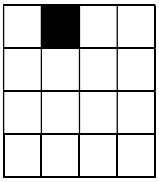
同理可由 $m=4$ 方阵生成 $m=8$ 的方阵：

这样就构成了整个比赛的安排表。在算法设计中，用数组 a 记录 2^n 个球队的循环比赛表，整个循环比赛表从最初的 1×1 方阵按上述规则生成 2×2 的方阵，再生成 4×4 的方阵……直到生成出整个循环比赛表为止。

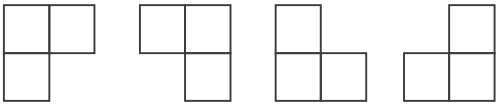
日期/客场/主场	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

5.6　棋盘覆盖

【问题描述】在一个 $2^k\times 2^k$ 方格组成的棋盘中，若恰有一个方格与其他方格不同，则称该方格为一特殊方格，且称该棋盘为一特殊棋盘。现在要用以下 4 种不同形态的 L 形骨牌覆盖一个给定的特殊棋盘上除特殊方格以外的所有方格，且任何 2 个 L 型骨牌不得重叠覆盖。求一种覆盖方法。



特殊棋盘，不能把骨牌放在黑色方格上面。



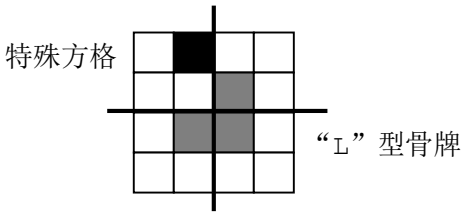
4 种 L 形骨牌

【分析】

尺寸为 $2^k\times 2^k$，我们可以想到将其分割成四个尺寸为 $2^{k-1}\times 2^{k-1}$ 的子棋盘。

接下来，将 L 型放置在棋盘的正中央，使四个子棋盘都变成特殊棋盘（**一个子棋盘含有特殊方格，另外三**

个子棋盘恰好被骨牌覆盖，如下图）。此时问题也变成了四个相同的子问题，只需简单地对四小块区域进行递归，就可以解决这道问题了。



代码的时间复杂度和空间复杂度都是$O(4^k)$。由于覆盖一个$2^k \times 2^k$棋盘所需的L型骨牌个数为$(4^k-1)/3$，所以算法已经是最优的了。

5.7　删除多余括号

【问题描述】输入一个表示算式的字符串（**含四则运算、乘方、括号**），要求去掉可能含有的多余括号，结果要保持原表达式中变量和运算符的相对位置不变，且与原表达式等价。注意，题目只是要求你去括号，并没有要求你化简表达式！此外，“+”和“-”不会用作正负号。

【分析】

符合“多余括号”的情况：

① 括号内没有运算符；

② 括号左侧是加法运算符，右侧是加法或减法运算符；

③ 括号左侧是乘法运算符，右侧是乘法或除法运算符。

对于表达式求值的问题，我们常采用这样的思路处理：找出式中最后运算的运算符A，先递归地求出其左右两边的值，再将得到的值进行A运算。当然这次没有必要进行运算，只需分析括号内的运算符情况就可以了。

```
#include <iostream>
#include <cstring>
using namespace std;

int a[1024];
char s[1024];
int cal(int st, int stp, int prev)
{
    int t, min=4, min_i;
    for (int i=st; i<=stp; i++)
    {
        switch (s[i])
        {
        case '^':
            if (min>3) min=3, min_i=i;
            break;
        case '*': case '/':
            if (min>2) min=2, min_i=i;
            break;
        case '+': case '-':
            if (min>1) min=1, min_i=i;
            break;
```

```
        case '(':
            i++;
            for (t=1;t!=0;i++)
            {
                if (s[i]=='(') t++;
                if (s[i]==')') t--;
            }
            i--;
            break;
        };
    }

    if (min==4)
    {
        if (s[st]=='(' && s[stp]==')')
        {
            t=cal(st+1,stp-1,0);
            if (t>=prev)
            {
                a[st]=a[stp]=1;
                return t;
            }
        }
        return 4;
    }

    cal(st,min_i-1,min);
    if (s[min_i]=='+' || s[min_i]=='*')
        cal(min_i+1,stp,min);
    else
        cal(min_i+1,stp,min+1);

    return min;
}
int main()
{
    cin>>s;
    int sc=strlen(s);
    cal(0,sc-1,0);
    for (int i=0;i<sc;i++) if (!a[i]) cout<<s[i];
    cout<<endl;
    return 0;
}
```

5.8　聪明的质监员[①]

【问题描述】小 T 是一名质量监督员，最近负责检验一批矿产的质量。这批矿产共有 n 个矿石，从 1 到 n 逐一编号，每个矿石都有自己的重量 w_i 以及价值 v_i。检验矿产的流程是：

1．给定 m 个区间 $[L_i, R_i]$；

2．选出一个参数 W；

3．对于一个区间 $[L_i, R_i]$，计算矿石在这个区间上的检验值 Y_i：

$$Y_i=\sum_j 1\times\sum_j v_j，\text{其中 } j\in[L_i, R_i] \text{ 且 } w_j\geqslant W，j \text{ 是矿石编号}$$

这批矿产的检验结果 Y 为各个区间的检验值之和。即：

$$Y=\sum_{i=1}^{m} Y_i$$

若这批矿产的检验结果与所给标准值 S 相差太多，就需要再去检验另一批矿产。小 T 不想费时间去检验另一批矿产，所以他想通过调整参数 W 的值，让检验结果尽可能的靠近标准值 S，即使得 $S-Y$ 的绝对值最小。请你帮忙求出这个最小值。

【分析】

注意求 Y_i 的公式。很明显，随着 W 的增大，符合条件的矿石减少，Y_i、Y 都会减小。所以 Y 是一个随 W 变化而变化的单调函数，也就是说 Y 具有二分性质。利用二分查找就可以找到最接近 S 的 Y。

由于区间很多，Y_i 需要用前序和求。具体求法见 191 页“(2) 带条件的区间求和”。

```
#include <iostream>
#include <cstring>
using namespace std;
inline long long _abs(long long x) { return x>0 ? x : -x; }

const int N=200009;
int n,m;
int w[N],v[N], L[N],R[N];
long long S;

int sumN[N];
long long sumV[N];
inline long long Y(int W)
{
    long long result=0;

    sumN[0]=sumV[0]=0;
    for (int i=1; i<=n; i++)
        if (w[i]>=W)
        {
            sumN[i]=sumN[i-1]+1;
            sumV[i]=sumV[i-1]+v[i];
        }
        else
        {
            sumN[i]=sumN[i-1];
```

[①] 题目来源：NOIP2011 day2 第二题

```
            sumV[i]=sumV[i-1];
        }

    for (int i=1; i<=m; i++)
    {
        int &l=L[i], &r=R[i];
        result+=(sumN[r]-sumN[l-1]) * (sumV[r]-sumV[l-1]);
        if (result>10000000000000LL) return 10000000000000LL;     // 防溢出
    }
    return result;
}

int maxW=0;
int main()
{
    ios::sync_with_stdio(false);
    cin>>n>>m>>S;

    for (int i=1; i<=n; i++)
    {
        cin>>w[i]>>v[i];
        if (w[i]>maxW) maxW=w[i];
    }
    for (int i=1; i<=m; i++)
        cin>>L[i]>>R[i];

    // 二分查找
    int x=0, y=maxW+1, mid;
    while (x<y)
    {
        mid = (x+y)/2;
        if (Y(mid)>S) x=mid+1; else y=mid;
    }
    long long a,b;
    a=_abs(S-Y(x));          // 由于Y不一定等于S，所以要判断最接近S的两个数。
    b=_abs(S-Y(x-1));

    if (a>b) cout<<b<<endl; else cout<<a<<endl;
    return 0;
}
```

5.9　模板

这是分治算法模板的一种，另外一种模板是二分算法。

```
void solve(p)            // p表示问题的范围、规模或别的东西。
{
    if (p的规模够小) { 用简单的办法解决 }
```

```
    // 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题。
    // 一般把问题分成规模大致相同的两个子问题。
    for (int i=1;i<=k;i++)
        把p分解，第i个子问题为pi

    // 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题。
    for (int i=1;i<=k;i++)
        solve(pi);

    // 合并：将各个子问题的解合并为原问题的解。
    ……
}
```

5.10　小结

分治法：对于一个规模为 n 的问题，若该问题可以容易地解决（**比如说规模 n 较小**）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，**递归**地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

分治法所能解决的问题一般具有以下几个特征：

1．该问题的规模缩小到一定的程度就可以容易地解决。
2．该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。
3．利用该问题分解出的子问题的解可以合并为该问题的解。
4．该问题所分解出的各个子问题是相互独立的，即子问题之间没有交集。

第六单元　动态规划

6.1　导例：数字三角形[①]

【问题简述】有一个层数为 n（n≤1000）的数字三角形（**如下图**）。现有一只蚂蚁从顶层开始向下走，每走下一级时，可向左下方向或右下方向走。求走到底层后它所经过数字的总和的最大值。

```
          1
        6   3
      8   2   6        最大值＝1＋3＋6＋6＋7＝23
    2   1   6   5
  3   2   4   7   6
```

(1) 数据存储

```
          1                0 0 0 0 0 0
        6   3              0 1 0 0 0 0
      8   2   6      →     0 6 3 0 0 0
    2   1   6   5          0 8 2 6 0 0
  3   2   4   7   6        0 2 1 6 5 0
                           0 3 2 4 7 6
每次往左下或右下方走      每次往下方或右下方走
```

边界处理：

1. 最外层有一圈0，可以防止在逆推法中出界。
2. 由于右侧是0，所以决策时不会往那里走。
   如果允许负数，可以把0换成一个很小的数，也可以想其他方法。

(2) 递推

1. **划分阶段**：按行划分阶段，一行为一个阶段。
2. **状态表示**：令 $f(i,j)$ 为第 i 行第 j 列上点到起点的最大和。所有下标从 1 开始。
3. **状态转移方程**：

- 逆推：状态转移方程为 $f(i,j)=\max\begin{cases}f(i+1,j)\\f(i+1,j+1)\end{cases}+a(i,j)$。结果为 $f(1,1)$

  计算顺序：先把 $f(i+1)$ 一组算好，然后再计算 $f(i)$ 的一组……最后直接输出 $f(1,1)$。

```
int a[N][N], f[N][N];                // a是变形的数字三角形，f保存计算结果
……
for (int i=n;i>0;i--)
    for (int j=1;j<=i;j++)
        f[i][j] = max(f[i+1][j], f[i+1][j+1]) + a[i][j];
cout<<f[1][1];
```

- 顺推：状态转移方程为 $f(i,j)=\max\begin{cases}f(i-1,j)\\f(i-1,j-1)\end{cases}+a(i,j)$。结果是 $\max\{f(n,1),\ \cdots,\ f(n,n)\}$。

  计算顺序：先把 $f[i]$ 一组算好，再计算 $f[i+1]$ 的一组……在比较之后输出结果。

```
int a[N][N], f[N][N], result=0;      // result表示计算结果
……
for (int i=1;i<=n;i++)
    for (int j=1;j<=i;j++)
        f[i][j] = max(f[i-1][j], f[i-1][j-1]) + a[i][j];
for (int i=1;i<=n;i++) if (f[n][i]>result) result=f[n][i];
```

[①] 题目来源：IOI'94

```
cout<<result;
```

(3) 记忆化搜索——用递归代替递推

如果写出了状态转移方程，却不知道如何递推计算，就可以使用记忆化搜索。

以逆推的状态转移方程为例：

```
bool visited[N][N];
// visited[i][j]表示f[i][j]是否算过。
// 也可以用f[i][j]=-1表示没算过。其实，只要把“算过”和“没算过”区分开就可以。
int a[N][N], f[N][N];
……
int F(int x, int y)
{
    if (visited[x][y]) return f[x][y];        // 如果算过，直接返回结果，否则递归计算。
    if (x>n) return 0;                        // 边界处理

    visited[x][y] = true;
    return f[x][y] = max(F[i+1][j], F[i+1][j+1]) + a[i][j];
}
……
memset(visited,0,sizeof(visited));
cout<<F(1,1);
```

它的时间复杂度尽管也是 $O(n^2)$，但运行起来要比递推慢得多。

(4) 记录路径

最大值是怎么来的？为了解决这个问题，可以再开一个数组 $g[N][N]$ 来记录每一步的选择。这样，$g[i][j]$ 就表示“$f(i,j)$ 是从哪里来的”，或者说“用状态转移方程中的哪一条”。然后通过递归就可以找到路径上的数。下面以顺推法为例。

```
int a[N][N], f[N][N], g[N][N];            // g[i][j]记录着“选择”
void printpath(int i, int j)              // 输出时务必注意顺序
{
    if (i==0||j==0)
        return;
    else if (g[i][j]==1)
        printpath(i-1,j);
    else
        printpath(i-1,j-1);
    cout<<a[i][j]<<' ';
}
```

```
int result,p;                             // p表示终点的位置
for (int i=1;i<=n;i++)
{
    result=p=0;
    for (int j=1;j<=i;j++)
    {
        f[i][j] = f[i-1][j] + a[i][j];
```

```
        g[i][j] = 1;                          // 状态转移方程中有两个选择，现在是第一个。
        if (f[i][j] > f[i-1][j-1] + a[i][j])
        {
            f[i][j] = f[i-1][j-1] + a[i][j];
            g[i][j] = 2;
        }
        if (result > f[i][j]) result=f[i][j],p=j;
    }
}
cout<<result;
printpath(n, p);
```

(5) 使用滚动数组

“空间换时间”虽然很好，但是有的时候，数组根本开不下！

递推的时候，我们发现尽管 f 有 n 行，但只有两行参与了计算。所以可以把 f 变成 $f[1][N]$ 来节省空间。计算的时候，要让 i 和 $i-1$ 对 2 取模，即

```
f[i%2][j] = max(f[(i-1)%2][j], f[(i-1)%2][j-1]) + a[i][j];
```

如果看着不舒服，可以改成：

```
#define F(i,j) f[(i)%2][j]                // 注意括号
……
F(i,j) = max(F(i-1,j), F(i-1,j-1)) + a[i][j];
```

不过，要求输出字典序最小[①]的解决方案时，最好不要使用滚动数组。

(6) 非完美解法

如果你在竞赛时被动态规划题卡住，你还不想得 0 分，你可以考虑以下几种方法：

① 暴力搜索

```
int result = 0;
void search(int x, int y, int depth, int sum)
{
    if (depth==n)
    {
        if (sum>result) result=sum;
        return;
    }
    else
    {
        search(x+1,y,depth+1,sum + a[x+1][y]);
        search(x+1,y+1,depth+1,sum + a[x+1][y+1]);
    }
}
```

调用 try(1,1,1,a[1][1])即可。时间复杂度 $O(2^n)$。

② 贪心＋搜索

对于某些求最小值的问题，可以先用贪心算法得到较优解。然后开始搜索。搜索时只要计算结果比已求得

[①] “字典序最小”：一要保证步骤最少，二要保证在步骤相同的情况下，优先选择序号靠前的决策。

的最小值大就剪枝。搜索到顶点时更新最小值。

不过，求最大值的问题不能使用此方法。

③ 随机化

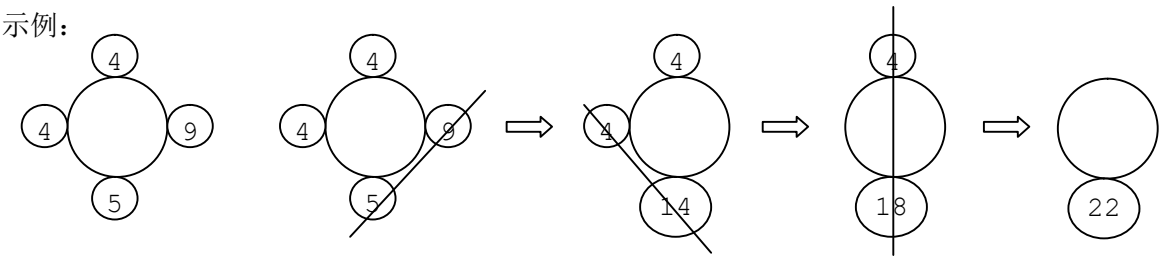
从最顶层开始，让向左走和向右走的概率相等[①]。当然可以加入贪心思想——让较大值方向的概率大一些。将这个过程重复若干次，总有可能得到正确答案。

```
int result=0;
srand(time(NULL));
for (int t=1; t<=10000; t++)
{
    int x=0, v=0;
    for (int i=1; i<=n; i++)
    {
        double p=(double)rand()/(double)RAND_MAX;
        if (p<0.5) x+=0; else x+=1;
        v+=a[i][x];
    }
    if (v>result) result=v;
}
```

6.2　区间问题：石子合并[②]

【问题简述】n 堆石子围成一圈，每堆石子的量 $a[i]$ 已知。每次可以将相邻两堆合并为一堆，将合并后石子的总量记为这次合并的得分。$n-1$ 次合并后石子成为一堆。求这 $n-1$ 次合并的得分之和可能的最大值和最小值。（$n\leqslant 100$，$1\leqslant i\leqslant n$）

示例：



这种合并方法的总得分＝14＋18＋22＝54

(1) 环的处理方法

以某一点作为起点，按顺时针或逆时针的顺序把环上的元素复制两遍。处理时注意起点范围（0～$n-1$）和最大长度（n）。

例如上面的示例，可以变成：5 9 4 4 5 9 4 4，这样就包含了分别以 5、9、4、4 为起点的 4 个环。

(2) 求解

1. **递推思路**：计算将第 i 堆至第 j 堆完全合并所能获得的最大得分。这是此题的关键。考虑模拟每种合并后的具体情形是行不通的。把问题变成这样后就好解决了。
2. **划分阶段**：以合并的次数作为标准划分阶段。
3. **确定状态**：$f_1(i,j)$ 表示第 i 堆至第 j 堆合并所能获得的最大价值，$f_2(i,j)$ 表示第 i 堆至第 j 堆合并所能获得

[①] 实际上，应该让远离中心的方向的概率大一些，因为在等可能的情况下，层数越多，终点在中间位置的概率越大。

[②] 题目来源：NOI'95

的最小价值。

4．**状态转移方程：** $f_1(i,j)=\max\{f_1(i,k)+f_1(k+1,j)+d(i,j)\}$

$f_2(i,j)=\min\{f_2(i,k)+f_2(k+1,j)+d(i,j)\}$，其中 $1\leqslant i\leqslant k<j\leqslant n$

边界状态： $f_1(i,i)=f_2(i,i)=0$

$d(i,j)$ 表示当次合并的得分，即从 i 到 j 的石子数量，$d(i,j)=a[i]+a[i+1]+\cdots+a[j]$，可用前序和求出。

5．递推时注意，循环的最外层不是 i，也不是 j，而是 $j-i$！

最后从 $f_1(1,n)$ 到 $f_1(n,n+n)$，从 $f_2(1,n)$ 到 $f_2(n,n+n)$，找出最值即可。

```
#include <cstring>
#include <iostream>
using namespace std;
const int INF = 10000000;
inline int d(int i,int j) { return s[j]-s[i-1]; }

int f1[101][101],f2[101][101];
int v[201], s[201];
int n;

int main()
{
   memset(f1, 0, sizeof(f1)); memset(f2, 0, sizeof(f2));
   memset(s, 0, sizeof(s));
   cin>>n;

   for (int i=1;i<=n;i++)
   {
      cin>>v[i];
      v[n+i]=v[i];         // 把环拉成链
   }
   for (int i=1;i<=n+n;i++) s[i]=s[i-1]+v[i];      // 前序和

   for (int p=1;p<n;p++)
      for (int i=1,j=i+p;(i<n+n)&&j<=(n+n);i++,j=i+p)
      {
         f1[i][j]=0; f2[i][j]=INF;
         for (int k=i;k<j;k++)
         {
            f1[i][j] = max(f1[i][j], f1[i][k]+f1[k+1][j]+d(i,j));
            f2[i][j] = min(f2[i][j], f2[i][k]+f2[k+1][j]+d(i,j));
         }
         cout<<"f2["<<i<<"]["<<j<<"]="<<f2[i][j]<<endl;
      }

   int r1=0, r2=INF;
   for (int i=1;i<=n;i++)
   {
      if (f1[i][i+n-1]>r1) r1=f1[i][i+n-1];
      if (f2[i][i+n-1]<r2) r2=f2[i][i+n-1];
   }
```

```
    cout<<r1<<" "<<r2<<endl;
    return 0;
}
```

时间复杂度：$O(n^3)$

(3) 能量项链[①]

【问题简述】有一串能量项链，上有 N 颗能量珠。能量珠有头标记与尾标记。并且，对于相邻的两颗珠子，前一颗珠子的尾标记一定等于后一颗珠子的头标记。如果前一颗能量珠的头标记为 m，尾标记为 r，后一颗能量珠的头标记为 r，尾标记为 n，则聚合后释放的能量为 $m\times r\times n$，新产生的珠子的头标记为 m，尾标记为 n。

现在要把所有的珠子聚合成一个珠子。请你设计一个聚合顺序，使一串项链释放出的总能量最大。

【分析】

思路是相似的——阶段、状态都一样，而得分的方式变了。

1. 设第 i 个珠子的头标记是 value[i]，规定所有下标都是从 1 开始。
2. **状态转移方程**：设 $f(i,j)$ 为从 i 到 j 这一条链的最大值，那么

   $f(i,j)=\max\{f(i,k)+f(k,j)+\text{value}[i]*\text{value}[k+1]*\text{value}[j+1]\}$，其中 $i\leqslant k<j$

   边界条件 $f(k,k)=0$。

6.3　坐标问题

(1) 单向取数问题

【问题描述】一个 $m\times n$ 的方格，每个格子都有一个数字。现在从方格的左上角出发，到右下角停止。要求只能往右走或往下走，且一次只能走一步。现在使经过的所有数字的和最大，问最大值是多少？（$1\leqslant m,n\leqslant 1000$）

1	5	9	8	2
8	6	4	7	3
7	2	5	1	6
1	0	1	4	3

【分析】

很容易想到贪心算法，即每次往数值更大的方向走。不过它是错误的。正确的做法是动态规划：

1. **划分阶段**：每走一步为一个阶段。
2. **状态表示**：$f(r,c)$ 表示从起点出发走到第 r 行第 c 列[②]时经过数字总和的最大值。
3. **状态转移方程**：$f(r,c)=\max\begin{cases}f(r-1,c)\\ f(r,c-1)\end{cases}+\text{map}(r,c)$

   边界处理：让方格的下标从 1 开始，这样方格就可以多出一圈——0。

(2) 变式问题

① 必须经过某点

解决方法：将取数过程分成两部分。第一部分的起点是 (1,1)，终点是这个点；第二部分的起点是这个点，终点是 (m,n)（**第 m 行第 n 列**）。比如，要求必须过 (3,2)，就可以按下图把问题一分为二：

1	5	9	8	2

[①] 题目来源：NOIP2006 第一题

[②] 也可以按坐标建立状态，不过它不如按行列适合 CPU 的工作方式。

8	6	4	7	3
7	2	5	1	6
1	0	1	4	3

② 不能经过某点

一个简单的办法是：把那个点的值设置成－INF（**负无穷大**）。由于经过此点要“付出巨大的代价”，所以状态不会从这点转移而来。

③ 竖直方向上最多有连续两个点相邻

解决方法：在状态中加一维，表示“已经连续向下走的次数”。

1．**状态表示**：$f(r,c,i)$ 表示从起点出发走到第 r 行第 c 列时经过数字总和的最大值，i 表示“已经连续向下走的次数”。

2．**状态转移方程**：$f(r,c,0)=\max\begin{cases}f(r,c-1,0)\\f(r,c-1,1)\end{cases}+\mathrm{map}(r,c)$

$f(r,c,1)=f(r-1,c,0)+\mathrm{map}(r,c)$

3．**输出**：$f(m,n,0)$ 和 $f(m,n,1)$ 中的最大值。

④ 传送门#1

到达点 (a,b) 后，便立刻跳转到 (a',b')，其中 $b'>b$（**否则会发生无限传送**）。

解决方法：分两种情况讨论，即经过传送门、不经过传送门。

如下图，传送门的入口为(4,3)，出口为(2,5)，则整个过程可以分为两个子问题。

1	5	9	1	3	8	2
8	6	4	2	-∞	7	3
7	2	5	1	9	1	6
1	0	-∞	8	2	4	3
3	2	7	5	1	6	4
0	5	8	3	4	5	1

不经过传送门

1	5	9	1	3	8	2
8	6	4	2	5	7	3
7	2	5	1	9	1	6
1	0	1	8	2	4	3
3	2	7	5	1	6	4
0	5	8	3	4	5	1

经过传送门

⑤ 传送门#2

到达点 (a,b) 后，便立刻跳转到 (a',b')，反过来也能跳转，其中 $a<a'$，$b'>b$（**否则会发生无限传送**）。

解决方法：分三种情况讨论，即不经过传送门、先经过左侧的传送门、先经过右侧的传送门。

1	5	9	1	3	8	2
8	6	4	2	-∞	7	3
7	2	5	1	9	1	6
1	0	-∞	8	2	4	3
3	2	7	5	1	6	4
0	5	8	3	4	5	1

不经过传送门

1	5	9	1	3	8	2
8	6	4	2	5	7	3
7	2	5	1	9	1	6
1	0	1	8	2	4	3
3	2	7	5	1	6	4
0	5	8	3	4	5	1

先进入左侧的传送门

1	5	9	1	3	8	2
8	6	4	2	5	7	3
7	2	5	1	9	1	6
1	0	1	8	2	4	3
3	2	7	5	1	6	4
0	5	8	3	4	5	1

先进入右侧的传送门

(3) 传纸条[①]

【问题简述】一个 $m\times n$ 的方格，每个格子都有一个数字，但**左上角和右下角都是 0**。现在从方格的左上角引出两条路径，它们同时出发，都到右下角停止[②]。要求只能往右走或往下走，一次只能走一步，且两条路径不能交叉、重合。现在使经过的所有数字的和最大，问最大值是多少？（$1\leqslant m,n\leqslant 50$）

【分析】

如果还使用(1)的动态转移方程，就很有可能在第一条路径选择完成后，第二条路径无法到达终点！所以要同时考虑两条路径。

思路 1：

1. **划分阶段**：每走一步为一个阶段（**一次只移动一张纸条**）。
2. **状态表示**：设两张纸条分别位于第 r_1 行第 c_1 列、第 r_2 行第 c_2 列，那么 $f(r_1,c_1,r_2,c_2)$ 表示两张纸条从起点出发，分别到达这两个点时经过的数字和的最大值。
   为了保证道路不交叉，应有 $x_1<r_2$。
3. **状态转移方程**：$$f(r_1,c_1,r_2,c_2)=\max\begin{cases}f(r_1-1,c_1,r_2,c_2)+\text{map}(r_1,c_1) & (\downarrow\times)\\ f(r_1,c_1-1,r_2,c_2)+\text{map}(r_1,c_1) & (\rightarrow\times)\\ f(r_1,c_1,r_2-1,c_2)+\text{map}(r_2,c_2) & (\times\downarrow)\\ f(r_1,c_1,r_2,c_2-1)+\text{map}(r_2,c_2) & (\times\rightarrow)\end{cases}$$
4. **一个优化**：因为 $r_1+c_1=r_2+c_2$，所以可以去掉一维。这样时间可由四次方降到三次方。
5. **输出**：f(m, n-1, m-1, n)

思路 2：

可以发现，只要知道移动步数和两个横坐标，那么两个纵坐标就可以计算出来。

1. **划分阶段**：每走一步为一个阶段。
   与思路 1 不同的是，这一次两张纸条要同时移动。
2. **状态表示**：设两张纸条分别位于第 r_1 行第 c_1 列、第 r_2 行第 c_2 列，i 为移动步数，那么 $f(i,r_1,r_2)$ 表示两张纸条从起点出发，经过 i 步分别到达这两个点时经过的数字和的最大值。
   为了保证道路不交叉，应有 $r_1>r_2$。
3. **状态转移方程**：$$f(i,r_1,r_2)=\text{map}(r_1,c_1)+\text{map}(r_2,c_2)+\max\begin{cases}f(i-1,r_1-1,r_2-1) & (\downarrow\downarrow)\\ f(i-1,r_1-1,r_2) & (\downarrow\rightarrow)\\ f(i-1,r_1,r_2-1) & (\rightarrow\downarrow)\\ f(i-1,r_1,r_2) & (\rightarrow\rightarrow)\end{cases}$$
   $c_1=i+2-r_1$，$c_2=i+2-r_2$，同时 $r_1+c_1=r_2+c_2$。
4. **输出**：$f(m+n-3,\ m,\ m-1)$
   $m+n-3$ 是从左上角出发，到右下角的移动步数。

6.4　背包问题

参见 83 页“第七单元　背包专题”。

[①] 题目来源：NOIP2008 第三题

[②] 原题是纸条先从左上角走到右下角，然后从右下角再回到左上角。

6.5　编号问题

(1) 最长非降子序列（LIS）

【问题描述】一个序列 a_1，a_2，a_3，…，a_n 共 N 个元素。现在请用动态规划的方法求出从序列找到一个长度最长、且前面一项不大于它后面任何一项的子序列。只需输出序列的长度。$N \leqslant 1000$。

【分析】

1．**递推思路**：$f(i)$ 表示对于前 i 个数组成的序列，保留第 i 个数时能取得的非降子序列的最大长度。

2．**状态转移方程**：$f(i)=\max\{f(j)\}+1$（$a_j>a_i$ 且 $i>j$）

```
int f[1000];
int LIS(int *a, int n)
{
    int max, k;
    f[0]=0;
    for (int i=1;i<=n;i++)
    {
        max=k=0;
        for (int j=i-1;j>0;j--)
            if ((a[j]<a[i])&&(f[j]>=max)) max=f[j], k=j;
        f[i]=f[k]+1;
    }

    max=0;
    for (int i=1;i<=n;i++) if (f[i]>max) max=f[i];
    return max;
}
```

(2) 最长公共子序列（LCS）

【问题描述】有两个序列 a 和 b。求一个最长的序列 p，使它既是 a 的子序列，又是 b 的子序列。输出序列 p 的长度。（a、b 长度小于 1000）

【分析】

1．**状态表示**：$f(i,j)$ 表示 a 的前 i 个元素、b 的前 j 个元素中最长公共子序列的长度。

2．**状态转移方程**：$f(i,j)=\max\begin{cases} f(i-1,j) \\ f(i,j-1) \\ f(i-1,j-1)+(a_i==b_j?1:0) \end{cases}$

```
int f[1001][1001];
int LCS(int *a, int m, int *b, int n)         // a中m个元素，b中n个元素
{
    memset(f,0,sizeof(f));
    for (int i=1; i<=m; i++)
        for (int j=1; j<=n; j++)
        {
            if (a[i]==b[j]) f[i][j]=f[i-1][j-1]+1;
            f[i][j] = max(f[i][j], max(f[i-1][j], f[i][j-1]));
        }
```

```
    return f[m][n];
}
```

6.6　递归结构问题

(1) 乘积最大[①]

【问题简述】在一个长度为 n 的非 0 数字串中插入 k 个乘号，使表达式的值最大。（$6 \leqslant n \leqslant 40$，$1 \leqslant k \leqslant 6$）

【分析】

1. **划分阶段**：以一个乘号为一个阶段。
2. **状态表示**：$f(i,l)$ 表示前 i 个数字插入 l 个乘号之后的最大乘积。
3. **状态转移方程**：$f(i,l) = \max\{f(j, l-1) \times s(j+1, n)\}$，$l<j<i$，$l \leqslant \min\{k, i-1\}$

   边界条件：$f(i,0) = s(1,i)$

   其中 $s(a,b)$ 表示连接第 a 个数字到第 b 个数字之后表示的整数。

```
#include <iostream>
#include <cstring>
using namespace std;
struct hp {……};          // 见131页“11.7  高精度算法（压位存储）!”。
int n, k;
hp f[51][21], s[51][51];

int main()
{
    char c;
    long long t;

    cin>>n>>k;
    memset(f,0,sizeof(f)); memset(s,0,sizeof(s));

    for (int i=1;i<=n;i++)
    {
        cin>>c;
        s[i][i]=c-'0';

        t=1;
        for (int j=i-1;j>0;j--)
            s[j][i]=s[j][j]*(t*=10)+s[j+1][i];       // 递推计算s

        f[i][0]=s[1][i];
    }

    for (int i=1;i<=n;i++)
        for (int l=1; l<=min(i-1, k); l++)
        {
            f[i][l] = 0;
```

[①] 题目来源：NOIP2000 第二题

```
            for (int j=l;j<i;j++) f[i][l] = max(f[i][l], f[j][l-1]*s[j+1][i]);
        }
    cout<<f[n][k]<<endl;
    return 0;
}
```

(2) 加分二叉树[①]

【问题简述】设一个 n 个结点的二叉树的中序遍历为（1,2,3,…,n），其中数字 1,2,3,…,n 为结点编号。每个结点都有一个分数（**均为正整数**），记第 i 个结点的分数为 d_i，二叉树及它的每个子树都有一个加分，任一棵子树（**也包含二叉树本身**）的加分＝左子树的加分×右子树的加分＋根的分数

若某个子树为主，规定其加分为 1，叶子的加分就是叶结点本身的分数。不考虑它的空子树。

求一棵符合中序遍历为（1,2,3,…,n）且加分最高的二叉树。要求输出最高加分和前序遍历。

【分析】

本题中的树是无根树，需要枚举节点作为根的情况，重复有根树的动态规划过程。

1．**状态表示**：$f(i,j)$ 表示由第 i 个元素到第 j 个元素组成的二叉树的最大加分。

2．**状态转移方程**：$f(i,j) = \max\{f(i,k-1)\times f(k+1,j)+d_k\}$，$i\leqslant k\leqslant j$（**实际上，这里的 k 表示根结点**）

边界条件：$f(i,i) = d_i$

3．递推时注意，循环的最外层不是 i，也不是 j，而是 $j-i$！

```
#include <iostream>
#include <cstring>
using namespace std;

int n, root[31][31];
unsigned int f[31][31], d[31];

void preorder(int i, int j)        // 按前序遍历输出最大加分二叉树
{
    int k=root[i][j];
    if (k==0) return;

    cout<<k<<" ";
    preorder(i, k-1);
    preorder(k+1, j);
}

int main()
{
    memset(root, 0, sizeof(root));
    memset(f, 0, sizeof(f));
    memset(d, 0, sizeof(d));
```

[①] 题目来源：NOIP2003 第三题

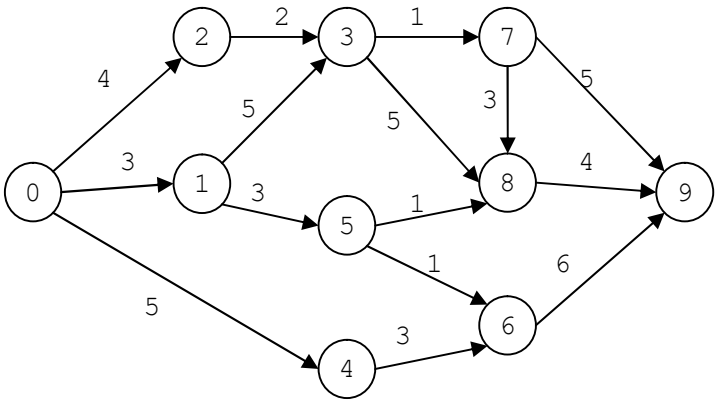
```
    cin>>n;
    for (int i=1; i<=n; i++) cin>>d[i];
    for (int i=0; i<=n; i++)        // 计算单个结点构成的二叉树的加分，并记录根结点
    {
        f[i][i]=d[i];
        root[i][i]=i;
        f[i+1][i]=1;
    }
    for (int p=1; p<n; p++)         // 依次计算间距为d的两个结点构成的二叉树的最大加分
        for (int i=1; i<=n-p; i++)
        {
            int j=i+p;
            for (int k=i; k<=j; k++)
            {
                int temp = f[i][k-1] * f[k+1][j] + d[k];
                if (temp > f[i][j]) f[i][j] = temp, root[i][j] = k;
            }
        }

    cout<<f[1][n]<<endl;
    preorder(1, n);
    return 0;
}
```

6.7　DAG 上的最短路径

如果用动态规划求 DAG 上的最短路径，应该先进行拓扑排序。

(1) 特殊 DAG 的最短路径

【问题描述】下图的拓扑排序序列为 0,1,2……n-1，求结点 0 到其他各点的最短路径长。



邻接矩阵（**未填充部分为∞**）

i\j	0	1	2	3	4	5	6	7	8	9
0		3	4		5					
1				5		3				
2				2						
3								1	5	
4							3			
5							1		1	
6										6
7										5
8										4
9										

【分析】

1．**状态表示**：设 $f(x)$ 表示结点 0 到结点 x 的最短路径长度。

2．**状态转移方程**：$f(x)=\min(f(i)+G[i][x])$，其中结点 i 是结点 x 的前趋。

边界条件：$f(0)=0$

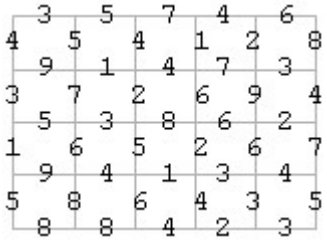
```
int G[N][N], n;
int f[N];

f[0]=0;
for (int i=1; i<n; i++) f[i]=INF;

for (int x=0; x<n; x++)
    for (int i=0; i<n; i++)
        f[x] = min(f[x], f[i]+G[i][x]);

for (int i=0; i<n; i++) cout<<f[i]<<" ";
```

(2) 关键工程

参见 163 页“13.8　拓扑排序”。

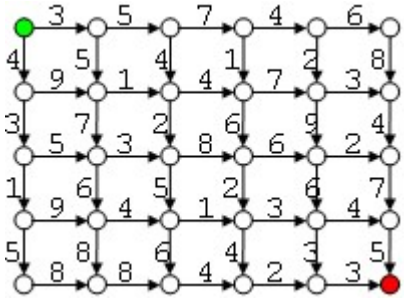
(3) 街道

【问题描述】下图是一个 $m \times n$ 的街区。每条马路（**最短的边算一条马路**）上有一个数字。从左上角出发到右下角，路上只能往右或往下走。问经过的数字的和最大可以达到多少。



【分析】

转化思路：构造出一个 DAG。



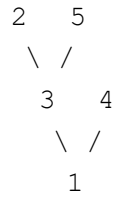
由于这道题道路整齐，所以求解起来更容易一些。

6.8　树形动态规划*

(1) 苹果树

【问题描述】有一棵苹果树，如果树枝有分叉，一定是分 2 叉（**就是说没有只有 1 个儿子的结点**）。这棵树共有 n 个结点（**叶子点或者树枝分叉点**），编号为 1～n，树根编号一定是 1。

我们用一根树枝两端连接的结点的编号来描述一根树枝的位置。下面是一颗有 4 个树枝的树：



现在这颗树枝条太多了，需要剪枝。但是一些树枝上长有苹果。已知需要保留的树枝数量为 q，求出最多能留住多少苹果。

数据规模：$1<n\leqslant 100$，$1\leqslant q\leqslant n$，每个树枝上的苹果数量不超过 30000。

【分析】

1．**状态表示**：$f(i,n)$ 表示在以 i 为根结点的二叉树中保留 n 个树枝后，留住苹果的最大值。

2．**状态转移方程**：$f(i,n)=\max\{f(i\text{ 的左儿子},k)+f(i\text{ 的右儿子},n-k)+\text{i→num}\}$，其中 num 指该结点与父亲结点连接成的树枝上的苹果数，$0\leqslant k\leqslant j$

3．**实现**：由于要把儿子的信息传递给父亲，所以用后序遍历（**下面的代码是后序遍历，虽然看起来不像**）。

```
struct node
{
    int num;                        // 指该结点与父亲结点连接成的树枝上的苹果数
    node *leftchild, *rightchild;
} mem[N];

int postorder(node *i, int a)
{
    if (i==NULL) return 0;
    int result=0;
    for (int k=0; k<=a; k++)
        result = max(result,
            postorder(i->leftchild, k) + postorder(i->rightchild, a-k) + i->num);
    return result;
}
```

(2) 选课[①]

【问题简述】某大学有 m 门功课，每门课有个学分，每门课有一门或没有直接先修课（**若课程 a 是课程 b 的先修课，那么只有学完了课程 a，才能学习课程 b**）。一个学生要从这些课程里选择 n 门课程学习，他能获得的最大学分是多少？

【输入格式】第一行有两个整数 m，n 用空格隔开。（$1\leqslant n\leqslant m\leqslant 1000$）

接下来的 m 行：第 $I+1$ 行包含两个整数 k_i 和 s_i，k_i 表示第 I 门课的直接先修课，s_i 表示第 I 门课的学分。若 $k_i=0$ 表示没有直接先修课（$1\leqslant k_i\leqslant m$，$1\leqslant s_i\leqslant 10$）。

【输出格式】只有一个数，是选 n 门课程的最大得分。

【分析】

1．**状态表示**：$f(i,c)$ 表示在以 i 为根结点的二叉树中取 c 门课程后得到的学分的最大值。

[①] 题目来源：CTSC'97。原题要求输出具体的解。

2．**状态转移方程：**

f(*i*,*c*)＝max{*f*(i->leftchild，*k*－1)＋i->v＋*f*(i->rightchild，*c*－*k*)}，0≤*k*≤*j*

3．**实现：**

- 把 0 作为顶点。
- 需要把多叉树转化为二叉树（**左儿子右兄弟**）。
- 后序遍历

```
#include <iostream>
#include <cstring>
using namespace std;
// -1表示没有结点。
struct node
{
    int value, leftchild, rightchild;
} a[1002];
int F[1002][152], parent[1002];
int m,n;
#define f(x,y) F[(x)+1][(y)+1]

int postorder(int x, int y)
{
    if (f(x,y)>=0) return f(x,y);
    int m=postorder(a[x].rightchild, y);                    // 只有右子树的情况
    for (int k=1; k<=y; k++)
        m = max(m, postorder(a[x].leftchild,k-1) + a[x].value +
                   postorder(a[x].rightchild,y-k));
    return f(x,y)=m;
}

int main()
{
    cin>>m>>n;
    memset(F,0,sizeof(F));
    memset(parent,0,sizeof(parent));
    memset(a,-1,sizeof(a));

    // 树变二叉树
    int k,s;
    for (int i=1; i<=m; i++)
    {
        cin>>k>>s;
        a[i].value=s;
        if (parent[k]==0)
            a[k].leftchild=i;
        else
            a[parent[k]].rightchild=i;
        parent[k]=i;
    }
```

```
    // 递推的边界处理
    for (int i=-1; i<=m; i++)
        for (int j=-1; j<=n; j++)
            f(i,j) = (i==-1 || j==0) ? 0: (-1);

    postorder(a[0].leftchild, n);
    cout<<f(a[0].leftchild,n);
    return 0;
}
```

6.9　状态压缩类问题：过河[①]

【问题简述】一个独木桥，可看做一个数轴，上面每个点的坐标分别为 0、1、2、……、L（$L\leqslant 10^9$）。青蛙从坐标为 0 的点出发，不停地跳跃，直到跳到或超过 L 点。它一次跳跃的距离最小为 S，最大为 T（包括 S、T，$1\leqslant S\leqslant T\leqslant 10$）。

独木桥上有 M（$M\leqslant 100$）个石子，位置都是已知的，并且不会重叠。青蛙讨厌踩到石子上。问：青蛙若想通过独木桥，最少要踩几个石子？

【分析】

很容易想出，若 $f(i)$ 表示从起点到达 i 坐标点所踩到石子的最小个数，则

$f(i)=\min\{f(i-k)\}+f(i)$，$s\leqslant k\leqslant t$

但是，我们无法开长度为 1000000000 的数组，即使能开，程序也不可能在 1s 内结束。

仔细观察数据规模，就会发现，石子的数量非常稀少！所以，长长的空隙一定可以被压短。

以下代码首先对 stone 进行了排序，然后令 L=stone[i]-stone[i-1]。当 L%t==0 时，令 k=t；当 L%t!=0 时，令 k=L%t。然后令 k 为 k+t。

最后判断如果 k>L，那么 map[]数组中 stone[i]和 stone[i-1]两石头的距离就被等效成为 L；如果 k<=L，那么 map[]数组中 stone[i]和 stone[i-1]两石头的距离就被等效成为 k。

接下来就可以用动态规划了。

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

int stone[101], map[100001], f[100001];
int s,t,m,p=0,q;

int main()
{
    int l,k,result;
    memset(stone,0,sizeof(stone));
    memset(map,0,sizeof(map));
    memset(f,0,sizeof(f));

    cin>>l>>s>>t>>m;
    for(int i=1;i<=m;i++) cin>>stone[i];            // 读入石子坐标
    sort(stone+1,stone+m+1);
```

[①] 题目来源：NOIP2005 第二题

```
    for(int i=1;i<=m;i++)                                  // 缩短数组，p为map[]长度
    {
        l=stone[i]-stone[i-1];
        if(l%t==0) k=t; else k=l%t;
        k+=t;
        if(l<k) k=l;
        p+=k;
        map[p]=1;
    }

    for(int i=1;i<=p+t;i++)                                // 动态规划
    {
        result=200;
        for(int j=i-t;j<=i-s;j++)
            if(j>=0 && f[j]<result) result=f[j];
        f[i]=result+map[i];
    }
    result=200;
    for(int i=p+1;i<=p+t;i++) if (f[i]<result) result=f[i];   // 找最小值
    cout<<result;
    return 0;
}
```

6.10　Bitonic 旅行

“货郎担问题”是 NP 问题，只能用搜索解决。后来，J. L. Bentley 提出了的变形——Bitonic Tour 问题（**又称双调旅程问题**）。这个新问题可以用动态规划解决。

【问题描述】已知地图上 n 个旅行须到达城市的坐标，要求从最西端的城市开始，严格地由西向东到最东端的城市，再严格地由东向西回到出发点。除出发点外，每个城市经过且只经过一次。给出路程的最短值。（$1 \leqslant n \leqslant 1000$）

【分析】

可以看出来，如果以城市来表示状态，将与搜索无异！

1. **递推之前的预处理**：将地点按从东到西编号（**按横坐标大小排序，然后扫描**）。
2. **递推思路**：从最东端开始，找两条到最西端的路径。每加入一个地点为一个阶段。
3. **状态表示**：$f(i,j)$ 表示从地点 i 到最东再到地点 j 路程的最小值。假设 i 是走在前面的点，即 $i \geqslant j$。
4. **状态转移方程**：
   - 如果 $i=j$，即 i 和 j 处在同一点，那么 $f(i,j)=f(i,i)=f(i,i-1)+\text{dist}(i,i-1)$
   - 如果 $i=j+1$，即 j 在 i 的紧邻的靠后一点，那么 $f(i,j)=\min\{\ f(j,k)+\text{dist}(i,k)\ \}$，$1 \leqslant k \leqslant j$。
   - 如果 $i>j+1$，即 j 离 i 在后面一个距离的范围以上，那么 $f(i,j)=f(i-1,j)+\text{dist}(i,i-1)$
   - 其中 $\text{dist}(a,b)$ 是 a、b 两点间的距离。
5. **时间复杂度**：$O(n^2)$

```
// 需要：<cmath>
inline int sqr(int x) { return x*x; }
```

```
double x[N], y[N];          // 每个点的坐标。假设x、y已经按照从西向东的顺序排列好。
double f[N][N], dist[N][N];
int n;
double BitonicTour()
{
    // 计算两点间的距离
    for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
        {
            dist[i][j]=sqrt(sqr(X[i]-X[j])+sqr(Y[i]-Y[j]));
            f[i][j]=INF;
        }

    f[0][0]=0.0;
    for (int i=1;i<n;i++)
        for (int j=0;j<i;j++)
            if (i==j+1)
                for (int k=0;k<=j;k++)
                    f[i][j] = min(f[i][j], f[j][k]+dist[k][i]);
            else if (j<i-1)
                f[i][j]=f[i-1][j]+dist[i][i-1];
            else
                continue;
    return f[n-1][n-2];
}
```

6.11　小结

动态规划：各个阶段采取的决策依赖于当前状态，又随即引起状态的转移，在变化的状态中产生一个决策序列，这种解决多阶段决策最优化问题的方法为动态规划方法。

1．动态规划的基本思想

动态规划的基本思想是“空间换时间”，努力避免重复解决相同问题或子问题。由于这个原因，动态规划的速度比搜索快得多。

2．动态规划的原理

动态规划满足两个原理：最优化原理、无后效性原则。

最优化原理指无论过去的状态和决策如何，对前面的决策所形成的当前状态而言，余下的诸决策必须构成最优策略。通俗地说，如果全局最优，那么一定有局部最优。如果问题满足最优化原理，则说该问题具有最优子结构。

无后效性原则指某阶段的状态一旦确定，则此后过程的演变不再受此前各状态及决策的影响。也就是说，“未来与过去无关”。具体地说，如果一个问题被划分各个阶段之后，阶段 I 中的状态只能由阶段 $I+1$ 中的状态通过状态转移方程得来，与其他状态没有关系，特别是与未发生的状态没有关系，这就是无后效性。

3．用动态规划解题的方法

动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。这些决策形成了一个决策序列，同时确定了完成整个过程的一条活动路线(通常是求最优的活动路线)。如图所示。动态规划的设计都有着一定的模式，一般要经历以下几个步骤。

初始状态→决策 1→决策 2→…→决策 n→结束状态

① 划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。

② 确定状态和状态变量：注意状态必须满足无后效性。

③ 确定决策：找到子问题是进行动态规划的重要一步。动态规划和递推更多应考虑本问题由哪些已解决子问题构成，而不是考虑本问题对将来哪些问题有贡献。

④ 确定边界条件，写出状态转移方程。

⑤ 编程

如果某一个问题是从已知的动态规划问题变形而来的，你可以考虑在原有的状态转移方程的基础上加一维。

4．动态规划的实现

动态规划有两种实现方法：递推和记忆化搜索。

使用递推法时要注意计算顺序，题型不同，计算顺序也不同。如果计算顺序不明显，也可采用记忆化搜索。但是，记忆化搜索要比直接递推慢几倍。

动态规划往往要耗费大量内存空间。在空间紧张时，可以采用滚动数组来优化动态规划。

第七单元　背包专题[①]

7.1　部分背包问题

参见 49 页“4.1　装载问题”。部分背包问题是贪心算法问题，其他背包问题都是动态规划问题。

7.2　0/1 背包问题！

【问题描述】有 n 件物品和一个容量为 C 的背包。第 i 件物品的重量是 $w[i]$，价值是 $v[i]$。求解将哪些物品装入背包可使价值总和最大。

(1) 二维数组表示

1. **定义状态**：$f[i][c]$ 表示前 i 件物品恰放入一个容量为 c 的背包可以获得的最大价值。

2. **状态转移方程**：$f[i][c]=\max\begin{cases}f[i-1][c] & \text{(不选这件物品)}\\ f[i-1][c-w[i]]+v[i] & \text{(选择这件物品)}\end{cases}$

```
// 注意边界处理
for (int i=1;i<=n;i++)
    for (int c=0;c<=C;c++)
    {
        f[i][c]=f[i-1][c];
        if (c>=w[i]) f[i][c] = max(f[i][c], f[i-1][c-w[i]] + v[i]);
    }
```

时间复杂度、空间复杂度：都是 $O(NC)$

(2) 一维数组表示

1. **定义状态**：由于递推的过程和雪天环形路上的扫雪车类似，所以可以把 i 省略。

2. **状态转移方程**：$f[c]=\max\begin{cases}f[c] & \text{(不选这件物品)}\\ f[c-w[i]]+v[i] & \text{(选择这件物品)}\end{cases}$

递推时注意 c 要从 C 开始，倒着推。

```
// 注意边界处理
for (int i=1;i<=n;i++)
    for (int c=C;c>=0;c--)
        if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + v[i]);
```

时间复杂度：$O(NC)$

空间复杂度：降到了 $O(C)$

(3) 一维之下的一个常数优化

其实没有必要让循环下限为 0。

```
int bound, sumw=0;
for (int i=1;i<=n;i++)
{
    sumw+=w[i];
```

[①] 本单元内容来自 dd_engi 的《背包九讲》。关于背包问题状态转移方程的具体解释可以在这本书中找到。

```
    bound = max(C - sumw, w[i]);
    for (int c=C;c>=bound;c--)
        if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + v[i]);
}
```

(4) 初始化时的细节

如果要求“恰好装满”，那么初始化时应该让$f[0]=0$,其他的$f[i]=-INF$。这样就可以知道是否有解了。如果不用“恰好”，那么应该让f的所有元素都置0。

7.3　完全背包问题

【问题描述】有n**种**物品和一个容量为C的背包。第i**种**物品的重量是$w[i]$，价值是$v[i]$，**数量无限**。求解将哪些物品装入背包可使价值总和最大。

(1) 基本算法

1. **状态转移方程**：$f[i][c]=\max\{f[i-1][c-k\times w[i]]+k\times v[i]\}$，$0\leqslant k\times w[i]\leqslant c$

   时间复杂度$O(C\times\sum\frac{C}{w[i]})$，比较大。

2. **一个简单的优化**：如果物品i、j满足$w[i]\leqslant w[j]$且$v[i]\geqslant v[j]$，就可以把物品j去掉。

   不过它不能改善最坏情况的复杂度（**最坏情况**：**根本没有可以去掉的东西**）。

3. **另一种优化**：首先将重量大于C的物品去掉，然后使用类似桶排序或计数排序的方法，计算出费用相同的物品中哪个价值最高。

(2) 更优的算法

```
// 内层的for和外层的for可以互换。
for (int i=1;i<=n;i++)
    for (int c=0;c<=C;c++)                  // 这里发生了变化——循环次序变了
        if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + v[i]);
```

时间复杂度：$O(NC)$

转化为二维，状态转移方程就是：$f[i][c]=\max\begin{cases}f[i-1][c]\\f[i][c-w[i]]+v[i]\end{cases}$　（**第二行变了**）

7.4　多重背包问题

【问题描述】有n种物品和一个容量为C的背包。第i种物品的重量是$w[i]$，价值是$v[i]$，**数量为$a[i]$**。求解将哪些物品装入背包可使价值总和最大。

【分析】

二进制法：按照二进制分割物品。比方说，物品i有13个，就可以把它分成系数为1、2、4、6，共4个0/1背包的物品。（$13=2^0+2^1+2^2+6$）

```
for (int i=1;i<n;i++)
{
    if (w[i]*a[i]>C)                        // 如果物品够多，问题其实就是完全背包问题
```

```
    {
        for (int c=0;c<=C;c++)          // 完全背包
            if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + v[i]);
    }
    else
    {
        int k=1, amount=a[i];
        while (k<amount)
        {
            // 是否取一个重量为k×w[i]，价值为k×v[i]的物品？
            for (int c=k*w[i];c>=0;c--)
                if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + k*v[i]);

            // 继续分割
            amount-=k;
            k+=k;
        }
        // 把剩下的作为单独一个物品
        for (int c=amount*w[i];c>=0;c--)
            if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + amount*v[i]);
    }
}
```

时间复杂度：$O(V\times\sum\log w[i])$

7.5　二维费用的背包问题

【问题描述】有 n 件物品和一个容量为 C、**容积为 U** 的背包。第 i 件物品的重量是 $w[i]$，**体积是 $u[i]$**，价值是 $v[i]$。求解将哪些物品装入背包可使价值总和最大。

(1) 0/1 背包的表示方法

费用加了一维，只需把状态也加一维。

1. **状态表示**：设 $f[i][c][u]$ 为前 i 件物品付出两种代价分别为 c 和 u 时可以获得的最大价值。
2. **状态转移方程**：$f[i][c][u]=\max\begin{cases}f[i-1][c][u]\\f[i-1][c-w[i]][u-u[i]]+v[i]\end{cases}$

   当然，为了节省空间，可以把 i 去掉。
3. **一个启示**：当发现由熟悉的动态规划题目变形而来的题目时，在原来的状态中加一维以满足新的限制，这是一种比较通用的方法。

(2) 限制物品总个数的 0/1 背包

【问题描述】有 n 件物品和一个容量为 C 背包。第 i 件物品的重量是 $w[i]$，价值是 $v[i]$。现在要求**转入背包的物品个数不超过 M**。求解将哪些物品装入背包可使价值总和最大。

只需把问题变一下——有 N 件物品和一个容量为 C、容积为 M 的背包。第 i 件物品的重量是 $w[i]$，体积是 1，价值是 $v[i]$。求解将哪些物品装入背包可使价值总和最大。

把最大个数看做一种容积就可以了。

(3) 二维费用的完全背包和多重背包问题

循环时仍然按照完全背包（**顺序循环**）和多重背包（**分割**）的方法操作，只不过比完全背包和多重背包多了一维。

(4) 二维费用背包的另一种解法

把背包的容量和费用看作一个复数。详见《背包九讲》。

7.6　分组的背包问题

【问题描述】有 n 件物品和一个容量为 C 的背包。第 i 件物品的重量是 $w[i]$，价值是 $v[i]$。**这些物品被划分为 K 组，每组中的物品互相冲突，最多选一件。**求解将哪些物品装入背包可使价值总和最大。

1. **状态表示**：设 $f[k][c]$ 为前 k **组**物品花费 c 时可以获得的最大价值。
2. **状态转移方程**：$f[k][c]=\max\begin{cases}f[k-1][c]\\ f[k-1][c-w[i]]+v[i] & \text{物品}i\text{属于第}k\text{组}\end{cases}$

注意循环的顺序。

```
for (int k=1;i<=K;i++)
    for (int c=C;c>=0;c--)
        for each (int i in 第k组)        // 伪代码，指“for (所有属于组k的物品i)”
            if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + v[i]);
```

在“金明的预算方案”（NOIP2006,2）中，就可以把主件、附件组合成一个分组背包（**一组最多 4 个物体，最少 1 个物体**）。

7.7　有依赖的背包问题

【问题描述】依赖关系以图论中“森林”的形式给出，也就是说，主件的附件仍然可以具有自己的附件集合，限制只是每个物品最多只依赖于一个物品（**只有一个主件**）且不出现循环依赖。

1. **第一种思想**：将每个主件及其附件集合转化为物品组。不过，由于附件可能还有附件，就不能将每个附件都看作一个一般的 01 背包中的物品了。若这个附件也有附件集合，则它必定要被先转化为物品组，然后用分组的背包问题，解出主件及其附件集合所对应的附件组中各个费用的附件所对应的价值。
2. **第二种思想**：每个父结点都需要对它的各个儿子的属性进行一次 DP 以求得自己的相关属性。
3. **第三种思想**：这已经触及到了“泛化物品”的思想，你会发现这个“依赖关系树”每一个子树都等价于一件泛化物品，求某结点为根的子树对应的泛化物品相当于求其所有儿子的对应的泛化物品之和。

7.8　泛化物品

有这样一种物品，名字叫做泛化物品。有一个函数 $h(x)$，当分配给物品的费用为 a 时，能得到的价值就是 $h(a)$。

实际上，前面总结的所有背包都可以看做泛化物品，只不过在某些时候 h 的值为 0。

如果面对两个泛化物品 h 和 l，要用给定的费用从这两个泛化物品中得到最大的价值，怎么求呢？事实上，对于一个给定的费用 v，只需枚举将这个费用如何分配给两个泛化物品就可以了。同样的，对于 0～C 的每一个整数 c，可以求得费用 c 分配到 h 和 l 中的最大价值 $f(v)$：

$$f(v)=\max\{h(k)+l(c-k)\},\ 0\leqslant k\leqslant c。$$

可以看到，f 也是一个由泛化物品 h 和 l 决定的定义域为 0～C 的函数，也就是说，f 是一个由泛化物品 h 和 l 决定的泛化物品，f 是 h 与 l 的和。这个运算的时间复杂度取决于背包的容量，是 $O(V^2)$。

泛化物品的定义表明：在一个背包问题中，若将两个泛化物品代以它们的和，不影响问题的答案。事实上，对于其中的物品都是泛化物品的背包问题，求它的答案的过程也就是求所有这些泛化物品之和的过程。设此和为 s，则答案就是 $s[V]$ 中的最大值。

一般而言，求解背包问题，即求解这个问题所对应的一个函数，即该问题的泛化物品。而求解某个泛化物品的一种方法就是将它表示为若干泛化物品的和然后求之。

7.9　混合背包问题

【问题描述】还是背包问题，不过有的物品只能取一次（0/1 **背包**），有的可以取无限次（**完全背包**），有的只能取有限次（**多重背包**）。怎么解决？

由于我们按物品划分阶段，所以没有必要想太多。下面给出伪代码：

```
for (int i=1; i<N; i++)                    // 不变
    if (物品i属于0/1背包)
    {
        // 按照0/1背包的做法取物品i
        for (int c=C;c>=0;c--)
            if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + v[i]);
    }
    else if (物品i属于完全背包)
    {
        // 按照完全背包的做法取物品i
        for (int c=0;c<=C;c++)
            if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + v[i]);
    }
    else if (物品i属于多重背包)
    {
        // 按照多重背包的做法取物品i
        ……
    }
```

7.10　特殊要求

(1) 输出字典序最小的最优方案

这里“字典序最小”的意思是 1～N 号物品的选择方案排列出来以后字典序最小。

一般而言，求一个字典序最小的最优方案，只需要在转移时注意策略：以 0/1 背包为例。在某一阶段，两种决策的结果相同，就应该按照前者来输出方案。

(2) 求方案总数

以 0/1 背包为例。它的状态转移方程为 $f[i][c]=\max\begin{cases}f[i-1][c] & ①\\ f[i-1][c-w[i]]+v[i] & ②\end{cases}$

除了可以求可得到的最大价值外，还可以得到装满背包或将背包装至某一指定容量的方案总数。

只需把状态转移方程中的 max 改成 sum(**求和**)，并将初始条件设为 $f[0][0]=1$，就可以通过 $f[n-1][C]$ 求出方案总数。

事实上，这样做可行的原因在于**状态转移方程已经考察了所有可能的背包组成方案**。

(3) 最优方案的总数

这里的最优方案是指物品总价值最大的方案。以 01 背包为例。

结合求最大总价值和方案总数两个问题的思路，最优方案的总数可以这样求：开一个数组 $g[i][c]$，表示这个子问题的最优方案的总数。在求 $f[i][c]$ 的同时，顺便就把 $g[i][c]$ 带下来了。

代码如下：

```
for (int i=1;i<=n;i++)
    for (int c=0;c<=C;c++)
    {
        f[i][c]=f[i-1][c];
        if (c>=w[i]) f[i][c] = max(f[i][c], f[i-1][c-w[i]] + v[i]);
        g[i][c]=0;

        if (f[i][c]==f[i-1][c])
            g[i][c] += g[i-1][c];
        if (f[i][c] == (f[i-1][c-w[i]] + v[i]))
            g[i][c] += g[i-1][c-w[i]] + v[i];
        // 注：如果两个状态的值相等，那么计算g时应该把两部分都算进去。这就是必须单独求g的原因。
    }
```

(4) 求次优解、第 K 优解

对于求次优解、第 K 优解类的问题，如果相应的最优解问题能写出状态转移方程、用动态规划解决，那么求次优解往往可以相同的复杂度解决，第 K 优解则比求最优解的复杂度上多一个系数 K。

其基本思想是将每个状态都表示成有序队列，将状态转移方程中的 max/min 转化成有序队列的合并。

以 0/1 背包为例。0/1 背包的状态转移方程为 $f[i][c]=\max\begin{cases}f[i-1][c] & ①\\ f[i-1][c-w[i]]+v[i] & ②\end{cases}$ 如果要求第 K 优解，那么状态 $f[i][c]$ 就应该是一个大小为 K 的数组 $f[i][c][K+1]$。其中 $f[i][c][k]$ 表示前 i 个物品、背包大小为 c 时，第 k 优解的值。显然可以认为 $f[i][c][K+1]$ 是一个有序队列。

然后可以说：$f[i][c]$ 这个有序队列是由①、②这两个有序队列合并得到的。有序队列①即 $f[i-1][c][K]$，②则理解为在 $f[i-1][c-w[i]][K]$ 的每个数上加上 $v[i]$ 后得到的有序队列。合并这两个有序队列并将结果的前 K 项储存到 $f[i][c][K+1]$ 中的复杂度是 $O(K)$。最后的答案是 $f[N][C][K]$。总的复杂度是 $O(CNK)$。

实际上，一个正确的状态转移方程的求解过程已经覆盖了问题的所有方案。只不过由于是求最优解，所以其它达不到最优的方案都被忽略了。因此，上面的做法是正确的。

另外，要注意题目对于“第 K 优解”的定义，将策略不同但权值相同的两个方案是看作同一个解还是不同的解。如果是前者，则维护有序队列时要保证队列里的数没有重复的。

7.11　背包问题的搜索解法

(1) 代码

对于 0/1 背包问题，简单的深搜的复杂度是 O(2^n)。就是枚举出所有 2^n 种将物品放入背包的方案，然后找最优解。代码如下（调用 try(1,0,0)即可）：

```
int max=0;
void try(int i, int curw, int curv)   // i是当前物体，curw是当前重量，curv是当前的价值
{
    if (i>n)
    {
        if (curv > max) max = curv;
        return;
    }

    if (curw + w[i] <= C) try(i+1, curw + w[i], curv + v[i]);
    try(i+1, curw, curv);
}
```

(2) 预处理和剪枝

预处理：在搜索中，可以认为顺序靠前的物品会被优先考虑。

所以，可以利用贪心的思想，将更有可能出现在结果中的物品的顺序提前，可以较快地得出贪心的较优解，也更有利于最优性剪枝。这需要按照“性价比”（**权值/费用**）来排列搜索顺序。

另一方面，若将费用较大的物品排列在前面，可以较快地填满背包，有利于可行性剪枝。

最后一种可以考虑的方案是：在开始搜索前将给定的物品的顺序打乱。这样可以避免命题人设置的陷阱。

可行性剪枝：即判断按照当前的搜索路径搜下去能否找到一个可行解，例如：若将剩下所有物品都放入背包仍然无法将背包充满（**设题目要求必须将背包充满**），则剪枝。

最优性剪枝：即判断按照当前的搜索路径搜下去能否找到一个最优解，例如：若加上剩下所有物品的权值也无法得到比当前得到的最优解更优的解，则剪枝。

(3) 搜索还是 DP

在看到一道背包问题时，应该用搜索还是动态规划呢？

如果一个背包问题可以用动态规划求解，V 一定不能很大，否则 O(VN) 的算法无法承受，而一般的搜索解法都是仅与 N 有关，与 V 无关。

所以，V 很大时，应该优先考虑搜索；N 较大时，应该优先考虑动态规划。

另外，当想不出合适的动态规划算法时，就只能用搜索了。

7.12　子集和问题

【问题描述】给定一个整数的集合 S 和一个整数 X，问是否存在 S 的一个子集满足其中所有元素的和为 X。

【分析】

子集和问题是一个 NPC 问题，与 0/1 背包问题并不相同。

这个问题有一个时间复杂度为 $O(2^{N/2})$ 的较高效的搜索算法，其中 N 是集合 S 的大小。

第一步思想是二分。将集合 S 划分成两个子集 S_1 和 S_2，它们的大小都是 $N/2$。对于 S_1 和 S_2，分别枚举出它们所有的 $2^{N/2}$ 个子集和，保存到某种支持查找的数据结构中（**如散列表**）。然后就要将两部分结果合并，寻找是否有和为 X 的 S 的子集。

事实上，对于 S_1 的某个和为 X_1 的子集，只需寻找 S_2 是否有和为 $X-X_1$ 的子集。假设采用的散列表是理想的，每次查找和插入都仅花费 $O(1)$ 的时间。两步的时间复杂度显然都是 $O(2^{N/2})$。

实践中，往往可以先将第一步得到的两组子集和分别排序，然后再用两个指针扫描的方法查找是否有满足要求的子集和。这样的实现，在可接受的时间内可以解决的最大规模约为 $N=42$。

第八单元　排序算法

8.1　常用排序算法

(1) 使用 STL 算法!

平均时间：O($n$log$n$)

头文件：<algorithm>

① 调用方法：sort(第一项的地址，最后一项的地址+1);

如 sort(&a[0],&a[n]);或 sort(a,a+n);

注意，STL 的区间是左闭右开区间。

② 自定义规则的排序：有时排序的条件不止一个，或不方便对原数据进行排序，就需要自定义比较规则。这时需要建立一个函数，把“小于”解释明白。例如：

```
bool cmp(const int &i, const int &j) {return w[i]<w[j];}      // 自定义比较规则
……
sort(a, a+n, cmp);
```

cmp 要讲清 a[i]和 a[j]的比较方法。对于上面的代码，就是“如果 w[i]<w[j]，那么 a[i]就排在 a[j]的前面”。

(2) 快速排序!

平均时间：O($n$log$n$)

快速排序俗称“快排”，是基于比较的排序算法中最快的一种算法。

```
void quicksort(int *a, int start, int end)
{
    int low=start, high=end;
    int temp, check=a[start];

    // 划分：把比check小的数据放到它的左边，把比check大的数放到它的右边。
    do
    {
        while (a[high]>=check&&low<high) high--;    // 注意，不要写成“low<=high”!
        if (a[high]<check)
            temp=a[high], a[high]=a[low], a[low]=temp;

        while (a[low]<=check&&low<high) low++;      // 注意，不要写成“low<=high”!
        if (a[low]>check)
            temp=a[high], a[high]=a[low], a[low]=temp;
    }
    while (low!=high);

    a[low]=check;
    low--,high++;

    // 递归：对已经划分好的两部分分别进行快速排序
    if (low>start) quicksort(a, start, low);
```

```
    if (high<end) quicksort(a, high, end);
}
```

快速排序的版本很多，上面只是众多版本中的一种。

快速排序的三个优化方法：

1．规模很小时（**如** end－start＜10），使用插入排序代替快速排序。

2．使用栈模拟递归。

3．极端数据（**如比较有序的数组**）会使快速排序变慢，甚至退化为冒泡排序。可以采用“三者取中法”来解决这个问题：令 check 等于 a[start]、a[end]、a[(start+end)/2]中的中间值。

第三种方法可以消除坏数据（**基本有序的数据，它可以使快速排序退化为** $O(n^2)$ **时间**）对排序的影响。

(3) 归并排序

时间复杂度：$O(n\log n)$

注意：

1．其他排序算法的空间复杂度是 O(1)，而归并排序的空间复杂度很大，为 $O(n)$。

2．下面的 end 是“末尾索引＋1”，即数列末尾要留一个空位。

```
int temp[N];
void mergesort(int *a, int start, int end)
{
    if (start+1>=end) return;
    // 划分阶段、递归
    int mid = start+(end-start)/2;
    mergesort(a, start, mid);
    mergesort(a, mid, end);

    // 将mid两侧的两个有序表合并为一个有序表。
    int p=start,q=mid,r=start;
    while (p<mid || q<end)
        if (q>=end || (p<mid && a[p]<=a[q]))
            temp[r++]=a[p++];
        else
            temp[r++]=a[q++];

    for (p=start;p<end;p++) a[p]=temp[p];
}
```

在(end-start)不太大时，可以用插入排序代替归并排序。

归并排序还有另一种写法：开始复制的时候，把第二个子数组中元素的顺序颠倒了一下。

```
int temp[N];                          // “临时安置点”
void mergesort(int *a, int start, int end)
{
    if (start==end) return;

    int mid = start+(end-start)/2;
```

```
    mergesort(a, start, mid);
    mergesort(a, mid, end);

    // 合并
    for (int i=mid; i>=start; i--) temp[i]=a[i];
    for (int j=1;j<=end-mid;j++) temp[end-j+1]=a[j+mid];

    for (int i=start,j=end,k=start; k<=end; k++)
        if (temp[i]<temp[j])
            a[k]=temp[i++];
        else
            a[k]=temp[j--];
}
```

在(end-start)不太大时，也可以用插入排序代替归并排序。

8.2　简单排序算法

以下三种排序算法的时间复杂度是O(n^2)。在这三种排序算法中，比较好的是插入排序。

(1) 插入排序！

基本思想：假设前 i 个元素已经排好序，然后把第 $i+1$ 个元素插入到合适的位置上。

```
void ins_sort(int *a, int start, int end)
{
    int j,k;
    for (int i=start+1; i<end; i++)
    {
        for (k=a[i],j=i-1; k<a[j] && j>=start; j--)
            a[j+1]=a[j];
        a[j+1]=k;
    }
}
```

(2) 选择排序！

基本思想：处理第 i 个元素时，找到它后面所有数的最小值。如果这个最小值比它小，就互相交换。

```
void swap_sort(int *a, int start, int end)
{
    int temp;
    for (int i=start; i<end; i++)
        for (int j=i+1; j<end; j++)
            if (a[i]>a[j])
                temp=a[i],a[i]=a[j],a[j]=temp;
}
```

(3) 冒泡排序！

基本思想：对相邻两个元素进行比较，并将较小的数放到前面。重复 n 组即可完成排序。

```
void bubble_sort(int *a, int start, int end)
```

```
{
    int temp;
    for (int i=start; i<end; i++)
        for (int j=end; j>i; j--)   // 从start到i，在“冒泡”过程中，元素已经排好序。
            if (a[j-1]>a[j])
                temp=a[j-1], a[j-1]=a[j], a[j]=temp;
}
```

8.3　线性时间排序

以下几种排序算法的时间复杂度为$O(n)$，因为它们使用的是基于数字本身的排序。

(1) 桶排序!

基本思想：设计M个桶，根据元素本身进行分配。因为M个桶是按顺序排列的，所以分配元素之后元素也会按顺序排列。

待排序的数据**范围不太大**时可以考虑这样排序**(比如**1~1000**可以考虑,而**1~100000**就必须用快速排序)**。

下面假设每个数都位于区间[1, M]。

```
int cnt[M];

memset(cnt,0,sizeof(cnt));
for (int i=0; i<N; i++)
    cnt[a[i]]++;

// 从cnt[0]开始挨个列举就行了。例如
for (int i=0; i<M; i++)
    while ((cnt[i]--)>0)
        cout<<i<<" ";
```

(2) 计数排序

基本思想:对于序列中的每一元素x,确定序列中小于x的元素的个数。下面假设每个数都位于区间[1, m]。

```
int a[N], b[N], cnt[M];
……
memset(cnt,0,sizeof(cnt));
for (int i=0;i<n;i++) cnt[a[i]]++;
for (int i=1;i<M;i++) cnt[i]+=cnt[i-1];
for (int i=n-1;i>0;i--)
{
    b[cnt[a[i]]] = a[i];
    cnt[a[i]]--;
}
// 输出结果
for (int i=0;i<n;i++) cout<<b[i]<<' ';
```

(3) 基数排序*

基本思想：对n个元素依次按k,k-1,...1位上的数字进行桶排序。

```
int tmp[N], cnt[10];
int n;
int maxbit(int *data, int n)                        // 辅助函数，求数据的最大位数
{
    int d = 1;                                      // 保存最大的位数
    int p =10;
    for (int i = 0;i < n; i++)
        while (data[i] >= p)
            p *= 10, d++;
    return d;
}

void radixsort(int *data, int n)                    // 基数排序
{
    int d = maxbit(data,n);
    int i,j,k;
    int radix = 1;
    for (i = 1; i<= d;i++)                          // 进行d次排序
    {
        for (j = 0;j < 10;j++)
            cnt[j] = 0;                             // 每次分配前清空计数器
        for (j = 0;j < n; j++)
        {
            k = (data[j]/radix)%10;                 // 统计每个桶中的记录数
            cnt[k]++;
        }
        for (j = 1;j < 10;j++)
            cnt[j] = cnt[j-1] + cnt[j];             // 将tmp中的位置依次分配给每个桶
        for (j = n-1;j >= 0;j--)                    // 将所有桶中记录依次收集到tmp中
        {
            k = (data[j]/radix)%10;
            cnt[k]--;
            tmp[cnt[k]] = data[j];
        }
        for (j = 0;j < n;j++)                       // 将临时数组的内容复制到data中
            data[j] = tmp[j];
        radix = radix*10;
    }
}
```

8.4　使用二叉树的排序算法*

(1) 二叉树排序

有关二叉排序树的代码参见 113 页“10.4　二叉排序树”。

二叉树排序的步骤很简单，就是先把每个元素插入到 BST 中，然后中序遍历。

时间复杂度：O($n$log$n$)

应该清楚地认识到，因为二叉排序树很容易变得不平衡，并且它的空间占用比较大，插入结点也要花费一

些时间，所以，二叉树排序比快速排序慢很多。

使二叉排序树不平衡的方法：数据基本有序，BST 退化成长长的链表；或数据使 SBT 形成堆积的“人”字形。

(2) 堆排序

有关堆的代码参见 115 页“10.5　堆和优先队列*”。使用最大值堆，因为这样做可以不占用额外的空间。堆排序的步骤也不难。操作方法如下：

① 将整个数组转化为一个堆（**使用** buildheap **完成**）。

② 将堆顶的最大元素取出（removemax），并把它放到数组的最后（**准确的说，位置是堆中元素个数减** 1）。

③ 剩余元素重新建堆。

④ 重复②，直到堆为空。

时间复杂度：$O(n\log n)$

堆排序与其他 $O(n\log n)$ 的排序算法相比要慢很多。堆排序适用于寻找第 k 大（小）元素。

8.5　小结

搜索算法的比较：

1．稳定性

插入排序、冒泡排序、二叉树排序、归并排序及其他线性时间排序是稳定的；

选择排序、希尔排序（《**资料**》**里没有总结**）、快速排序、堆排序是不稳定的。

2．时间复杂度

插入排序、冒泡排序、选择排序的时间复杂度为 $O(n^2)$；

其它非线性时间排序的时间复杂度为 $O(n\log n)$；

线性时间排序的时间复杂度为 $O(n)$。

3．辅助空间

线性时间排序、归并排序的辅助空间为 $O(n)$，其它排序的辅助空间为 $O(1)$。

4．其它方面

插入、冒泡排序的速度较慢，但参加排序的序列局部或整体有序时，这种排序能达到较快的速度。在这种情况下，快速排序反而慢了。

当 n 较小时，对稳定性不作要求时宜用选择排序，对稳定性有要求时宜用插入或冒泡排序。若待排序的记录的关键字在一个明显有限范围内时，且空间允许是用桶排序。

当 n 较大时，关键字元素比较随机，对稳定性没要求宜用快速排序。

当 n 较大时，关键字元素可能出现本身是有序的，对稳定性有要求时，空间允许的情况下，宜用归并排序。

当 n 较大时，关键字元素可能出现本身是有序的，对稳定性没有要求时宜用堆排序。

第九单元　基本数据结构

9.1　线性表（顺序结构）

线性表可以用普通的一维数组存储。

你可以让线性表可以完成以下操作（**代码实现很简单，这里不再赘述**）：

1．返回元素个数。
2．判断线性表是否为空。
3．得到位置为 p 的元素。
4．查找某个元素。
5．插入、删除某个元素：务必谨慎使用，因为它们涉及大量元素的移动。

9.2　线性表（链式结构）

(1) 单链表！

1．定义：下面有一个空链表，表头叫 head，并且表内没有任何元素。

```
struct node
{
    int value;
    node *next;
} arr[MAX];
int top=-1;
node *head = NULL;
```

2．内存分配：在竞赛中不要用 new，也不要用 malloc、calloc——像下面一样做吧。

```
#define NEW(p)       p=&arr[++top];p->value=0;p->next=NULL
node *p;
NEW(head);                          // 初始化表头
NEW(p);                             // 新建结点
```

3．插入：把 q 插入到 p 的后面。时间复杂度 O(1)。

```
if (p!=NULL && q!=NULL)             // 先判定是否为空指针。如果不是，继续。
{
    q->next=p->next;
    p->next=q;
}
```

4．删除：把 p 的下一元素删除。时间复杂度 O(1)。

```
if (p!=NULL && p->next!=NULL)       // 先判定是否为空指针。如果不是，继续。
{
    node *q=p->next;
    p->next=q->next;
    // delete(q);                   // 如果使用动态内存分配，最好将它的空间释放。
}
```

5．查找或遍历：时间复杂度 O(n)。

```
node *p=first;
while (p!=NULL)
{
```

```
    // 处理value
    // cout<<p->value<<'\t';
    p=p->next;
}
```

(2) 静态链表

指针的作用就是存储地址。如果我们找到了替代品，就可以放弃指针了。

需要把上面的定义改一下：

```
struct node
{
    int value;
    int next;         // 表示下一元素在arr中的下标
} arr[MAX];
```

(3) 循环链表

和单链表有一个重大区别：单链表最后一个元素的 next 指向 NULL，而循环链表最后一个元素的 next 指向 first。

遍历时要留心，不要让程序陷入死循环。

一个小技巧：如果维护一个表尾指针 last，那么就可以在 O(1)的时间内查找最后一个元素。同时可以防止遍历时陷入死循环。

(4) 双链表

1. 定义：下面有一个空链表，表头叫 first。

```
struct node
{
    int value;
    node *next, *prev;
} arr[MAX];
int top=-1;
node *first = NULL;  // 根据实际需要可以维护一个表尾指针last。
```

2. 内存分配：最好不要使用 new 运算符或 malloc、calloc 函数。

```
#define NEW(p)       p=arr+(++top);p->value=0;p->next=NULL;p->prev=NULL
node *p;
NEW(head);                          // 初始化表头
NEW(p);                             // 新建结点
```

3. 插入：把 q 插入到 p 的后面。时间复杂度 O(1)。

```
if (p==NULL||q==NULL)               // 先判定是否为空指针。如果不是，继续。
{
    q->prev=p;
    q->next=p->next;
    q->next->prev=q;
    p->next=q;
}
```

4. 删除：把 p 的下一元素删除。时间复杂度 O(1)。

```
if (p==NULL||p->next==NULL)         // 先判定是否为空指针。如果不是，继续。
```

```
{
    node *q=p->next;
    p->next=q->next;
    q->next->prev=p;
    // delete(q);                    // 如果使用动态内存分配，最好将它的空间释放。
}
```

5．查找或遍历：从两个方向开始都是可以的。

(5) 将元素插入到有序链表中*

```
void insert(const node *head, node *p)
{
    node *x, *y;
    y=head;
    do
    {
        x=y;
        y=x->next;
    } while ((y!=NULL) && (y->value < p->value);
    x->next=p;
    p->next=y;
}
```

9.3 栈

(1) 栈的实现!

操作规则：先进后出，先出后进。

1．`int stack[N], top=0;`　　// top 表示栈顶位置。

2．**入栈**：`inline void push(int a) { stack[top++]=a; }`

3．**出栈**：`inline int pop() { return stack[--top]; }`

4．**栈空的条件**：`inline bool empty() { return top<0; }`

如果两个栈有相反的需求，可以用这种方法节省空间：用一个数组表示两个栈。分别用 top1、top2 表示栈顶的位置，令 top1 从 0 开始，top2 从 $N-1$ 开始。

(2) DFS 和栈

递归其实也用到了栈。每调用一次函数，都相当于入栈（**当然这步操作"隐藏在幕后"**）。函数调用完成，相当于出栈。

一般情况下，调用栈的空间大小为 16MB。也就是说，如果递归次数太多，很容易因为栈溢出导致程序崩溃，即“爆栈”。

为了防止“爆栈”，可以将递归用栈显式实现。如果可行，也可以改成迭代、递推等方法。

使用栈模拟递归时，注意入栈的顺序——逆序入栈，后递归的要先入栈，先递归的要后入栈。

下面是非递归版本的 DFS 模板：

```
stack <int> s;                          // 存储状态
void DFS(int v, …)
```

```
{
    s.push(v);                                  // 初始状态入栈
    while (!s.empty())
    {
        int x = s.top(); s.pop();       // 获取状态

        // 处理结点
        if (x达到某种条件)
        {
            // 输出、解的数量加1、更新目前搜索到的最优值等
            …
            return;
        }

        // 寻找下一状态。当然，不是所有的搜索都要这样寻找状态。
        // 注意，这里寻找状态的顺序要与递归版本的顺序相反，即逆序入栈。
        for (i=n-1;i>=0;i--)
        {
            s.push(… /*i对应的状态*/);
        }
    }

    // 无解
    cout<<"No Solution.";
}
```

9.4　队列

(1) 顺序队列!

操作规则：先进先出，后进后出。

1. **定义**：`int queue[N], front=0, rear=0;`
   `front` 指向队列首个元素，`rear` 指向队列尾部元素的右侧。
2. **入队**：`inline void push(int a) { queue[rear++]=a; }`
3. **出队**：`inline int pop() { return queue[front++]; }`
4. **队空的条件**：`inline bool empty() { return front==rear; }`

(2) 循环队列!

循环队列——把链状的队列变成了一个环状队列。与上面的链状队列相比，可以节省很大空间。

1. **定义**：`int queue[N], front=0, rear=0;`
   `front` 指向队列首个元素，`rear` 指向队列尾部元素的右侧。
2. **入队**：`inline void push(int a) { queue[rear]=a; rear=(rear+1)%N; }`
3. **出队**：`inline int pop() { int t=queue[front]; front=(front+1)%N; return t; }`
4. **队满或队空的条件**：`inline bool empty() { return front==rear; }`
   队满和队空都符合上述条件。怎么把它们区分开呢？
   第一种方法：令队列的大小是 $N+1$，然后只使用 N 个元素。这样队满和队空的条件就不一样了。
   第二种方法：在入队和出队同时记录队列元素个数。这样，直接检查元素个数就能知道队列是空还是满。

(3) BFS 和队列

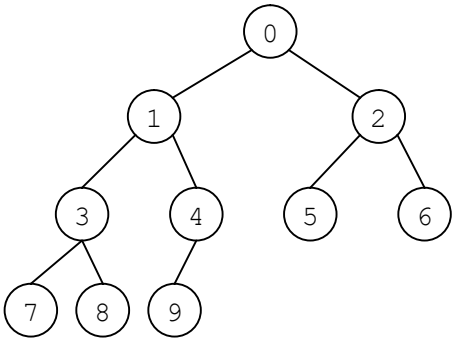
BFS 要借助队列来完成，并且，将队列改成堆栈，BFS 就变成了 DFS。BFS 的具体实现见 44 页“3.7　代码模板”。

9.5　二叉树

(1) 二叉树的链表存储法！

```
struct node
{
    int value;
    node *leftchild, *rightchild;
    //int id;            // 结点编号。
    //node *parent;      // 指向父亲结点。
} arr[N];
int top=-1;
node * head = NULL;
#define NEW(p)  p=&arr[++top]; p->leftchild=NULL;      \
                    p->rightchild=NULL; p->value=0
```

(2) 完全二叉树的一维数组存储法！



如果一个二叉树的结点严格按照从上到下、从左到右的顺序填充，就可以用一个一维数组保存。

下面假设这个树有 n 个结点，待操作的结点是 r（$0 \leqslant r < n$）。

操作	宏定义		r 的取值范围
r 的父亲	#define parent(r)	(((r)-1)/2)	$r \neq 0$
r 的左儿子	#define leftchild(r)	((r)*2+1)	$2r+1 < n$
r 的右儿子	#define rightchild(r)	((r)*2+2)	$2r+2 < n$
r 的左兄弟	#define leftsibling(r)	((r)-1)	r 为偶数且 $0 < r \leqslant n-1$
r 的右兄弟	#define rightsibling(r)	((r)+1)	r 为奇数且 $r+1 < n$
判断 r 是否为叶子	#define isleaf(r)	((r)>=n/2)	$r < n$

(3) 二叉树的遍历！

1．前序遍历

```
void preorder(node *p)
```

```
{
    if (p==NULL) return;

    // 处理结点p
    cout<<p->value<<' ';

    preorder(p->leftchild);
    preorder(p->rightchild);
}
```

2．中序遍历

```
void inorder(node *p)
{
    if (p==NULL) return;

    inorder(p->leftchild);

    // 处理结点p
    cout<<p->value<<' ';

    inorder(p->rightchild);
}
```

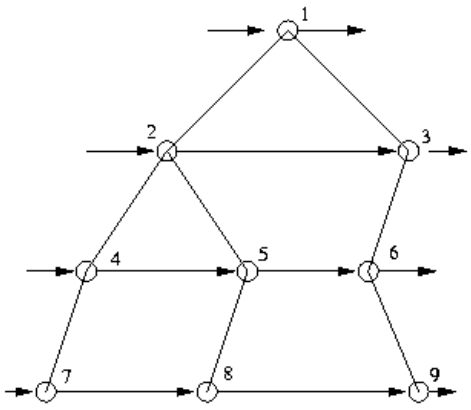
3．后序遍历

```
void postorder(node *p)
{
    if (p==NULL) return;

    postorder(p->leftchild);
    postorder(p->rightchild);

    // 处理结点p
    cout<<p->value<<' ';
}
```

假如二叉树是通过动态内存分配建立起来的，在释放内存空间时应该使用后序遍历。

4．宽度优先遍历（BFS）

首先访问根结点，然后逐个访问第一层的结点，接下来逐个访问第二层的结点……




```
node *q[N];
void BFS(node *p)
{
    if (p==NULL) return;

    int front=1,rear=2;
    q[1]=p;
    while (front<rear)
    {
        node *t = q[front++];
        // 处理结点t
        cout<<t->value<<' ';

        if (t->leftchild!=NULL) q[rear++]=t->leftchild;
        if (t->rightchild!=NULL) q[rear++]=t->rightchild;
    }
}
```

对于完全二叉树，可以直接遍历：

```
for (int i=0; i<n; i++) cout<<a[i]<<' ';
```

(4) 二叉树重建

【问题描述】二叉树的遍历方式有三种：前序遍历、中序遍历和后序遍历。现在给出其中两种遍历的结果，请输出第三种遍历的结果。

【分析】

前序遍历的第一个元素是根，后序遍历的最后一个元素也是根。所以处理时需要到中序遍历中找根，然后递归求出树。

注意！输出之前须保证字符串的最后一个字符是'\0'。

1．中序＋后序→前序

```
void preorder(int n, char *pre, char *in, char *post)
{
    if (n<=0) return;
    int p=strchr(in, post[n-1])-in;
    pre[0]=post[n-1];
    preorder(p, pre+1, in, post);
    preorder(n-p-1, pre+p+1, in+p+1, post+p);
```

```
}
```

2．前序＋中序→后序

```
void postorder(int n, char *pre, char *in, char *post)
{
    if (n<=0) return;
    int p=strchr(in, pre[0])-in;
    postorder(p, pre+1, in, post);
    postorder(n-p-1, pre+p+1, in+p+1, post+p);
    post[n-1]=pre[0];
}
```

3．前序＋后序→中序

“中＋前”和“中＋后”都能产生唯一解，但是“前＋后”有多组解。下面输出其中一种。

```
bool check(int n, char *pre, char *post)        // 判断pre、post是否属于同一棵二叉树
{
    bool b;

    for (int i=0; i<n; i++)
    {
        b=false;
        for (int j=0; j<n; j++)
            if (pre[i]==post[j])
            {
                b=true;
                break;
            }
        if (!b) return false;
    }
    return true;
}

void inorder(int n, char *pre, char *in, char *post)
{
    if (n<=0) return;
    int p=1;
    while (check(p, pre+1, post)==false && p<n)
        p++;

    if (p>=n) p=n-1;            // 此时，如果再往inorder里传p，pre已经不含有效字符了。
    inorder(p, pre+1, in, post);
    in[p]=pre[0];
    inorder(n-p-1, pre+p+1, in+p+1, post+p);
}
```

(5) 求二叉树的直径*

从任意一点出发，搜索距离它最远的点，则这个最远点必定在树的直径上。再搜索这个最远点的最远点，

这两个最远点的距离即为二叉树的直径。

求树、图（**连通图**）的直径的思想是相同的。

```
// 结点编号从1开始，共n个结点。
struct node
{
    int v;
    node *parent, *leftchild, *rightchild;
} a[1001], *p;
int maxd;
bool T[1003];
#define t(x) T[((x)==NULL)?0:((x)-a+1)]
node *p;
void DFS(node * x, int l)
{
    if (l>maxd) maxd=l, p=x;
    if (x==NULL) return;
    t(x)=false;
    if (t(x->parent)) DFS(x->parent, l+1);
    if (t(x->leftchild)) DFS(x->leftchild, l+1);
    if (t(x->rightchild)) DFS(x->rightchild, l+1);
}

int distance(node *tree)          // tree已经事先读好
{
    maxd=0;
    memset(T, 0, sizeof(T));
    for (int i=1; i<=n; i++)
        T[i]=true;
    DFS(tree,0);

    maxd=0;
    memset(T, 0, sizeof(T));
    for (int i=1; i<=n; i++) T[i]=true;
    DFS(p,0);

    return maxd;
}
```

9.6　并查集！

并查集最擅长做的事情——将两个元素合并到同一集合、判断两个元素是否在同一集合中。

并查集用到了树的父结点表示法。在并查集中，每个元素都保存自己的父亲结点的编号，如果自己就是根结点，那么父亲结点就是自己。这样就可以用树形结构把在同一集合的点连接到一起了。

(1) 并查集的实现

```
struct node
```

```
{
    int parent;                     // 表示父亲结点。当编号i==parent时为根结点。
    int count;                      // 当且仅当为根结点时有意义：表示自己及子树元素的个数
    int value;                      // 结点的值
} set[N];

int Find(int x)                         // 查找算法的递归版本（建议不用这个）
{
    return (set[x].parent==x) ? x : (set[x].parent = Find(set[x].parent));
}

int Find(int x)                         // 查找算法的非递归版本
{
    int y=x;
    while (set[y].parent != y)          // 寻找父亲结点
        y = set[y].parent;

    while (x!=y)                        // 路径压缩，即把途中经过的结点的父亲全部改成y。
    {
        int temp = set[x].parent;
        set[x].parent = y;
        x = temp;
    }
    return y;
}

void Union(int x, int y)                // 小写的union是关键字。
{
    x=Find(x); y=Find(y);               // 寻找各自的根结点
    if (x==y) return;                   // 如果不在同一个集合，合并

    if (set[x].count > set[y].count)    // 启发式合并，使树的高度尽量小一些
    {
        set[y].parent = x;
        set[x].count += set[y].count;
    }
    else
    {
        set[x].parent = y;
        set[y].count += set[x].count;
    }
}

void Init(int cnt)                      // 初始化并查集，cnt是元素个数
{
    for (int i=1; i<=cnt; i++)
    {
        set[i].parent=i;
```

```
        set[i].count=1;
        set[i].value=0;
    }
}

void compress(int cnt)                  // 合并结束，再进行一次路径压缩
{
    for (int i=1; i<=cnt; i++) Find(i);
}
```

说明：

- 使用之前调用 Init()！
- Union(x,y)：把 x 和 y 进行启发式合并，即让节点数比较多的那棵树作为“树根”，以降低层次。
- Find(x)：寻找 x 所在树的根结点。Find 的时候，顺便进行了路径压缩。
  上面的 Find 有两个版本，一个是递归的，另一个是非递归的。
- 判断 x 和 y 是否在同一集合：if (Find(x)==Find(y)) ……
- 在所有的合并操作结束后，应该执行 compress()。
- 并查集的效率很高，执行 m 次查找的时间约为 O(5m)。

(2) 团伙

【问题描述】

某地的黑社会组织猖獗，警方经过长期的调查，初步获得了一些资料：整个组织有 n 个人，任何两个认识的人不是朋友就是敌人，而且满足：① 朋友的朋友是朋友；② 敌人的敌人是朋友。

所有是朋友的人组成一个团伙。现在，警方拥有关于这 n 个人的 m 条信息（即某两个人是朋友，或某两个人是敌人），请你计算出该地最多可能有多少个团伙。

【分析】

对于朋友信息，很容易想到用并查集进行集合合并。

对于敌人信息，我们也会想到并查集：如果 3 和 5 是敌人，他们应该在同一集合；接下来 3 和 8 成为敌人，也处在同一集合。这样，5 和 8 就成为了朋友，同一集合内出现了两种关系——朋友、敌人。

为了解决这个问题，我们需要仔细分析一下。为了利用并查集，我们要统一关系：要么集合内都是朋友，要么都是敌人。通过上面的例子可以看出，集合内不能都是敌人。所以集合内应该都是朋友。

3 和 5 为敌，就是 3 的敌人和 5 为朋友。我们不妨将 3 的敌人记作 3'，那么 8 和 3'也是朋友。同样，3 和 5'、3 和 8'也是朋友。这样就可以用并查集表示敌人的关系了。

(3) 银河英雄传说[①]

【问题描述】

银河系的两大军事集团在巴米利恩星域爆发战争。泰山压顶集团派宇宙舰队司令莱因哈特率领十万余艘战舰出征，气吞山河集团点名将杨威利组织麾下三万艘战舰迎敌。

杨威利擅长排兵布阵，巧妙运用各种战术屡次以少胜多，难免恣生骄气。在这次决战中，他将巴米利恩星域战场划分成 30000 列，每列依次编号为 1，2，…，30000。之后，他把自己的战舰也依次编号为 1，2，…，30000，让第 i 号战舰处于第 i 列(i=1，2，…，30000)，形成“一字长蛇阵”，诱敌深入。这是初始阵形。当进犯之敌到达时，杨威利会多次发布合并指令，将大部分战舰集中在某几列上，实施密集攻击。合并指令为“M i j”，含义为让第 i 号战舰所在的整个战舰队列，作为一个整体（头在前尾在后）接至第 j 号战舰所在的

[①] 题目来源：NOI2002

战舰队列的尾部。显然战舰队列是由处于同一列的一个或多个战舰组成的。合并指令的执行结果会使队列增大。

然而，老谋深算的莱因哈特早已在战略上取得了主动。在交战中，他可以通过庞大的情报网络随时监听杨威利的舰队调动指令。

在杨威利发布指令调动舰队的同时，莱因哈特为了及时了解当前杨威利的战舰分布情况，也会发出一些询问指令："C i j"。该指令意思是，询问电脑，杨威利的第 i 号战舰与第 j 号战舰当前是否在同一列中，如果在同一列中，那么它们之间布置有多少战舰。

作为一个资深的高级程序设计员，你被要求编写程序分析杨威利的指令，以及回答莱因哈特的询问。

【分析】

由于我们要得到的信息除了某条战舰在哪个队列，还要知道它在该队列中的位置。因此需要对并查集进行扩充。

原有的并查集可以得到 parent、count，即战舰的父亲节点、战舰本身和它后面的战舰数量（**仅在战舰的父节点是它本身时有意义**）。可见，光有这两个数据是不能求出任意两艘战舰的战舰数量的。

设 depth 表示战舰到其父亲节点的战舰数量（包括父亲节点本身），即深度。

刚开始时，每个节点的 parent 是它自己，depth 等于 0，count 等于 1。

在回答"M i j"时，先对 i、j 进行查找和路径压缩，如果 i、j 的父亲 f1、f2 不相等，则进行合并：令 parent[f2]=f1，depth[f2]=count[f1]，count[f1]=count[f1]+count[f2]。

（**此外，路径压缩时，如果结点 a 的最终的父亲是 f，则需要令** depth[a]+=depth[f]）

在回答"C i j"时，先进行查找和路径压缩。如果位于同一集合，则输出 i、j 的 depth 的差值减去 1，否则输出-1。

(4) 可爱的猴子[①]

【问题描述】树上挂着 n 只可爱的猴子，编号为 1，…，n（2≤n≤200000）。猴子 1 的尾巴挂在树上，每只猴子有两只手，每只手可以最多抓住一只猴子的尾巴。所有的猴子都是悬空的，因此如果一旦脱离了树，猴子会立刻掉到地上。第 0，1，…，m（1≤m≤400000）秒钟每一秒都有某个猴子把它的某只手松开，因此常常有猴子掉到地上。现在请你根据这些信息，计算出每个猴子掉在地上的时间。

【分析】

如果把连在一起的猴子看成一个集合，每次松手就是断开了集合之间的某些联系或者直接将一个集合分离成两个。

我们要求的是每只猴子第一次脱离猴子 1 所在集合的时间。

难道要用"分查集"来实现吗？

我们不妨反过来想，如果时间从第 m 秒开始倒流，则出现的情形就是不断有某只猴子的手抓住另一只猴子。则我们要求的就转化成了：每只猴子最开始在什么时候合并到猴子 1 所在的集合。这样就可以应用并查集了。

设在第 t 秒钟，猴子 i 抓住（实际上是放开）了猴子 j，那么此时就将 i 所在的集合与 j 所在的集合合并。

如果需要合并，并且原先猴子 i 与猴子 j 在同一个集合，那么就将猴子 j 所在集合的所有猴子掉落的时刻都设为 t。

为了枚举某一个集合里的所有元素，我们还需要用一个链表结构与并查集共同维护猴子的集合。

9.7　小结

数据结构是计算机科学的重要分支。选择合适的数据结构，可以简化问题，减少时间的浪费。

[①] 题目来源：POI2003

1．线性表

线性表有两种存储方式，一种是顺序存储，另一种是链式存储。前者只需用一维数组实现，而后者既可以用数组实现，又可以用指针实现。

顺序表的特点是占用空间较小，查找和定位的速度很快，但是插入和删除元素的速度很慢（**在尾部速度快**）；

链表和顺序表正好相反，它的元素插入和删除速度很快，但是查找和定位的速度很慢（**同样，在首尾速度快**）。

2．栈和队列

栈和队列以线性表为基础。它们的共同点是添加、删除元素都有固定顺序，不同点是删除元素的顺序。队列从表头删除元素，而栈从表尾删除元素，所以说队列是先进先出表，堆栈是先进后出表。

栈和队列在搜索中有非常重要的应用。栈可以用来模拟深度优先搜索，而广度优先搜索必须用队列实现。

有时为了节省空间，栈的两头都会被利用，而队列会被改造成循环队列。

3．二叉树

上面几种数据结构都是线性结构。而二叉树是一种很有用的非线性结构。二叉树可以采用以下的递归定义：二叉树要么为空，要么由根结点、左子树和右子树组成。左子树和右子树分别是一棵二叉树。

计算机中的树和现实生活不同——计算机里的树是倒置的，根在上，叶子在下。

完全二叉树：一个完全二叉树的结点是从上到下、从左到右地填充的。如果高度为 h，那么 0～h-1 层一定已经填满，而第 h 层一定是从左到右连续填充的。

通常情况下，二叉树用指针实现。对于完全二叉树，可以用一维数组实现（**事先从 0 开始编号**）。

访问二叉树的所有结点的过程叫做二叉树的遍历。常用的遍历方式有前序遍历、中序遍历、后序遍历，它们都是递归完成的。

4．树

树也可以采用递归定义：树要么为空，要么由根结点和 n（$n\geqslant 0$）棵子树组成。

森林由 m（$m\geqslant 0$）棵树组成。

二叉树不是树的一种，因为二叉树的子树中有严格的左右之分，而树没有。这样，树可以用父结点表示法来表示（**当然，森林也可以**）。并查集的合并、查询速度很快，它就是用父结点表示法实现的。

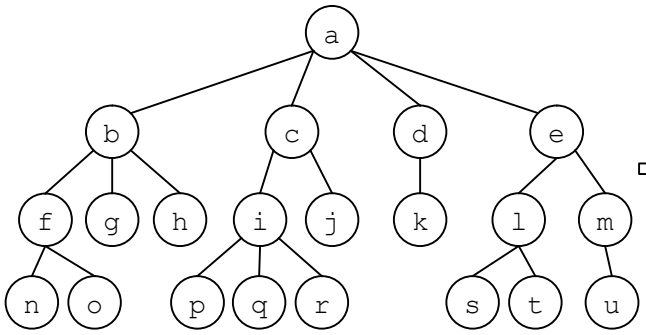
不过父结点表示法的遍历比较困难，所以常用“左儿子右兄弟”表示法把树转化成二叉树。

树的遍历和二叉树的遍历类似，不过不用中序遍历。它们都是递归结构，所以可以在上面实施动态规划。

树作为一种特殊的图，在图论中也有广泛应用。

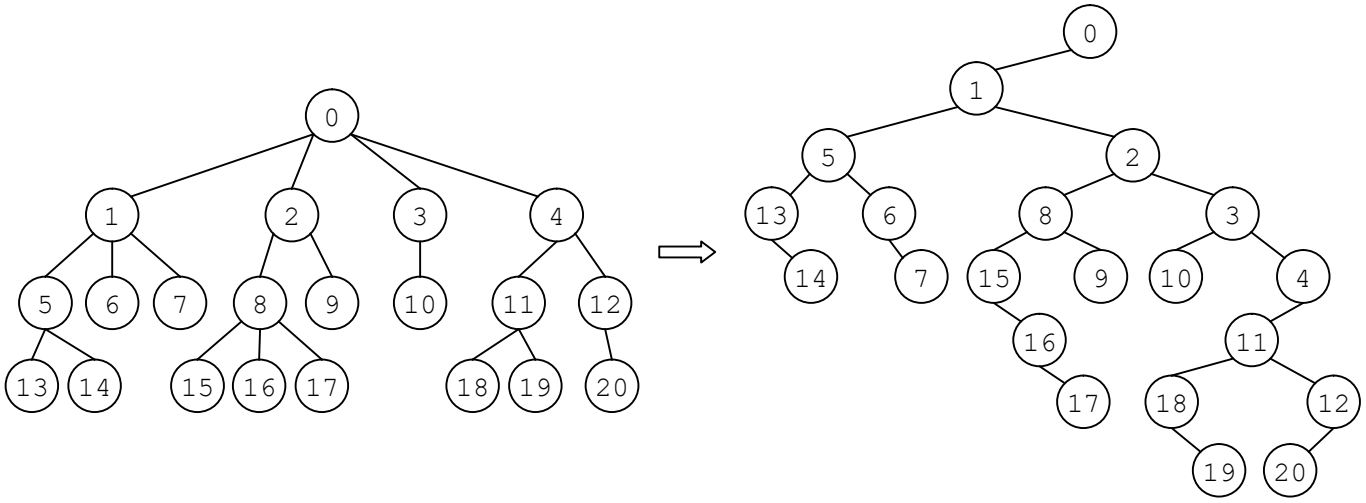
树的表示方法有很多种。第一种是父节点表示法，它适合并查算法，但不便遍历。

第二种是子节点表表示法：



索引	值	父结点	子结点表
0	a	-1	1 → 2 → 3 → 4
1	b	0	5 → 6 → 7
2	c	0	8 → 9
3	d	0	10
4	e	0	11 → 12
5	f	1	13 → 14
6	g	1	
7	h	1	
8	i	2	15 → 16 → 17
9	j	2	
10	k	3	
11	l	4	18 → 19
12	m	4	20
13	n	5	
14	o	5	
15	p	8	
16	q	8	
17	r	8	
18	s	11	
19	t	11	
20	u	12	

第三种是“左儿子右兄弟”表示法：



第十单元　查找与检索

10.1　顺序查找

```
int find(int *a, int x, int y, int v)
{
    for (int i=x; i<=y; i++)
        if (a[i]==v) return true;
    return false;
}
```

10.2　二分查找！

注意：

- 必须使用顺序存储结构，不能使用链表。
- 要查找的线性表必须是**有序**的。
- 一般情况下，“使最大值最小”和数以十万计的数据规模暗示着你：问题要用二分法解决。

(1) 普通的二分查找（非递归）

将在 a 的区间 $[x,y)$ 内寻找 v。

```
int bsearch(int *a, 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
}
```

(2) 二分查找求下界

下面函数会寻找 a 的区间 $[x,y)$ 中大于等于 v 的第一个数，使得 v 插入到 a 中对应位置，然后 a 还是一个有序的数组。

```
int lower_bound(int *a, int x, int y, int v)
{
    while (x<y)
    {
        int mid=x+(y-x)/2;
        /*
        v==a[mid]：至少找到一个，但前面可能还有。
        v<a[mid]：不能在mid的后面。
```

```
        v>a[mid]：mid和前面都不可以。
        */
        if (v<=a[mid]) y=mid;  else x=mid+1;
    }
    return x;
}
```

(3) 二分查找求上界

函数应该寻找寻找 a 的区间 $[x,y)$ 小于等于 v 的最后一个数，使得 v 插入到 a 中对应位置，然后 a 还是一个有序的数组。

只需对(2)中代码进行如下修改：

```
int upper_bound(int *a, int x, int y, int v)
{
    while (x<y)
    {
        int mid=x+(y-x)/2;
        /*
        v==a[mid]：至少找到一个，但右边可能还有。
        v<a[mid]：mid和后面都不可以。
        v>a[mid]：不能在mid的前面。
        */
        if (v>=a[mid]) x=mid;  else y=mid-1;
    }
    return y;
}
```

10.3　查找第 k 小元素!

和快速排序相近。不同的是，在“递归”这一阶段只需对“有第 k 小数据”的一部分进行排序，另一部分就不用管了。理想情况下，该算法的复杂度能达到线性水平。

```
int part(int *a, int start, int end)
{
    int low=start, high=end;
    int temp, check=a[start];

    do
    {
        while (a[high]>=check&&low<high)
            high--;
        if (a[high]<check)
            temp=a[high], a[high]=a[low], a[low]=temp;

        while (a[low]<=check&&low<high)
            low++;
        if (a[low]>check)
            temp=a[high], a[high]=a[low], a[low]=temp;
    }
```

```
    while (low!=high);

    a[low]=check;
    return low;
}

int find(int *a, int start, int end, int k)
{
    if (start==end)
        return a[start];

    int p = part(a, start, end);
    int q = p-start+1;                  // 计算p位置的“排名”

    // 只对包含第k小元素的部分进行查找和排序。
    if (k <= q)
        return find(a, start, p, k);
    else
        return find(a, p+1, end, k-q);
}
```

如果 k 不太大，可以用堆（**优先队列**）解决。

10.4　二叉排序树[①]

(1) 二叉排序树

二叉排序树的两个重要性质：

- 设 R 为任意结点，则 R 的左儿子$<R$，R 的右儿子$\geqslant R$。（**如果你需要，可以把不等号的方向调换一下**）
- 对二叉排序树进行中序遍历，得到的一定是从小到大排好的结果。

在本节，二叉树使用一般的定义方法，即 101 页的 `struct node`。

(2) 插入一个元素

插入时从顶端的根结点开始。假设插入的值为 a，对于结点 p，如果 $a<p$，就向左走，否则向右走，直到“无路可走”。

```
node * insert(int v, node * p)              // 调用方法：head = insert(x, head);
{
    if (p==NULL)
    {
        NEW(p);
        p->value = v;
    }
    else
        if (v < p->value)
```

[①] Binary Search Tree 的译名有很多，如“二叉排序树”、“二叉查找树”等。

```
            p->leftchild=insert(v, p->leftchild);
        else
            p->rightchild=insert(v, p->rightchild);
    return p;
}
```

(3) 删除一个元素*

删除的难点在于删除某一个结点之后，必须保持二叉排序树的性质。解决这个问题的最简单办法是用另外一个数来替换被删除的数，说得具体一点，就是用右子树中的最小值来替换被删除的值。

```
// 删除最小值（查找最小值，只需一路向左）。
// 注意：这个最小值a要么是叶子，要么只有右儿子。
// 如果a有右儿子，那么只需把a的右儿子放到a的位置上。
node * removemin(node * p, int &t)        // 调用：head = removeitem(head, x);
{
    if (p->leftchild == NULL)
    {
        t = p->value;
        return p->rightchild;
    }
    else
        p->leftchild = removemin(p->leftchild, t);
    return p;
}

node * removeitem(int value, node * p)       // 调用：head = removeitem(x, head);
{
    if (p==NULL) return NULL;

    if (value < p->value)
        p->leftchild = removeitem(value, p->leftchild);
    else if (value > p->value)
        p->rightchild = removeitem(value, p->rightchild);
    else
    {
        if (p->leftchild == NULL)             // 如果只有右儿子，就直接替换
            return p->rightchild;
        else if (p->rightchild == NULL)       // 如果只有左儿子，也直接替换
            return p->leftchild;
        else
            p->rightchild = removemin(p->rightchild, p->value);
    }
    return p;
}
```

(4) 查找一个元素

和插入的过程类似。如果要找的值就是结点，停止；如果要找的值比结点小，就往左走，否则往右走。

```
// 如果找到了，就返回一个包含结点的指针；如果找不到，就返回NULL。
node * find(int value, node * p)
{
    if (p==NULL) return NULL;
    if (value == p->value)
        return p;
    else if (value < p->value)
        return find(value, p->leftchild);
    else
        return find(value, p->rightchild);
}
```

10.5　堆和优先队列*

(1) 堆

这里介绍的堆指二叉堆，是完全二叉树，它可以分为最大值堆和最小值堆。

- 最大(小)值堆中，结点一定不小(大)于两个儿子的值。
- 在堆中，两兄弟的大小没有必然联系。
- 最大(小)值堆的根结点是整个树中的最大(小)值。

本节约定：

- 堆是最大值堆。修改代码中有标记的部分，就可以把最大值堆变成最小值堆。
- 二叉树使用一维数组，即 `int heap[N]`。
- 使用宏定义访问结点。有关宏定义的内容见 101 页。
- 假设二叉树中已经有 n 个元素，下标从 0 开始。
- `inline void swap(int &a, int &b) { int t=a; a=b; b=t; }`

(2) 插入一个元素

插入的过程和 BST 正好相反。在堆中插入元素，首先要把元素放到末尾，然后通过不断往上“拱”，把元素“拱”到正确的位置。

```
void insert(int value)
{
    int curr = n++;
    heap[curr] = value;          // 先放到末尾
    // 如果这个值在curr位置上却比父亲的值大，就把它与父亲交换
    while ((curr!=0) && (heap[curr] > heap[parent(curr)])) /* 把“>”换成“<”*/
    {
        swap(heap[curr], heap[parent(curr)]);
        curr = parent(curr);
    }
}
```

(3) 插入一组元素

最快的方法不是挨个插入。因为这些元素在一起，所以可以通过交换某些值来完成建堆。

说明：首先要将所有元素装进 heap[]中，并将 n 设置为正确的值。然后调用 buildheap()。

```
void siftdown(int pos)                          // 使元素往下“拱”
{
    while (!isleaf(pos))
    {
        int i = leftchild(pos);
        int r = rightchild(pos);
        if ((r < n) && (heap[i]<heap[r]))       /* 把“<”换成“>”*/
            i = r;
        /* 把“<=”换成“>=”*/
        if (heap[i] <= heap[pos]) return;       // 交换之前，已经假设pos的左右子树都是堆

        swap(heap[i], heap[pos]);
        pos = i;
    }
}

void buildheap()
{
    // 保证每个有儿子的子树都能调用到siftdown
    for (int i = n/2-1; i>=0; i--) siftdown(i);
}
```

(4) 删除一个元素*

1．删除最大（小）值

siftdown 的定义在“(3) 插入一组元素”中。

```
int removemax()
{
    if (n==0) return 0;         // 堆已经是空的
    n--;
    swap(heap[0], heap[n]);

    if (n!=0) siftdown(0);
    return heap[n];
}
```

2．删除任意值

这个函数也可以用于修改任意值：先删除后插入。它的实际用途是修改优先队列中某任务的优先级。

```
int removeitem(int pos)
{
    n--;
    swap(heap[pos], heap[n]);

    while ((pos!=0) && (heap[pos] > heap[parent(pos)]))    /*把“>”换成“<”*/
```

```
        swap(heap[pos], heap[parent(pos)]);

    siftdown(pos);
    return heap[n];
}
```

(5) 查找一个元素*

在最大(小)堆中找最大(小)值是很快的，就O(1)。然而，找其他元素的速度很慢(**直接遍历的代价为**O(n))。

一种解决方法：在建堆的同时，设置一个辅助结构(**如**BST)，记录堆中某元素的位置。

10.6　哈夫曼（Huffman）树

(1) 建立哈夫曼树

1. 哈夫曼树(**最优二叉树**)：带权路径长度最小的二叉树。
   - 树的路径长度：一棵树的每一个叶结点到根结点的路径长度的和。
   - 带权二叉树：给树的叶结点赋上某个实数值(**称叶结点的权**)。
   - 带权路径长度：各叶结点的路径长度与其权值的积的总和。
2. 如何构建哈夫树：贪心策略——权越大离根越近。
   ① 首先，创建n个初始的Huffman树，每棵树只包含单一的叶结点，叶结点记录对应字母。
   ② 接着拿走权最小但没有被处理的两棵树，再把它们标记为Huffman树的叶结点。
   ③ 把这两个叶结点标记为一个分支结点的两个子结点，而这个结点的权即为两个叶结点的权之和。
   ④ 重复上述步骤，直到序列中只剩下一个元素。

```
int n;
struct node
{
    int w;
    int parent, leftchild, rightchild;
} h[M];

node * buildtree(int *weight)      // weight[i]表示结点i的权值。返回值是Huffman的树根。
{
    int p1, p2;
    int min1, min2;
    memset(h,-1,sizeof(h));
    for (int i=0; i<n; i++) h[i].w=weight[i];

    int m=2*n-1;
    for (int i=n; i<m; i++)
    {
        min1=min2=INF;
```

```
        for (int j=0; j<i; j++)
            if (h[j].parent==-1)
            {
                if (h[j].w < min1)
                    min2=min1, min1=h[j].w, p2=p1, p1=j;
                else if (h[j].w < min2)
                    min2=h[j].w, p2=j;
            }
        h[p1].parent=i;
        h[p2].parent=i;
        h[i].leftchild=p1;
        h[i].rightchild=p2;
        h[i].w=h[p1].w+h[p2].w;
    }
    return &h[2*n-2];
}
```

（2）哈夫曼编码

哈夫曼码：哈夫曼树的非叶结点到左右孩子的路径分别用 0，1 表示，从根到叶的路径序列即为哈夫曼码。它的特点如下：

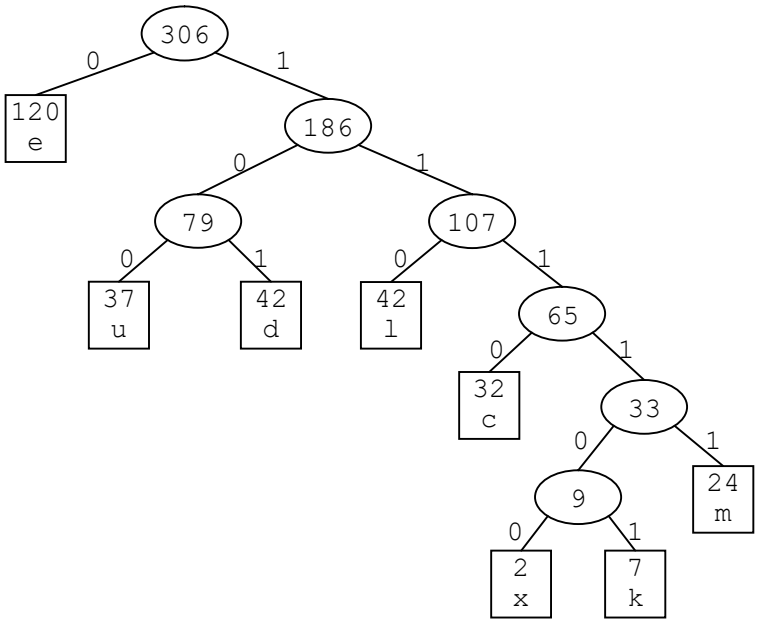
① 代码长度取决于对应字母的相对使用频率（**或者“权”**），因此它是一种变长编码。

② 任何一字符的编码不是更长编码的前缀部分（**否则在解码时会发生混淆**）。

举一个例子：假如有几个字母，代码如下

字母	频率	代码	位数
c	32	1110	4
d	42	101	3
e	120	0	1
k	7	111101	6
l	42	110	3
m	24	11111	5
u	37	100	3
z	2	111100	6

则代码对应的哈夫曼树为：



那么按照上面规则，“cell”对应的代码就是“11100110110”，“1011001110111101”对应的单词就是“duck”。

解码过程如下：从左到右逐位判别代码串，直到确定一个字母。具体的字母需要通过 Huffman 树确定。

10.7　哈希（Hash）表

(1) 实现

哈希表，即散列表，可以快速地存储和查询记录。理想哈希表的存储和查询时间都是 O(1)。

本《资料》中哈希表分以下几部分：散列函数、存储和查找时的元素定位、存储、查找。删除操作因为不常用，所以只给出思想，不给出代码。

根据实际情况，可选择不同的散列方法。

以下代码假设哈希表不会溢出。

```
// N表示哈希表长度，是一个素数，M表示额外空间的大小，empty代表“没有元素”。
const int N=9997, M=10000, empty=-1;
int a[N];

void init()                          // 初始化哈希表
{
    memset(a,empty,sizeof(a));       // 注意，只有empty等于0或-1时才可以这样做!
    memset(bucket,empty,sizeof(bucket));
    memset(first,0,sizeof(first));
}

inline int h(int);                   // 散列函数
int *locate(int, bool);              // 用于存储和查找的定位函数，并返回对应位置。
                                     // 如果用于存储，则第二个参数为true，否则为false[①]。

void save(int x)                     // 存储数据
{
    int *p = locate(x, true);
```

[①] 《资料》为了方便地介绍几种散列方法，就把查找和存储数据时的定位函数写到一起了。

```
    if (p!=NULL) *p=x;
}

bool isexist(int x)                  // 查找数据
{
    int *p = locate(x,false);
    return (p!=NULL && *p==x);
}
```

(2) 散列函数

为了达到快速存储和查找的目的，就必须在记录的存储位置和它的关键字之间建立一个确定的对应关系 h。这个关系 h 叫做哈希函数。

哈希表存取方便但存储时容易冲突：即不同的关键字可以对应同一哈希地址。如何确定哈希函数和解决冲突是关键。以下是几种常见的哈希函数的构造方法：

1. **取余数法**：$h(x) = x\%p$（$p \le N$，**且最好是素数**）
2. **直接定址法**：$h(x)=x$ 或 $h(x)=a*x+b$
3. **数字分析法**：取关键字的若干数位（**如中间两位数**）组成哈希地址。
4. **平方取中法**：关键字平方后取中间几位数组成哈希地址。
5. **折叠法**：将关键数字分割成位数相同的几部分（**最后一部分的位数可以不同**）然后取几部分的叠加和（**舍去进位**）作为哈希地址。
6. **伪随机数法**：事先产生一个随机数序列 $r[]$，然后令 $h(x)=r[x]$。

设计哈希函数时，要注意：

对关键码值的分布并不了解——希望选择的散列函数在关键码范围内能够产生一个大致平均的关键码值随机分布，同时避免明显的聚集可能性，如对关键码值的高位或低位敏感的散列函数。

对关键码值的分布有所了解——应该使用一个依赖于分布的散列函数，避免把一组相关的关键码值映射到散列表的同一个槽中。

(3) 开散列方法

哈希表中难免会发生冲突。使用开散列方法可以解决这个问题。常用操作方法是“拉链法”，即相同的地址的关键字值均链入对应的链表中。

如果散列函数很差，就容易形成长长的链表，从而影响查找的效率。

下面是用“拉链法”处理冲突时的定位函数：

```
int size=-1;
struct node {int v; node * next;} *first[N], mem[M];
#define NEW(p)              p=&mem[++size]; p->next=NULL

int * locate(int x, bool ins=false)
{
    int p=h(x);
    if (a[p]==x && !ins) return &a[p];
```

```
    // 处理冲突
    node *q = first[p];
    if (ins)
        if (q==NULL)
        {
            NEW(q);
            first[p]=q;
            return &q->v;
        }
        else
        {
            while (q->next!=NULL) q=q->next;
            node *r; NEW(r);
            q->next=r;
            return &r->v;
        }
    else
        while (q!=NULL)
        {
            if (q->v == x) return &q->v;
            q=q->next;
        }
    return NULL;
}
```

(4) 闭散列方法（开地址方法）

处理冲突的另一种方法是为该关键字的记录找到另一个“空”的哈希地址。在处理中可能得到一个地址序列 $g(i)$（$i=1, 2, \cdots, k$；$0 \leqslant g(i) \leqslant n-1$），即在处理冲突时若得到的另一个哈希地址 $g(1)$ 仍发生冲突，再求下一地址 $g(2)$，若仍冲突，再求 $g(3)$……怎样得到 $g(i)$ 呢？

溢出桶法：设一个溢出桶，不管得到的哈希地址如何，一旦发生冲突，都填入溢出桶。

再哈希法：使用另外一种哈希函数来定位。

线性探查：$g(i)=(h(x)+d_i) \ \% \ N$，其中 $h(x)$ 为哈希函数，N 为哈希表长，d_i 为增量序列。

1. 线性探测再散列：$d_i=1, 2, 3, \cdots, m-1$
2. 二次探测再散列：$d_i=1^2, -1^2, 2^2, -2^2, 3^2, -3^2, \cdots, k^2, -k^2$
3. 伪随机探测序列：事先产生一个随机数序列 random[]，令 $d_i=$random[i]。

下面是用溢出桶处理冲突时的定位函数：

```
int bucket[M], top=-1;              // 用于闭散列方法（溢出桶）

int * locate(int x, bool ins=false)
{
    int p=h(x);
    if (a[p]==x && !ins)        // 在查找模式下碰到了所需的元素
        return &a[p];
    else if (ins)
    {
        if (a[p]==empty)        // 可以插入
```

```
            return &a[p];
        else                        // 处理冲突
            return &bucket[++top];
    }
    else                            // 到溢出桶中寻找元素
        for (int i=0; i<=top; i++)
            if (bucket[i]==x) return &bucket[i];
    return NULL;
}
```

下面是用线性探查处理冲突的定位函数，当然，它也可以用于再哈希法处理冲突：

```
inline int g(int p, int i) {return (p+i)%N;}     // 根据需要来设计
int * locate(int x, bool ins=false)
{
    int p=h(x);
    int p2, c=0;
    if (a[p]==x && !ins)
        return &a[p];
    else if (ins)
    {
        do
        {
            p2 = g(p, c++);
        } while (a[p2]!=empty);
        return &a[p2];
    } else {
        do
        {
            p2 = g(p, c++);
        } while (a[p2]!=x && a[p2]!=empty);
        if (a[p2]==x) return &a[p2];
    }
    return NULL;
}
```

闭散列方法的优点是节省空间。不过，无论是溢出桶，还是线性探查，都会在寻址过程中浪费时间。线性探查的探查序列如果太长，就会使一些其他元素被迫散列在其他位置，从而影响了其他元素的查找效率。

(5) 删除*

如果使用开散列方法，那么可以直接删除元素。然而，使用闭散列方法，是不可以直接删除元素的。假如直接删除，很有可能会影响其他元素的查找。

在这种情况下，有两种删除方法：一种是交换法，另一种是标记法。

交换法：在删除某元素时，不要立刻把它清除。按照线性探查函数继续寻找，直到没有数值为止。将遇到的最后一个数值与它交换。当然，交换之前还要进行类似的操作，可谓“牵一发而动全身”。

标记法：开一个标记数组 `flag[]`。如果第 i 个元素被删除了，就将 `flag[`i`]` 设为 `true`。

1. 插入元素时，如果所在位置有标记，就把元素放到这里，并把标记清除。
2. 查找元素时，如果经过标记，就跳过去继续查找。

3．为了哈希表的效率，应该定期清理表中的标记（**或重新散列所有元素**）。

第十一单元　数学基础

11.1　组合数学

(1) 加法定理与乘法原理

1．加法原理：做一件事情，完成它可以有 ***n* 类办法**，在第一类办法中有 m_1 种不同的方法，在第二类办法中有 m_2 种不同的方法，……，在第 n 类办法中有 m_n 种不同的方法。那么完成这件事共有 $N=m_1+m_2+\cdots+m_n$ 种不同的方法。

2．乘法原理：做一件事情，完成它需要分成 ***n* 个步骤**，做第一步有 m_1 种不同的方法，做第二步有 m_2 种不同的方法，……，做第 n 步有种 m_n 不同的方法，那么完成这件事有 $N=m_1m_2\cdots m_n$ 种不同的方法。

3．两个原理的区别：一个与分类有关，一个与分步有关；加法原理是“分类完成”，乘法原理是“分步完成”。

(2) 排列与组合

1．排列

- 概念：从 n 个不同元素中，任取 $m(m\leqslant n)$ 个元素按照一定的顺序排成一列，叫做从 n 个不同元素中取出 m 个元素的一个排列。
- 排列数：从 n 个不同元素中取出 $m(m\leqslant n)$ 个元素的所有排列的个数，叫做从 n 个不同元素中取出 m 个元素的排列数，用符号 A_n^m 表示。
- 计算公式：$\mathrm{A}_n^m=n(n-1)(n-2)\cdots\cdots(n-m+1)=\dfrac{n!}{(n-m)!}$

2．组合

- 概念：从 n 个不同元素中，任取 m(m≤n) 个元素并成一组，叫做从 n 个不同元素中取出 m 个元素的一个组合。
- 组合数：从 n 个不同元素中取出 m(m≤n) 个元素的所有组合的个数，叫做从 n 个不同元素中取出 m 个元素的组合数，用符号 C_n^m 表示。
- 计算公式：$\mathrm{C}_n^m=\dfrac{\mathrm{A}_n^m}{m!}=\dfrac{n(n-1)(n-2)\cdots\cdots(n-m+1)}{m!}=\dfrac{n!}{m!(n-m)!}$
- 组合恒等式：
  ① $\mathrm{C}_n^m=\mathrm{C}_n^{n-m}$
  ② $\mathrm{C}_n^m=\mathrm{C}_{n-1}^{m-1}+\mathrm{C}_{n-1}^m$
  ③ $\mathrm{C}_n^n+\mathrm{C}_{n+1}^n+\mathrm{C}_{n+2}^n+\cdots+\mathrm{C}_{n+r}^n=\mathrm{C}_{n+r+1}^{n+1}$
  ④ $\mathrm{C}_n^0+\mathrm{C}_n^1+\mathrm{C}_n^2+\cdots+\mathrm{C}_n^n=2^n$
  ⑤ $\mathrm{C}_n^0+\mathrm{C}_n^2+\mathrm{C}_n^4+\cdots=\mathrm{C}_n^1+\mathrm{C}_n^3+\mathrm{C}_n^5+\cdots=2^{n-1}$

3．高中数学知识补充：

- n 个人围着一张圆桌坐在一起，共有 $(n-1)!$ 种坐法。
- 把 r 个相同的球放到 n 个不同颜色的盒子中去，共有 C_{n+r-1}^r 种方法。
- 从 n 个排成一排的数中取 m 个数，且数字之间互不相邻，共有 C_{n-m+1}^m 种取法。
- Catalan 数列，其数列为 1，2，5，14，42，132，429……，通项公式为 $h_n=\dfrac{\mathrm{C}_{2n}^n}{n+1}$。

4．二项式定理：$(a+b)^n=\mathrm{C}_n^0a^nb^0+\mathrm{C}_n^1a^{n-1}b^1+\cdots+\mathrm{C}_n^ra^{n-r}b^r+\cdots+\mathrm{C}_n^na^0b^n$
其中 $a^{n-r}b^r$ 的**二项式系数**为 C_n^r。

(3) 鸽巢原理（抽屉原理）

1．简单形式：如果 $n+1$ 个物体被放进 n 个盒子，那么至少有一个盒子包含两个或更多的物体。

2．加强形式：令 q_1，q_2，…，q_n 为正整数。如果将 $q_1+q_2+\cdots+q_n-n+1$ 个物体放入 n 个盒子内，那么或者第一个盒子至少含有 q_1 个物体，或者第二个盒子至少含有 q_2 个物体，…，或者第 n 个盒子含有 q_n 个物体

- 推论 1：m 只鸽子进 n 个巢，至少有一个巢里有$\left\lceil\frac{m}{n}\right\rceil$只鸽子。
- 推论 2：$n(m-1)+1$ 只鸽子进 n 个巢，至少有一个巢内至少有 m 只鸽子。
- 推论 3：若 m_1，m_2，……，m_n是正整数，且$\frac{m_1+\cdots+m_n}{n}>r-1$，则至少有一个不小于 r。

(4) 容斥原理

1．集 S 的不具有性质 P_1，P_2，…，P_m 的物体的个数

$|A_1\cap A_2\cap\cdots\cap A_m|=|S|-\sum|A_i|+\sum|A_i\cap A_j|-\sum|A_i\cap A_j\cap A_k|+\cdots+(-1)^m|A_1\cap A_2\cap\cdots\cap A_m|$

2．推论：至少具有性质 P_1，P_2，...P_m 之一的集合 S 的物体的个数有

$|A_1\cup A_2\cup\cdots\cup A_m|=|S|-|A_1\cap A_2\cap\cdots\cap A_m|=$

$\sum|A_i|-\sum|A_i\cap A_j|+\sum|A_i\cap A_j\cap A_k|+\cdots+(-1)^{m+1}|A_1\cap A_2\cap\cdots\cap A_m|$

11.2　组合数的计算！

(1) 使用加法递推——O（n^2）

$C_n^m=C_{n-1}^{m-1}+C_{n-1}^m$　（边界条件$C_n^0=C_n^n=1$）

```
int C[1001][1001];                          // 根据实际需要开数组，必要时采用高精度类型
……
memset(C,0,sizeof(C));
for (int i=0;i<=n;i++)
{
    C[i][0]=1;
    for (int j=0;j<=i;j++)
        C[i][j]=C[i-1][j-1] + C[i-1][j];
}
```

(2) 使用乘法递推——O（n）

$C_n^m=\frac{n-m+1}{m}C_n^{m-1}$，边界条件$C_n^0=1$。必须先乘后除，否则除不开。

一个小优化：$C_n^m=C_n^{n-m}$（$m>\frac{n}{2}$时用）

```
int C[1001];                                // C[m]其实表示C[n][m]，必要时采用高精度类型
……
C[0]=1;
if (m>n-m) m=n-m;
for (int i=1;i<=m;i++)
    C[i] = (n-i+1) * C[i-1] / i;            // 如果怕溢出，可以把中间结果转化成long long。
```

11.3　排列和组合的产生（无重集元素）！

```
int item[N];                                // 第i位要放置的数字
bool used[N];
int n,m;
```

(1) 全排列

将 n 个数字 1～n 进行排序，有多少种排序方法？

使用深度优先搜索，对 n 个位置逐个进行试探。时间复杂度为 O(n!)。

```
void full_ permutation(int depth)
{
    if (depth==n)
    {
        // print();                    // 输出结果
        return;
    }

    for (int i=0; i<n; i++)
        if (!used[i])
        {
            used[i]=true;
            item[depth]=i+1;
            try(depth+1);

            used[i]=false;              // 别忘记清除”使用”标记
        }
}
```

(2) 一般组合

从 n 个元素 1～n 中任取 m 个元素，有多少种取法？

一个合法的组合有这样一个特点：排在右面的数字一定严格大于左面的数字。比如说某一位上取了 3，那么从 4 开始搜索下一位就可以了。

```
void combination(int depth, int p)
{
    if (depth==m)
    {
        // print();                 // 输出结果
        return;
    }

    for (int i=p+1 ; i<n-(m-depth) ; i++)
    {
        // 由于后面的元素一定前面的大，所以不需要标记used了。
        item[depth]=i;
        try(depth+1);
    }
}

combination(0,0);
```

(3) 全组合

输入 n 个数，求这 n 个数构成的集合的所有非空子集。

和一般组合不同，这次只要产生一个解，就马上输出。

```
void full_combination(int l, int p)
{
    for (int i=0; i<l; i++)                 // 每次进入递归函数都输出
        cout<<item[i]<<" ";
    cout<<endl;
    for (int i=p; i<n; i++)
    {
        item[l] = i;                        // 在l位置放上该数
        full_combination(l+1, i+1);         // 填下一个位置
    }
}

full_combination(0, 0);
```

注意：对于一个整数，每一位不是 0 就是 1，所以可以用整数来表示一个集合。具体实例可参见 25 页“2.8 Healthy Holsteins”。

(4) 由上一排列产生下一排列

① 从右往左寻找第一个小于右边的数，位置为 j。

② 在 j 位置的右边寻找大于 a_j 的最小数字 a_k（**位置** k）

③ 将 a_j 与 a_k 的值进行交换

④ 将数列的 j+1 位到 n 位倒转。

```
int a[N];                                   // 初始化：a[i]是字典序最小的排列，0≤i＜N
int j,k, p,q, temp;

j=(n-1) - 1;
while ((j>=0)&&(a[j]>a[j+1])) j--;          // 从右往左寻找第一个小于右边的数，位置为j。

if (j>=0)                                   // 如果j<0说明已经排完了。
{
    k=n-1;
    while (a[k]<a[j])  k--;                 // 在j位置的右边寻找大于aj的最小数字ak（位置k）
    swap(a[j], a[k]);                       // 将aj与ak的值进行交换
    for (p=j+1,q=n-1; p<q; p++,q--)         // 将数列的j+1位到n位倒转
        swap(a[p], a[q]);
}
```

STL 中有与此相同的算法。头文件为<algorithm>。

next_permutation(序列第一项的地址，序列最后一项的地址+1)：产生下一排列。

prev_permutation(序列第一项的地址，序列最后一项的地址+1)：产生上一排列。

这两个函数能够用于可重集的排列。

(5) 由上一组合产生下一组合

① 从右向左寻找可以往下取一个元素的数，位置为j。

(举个例子：从 7 个数中取 4 个数，有一个组合为 1367，那么 6、7 就不能再往下取了)

② 数列的j位到n位重新取元素。

注意：

① 从N个**连续**元素中取M个元素。如果元素序号不连续，就需要修改下面的“+1”。

② 右侧的数字一定严格大于左侧的数字。

```
int a[M];                // 初始化：a[i]是字典序最小的排序，0≤i<M，1≤a[i]≤N
……
int j=m-1;
while ((j>=0)&&(a[j]==n-(m-1 -j))) j--;

if (i>=0)
{
    a[j]++;
    for (int k=j+1; k<m; k++) a[k]=a[k-1]+1;
}
```

11.4　排列和组合的产生（有重集元素）

(1) 全排列

输入n个数，输出由这n个数构成的排列，不允许出现重复的项(**但n个数本身可以重复**)。

实际上，重复排列的产生是由于同一个位置被多次填入了相同的数，并且这多次填入又是在同一次循环过程中完成的。所以，可以通过统计每个本质不同的数的个数（**输入时注意剔除重复数值**），使得循环长度由n变为m，这样，i一旦自增，就再也不会指向原先填入过的数了。

```
#include <iostream>
using namespace std;
const int N=10;
int n, m;                                  // 共有n个数，其中互不相同的有m个
int rcd[N];                                // 记录每个位置填的数
int used[N];                               // 标记m个数可以使用的次数
int num[N];                                // 存放输入中互不相同的m个数
void unrepeat_permutation(int l)
{
    int i;
    if (l == n)                            // 填完了n个数，则输出
    {
        for (i=0; i<n; i++)
            cout<<rcd[i]<<" ";
        cout<<endl;
        return;
    }
    for (i=0; i<m; i++)                    // 枚举m个本质不同的数
```

```
        if (used[i] > 0)                    // 若数num[i]还没被用完，则可使用次数减1
        {
            used[i]--; rcd[l] = num[i];     // 在l位置放上该数
            unrepeat_permutation(l+1);      // 填下一个位置
            used[i]++;                      // 可使用次数恢复
        }
}
int main()
{
    int i, j, val;
    cin>>n;
    m = 0;
    for (i=0; i<n; i++)
    {
        cin>>val;
        for (j=0; j<m; j++)
            if (num[j] == val)
            {
                used[j]++; break;
            }
        if (j == m)
            num[m] = val, used[m++] = 1;
    }

    unrepeat_permutation(0);
    return 0;
}
```

(2) 全组合

输入 *n* 个数，求这 *n* 个数构成的集合的所有子集，不允许输出重复的项（**但 *n* 个数本身可以重复**）。

需要注意的是递归调用时，第二个参数是 *i*，不再是全组合中的 *i*+1！

```
#include <iostream>
using namespace std;
const int N=10;
int n, m;                                   // 输入n个数，其中本质不同的有m个
int rcd[N];                                 // 记录每个位置填的数
int used[N];                                // 标记m个数可以使用的次数
int num[N];                                 // 存放输入中本质不同的m 个数
void unrepeat_combination(int l, int p)
{
    int i;
    for (i=0; i<l; i++)                     // 每次都输出
        cout<<rcd[i]<<" ";
    cout<<endl;
    for (i=p; i<m; i++)                     // 循环依旧从p开始，枚举剩下的本质不同的数
        if (used[i] > 0)                    // 若还可以用，则可用次数减1
```

```
        {
            used[i]--; rcd[l] = num[i];          // 在l位置放上该数
            unrepeat_combination(l+1, i);        // 填下一个位置
            used[i]++;                           // 可用次数恢复
        }
}
int main()
{
    int i, j, val;
    cin>>n;
    m = 0;
    for (i=0; i<n; i++)
    {
        cin>>val;
        for (j=0; j<m; j++)
            if (num[j] == val)
            {
                used[j]++; break;
            }
        if (j == m)
            num[m] = val, used[m++] = 1;
    }

    unrepeat_combination(0, 0);
    return 0;
}
```

排列与组合的应用：搜索问题中有一些本质上就是排列组合问题，只不过加上了某些剪枝和限制条件。解这类题目的基本算法框架常常是类循环排列、全排列、一般组合或全组合。而不重复排列与不重复组合则是两种非常有效的剪枝技巧。

11.5　秦九韶算法

【问题描述】已知一个一元 n 次多项式函数：$f(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$，已知 x_0，求 $f(x_0)$。

【方法 1】直接把 x_0 代入到式子中。时间复杂度 $O(n^2)$ 或 $O(n\log n)$。

【方法 2】

使用秦九韶算法：首先把多项式改写

$$\begin{aligned}f(x_0)&=a_nx_0^n+a_{n-1}x_0^{n-1}+\cdots+a_1x_0+a_0\\&=(a_nx_0^{n-1}+a_{n-1}x_0^{n-2}+\cdots+a_1)x_0+a_0\\&=((a_nx_0^{n-2}+a_{n-1}x_0^{n-3}+\cdots+a_2)x_0+a_1)x_0+a_0\\&=(\cdots((a_nx_0+a_{n-1})x_0+a_{n-2})x_0+\cdots+a_1)x_0+a_0\end{aligned}$$

然后令 $v_k=(\cdots(a_nx_0+a_{n-1})x_0+\cdots+a_{n-(k-1)})x_0+a_{n-k}$，则递推式为：

$$\begin{cases}v_0=a_n\\v_k=v_{k-1}x_0+a_{n-k}\end{cases}$$，其中 $1\leqslant k\leqslant n$

求出 v_n 即可。时间复杂度 $O(n)$。

秦九韶算法的一个典型应用是 N 进制变十进制。

11.6　进制转换（正整数）

(1) 十进制变 N 进制

短除法：不断地除 N，直到那个数变成 1。把所有的余数连接到一起，就是转换后的 N 进制数。

```
// bit[]是对应的N进制位，top是N进制下的最后一位数字的序号（从0开始）
void convertTo(int num, int base, int *bit, int &top)
{
    top=-1;
    do
    {
        bit[++top] = num%base;      // 可直接输出num%base
        num/=base;
    } while (num>0);
}
```

(2) N 进制变十进制

大家应该知道怎样将 N 进制转变为十进制。现在，用秦九韶算法进行计算。

```
// bit[]是对应的N进制位，top是N进制下的最后一位数字的序号（从0开始）
// 返回值是十进制数。
int convertFrom(int base, int *bit, int top)
{
    int ans=0;
    for (int i=top; i>=0; i--)
    {
        ans*=base;
        ans+=bit[i];
    }
    return ans;
}
```

11.7　高精度算法（压位存储）！

有的时候，数字会大到连 long long 都不能承受的程度。这时，我们可以自己模拟大数的各种运算。

所谓压位存储，就是在高精度数内部采用 10000 进制（即每四位放到一个数中）进行存储。它与 10 进制（即一个数位对应一个数）相比速度要快一些。

高精度数内部也可以采用 100000000 进制，但是这样就不能计算乘除法了。

(1) 定义

编程时这样做——假设 hp 是高精度类型。

先用宏定义：#define hp long long[①]，然后集中精力编写算法代码。

最后直接删除这条宏定义，把真正的高精度算法写出来。这样做的好处是无需再修改算法代码，减小了维护代价。

```
const int MAX=100;
```

[①] 不使用宏定义，也可以用 typedef long long hp;

```
struct hp
{
    int num[MAX];

    hp & operator = (const char*);
    hp & operator = (int);
    hp();
    hp(int);

    // 以下运算符可以根据实际需要来选择。
    bool operator > (const hp &) const;
    bool operator < (const hp &) const;
    bool operator <= (const hp &) const;
    bool operator >= (const hp &) const;
    bool operator != (const hp &) const;
    bool operator == (const hp &) const;

    hp operator + (const hp &) const;
    hp operator - (const hp &) const;
    hp operator * (const hp &) const;
    hp operator / (const hp &) const;
    hp operator % (const hp &) const;

    hp & operator += (const hp &);
    hp & operator -= (const hp &);
    hp & operator *= (const hp &);
    hp & operator /= (const hp &);
    hp & operator %= (const hp &);
};
```

实现运算符的代码最好写到结构体的外面。

“<<”和“>>”由于不是 hp 的成员，所以必须写到结构体的外面。

(2) 赋值和初始化

```
// num[0]用来保存数字位数。另外，利用10000进制可以节省空间和时间。
hp & hp::operator = (const char* c)
{
    memset(num,0,sizeof(num));
    int n=strlen(c),j=1,k=1;
    for (int i=1;i<=n;i++)
    {
        if (k==10000) j++,k=1;              // 10000进制，4个数字才算1位。
        num[j]+=k*(c[n-i]-'0');
        k*=10;
    }
    num[0]=j;
    return *this;
}
```

```
hp & hp::operator = (int a)
{
    char s[MAX];
    sprintf(s,"%d",a);
    return *this=s;
}

hp::hp() {memset(num,0,sizeof(num)); num[0]=1;}    // 目的：声明hp时无需显式初始化。
hp::hp (int n) {*this = n;}                        // 目的：支持“hp a=1;”之类的代码。
```

(3) 比较运算符

小学时候学过怎么比较两个数的大小吧？现在，虽为高中生，小学知识却从未过时……

```
// 如果位数不等，大小是可以明显看出来的。如果位数相等，就需要逐位比较。
bool hp::operator > (const hp &b) const
{
    if (num[0]!=b.num[0]) return num[0]>b.num[0];
    for (int i=num[0];i>=1;i--)
        if (num[i]!=b.num[i])
            return (num[i]>b.num[i]);
    return false;
}
bool hp::operator < (const hp &b) const {return b>*this;}
bool hp::operator <= (const hp &b) const {return !(*this>b);}
bool hp::operator >= (const hp &b) const {return !(b>*this);}
bool hp::operator != (const hp &b) const {return (b>*this)||(*this>b);}
bool hp::operator == (const hp &b) const {return !(b>*this)&&!(*this>b);}
```

(4) 四则运算

如果没学过竖式，或者忘了怎么用竖式算数，那么你就悲剧了……

1. 加法和减法

```
// 注意：最高位的位置和位数要匹配。
hp hp::operator + (const hp &b) const
{
    hp c;
    c.num[0] = max(num[0], b.num[0]);
    for (int i=1;i<=c.num[0];i++)
    {
        c.num[i]+=num[i]+b.num[i];
        if (c.num[i]>=10000)                       // 进位
        {
            c.num[i]-=10000;
            c.num[i+1]++;
        }
    }
    if (c.num[c.num[0]+1]>0) c.num[0]++;           // 9999+1，计算完成后多了一位
```

```
    return c;
}

hp hp::operator - (const hp &b) const
{
    hp c;
    c.num[0] = num[0];
    for (int i=1;i<=c.num[0];i++)
    {
        c.num[i]+=num[i]-b.num[i];
        if (c.num[i]<0)                                  // 退位
        {
            c.num[i]+=10000;
            c.num[i+1]--;
        }
    }
    while (c.num[c.num[0]]==0&&c.num[0]>1) c.num[0]--; // 100000000-99999999

    return c;
}

hp & hp::operator += (const hp &b) {return *this=*this+b;}
hp & hp::operator -= (const hp &b) {return *this=*this-b;}
```

2．乘法

```
hp hp::operator * (const hp &b) const
{
    hp c;
    c.num[0] = num[0]+b.num[0]+1;
    for (int i=1;i<=num[0];i++)
    {
        for (int j=1;j<=b.num[0];j++)
        {
            c.num[i+j-1]+=num[i]*b.num[j];           // 和小学竖式的算法一模一样
            c.num[i+j]+=c.num[i+j-1]/10000;          // 进位
            c.num[i+j-1]%=10000;
        }
    }
    while (c.num[c.num[0]]==0&&c.num[0]>1) c.num[0]--; // 99999999*0

    return c;
}

hp & hp::operator *= (const hp &b) {return *this=*this*b;}
```

3．除法

高精度除法的使用频率不太高。

以下代码的缺陷是：如果 b 太大，运算速度会非常慢。该问题可以用二分查找解决。

```
hp hp::operator / (const hp &b) const
{
    hp c, d;
    c.num[0] = num[0]+b.num[0]+1;
    d.num[0] = 0;
    for (int i=num[0];i>=1;i--)
    {
        // 以下三行的含义是：d=d*10000+num[i];
        memmove(d.num+2, d.num+1, sizeof(d.num)-sizeof(int)*2);
        d.num[0]++;
        d.num[1]=num[i];

        // 以下循环的含义是：c.num[i]=d/b; d%=b;
        while (d >= b)
        {
            d-=b;
            c.num[i]++;
        }
    }
    while (c.num[c.num[0]]==0&&c.num[0]>1) c.num[0]--; // 99999999/99999999

    return c;
}
hp hp::operator % (const hp &b) const
{
    ……            // 和除法的代码一样。唯一不同的地方是返回值：return d;
}

hp & hp::operator /= (const hp &b) {return *this=*this/b;}
hp & hp::operator %= (const hp &b) {return *this=*this%b;}
```

4．二分优化的除法

高精度除法速度慢，就慢在上面的 `while (d>=b)` 处。如果我们用二分法去猜 `d/b` 的值，速度就快了。

```
hp hp::operator / (const hp& b) const
{
    hp c, d;
    c.num[0] = num[0]+b.num[0]+1;
    d.num[0] = 0;
    for (int i=num[0];i>=1;i--)
    {
        // 以下三行的含义是：d=d*10000+num[i];
        memmove(d.num+2, d.num+1, sizeof(d.num)-sizeof(int)*2);
        d.num[0]++;
        d.num[1]=num[i];

        // 以下循环的含义是：c.num[i]=d/b; d%=b; 利用二分查找求c.num[i]的上界。
        // 注意，这里是二分优化后除法和朴素除法的区别！
```

```
        int left=0, right=9999, mid;
        while (left < right)
        {
            mid = (left+right)/2;
            if (b*hp(mid) <= d) left=mid+1;
            else right=mid;
        }
        c.num[i]=right-1;
        d=d-b*hp(right-1);
    }
    while (c.num[c.num[0]]==0&&c.num[0]>1) c.num[0]--; // 99999999/99999999

    return c;         // 求余数就改成return d;
}
```

(5) 输入/输出

有了这两段代码，就可以直接用 cout 和 cin 输出、输入高精度数了。

```
ostream & operator << (ostream & o, hp &n)
{
    o<<n.num[n.num[0]];
    for (int i=n.num[0]-1;i>=1;i--)
    {
        o.width(4);
        o.fill('0');
        o<<n.num[i];
    }
    return o;
}

istream & operator >> (istream & in, hp &n)
{
    char s[MAX];
    in>>s;
    n=s;
    return in;
}
```

11.8　快速幂!

如果用朴素算法计算 a^{41}，就需要做 41 次乘法运算。时间复杂度 O(n)。

使用分治算法，就有：$a^{41}=(a^{20})^2 \cdot a$，$a^{20}=(a^{10})^2$，$a^{10}=(a^5)^2$，$a^5=(a^2)^2 \cdot a$，$a^2=(a)^2$。只需 7 次乘法运算。时间复杂度 O(log$n$)。

(1) 递归算法

```
long long quickpow(long long a, long long b)
{
    if (b == 0) return 1;
```

```
    long long r=quickpow(a, b/2);
    r*=r;
    if (b%2) r*=a;                    // 即b%2==1

    return r;
}
```

(2) 非递归算法

```
long long quickpow(long long a, long long b)
{
    long long d=1, t=a;
    while (b>0)
    {
        if (t==1)  return d;
        if (b%2) d=d*t;
        b/=2;
        t*=t;
    }
    return d;
}
```

11.9　表达式求值

【问题描述】给出一个合法的表达式，请输出它的计算结果。表达式满足以下条件：

① 只有+、-、*、/、^（**乘方**）运算和括号。

② 所有操作数都是非负数。计算过程中（**包括最后结果**）可能会出现负数，但不会超过 int 的表示范围。

③ 表达式开头、结尾或表达式内部可能有多余的空格。

(1) 模拟

设两个栈，一个是操作数栈，用来存放操作数，如 3、4、8 等，另一个是运算符栈，用来存放运算符。

首先，将“(”压进运算符栈的栈底[①]。然后，依次扫描，按照栈的后进先出原则进行：

① 遇到操作数，进操作数栈；

② 遇到运算符时，则需将此运算符的优先级与栈顶运算符的优先级比较。

若高于栈顶元素则进栈，继续扫描下一符号。否则，将运算符栈的栈顶元素退栈，形成一个操作码 Q，同时操作数栈的栈顶元素两次退栈，形成两个操作数 a、b，让计算机对操作数与操作码完成一次运算操作，即 aQb，并将其运算结果存放在操作数栈中。

注意：在向 calc 函数传入表达式之前，必须在表达式两端加一层小括号！

```
#define Pop   {   pos--;                                          \
                  switch (symbol[pos+1]) {                        \
                     case '+': number[pos]+=number[pos+1];break;  \
                     case '-': number[pos]-=number[pos+1];break;  \
```

[①] 这样能保证在表达式结束时完成所有未完成的计算。当然可以单独判断并“收尾”。

```
                    case '*': number[pos]*=number[pos+1];break;     \
                    case '/': number[pos]/=number[pos+1];break;     \
                    case '^': number[pos]=pow(number[pos],number[pos+1]);break;\
                };                                                  \
            }
#define Push                symbol[++pos]=exp[i];
#define SkipBlank           { while(c==' ') i++; }      // 跳过空白
#define Is_it(x,a,b)        (x==a||x==b)                // 判断x是否为a或b
#define Is_them(x,a,b,c,d)  (x==a||x==b||x==c||x==d)    // 判断x是否为a、b、c、d之一

int number[1000];                       // 操作数栈
char symbol[1000];                      // 运算符栈
int pos;
int pow(int, int);                      // 乘方运算的代码见37页"4.3  快速幂!"。
int val(char *s, int &i)                // 字母变数字
{
    int n=0;
    while (s[i]>='0' && s[i]<='9')
    {
        n=n*10+(s[i]-'0');
        i++;
    }
    return n;
}

#define c exp[i]
int calc(char *exp, int n)
{
    pos=number[0]=0;
    int i=0;
    char t;
    while (i<=n)
    {
        SkipBlank
        while (c=='(')                  // 处理左括号
        {
            SkipBlank
            Push
            i++;
        }
        SkipBlank
        number[pos]=val(exp,i);
        SkipBlank
        do
        {
            if (c==')')                 // 处理右括号
            {
                while (symbol[pos]!='(')
```

```
                Pop
            pos--;
            number[pos]=number[pos+1];
        } else {
            while (true)                    // 运算符入栈或出栈处理
            {
                if (Is_it(c,'+','-') && symbol[pos]!='(')
                    Pop
                else if (Is_it(c,'*','/') && Is_it(symbol[pos],'*','/'))
                    Pop
                else if (Is_them(c,'+','-','*','/') && symbol[pos]=='^')
                    Pop
                else
                    break;
            }
            SkipBlank
            Push
        }
        t=exp[i];         // 接下来可能会遇到空格。如果不这样，会出错的。
        i++;
        SkipBlank
    } while (i<=n && t==')');
  }
  return number[0];
}
```

(2) 分治

找出式中最后运算的运算符 *A*，先递归地求出其左右两边的值，再将得到的值进行 *A* 运算。

```
int pow(int, int);                      // 乘方运算的代码见136页“11.8  快速幂!”。
int str2int(char *s, int x, int y)      // 字符串变数字
{
  int n=0;
  for (int i=x; i<=y; i++)
    if (s[i]>='0' && s[i]<='9')
      n=n*10+(s[i]-'0');
    else if (s[i]!=' ')
      break;
  return n;
}

int calc(char *exp, int x, int y)
{
  int lv=0, p=-1;

  // 去除两端空格
  while (exp[x]==' ' && x<=y) x++;
  while (exp[y]==' ' && x<=y) y--;
```

```
    for (int i=x; i<=y; i++)
    {
        char &c = exp[i];
        if (c==' ') continue;
        if (c=='(') lv++;
        if (c==')') lv--;

        if (lv==0)
        {
            // 当出现优先级运算时，采取优先级较低的运算符；优先级相同时，采取靠后的运算符。
            if (Is_it(c,'+','-'))
                p=i;
            else if (Is_it(c,'*','/') && (p==-1 || !Is_it(exp[p],'+','-')))
                p=i;
            else if (p==-1 && c=='^')
                p=i;
        }
    }

    if (p==-1)
    {
        // 去除两端括号
        if (exp[x]=='(' && exp[y]==')') return calc(exp, x+1, y-1);
        if (exp[x]=='(') return calc(exp, x+1, y);
        if (exp[y]==')') return calc(exp, x, y-1);

        // 字符变数字
        return str2int(exp, x, y);
    } else {
        int a=calc(exp, x, p-1);
        int b=calc(exp, p+1, y);
        switch (exp[p])
        {
            case '+': return a+b;
            case '-': return a-b;
            case '*': return a*b;
            case '/': return a/b;
            case '^': return pow(a,b);
        };
    }
    return 0;
}
```

(3) 表达式树

用分治和递归的思想构建表达式树。对树进行一次遍历，边遍历边计算，就可以算出表达式的值。

对表达式树进行前序遍历、中序遍历和后序遍历，分别会得到前缀表达式、中缀表达式和后缀表达式。其

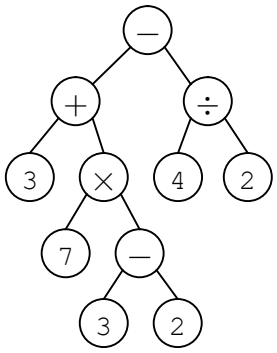
中后缀表达式不需要括号。以下图为例，三种表达式分别为：

前缀表达式：- + 3 * 7 - 3 2 / 4 2

中缀表达式：3 + 7 * 3 - 2 - 4 / 2

后缀表达式：3 7 3 2 - * + 4 / 2 -

3＋7×(3－2)－4÷2



11.10　解线性方程组*

解线性方程组 $a[][]x[]=b[]$ 的常用办法是高斯消元法（**我们学过二元方程组的加减消元法，推广一下，就是高斯消元法**）。

```
inline double fabs(double x) { return (x)>0?(x):-(x); }
const double eps = 1e-10;
```

(1) 列主元

```
// n是未知数个数，a[][]是增广矩阵。函数返回是否有唯一解，若有解则把解保存在b[]中
int gauss_cpivot(int n,double a**,double *b)
{
    int i,j,k,row;
    double maxp,t;
    for (k=0;k<n;k++)
    {
        for (maxp=0,i=k;i<n;i++)
            if (fabs(a[i][k])>fabs(maxp)) maxp=a[row=i][k];
        if (fabs(maxp)<eps) return 0;
        if (row!=k)
        {
            for (j=k;j<n;j++)
                t=a[k][j],a[k][j]=a[row][j],a[row][j]=t;
            t=b[k],b[k]=b[row],b[row]=t;
        }
        for (j=k+1;j<n;j++)
        {
            a[k][j]/=maxp;
            for (i=k+1;i<n;i++) a[i][j]-=a[i][k]*a[k][j];
        }
        b[k]/=maxp;
        for (i=k+1;i<n;i++) b[i]-=b[k]*a[i][k];
```

```
    }
    for (i=n-1;i>=0;i--)
        for (j=i+1;j<n;j++)
            b[i]-=a[i][j]*b[j];
    return true;
}
```

(2) 全主元

```
// n是未知数个数，a[][]是增广矩阵。函数返回是否有唯一解，若有解则把解保存在b[]中
bool gauss_tpivot(int n,double a**,double *b)
{
    int i,j,k,row,col,index[MAXN];
    double maxp,t;
    for (i=0;i<n;i++)
        index[i]=i;
    for (k=0;k<n;k++)
    {
        for (maxp=0,i=k;i<n;i++)
            for (j=k;j<n;j++)
                if (fabs(a[i][j])>fabs(maxp)) maxp=a[row=i][col=j];
        if (fabs(maxp)<eps) return false;
        if (col!=k)
        {
            for (i=0;i<n;i++)
                t=a[i][col],a[i][col]=a[i][k],a[i][k]=t;
            j=index[col],index[col]=index[k],index[k]=j;
        }
        if (row!=k)
        {
            for (j=k;j<n;j++)
                t=a[k][j],a[k][j]=a[row][j],a[row][j]=t;
            t=b[k],b[k]=b[row],b[row]=t;
        }
        for (j=k+1;j<n;j++)
        {
            a[k][j]/=maxp;
            for (i=k+1;i<n;i++)
                a[i][j]-=a[i][k]*a[k][j];
        }
        b[k]/=maxp;
        for (i=k+1;i<n;i++) b[i]-=b[k]*a[i][k];
    }
    for (i=n-1;i>=0;i--)
        for (j=i+1;j<n;j++)
            b[i]-=a[i][j]*b[j];
    for (k=0;k<n;k++) a[0][index[k]]=b[k];
    for (k=0;k<n;k++) b[k]=a[0][k];
```

```
    return true;
}
```

第十二单元　数论算法

12.1　同余的性质！

注意下面三个式子，它可以保证在计算中不会发生溢出。

- $(a+b)\bmod m=(a \bmod m+b \bmod m)\bmod m$
- $(a-b)\bmod m=(a \bmod m-b \bmod m+m)\bmod m$
- $(ab)\bmod m=(a \bmod m)\times(b \bmod m)\bmod m$

12.2　最大公约数、最小公倍数！

求最大公约数只需用欧几里得算法（辗转相除法）。

求最小公倍数需要用到最大公约数与最小公倍数的关系：$\mathrm{lcm}(a,b)=\frac{a\times b}{\gcd(a,b)}=\frac{a}{\gcd(a,b)}\times b$。可以看到，程序先做除法后做乘法，这样可防止计算过程中发生溢出。

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

让 gcd 运算次数最多的一对数是斐波那契数。即使这样，gcd 也不会爆栈。

gcd 可以改写成非递归算法：

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

12.3　解不定方程 ax＋by＝c！*

(1) 讨论是否有解：如果 $c \bmod \gcd(a,b)\neq 0$，那么方程无解。

(2) 转化：方程可变为 $a'x+b'y=c'$，其中 $a'=\frac{a}{\gcd(a,b)}$，$b'=\frac{b}{\gcd(a,b)}$，$c'=\frac{c}{\gcd(a,b)}$

在求解之前一定先转化，否则 x、y 可能是错误的。

(3) 求解：和 gcd 很像。

```
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;
    }
}
```

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

12.4　同余问题*

(1) 同余式

求同余式 $ax\equiv b \pmod m$ 的解。

1. 有解的充要条件是：存在一个整数 n，使得 $ax-mn=b$。
2. 求解过程：只需把它转化为 $ax-mn=b$。于是，x 和 n 就可以用 exgcd 求出来了。
3. 若 $\gcd(a,m)=d$，$b\%d==0$，则同余式 $ax\equiv b \pmod m$ 恰有 d 个解[①]。

(2) 同余式组

① 已知 $\begin{cases}x\equiv b_1 \pmod{m_1}\\x\equiv b_2 \pmod{m_2}\end{cases}$，求 x。

1. 有解的充要条件是：$(b_1-b_2)\%\gcd(m_1,m_2)==0$
2. 求解过程：设 p，q 为两个整数，则

$$\begin{cases}x\equiv b_1 \pmod{m_1}\\x\equiv b_2 \pmod{m_2}\end{cases}\Leftrightarrow\begin{cases}x-pm_1=b_1\cdots\cdots①\\x-qm_2=b_2\cdots\cdots②\end{cases}$$

②－①，得 $pm_1-qm_2=b_2-b_1$

利用 exgcd 求出 p，q，则 $x=b_1+pm_1=b_2+qm_2$

3. 通解：x 对模 $\mathrm{lcm}(m_1,m_2)$ 有唯一解，可据此构造通解。

② 已知 $\begin{cases}x\equiv b_1 \pmod{m_1}\\x\equiv b_2 \pmod{m_2}\\\cdots\\x\equiv b_n \pmod{m_n}\end{cases}$，求 x。

1. 求解过程：可以先解 $\begin{cases}x\equiv b_1 \pmod{m_1}\\x\equiv b_2 \pmod{m_2}\end{cases}$，然后构造出一个新的同余式 $x\equiv b_2' \pmod{\mathrm{lcm}(m_1,m_2)}$。

接下来将其与第三个式子联立……

2. 中间有一步出现无解，则同余式组无解。

否则 x 对模 $\mathrm{lcm}(m_1,m_2,\cdots,m_n)$ 有唯一解，可据此构造通解。

12.5　素数和素数表

(1) 素数定理

设 $\pi(x)$ 为不超过 x 的素数个数，那么 $\pi(x)\sim\frac{x}{\ln x}$（$\pi(x)$ **和** $\frac{x}{\ln x}$ **大小相近**）。据此可以估计素数个数。

[①] 在同余问题中，“不同解”指的是互不同余的解，因为只要有解，解就有无数个。

(2) 判断素数!

需要头文件：<cmath>

```
bool isprime(int x)
{
    int n = sqrt(x)+0.5;        // 假如sqrt(9)返回2.999999999，n＝2，结果可就错了。
    for (int i=2;i<=n;i++)
        if (x%i==0) return false;
    return true;
}
```

两个优化：

① 质数除了 2 都是奇数。所以除了 2，只判断奇数因数就可以了。

② 先在素数表内找因数，如果质数表内的最大因数仍未达到$\sqrt{x}$，再继续枚举。如果这个数是素数，就加入到素数表中。

(3) 筛法产生素数表!

```
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;
    }
```

这种筛法虽然有点“朴素”，但是已经够用了。

12.6　分解质因数

以下两行表示 $n=p_1^{a_1}p_2^{a_2}\cdots p_k^{a_k}$，其中 p_1、$p_2\cdots p_k$ 是 n 的质因数。

```
int n;
int p[N], a[N], k;
```

(1) 分解质因数

方法一：产生一个从 2 到$\sqrt{n}$的质数表，然后从 2 开始除。在除干净之后换下一个因数。

方法二：不产生质数表，直接从 2 开始除。在除干净之后换下一个因数。**(注：n 是素数时速度会非常慢。)**

```
int i=2;
top=-1;
while (n>1)
{
    if (n%i==0)
    {
        p[++k]=i;
```

```
        a[k]=0;
        while (n%i==0) n/=i, a[k]++;
    }
    else
        i++;                    // 换下一个因数（提示：如果上面的n%i==0为真，那么i肯定是质数）
}
```

分解质因数的实际应用是使用因数表优化乘除法。

(2) 除数函数

设 $d(n)$ 为 n 的所有因数的个数，由乘法原理可知，

$$d(n)=(a_1+1)(a_2+1)\cdots(a_k+1)。$$

(3) 欧拉函数

设 $\phi(n)$ 为 1,2,3,$\cdots n$ 中与 n 互素的元素个数，那么 $\phi(n)$ 可以用容斥原理计算。

但是容斥原理比较复杂，计算速度也非常慢。所以要用另一种形式求解：

$$\phi(n)=n(1-\frac{1}{p_1})(1-\frac{1}{p_2})\cdots(1-\frac{1}{p_k})=n\cdot\frac{p_1-1}{p_1}\cdot\frac{p_2-1}{p_2}\cdot\cdots\cdot\frac{p_k-1}{p_k})$$

可以直接计算：

```
int phi(int n)
{
    int m = (int)sqrt(n);
    int ans=n;
    // 这里是边分解质因数边计算。如果已经分解质因数，就可以直接计算了。
    for (int i=2;i<=m;i++)
        if (n%i==0)
        {
            ans=ans / i * (i-1);
            while (n%i==0) n/=i;
        }
    if (n>1) ans=ans / n * (n-1);          // 防质数
    return ans;
}
```

如果要产生欧拉函数表，可以用递推的方法完成：

```
void phi_table(int *a, int n)
{
    memset(a,0,sizeof(int)*(n+1));
    a[1]=1;

    for (int i=2; i<=n; i++)
        if (!a[i])
            for (int j=1; j<=n; j+=i)
            {
                if (!a[j]) a[j]=j;
                a[j]=a[j]/i*(i-1)
            }
```

```
}
```

第十三单元　图与图论算法

注意：以下标识符适用于本单元的所有程序。

```
int n, m;                    // n代表结点个数，m代表边的个数（有的教材分别用V、E表示）。
int G[N][N];                 // 用邻接矩阵存储的图
int u[M], v[M], w[M];        // 用边目录存储的图（u是起点，v是终点，w是权）
int first[N], next[M];       // 用邻接表存储的图（不使用指针）
vector<int> g[N];            // 用邻接表存储的图（使用矢量）
edge *adj[N];                // 用邻接表存储的图（使用指针，edge的定义见149页“邻接表”）
const int INF=100000000;     // 不要设置得过大，以防溢出
```

1．注意，其他的信息学资料用 G[i][j]=0 表示边不存在，而《资料》用 G[i][j]=**INF** 表示这条边**不存在**！

2．若无特殊说明，邻接表使用 edge *adj[N]。

3．为了保险，开数组时有 **N＞n，M＞m**。

13.1　图的实现

(1) 邻接矩阵！

开一个二维数组 G。$G[i][j]$ 表示边 (i,j) 的权。如果边 (i,j) 不存在，就令 $G[i][j]$=INF（**当然**，(i,i) **不是一条边**，$G[i][i]$ **也等于** INF）。

邻接矩阵最大的缺点就是内存空间占用太大，内存浪费严重。

(2) 边目录！

设置三个数组 $u[M]$、$v[M]$、$w[M]$，分别表示起点、终点和权。

一般情况下，从文件中读取的图都是用边目录来表示的。

(3) 邻接表（链表）！

用一个列表列出所有与现结点之间有边存在的结点名称。

```
struct edge
{
    int u,v,w;
    edge *next;
} mem[M];                          // mem相当于动态内存分配。
int size=-1;
#define NEW(p) p=&mem[++size]; p->next=NULL
edge *adj[N];                      // adj[i]代表以i为起点的边。
……
memset(adj, 0, sizeof(adj));
for (int e=0; e<m; e++)
{
    edge *p;
    NEW(p);
    cin>>(p->u)>>(p->v)>>(p->w);
```

```
    p->next=adj[p->u];
    adj[p->u]=p;
}
```

如果想检查从 a 出发的所有边，那么可以

```
for (edge *e=adj[a]; e!=NULL; e=e->next)
{
    // e->u是起点，e->v是终点，e->w是权
}
```

(4) 邻接表（静态数组）！

注意，在这个“邻接表”里放置的元素是边的序号，不是点的序号。所以还要和边目录配合使用。

```
int first[N];                       // first[u]表示从u出发的第一条边的序号
int u[M],v[M],w[M], next[M];        // next[e]表示编号为e的下一条边的序号
......
memset(first, -1, sizeof(first));
for (int e=0; e<m; e++)
{
    cin>>u[e]>>v[e]>>w[e];
    next[e]=first[u[e]];            // 插入一条边
    first[u[e]]=e;
}
```

如果想检查从 a 出发的所有边，那么可以

```
for (int e=first[a]; e!=-1; e=next[e])
{
    // u[e]是起点，v[e]是终点，w[e]是权
}
```

(5) 邻接表（STL）！

头文件：<vector>

这种邻接表也要和边目录配合使用。只需把(4)中的代码换成以下代码：

```
vector<int> g[N];                   // g[u][i]表示从u出发的第i条边的序号
int u[M],v[M],w[M];                 // 同样要和边目录配合使用
......
for (int e=0; e<m; e++)
{
    cin>>u[e]>>v[e]>>w[e];
    g[u].push_back(e);
}
```

如果想检查从 a 出发的所有边，那么可以

```
for (int i=0; i<g[a].size(); i++)
{
    int &e=g[a][i];
    // u[e]是起点，v[e]是终点，w[e]是权
```

```
}
```

13.2　图的遍历

(1) 深度优先遍历（递归）！ ［邻接矩阵］

这是基于邻接矩阵的遍历。如果需要，可以改成基于邻接表的遍历。

```
bool visited[N];
void DFS(int start)
{
    visited[start]=true;
    // 处理点start
    cout<<start<<' ';

    for (int i=0; i<n; i++)
        if ((!visited[i]) && (G[start][i]!=INF))
            DFS(i);
}
```

调用：

```
memset(visited,0,sizeof(visited));
for (int i=0;i<n;i++)
    if (!visited[i]) DFS(i);
```

(2) 广度优先遍历！ ［邻接矩阵］

这是基于邻接矩阵的遍历。如果需要，可以改成基于邻接表的遍历。

```
queue<int> q;
bool visited[N];
void BFS(int start)
{
    q.push(start); visited[start]=true;
    do
    {
        int a=q.front(); q.pop();
        // 处理点a
        cout<<a<<' ';

        for (int i=0;i<n;i++)
            if  ((!visited[i]) && (G[a][i]!=INF))
            {
                q.push(i);
                visited[a]=true;
            }
    } while (!q.empty());
}
```

调用：

```
memset(visited,0,sizeof(visited));
for (int i=0;i<n;i++)
```

```
    if (!visited[i]) BFS(i);
```

13.3　连通性问题

(1) 两个小问题

1．判断两点是否连通：

- 方法一——使用 Floyd 算法，时间复杂度 $O(n^3)$
- 方法二——从起点出发，使用 DFS 遍历，可以找到与它连通的其他点。时间复杂度 $O(n^2)$
- 方法三——**（仅用于无向图）**使用并查集。时间复杂度 $O(an)$**（a 是一个小常数）**。

2．统计无向图的强连通分量个数：使用并查集，最后只需统计父亲结点个数。

(2) 强连通分量（Kosaraju 算法）　[邻接矩阵]

该算法可用来计算有向图的强连通分量个数，并收缩强连通分量。

这个算法可以说是最容易理解，最通用的算法，其比较关键的部分是同时应用了原图 G 和反图 G'（）。操作步骤如下：

① 对原图进行 DFS 并将出栈顺序进行逆序，得到的顺序就是拓扑顺序；

② 将原图每条边进行反向；

③ 按照①中生成顺序再进行 DFS 染色，染成同色的即一个强连通块。

该算法具有一个隐藏性质：如果我们把求出来的每个强连通分量收缩成一个点，并且用求出每个强连通分量的顺序来标记收缩后的结点，那么这个顺序其实就是强连通分量收缩成点后形成的有向无环图的拓扑序列。

```
int dfn[N], top;
int color[N], cnt;
void dfs1(int k)
{
    color[k] = 1;
    for(int i=0; i<n; i++)
        if(G[k][i]!=INF && !color[i])
            dfs1(i);
    dfn[top++] = k;                 // 记录第cnt个出栈的顶点为k
}
void dfs2(int k)
{
    color[k] = cnt;                 // 本次DFS染色的点，都属于同一个scc，用num数组做记录
    for(int i=0; i<n; i++)
        if(G[i][k]!=INF && !color[i])           // 注意，我们在访问原矩阵的对称矩阵
            dfs2(i);
}
int Kosaraju()                      // 返回强连通分量个数
{
    top=cnt=0;
    memset(color, 0, sizeof(color));
    for(int i=0; i<n; i++)              // DFS求得拓扑排序
        if(!color[i]) dfs1(i);
    /*
      我们本需对原图的边反向，但由于我们使用邻接矩阵储存图，
```

```
      所以反向的图的邻接矩阵，即原图邻接矩阵的对角线对称矩阵，
      所以我们什么都不用做，只需访问对称矩阵即可
   */
   memset(color, 0, sizeof(color));
   for(int i=n-1; i>=0; i--)
      if(!color[dfn[i]]){        // 按照拓扑序进行第二次DFS
         cnt++;
         dfs2(dfn[i]);
      }
   return cnt;
}
```

(3) 强连通分量（Tarjan 算法）　[邻接表]

该算法的效率要高于 Kosaraju 算法。

任何一个强连通分量，必定是对原图的深度优先搜索树的子树（**记住这句话**）。那么，我们只要确定每个强连通分量的子树的根，然后根据这些根从树的最低层开始，一个一个的拿出强连通分量。

我们维护两个数组，一个是 index，一个是 low。其中 index[i]表示顶点 i 的开始访问时间。low[i]是最小访问时间，初始化为 index[i]，维护时 low[i]取它与 low[j]的最小值，其中 j 是与顶点 i 邻接但未删除的顶点。

在一次深搜的回溯过程中，如果发现 low[i]＝index[i]，那么，当前顶点就是一个强连通分量的根（**因为如果它不是强连通分量的根，那么它一定是属于另一个强连通分量，而且它的根是当前顶点的祖宗，那么存在包含当前顶点的到其祖宗的回路，可知 low[i]一定被更改为一个比 index[i]更小的值**）。

拿出强连通分量的方法很简单。如果当前结点为一个强连通分量的根，那么它的强连通分量一定是以该根为根结点（**剩下结点**）的子树。在深度优先遍历的时候维护一个堆栈，每次访问一个新结点，就压入堆栈。

因为当前结点是这个强连通分量中最先被压入堆栈的，那么在当前结点以后压入堆栈的并且仍在堆栈中的结点都属于这个强连通分量。

算法实现——对于所有未访问的结点 x，都进行以下操作：

① 初始化 index[x]和 low[x]；

② 对于 x 所有的邻接顶点 v：

如果没有访问过，则用同样方法访问 v，同时维护 low[x]；

如果访问过，但没有删除，就维护 low[x]。

③ 如果 index[x]＝low[x]，那么输出相应的强连通分量

```
enum _flag { NOTVIS=0, VIS, OVER } flag[N];
// NOTVIS、VIS、OVER分别表示顶点没有被访问过、顶点被访问过但未删除、顶点已被删除的状态。
int color[N];            // color[i]表示顶点i所属的强连通分量
int stack[N], top;       // 堆栈，辅助作用
int low[N];              // 很关键，与其邻接但未删除顶点的最小访问时间
int index[N];            // 顶点访问时间

void DFS(int x, int &sig, int &count)     // 深搜过程，该算法的主体都在这里
{
   stack[++top] = x;
```

```
    flag[x] = VIS;
    low[x] = index[x] = ++sig;
    for (edge *e=adj[x]; e!=NULL; e=e->next)
    {
        int &v=e->v;
        if (flag[v]==NOTVIS)
        {
            DFS(v, sig, count);
            if (low[v]<low[x]) low[x]=low[v];
        }
        else if (flag[v]==VIS && index[v]<low[x])
            low[x]=index[v];
            // 该部分的index应该是low，但是根据算法的属性，使用index也可以，且时间更少
    }
    if (low[x]==index[x])
    {
        count++;
        int t;
        do
        {
            t=stack[top--];
            color[t] = count;
            flag[t] = OVER;
        } while (t!=x);
    }
}

int Tarjan()
{
    int sig, count;
    memset(flag, 0, sizeof(flag));
    sig=count=top=0;
    for (int i=0; i<n; i++)
        if (flag[i]==NOTVIS)
            DFS(i,sig,count);
    return count;
}
```

(4) 强连通分量（Gabow 算法）　[邻接表]

Gabow 算法与 Tarjan 算法的核心思想实质上是相通的，就是利用强连通分量必定是 DFS 的一棵子树这个重要性质，通过找出这个子树的根来求解强分量。

具体实现是利用一个栈 S 来保存 DFS 遇到的所有树边的另一端顶点，在找出强分量子树的根之后，弹出 S 中的顶点一一进行编号。

与 Tarjan 算法不同的是，Tarjan 算法通过一个 low 数组来维护各个顶点能到达的最小前序编号，而 Gabow 算法通过维护另一个栈来取代 low 数组，将前序编号值更大的顶点都弹出，然后通过栈顶的那个顶点来判断是否找到强分量子树的根。

```
int pre[N];
```

```
int color[N];
int S[N], P[N];      // 两个栈，S用来保存所有结点，P用来维护路径
int top_s, top_p;
int cnt, id;
void DFS(int x)
{
    int v;
    pre[x] = cnt++;  // 对前序编号编号
    S[++top_s]=x;    // 将路径上遇到的树边顶点入栈
    P[++top_p]=x;
    for (edge *e=adj[x]; e!=NULL; e=e->next)
    {
        v=e->v;
        if (pre[v] == -1)                    // 如果以前未遇到当前顶点，则对其进行DFS
            DFS(v);
        else if (color[v] == - 1)            // 如果当前顶点不属于强分量，
            while (pre[P[top_p]] > pre[v]) // 就将路径栈P中大于当前顶点pre值的顶点都弹出
                top_p--;
    }
    if (P[top_p] == x)                       // 如果P栈顶元素等于x，则找到强分量的根—x
    {
        top_p--;
        id++;
        do
        {
            v = S[top_s--];                  // 把S中的顶点弹出编号
            color[v] = id;
        } while (v != x);
    }
}
int Gabow()
{
    top_s=top_p=-1;
    memset(pre,-1,sizeof(pre));
    memset(color,-1,sizeof(color));
    cnt=id=0;
    for (int v=0; v<n; v++)
        if (pre[v] == -1)
            DFS(v);
    return id;                               // 返回id的值，这恰好是强连通分量的个数
}
```

(5) 有向图的传递闭包（Floyd-Warshall 算法）　[邻接矩阵]

时间复杂度：$O(n^3)$

Floyd-Warshall 算法会把图上任意两个点的连通性都算出来。

```
bool f[N][N];            // 如果存在一条从i出发，到j结束的路径，f[i][j]=true。
……
```

```
// 预处理——可以在读图时完成
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        f[i][j] = (G[i][j]!=INF);

for (int k=0; k<n; k++)
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            f[i][j] = f[i][j] || (f[i][k] && f[k][j]);
```

13.4　欧拉回路　[邻接矩阵]

【问题描述】在一个**无向图**中，一条包含所有**边**，且其中每一条边只经过一次的路径叫做欧拉通路。若这条路径的起点与终点为同一点，则为欧拉回路。请找出图中的欧拉回路。

判定一个图是否存在欧拉通路或欧拉回路：

定理 1：一个图有欧拉回路当且仅当它是连通的（**即不包括** 0 **度的结点**）且每个结点都有偶数度。

定理 2：一个图有欧拉通路当且仅当它是连通的且除两个结点外，其他结点都有偶数度。

定理 3：在定理 2 的条件下，含奇数度的两个结点中，一个必为欧拉通路的起点，另一个必为终点。

```
int G[N][N];      // 无向图的邻接矩阵。如果两点连通，则为1，否则为0
int cnt[N];
int circuit[N], pos;
int start,oddnumber;

void search(int i)
{
    for (int j=0; j<n; j++)
        if (G[i][j]==1)
        {
            G[i][j]=G[j][i]=0;
            search(j);
        }
    circuit[pos++]=i;
}

bool find_circuit()                    // 返回false表示无欧拉回路
{
    start=oddnumber=0;
    memset(cnt, 0, sizeof(cnt));
    for (int i=0; i<n; i++)            // 统计结点入度
        for (int j=0; j<n; j++)
            cnt[i]+=G[i][j];

    for (int i=0; i<n; i++)
        if (cnt[i]%2==1)
        {
            start=i;
```

```
            oddnumber=oddnumber+1;
        }

    if (oddnumber>2 || oddnumber==1)
        return false;
    else
    {
        pos=0;
        search(start);
        for (int i=0; i<pos; i++)
            cout<<circuit[i]<<"--->"<<endl;
    }
    return true;
}
```

13.5　最小生成树（MST）

已知 n 个城市，并且已知它们之间的距离。问怎样修路才能保证道路总长最短，并且每个城市都被连接？

(1) Prim 算法　[邻接矩阵]

Prim 算法是贪心算法，贪心策略为：找到目前情况下能连上的权值最小的边的另一端点，加入之，直到所有的顶点加入完毕。

Prim 适用于稠密图。

朴素 Prim 的时间复杂度是 $O(n^2)$，因为在寻找离生成树最近的未加入顶点时浪费了很多时间。所以，可以用堆进行优化。堆优化后的 Prim 算法的时间复杂度为 $O(m\log n)$。

堆优化 Prim 的代码比较复杂，并查集优化的 Kruskal 算法与它相比，要好很多。

```
int minEdge[N], cloest[N];              // 与点N连接的最小边

int Prim(int start=0)                   // start的出度不能为0!
{
    int ans=0, k=0, min;

    // 加入第一个点
    for (int i=0;i<n;i++)
    {
        minEdge[i]=G[start][i];
        cloest[i]=start;
    }
    minEdge[start]=0;

    for (int i=0;i<n-1;i++)
    {
        min=INF;              // 寻找离生成树最近的未加入顶点k
        for (int j=0;j<n;j++)
            if (minEdge[j]!=0 && minEdge[j]<min) min=minEdge[k=j];

        // 把找到的边加入到MST中
        ans+=minEdge[k];
```

```
        minEdge[k]=0;           // 加入完毕。以后不用再处理这个点。

        // 重新计算最短边
        for (int j=0;j<n;j++)
            if (G[k][j]<minEdge[j])
            {
                minEdge[j]=G[k][j];
                cloest[j]=k;
            }
    }
    return ans;
}
```

(2) Kruskal 算法！ [边目录]

Kruskal 算法是贪心算法，贪心策略为：选目前情况下能连上的权值最小的边，若与以生成的树不够成环，加入之，直到 n-1 条边加入完毕。

时间复杂度为 O($n$log$m$)，最差情况为 O($m$log$n$)。相比于 Prim，这个算法更常用。

```
int parent[N], rank[M];                                     // p代表并查集，r是边的序号

int comp (const int i, const int j) {return w[i]<w[j];}    // 排序时使用
int find (int x)                                            // 带路径压缩的查找函数
{
    return parent[x]==x ? x : parent[x] = find(parent[x]);
}

int Kruskal()
{
    int ans = 0;
    for (int i=0;i<n;i++) parent[i]=i;        // 初始化并查集
    for (int i=0;i<m;i++) rank[i]=i;          // 边的序号（下面要按照边的权值大小来排序）
    sort(rank, rank+m, comp);                 // 按照边的权值大小排序

    for (int i=0;i<m;i++)
    {
        int e=rank[i];
        int x=find(u[e]), y=find(v[e]);       // 找出当前边的两个端点所在集合的编号

        if (x!=y)                             // 如果不在同一集合，合并
        {
            ans += w[e];
            parent[x] = y;
        }
    }
    return ans;
}
```

13.6　单源最短路问题（SSSP 问题）

约定：从 start 到点 i 的距离为 d[i]。如果 d[i]==INF，说明 start 和 i 不连通。

(1) Dijkstra 算法！ [邻接矩阵]

Dijkstra 算法是贪心算法。它只适用于**所有边的权都大于 0 的图**。

基本思想是：设置一个顶点的集合 S，并不断地扩充这个集合，当且仅当从源点到某个点的路径已求出时它才属于集合 S。

开始时 S 中仅有源点，调整非 S 中点的最短路径长度，找当前最短路径点，将其加入到集合 S，直到所有的点都在 S 中。

时间复杂度：$O(n^2)$

```
bool visited[N];                // 是否被标号
int d[N];                       // 从起点到某点的最短路径长度
int prev[N];                    // 通过追踪prev可以得到具体的最短路径（注意这里是逆序的）

void Dijkstra(int start)
{
    // 初始化：d[start]=0，且所有点都未被标号
    memset(visited, 0, sizeof(visited));
    for (int i=0; i<n; i++) d[i]=INF;
    d[start]=0;

    // 计算n次
    for (int i=0; i<n; i++)
    {
        int x, min=INF;
        // 在所有未标号的结点中，选择一个d值最小的点x。
        for (int a=0; a<n; a++) if (!visited[a] && d[a]<min) min=d[x=a];
        // 标记这个点x。
        visited[x]=true;
        // 对于从x出发的所有边（x，y），更新一下d[y]。
        for (int y=0; y<n; y++) if (d[y] > d[x]+G[x][y])
        {
            d[y] = d[x]+G[x][y];
            prev[y] = x;                    // y这个最短路径是从x走到y。
        }
    }
}
```

(2) 使用优先队列的 Dijkstra 算法*　[邻接表]

朴素的 Dijkstra 算法在选 d 值最小的点时要浪费很多时间，所以可以用优先队列（**最小堆**）来优化。

时间复杂度：$O[(n+m)\log m]$，最差情况（**密集图**）为 $O(n^2 \log m)$

```
// 需要的头文件：<queue>、<vector>、<utility>
typedef pair<int, int> pii;         // 将终点和最短路径长度“捆绑”的类型
// 定义一个优先队列，d值最小的先出列
```

```
priority_queue < pii, vector<pii>, greater<pii> > q;
int d[N], prev[N];

void Dijkstra(int start)
{
    for (int i=0; i<n; i++) d[i]=INF;
    d[start]=0;
    q.push (make_pair(d[start], start));

    while (!q.empty())
    {
        // 在所有未标号的结点中，选择一个d值最小的点x。
        pii u=q.top(); q.pop();
        int x=u.second;

        if (u.first!=d[x]) continue;      // 已经计算完

        for (edge *e=adj[x]; e!=NULL; e=e->next)
        {
            int &v=e->v, &w=e->w;
            if (d[v] > d[x]+ w)
            {
                d[v] = d[x]+ w;          // 松弛
                prev[v]=x;
                q.push(make_pair(d[v], v));
            }
        }
    }
}
```

(3) Bellman-Ford 算法　[边目录]

Bellman-Ford 算法是迭代法，它不停地调整图中的顶点值（**源点到该点的最短路径值**），直到没有点的值调整了为止。

该算法除了能计算最短路，还可以检查负环（**一个每条边的权都小于 0 的环**）。如果图中有负环，那么这个图不存在最短路。

```
bool Ford(int start)                    // 有负环则返回false
{
    // 初始化
    for (int i=0; i<n; i++) d[i]=INF;
    d[start]=0;

    for (int k=0;k<n-1;k++)             // 迭代n次
        for (int i=0; i<m; i++)         // 检查每条边
        {
            int &x=u[i], &y=v[i];
```

```
            if (d[x]<INF) d[y]=min(d[y],d[x]+w[i]);
        }

    // 下面的代码用于检查负环——如果全部松弛之后还能松弛，说明一定有负环
    for (int i=0; i<m; i++)            // 再次检查每条边
    {
        int &x=u[i], &y=v[i];
        if (d[y]>d[x]+w[i]) return false;
    }
    return true;
}
```

(4) SPFA! [邻接表]

SPFA 是使用队列实现的 Bellman-Ford 算法。操作步骤如下：

① 初始队列和标记数组。

② 源点入队。

③ 对队首点出发的所有边进行松弛操作（**即更新最小值**）。

④ 将不在队列中的尾结点入队。

⑤ 队首点更新完其所有的边后出队。

```
queue<int> q;
bool inqueue[N];                    // 是否在队列中
int cnt[N];                         // 检查负环时使用：结点进队次数。如果超过n说明有负环。

bool SPFA(int start)                // 有负环则返回false
{
    // 初始队列和标记数组
    for (int i=0; i<n; i++) d[i]=INF;
    d[start]=0;
    memset(cnt,0,sizeof(cnt));
    q.push(start);                              // 源点入队
    cnt[start]++;
    while (!q.empty())
    {
        int x=q.front(); q.pop();
        inqueue[x]=false;

        // 对队首点出发的所有边进行松弛操作（即更新最小值）
        for (edge *e=adj[x];e!=NULL;e=e->next)
        {
            int &v=e->v, &w=e->w
            if (d[v]>d[x]+w)
            {
                d[v] = d[x]+w;
                // 将不在队列中的尾结点入队
                if (!inqueue[v])
                {
```

```
                inqueue[v]=true;
                q.push(v);

                if (++cnt[v]>n) return false;         // 有负环
            }
        }
    }
    return true;
}
```

13.7　每两点间最短路问题（APSP 问题）！

此类问题使用 Floyd-Warshall 算法解决。它是动态规划算法。

1．**状态表示**：$f(i,j)$ 表示从点 i 到点 j 的最短距离。

2．**状态转移方程**：$f(i,j)=\min\{f(i,k)+f(k,j)\}$　（i 到 k、k 到 j 都是连通的）

3．**时间复杂度**：$O(n^3)$

```
int f[N][N], prev[N][N];    // 追踪prev可以得到最短路。
int len=INF;                // 最小环的长度

void Floyd()
{
    // 初始化——可以在读图时完成
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            f[i][j] = G[i][j];

    memset(prev, -1, sizeof(prev));
    /* len=INF; */

    // 计算。注意，k在最外面。
    for (int k=0; k<n; k++)
    {
        /* 如果求最小环，请将下面的代码插入到这里。 */
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                if (f[i][k] + f[k][j] < f[i][j])
                {
                    f[i][j] = f[i][k] + f[k][j];
                    prev[i][j] = k;
                }
    }
}
// cout<<f[i][j]<<endl;
// 通过递归调用追踪prev，就可以得到最短路径上的结点。
```

Floyd 算法可以用来求最小环（**无向图！**）。将以下代码插入到上面的标记处即可。

```
for (int i=0; i<k; i++)
    for (int j=i+1; j<k; j++)
        len = min(len, G[i][j]+f[i][k]+f[k][j]);
// G是无向图！len为最小环的和
```

13.8　拓扑排序

AOV 网（Activity On Vertex NetWork）：用顶点表示活动，弧表示活动间的优先关系的有向图。

【问题描述】有 *N* 项活动，用 AOV 网表示出来，A→B 指 A 必须在 B 之前完成。请输出一个合理的活动顺序。

AOV 网中不能存在有向环。如果存在环，则无法拓扑排序。所以，拓扑排序可用来检查有向图中是否有环。

(1) DFS！［**邻接矩阵**］

对于每个顶点，都先搜索并输出它的前趋。使用时调用 topsort。

```
int vis[N];                 // 结点访问情况：0表示未访问，1表示正在访问，2表示已访问。
int a[N], a_top;            // 结果保存在数组a中
bool DFS(int v)
{
    vis[v]=1;

    for (int i=0; i<n; i++)
        if (G[i][v]!=INF)               // 找到前趋
        {
            if (vis[i]==1)              // 这个点进入两次，说明出现了环
                return false;
            else if (vis[i]==0)
                if (!DFS(i)) return false;
        }

    // 处理点v
    vis[v]=2;
    a[a_top++]=v;
    return true;
}

bool topsort()
{
    memset(visited, 0, sizeof(visited));
    a_top=0;

    for (int i=0;i<n;i++)
        if (!visited[i])
            if (!DFS(i)) return false;
    return true;
}
```

(2) 模拟堆栈#　[邻接矩阵]

注意：下面代码不能把所有带环的情况都检查出来！

```
int cnt[N];                 // cnt[i]表示结点i的入度
int a[N], a_top;            // 结果保存在数组a中
bool topsort()
{
    int i, top = -1;
    a_top=0;
    memset(cnt,0,sizeof(cnt));
    for (i=0; i<n; i++)
        if (cnt[i]==0)          //  下标模拟堆栈
            cnt[i] = top, top = i;

    for (i=0; i<n; i++)
        if (top == -1)
            return false;           // 存在回路
        else
        {
            int j = top;
            top = cnt[top];
            a[a_top++]=j;
            for (int k=0; k < n; ++k)
                if (G[j][k] && ((--cnt[k]) == 0))
                    cnt[k] = top, top = k;
        }
    return true;
}
```

(3) 使用辅助队列！　[邻接矩阵]

① 记录每个结点入度。
② 将入度为零的点加入队列。
③ 依次对入度为零的点进行删边操作，同时将新得到的入度为零的点加入队列。
④ 继续对队列中未进行操作的点进行操作。

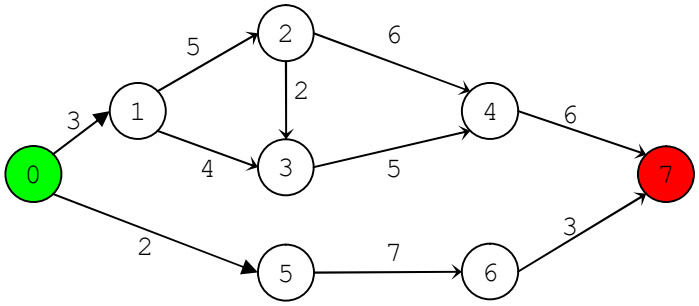
```
queue<int> q;
int cnt[N];                 // cnt[i]表示结点i的入度
int a[N], a_top;            // 结果保存在数组a中

bool topsort()
{
    memset(cnt,0,sizeof(cnt));
    // 记录每个结点的入度。
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            if (G[i][j]<INF) cnt[j]++;

    // 将入度为零的点加入队列。
```

```
    for (int i=0;i<n;i++)
        if (cnt[i]==0) q.push(i);

    while (!q.empty())
    {
        int i=q.front(); q.pop();
        a[a_top++]=i;                          // 记录结点
        for (int j=0;j<n;j++)
        {
            if (G[i][j]<INF)
            {
                cnt[j]--;                      // 依次对入度为零的点进行删边操作
                if (cnt[j]==0) q.push(j);      // 将新得到的入度为零的点加入队列
            }
        }
    }

    // 如果排序结束，但存在入度不为0的点，就说明有环。
    for (int i=0;i<n;i++) if (cnt[i]!=0) return false;

    return true;                               // 图中没有环
}
```

13.9　关键路径

AOE 网（Activity on Edge）：它是一个带权的有向无环图，可用来估算工程的完成时间。

以下图为例。绿色的 0 叫做源点，红色的 7 叫做汇点；其他的每一个点都叫做一个事件；一条边叫做一项活动，权代表活动持续的时间。



其中，路径最长的路径叫做关键路径，影响工程进度的活动叫关键活动，并且关键路径上的活动一定是关键活动。

最早开工时间和最晚开工时间相等的工程是关键活动。

【问题描述】假设以 AOE 网表示一个施工流程图，弧上的权值表示完成该项子工程所需的时间。求：

① 完成整个工程的最短时间；

② 在满足依赖关系，并且不影响最短时间的前提下，每项工程的最早开工时间和最晚开工时间；

③ 关键路径

问题①、③以问题②为基础，比较容易实现。所以下面两个程序将解决问题②。

(1) 对 DAG 求关键路径　[邻接矩阵/邻接表]

这是 DAG 上的动态规划。注意，**只有图的拓扑排序序列为 0,1,2,……时才能用这种算法！**

1. **状态表示**：设 $f(i)$ 为事件 i 的最早开工时间，$g(i)$ 为事件 i 的最晚开工时间。
2. **状态转移方程**：

$f(i)=\min\{f(j)+G[j][i]\}$

$g(j)=\min\{g(j)-G[j][i]\}$

其中 j 是 i 的前趋。

3. **边界条件**：$f(0)=0$，$g(n)=f(n-1)$
4. 计算时先顺推出 $f(n)$，然后令 $g(n)=f(n)$，再倒推出 $g(0)$。如果关键路径通过点 i，则 $f(i)=g(i)$。

```
f[0]=0;
for (int i=1; i<=n; i++)
{
    int max=0;
    for (int j=0; j<=n; j++)
        if (G[j][i]!=INF)
            if (G[j][i]+f[j]>max) max=G[j][i]+f[j];
    f[i]=max;
}

g[n]=f[n];
for (int i=n-1; i>=0; i--)
{
    int min=INF;
    for (int j=0; j<=n; j++)
        if (G[i][j]!=INF)
            if (g[j]-G[i][j]<min) min=g[j]-G[i][j];
    g[i]=min;
}

// 寻找关键路径
for (int i=1; i<n; i++) if (et[i]==eet[i]) cout<<i<<"->";
cout<<n;
```

(2) 对拓扑序列求关键路径　[邻接矩阵]

这仍然需要动态规划。

1. **划分阶段**：以**拓扑序列**划分阶段，一项工程为一个阶段。
2. **状态表示**：设 $f(i)$ 为事件 i 的最早开工时间，$g(i)$ 为事件 i 的最晚开工时间。
3. **状态转移方程**：

$f(i)=\min\{f(j)+G[j][i]\}$

$g(j)=\min\{g(j)-G[j][i]\}$

其中 $j<i$。由于序列由拓扑排序产生，j 要么是 i 的前趋，要么和 i 无依赖关系。

计算时先顺推出 $f(n)$，然后令 $g(n)=f(n)$，再倒推出 $g(0)$。

```
#include <iostream>
#include <cstring>
using namespace std;
```

```
const int INF=100000000, N=10000;
int G[N][N];
int a[N];            // 拓扑排序序列
int f[N],g[N],prev[N];
int n,m;

bool topsort();      // 拓扑排序，代码见164页，但是所有“<n”要改成“<=n”。

void print(int i)
{
    if (i==-1) return;
    print(prev[i]);
    cout<<i<<" ";
}

int main()
{
    cin>>n>>m;
    for (int i=0; i<=n; i++)
        for (int j=0; j<=n; j++)
            G[i][j]=INF;
    for (int i=1; i<=m; i++)
    {
        int u,v,w;
        cin>>u>>v>>w;
        G[u][v]=w;
    }
    memset(f,0,sizeof(f));
    memset(g,0,sizeof(g));
    memset(prev,-1,sizeof(prev));
    if (!topsort()) return 0;

    for (int i=1; i<=n; i++)
        for (int j=0; j<i; j++)
        {
            int &x=a[j], &y=a[i];
            if (G[x][y]!=INF && f[y]<f[x]+G[x][y])
            {
                f[y]=f[x]+G[x][y];
                prev[y]=x;
            }
        }
    g[n]=f[n];
    for (int i=n-1; i>=0; i--)
        for (int j=n; j>i; j--)
        {
            int &x=a[j], &y=a[i];
            if (G[y][x]!=INF && g[y]<g[x]-G[y][x])
```

```
            g[y]=g[x]-G[y][x];
        }

    cout<<"Len="<<f[a[n]]<<endl;
    print(a[n]);
    return 0;
}
```

13.10　二分图初步

二分图是一个比较重要的模型。

无向图 G 为二分图的充分必要条件是，G 至少有两个顶点，且其所有回路的长度均为偶数。

(1) 二分图的判定　[邻接表]

对二分图的结点进行染色。如果处于同一组，则应该染成同色，否则为不同色。在染色过程中，如果发生矛盾，说明此图不是二分图。

```
int color[N];                                   // 结点的颜色（需要初始化为0）

bool DFS(int i)
{
    for (edge *e = adj[i]; e!=NULL; e=e->next)  // 检查每一条边
    {
        int &v=e->v;
        if (!color[v])                          // 如果与当前结点相邻的结点还未染色
        {
            color[v]=1-color[i];
            if (!DFS(v)) return false;
        }
        // 如果两个相邻结点染色相同，说明此图不是二分图，返回无解。
        if (color[v] == color[i]) return false;
    }
    return true;
}
```

(2) 匹配的概念

匹配：给定一个二分图 G，在 G 的一个子图 M 中，M 的边集中的任意两条边都不依附于同一个顶点，则称 M 是一个匹配。

最大匹配：图中包含边数最多的匹配称为图的最大匹配。

完美匹配：如果所有点都在匹配边上，称这个最大匹配为完美匹配。

下面给出关于二分图最大匹配的三个定理：

① **最大匹配数＋最大独立集数＝*n***

② 二分图的最小覆盖数＝最大匹配数

③ 最小路径覆盖＝最大独立集

最大独立集是指求一个二分图中最大的一个点集，该点集内的点互不相连。

最小顶点覆盖是指在二分图中，用最少的点，让所有的边至少和一个点有关联。

最小路径覆盖是指一个不含圈的有向图 G 中，G 的一个路径覆盖是一个其结点不相交的路径集合 P，图中

的每一个结点仅包含于 P 中的某一条路径。路径可以从任意结点开始和结束，且长度也为任意值，包括 0。

(3) 匈牙利算法（DFS）　[邻接矩阵]

首先给出增广路的定义（也称增广轨或交错轨）。

增广路径：若 P 是图 G 中一条连通两个未匹配顶点的路径，并且属于 M 的边和不属于 M 的边（即已匹配和待匹配的边）在 P 上交替出现，则称 P 为相对于 M 的一条增广路径。

由增广路的定义可以推出下述三个结论：

① P 的路径长度必定为奇数，第一条边和最后一条边都不属于 M。

② P 经过取反操作可以得到一个更大的匹配 M'。

③ M 为 G 的最大匹配当且仅当不存在相对于 M 的增广路径。

求二分图的最大匹配常用匈牙利算法，它是通过不断地寻找增广轨来实现的。很明显，如果二分图的两部分点分别为 X 和 Y，那么最大匹配的数目应该小于等于 $\min\{X,Y\}$。因此我们可以枚举某一部分里的每一个点，从每个点出发寻找增广轨，最后，该部分的点找完以后，就找到了最大匹配的数目。当然我们也可以通过记录来找出这些边。

匈牙利算法找一条增广路的复杂度为 $O(m)$，最多找 n 条增广路，故时间复杂度为 $O(mn)$。

```
int xN, yN;                     // 表示X、Y两部分点的数目。
int matchX[N], matchY[N];       // 表示与对应的X、Y部分匹配的点。
bool used[N];                   // 标记y[b]中的点是否被使用过。
bool SearchPath(int x)          // 从x出发寻找增广路，如果找不到则返回false。
{
    for(int y = 0; y < yN; y++)
        if(G[x][y] && !used[y])
        {
            used[y] = true;
            if(matchY[y] == -1 || SearchPath(matchY[y]))
            {
                matchY[y] = x; matchX[x] = y;
                return true;
            }
        }
    return false;
}
int MamatchXatch()                          // 返回值表示最大匹配数
{
    /* 事先需要对xN和yN初始化，即算出X、Y两部分点的数目！ */
    int ret = 0;
    memset(matchX, -1, sizeof(matchX));
    memset(matchY, -1, sizeof(matchY));
    for(int x = 0; x < xN; x++)
        if(matchX[x] == -1)
        {
            memset(used, false, sizeof(used));
            if(SearchPath(x)) ret++;
        }
    return ret;
}
```

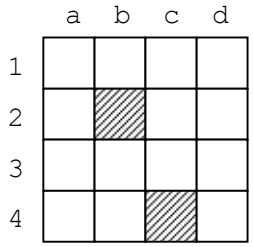
(4) 棋盘

【问题描述】已知 m 行 n 列的棋盘和其禁止落子的位置（**棋子同样无法通过**），求能够放到棋盘上且不会互相攻击的车的最大个数。如果换成 1×2 的、互不覆盖的骨牌，最多能放多少个？

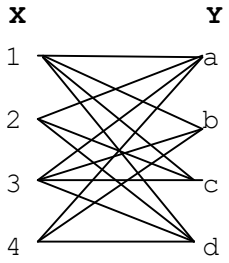
【分析】

对第一个问题建图：对应棋盘的每一行都有一个顶点属于 X 集合；对应棋盘的每一列，相应也有一个顶点属于 Y 集合；每个可以落子的位置对应一条边。这样就可以构建出一个二分图。

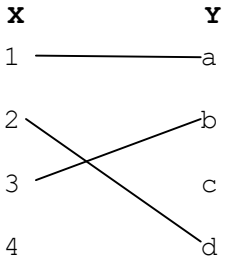
任意两车不同行也不同列（**除非中间有禁止落子的位置**），那么对应在二分图中任意两条边都不共享顶点。所以车相当于一个匹配。求出最大匹配，即车的最大个数。



(1)



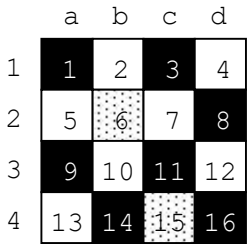
(2)



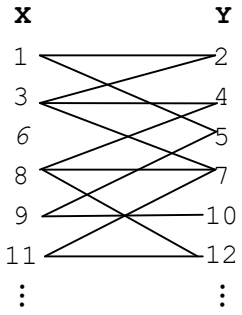
(3)

对第二个问题建图：将棋盘进行黑白染色，相邻的格子染成不同的颜色。染色后，格子分为黑白两类，对应二分图的 X 和 Y 集合，相邻的格子间在二分图中有一条边。

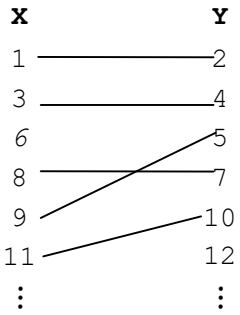
每张骨牌一定覆盖两个相邻的格子，因此每张骨牌对应一条边，最大匹配数即为所求骨牌数。



(1)



(2)



(3)

(5) 不要早恋

【问题描述】一个极度封建的老师要带同学们出去玩，但是他怕在途中同学之间发生恋情[①]。老师研究了一下发现，满足下面几种条件的两个同学之间发生恋情的可能性很小：

① 身高差＞40；② 性别相同（**同性恋？！**）； ③ 爱好不同的音乐； ④ 爱好同类型的运动。

给出这些同学的身高、性别、爱好的音乐、爱好的运动信息，问：按照这位老师的思想，有多少个同学不会“发生恋情”？

【分析】

显然如果我们用满足上面条件的同学之间建边那么最后建立起来的就不是二分图了。

[①] 这位老师真是麻烦——把男生和女生全部分开，问题不就全都解决了吗？

稍微观察一下，男生之间我们是随便带的，女生也是，因为他们彼此性别相同。因此我们就可以把男女分为两部分。

那么男女之间如何建边？如果我们把男女满足不发生恋情的连起来，那么求出来的最大匹配没有代表性，不能得到我们想要的结果。

因此我们用反建法，把男女中可能发生恋情的建立边。也就是说把身高差小于等于 40、爱好相同音乐或爱好不同类型运动的男女同学之间用边连起来。我们把可能发生恋情的男女相连，那么最大独立集就是不可能发生恋情的人的集合吗。

13.11　小结

1．图的概念

图可用 $G=(V,E)$ 来表示，其中 V 是顶点（vertex）的集合，E 是边（edge）的集合。E 中的每条边是 V 中的一对顶点 (u,v)。

如果 (u,v) 是无序对，则 G 是无向图，否则 G 是有向图。

如果任意两个顶点之间最多有一条边，并且没有由自己指向自己的边，那么 G 是简单图。绝大多数情况下，我们只研究简单图。

2．顶点和边

边数比较少的这图叫稀疏图，否则叫稠密图。

图可以用图形表示。顶点用圆圈表示，边用线段表示。如果是有向图，线段上要有表示方向的箭头。

如果 u 和 v 之间有一条边，则称 u 和 v 互为邻接顶点。

一个顶点 v 的度分为入度和出度。v 的入度是以 v 为终点的有向边的数量，v 的出度是以 v 为起点的有向边的数量。

某些图的边具有一个值，叫做权。带有权的图叫做带权图，或者叫做网络。

从顶点 v_1 出发，路过 v_a、v_b、……、v_k，到达了 v_2，我们就说 $(v_1v_av_b\cdots v_kv_2)$ 为从顶点 v_1 到 v_2 的路径。对于非带权图，路径的长度是路径上边的条数；对于带权图，路径的长度等于路径上边的权值的总和。

如果路径的起点和终点是同一个点，我们就说这个路径是回路或环。

按照某种顺序访问图中所有结点，这种操作叫图的遍历（**或者叫图的周游**）。常用的遍历方法有两种：深度优先遍历和广度优先遍历。

3．连通性

如果一个图的任意两个结点之间都有一条以上的路径相连，我们就称该图为连通图。

如果 $G=(V,E)$ 成立，且有集合 V' 是 V 集合的子集，集合 E' 是集合 E 的子集，那么我们就说图 $G'=(V',E')$ 是图 $G=(V,E)$ 的子图。

极大连通子图，可以理解为将连通子图扩展的尽量大。

非连通图的极大连通子图被称为连通分量；有向图中的连通图叫做强连通图；非强连通图中的极大连通子图被称为强连通分量。

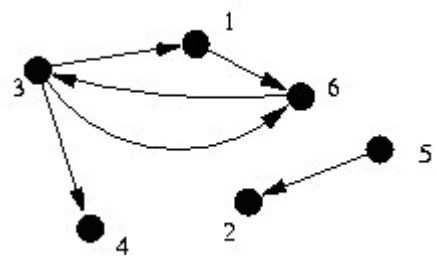
下面（**第 4 小节**）的图有 4 个强连通分量，分别是{1,3,6}、{4}、{2}、{5}。

一个连通图的生成树是该图的极小连通子图，在有 N 个结点的情况下，生成树仅有 $N-1$ 条边。

4．图的表示方法

选择一个好的方法来表示一张图是十分重要的，因为不同的图的表示方法有着不同的时间及空间需求。

通常，结点被用数字编号来表示，我们可以通过它们的编号数字来对他们进行索引。 这样，似乎所有问题都集中在如何表示边上了。



常用的表示法有三种：边目录、邻接矩阵、邻接表。

(1) 边目录：将所有边列成一张表，用结点来表示边。上图用边目录表示如下：

边的序号	起点	终点
1	3	4
2	3	6
3	3	1
4	1	6
5	6	3
6	5	2

(2) 邻接矩阵：邻接矩阵是一个 $n \times n$ 的数组（n **为结点数**）。

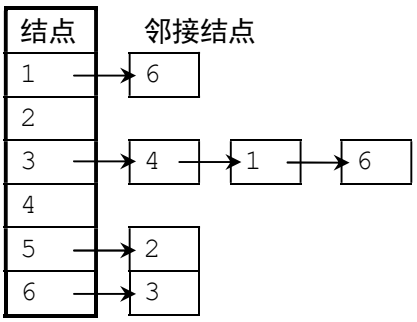
在非带权图中，如果 E 中存在边 (i,j)，则对应数组的 (i,j) 元素的值为 1，否则为 0；在带权图中，如果 E 中存在边 (i,j)，则对应数组的 (i,j) 元素的值为权，否则为 0（**或** INF）。

上图用邻接矩阵表示如下：

起\终	1	2	3	4	5	6
1	0	0	0	0	0	1
2	0	0	0	0	0	0
3	1	0	0	1	0	1
4	0	0	0	0	0	0
5	0	1	0	0	0	0
6	0	0	1	0	0	0

(3) 邻接表：用一个列表列出所有与现结点之间有边存在的结点名称。用一个长度为 n 的数组来实现（N **为结点个数**），数组的每一个元素都是一个链表表头。数组的第 i 元素所连接的链表连接了所有与结点 i 之间有边的结点名称[①]。

上图用邻接表表示如下：



[①] 当然邻接表可以和边目录配合使用。此时，邻接表内存储的是边的序号。

以上三种表示方法既可以用于有向图，也可以用于无向图。用于无向图时须进行特殊处理，例如加入两个互为反向的边。

（4）表示法的选择

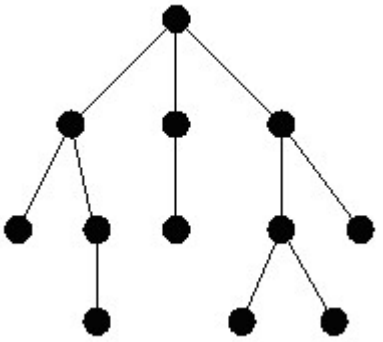
如果你发现一种表示方法可以在规定空间与时间范围内实现你的算法解决问题，并且可以使你的程序变得简洁、容易调试，那它通常就是对的。记住，编写与调试的简单是最重要的，我们不需要为一些毫无意义的加快程序速度，减少使用空间来浪费编写与调试的时间。

我们还要通过题目给我们的信息，确定我们要对图进行哪些操作，权衡后来决定表示方法。下表为三种表示方法进行某项操作的时空复杂度（N 为结点数，M 为边数，d 为图中度的最大值）：

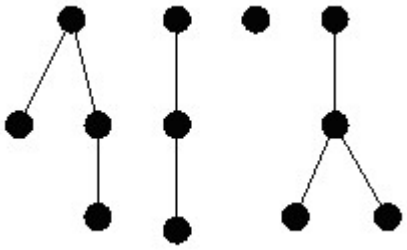
功效表	边目录	邻接矩阵	邻接表
消耗空间	$2M$	N^2	$2M$
连接判断的消耗时间	M	1	d
检查某结点所有相邻结点的消耗时间	M	N	d
添加边的消耗时间	1	1	1
删除边的消耗时间	M	2	$2d$

5．特殊的图

连通且无环的无向图被称作树。不连通且无环的无向图被称为森林。

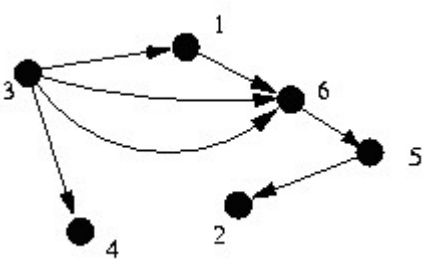


树



森林

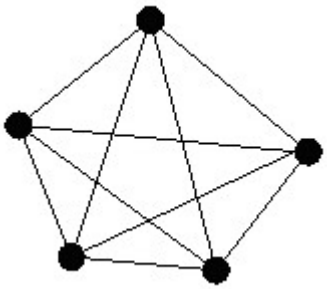
有向无环图（Directed Acyclic Graph）被称为 DAG，是他的英文缩写。



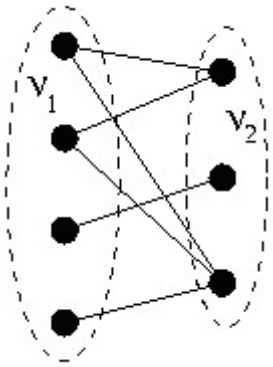
有向无环图

一个图的任意结点与其它所有结点有边相连的图被称为完全图。完全图有 $n(n-1)/2$ 条边。

如果一个图的所有结点可以被分为两个组，同组之间的任意结点都没有边相连，即所有边连接的都是不同组的两个结点，这样的图被称为二分图。



完全图



二分图

6．最短路径

两个顶点之间的最短路径是指图中连接这两个顶点的路径中代价最小的一条，一条路径的代价是指路径上所有边的权的和。

Dijkstra、Bellman-Ford 算法用于解决单源最短路问题，即源点（**固定起点**）到其他点的距离。前者应用贪心算法，后者用到了迭代法。不过，如果允许负权，那么 Dijkstra 算法就不再适用。

Floyd-Warshall 算法用于找出每对点之间的最短距离。这个算法是动态规划算法，它通过考虑最佳子路径来得到最佳路径。

如果图是非带权图，最短路径指路径中边的数量最少。这种情况下，用宽度优先搜索就可以了。

7．最小生成树

一棵生成树的图是一张无向的连通图。而一棵最小生成树是一棵权和最小的生成树（**即树所有的权值和最小**）。

Prim 算法和 Kruskal 算法都可以产生最小生成树，它们都利用了贪心算法。前者适合稠密图，而后者适合稀疏图。

第十四单元　STL 简介

STL（Standard Template Library，**标准模板库**）是 C++标准程序库的核心。现在，NOIP 对 STL 已经解禁。因为 STL 是 C++特有的库，所以在某种程度上，C++党比 C 党和 P 党更具有优势。

合理使用 STL 可以大大地简化代码，提高编程效率。例如 NOIP2004 的第二题《合并果子》，它用到了堆的建立和维护，这需要大量的代码。如果允许使用 STL 的优先队列（priority_queue），那么寥寥几行代码就可以解决问题了。

然而，我们要清楚地认识到，由于 NOIP 在编译时不加任何优化开关（-O2）[①]，STL 的速度甚至比直接编程还慢，真正的优势难以显现。并且，如果忘记某个容器的成员函数或某个算法的参数[②]，DEV-C++不可能让你很快找到它们的定义。所以我们在学习和使用 STL 时还要注意这两个问题。

STL 内容非常丰富，这里只介绍最常用的部分。如果想了解更多内容，可以浏览一些英文的专业资料。

14.1　STL 概述

STL 的一个重要特点是数据结构和算法的分离。尽管这是个简单的概念，但这种分离确实使得 STL 变得非常通用。例如，由于 STL 的 sort()函数是完全通用的，你可以用它来操作几乎任何数据集合，包括链表，容器和数组。

STL 另一个重要特性是它不是面向对象的。为了具有足够通用性，STL 主要依赖于模板而不是封装，继承和虚函数（多态性）。你在 STL 中找不到任何明显的类继承关系。这好像是一种倒退，但这正好是使得 STL 的组件具有广泛通用性的底层特征。另外，由于 STL 是基于模板，内联函数的使用使得生成的代码短小高效。

STL 的头文件与其他头文件不同，它是“开源”的。如果你想深入了解 STL 到底是怎么实现的，最好的办法是写个简单的程序，将程序中涉及到的模板源码给复制下来，稍作整理，就能看懂了。

STL 组件主要包括容器（Container）、迭代器（Iterator）、算法（Algorithm）等。容器是管理某类对象的集合，迭代器用于在对象集合上的遍历，而算法用于处理集合内的元素。

STL 组件都存在于 std 命名空间中，使用时不要忘记“using namespace std;”。

约定：

1．凡是带有“pos”、“begin”、“end”字样的都是迭代器。至于是什么迭代器，你应该能看出来。

2．“op”和“comp”是函数（**准确地说是函数对象**）。“op”一般是一元的，“comp”一般是二元的。

3．“fun”也是函数，只有返回值，没有形参。

4．“op”、“comp”一般代替默认运算符。但是“op”、“comp”、“fun”的返回值类型要结合具体问题来分析，不要一概而论。

5．“e”和“elem”表示容器内或即将要加入到容器内的元素。

14.2　常用容器

按照元素排列顺序的方式，容器可分为序列容器和关联容器。序列容器中元素的排列顺序取决于插入时机和顺序，而关联容器的排列顺序取决于特定准则。

常用的序列容器为 vector、deque、list；常用的关联容器为 map、multimap、set、multiset。

[①] 一旦使用此开关，就没有人能写出比 STL 还快的代码了。

[②] 忘记形参，你可以故意犯编译错误，使形参自己“跑”出来。忘记成员或函数的拼写，你就只能认倒霉了。

(1) STL 容器共同的操作！

下面的操作对常用的容器都适用。为了方便，就以 vector 为例。

1. 初始化
   - 产生一个元素为 int 类型的空容器：vector<int> v;
   - 以另一个容器元素（int）为初始值完成初始化：

```
list<int> l;
……
vector<int> v(l.begin(), l.end());
```

   - 以 int 类型数组元素为初值完成初始化：

```
int arr[]={1,2,3,4,5,6,7,8,9,10};
……
vector<int> v(arr, arr + sizeof(arr)/sizeof(int));
```

   注意，区间是左闭右开区间！
2. 与大小有关的操作
   - v.size()：返回 v 中的元素数量。
   - v.empty()：判断 v 中元素是否为空。
   - v.max_size()：返回 v 能容纳的最大元素数量。
3. 比较：==、!=、<、<=、>、>=

比较操作两端的容器必须是同一类型。比较运算将按字典序比较两个容器元素，当所有元素按序相等时，两容器才相等。

4. 赋值和交换
   - a=b：用 b 中元素取代 a。
   - a.swap(b)或 swap(a,b)：交换 a、b 中的元素。
5. 与迭代器（iterator）有关的操作
   - v.begin()：返回一个迭代器，指向第一个元素。
   - v.end()：返回一个迭代器，指向最后一个元素之后。
   - v.rbegin()：返回一个逆向迭代器，指向逆向遍历的第一个元素。
   - v.rend()：返回一个逆向迭代器，指向逆向遍历的最后一个元素之后。
6. 元素操作
   - v.insert(pos, e)：将元素 e 的拷贝安插于迭代器 pos 所指的位置。
   - v.erase(pos)：移除 pos 处的元素。
   - v.erase(begin, end)：移除[begin，end)区间内的所有元素。
   - v.clear()：移除所有元素。

(2) vector（矢量）

头文件：<vector>。

vector 的元素可以是任意类型 T，但必须具备赋值和拷贝能力（**具有 public 拷贝构造函数和重载的赋值操作符。**int、char、float、double **等已经具有这两个能力，无需重载**）。

vector 支持随机存取，甚至你完全可以把它视为一个元素个数可以变化的数组。

vector 的其他操作如下：

1. 内存分配
   - v.capacity()：返回重新分配空间前可容纳的最大元素数量。
   - v.reserve(n)：扩大容量为 n。

- 使用之前应该估计使用空间并把内存分配好。（**在信息学竞赛中，由于分配内存要耗费时间，所以 vector 并不是很常用。**）

2. 元素存取
   - v.at(i)：返回索引 i 所标识的元素。对 i 会进行越界检查。
   - v[i]：返回索引 i 所标识的元素，和数组用法几乎完全相同。对 i 不进行越界检查。
   - v.front()：返回第一个元素。不检查第一个元素是否存在。
   - v.back()：返回最后一个元素。不检查最后一个元素是否存在。
3. 插入元素
   - v.insert(pos, e)：在 pos 位置插入元素 e 的副本，并返回新元素位置。
   - v.insert(pos, n, e)：在 pos 位置插入 n 个元素 e 的副本。
   - v.insert(pos, begin, end)：在 pos 位置插入区间[begin, end)内所有元素的副本。
     注意，以上三种操作涉及大量元素的移动，可能会严重影响效率。
   - v.push_back(e)：在尾部添加一个元素 e 的副本。
4. 移除元素
   - v.erase(pos)：删除 pos 位置的元素，返回下一个元素的位置。
   - v.erase(begin, end)：删除区间[begin, end)内所有元素，返回下一个元素的位置。
     注意，以上两种操作涉及大量元素的移动，可能会严重影响效率。
   - v.pop_back()：移除最后一个元素但不返回最后一个元素。
   - v.clear()：移除所有元素，清空容器。
   - v.resize(num)：将元素数量改为 num（增加的元素用默认构造函数产生，多余的元素被删除）。
   - v.resize(num, e)：将元素数量改为 num（增加的元素是 e 的副本）。

   由 vector 产生的迭代器持续有效，除非发生以下两种情况：
   （1）插入元素。
   （2）容量变化而引起内存重新分配。

(3) deque（双端队列）

头文件：<deque>。

deque 和 vector 类似，但向 deque 两端添加或删除元素的开销很小。

deque 的内存管理比较复杂，随机访问的性能不如 vector，插入、删除的性能不如 list。如果不需要快速地从容器头部插入和删除数据，最好还是用 vector。

不要对 deque 使用迭代器！

deque 特有的操作如下：

- d[i]：返回 d 中下标为 i 的元素的引用。
- d.front()：返回第一个元素的引用。
- d.back()：返回最后一个元素的引用。
- d.pop_back()：删除尾部的元素。该函数没有返回值。
- d.pop_front()：删除头部的元素。该函数没有返回值。
- d.push_back(e)：在尾部添加一个元素 e 的副本。
- d.push_front(e)：在头部添加一个元素 e 的副本。

(4) list（表）

头文件：<list>

list 使用双向链表管理元素。显然 list 不支持随机存取，也不能对 list 使用“[]”运算符，但是元素的插入和删除速度很快。

list 的其他操作如下：

1. 元素存取
   - l.front()：返回第一个元素。不检查第一个元素是否存在。
   - l.back()：返回最后一个元素。不检查最后一个元素是否存在。
2. 插入元素
   - l.insert(pos, e)：在 pos 位置插入元素 e 的副本，并返回新元素位置。
   - l.insert(pos, n, e)：在 pos 位置插入 n 个元素 e 的副本。
   - l.insert(pos, begin, end)：在 pos 位置插入区间[begin, end)内所有元素的副本。
   - l.push_back(e)：在尾部添加一个元素 e 的副本。
   - l.push_front(e)：在头部添加一个元素 e 的副本。
3. 移除元素
   - l.pop_back()：移除最后一个元素。没有返回值。
   - l.pop_front()：移除第一个元素。没有返回值。
   - l.erase(pos)：删除 pos 位置的元素，返回下一个元素的位置。
   - l.erase(begin, end)：删除区间[begin, end)内所有元素，返回下一个元素的位置。
   - l.remove(val)：移除所有值为 val 的元素。
   - l.remove_if(op)：移除所有满足“op(val)==true”的元素。
   - l.clear()：移除所有元素，清空容器。
   - l.resize(num)：将元素数量改为 num（增加的元素用默认构造函数产生，多余的元素被删除）。
   - l.resize(num,e)：将元素数量改为 num（增加的元素是 e 的副本）。
4. 其他操作
   - l.unique()：移除重复元素。
   - l.unique(op)：移除满足“op(val) == true”的重复元素。
   - l1.splice(pos, l2)：将 l2 内的所有元素转移到 l1 的迭代器之前。
   - l1.splice(pos, l2, l2pos)：将 l2 内 l2pos 所指元素转移到 l1 内的 pos 之前。
   - l1.splice(pos, l2, l2begin, l2end)：将 l2 内[l2begin, l2end)区间内所有元素转移到 l1 的 pos 之前。
   - l.sort()：以 operator <为准则对所有元素排序。
   - l.sort(op)：以 op（**定义“小于”关系**）为准则对所有元素排序。
   - l1.merge(l2)：假设 l1 和 l2 都已排序，将 l2 全部元素转移到 l1 并保证合并后仍是有序表。
   - l.reverse()：将所有元素反序

无论是安插，还是删除，指向其他元素的指针、引用和迭代器都不会失效。

(5) 关联容器

map（**映射**）、multimap（**多重映射**）、set（**集合**）、multiset（**多重集合**）都用平衡二叉树（**红黑树**）来实现。

先介绍 map 和 set 的区别。set 实际上就是一组元素的集合，但其中所包含的元素的值是唯一的，且是按一定顺序排列的。集合中的每个元素被称作集合中的实例。其内部通过链表的方式来组织。

map 提供一种“键—值”关系的一对一的数据存储能力，类似于字典。其“键”在容器中不可重复，且按一定顺序排列。由于其是按链表的方式存储，它也继承了链表的优缺点。

multiset 和 set 不同之处在于，multiset 中元素的值可以不唯一。multimap 也类似，在 multimap 中“键”可以不唯一。

关联容器的特点：

1. 关联容器对元素的插入和删除操作比 vector 快，但比 list 慢。
2. 关联容器对元素的检索操作比 vector 慢，但是比 list 要快很多。关联容器查找的复杂度基本是

$O(\log n)$。

3. set 区别于 vector、deque、list 的最大特点就是 set 是内部排序的，这在查询上虽然逊色于 vector，但是却大大的强于 list。

头文件：<map>和<set>。

以下内容适用于 set 和 multiset：

1. 定义
   - 内部升序排列：set< int, less<int> > s;
   - 内部降序排列：set< int, greater<int> > s;
   - 和其他容器一样，set 也可以用预定义的区间来初始化。
2. 查询
   - s.count(10)：返回 s 中值为 10 的具体数目。但对于 set 来说，返回值不是 0 就是 1。
   - s.empty()：判断集合是否为空集。
   - s.size()：返回集合的元素数量。
3. 插入和删除
   - s.insert(e)：将 e 插入到 set 中。
     返回值是一个 pair[①]，其 first 是指向插入后元素的迭代器，second 表示插入是否成功（如果其 second 为 false，说明元素已经存在）。
   - s.insert(begin, end)：将区间[begin, end)中的值插入到 s 中。
   - s.erase(e)：将 e 删除。返回值为 e 的数量（**对于 multiset 来说，被删除的可能不止一个数**）。
   - s.erase(pos)：将 pos 处的元素删除。
   - s.erase(begin, end)：将[begin, end)处的元素删除。
4. 迭代器：cbegin、cend、crbegin、crend 返回只读迭代器。

以下内容适用于 map 和 multimap：

1. 定义
   - map<int, string> m：前面的 int 是键的类型，后面的 string 是值的类型。
   - 和其他容器一样，set 也可以用预定义的区间来初始化。
2. 查询
   - m.at(3)或 m[3]：返回一个引用，指向键为 3 时的对应值。**注意，它和数组下标完全不是一回事儿！**
     如果元素不存在，map 会自动建立这个元素。
   - m.count(3)：返回 s 中键为 3 的具体数目。但对于 map 来说，返回值不是 0 就是 1。
   - m.find(3)：返回指向键为 3 的元素的迭代器。如果不存在，则返回 m.end()。
   - m.empty()：判断映射是否为空映射。
   - m.size()：返回映射的元素数量。
3. 插入和删除
   - m.insert(pair)：将元素插入到 set 中。pair 的 first 是键，second 是值。
     可以定义一个 pair：pair <int, string> p(10, "Hello");
     也可以用 make_pair（<utility>）建立一个 pair：make_pair(10, "Hello");
   - m.insert(begin, end)：将区间[begin, end)中的值插入到 s 中。该区间应该是 map 类型的。
   - m.erase(e)：将键为 e 的元素删除。返回值为被删除的 e 的数量（**对于 multimap 来说，被删除的可能不止一个元素**）。
   - m.erase(pos)：将 pos 处的元素删除。

[①] pair 是一个将两种类型“捆绑”到一块的类型。它有两个成员：first 和 second，分别表示这两种类型的值。

- m.erase(begin, end)：将[begin, end)处的元素删除。

4．迭代器：cbegin、cend、crbegin、crend 返回只读迭代器。

(6) 功效表

	vector	deque	list	map/multimap	set/multiset
内部结构	动态数组	数组的数组	双向链表	平衡二叉树	平衡二叉树
元素形式	值	值	值	键—值	值
元素是否可以重复	√	√	√	map：×（键） multimap：√	set：× multiset：√
可随机存取	√	√	×	map：√（键） multimap：×	×
迭代器类型	随机存取	随机存取	双向	双向 键被视为常数	双向 值被视为常数
元素搜寻速度	慢	慢	非常慢	对键来说快	快
在哪里安插、移除速度快	尾部	头尾两端	任何位置	×	×
何时安插、移除会导致迭代器失效	重新分配时	任何时候	不会	不会	不会

(7) 字符串

头文件：<string>

可以把字符串理解为 vector<char>。vector 上的各种操作同样适用于 string。

假设待处理的字符串为 str 和 str2，即：string str, str2;

1．输出字符串 str：cout<<str;

2．输入字符串 str：

- cin>>str;

  输入时会忽略空格、回车、TAB 等字符，并且在一个或多个非空格字符后面输入空格时，会终止输入。
- getline(cin, str, '\n');　　// 遇到'\n'，即回车才终止读入

3．求字符串 str 的长度：str.length()

4．访问序号为 10 的字符：str[10]　　// 和字符数组一样

5．提取子串：str.substr(a, b);　　// 返回值是从第 a 个字符开始，长度为 b 的子串。

6．连接字符串：可以直接使用“+”或“+=”运算符连接字符串。

- 由于“+”是通过运算符重载实现的，所以两个操作数中至少有一个是 string 对象。"a"+"b"就是非法的。
- “+”涉及新对象的创建，所以会引起一定数量的开销。

7．在 str 末尾追加字符/字符串：

- str.append(字符串对象);　　// 追加字符串
- str.append("字符串");
- str.append(10, 'a');　　// 追加 10 个 a
- str.append(str2, 3, 5);　　// 从 str2 的第 3 个字符开始，取 5 个字符，追加到 str 末尾
- str.push_back('*');　　// 追加单个字符

8．在 str 的第 k 位置插入字符串：

- str.insert(k, str2);　　// 插入 str2
- str.insert(k, "字符串");
- str.insert(k, str2, n);　　// 插入 str2 的前 n 个字符

- str.insert(k, "字符串", n);　　// 插入字符数组中的前 n 个字符
- str.insert(k, str2, a, b);　　// 相当于 str.insert(k, str2.substr(a,b));

9. 用 str2 的一部分替换 str 的一部分：
   - str.replace(k, n, str2);　　// 用 str2 的前 n 个字符替换 str 中从 k 开始的 n 个字符
   - str.replace(k, n, "字符串");
   - str.replace(k, n1, str2, n2);
     // 用 str2 的前 n2 个字符替换 str 中从 k 开始的 n1 个字符
   - str.replace(k, n1, str2, k2, n2);
     // 相当于 str.replace(k, n1, str2.substr(k2,n2))
   - str.replace(k, n1, n2, '**');　// 用 n2 个'*'替换 str 中从 k 开始的 n 个字符
10. 把 str 转换成字符数组：str.c_str()
11. 字符串比较：可以直接使用==、!=、>、<、>=、<=运算符，因为它们已经被重载。
12. 搜索子串：
    - str.find('a', k);　　// 从 k 开始寻找'a'。k 可以省略，代表从头开始搜索。
    - str.find("字符串", k);
    - str.find(str2, k);
      如果找到了对应的子串，就返回索引位置，否则返回 string::npos。

14.3　容器适配器

容器适配器包括 queue、stack、priority_queue。适配器是容器的接口，它本身不能直接保存元素，它保存元素的机制是调用另一种顺序容器去实现，即可以把适配器看作“它保存一个容器，这个容器再保存所有元素”。

STL 中提供的三种适配器可以由某一种顺序容器去实现。默认下 stack 和 queue 基于 deque 容器实现，priority_queue 则基于 vector 容器实现。当然在创建一个适配器时也可以指定具体的实现容器，创建适配器时在第二个参数上指定具体的顺序容器可以覆盖适配器的默认实现。

由于适配器的特点，一个适配器不是可以由任一个顺序容器都可以实现的。

栈 stack 的特点是后进先出，所以它关联的基本容器可以是任意一种顺序容器，因为这些容器类型结构都可以提供栈的操作有求，它们都提供了 push_back、pop_back 和 back 操作。

队列 queue 的特点是先进先出，适配器要求其关联的基础容器必须提供 pop_front 操作，因此其不能建立在 vector 容器上。

(1) queue（队列）！

头文件为<stack>。

1. q.push(a)：使 a 入队。
2. q.front()：返回队列头部元素，但是元素不出队。
3. q.back()：返回一个引用，指向队列尾部元素，但是元素不出队。
4. q.pop()：元素出队。该函数没有返回值。
5. q.size()：返回队列中元素个数。
6. q.empty()：判断队列是否为空。

(2) stack（堆栈）！

头文件为<stack>。

1. s.push(a)：使 a 入栈。
2. s.top()：返回栈顶元素，但是元素不出栈。
3. s.pop()：元素出栈。该函数没有返回值。

4. s.size()：返回堆栈中元素个数。
5. s.empty()：判断堆栈是否为空栈。

(3) priority_queue（优先队列）！

优先队列和队列不同，队列中元素按照进入的先后顺序出队，而优先队列中元素按照优先级出队。优先队列用二叉堆实现。

头文件：priority_queue 存在于<queue>，vector 存在于<vector>，greater 和 less 存在于<functional>。

1. 定义一个优先队列
   - 最小值优先出队：priority_queue< int,vector<int>,greater<int> > q;
   - 最大值优先出队：priority_queue< int,vector<int>,less<int> > q;
   - 定义队列时要注意，后面两个“>”间有一个空格。如果没有空格，编译器会将其误认为“>>”运算符而无法正确编译。
2. q.push(a)：使 a 入队。
3. q.top()：返回优先级最高的元素，但不会移除元素。
4. q.pop()：移除优先级最高的元素。该函数没有返回值。
5. q.empty()：判断队列是否为空。
6. q.size()：返回队列中已有元素的个数。

如果需要使用自己的结构体，你需要重载复制构造函数和“>”（“<”）运算符。使用 less（**最大值先出队**），则需要重载“<”运算符；使用 greater（**最小值先出队**），则需要重载“>”运算符。例如：

```
struct MyStruct
{
    int v;
    MyStruct(int i):v(i) {}

    bool operator < (const MyStruct & b) const {return v < b.v;}
};
priority_queue < MyStruct,vector<MyStruct>,less<MyStruct> > q;
```

14.4　常用算法

STL 提供了一些标准算法来处理容器内的元素，例如搜寻、排序、复制、数值运算等。算法所在的头文件为<algorithm>，涉及数值运算的算法位于<numeric>中。

切记，STL 中所有算法的处理都是左闭右开区间[begin, end)！

(1) for_each（遍历）

```
for_each(begin, end, op);
```

- 作用：对区间[begin, end)中的每一个元素 elem 调用 op(elem)。
- 返回值：返回进行 op 操作之后的容器副本。
- op 可以改变元素的值，但是 op 的返回值将被忽略。
- 时间复杂度：线性，与区间元素个数成正比。

(2) 非变动性算法

非变动性算法既不改变元素次序也不改变元素值。

1. 元素计数
   - count(begin, end, value)：计算区间中值为 value 的元素个数。
   - count(begin, end, op)：计算区间中满足“op(elem) == true”的元素个数。
2. 最值
   - min_element(begin, end, *comp*)：求区间最小元素的位置。
   - max_element(begin, end, *comp*)：求区间最大元素的位置（**都是迭代器**）。
   - 如果省略 comp，将使用默认的 operator <和 operator >进行比较。
3. 搜寻元素
   - find(begin, end, value)：返回区间中第一个元素值等于 value 的元素位置。
   - find_if(begin, end, op)：返回区间中第一个满足“op(elem) == true”的元素位置。
   - **如果没有找到匹配元素，则返回你所传入的 end，不是 NULL！后面所有涉及查找的算法都是如此！**
4. 匹配
   - search(begin1, end1, begin2, end2, *comp*)：返回区间[begin2, end2)在区间[begin1, end1)中第一次出现的位置。
   - search_n(begin, end, n, v, *comp*)：搜寻区间[begin, end)中第一段“n 个连续元素 v”。
   - 如果省略 comp，将使用默认的 operator ==作为比较依据。
5. 判断是否等价
   - equal(begin1, end1, begin2)：判断[begin1, end1)内每个元素是否和从 begin2 开始的对应元素相等。

以上几个算法的时间复杂度都是 O(n)。

(3) 变动性算法

变动性算法直接改变元素值或者在复制到另一区间的过程中改变元素值。

1. 复制
   - copy(begin1, end1, begin2)：将[begin1, end1)区间内的元素复制到 begin2 位置之后[①]。
   - copy_backward(begin1, end1, end2)：将[begin1, end1)区间内的元素复制到 end2 位置之前。
   - copy_if(begin1, end1, pos, op)：将[begin1, end1)区间内满足“op(elem) == true”的元素复制到 pos 位置之后。
2. 变换
   - transform(begin, end, pos, op)：对[begin1, end1)的元素进行 op 一元变换，并复制到 pos 处。
   - transform(begin1, end1, begin2, pos, op)：对[begin1, end1)的元素和从 begin2 开始的区间的对应元素进行 op 二元变换，并复制到 pos 处。
     常用的二元变换（<functional>）：plus、minus、multiplies、divides，使用时要指明类型，例如 plus<int>。
3. 交换
   - swap(a, b)：交换两个容器的值。容器的类型必须相同。
   - swap_ranges(begin1, end1, begin2)：交换两个区间的元素。
4. 填充
   - fill(begin, end, value)：以给定值 value 替换[begin, end)的每个元素。

[①] 凡是把区间复制到某个迭代器上的函数，返回值都是指向目标位置上、最右端元素右侧的迭代器。后面不再重复解释。

- fill_n(pos, n, value)：以给定值 value 替换从 pos 开始的 n 个元素。
- generate(begin, end, fun)：以某项操作的结果替换[begin, end)的每个元素。
  常用操作是填充随机数：generate(v.begin(), v.end(), rand);
  这是一个诡异的操作，它不用 for 却能从文件中读入一组数：generate(v.begin(), v.end(), istream_iterator<int>(cin, int));
- generate_n(pos, n, fun)：以某项操作的结果替换从 pos 开始的 n 个元素。

5. 替换

- replace(begin, end, oldval, newval)：将[begin, end)的所有 oldval 换成 newval。
- replace_if(begin, end, op, newval)：将[begin, end)的所有符合“op(elem) == true”的值换成 newval。

(4) 移除性算法

移除性算法移除某区间内的元素或者在复制过程中移除元素值。

- remove(begin, end, value)：移除区间[begin, end)内和 value 相等的元素。
- remove_if(begin, end, op)：移除区间[begin, end)内满足“op(elem)==true”的元素。
  remove 和 remove_if 只是将未移除元素向前移动，覆盖移除元素，并返回新的终点，并没有真正删除。真正删除元素，需要使用容器的 erase。
- remove_copy(begin, end, pos, value)：将不等于 value 的元素全部复制到 pos 处。
- remove_copy_if(begin, end, pos, op)：将不满足“op(elem)==true”的元素全部复制到 pos 处。
- unique(begin, end, *op*)：移除相邻的重复元素。op 代替默认的 operator ==。
- unique_copy(begin, end, pos, *op*)：移除相邻的重复元素，并复制到 pos 处。op 代替默认的 operator ==。
  注意，如果想删除所有重复元素，需要先排序！

(5) 变序性算法

变序性算法通过元素的赋值和交换改变元素顺序。

- reverse(begin, end)：将元素的次序逆转（**使用双向迭代器**）。
- reverse_copy(begin, end, pos)：逆转元素次序，将结果复制到 pos 处。
- rotate(begin, middle, end)：将[first, middle)区间内的元素和[middle, end)区间内的元素交换。
- rotate_copy(begin, middle, end, pos)：旋转的同时将结果复制到 pos 处。
- random_shufle(begin, end)：将元素次序随机打乱。
- partition(begin, end, op)：通过交换元素使符合“op(elem) == true”的元素移到前面。这是快速排序中的一步。
- stable_partion(begin, end, op)：与 partion 类似，但保持符合准则与不符合准则元素的相对位置。

(6) 排序算法!

排序算法需要动用随机存取迭代器，所以不能对 list、set、map 等使用排序算法。

1. 对所有元素排序：

- sort(begin, end, *comp*)：对区间[begin, end)内的所有元素排序。如果省略 comp，将使用默认的 operator <进行比较。
- stable_sort(begin, end, *comp*)：和 sort 类似。区别是 stable_sort 会保持相等元素原

来的相对次序。

以上两个算法的时间复杂度都是 O($n$log$n$)。

2．局部排序：

- partial_sort(begin, sortEnd, end, *comp*)：对区间[begin, end)内的元素排序，使区间[begin, sortEnd)内的元素有序。如果省略 comp，将使用默认的 operator <进行比较。

时间复杂度在 O(n)与 O($n$log$n$)之间。

3．根据第 n 个元素排序：

- nth_element(begin, pos, end, *comp*)：对区间[begin, end)内的元素排序，使所有在位置 pos 之前的元素都小于等于它，所有在位置 pos 之后的元素都大于等于它，从而得到两个分隔开的子序列。如果省略 comp，将使用默认的 operator <进行比较。

平均时间为 O(n)。

(7) 有序区间算法

区间已经排序了。

1．二分查找：

- binary_serach(begin, end, value, *comp*)：判断已序区间[begin, end)内是否包含和 value 相等的元素。如果省略 comp，将使用默认的 operator <进行比较。
  返回值只说明搜寻的值是否存在，不指明位置。
- lower_bound(begin, end, value, *comp*)：返回第一个大于等于 value 的元素位置，即可以插入 value 且不破坏区间有序性的第一个位置。
- upper_bound (begin, end, value, *comp*)：返回第一个大于 value 的元素位置，即可以插入 value 且不破坏区间有序性的最后一个位置。

以上几个算法，如果使用随机迭代器，则为对数复杂度，否则为线性复杂度。

2．其他

- equal_range(begin, end, value)：返回全部等于给定值的元素构成的区间。返回值类型为 pair，其 first 和 second 分别表示区间的始端和末端。
- merge(begin1, end1, begin2, end2, pos, *comp*)：将两个有序区间的元素合并到 pos 处，如果省略 comp，将使用默认的 operator <进行比较。

(8) 集合算法

区间应该是已经排好序的。如果用 set，顺序就自动排好了。

- includes(begin1, end1, begin2, end2, *comp*)：判断第二个区间是否为第一个区间的子集。如果省略 comp，将使用默认的 operator <作为比较依据。
- set_union(begin1, end1, begin2, end2, pos, *comp*)：构造一个已排序的并集。返回一个迭代器 pos2，使并集存在于[pos, pos2)中。
- set_intersection(begin1, end1, begin2, end2, pos, *comp*)：构造一个已排序的交集。
- set_difference(begin1, end1, begin2, end2, pos, *comp*)：构造一个已排序序列，包含在第一个序列但不在第二个序列的元素。
- set_symmetric_difference(begin1, end1, begin2, end2, pos, *comp*)：构造一个已排序序列，包括所有只在两个序列之一中的元素。

(9) 堆算法

优先队列就是用二叉堆来实现的，为了更好地使用 STL，我们选择堆！堆的代码也不繁琐，只要找到一个支持随机存取的容器……

- 堆的特点：

① 第一个元素总是最大。

② 总是能够在对数时间内增加或移除一个元素。

- make_heap(begin, end, *comp*)：将某区间内的元素转化为堆。时间复杂度 O(*n*)。如果省略 comp，将使用默认的 operator <作为比较依据。
- push_heap(begin, end, *comp*)：[begin, end-1)原本就是堆。现在将 end 之前的那个元素加入堆中，使区间[begin, end)重新成为堆。时间复杂度 O(log*n*)。
- pop_heap(begin, end, *comp*)：从区间[begin, end)取出第一个元素（**最大值**），放到最后位置，然后将区间[begin, end-1)重新组成堆。时间复杂度 O(log*n*)。
- sort_heap(begin, end, *comp*)：将 heap 转换为一个有序集合。时间复杂度 O(*n*log*n*)。

(10) 数值算法

以不同的方式组合数值元素。头文件<numeric>，另外，几个函数对象在<functional>中。

- accumulate(begin, end, initValue, *op*)：返回一个数值。
  对于一个序列(a1,a2,a3,…)，如果有 op，将计算 initValue op a1 op a2 op a3 op …；
  如果没有 op，计算 initValue + a1 + a2 + a3 + …
- adjacent_difference(begin, end, pos, *op*)：使区间内的每个元素和它的前一个元素发生 op 作用，将结果复制到 pos 中。
  如果没有 op，则求每个元素减去其前一元素之后的差。
- partial_sum(begin, end, pos, *op*)：将每个元素和其先前所有元素组合，结果复制到 pos 中。如果没有 op，就直接求和。
- inner_product(begin1, end1, begin2, initValue)：返回一个数值。
  计算 initValue + (a1*b1) + (a2*b2) + (a3*b3) + …
- inner_product(begin1, end2, begin2, initValue, op1, op2)：返回一个数值。
  计算 initValue op1 (a1 op2 b1) op1 (a2 op2 a2) op1 …
- 以上几个 op 是二元函数。此外，还有四个常用的函数：plus、minus、multiplies、divides。
  使用时注意标明类型，如 plus<int>。

后记：算法很多，如果想记住，首先要做一件很重要的事——背单词。如果你能发现各算法的形参的共同点，你就不觉得算法难记了。

14.5　迭代器

头文件：<iterator>

可将迭代器理解为指针，也可以像使用指针一样使用迭代器。迭代器对于访问除容器适配器定义的内容之外的所有 STL 容器的内容非常重要。容器适配器不支持迭代器。

迭代器有五类，它们所支持的操作如下表所示（**假设 p 是迭代器**）：

迭代器类型	输出迭代器	输入迭代器	前向迭代器	双向迭代器	随机迭代器
缩写	Output	Input	Forward	Bidirectional	Random
读取	不支持	x = *p	x = *p	x = *p	x = *p
操作	不支持	p->a（**假如 a 是 p 所属类**	p->a（**假如 a 是 p 所属类型的成员**）	p->a（**假如 a 是 p 所属类型的成员**）	p[i] p->a（**假如 a 是 p 所属类型的成员**）

		型的成员）			
写入	*p = x	不支持	*p = x	*p = x	*p = x
迭代	++	++	++	++、--	++、--、+、-、+=、-=
比较	不支持	==、!=	==、!=	==、!=	==、!=、<、>

迭代器具有兼容关系，且兼容关系可以传递：前向迭代器兼容输出迭代器和输入迭代器，双向迭代器兼容前向迭代器，而随机迭代器则兼容双向迭代器。

迭代器不但可以用来访问元素，在整个 STL 中，它还起到了纽带的作用，把算法和容器紧紧地联系在了一起。

14.6　示例：合并果子[①]

【问题描述】

有 n 堆（$1 \leqslant n \leqslant 10000$）果子，每堆重量已知。每次可以把两堆果子合并，该次消耗的体力为两堆果子重量之和。在合并果子时总共消耗的体力等于每次合并所耗体力之和。

现在要把这堆果子合并成一堆。求最小的体力耗费值。

【分析】

很明显是一道贪心题：每次取数量最少的两堆，合并，再放回去，排好序。说起来容易，做起来就没那么简单了——在把合并后的堆放回去之后，要对那些果子进行局部排序……

现在，用 STL 完成！

【代码】

```
#include <iostream>
#include <queue>
#include <functional>
using namespace std;

priority_queue < int, vector<int>, greater<int> > q;

int main()
{
    int n, temp, r=0;
    cin>>n;
    for (int i=0; i<n; i++) { cin>>temp; q.push(temp); }

    while (q.size()>1)
    {
        int a=q.top(); q.pop();
        int b=q.top(); q.pop();
        q.push(a+b);
        r+=a+b;
    }

    cout<<r<<endl;
```

[①] 题目来源：NOIP2004 第二题。当时不允许使用 STL，否则……

```
    return 0;
}
```

附录 A 思想和技巧

A.1 时间/空间权衡

(1) 时间换空间

信息学竞赛的特点是内存广阔，时间紧张，因此这种思想的应用不是太广泛。

“时间换空间”典型实例是迭代加深搜索：当广度优先搜索的空间开销太大，以至于无法承受时，可以考虑“模拟广搜的深搜”，即迭代加深搜索。

(2) 空间换时间

与上一种思想相比，这种思想广泛应用于信息、学的各个方面，如动态规划。

一个体现“空间换时间”思想的常用做法是保存中间结果——把频繁使用的数值，或者需要重复计算的内容保存下来，以便下次直接使用。

A.2 试验、猜想及归纳

有时，当问题过于复杂，无从下手，无法联系自己的知识时，我们往往希望“尽己所能，先解决规模更小的问题，找找问题是否存在规律吧”。这种“缩小规模”、“找规律”的想法就是试验思维。试验和猜想经常是结合在一起的。

遇到一个问题，基础好、经验丰富的同学更容易形成猜想。所以，多做题还是很有意义的。

【问题简述】[①] m，$n\in N$，$1\leqslant m$，$n\leqslant k\leqslant 10^9$，且 $(n^2-mn-m^2)^2=1$，输入 k，求 m，n 使 m^2+n^2 最大。

【分析】

从数据的规模可“猜想”本题一定蕴含着更为简单的数学关系，但是题目的数学关系不明显，数学分析的难度太大。而用简单的两重循环枚举可以很方便的算出小数据的情况。

k	1	2	3,4	5,6,7	8,9,10,11,12	13…
m	1	2	3	5	8	13…
n	1	1	2	3	5	8…

通过这些试验，我们猜想符合条件的 m，n 成斐波那契数列。实际上：

$n^2-mn-m^2=-(m^2+mn-n^2)=-[(m+n)^2-mn-2n^2]=-[(m+n)^2-n(m+n)-n^2]$

A.3 模型化

当我们面对一个新问题时，通常的想法是通过分析，不断的转化和转换，得到本质相同且熟悉的或抽象的、一般的一个问题。这就是化归思想。

模型化的方向主要有图论模型、数学模型和规划（**动态规划**）模型。

图论模型是最用的模型，并且可利用的图论算法很多。通过图论，本来复杂、凌乱的数据关系可以变得简洁、明了了，这样问题求解的思路也清晰了。

在数学模型中，使用得最多的是各种组合数学模型。当然还有分类、分治、递归等思维。

对于计数问题，首先可以考虑建立组合数学模型，最常用的组合数学模型是递推关系。

常见的规划模型主要包括整数规划及动态规划模型。整数规划是所有变量均取整数的规划问题，这类问题的算法通常是阶乘级的。而动态规划的算法复杂度却是多项式级的。带约束的多变量的求解问题，特别是约束

[①] 题目来源：NOI'95

条件中有满足某个函数的最大（小）值，可考虑建立规划模型。建立规划模型时，动态规划是首选，如果选整数规划，则要注意算法的优化。

下面举一个简单的例子[①]：

【问题简述】N 位同学（$N \leqslant 100$）站成一排，音乐老师要请其中的 $(N-K)$ 位同学出列，使得剩下的 K 位同学排成合唱队形——若这 K 的个人的身高分别为 T_1、T_2、…、T_K，则他们的身高满足

$$T_1 < \cdots < T_i > T_i + 1 > \cdots > T_K \ (1 \leqslant i \leqslant K)。$$

已知所有 N 位同学的身高，问最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

【分析】

很明显，这是最长非降子序列的模型。不过这次是两个相连的序列。

最基本的想法是：枚举中间最高的一个人，接着对它的左边求最长上升序列（**注意序列中最高的同学不应高过基准**），对右边求最长下降序列（**同样的，序列中最高的同学不应高过基准**）。时间复杂度为 $O(n^3)$，算法实现起来也很简单。

接着对这个算法进行分析。设 $a[i]$ 表示了 1～i 的最长上升序列，$b[i]$ 表示了 i～n 的最长下降序列，那么，$f[i]=a[i]+b[i]-1$（**两个数组中 i 被重复计算了**）。

那么，我们只需要先求好最长上升和下降序列，然后枚举中间最高的同学。

A.4 随机化*

【问题简述】[②]给出一棵传染病传播树，其中根结点是被感染结点。每个时刻，被感染结点都会将传染病传播到它的子结点。但是，如果某时刻 t 切段 A 与其父结点 B 之间的边，则时刻 t 以后，传染病不会从 B 传染给 A（即 A **不会患病**）。

每个时刻，只能切断一条传播途径。问每个时刻怎样切段传播途径，使最少的人被感染。

(1) 随机化搜索

处理每一层时，都任意切断一条转播途径。当没有传播途径可以被切断时，更新被感染人数的最小值。重复若干次，基本上就能得到正确答案。

(2) 随机＋贪心

由于贪心算法的时间复杂度非常低，所以可以考虑在策略选择中引入随机因子，并通过大量重复运算来获得最优解。

就本题而言，根据经验，每次将一个子孙数量最多的结点从原树中切开，结果会比较好。但这样并不能适应所有情况，有时，选子孙数第二多的或第三多的反而能使以后的最终结果要好一些。所以，可以多次贪心，每次使子孙数量最多的结点被切的几率大一些。

A.5 动态化静态

动态内存分配的速度是比较慢的，且容易造成内存泄露，甚至产生错误结果。

[①] 题目来源：NOIP2004 第三题
[②] 题目来源：NOIP2003 第四题

如果想使用链表、二叉树等动态结构，可以事先开一个较大的静态数组，需要分配内存时，就像下面一样模拟系统的动态内存分配。

使用下面的NEW宏分配内存时，切记：① arr数组要足够大；② 传入的“参数”必须是**指针**！

```
struct node
{
    int v;
    node * next;
} a[100000], *head, *p, *q;
int top=-1;
// 用NEW(p)“动态”分配内存。数组的空间是事先开好的，引用一下就行了。
#define NEW(p)       p=&a[++top];p->v=0;p->next=NULL;
……
NEW(head);
NEW(p);
head->next=p;
……
```

A.6 前序和！

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。

(1) 区间求和

【问题描述】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（$0<n\leqslant 1000000$，$0<m\leqslant 1000000$，$0<i<j\leqslant n$）。

【分析】

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。你需要记住以下做法：

1. 用 $S[i]$ 表示前 i 个数字的和。
2. $S[i]$ 的递推公式：$S[i]=S[i-1]+a[i]$，边读入数据边求出 S。
3. 第 i 项到第 j 项中所有元素的和$=S[j]-S[i-1]$。用O(1)的时间就算出来了。

(2) 带条件的区间求和[①]

问题见61页“5.8 聪明的质监员”。这里讲述如何快速求出 Y_i 的值。

由于对每个区间求和，W 都不发生变化，所以可以预处理，把不符合条件的矿石“抹掉”，计算前序和时不把它们包括在内。求和的时候，由于这些矿石是“打酱油的”，所以不会对结果造成影响。

设sumN[i]表示从1到i中合格的矿石数量，sumV[i]表示从1到i中合格的矿石的总价值。

那么合格矿石就有sumN[i]=sumN[i-1]+1，sumV[i]=sumV[i-1]+v[i]，

不合格矿石的sumN[i]、sumV[i]和sumN[i-1]、sumV[i-1]，相当于直接无视。

最后，$Y_i=\sum_j 1\times\sum_j v_j$=(sumN[r]-sumN[l-1])×(sumV[r]-sumV[l-1])

(3) 最大子矩阵和

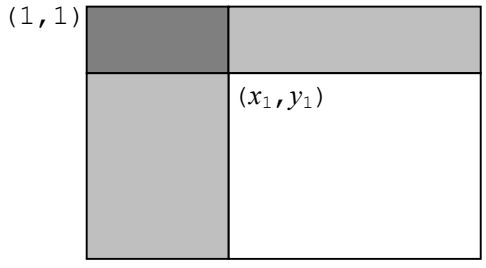
【问题描述】给出一个 $m\times n$ 的矩阵。求一个子矩阵，使子矩阵中每个数加到一起的总和最大。

[①] 本题是NOIP2011 day2《聪明的质监员》的子问题。

【分析】

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。

最大子序列和可用动态规划求出：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。

A.7 状态压缩*

(1) 集合的表示方法

1. 建立一个集合a（unsigned int[①]），规定它有n个（$1\leqslant n\leqslant 32$）元素。

   那么，第i位（$0\leqslant i<n$）如果为 1，就代表“这里有元素”，否则代表“这里没有元素”。当然，也可以理解为“第i个元素在这里”和“第i个元素不在这里”。

序号	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
元素	0	0	0	1	1	0	1	0	1	0	0	1	1	1	1	0	0	1	0	1	0	0	0	1	0	1	0	0	1	0	1	1

   - 当然，可以用多个 bit 代表一个元素的不同状态。

     如 The Clocks（IOI'94）中，可以将 3bit 作为一个钟的状态（**因为要给进位留出空间**）。这样变换 5 就可以表示为：#define move5((a + 0x209208) & 0x36DB6DB)
     化为二进制，即((a + 00000100000100100100000001000) & 01101101101101101101101101011)
     下文只介绍元素长度为 1 的集合的用法。
   - 以下有$0\leqslant k<n$。

2. 读取第k位（**元素k是否存在**）：　　#define isexist(A,k)　　A>>k&1

3. 将第k位设置为 1（**把k放入集合中**）：　　#define add(A,k)　　A|=1<<k

4. 将第k位设置为 0（**把k从集合中清除**）：　　#define remove(A,k)　　A&=~(1<<k)

5. 将第k位取反：　　#define rev(A,k)　　A^=1<<k

6. 将第k_1～k_2位取反：　　#define rev2(A,k1,k2)　　A^=((1<<(k2-k1+1))-1)<<k2

7. 枚举所有组合：从 0 开始不断地加 1。每一个数都代表一个组合。

[①] 在进行位运算的子程序中，推荐全部使用无符号数据类型，因为可以避免符号位的影响。

(2) 集合的交、并、补、差

1. 交集：　　#define cross(A,B)　　A&B
2. 并集：　　#define union(A,B)　　A|B
3. 全集：　　#define fill(I,n)　　I=(1<<n)-1
4. 补集：　　#define opp(I,A)　　A^I
5. 如果 S' 是 S 的真子集，那么有 $S'<S$。
6. A 包含 B ⇔ $A\&B==B$

(3) 集合中的元素

1. 去掉最后一位（10011→1001）　　ans = x>>1
2. 在最后加一个 0（11101→111010）　　ans = x<<1
3. 在最后加一个 1（110→1101）　　ans = (x<<1)+1
4. 取末 k 位（如 k=3 时，11101→101）　　ans = x&((1<<k)-1)
5. 把末 k 位变成 1（如 k=2 时，1000→1011）　　ans = x|((1<<k)-1)
6. 末 k 位取反（如 k=3 时，11010→11101）　　ans = x^((1<<k)-1)
7. 把右边连续的 1 变成 0（如 1011→1000）　　ans = x&(x+1)
8. 把右边连续的 0 变成 1（如 1100→1111）　　ans = x|(x-1)
9. 把右起第一个 0 变成 1（如 1001→1011）　　ans = x|(x+1)
10. 把右起第一个 1 变成 0（如 1110→1100）　　ans = x-(x&(-x))
11. 取右边连续的 1（如 10111→111）　　ans = (x^(x+1))>>1
12. 去掉右起第一个 1 的左边（如 11010→10）　　ans = x&(-x)

(4) 统计*

1. 是否恰好只有一个 true　　b = !(x&(x-1))&x
2. 判断是否有两个相邻的 true　　b = x>>1&x
3. 是否有三个相邻的 true　　b = x>>1&x>>2&x
4. 统计 true 的个数的奇偶性

```
x^=x>>1; x^=x>>2; x^=x>>4; x^=x>>8; x^=x>>16;
// 运算结果的第i位表示在原始数据中从第i位到最高位true数目的奇偶性，
// 有了这个结果，我们就可以很方便地得到任意一段的奇偶性：
// 如果想要得到k1~k2位中true个数的奇偶性，直接计算(x>>k1^x>>(k2+1))&1即可。
```

5. 统计 true 的数目（基于二分思想）

```
int count(unsigned int x)
{
```

```
    x=(x&0x55555555)+(x>>1&0x55555555);
    x=(x&0x33333333)+(x>>2&0x33333333);
    x=(x&0x0F0F0F0F)+(x>>4&0x0F0F0F0F);
    x=(x&0x00FF00FF)+(x>>8&0x00FF00FF);
    x=(x&0x0000FFFF)+(x>>16&0x0000FFFF);
    return x;
}
```

6. 反转位的顺序

```
unsigned int rev(unsigned int x)
{
    x=(x&0x55555555)<<1|(x>>1&0x55555555);
    x=(x&0x33333333)<<2|(x>>2&0x33333333);
    x=(x&0x0F0F0F0F)<<4|(x>>4&0x0F0F0F0F);
    x=(x&0x00FF00FF)<<8|(x>>8&0x00FF00FF);
    x=(x&0x0000FFFF)<<16|(x>>16&0x0000FFFF);
    return x;
}
```

(4) 其他*

1. 计算绝对值：

```
inline int abs(int x)
{
    int y=x>>31;
    return (x+y)^y;
}
```

2. 求较大值：

```
inline int max(int x,int y)
{
    int m=(x-y)>>31;
    return y&m|x&~m;
}
```

3. x 与 a,b 两个变量中的一个相等，现在要切换到另一个：x^=a^b

4. 不使用额外空间交换两个变量（**注：两变量的值不能相等！**）：

```
void swap(int& x,int& y) {x^=y;y^=x;x^=y;};
```

5. 求平均值：int ave(int x,int y){return (x&y)+((x^y)>>1);};

A.8 抽样测试法*

(1) 等价表达式[①]

【问题简述】给出一个表达式，再给出一组表达式。请问哪些表达式和第一个表达式是等价的（**结果相等**）。

表达式符合以下条件：

1. 表达式只可能包含一个变量 a。

[①] 题目来源：NOIP2005 第四题

2．表达式中出现的数都是正整数，而且都小于10000。

3．表达式中可以包括四种运算+、-、*、^（**乘方**），以及小括号(、)。

4．幂指数只可能是1到10之间的正整数（**包括**1**和**10）。

5．表达式内部，头部或者尾部都可能有一些多余的空格。

【分析】

有关表达式计算的内容，可以参考137页的“11.9 表达式求值”，这里不再给出代码。

两个表达式如果等价，那么无论 a 为何值，两个表达式计算出的值都相等。这时，我们以不同的 a 值代入各式，可以快速排斥那些不同的表达式，留下的便是等价的了。

以下是几种有效的取值方法：

① 取随机函数生成的数列。这种方法比较有效，无规律。

② 取伪随机数列。这是一种比较便于人工控制的手段。

③ 取实数。由于其他皆为整数，小数部分便成为判断的优越条件。

一般情况下取4～7组值便可通过极大部分情况，实数需要更小。

(2) Miller-Rabin 素性测试*

在测试质数时，抽样法是一个非常有用的工具。下面给出一种质数判定方法：

对于待判定的整数 n。设 $n-1=d\times 2^s$（d**是奇数**）。对于给定的基底 a，若 $a^d\equiv 1 \pmod n$，或存在 $0\leqslant r<s$ 使 $a^{d\times 2^r}\equiv -1 \pmod n$，则称 n 为以 a 为底的强伪质数。利用二分法，可以在O(log$n$)的时间内判定 n 是否为以 a 为底的强伪质数。

对于合数 c，以小于 c 的数为底，c 至多有1/4的机会为强伪质数。

如果不是随机抽样，而是抽样特殊情况——最小的几个质数，则：

如果只用2一个数进行测试，最小的强伪质数（**反例**）是2047，所以一个数显然不够；

如果用2和3两个数进行测试，最小的强伪质数为1373653，大于 10^6；

如果用2,3,5进行测试，最小的强伪质数为25326001，大于 2×10^7；

如果用2,3,5,7进行测试，最小的强伪质数为3215031751，大于 3×10^9，已经比32位带符号整数的最大值还大了。

可见，通常只要抽取2,3,5,7这几个固定的数进行测试就能保证测试的正确性了。

A.9 离散化*

离散化是指把无限空间中有限的个体映射到有限的空间中去，以此提高算法的时空效率。

离散化是程序设计中一个非常常用的技巧，它可以有效的降低时间复杂度。其基本思想就是在众多可能的情况中“只考虑我需要用的值”。离散化可以改进一个低效的算法，甚至实现根本不可能实现的算法。要掌握这个思想，必须从大量的题目中理解此方法的特点。

离散化的应用非常广泛，下面举两个例子说明一下。

【问题描述】[①]给定平面上 n 个点的坐标，求能够覆盖所有这些点的最小矩形面积。矩形可以倾斜放置（**边不**

[①] 题目来源：UVA10173。内容转自Martix67：http://www.matrix67.com/blog/archives/108。

必平行于坐标轴）。

【分析】

这里的倾斜放置很不好处理，因为我们不知道这个矩形最终会倾斜多少度。假设我们知道这个矩形的倾角是 α。因为它是一个实数，有无数种可能，我们不可能枚举每一种情况，所以我们需要一种方法，把这个“连续的”变量变成一个一个的值，变成一个“离散的”变量。这个过程也就是离散化。

我们可以证明，最小面积的矩形不但要求四条边上都有一个点，而且还要求至少一条边上有两个或两个以上的点。试想，如果每条边上都只有一个点，则我们总可以把这个矩形旋转一点使得这个矩形变“松”，从而有余地得到更小的矩形。于是我们发现，矩形的某条边的斜率必然与某两点的连线相同。如果我们计算出了所有过两点的直线的倾角，那么 α 的取值只有可能是这些倾角或它减去 90 度后的角（**直线按“\”方向倾斜时**）这么 C_2^n 种。我们说，这个“倾角”已经被我们“离散化”了。

对于某些坐标虽然已经是整数（**已经是离散的了**）但范围极大的问题，我们也可以用离散化的思想缩小这个规模。

【问题描述】[①]给定平面上的 n 个矩形（**坐标为整数，矩形与矩形之间可能有重叠的部分**），求其覆盖的总面积。

（n≤100，**坐标范围位于** $[-10^8, 10^8]$，**且都是整数**）

【分析】

朴素的想法就是开一个与二维坐标规模相当的二维 bool 数组模拟矩形的“覆盖”（**把矩形所在的位置填上** true）。可惜这个想法在这里有些问题，因为这个题目中坐标范围相当大。

但我们发现，矩形的数量 n≤100 远远小于坐标范围。每个矩形会在横纵坐标上各“使用”两个值，100 个矩形的坐标也不过用了 -10^8 到 10^8 之间的 200 个值。也就是说，实际有用的值其实只有这么几个。这些值将作为新的坐标值重新划分整个平面，省去中间的若干坐标值没有影响。

我们可以将坐标范围“离散化”到 1 到 200 之间的数，于是一个 200×200 的二维数组就足够了。实现方法正如本文开头所说的“排序后处理”。对横坐标（**或纵坐标**）进行一次排序并映射为 1 到 $2n$ 的整数，同时记录新坐标的每两个相邻坐标之间在离散化前实际的距离是多少。

A.10 Flood Fill*

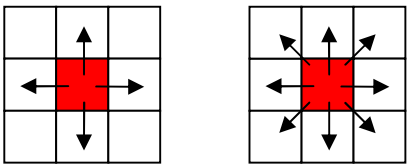
Flood Fill 译名很多，如“水流式填充法”、“种子染色法”。这种方法在计算机图形学领域有广泛的应用。在信息学竞赛中，这种方法多用于数据的预处理，如判断图的连通性。

Flood Fill 可以用深度优先搜索，广度优先搜索或广度优先扫描来实现。深度优先搜索和广度优先搜索都比较容易实现，这里不再给出代码。广度优先扫描不常见，具体内容可见 USACO 教程。

实现时先寻找一个未被标记的结点，对它做标记（**标号**），然后扩展。将所有由它扩展出的结点标上与它相同的标号，然后再找另一个未被标号的结点重复该过程。这样，标号相同的结点就属于同一个连通子图。

[①] 题目来源：VOJ1056

每次扩展的结点数因问题而异。常见的是“四连”和“八连”：



附录 B 调试

B.1 常见错误类型

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**：作为 OIer，应该熟练运用 C++，不应该犯语法错误。也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[①]，把“==”错写成“=”……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. **浮点数直接判断相等**：在 NOIP 中不太常见了。
8. **格式错误、文件名错误、文件放错位置……**：这些错误的原因都是没有审好题，没有认真查看主办方的规定。所以，大家千万不要犯此类错误！
9. **被黑**：这不是你的错误。不过如果你怀疑你被黑，你应该去申诉，而不是发牢骚，甚至去做不该做的事情。

B.2 调试过程

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！如果忽略，很容易因为长时间找不到错误而造成紧张、焦虑的情绪，从而影响答题。
2. **黑箱测试**：测试示例。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**）。

   如果代码是从其他地方粘贴过来的，就更应该仔细检查一下。

B.3 调试功能

1. 一般的 IDE 都会有以下几种调试功能，大家应该牢记它们的快捷键并熟练运用（**虽然很不好用**）：

   - 断点（Ctrl+F5，**本节中提到的快捷键都是 DEV-C++中的**）
   - 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）

[①] 由于“i”和“j”长得很像，所以写错的几率非常高，特别是在 for 中。写代码时注意！

- 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**。
     输出也有技巧——程序的输出要“好看”一些，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。
   - 判断某代码段是否运行：写一句能让程序崩溃的代码（如 a[-1000000]=0;），观察程序是否崩溃。
   - 过分依赖工具是没有好处的——如果你还在参加信息学竞赛。
3. 计时
   - 显示系统时间：system("time<nul");
   - 计时：double start = clock() / (double)CLOCKS_PER_SEC;
     需要数秒时，double end = clock() / (double)CLOCKS_PER_SEC;
     计算 end-start 即可。
   - 卡点：设一个变量 cnt，表示程序进行运算的次数。把它放到循环或递归中，判断，如果超过某一个数（不到 5,000,000），就说明“超时”，应该立刻结束程序。

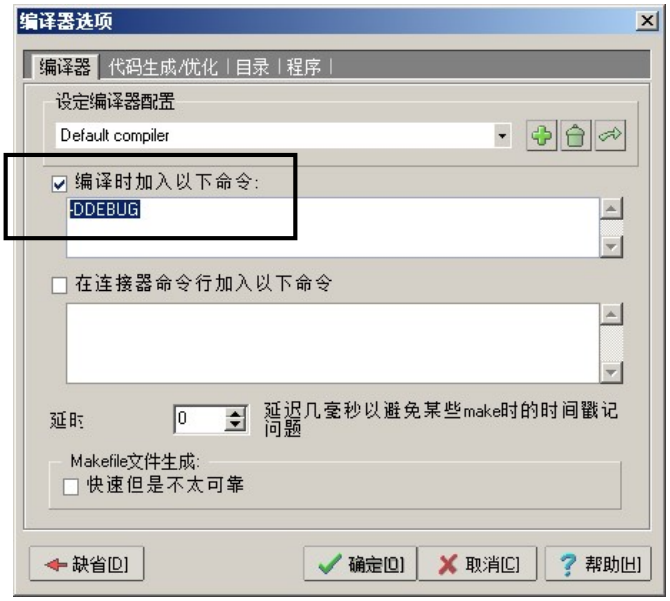
B.4 符号 DEBUG 的应用

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



这样，在竞赛的时候就不会因为忘记删除调试代码而丢分了。

B.5 代码审查表

问题种类	审查项	重要性
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？	
	“{”、“}”是否各占一行，并且对齐？	
	一行代码是否只做一件事？	
	++、--、+=、-=……是否单独存在？	重要
	if、for、while 自己占一行，接下来不管多少语句都要用花括号包好。	
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。	重要
	在进行比较、计算、输出之前，你的变量是否已经初始化？	重要
	max、min 的初始值是否正确？（max **应该是** 0，min **应该是一个大数**）	重要
	变量的精度够吗？	重要
表达式	你是否在复杂表达式中加了括号？	重要
	你是否写了数学表达式（**如** 1<a<=n）？赶快改过来吧。	重要
	逻辑判断是否存在“安全隐患”（**如浮点数直接判断相等**）？	重要
	运算过程中，会发生溢出吗？	重要
函数	你是否知道，函数名应该以动词开头，而变量名应该是名词？	重要
	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？	重要
	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？	重要
数组	你知道数组编号的下限吗？是 0，还是 1？	重要
	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？	重要
指针	定义指针之后，你是否马上把初始化为 NULL？	重要
	你是否还在用动态内存分配？	
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。	重要
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？	重要
	不等号中是否包含相等？你有没有把“<”写成“<=”，或写成别的？	重要
	你是否把“==”写成“=”？	重要
	你是否在 for 循环内部修改了循环变量？（**不要改**）	重要
输入/输出	文件名正确吗？拼写正确吗？大小写正确吗？ 有没有把“.”写成“,”，或把“destroy”写成“destory”？	重要
	符合题目规定的格式吗？	重要
	数据规模超过几千时，你还在使用流来输入输出数据吗？	
	用流来输入/输出 long long 类型的数据！	

问题种类	审查项	重要性
其他	memset 的第二个参数是多少？是 0 或-1 吗？（**不可以是其他值**）	重要
	你是否把 STL 中的容器换成了自己的代码？	

B.6 故障检查表

问题种类	审查项	重要性
程序崩溃	你在运行程序之前是否放了一个 in 文件？你打的文件名正确吗？	
	数组下标是否出现了负数，或者超过了上限？	重要
	是不是递归的深度太大？	重要
	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？	重要
	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？	重要
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？	重要
	变量名打错了吗？	重要
	是否在运算过程中发生了溢出？	重要
	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'？	重要
	是否把“==”错打成“=”，或者在不等号中错误地包含等号？	重要
	是否尝试指针或动态内存分配，结果出现了问题？	

B.7 命令行和批处理*

NOI 及更高级别的比赛都使用 Linux 系统（Ubuntu）。再用 DEV-C++就落伍了……

(1) 常用命令

目的	Windows	Linux
列举文件夹内的文件	dir	ls
改变/建立目录	cd/md 目录名或路径	cd/mkdir 目录名或路径
删除目录（递归删除）	rd /s 目录	rm -r 目录
比较文件内容	fc 文件 1 文件 2	diff 文件 1 文件 2
删除文件	del 文件	rm 文件
文件改名	ren 文件名 修改后的文件名	mv 文件名 修改后的文件名 （mv **实际上是移动文件**）
回显	echo 文本	echo 文本
复制文件（1→2）	copy 文件 1 文件 2	cp 文件 1 文件 2

复制目录	copy 目录1 目录2	cp -r 目录1 目录2
退出	exit	exit
查看进程	tasklist	ps（用“ps ax”可列出更多进程）
关闭 PID 为 1234 的进程	taskkill /pid 1234	kill 1234
关闭 Lazarus IDE	taskkill /im lazarus.exe	killall lazarus
用自己的数据测试程序，并准备对拍 （程序名为 test）	进入到程序所在文件夹，输入： test.exe <test.in >test.out （“>”表示覆盖，“>>”表示追加。）	进入到程序所在的文件夹，输入： ./test <test.in >test.out （“>”表示覆盖，“>>”表示追加。）
查看文件 test.out 的内容	type test.out 分页显示：type test.out | more	cat test.out 分页显示：cat test.out | more
对程序 test 计时	没有此功能	time ./test

(2) 简易对拍器（Windows）

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
```

(3) 简易对拍器（Linux）

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
```

(4) 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[①]，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>）。

如果调试 test.exe（**别忘记编译时加入参数-g**），输入“gdb test.exe”进入调试器。以下是 gdb 的常用命令：

[①] Windows XP 下的设置方法：右击“我的电脑”，选择“属性”，进入“高级”选项卡。单击“环境变量”按钮，在弹出的对话框中找到 PATH，点击“编辑”，在后面加入 g++所在路径，如“C:\Program Files\Dev-Cpp\bin”。

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。 l main：显示 main()附近 10 行。 如果不带参数，将继续显示上次代码的后 10 行。
b cl	break clear	设置断点。 取消断点。	b 24 或 cl 24：在第 24 行设置/取消断点。 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。 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。 disp x+1：自动输出表达式“x+1”的值。 dis disp 或 en disp：禁用/启用所有 Watch。
n s	next 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。 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 call print()：调用 print()
i	info	显示信息。	i b：显示所有断点 i lo（info locals）：显示所有局部变量。 i disp：显示所有“Watch”。
d	delete	删除。	d disp：删除所有“Watch”。 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()。

附录 C 竞赛经验和教训

C.1 赛前两星期

1. 你是否已经掌握了 NOIP 范围内的算法？

2. 你用过竞赛规定的 IDE 吗(**大多数省份是** DEV-C++)？你能不能用它调试程序，改正自己程序中的错误？

   （**如果不能，你会用其他方法调试程序吗，比如屏幕输出计算结果？**）

3. 递归、递推、枚举、回溯、搜索、贪心、分治、动态规划……你是否都有解题思路？
4. 栈、队列、二叉树、二叉排序树、堆、图……你是否能熟练地写出代码？
5. 快速排序、归并排序、二分查找、求最大公约数、求组合数……你会背吗？
6. 你能否很快地想起高精度算法？
7. 你还在做难题吗？把难题扔到九霄云外去吧。
8. 你是否做了一组套题？你能合理地安排时间吗？
9. 你有清晰的解题策略吗？（换句话说，你会答高考卷吗？不会是从第一道开始认真做，按顺序做到最后一题吧……）
10. 你的心态如何？你敢不敢对自己说：“我准备好了！”？
    引用一句大牛的话：“请记住，NOIP 不怕暴力，怕瞎算；不怕不会，怕不敢。”

C.2 赛前 30 分钟[①]

赛前 30 分钟，你应该做好一切准备工作。

1. 将显示分辨率调成合适的大小。如果还在用大脑袋的电视机，记得把刷新率调到最大。
2. 调整编辑器的字体、字号和颜色，字号要稍稍大一些。
3. 构建一个程序模板：

```
#include <iostream>
#include <fstream>
#include <cstring>
#include <algorithm>
#include <cstdlib>
using namespace std;

#define DEBUG

int n,m;
int a[100000];

int main()
{
    ios::sync_with_stdio(false);
    freopen(".in","r",stdin);
    freopen(".out","w",stdout);

    cin>>n;
    for (int i=0; i<n; i++) cin>>a[i];
```

[①] 不同省份，考前的情况也是不同的。有的省份可能不允许在竞赛开始之前碰电脑。

```
#ifdef DEBUG

#endif

    return 0;
}
```

4．按照要求建立自己的文件夹。

5．如果还有时间，再建立几个模板，以节省时间：

- 回溯法/DFS 模板
- 随机数据生成器
- 高精度算法

6．等待，等待，淡定地等待……祝每一个阅读到这里的人能在竞赛的这一刻发挥超常！

C.3 解题表

多做题，你就不会觉得下面的表很长了。

阶段	问题种类	审查项	重要性
审题	试题第一页	程序运行时间确实是 1s 吗？	
		内存限制是否为 128MB？（**如果是，你就可以随便浪费内存了**）	
		全文比较时是否过滤了行末空格和行尾回车？	重要
	读题	你的源代码文件名是否正确？输入输出文件名是否正确？拼写和大小写都正确吗？（**一个建议：直接到题目中把文件名复制下来，粘贴到你的程序中**）	重要
		10^8 是几位数？9E10 是多少，是几位数？[①] （10^8 **是 1 后面跟 8 个 0，是 9 位数**；9E10 **=** 9×10^{10} **而非** 9^{10}，**是 11 位数**）	
		未知量是什么？已知数据是什么？	
		你是否注意到了题目中的所有细节？你能将它们分解成不同要点吗？	重要
		给你一串数，第一个数据的编号是 0，还是 1？	重要
		m、n、s、t、x、y 有特别的用途吗？ （**题目中的字母做了什么事情，代码里对应的变量也要做相同的事情**）	
		如果题目是一个矩阵，读取顺序是先行后列，还是先列后行？ 对于正方形的示例数据，你是否更加认真地思考这个问题？	重要
思考	数据规模	数据范围如何？	重要
		int 够用吗？long long 够用吗？用不用高精度算法？	重要
		数组开得下吗？	
		计算时是否容易发生溢出现象？	
		你是否已经根据数据规模确定了复杂度和可能的算法？	重要
	数据结构	需要数组吗？一维数组还是二维数组？	
		数组大小是否已经比最大数据规模还大一些？	重要
		问题是否符合先进先出或先进后出的性质？需要队列或栈吗？	
		问题是否递归定义？需要树吗？	

[①] 问这个问题，是要提醒一件事：捐 3000 元钱，不要按 4 个 0——把位数搞错了。

阶段	问题种类	审查项	重要性
		需要图吗？问题适合用邻接矩阵、邻接表，还是边目录？	
	算法和思路	你是否知道你的第一感觉往往是错误的？	重要
		想想所有可能的算法——然后选有效的中最笨的！ 不要用你写不出代码的算法！	重要
		你的算法适合数据规模吗？	重要
		你能否找出一个反例来打破你的算法？	重要
		如果你不能解决所提出的问题，可先解决一个与此有关的问题。 你能不能想出一个更容易着手的有关问题？ 你能否解决规模更小的问题？	
		你以前见过类似的问题吗？你能否利用它？	
开始编程	写代码	你敢说你的算法能产生正确答案吗？算法确实正确吗？	重要
		你是否利用了你事先写好的框架？	
		你是否通过草纸和代码注释记录了你的思路？	重要
		逐步完善（**迭代式开发**）	
		把算法和实现中的已知错误写到明显的位置！	重要
		写出代码，一段一调试。	
	调试	你知道吗？即使是编程高手写程序，他也不敢保证自己的代码没有错误！	重要
		你是急着编译，还是慢慢地坐在那里，检查代码是否打错？ （**运行之前要静态查错！**）	重要
		知道什么叫排查隐患吗？为什么有白箱测试？ （**调试的时候不要着急，一个模块一个模块地检查**）	
		输出语句用完之后，你是直接删除，还是把它注释掉？	
		你能否熟练利用 IDE 的调试功能？会用 gdb 吗？	
	优化	你是否对你的代码做了备份？	重要
		根据需要不断优化。	
		你是否用大数据验证了你的优化效果？	重要
		你做的优化效果明显吗？是不是在猪身上抹香水？	
		见好就收，否则你会给自己带来麻烦。	
完成一道题	检查	你确信你的程序能够正确地从文件中读入数据，并输出到指定文件中吗？	重要
		你确定你的程序没有在屏幕上输出文字吗？	重要
		你的程序会不会暂停，等待你的输入？	重要
		程序结束之前，文件是否被程序关闭？	
		你还需要改代码吗？不需要的话，就把调试代码全部删干净，并把源代码放到指定的文件夹中。	重要
降低损失	找错误	你的输出应该是什么样的？	
		你的算法正确吗？你使用的数据结构是否需要变化？	
		你已经用了多长时间来查错？ 花 20 分钟查错是可以的，但是花 60 分钟查错就不如做另外一道题。	
		你找到线索了吗？	
	放弃一道题	当你已经花了很多时间去优化一道题，什么时候应该去看下一道题？	
		想一想，是继续把问题抠出来，还是争取多拿分数？	重要

阶段	问题种类	审查项	重要性
		（**选择解决更小数据规模的问题，使用简单但是能拿部分分数的算法**）	
技巧	变量	不要把变量名打错。	
		不要随便地复制和粘贴。完成后马上检查所有的字母。	
		起一个有意义的变量名。	
		定义变量和指针后立刻初始化！	重要
		不要使用有冲突的名称，即使是大小写不同也不行。	
		指出数组的下限是 0，还是 1？	
	注释	写简单、精炼的注释。	
		解释难懂的代码。	
		用注释把不同模块分割开。	
	其他	知道吗？合理的空白和缩进对你是有很大益处的！	
		如果能节省时间的话，就多浪费一些内存吧。	
		如果可能，就用最暴力的方法解决问题。	
		KISS，Keep it simple & stupid!	
		保留下所有的代码版本！	
		可以的话，尽量避免指针。	
		把动态内存分配当做自杀的手段——需要的话，使用“动态化静态”。	重要
		你在展示你的编程技巧吗？记住，NOIP 不是美术比赛，也不是设计竞赛！	
		判断相等时把常量放在前面，可防止把“==”打成“=”，如 NULL==p。	

其他内容：

1．数据范围与时间复杂度：从某种意义上说，问题规模也暗示了你可能的算法。

- 对于 $n \geqslant 1,000,000$ 的情况，要思考 $O(n)$ 的算法。如贪心。
- 对于 $10,000 \leqslant n \leqslant 1,000,000$，可以是二分的题，也可以是数据结构题，如并查集。
  用二分算法解决问题之前一定要思考下，函数是否是单调，如果不单调就无法二分。
- $500 \leqslant n \leqslant 1,000$，这种问题的算法必须控制在 $O(n2)$。
- $50 \leqslant n \leqslant 100$，这种问题只有可能两种情况，一种就是简单题，另一种就是搜索。
  如果是后者，一定要注意减枝，因为很有可能会超时。
- $20 \leqslant n \leqslant 50$， 这类问题基本就是简单题或者数学题。
- $n \leqslant 20$，这种题的复杂度往往是 $O(2n)$ 或者 $O(n!)$。首先考虑使用 DP 的方法，其次才是搜索和枚举，如果可以打表的话尽量打表。

2．理解题意的时候千万不要想当然。只去做题目说的东西，不要假设任何题目没有提及到的条件。

- 如《能量项链》（NOIP2006,1），示例数据中是 4 个珠子沿某个特定方向连续聚合，而实际上还有其他情况。
- 另外一个实例：《字符串的展开》（NOIP2007,2），输入数据长度在 100 以内，但是输出数据最长可达 6000。

3．非明文禁止者，皆不无可能。

4．“分段讨论”：在没有把握时，针对不同的数据规模写不同的程序，以提高得分率。例如：

```
if (k==0)
    直接模拟（第一种数据）
else if (k==1)
    枚举（第二种数据）
else
    ……              // 其他算法，当然不一定正确
```

5. 有序化：没思路时，先对数据进行有序化处理。

C.4 测试数据

1. 构造测试数据：一道合格的试题，应该有以下三种测试数据。
   - **小数据**：可以人工构造，也可以枚举生成。测试小数据目的是检验算法的正确性。
   - **大数据**：一般用数据生成器产生。测试大数据目的是检验算法的效率。
   - **极端数据**：极端数据非常关键。你需要检查你的程序在极端情况下程序会不会爆，会不会出错。
2. 极端数据种类
   - “擦边球”：恰好在边界附近，且欲越界的数据。
   - 边界大小：取最大或最小值，最多或最少值加减 1 的数据。
   - “0”：输入 0，或者在计算中出现 0。
   - “1”：只有 1 个数据，或者在循环、迭代中只有 1 个元素。
   - “1^{st}”：有序集合的第一个或最后一个数据元素。
   - 越界：使数组越界（**非常大或小于** 0）的数据。
   - 数据量最大或最小的数据。
   - 计算量最大或最小的数据。
3. 把比较有代表性的数据留下来，以便于优化时的比对。
4. **记住，很多示例数据是骗人的**。不要信任示例数据。
5. 记住，测试数据只是用来发现错误，而不是用来改正错误。**依靠测试数据改正错误，越改越糊涂！**
6. 让我们通过《侦探推理》（NOIP2003,2），体会一下“非明文禁止者，皆无不可能”：

   本题要求程序识别五种证词，分别是“I am guilty.”、“I am not guilty.”、“XXX is guilty.”（XXX **是人名**）、“XXX is not guilty.”、“Today is XXX.”（**星期一～星期日**）。如果说出其他证词，程序一律忽略不计。
   - 测试点#1：有一位同志，名字叫做“GUILTY”。如果你事先把证词的大小写变化了，那么就完了。
   - 测试点#2：有一位诚实的同志，既承认自己有罪，又承认自己无罪。
   - 测试点#4：有七位同志，名字分别是“MONDAY”、……、“SUNDAY”。
   - 测试点#7：“I am not guity.”！按照约定，这句话作废（guilty **缺一个字母**）。
   - 测试点#8：“SUE is guity.”！同样要作废。
   - 测试点#9：“I is not guilty.”，I 是人名。
   - 测试点#10：有一个人的名字长达 204 个字符，翻译成汉语，竟然是一段话：曾经有一段真挚的感情。芳，在我面前，我没有珍惜。等到失去了以后，才追悔莫及。人世间最痛苦的事莫过于此。如果上天能给我一个再来一次的机会，我会对那个女孩子说三个字……

     （**接下来这个人说了三句话**：“I love you!”、“If there must be a deadline,”、“I hope it is 10000 years!!!”。**按照约定，这三句话都要作废。所以，实际上他一句话也没有说。**）

C.5 交卷前 5 分钟

1. 停止编程。
2. **删除一切调试代码**！确认：
   - 屏幕上没有输出任何信息。
   - 程序结束后没有等待用户输入。

- main()中有 return 0;
- 没有调用 system 函数。
- 输入、输出文件名是正确的。
- 输入、输出格式是正确的。

- 使用 FILE，在程序退出之前已经把文件关闭（**使用流或重定向时，文件会自动关闭**）。

3. **把需要提交的程序复制到规定的文件夹，并改成规定的文件名**。

C.6 避免偶然错误

1. 思路
   - 不要轻易相信自己的第一感觉。在信息学竞赛中，第一感觉往往是错误的。
   - 动手之前先确认算法是正确的。编得差不多，然后发现算法错了，这就悲剧了。

   - 把思路和接口、伪代码等东西画在草纸上，写在注释里。（**否则一旦思路挂掉，就再也想不起来了。**）

   - 竞赛结束之前半个小时不要再想新算法了。这时有人交卷，如果你心理素质不好，会比较慌张的。这时你应该好好地调试程序，或者确认万事大吉后交卷。
2. 文件操作

   - 定时存盘！（**防的不是停电，而是 IDE 崩溃。**）

   - 在对代码进行重大修改之前，一定要对代码做备份！

     一方面可以防止把程序改“坏”，另一方面可以用来检查数据正确性（**如果代码绝对正确**）。

   - **务必检查文件名拼写、大小写是否正确，并且不要使用相对路径或绝对路径！**
     最好直接从题目文件中复制文件名，并使用题目附带的数据文件测试。
   - freopen("prob.cpp","w",stdout);——别做这样的傻事。
3. 编码
   - 建议使用“复制”和“粘贴”来输入文件名。

     （**曾经有一位同志把文件名中的“.”打成“,”，在竞赛结束后还未发现错误！**）

   - 为了避免结构上的混乱，最好控制好代码缩进，并在 if、while、for……后面使用花括号。

   - 在必要的地方写注释，解释代码的功能。（**便于调试和查错**）

   - 少用复制粘贴（**我指的是代码，不是上面说的文件名**）。复制粘贴之后要马上检查变量的正确性。

     特别是 for 要更加仔细检查。
   - 不要随意删除代码——用注释代替删除。
   - 把所有头文件都加上——DEV-C++很欠，它会自动而且偷偷地加头文件。如果你没有引用<cstring>而使用 memset，在 DEV-C++中能编译，但是在测评时直接 CE——0 分！
4. 做题

   - 正确地选择题目去做（**最擅长、最简单的先完成**）。

   - 合理地安排时间和解题顺序。
   - 复赛中一定要提高正确率，同时还要兼顾解题速度。
   - “一气呵成”，拿到题目就开始编码，这样是要吃大亏的。
   - 保险总比冒险好。

   - 被题卡住太长时间（**没思路、查不到错误、调试出错**……），就应该放弃这道题，换下一道题。

5. RP（人品）

- 如果意识到 RP 对 NOIP 的影响，就不应该随意虐人和泡 MM，以免损失 RP。
- 不要让旁边的人感觉你的速度很快——如果速度很快，那么程序的结果很可能是错误的。
- 考前要攒 RP，以免在赛场上看错题、写错算法、调试受挫、忘删调试代码，或电脑意外停电！
- 根据《RP 导论》的理论，送大家一个考前快速攒 RP 的方法：

  ① DotA、星际、红警——开对战模式，电脑数量最大，智力最高。几局之后，RP++++，游戏技术随之提高，可谓一举两得！如果有变速软件，就把速度调到最大！

  ② CS、HL——拿好你的小刀（**撬棍**），不要 CAMP，不要抢对手的武器。是练习近身战的时候了……

  记住，不许作弊，不许用修改器，不许改对手武器，否则会发生“while (rp) rp--;”！

  ③ 模拟器——不要限速，把 FPS 调到最大！这个时候的超级马利就很好玩了……

  ④ 三国杀——快，快，一定要快！别装装备，别使技能……

C.7 骗分

在走投无路之前，不要考虑骗分。

1. **非最优算法：**
   - 贪心法
   - 枚举法
   - 暴力搜索：在关键时刻能够救命的算法。
   - 随机化＋搜索
   - “分段讨论”

2. **“交表”**：对于某些问题，如果答案是固定的（**如** NOIP2008 **的“火柴棍游戏”**），不会解，可以事先用某种方法算好，然后直接输出答案。

3. **“样例”**：针对样例来写一个算法，或者干脆直接输出样例（**如** NOIP2006 **的“能量项链”**）。

   需要注意的是，现在这种骗分方法不太容易得分了。

4. **“无解”**：题目中说“如果无解，请输出-1”。你直接输出-1，肯定有 10~20 分。
   题目中说“如果无解，请输出 Failed”，你就直接输出 Failed……
   如果题目中什么都没说，那么你可以尝试一个可能的结果。至于能不能得分，就看你的 RP 了。

附录 D 学习建议

如果你不喜欢信息学，或者抱着功利目的去学习信息学，那么请你赶快抛弃它。现在（**指 2014 年及以后参加高考的同学**）竞赛获奖不再享有加分[①]和保送的“好处”。

如果你喜欢信息学，一定要好好地学习它。你所学习的知识、你所锻炼的能力对你的未来是有很大益处的。

D.1 学习方法

1. 做一些竞赛真题和网上的模拟题，熟悉比赛的题型和要求，找出自己的不足，加强训练。
2. 拿出一定时间来看书。不要光看书不做题，也不要光做题不看书。
3. 形成适合自己且合理的编码习惯。不要让自己因为混乱的代码而挂掉。
4. 做题时不能只写解题思路，应该把代码写出来，自己构造数据并调试。
5. 不要急着提交程序或看题解。先写出你自己的程序，用你自己构造的数据检查，用你自己的方法调试，把错误都改完之后再继续。
6. 合理安排时间——花一天时间抠一道题，还不如做点其他事情。
7. 适当地休息。写代码，特别是调试程序会耗费大量精力。
8. 多和他人交流。同学之间应该互相帮助，每个同学都不应该有“教会徒弟，饿死师父”的错误观念。
9. 当算法正确、程序错误时，暂时放弃，过几天再重打一遍。
10. 适当做一些在 NOIP 范围内的难题，但是不要太难。
    不要以为基础题都是简单题，也不要以为普及组的题都是简单题。
11. 一天不要做太多题。做对并且获得有益的启示才是最重要的。
12. 因为 NOIP 既不加分也不保送，所以你要处理好 OI 与高考的关系。

D.2 学习能力

1. 具备自学能力。
2. 熟练地运用 C++语言编程，并熟练掌握文件的输入/输出操作。
3. 增强自己编写代码和调试的熟练程度。
4. 适度提高做题的速度（**当然，不要太快**）——不要因为打字速度和熟练程度而耽误重要的事情。
5. 提高自己设计测试数据的能力。
6. 提高自己做题的正确率（**还有得分率**）。
7. 熟练运用基本算法（**递推、递归、动态规划、贪心、搜索**）。
8. 熟练运用基本数据结构（**字符串、二叉树、图等**）以及相关算法（**最短路径、最小生成树等**）。
9. 善于表达自己的观点，善于与他人交流。
10. 提高自己的抗干扰能力。不要因为显示器、键盘、鼠标或 IDE 等次要因素而挂掉。

D.3 关于清北学堂

清北学堂是一个在北京开设的奥赛和自主招生集训班。你可以考虑参加清北学堂的信息学集训班[②]。

清北学堂的网站是 www.qbxt.cn 和 www.topschool.org。在寒暑假、五一之前的一至两个月会下发

[①] 这里指辽宁省不再给竞赛一等奖加分。其他部分省市可能仍有奥赛加分政策。

[②] 我（**主编**）不是在做广告。设这一小节的原因是我们学校没有竞赛班，甚至没有懂竞赛的老师，但仍有人搞 NOIP。

集训通知。这时你需要及时和负责教师（**如果有**）取得联系，当然，你也可以自己报名。

集训班的学费在 1500~1900 左右，如果报名人数较多（**不一定是信息学**），价格会低一些。此外，车票和住宿等花费一般要超过 500 元。所以，家里条件不足的最好不要报名。

在进行集训之前，你需要有比较牢的基础，对常用算法要有一定的了解，否则你会吃大亏的(**听不懂**.....)。

附录 E 竞赛简介

E.1 从 NOIP 到 IOI

中学阶段有五大科竞赛：数学、物理、化学、生物、信息学。这五科竞赛的目的都是选拔人才、培养学生能力、普及学科知识。

信息学竞赛的路是很长的。如果你想在 IOI 上获得金牌，你需要经历以下的竞赛：

首先，你需要在全国青少年信息学奥林匹克联赛（National Olympiad in Informatics in Provinces，**简称** NOIP **或联赛**）上获得一等奖。注意，你需要先在初赛上胜出，得到参加复赛的资格后，再到复赛上拿一等奖。

NOIP 一等奖之后，你需要参加省队选拔赛，还要参加省里的培训活动。每个省有四五人可以参加全国青少年信息学奥林匹克竞赛（National Olympiad in Informatics，**简称** NOI **或信息学奥赛**）。

NOI 以省为单位来组织。你需要在 NOI 上获得一等奖，并且是前几名，才有机会进入国家队。

NOI 获胜后，你就叫“神犇”了。你要参加冬令营（WC），进行培训，还要参加国家队选拔赛（CTSC）。只有冬令营和选拔赛两者的成绩都合格，你才有资格代表国家参加国际信息学奥林匹克竞赛（International Olympiad in Informatics，**简称** IOI）。

征服了国内的对手之后，你就可以代表中国参加国际竞赛了。

如果你真的从 NOIP 走到了 IOI，那么恭喜你，全国的 OIer 们都会祝福你获得金牌。

E.2 NOIP 简介

(1) NOIP 时间和题型

NOIP 分普及组和提高组，前者是初中的，后者是高中的。二者题目不完全相同，提高组难度高于普及组。

1．初赛

初赛在 10 月份第三个星期六[①]的 14:30～16:30 进行，时间为 2 小时，内容全部为笔试，满分 100 分。

试题由四部分组成：

① 选择题（**共** 20 **题**，**每题** 1.5 **分**，**共计** 30 **分**）：提高组的前 10 道题为单选题，后 10 道题为不定项选择题（**只有全部选对才得分，否则不得分**）；普及组的前 20 道题都是单选题。

② 问题求解题（**共** 2 **题**，**每题** 5 **分**，**共计** 10 **分**）：试题给出一个叙述较为简单的问题，要求学生对问题进行分析，找到一个合适的算法，并推算出问题的解。考生给出的答案与标准答案相同，则得分；否则不得分。

③ 程序阅读理解题（**共** 4 **题**，**每题** 8 **分**，**共计** 32 **分**）：题目给出一段程序（**不一定有关于程序功能的说明**），考生通过阅读理解该段程序给出程序的输出。输出与标准答案一致，则得分；否则不得分。

④ 程序完善题（**共** 2 **题**，**共计** 28 **分**）：题目给出一段关于程序功能的文字说明，然后给出一段程序代码，在代码中略去了若干个语句或语句的一部分并在这些位置给出空格，要求考生根据程序的功能说明和代码的上下文，填出被略去的语句。填对则得分；否则不得分。

[①] 准确地说，是处在 10 月份中旬的星期六。复赛时间也是如此。

2．复赛

复赛在11月份第三个星期六和星期日的上午8:30～11:30进行。

复赛的题型和考试形式与NOI类似，全部为上机编程题，但难度比NOI低。

普及组的复赛时间为3小时，共4道题，每题100分，共计400分。

提高组的复赛包括一试和二试，分两天进行。每次测试时间为3小时，有3道题，每题100分。选手的总分为两次测试的分数的总和，最高600分。

2010年及以前提高组只有一试，共4道题，满分400分。

每一试题包括：题目、问题描述、输入输出要求、样例数据（**部分题目有样例的说明**）。测试时，测试程序为每道题提供了10－20组测试数据，考生程序每答对一组得5－10分，累计分即为该道题的得分。

NOIP的程序将在NOI Linux 1.3（Ubuntu系统）的环境下测评，G++编译器版本为4.4.5，评测系统为Arbiter。2011年测评器CPU为P4 3.0GHz，内存大小为1GB。

（2）初赛内容与要求

计基算本机常的识	1．计算机和信息社会（**信息社会的主要特征、计算机的主要特征、数字通信网络的主要特征、数字化**） 2．信息输入输出基本原理（**信息交换环境、文字图形多媒体信息的输入输出方式**） 3．信息的表示与处理（**信息编码、微处理部件**MPU**、内存储结构、指令，程序，和存储程序原理、程序的三种基本控制结构**） 4．信息的存储、组织与管理（**存储介质、存储器结构、文件管理、数据库管理**） 5．信息系统组成及互连网的基本知识（**计算机构成原理、槽和端口的部件间可扩展互连方式、层次式的互连结构、互联网络、**TCP/IP **协议、**HTTP **协议、**WEB **应用的主要方式和特点**） 6．人机交互界面的基本概念（**窗口系统、人和计算机交流信息的途径（文本及交互操作）**） 7．信息技术的新发展、新特点、新应用等。
计基算本机操的作	1.Windows 和 LINUX 的基本操作知识 2.互联网的基本使用常识（**网上浏览、搜索和查询等**） 3.常用的工具软件使用（**文字编辑、电子邮件收发等**）
程序设计的基本知识	**数据结构** 1．程序语言中基本数据类型（**字符、整数、长整、浮点**） 2．浮点运算中的精度和数值比较 3．一维数组（**串**）、线性表、队列与栈 4．记录类型（PASCAL）/结构类型（C/C++） **程序设计** 1．结构化程序设计的基本概念

	2．阅读理解程序的基本能力 3．具有将简单问题抽象成适合计算机解决的模型的基本能力 4．具有针对模型设计简单算法的基本能力 5．程序流程描述（**自然语言/伪码**/NS **图/其他**） 6．程序设计语言（PASCAL/C/C++） **基本算法处理** 1．初等算法（**计数、统计、数学运算等**） 2．排序算法（**冒泡法、插入排序、合并排序、快速排序**） 3．查找（**顺序查找、二分法**） 4．简单搜索 5．字符串处理 5．回溯算法 6．递归算法

（3）复赛内容与要求

在初赛内容的基础上增加以下内容（*表示普及组不涉及）：

计算机软件	1．操作系统的使用 2．编程语言的使用
数据结构	1．指针类型 2．多维数组 3．单链表及循环链表 4．二叉树 5．文件操作（**从文本文件中读入数据，并输出到文本文件中**） 6．图*
程序设计	1．算法的实现能力 2．程序调试基本能力 3．设计测试数据的基本能力 4．程序的时间复杂度和空间复杂度的估计*
算法处理	1．离散数学知识的应用（**如排列组合、简单图论、数理逻辑**） 2．分治思想 3．模拟法 4．贪心法 5．简单搜索算法（**深度优先、广度优先**）、搜索中的剪枝 6．动态规划的思想及基本算法

E.3 常用语

1. 竞赛简称
   - OI：信息学奥林匹克竞赛
   - NOIP：National Olympiad in Informatics in Provinces，联赛，分为普及组（**初中生参加**）和提高组（**高中生参加**）。

     NOIP 初赛相当于其他竞赛科目的省级联赛，NOIP 复赛相当于其他竞赛科目的全国初赛。
   - NOI：National Olympiad in Informatics，全国竞赛。先进省队然后才能参加。
   - CTSC：Chinese Team Selection Contest，中国国家队选拔赛，选拔选手参加 IOI。CTSC 的题难度最大，甚至比 IOI 大。
   - IOI：International Olympiad in Informatics，国际信息学奥林匹克竞赛
   - WC：Winter Camp，冬令营
   - APIO：Asia-Pacific Informatics Olympiad，亚洲与太平洋地区信息学奥林匹克（**不属于 NOI 系列活动，有 NOIP 一等奖即可参加**）
   - ACM/ICPC：ACM International Collegiate Programming Contest，ACM 主办的大学生程序设计竞赛（**不属于 NOI 系列活动，以团体的形式组队参加**）
2. 选手
   - OIer：学信息学奥赛的人。
   - P 党：指用 Pascal 语言编程的 OIer。
   - C 党：指用 C 或 C++语言编程的 OIer。
   - 弱菜：新手
   - 大牛、神犇（bēn）：高手
3. 题目
   - 水题：特别简单的题——有编程基础的就能做上——只要认真审题，基本上就能得满分。
   - 水过：做完了一道水题。
   - 秒过：所有数据 0ms，毫无压力。
   - 暴 0：一个题目一个测试点也没过，得 0 分。
   - 被虐：被题目虐杀，自己完全做不出，或者对自己的表现很不满意。
   - 暴虐：这个是主动的，把题目虐杀，秒掉。
4. 评测
   - AC：Accepted，测试点全部正确。
   - AK：把一次比赛的所有的题目全都做对，得满分。
   - WA：Wrong Answer，答案错误，也可能是输出格式错误。
   - CE：Compile Error，编译错误，出现了某些语法错误，错误引用头文件，或者忘记引用头文件，还可能是忘记 using namespace std 而直接使用 std 的成员。
   - RE：Runtime Error，运行错误。可能是算法中数组下标越界，可能是因文件错误而引发的下标越界，也可能是爆栈。
   - TLE：Time Limit Error，超时。可能是算法太慢、发生死循环。
   - MLE：Memory Limit Error，超过空间限制。在 NOIP 中一般不会出现此类问题。尽管如此，做题之前仍然需要仔细阅读试题第一页的“内存限制”。
5. 算法与数据结构：
   - DFS：Depth First Search，深度优先搜索，简称“深搜”，搜索过程叫“一条道走到黑”。一般用递归实现。

- BFS：Breadth First Search，广度优先搜索，简称“广搜”、“宽搜”，搜索过程和扩散现象比较像。实现时需要队列。
- DP：Dynamic Programming，动态规划，简称“动规”
- DAG：Directed Acyclic Graph，有向无环图
- SPFA：Shortest Path Faster Algorithm，使用队列的 Bellman-Ford 算法。这是一个比较常用的算法。
- BST：Binary Search Tree，二叉排序树
- MST：Minimum-cost Spanning Tree，最小生成树
- SCC：Strong Connected Component，强连通分量
- LIFO：Last In & First Out，后进先出表，栈属于这种结构。
- FIFO：First In & First Out，先进先出表，队列属于这种结构。
- STL：Standard Template Library，标准模板库，是 C++自带的“现成的程序”。STL 的头文件与其他头文件不同，是“开源”的。
- ADT：Abstract Data Type，抽象数据类型，在软件开发中很常见。

6. 1..10：Pascal 语言里的用法，翻译成英语就是“1 to 10”。
7. IDE：Integrated Development Environment，集成开发环境
8. 难题：
   - 非确定性计算机：一种神奇的计算机（目前还不存在），能够猜测答案，并且一猜一个准。用非确定性计算机解 N 皇后问题，只需 O(n)的时间，就可以“猜”出一个合法的放置方式。
   - NP（Nondeterminism Problem）：能在非确定性计算机上用多项式时间解决的问题。
   - NPH（NP-Hard）：即使在非确定性计算机上也无法用多项式时间解决的问题。
   - NPC（NP-Complete，NP 完全性）：目前无法证明一个 NPC 问题不可以用多项式时间算法解决。
   - 不可解问题：无法通过编程来解决的问题。最经典的不可解问题是“停机问题”（给出一段程序，判断它在运行之后是否会发生死循环）。
9. TSP，Traveling Salesman Problem，旅行商问题，俗称“货郎担问题”，它是一个 NP 问题。描述如下：已知 n 个点和任意两点间的距离，给出过这些定点的最短闭合回路。
   平面直角坐标系上的 TSP 问题叫欧几里得旅行商问题。

E.4 第一次参加复赛……

1. 我们参加的是算法竞赛。所以，**答案是检验程序正误的唯一标准**。至于怎么得出答案，随你便。
2. 不要输出题目规定以外的东西，也不要从屏幕上读取任何内容（**如**“Please input n:”、“The result is …”、“**欢迎使用**”、“**版权所有**”、“Press any key to continue…”），否则 0 分。
3. 要严格按照题目规定格式来输入、输出文件。格式错误和计算结果错误等效——0 分。
4. 输入和输出要在指定文件中进行。直接从屏幕上输入、输出，得 0 分。
5. 文件结束应该有换行符。
6. 用绝对路径读取和存储文件，会得 0 分的。
7. 程序运行时间是有限制的。如果你的程序没有在规定时间内退出，或者在等待用户输入——“Press any key to continue…”，会得 0 分的。
8. 程序必须正常退出。在 main()中忘记 return 0，或者返回其他数值，即使答案正确也是 0 分。
9. 没有把源代码放入指定的文件夹，或者没有起正确的文件名（**注意扩展名**），或者文件夹内有规定以外的文件，你都会得 0 分。
10. 使用自己的头文件，得 0 分。
11. 如果源文件大小超过 100KB，那么没有人知道接下来能发生什么事情。

12. 如果做一道题同时交了 3 个源文件（.c、.cpp、.pas），那么测评机将测“.c”的那个。

13. 随便关机或重启的结果是 0 分。

14. 如果你的电脑倒霉地停电了，别自己处理，找老师。但是不要再指望还找到你的代码了……所以要攒 RP！

15. 被问题和算法困住是正常的事情，而忘记某个关键字的拼写、忘记某个语句的用法是不应该发生的事情。

16. 在赛场上，如果监考人员发现你携带书、纸、U 盘、手机、电子词典等物品，不管你是否使用，你都会被取消参赛资格。

17. 在测评时，如果你的程序有以下行为，将被视为作弊：
    - 运行其它程序
    - 打开或创建题目规定的输入/输出文件之外的其它文件
    - 改变文件系统的访问权限
    - 读写文件系统的管理信息
    - 使用除读写规定的输入/输出文件之外的其它系统调用
    - 试图访问网络
    - 使用 system 或其它线程/进程生成函数
    - 捕获和处理鼠标和键盘的输入消息
    - 读写计算机的输入/输出端口

18. 不使用任何算法而直接输出答案是合法行为。例如在竞赛时写出了下面两个程序，结果第一个得了满分，第二个得了 20 分。

```
// NOIP2008,2 - 火柴棍等式
// 写枚举算法，发现容易超时（算法不对），所以就事先算好再输出。
#include <fstream>
int i, n[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,1,2,8,9,6,9,29,39,38,65,88,128};
int main()
{
    std::ifstream fin("matches.in");
    std::ofstream fout("matches.out");
    cin>>i;
    cout<<n[i];
    return 0;
}
```

```
// NOIP2011 day1, 3 - Mayan游戏
// 这道题不会做，直接按照题目要求做——“如果无解，请输出-1”。
#include <fstream>
int main()
{
    std::ofstream fout("mayan.out");
    cout<<"-1";
    return 0;
}
```

19. 只要你不是 0 分，你就有三等奖了。
    只要你能够熟练运用 C++语言，即使没有学过什么算法，拿二等奖也很轻松。
    只要你努力学习，总有机会拿一等奖。

附录 F NOIP 复赛知识点分布

（注意：内容仅供参考）

年份	题目中文名称	题目英文名称	知识点（方框表示其他可行方法）	难度
2004	津津的储蓄计划	save	模拟法	易
	合并果子	fruit	贪心算法、堆［局部排序］	中
	合唱队形	chorus	动态规划［二分法］	易
	虫食算	alpha	解线性方程组［搜索、剪枝］	难
2005	谁拿了最多奖学金	scholar	字符串操作、数据处理	易
	过河	river	动态规划（**编号类**）	中
	篝火晚会	fire	数学推理	中
	等价表达式	equal	抽样测试法、模拟法（**栈**）［分治］	难
2006	能量项链	energy	动态规划（**区间类**）	中
	金明的预算方案	budget	动态规划（**背包类**）	易
	作业调度方案	jsp	模拟法（**似乎是在考察语文水平**）	易
	2^k进制数	digital	组合数学、高精度算法	中
2007	统计数字	count	快速排序［BST、Hash］	易
	字符串的展开	expand	模拟法（**字符串**）	易
	矩阵取数游戏	game	动态规划（**区间类**）、高精度算法	中
	树网的核	core	图的 Floyd 算法及简单处理	难
2008	笨小猴	word	模拟法（**字符串**）	易
	火柴棒等式	matches	枚举法	易
	传纸条	message	动态规划（**坐标类**）	中
	双栈排序	twostack	栈、二分图搜索	难
2009	潜伏者	spy	模拟法（**字符串**）	易
	Hankson 的趣味题	son	数论（**分解质因数**）	中
	最优贸易	trade	图论、动态规划	中
	靶形数独	sudoku	搜索和剪枝	中
2010	机器翻译	translate	模拟法（**队列**）	易
	乌龟棋	tortoise	动态规划（**编号类**）	易
	关押罪犯	prison	二分法和二分图判断［贪心和并查集］	中
	引水入城	flow	搜索、动态规划［Flood Fill］	难

2011	铺地毯	carpet	模拟法	易
	选择客栈	hotel	枚举法及优化	中
	Mayan 游戏	mayan	搜索（**似乎是在考察选手的心理素质**）	中
	计算系数	factor	组合数学	易
	聪明的质监员	qc	二分查找、前序和	中
	观光公交	bus	网络流算法［贪心算法］	难

近年来的一等奖分数线（**辽宁**）：

2011：335（**满分** 600，**下面各年的满分为** 400）

2010：210
2009：170
2008：230

附录 G 资料推荐

G.1 书籍

1. “白皮”：刘汝佳，《算法竞赛入门经典》，清华大学出版社
2. “黄皮”：吴文虎、王建德，《全国信息学奥林匹克竞赛培训教程》系列，清华大学出版社
3. “黑皮”：刘汝佳、黄亮，《算法艺术与信息学竞赛》，清华大学出版社（**这本书有一定的难度**）
4. 《全国青少年信息学奥林匹克联赛》，南京大学出版社
5. 王建德，《金牌之路》，陕西师范大学出版社
6. 吕品，《信息学奥林匹克提高篇》，北京大学出版社
7. 《算法艺术与信息学竞赛——学习指导》，清华大学出版社
8. 严蔚敏，《数据结构》，清华大学出版社
9. 《数据结构与算法分析》，电子工业出版社
10. 《算法导论》（**英语**）
11. 《C++ Primer》《C++程序设计语言》：看一遍目录，你就知道 C++其实并不像竞赛中那么简单。

G.2 网站

1. OJ：Online Judge，在线的评测网站
   - Tyvj：www.tyvj.cn，该网站由于资金原因，不太稳定，但不失为一个优秀的 OJ。
   - Vijos：www.vijos.org，该网站由于几年之前关闭，又刚刚“复活”，可能不太稳定。
   - POJ：北京大学办的，poj.grids.cn
   - TOJ：天津大学，acm.tju.edu.cn/toj
   - ZOJ：浙江大学，acm.zju.edu.cn
   - Rqnoj：www.rqnoj.cn。和 Tyvj 差不多，但是插一点广告。
2. 论坛：
   - CoderSpace：www.coderspace.net
   - OIBH（**信息学初学者之家**）：www.kuye.cn。原有域名已经“挂”了。
   - 百度 NOIP 贴吧
3. USACO 训练网站（**英语**）：train.usaco.org

   USACO 的题库是美国信息学竞赛的官方题库，虽然和 NOIP 有些脱节，但不失为好的信息学教材。
   USACO 题目明确，尤其是测试数据极具代表性。
   USACO 训练网站中多数为搜索题。想练习搜索可以优先考虑这里的题。
4. NoCow：www.nocow.cn。类似于维基百科，重要的是有 USACO 的译题和题解。
5. 竞赛官网：
   - NOI 官网：www.noi.cn
   - NOI 官网（**辽宁赛区，大连海事大学**）：noi.dlmu.edu.cn
   - USACO 官网（**英语**）：www.usaco.com
6. USACO 月赛试题及数据：假如寻找 2010 年 11 月月赛试题和数据，只需进入 ace.delos.com/NOV10。其他月赛的进入方法和此类似。

参考文献

[1] 刘汝佳，《算法竞赛入门经典》，清华大学出版社
[2] Clifford A. Shaffer，《数据结构与算法分析（第二版）》，电子工业出版社
[3] 《信息学奥林匹克竞赛辅导》，网络资料
[4] USACO Training Gateway
[5] 《NOIP 实用算法》，网络资料
[6] 《背包九讲》，网络资料
[7] ACM/ICPC 代码库，吉林大学计算机科学与技术学院
[8] 江涛，《NOIP 常用搜索算法技巧》
[9] 俞鑫，《棋盘中的棋盘——浅谈棋盘的分割思想》
[10] 胡伟栋，《浅析非完美算法在信息学竞赛中的应用》
[11] 黄劲松，《贪婪的动态规划》
[12] 陈彧，《信息学竞赛中的思维方法》
[13] 李博杰，《骗分导论·信息学竞赛》
[14] 骆可强，《论程序底层优化的一些方法与技巧》
[15] 韩文弢，《论 C++语言在信息学竞赛中的应用》
[16] 刘汝佳、黄亮，《算法艺术与信息学竞赛》，清华大学出版社
[17] 《基数排序》，百度百科
[18] 《强连通分量》，百度百科
[19] 《离散化》，百度百科
[20] *我是智障*，《骗分导论》
[21] *Meta_No1*，《搜索题的调试技巧以及一些附加内容》
[22] 孙慨然的 NOIP2005 解题报告
[23] 《普通高中课程标准实验教科书·数学·必修三》，人民教育出版社
[24] 林锐，《高质量 C++/C 编程指南》

计算机专业是朝阳还是夕阳?

转自《中国青年报》(2011 年 04 月 11 日 12 版)

编辑老师:

现在 IT 产业发展迅速,需要大量的人才,人家说计算机(IT)专业是非常受欢迎的专业,可我也听说,许多学计算机的大学生毕业后找不到工作。我喜欢计算机,想报考这个专业,可一听说就业困难又犹豫了,您能告诉我怎么办吗?

一个考生

从 1977 年恢复高考到 2000 年前,与各专业相比,计算机专业的录取分可以说是最高之一。2000 年以后,计算机不再是热门专业毕业生,录取分也一降再降。特别是近几年,计算机专业的就业是各个专业中最差之一,有学校的计算机专业甚至被教育部亮了黄牌。

从最高到最差,计算机专业经历了从热到冷的过程,原因是什么?

高学费是计算机专业扩招的主要动力

首先是由于开设计算机专业的学校太多太滥,办学质量差,培养出的合格人才太少。扩招前,一个学校每年招收计算机专业的学生一般不到 100 人,开设该专业的学校也不多,办学认真,师资水平高,办学质量好。1998 年扩招政策使众多大学都开设了计算机专业。据不完全统计,现在开设计算机专业的高校有 700 多所。高校每年招生的人数都很多。如果按照一个学校每年平均招 300 人来计算,计算机专业一年的毕业生就超过了 20 万。但是,大多数毕业生的专业素质不够,以至于一方面计算机专业的毕业生就业难,另一方面 IT 企业又难以在毕业生中招到合适的人才。实际上,专业素质较好的毕业生的就业前景是很广阔的,一些培养质量较好的学校的毕业生供不应求。

其次,教育部对各高校办学的资质并没有认证程序,对行业的人才需求没有行业调查数据,自然很难提出宏观就业指导意见。高学费是各高校热衷招收计算机专业学生的主要动力。越差的学校录取的分数越低,但学费却越高,有的达到每年 1 万多元。那些高考分数低的学生为了上学,就多花钱上了这些学校。不少学校没有足够的师资力量开设计算机专业,对学生指导不够。4 年下来,学生投入至少四五万元,却学不到太多有用的东西,更不会动手,就业难是理所当然的结果了。

选择计算机专业的 3 个理由

计算机技术是人类在 20 世纪最重要的发明之一。目前,计算机技术仍处于技术发展的上升期和活跃期。最近一些影响广泛的新技术和新产品,如智能手机、3G 通信、云计算、微博、社交网等,无一不是和计算机技术密切相关的。这些计算机的专业领域以及相关的应用领域,不但为计算机专业工作者提供了并将继续提供大量的工作岗位,而且为年轻一代的计算机专业工作者提供了广阔的舞台和发展空间。

第一,计算机(**信息技术**,**简称** IT)的应用将越来越广,将会更深入地影响人们的生活和工作。如现在流行的物联网,实际上就是在每一个“物”上都植入计算机(**处理器**)和通信装置,使得这些物和物之间、物和人之间都发生有机的联系,让“物”活起来。IT 行业在未来 20 年乃至更长的时间都是朝阳产业,它需要大量的专门化人才,就业前景非常好。

第二,计算机专业对人的训练是其他专业不能比拟的。严格地讲,计算机是一种装置,一种人发明的工具,属于技术和工程的范畴。但从事计算机需要非常缜密的逻辑思维能力,要有抽象具体问题、设计相应算法和构

建模型的能力。此外，还要有实际动手能力，即让计算机实现你的想法。

尽管学计算机专业的学生不像数学系的学生需要学习大量的数学并具备很强的推理和定理证明能力，但与一般的工科专业相比，它对高等数学的要求较高。此外，还要学习数理逻辑、高等代数等数学课程。在中国计算机发展早期，大部分人都是从数学专业转过来的。

但计算机又不同于数学。数学只在纸上演算完毕即可，而计算机专业要求用计算机解决问题，这就要求有很强的解决实际问题的技术和工程能力。计算机专业除了数学类课程外，还要学计算机基础课程和应用方面的课程，此外要有大量的实践时间，最重要的就是要编写程序。这是基本功。

第三，计算机专业人士容易转行。由于上述计算机专业所受到的特别训练，使得一个计算机专业人士除了从事计算机相关的专业工作外，还容易转行，比如从事技术管理、市场营销、职业经理人、媒体、政府管理、社会组织、教师等。这使得你在职场上游刃有余。当然，你要接受到好的训练才可能做好这些的工作，否则你可能还不如“农民工”。敝人有幸在 34 年前选择了计算机专业，现在从事社团工作已经 15 年，离开技术第一线，但将近 20 年的计算机专业的学习和工作对我的人生帮助很大，抛开技术层面的事，在思维方面也大有裨益。

报考计算机专业有四看

如果选择计算机专业，需要注意几个问题：

一、看这个学校是否有非常专业的教师，有多少教授和副教授，他们的专业背景如何，在从事什么研究，教授和副教授是否给本科生上课。千万不要光看这些学校是不是“211”学校，是不是有很多研究经费，因为有些东西和你无关，比如科研经费。一定程度上，科研经费多的学校，教授可能反而不给本科生上课，这点要特别注意。

二、看这些学校的实验条件如何，是否和企业有合作关系，学生是否有到企业实习的机会。如果没有，学校又没有课题训练学生，那你学完后基本上是不会动手的，毕业后肯定很难有竞争力。

三、看这个学校每年招多少学生，要优先考虑那些招生少的学校，比如每年不超过 200 人。如果每年招 400 人以上，4 年就是 1600 人。那肯定是放羊式的教学，建议暂不考虑。

四、看这个学校的学费多高。如果每年在六七千元以内，还可能有奖学金。这样的学校不错，因为它不把学生的学费当做主要的财政来源。如果在一万元以上，而学生又特多，这种学校就是以收学费为主要目的了，建议要非常慎重选择。

计算机专业除了大家热议的一两所“国内顶级”大学外，国内还有几所非常不错的大学是比较理想的选择，特别是以工科见长的学校。

当然，上述所说的计算机专业还包括相关专业，比如信息安全、网络工程、信息技术、软件工程等，现在这些专业的界限比较模糊，很难严格区分。既然受的是一种训练，那么具体哪个专业也就无妨了，关键要看辅导你的老师，看他是真有两下子，还是“纸上谈兵”。如果你能到你报考的学校参观一下，找学长了解一下，甚至能找“高人”咨询一下，那报考起来就会减少盲目性，更有针对性。

杜子德

（作者系中国计算机学会秘书长、研究员）

杜子德在 CCF NOI2012 开幕式上的讲话

杜子德 中国计算机学会秘书长

同学们，今天我要对你们讲几句话。

大家满怀信心参加一年一度的青少年信息学奥林匹克大赛，祝贺你们获得了一个非常难得的机会，这种机会使你们更有可能获得成功。

长期以来，在中国社会中有一种现象，就是大家大多为资格而战，过去的科举制度、现在的公务员招考以及大学入学考试无不如此。现在，一个公务员岗位有数千人报考，当上公务员之后，会顺着科长、处长、局长、部长乃至更高职位的阶梯往上爬，结果又怎么样了呢？是不是为官一时造福一方？本应如此，可是我们天天从媒体上看到触目惊心的贪官的案例，小到一个村官，大到部级干部。在你们面前有二种力量驱动你们做事，一种是为了资格，一种是为了一个崇高的目标即将来能够改变社会。你们成长到现在，大部分时间都是在为了资格而战，从幼儿园占坑班到小学阶段的奥数班，一直到高中的各种学科竞赛和特长比拼，大多数都是为了一个目标，那就是资格。

参加本次大赛的347名选手获得了参赛资格并大多获得到高校深造的机会，其中有65%的选手将获得奖牌，20%获得银牌，12%获得金牌，60名同学将进入国家集训队，其中4名代表中国参加国际大赛。你们中前100名部分选手将直接进入清华大学或北京大学深造。事情是否结束了？没有！大学毕业后，你们可能还将选择出国留学，取得学位，进而参加工作。回顾过去的二十多年，获得国际奖牌的中国选手就超过80名，我们数数，这些人中，有多少在科学、新技术、创办企业、社会管理或社会公益方面为社会作出了杰出贡献？有谁像比尔盖茨20岁时创办了微软公司？像乔布斯那样在21岁时创办了苹果公司？像林纳斯·托瓦兹（Linus Torvalds）在21岁写出了操作系统Linux？又有谁像扎克伯格那样20岁时创建了社交网络？没有！不过，我知道我们的国际金牌获得者王小川现在搜狐担任首席技术官，很有建树，竞赛委员会委员刘汝佳清华大学毕业后和其他人共同创办了和家装有关的尔宜居公司，很有勇气。可能还有其他人。但是，其他获奖选手大部分呆在国外过着安逸的生活。我也听说，今年有一位获得国际金牌选手被某公司高薪聘用，一时成为业界新闻。我想知道，这就是我们的追求吗？是什么影响了我们的创造力？是什么妨碍我们大胆创新？我们为什么总跟随在西方人后面享受人家创造的新文明、新技术和新工具？

诚然，资格和机会非常重要，它是成功的必要条件，没有必要指责同学们为资格而战，问题在于，你的价值追求是什么，你的终极目标是什么？如果没有崇高的目标驱使你，即便你再聪明再有机会，你也是一个平庸的人，不会对社会的发展有多大贡献。

Linux的发明人林纳斯说过，人做事的动机分三类：一是求生，二是社会生活，三是娱乐，当我们的动机上升到一个更高的阶段时，我们才会取得进步，我们不是仅仅为了求生，而是为了改变社会。这就是一个对计算机发展作出巨大贡献的一个芬兰青年的心声。同学们，我们要有改变社会的意识、勇气和能力，而不是仅限于过一个好的生活，因为对整个社会的发展负有责任，而不是一个旁观者。这就要求我们看到不合理的现象时要勇于改变，而不是任其蔓延，要用我们的智慧使得其他人过得更好。

和同学们相比，我远没有你们那样幸运，41年前因为家庭成份上不了中学，该上大学的时候因为文革时期大学停办而不能上大学。不过我还是幸运的，23岁上了大学，而后逐渐获得了做事的机会。16年前，阴差阳错把我派到学会工作，改变了我人生的轨迹。在学会秘书处这个位置上，我如果总是想着如何安逸和获得很多好处（尽管有人经常想通过送礼影响我的判断和价值观），那就不是一个称职的秘书长，我就应该下台。因此，我尽力想着改变，让这个信息学奥赛更好，让学会更好，让CCF的会员乃至全国的计算机专业人士从学会获得更好的服务，进而让这个社会更好。你们的条件、起点、智商和机会要比我好得多，我希望你们未来的成就更大。请在五年后、十年后、二十年后把你的成就告诉我们，告诉你的父母、你的老师、你的母校和你的祖国。

最后，我想说的是，组织这么大规模的一个活动需要付出许多，江苏省委省政府、教育厅、省科协、常州市政府、市教育局、市科协以及常州市高级中学在财政、设备和人力方面给出巨大支持，省长李学勇先生专门发来贺电祝贺和勉励我们，还有以及许许多多的专家和志愿者为我们提供服务，让我们怀着感恩的心情向他们表示衷心的感谢！

我也期望你们取得好成绩，不仅在程序设计方面，更在人生体验和社会责任感方面。

谢谢你们！

（2012 年 7 月 29 日常州）

多数奥赛金牌得主为何难成大器

高校只使劲“掐尖”不用心“育苗”是原因之一

李新玲 《中国青年报》（2012 年 08 月 07 日 03 版）

常州高级中学的体育馆里，347 名选手一人一台计算机，紧张地计算、编程，偌大的体育馆里只有敲击键盘的声音，在长达五个小时的比赛中，选手们可以喝水、吃面包补充能量。

这是 8 月上旬举行的 2012 年第 29 届全国青少年信息学奥林匹克竞赛（简称 NOI）的赛场。

这也是 NOI 最后一届与高考保送直接挂钩的比赛。教育部 2010 年年底宣布调整高考加分项目，其中一条就是参加全国中学生(数学、物理、化学、生物学、信息学)奥林匹克竞赛获得全国决赛一、二、三等奖的学生，不再具备高校招生保送资格。

最后一届与保送挂钩的竞赛上各大名校疯狂“抢”人

NOI 被誉为信息学的“巅峰赛事”，是国内信息学领域面向中学生的最高水平的大赛。每年通过全国联赛、全国竞赛、各省区比赛等系列赛事，有几百名学生被保送进入全国各大名牌高校。

政策的变化，让相关中学教师有点无所适从。湖南师大附中李淑平老师指导的学生每年都取得不错的成绩，但她现在心有疑虑：“不知明年这个比赛是不是还能吸引这么多学生。”

作为主办方，中国计算机学会考虑的是今后如何让高校能够真正选拔到自己看中的学生。

NOI 始于 1984 年，2000 年 9 月，在中国首次举办了国际信息学奥林匹克竞赛活动，获得成功并产生了良好的国际影响。教育部于 2001 年年初正式出台文件，明确提出在高中阶段取得 5 项（**数学、物理、化学、生物学、信息学**）学科奥赛省级一等奖的选手可免试保送上大学。

但是，NOI 的科学委员会对此并不领情，他们担心执行这一规定会引发一系列负面效应。之后，他们撰写了一份报告——《关于取消保送信息学奥赛省级比赛一等奖上大学的建议》，充分陈述了反对将信息学奥赛成绩和高考挂钩的理由，直接递交到中国科协并转呈教育部。

可是，这些建议没有得到足够重视。之后，奥赛不断曝出丑闻。NOI 全国竞赛没有出现过问题，但是在省级赛区中，出现过“钱分”交易。

“取消保送是我们一直呼吁的，现在马上就开始实施了。奥赛本身其实是在‘验成色’，要让高校来确定如何选择学生，而不是教育行政部门来决定。”这是 NOI 科学委员会副主席、北京航空航天大学教授尹宝林的观点。

每年 NOI 竞赛都有高校招生人员到现场争夺学生，今年是高校来的最多的一年。竞赛一结束，来自北京大学、清华大学、复旦大学、中国人民大学、上海交通大学等 13 所著名高校的招生老师就已经到达现场，开始发放预录取通知书。这届竞赛获得金牌和银牌的高二选手几乎全被北大、清华免试录取，其他年级的金牌选手也可以保留保送资格。

据江苏省参赛队有关负责老师说，金牌选手大部分被清华抢录了，银牌选手现在也很走俏，一名选手往往能收到三四个知名高校的免试录取通知，挑选的余地很大。光常州高级中学就有 6 人被北大、清华“抢”走。

清华大学计算机系随后在自己的招生官方微博发布消息：清华大学在第 29 届全国青少年信息学奥赛招生中成绩骄人。清华大学由招生办公室和计算机系组成的招生小组到现场对竞赛成绩优异、综合素质全面的高中生进行择优录取，全部 34 名金牌选手中有 28 人将进入清华深造，成绩排名前 14 的全部被清华录取。

特殊人才，掐尖之后要特殊培养

76 岁的 NOI 科学委员会名誉主席、清华大学教授吴文虎几乎坐镇了 NOI 的每一届比赛。

吴文虎清楚地记得 1986 年 NOI 开始步入正轨时，他第一次担当出题人出的题目。虽然使用的是最初级的 BASIC 语言，而且计算机刚刚进入中国，但那批中学生能在限定的一个半小时里完成题目，非常出色。

吴文虎至今记得那次竞赛的前几名学生。第一名叫李劲，来自上海，5 道题全部做出来，后来进入清华大

学，一直读完博士；第二名是来自广东韶关一中的廖恒，这名才上初二的学生在一个半小时内做出了三道题。对于这样的好苗子，吴文虎教授会同其他 NOI 科学委员会成员与有关部门协商，将廖恒转学到北京景山学校读高中。这个少年只用一年时间就把高中三年课程全部学完，并顺利通过了清华大学的测试，进入计算机专业读书。

说起当年高校招生录取的大胆和不拘一格，吴文虎感慨：“这样突破常规的培养和录取，现在反而做不到了。人才的选择和培养方式，不应受到过多干涉。”

1987 年，吴文虎带了几个在竞赛中脱颖而出的学生到美国访问，在苹果公司看到了苹果 2 的原型机，当时乔布斯刚刚被迫离开苹果公司。

面对这台当时最受欢迎的个人电脑，吴文虎问几个学生有没有信心制造出中国自己的产品来，几个学生不约而同回答：“只要给条件，我们就行！”这几个学生中就有李劲和廖恒。

回国后，吴文虎正好接到一个七五攻关项目。其中一个子项目是要做一个软件用于当时的中华学习机。他找了 10 个本科生，其中有两个是大一新生。这些学生使用吴文虎从上海计算机厂借来的计算机，用了一年时间，完成了程序设计。专家鉴定后给予了非常高的评价。

可是，这些有天赋更有兴趣的学生，在完成本科和硕士的学习后大部分都出国了。这让吴文虎感到无奈：“他们都很优秀，无论是自学能力还是创造力都很强，可他们大部分不会在国内念博士，在国外拿到博士学位后也不愿回来。”

吴文虎认为其中原因有多个，其中之一是高校只管掐尖，把好的学生拼命抢到自己学校，可是抢到手之后，对于怎么培养并没有太用心。把在某一方面非常出色的学生与其他学生放到一起，不考虑学生的个性发展，让他们很快就“泯然众人矣”。

对于高校培养人才的方式，负责输送人才的中学教师也颇有微词。常州高级中学的曹文老师每年指导的学生都会有十来个人和各大高校提前签约。曹文知道，有些学校虽然招生时不遗余力地“抢”人，但在日常的专业教学中，教材陈旧，教学方法单一，学生所学知识与应用脱节。甚至一些高校计算机系四年都不给学生安排实习。

“通过学科竞赛被高校录取的学生在某一方面比那些高考状元有专长，可是他们没有得到重视。”曹文老师说。

吴文虎认为，特殊的学生需要校方提供特别发展的条件。十几年前，有一个通过 NOI 竞赛进入清华大学计算机系读书的大一新生，吴文虎交给他一个语音识别软件的设计任务，只用了两个小时给他讲原理，然后给了他一本博士论文，这个学生半年就完成任务，并获得了当年最佳软件奖。

无论是参加奥赛还是人生不能只是为资格而战

虽然 NOI 是教育部认可的五大学科竞赛之一，但由于其竞赛内容与高考内容无关，吸引的更多是真正有兴趣的学生。可是，还是有一些学生参赛是为了拿到大学的通行证，甚至用一两年的时间全部备战 NOI，放弃其他学科的学习，可谓是孤注一掷。

对于这样的选手，中国计算机学会秘书长杜子德是“恨铁不成钢”：“回顾过去的 20 多年，获得国际奖牌（**信息学**）的中国选手就超过 80 名，我们数数，这些人中，有多少在科学、新技术、创办企业、社会管理或社会公益方面为社会作出了杰出贡献？有谁像比尔·盖茨 20 岁时创办了微软公司？像乔布斯那样在 21 岁时创办了苹果公司？像林纳斯·托瓦兹在 21 岁写出了操作系统 Linux？又有谁像扎克伯格那样 20 岁时创建了社交网络？没有！”

杜子德这番话让台下的参赛选手为之一振，他们没有想到在竞赛的开幕式上听到的不是一般常规性的致辞，而是引人思考的一连串反问。

“你们成长到现在，大部分时间都是在为了资格而战，从幼儿园占坑班到小学阶段的奥数班，一直到高中的各种学科竞赛和特长比拼，大多数都是为了一个目标，那就是资格。你们中有 65%的选手将获得奖牌，你们中的一部分人将直接进入清华大学或北京大学深造。事情是否结束了？没有！”

杜子德想告诉这些一路比上来的学子们，人生决不是为资格而战。“历届 NOI 获奖选手大部分呆在国外过着安逸的生活。我也听说，今年有一位获得国际金牌的选手被某公司高薪聘用，一时成为业界新闻。我想知道，这就是我们的追求吗？是什么影响了我们的创造力？是什么妨碍我们大胆创新？我们为什么总跟随在西方人后面享受人家创造的新文明、新技术和新工具？”

杜子德说：“我们要有改变社会的意识、勇气和能力，而不是仅限于过一个好的生活，我们要有对整个社会发展负有责任的抱负，而不是一个旁观者。我们要用我们的智慧使得其他人过得更好的远大理想。”

吴文虎也为那些为了生存而放弃科研的奥赛得主惋惜：“有些人到了国外，为了找一份工作而放弃科研，他们是有了安逸的生活，可是事业上没有太大的成就。”

当然，在一些学生身上也看到了希望。

清华大学计算机硕士研究生唐文斌是NOI科学委员会的学生委员，也是这次竞赛的出题人之一。

唐文斌曾是NOI比赛的选手，进入清华大学后，获准免修数据结构、程序设计等课程，从而赢得了时间。目前，他与同学一起设计的一个体感游戏，已经在APPSTORE里有40万的下载量。另一款街头速滑游戏也准备上线，而且已经有了投资。