NOIP 实用算法

By **Programet.cn**
Climber.pI 整理

1.模拟方法

a.用数学量和图形描述问题

计算机处理的是数学量。若要用计算机解决实际问题，需要把实际问题抽象为数学量，或者数字。比如，记事本把文字按 ASCII 码表转换为数字储存起 来，极品飞车把赛车的性能表示为数字，来权衡赛车的好坏。一个漂亮的算法，需要用数学量表示出来。

任务：现有两个软件工程的制作任务，你的团队可以接手其中任意一个。现要在两个中选择一个，需要考虑制作成本，制作成功的可能性，可获得经济收益的 多少，对你的团队知名度的影响等等因素。你如何量化地分析和解决这个问题？
提示：需要把每一项都转化为数值，必要时加入权值、计算期望。如果只考虑以上四个因素，可以得到以下数学式
综合收益=制作成功的概率*[(可获得经济收益-制作成本)*经济效益的权值+团队知名度的影响*社会效益的权值]
其中概率和两个权值是需要估计的值。
有时，我们需要用更直观的图形来描述实际问题。
图论就是一个绝佳的方法。图是一种表示离散量间关系的形式，它也是一种思想，常被用于建模。同时，前人也为我们提供了很多现成的图论算法，能够解决很多常 见问题，这些将在之后被提到。
矩阵也是一种常见的方法。有时矩阵被表示成三角形的形式，比如“杨辉三角”。矩阵常常和数学有关，在计算机计算时常常利用矩阵的递推式。这也将在后面被提 到。

b.模拟计算过程

模拟方法是最常见、最直接的算法构建方法。
任务：编程实现欧几里得算法（辗转相除法，求两个数的最大公约数 gcd(a,b)）
提示：
欧几里得算法原理：gcd(a,b)=gcd(b,a mod b)
欧几里得算法的图形描述——

```
| 168 63 |      | 168 63 | 2
               |  42    |
```

1.写下两个数　2.将两数相除，余数写在较大的数下面

```
  | 168 63 | 2                                  | 168 63 | 2
1 |  42 21 |                                1 |  42 21 | 2 ——整除
```

3.将每列最下面的数相除，余数写在被除数下面　4.重复步骤3直至整除，此时最后写下的余数即为开始时两数的最大公约数
这是一个简单的模拟算法，实际过程中，量间的关系往往比这复杂得多，因而，模拟算法可以是很复杂的。
有些模拟算法还涉及到图形和其他复杂的数据结构，这需要我们熟练地运用语言，巧妙地把其他关系转化为数学量间关系。

c.模拟时的优化

如果处理不当，模拟方法写出的程序会非常长。这要求我们在模拟是将相似的步骤合为一体，适时利用函数简化程序。
以上面的欧几里得算法为例：

```
/*实现时，若将左边一列数最下面的记为L[1000]、右边一列数记为R[1000]，显然是不明智的，因为只有每列最后一个数会在以后的计算中用到*/
/*实现方法一：及每一列最后一个数分别为L、R。输入即可是L、R，返回gcd(L,R)*/
int Euclid_1(int L,int R)
{
    for(;;)
    {
        L=L%R;
        if(L==0)return R;
        R=R%L;
        if(R==0)return L;
    }
}
/*我们发现有两步是相似的。因而我们可以把它简化为实现方法二*/
int Euclid_2(int L,int R)
{
    int t;
    for(;;)
    {
        t=R; R=L%R; L=t;
        if(R==0)return L;
    }
}
/*甚至我们可以写成递归形式。以下是实现方法三*/
int Euclid_3(int L,int R)
{
    if(L%R==0)return R;
    else return Euclid_3(R,L%R);
}
```

这个实例主要体现模拟算法的简化过程。虽然在这样小的程序段中，这样的简化作用不大，但遇到复杂的模拟问题，这种利用相似性的简化便非常有用了。应 当重视这样的代码优化。

d.高精度计算算法

竞赛中经常会用上一些很大的数，超出长整型的数值范围。这时我们需要使用高精度计算算法。这种算法可以把数值范围增加到我们想要的程度。

高精度函数往往包括加、减、乘、输入、输出五种。

实现高精度计算时，常常使用模拟算法—模拟小学竖式运算。我们把一个高精度数表示为一个数组 H[]，数组中的某一个数相当于高精度数的某一位。

要注意的是，输出时往往要求以十进制形式输出。因而，高精度数每一位都应便于直接输出。这样，每一位上的最大数都应是10^n-1。简单地说，若把 H[] 定为 unsigned 型，高精度数每一位上最大数最好为9999。这样既能尽量利用储存空间，又便于输出。

另外，高精度数有时会用到负数。在补码的体系中，仍然可以按正数的方法处理负数，但是要特别注意输出时的问题，和对溢出的防止。

任务：实现高精度运算加法

提示：设计函数 unsigned *HAdd(unsigned N1[],unsigned N2[],unsigned Ans[])，从末位起相加，注意是否进位。

显然，减法作为加法的逆运算，也很容易实现。另一种聪明的办法是，对减数每一位取补码，在做加法

即可。请读者自行实现高精度减法。

高精度乘法困难一些。我推荐的方法是，先考虑多位高精度数乘一位高精度数的过程。多位高精度数乘多位高精度数可以转化为多位高精度数乘另一高精度数每一 位，再将结果相加的过程。下面给出多位高精度数乘一位高精度数的源代码：

```
#define H_Bit 256 /*定义常数：高精度数位数*/

unsigned *HTimesInt(unsigned N1[],int N2,unsigned Ans[]) /*N1[]为多位高精度数，N2为高精度数的一位，Ans[]为另一高精度数，用于储存结果*/
/*这里允许N1与Ans 相同*/
{
    unsigned i,up=0;
    unsigned long temp;

    for(i=H_Bit-1;i&lt;=H_Bit;i--)
    {
        temp=N1[i]*N2+up;
        up=temp/10000;
        Ans[i]=(unsigned)(temp%10000);
    }

    return Ans;
    /*函数返回作为答案的高精度数首地址，这样更便于高精度运算函数的使用，例如连乘可以写成HTimesInt(HTimesInt(N1[],N2,Ans[]),N3,Ans[])*/
}
```

高精度数的输入输出需要专门的函数。针对不同语言的不同特点，可以比较容易地写出这两个函数。但要注意输出非首位数位上的“0”。

习题

模 拟方法的习题 ——感谢 深蓝评测系统 提供习题

2.排序算法与算法时空复杂度

a.简单排序算法

简单排序算法包括冒泡排序、插入排序、选择排序。这三种算法容易理解、编写方便，适用于数据规模较小的情形。
冒泡排序（Bubble Sort）的基本思路是：（以从小到大排序为例）从前到后逐个比较相邻两数，若前数大于后数，就将两数交换。不断重复这一过程，直至全部数字已按从小到大 排好。
考虑到实用性的问题，插入排序和选择排序这里不再介绍。
对于 NOIP 提高组而言，这些算法时间复杂度过高，很难应付较大的数据规模。建议尽量不要采用简单排序算法，除非你十分确信数据规模在可承受范围之内。

b.快速排序、堆排序

快速排序和堆排序是比简单排序快的排序算法，在竞赛中常常被采用。这里，我们介绍快速排序算法。堆排序的实现不作介绍，若想了解，可咨询谷歌或百 度。
快速排序（Quick Sort）基于分治思想。它的基本思路是：（以从小到大排序为例）取一个数作为标记元素，将比它大的数放在它右侧，比它小的数放在它左侧，再通过递归的方 法，将左侧的数用以上的方法排好，右侧的数也用以上的方法排好即可。
下面这个视频能很直观地比较冒泡排序（Bubble Sort）和快速排序（Quick Sort）：

在数据规模很大时，平均情况下快排比冒泡快很多。在处理 NOIP 提高组含排序的问题时，一般要选择快速排序或堆排序。
下面将介绍快速排序的实现（以从小到大排序为例）。
快排运用分治思想，因而要用函数的递归调用来实现：

```
void QuickSort(int a[],int st,int stp) //这里也可以定义成 void QuickSort(int *st,int len)。为了便于理解，我使用前一种写法。
{
    int mid;
    mid=partition(a[],st,stp); //partition()用于确定标记元素的位置。
    if(l<mid-1)QuickSort(a[],st,mid-1);
    if(mid+1<r)QuickSort(a[],mid+1,stp);
}
```

现在的关键问题在于如何写 partition()。
写法一：对于数列 5 6 7 5 3 8 1 6 2
1.取首个元素做标记元素，取出它，令指针 p 指向最右边的数的右边
——_6 7 5 3 8 1 6 2 p-
2.将 p 向左移动到小于标记元素的数（或空缺处）为止。若指向空缺，则跳到5；否则将该数和 p 移到空缺处—— p-2 6 7 5 3 8 1 6 _
3.将 p 向右移动到大于标记元素的数（或空缺处）为止。若指向空缺，则跳到5；否则将该数和 p 移到空缺处—— 2 _ 7 5 3 8 1 6 p-6
4.重复2和3。
2 p-1 7 5 3 8 _ 6 6

2 1 _ 5 3 8 p-7 6 6

2 1 p-3 5 _ 8 7 6 6
5.把标记元素放入空格处—— 2 1 3 5 p-5 8 7 6 6
写法二：写法二比写法一短一些，但理论上讲，写法二要慢一些（因为所作赋值运算多一些）。下面给出源代码与分析：

```
void QuickSort(long a[],long st,long stp) //这里将partition()结合进QuickSort()中
{
    long t,n,l,r;

    n=a[st];
    l=st+1;
    r=stp;

    for(;;)
    {
        for(;a[l]<=n && l<=stp;l++); //从右找，找到一个小于标记元素的数
        for(;a[r]>=n && r>st;r--); //从左找，找到一个大于标记元素的数

        if(l>=r)break; //如果 l 在 r 右侧，则跳出
        t=a[l];a[l]=a[r];a[r]=t; //交换，使小于标记元素的在左，大于标记元素的在右
    }
    a[st]=a[r]; //取出最右侧的小于标记元素的数写入空缺
    a[r]=n; //空缺处放入标记元素

    if(r-st>1)QuickSort(a,st,r-1);
    if(stp>l)QuickSort(a,l,stp);
}
```

以上快排实现方法的最差情形是排列整齐的情况，这时它的运行效率会很低。为了解决排列整齐的情形，我们可以使用随机快速排序法，即随机选取一个数作 为标记元素（实现时，将其与第一个数交换即可）。

c.算法时空复杂度

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 O() 表示。
通常表示时间复杂度时，我们只保留影响最大的项，并忽略该项的系数。
例如主要运算为赋值的算法，赋值做了3n^3+n^2+8次，则认为它的复杂度为 O(n^3)；若主要运算为比较，比较做了 4*2^n+2*n^4+700次，由于数据很大时，指数级增长的2^n 影响最大，我们认为它的时间复杂度为 O(2^n)。
常见的时间复杂度有下列几个：
O(n) ——贪心算法多数情况下为此时间复杂度
O(nlbn) ——有时带有分治思想的算法的时间复杂度（注 lbn 表示以2为底的 n 的对数）
O(n^2) ——有时动态规划的时间复杂度
O(n^3) ——有时动态规划的时间复杂度
O(2^n) ——有时搜索算法的时间复杂度
O(n!) ——有时搜索算法的时间复杂度
有时时间复杂度中含有两个或多个字母，比如遍历一个 m*n 的矩阵，时间复杂度为 O(m*n)。
要注意的是，时间复杂度相同的两个算法，它的实际执行时间可能会有数倍的差距，因而实现时要特别注意细节处的优化。

NOIP 提高组执行时限常常为1s。一般认为，将数据规模代入到时间复杂度，若所得值小于或接近于1000000，就是绝对安全的、不超时的。
例如，O(n^2)的动态规划算法，可承受的数据规模是 n≤1000；O(2^n)的搜索算法，可承受的数据规模是 n≤20；O(n!)的搜索算法，可承 受的数据规模是 n≤9。
实际上，以现在的 CPU 运行速度，5000000也应该不成问题。
空间复杂度：一个算法消耗储存空间（内存）的大小，用 O()表示。
空间复杂度的表示规则与时间复杂度类似。
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。
下面我们简单地分析一下简单排序算法、快速排序、堆排序的时空复杂度。
这三种算法都是基于比较的排序算法，以比较次数作为主要运算。
简单排序算法最差时需做 n^2次比较，平均情况下时间复杂度通常被认为是 O(n^2)。
快速排序最差时需做 n^2次比较，可以证明平均情况下需做 nlbn 次比较，时间复杂度是 O(nlbn)。
堆排序时间复杂度是 O(nlbn)。
空间上，三者都不需要额外开辟暂存数组，快排递归调用时需要使用稍多一些的储存空间。
综合来看，快速排序、堆排序优于简单排序算法。
另外，可以证明基于比较的排序算法时间复杂度下界为 O(nlbn)。

d.时空的简单优化方法

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。
+、- 运算非常快（减法是将减数取补码后与被减数相加，其中位运算更快一些，但是减法也比加法稍慢些。）
* 运算比加法慢得多
/ 运算比乘法慢得多
比较运算慢于四则运算
赋值运算慢于比较运算（因为要写内存）
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。
下面来举一个例子：计算组合数 C(m,n)——n 件物品取出 m 件的组合数。
C(m,n)可用公式直接计算。C(m,n)=n!/((n-m)!m!),C(m,n)=n(n-1)(n-2)…(n-m+1)/(n-m)!。显 然，有时所作的乘法少很多，会快一些。
可是这样算真的最快吗？
另一条思路是 C(m,n)=C(m,n-1)+C(m-1,n-1),递推下去,直到可利用 C(1,k)=k=C(k-1,k)为止。我觉得这种只用加法 的运算会快些，尽管加法多一些。（我没试验过，你可以去试一下。）
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——例：数组中每个数都只有0、1两种可能，则可以按位储存，空间压缩为原来的八分之一。
覆盖旧数据——例：矩阵中每一行的值只与上一行有关，输出只和最末行有关，则可将奇数行储存在第一行，偶数行储存在第二行，降低空间复杂度。
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。

e.线性时间排序

基于比较的排序算法时间复杂度下界为 O(nlbn)。因而，若还要降低复杂度，要放弃基于比较的排序

算法。
有一类排序算法叫做线性时间排序，它们的时间复杂度为O(n)。下面介绍一种。
计数排序思路：开辟暂存数组b[]，b[k]表示欲排序数组a[]中k出现的次数（需要遍历a[]），最后遍历b[]，可将a[]排好。
这种想法非常简单，实现也很容易。事实证明，在a[]取值范围很小（如整型范围）时，它是很高效的排序算法，比快排快不少。可是a[]取值范围较大（如长 整型范围）时，它的执行时间会变长，而且数组b[]有时开不出来。
实际上计数排序时间复杂度为O(n+m)，空间复杂度也为O(n+m)，m表示a[]取值范围。若m很大，则也不能在时限内执行完。

f.归并排序

有一种排序时极为常见的情形：有一张成绩表，记录着许多学生的成绩，要将他们按成绩排序，但成绩相同者的相对顺序不能改变。
换句话说，ABCDE五人，A、C、D成绩相同，显然排序完之后会排在一起，现在的要求是：他们排在一起的顺序也必须是ACD，不能是ADC、CAD…
这样的实际问题涉及到排序的稳定性。
排序的稳定性：一个排序算法，若可使排序前后关键字相同的项相对顺序不变，则称该排序算法是稳定的排序算法。
下面我们来考察常见排序算法的稳定性。
在编写合理的情况下，简单排序算法是稳定的；快速排序、堆排序是不稳定的（你可以好好想想这是为什么）。
在NOIP中，往往排序是没有附带其他项目的，也就不要求排序稳定。快速排序、堆排序仍然是最佳选择。
可是有没有时间复杂度为O(nlbn)的稳定的排序算法呢？有的。
归并排序基于分治思想：把要排序的数组平分两半，对两部分分别排序（递归地）后再合并起来。合并时，将一个数组按顺序插入另一个数组中，需要开辟一个暂存 数组。利用空间优化，可只用开辟一个与原数组等大的数组。
归并排序的源代码会放在本章的附件中。请读者自己研究。
归并排序的优缺点都很明显。无论情形如何，它的比较次数、赋值次数都稳定在nlbn，没有最差情况，运行时间与快速排序、堆排序相当。而且，它是稳定的排 序算法。
但是，它的内存占用回达到快速排序、堆排序的两倍，竞赛时使用容易造成内存超出限制。
NOIP初赛曾考察过归并排序。问题大意是：找出一个算法，使五个数在n次比较（两两比较）后一定能排定次序，求n的最小值。
在快速排序、堆排序的最差情况下，需要10次、9次比较。可是，使用归并排序只需要7次！记住：归并排序没有最差情况。

g.合理选用排序算法

下面是本章讲过的排序算法的优缺点比较：（这里只讲最主要的）

排序算法	时间复杂度	优点	缺点
简单排序	O(n^2)	编写方便	执行时间长
快排	O(nlbn)	执行时间短	很差情况下执行时间长、占用内存多
堆排序	O(nlbn)	执行时间短	编写有点麻烦，有较差的情况
计数排序	O(n+m)	编写方便,取值范围小时很高效	取值范围大时效率低、易超内存限制
归并排序	O(nlbn)	稳定的排序算法，无较差情况	占用内存很大

竞赛中首选快速排序、堆排序。但有时也应比较各排序的优缺点，依实际合理选用。

习题

排 序的习题 ——感谢 深蓝评测系统 提供习题

3.搜索

a.复杂的模拟问题与利用相似性

在讲模拟方法时我们讲过利用相似性来简化算法。现在，我们继续关注这个问题。
搜索算法是一种"模拟"思维的算法，比较接近平常的思维。与模拟算法相比，它更深刻地利用了相似性。
为了更好地说明，下面举一个例子：
有一把有 n 位字母的密码锁，每一位上的字母都可从 a 到 z 选取。现密码被遗忘，开锁时，请给出一个方便的方法，使每个字母组合都被尝试过。
最容易想到的方法便是，按 aa…aa，aa…ab，aa…ac，…，aa…az，aa…ba，…，zz…zy，zz…zz 这样的字典序来尝试。
我们可以这样考虑：
先选定第一位，再选定第二位，……，直到选定第 n 位，形成一个完整的字母组合。
具体地，在每一位的选取时，都从 a 开始，到后面位的字母组合全部尝试过，再跳到下一个字母；若（非首位）已经跳到 z 而还需再跳一个字母时，就跳到 a，同时 让它的前一位跳到下一个字母。
例如，n=3时，形成的字母组合的顺序是

```
a  a  a  - aaa
     b  - aab
     c  - aac
     ...
     z  - aaz
   b  a  - aba
     b  - abb
     ...
     z  - abz
   c  a  - aca
   ......
   z  z  - azz
b  a  a  - baa
   ......
   z  z  - bzz
c  a  a  - caa
.........
z  z  z  - zzz
```

以上描述的是一种常见的遍历方法。
我们注意到，选定每一位的过程是极其相似的。我们需要利用这种相似性。

b.函数的递归调用

结构化编程语言提供的最大好处无疑是函数的递归调用。
如果把函数看成解决某个问题的过程，那么递归就可以看成把问题变成相似而更小的问题的过程。注意这两个关键词：相似、更小。递归的本质是利用相似性。
我们接着讲上面提到的密码锁问题。现在我们要把尝试过的字母组合都输出到屏幕上。
我们用递归法来完成这个过程。写递归体一般分为两步：把大问题化成小问题、解决最小问题。

```
char string[1000],n;

void code(int Left) //递归体，Left 表示还需要决定的位数，这个值随问题的减小而递减。
```

```
{
    if(Left>=1) //把大问题化成小问题
        for(string[n-Left]='a';string[n-Left]<='z';string[n-Left]++)
            code(Left-1);
    if(Left==0) //解决最小问题
        printf("%s\n",string);
}

int main()
{
    fscanf("%d",&n);
    string[n]='\0';
    code(n);
    return 0;
}
```

分析这个方法，得知其时间复杂度为O(26^n)。
补充一句，上面的过程也可用n个for的嵌套来实现（你能做到吗？）。

c.栈与深度优先搜索

搜索分为深度优先搜索、广度优先搜索两种。下面用树区分这两种搜索方法。
对于树，从根节点开始，查找（遍历）各节点，分两种方式：
从某一节点向下扩展时，若先遍历其子节点，再查找其子节点的子节点—广度优先搜索。
从某一节点向下扩展时，若遇到子节点就"深入"到子节点的子节点，直到叶子节点再返回—深度优先搜索。
对于7顶点的完全二叉树，两种方式所到达的顶点顺序如下：

```
   1         1
 2   3     2   5
4 5 6 7   3 4 6 7
```

广度优先　深度优先
注意：搜索面对的数据结构往往不是树。
显然，深度优先搜索的扩展方式类似于上面叙述的函数递归。这里，我们先分析它。
深度优先搜索在实现时有两种方式：递归形式、非递归形式。
递归形式利用一个函数（以整型为例）int cal(…)：

```
int cal(…)
{
 int n;
 //运算
 n=cal(…); //递归，常在循环体中
 //运算
 return n; //返回
}
```

非递归形式利用一个栈，它可以被看作是递归形式的一个改变：
当要递归地调用cal(…)时，不立即调用cal(…)，而是将其参数压入栈。等函数运行完，再从栈中弹出其参数并再次执行函数cal(…)。
这样，最后被调用的最先执行。由于后查找的是深度最大的，这样的结果是深度优先。
深度优先搜索往往采用递归方法。

一代算法宗师迪杰斯特拉（E. W. Dijkstra）极力推崇递归法。它编写方便、清晰，只是内存消耗略比非递归大。非递归法往往在担心内存溢出时使用。
深度优先搜索的巨大优势就是可以率先到达叶子节点。对于"找出一种方法"、"找出其中一个解"这样的问题有速度上的优势。这里举例分析。
八数码问题：九宫格里有八个分别填上了数字1-8，形成最初构型。每步只能把空格上下左右的数之一与空格对调。现要求找出一种方法，使若干步对调后呈现目 标构型。例如：

```
5 7 2    1 2 3
4 _ 1 -> 4 5 6
6 3 8    7 8 _
```

思路：每次将一个构型变为另一个，再递归地检查后者能否到达目标构型。时间复杂度O(4^n)。
伪代码：
int EightNumbers(构型) 这里返回步骤数，若需要知道具体步骤，则用另用栈储存步骤。

```
{
 同源构型则返回32767
 同目标构型则返回0
 n[1]=EightNumbers(变化出的构型1)
 n[2]=EightNumbers(变化出的构型2)
 n[3]=EightNumbers(变化出的构型3)
 n[4]=EightNumbers(变化出的构型4)
 step=n[1-4]最小值
 step 为32767则返回32767
 返回 step+1
}
```

d.深度优先搜索的优化

显然，以上代码是可以优化的。深搜的优化过程也叫"剪枝"。考虑到实用性，这里我们只讲最简单的剪枝方法。
对于上面的伪代码，将"同源构型则返回32767"改为"同已查找构型则返回32767"就可以显著提高效率。
但是这里需要开辟一个数组（栈），记录之前"经过"的构型。如果想避免遍历数组，可以做成哈希表，而且要压缩储存才行。
还有的简单方法，编写的时候自然会想到的。关键是要权衡，用这样的方法所增加的编写难度是否配得上所节省的运行时间。编写难度增大，调试的难度也会增大 哦。

e.队列与广度优先搜索

上面讲过用栈来非递归地实现深搜。若是把栈改成队列呢？
经过分析，你会发现，这样修改后深搜变成了广搜！
用这样的方法可以实现广搜。
用数组实现队列时避免大量元素的移动。实现时，可以先算出队列元素的上限max，开辟a[max]，并设a[st]为队列起始（st初值为0），队列的第 i 项储存在a[(st+i-1)%max]。
这样，出队列只用将st=(++st%max)即可。
要注意的问题是，广搜类似于递推，往往需要开辟空间储存每一步"递推"所得到的值。

f.广度优先搜索的优化

广度优先搜索比较容易优化，运行时间往往比深搜短一些（不过内存占用比深搜大得多）。另外，广搜有时可以清晰地反映搜索深度。
若上面的八数码问题改为"现要求找出一种方法，使呈现目标构型经过的对调步数最少"，则用广搜更好。

广搜常用的优化方法：
哈希表法—记录队列中已有节点，用于判断是否需要扩展节点。
A*算法—构造估价函数。
双向广度优先搜索—从源节点、目标节点一起开始搜索。
由于 NOIP 提高组近年来几乎不出搜索题；可用搜索的题目，由于搜索时间复杂度太高，数据规模太大，搜索只能得部分分数。加之搜索思路较简单，搜索 法这里不再详细叙述。若想了解，大家可以用搜索引擎搜索。

习题

搜 索的习题 —感谢 深蓝评测系统 提供习题

4.贪心方法

a.工程计划模型

我们常常碰到这样的问题：完成一个工程需要若干个步骤，每个步骤都有若干种方法，图示——

步骤 a　步骤 b　步骤 c　...　步骤 n

　　方法 b1　方法 c1

方法 a1　方法 b2　方法 c2　　方法 n1

方法 a2　方法 b3　方法 c3

　　方法 c4

每个方法有一个权值（如效率、质量），其大小往往和其他步骤中选取的方法有关。有些时候权值无意义，表示方法不可选择。要求给出一个方法组合，是权 值和最大。

在这里，暂且把它称作"工程计划"。很多实际问题都可以归纳为这个模型。

对于不同形式的工程计划，我们有不同的解法。

若权值与整个过程或前后步骤的方法选择都有关，我们使用搜索算法——时间复杂度高得吓人。

若每个权值只与上（或下）一步或少数几步的方法选择都有关，我们使用动态规划——有比较高的效率，在下一章会讲到。

若每个权值与其他步骤的方法选择都没有关系，我们使用贪心方法。

b.部分背包与每步最优

强调：每个权值与其他步骤的方法选择都没有关系。这样每步最优就可以得到全局最优——每一步都取最大的权值就可以了。

换而言之，贪心算法要求，局部的贪心选择，可以组成全局的最优解。

在实际问题中，这是需要证明的。如果这个无法证明，贪心算法所得的解不是最优解，一般只是较优解（较优解可为搜索剪枝提供方便）。

下面是贪心算法最经典的例子：部分背包问题。（下一章会讲到另外两种背包问题。）

问题：有 N 件物品和一个最大载重为 M 的背包，每件物品都有相应的重量和价值。现要求给出一个存放方案，使背包中物品总价值最大。部分背包要求，每件物品都 可只装入它的一部分（部分重量有成比例的部分价值）。所涉及到的数字均为整数。

（注：有时该问题表述为体积形式，即背包体积有限，每件物品有体积和价值。在本系列我选择表述为重量形式。）

思路：背包中物品总价值最高，即单位重量物品价值最高。显然，应该多装单位重量价值高的物品。这样，我们先装入单位重量价值最高的物品，再装入第二高 的……直到重量达到 M（有必要时最后一件物品只装一部分），已达到物品总价值最高。

这个证明应该很严谨吧～

该算法时间复杂度 O(n)，效率很高；而且实现很容易。这些是贪心法最大的特点。

很多竞赛题看似可以用贪心法，其实贪心法得不到最优解，原因是每一步的选择对其他步骤有影响。

数字三角形问题：有一个数字三角形（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的数的最大 值。

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

如果用贪心法，每次向最大的方向走，得到结果为1+6+8+2+3=20。可是明明还有另一条路，1+3+6+6+7=23。

问题出在哪？每次的选择对后面的步骤会有影响！第三级选了8，就选不到第四、五级较大的数了。
这个问题正确的解法会在下一章介绍。
有一个很实用的小技巧：竞赛题会给出数据规模。通过数据规模，我们可以大致判断该用何种算法。贪心算法可承受的数据规模很大，一般都会上万。如果给出的数 据规模是100或1000，优先考虑动态规划吧。

c.构造贪心算法

构造与证明是贪心算法的难点，常常要求我们要有敏锐的观察力、多角度思考的变通能力、丰富的数学知识和推理能力。
下面举几个贪心算法的例子，供大家揣摩、掌握规律。
删数问题：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。
算法构造：
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。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。
证明思路：显然，在只删一个数字时，唯有步骤1的方法能使数变小；可推理得出，删多个数字时，所有最优的方法都可看做是对步骤1的重复。也就是说，以上方 法是最优策略之一。
在文末的附件中给出了这个算法的源代码。
工序问题：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。
算法构造：
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。
3.对 F 中的元素按 A[i]升序排列，S 中的按 B[i]降序排列。
证明思路：
1.F 中的能"拉开"A、B 加工同一件工件的结束时刻，为后面的工件加工"拉开时间差"，利于节省总时间。S 中的刚好相反。因而，F 中元素放在最前一定是 最优策略之一。
2.F 中 A[i]小的前置，可以缩短开始时 B 的空闲时间，但会使 F 所有工件"拉开的时间差"缩短。不过可以证明，后者带来的损失不大于前者获得的优势。对 称地，对 S 也一样。因而步骤3是可行的。
种树问题：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。
算法构造：
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。
证明思路：解法并不唯一，关键是证明没有比该解法更好的解法。按步骤1排序之后，会发现对于每个要求，在最右边的区域内种树所得的结果总不会差于在其他区 域种树。至于为什么这样排序，留给你——读者们思考吧。
在文末的附件中给出了这个算法的源代码。

习题

贪 心方法的习题 ——感谢 深蓝评测系统 提供习题

5.动态规划

a.另一种形式的工程计划
b.记忆化搜索
c.数字三角形：递推地思考问题
d.石子合并：状态的确定
e.街道问题：状态量维数的确定与无后效性
f.0-1背包：巧妙地选取状态量
g.Bitonic 旅行：最佳的状态转化方式
h.最长非降子序列模型
i.构造动态规划算法
j.动态规划、递推、广度优先搜索的区别与转化

a.另一种形式的工程计划

上一章我们讲过，若工程计划问题中某一步权值只和上一步或少数几步的选择有关，我们可以使用效率较高的动态规划算法。
看下面这个问题：

```
  4-9
 /   \
1-2-5-3-1
 \ \ / /
  1-7-8
```

上图的数字间连了线。现要从最左边的"1"走到最右边的"1"，每次都只能沿线向右走到右边一列的某个数上，要求找出一条路径，路径上的五个数之和 最大。
当然我们可以把这理解为工程计划问题的一种形式。
这里，每一步的选择与上一步相关；似乎也与前面几步的选择有关，但是注意，前几步的影响都可以表现在上一步上，上一步方法的选择已经可以独立决定这一步每 种选择的权值。此处适用动态规划。
换句"专业"点的话，这里满足"无后效性原则"，即之后过程不受之前具体过程的影响。这在后面会具体说明。

b.记忆化搜索

再讲动态规划之前，我们先接触一下记忆化搜索。
注意到，在深度优先搜索中，用于递归的函数 cal(…)有时会被用同样的参数调用多次。你可能很容易想到，如果在第一次调用时把参数和对应的函数值储存起 来，若以后再以同样的参数调用，就不用执行递归函数，直接取出所得的值，不是快得多？
如果你能想到这些，你就已经学会记忆化搜索了。
事实上，记忆化搜索能大幅提高效率的地方，往往是在动态规划的题目中。
动态规划能解决的问题，往往可以用记忆化搜索替代动态规划解决。如果你实在无法掌握动态规划，你可以选择使用记忆化搜索。不过你要注意以下事实：
动态规划程序实现较容易，程序段短，便于调试；记忆化搜索实现就比较繁琐。
虽然记忆化搜索可以把时间复杂度降低到动态规划的水平，但是实际执行时间会大于动态规划，甚至有几倍到十几倍的差距。
这里不给出记忆化搜索的代码了。我们还是首选动态规划吧～

c.数字三角形：递推地思考问题

上一章中我们讲过数字三角形问题：
有一个数字三角形（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的数的最大值。

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

对于这个问题，贪心法得不到最优解，搜索法效率太低。我们知道，深度优先搜索利用了递归式的思维，动态规划中，我们使用递推式的思维。
递归：要知道第五层时的最大值，就要知道第四层时的最大值；要知道第四层时的最大值，就要知道第三层时的最大值……而每一步推导的方法是相似的。
递推：知道第一层的最大值，就能知道第二层的最大值，也就能知道第三层的最大值……而每一步推导的方法是相似的。
对比之下，递推思维不是从结论入手的，有时容易失去方向；但是有时却可以有很高的效率。
动态规划和普通递推的区别在于动态规划需要在每一步作比较、取最优值。
对于数字三角形问题，我们可以这样思考：设二维数组 A[i][j]，表示走到第 i 行的第 j 个数时所经过的数字和的最大值。例如对图中三角 形，A[3][2]=max{1+6+2,1+3+2}。
这样，我们又可以得到递推关系 A[i][j]=p[i][j]+max{A[i-1][j-1],A[i-1][j]}（实现时注意 A[i-1][j- 1]或 A[i-1][j]不存在时的处理），其中 p[i][j]表示第 i 行的第 j 个数的数值。
此外，我们还需要一些初始值：p[i][j]（输入），A[1][1]=p[1][1]。
最终我们可以求出 A[5][j]，结论自然是 max{A[5][j]}啦～
分析这个算法，若层数大于5，则时间复杂度为 O(n^2)。若用搜索，时间复杂度为 O(2^n)。显然动态规划效率高很多。
为了更清楚地说明动态规划算法，我们先引入一些概念。
阶段—我们把问题划分为几步，在动态规划中，这叫做“划分阶段”。数字三角形中，每一层可看作是一个阶段。
状态—每一阶段有多种选择，不同的选择会有不同的结果，我们把每阶段的不同情形叫做“状态”。每一阶段包括多个状态。数字三角形中，表示走到第 j 个数时 所经过的数字和的最大值的变量叫做状态。
动态转移方程—我们可以用一个递推式表示某阶段到下一阶段的递推关系，这个递推式叫做动态转移方程。动态转移方程一般含有 max{}或 min{}。
决策—即对方法的选择。每个阶段都有一个决策。这样的选择是有范围的，这个范围叫做“决策允许集合”。
策略—一套完整的决策的组合。“最优策略”即最佳的决策组成的策略。
还有一些概念因为使用较少，这里不再详细介绍。
注意可以运用动态规划解决的问题的两个必需特性：
最优化原理—简单地说，最优策略某几个连续阶段上的决策组合，也是这几个阶段组成的子问题的最优策略。
无后效性原则—某阶段以后的决策，与该阶段之前的具体决策无关，只与该阶段的状态有关。
注意，有些时候我们认为不满足这两点的问题，换个角度看又是满足的。这正是动态规划的难点。接下来我们就是要寻找合适的角度，找出满足这两个关系的算法。

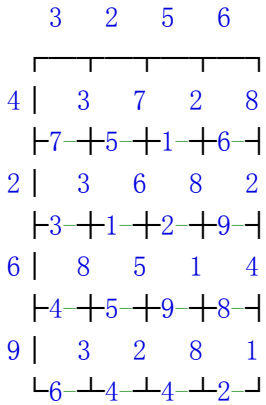
d.石子合并：状态的确定

用动态规划解决问题，首要也是最重要的步骤就是划分阶段、确定状态。这决定了是否能成功运用动态规划方法。因此。确定一个可行的递推思路是成功的关 键。

在这一过程中，要善于变通。
石子合并：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。
数据规模：N≤100，a[i]≤20000000
算法构造：
分析数据规模—算法可承受的最大数据规模 O(N^3)，搜索必然超时，贪心可承受数据规模太大，优先考虑动态规划。
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。
划分阶段—以合并的次数作为标准划分阶段。
确定状态—第 i 堆至第 j 堆合并所能获得的最大价值。
状态转移方程—f(i,j)=max{f(i,k)+f(k+1,j)},0<i≤k<j≤n
边界状态—f(i,i)=a[i]
分析知时间复杂度为 O(n^3)，满足要求。
递推求出 f(1,n)即可。动态规划特点是"思路难，实现易"，这里不再给出源代码。
另外，NOIP2006出现了一道石子合并的衍生问题。
在 Mars 星球上，每个 Mars 人都随身佩带着一串能量项链。在项链上有 N 颗能量珠。能量珠是一颗有头标记与尾标记的珠子，这些标记对应着某个正整数。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。因为只有这样，通过吸盘（吸盘是 Mars 人吸收能量的一种器官）的作用，这两颗 珠子才能聚合成一颗珠子，同时释放出可以被吸盘吸收的能量。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（Mars 单位），新产生的珠子的头标记为 m，尾标记为 n。
需要时，Mars 人就用吸盘夹住相邻的两颗珠子，通过聚合得到能量，直到项链上只剩下一颗珠子为止。显然，不同的聚合顺序得到的总能量是不同的，请你设计 一个聚合顺序，使一串项链释放出的总能量最大。
建议读者自己思考解决，以熟悉动态规划的思维过程。在附件中给出解法的源代码。

e.街道问题：状态量维数的确定与无后效性

下面是动态规划的经典问题之一：街道问题。
下图表示一个街道。现有类似这样的 N*N 的街道，且每段路有一个权值。现在要从左上角沿线走到右下角，每次只能向右或向下走。求经过路段权值和的最大值。
数据规模：N≤1000



分析数据规模—算法可承受的最大数据规模 O(N^2)，搜索必然超时，贪心可承受数据规模太大，优先考虑动态规划。
递推思路—从左上角向右下角推，求出走到每一点所经过权值和的最大值。
划分阶段—按斜向（/）划分

剩下的读者自己思考。时间复杂度应为O(n^2)。
上面问题中，若是改为从左上角沿线走到右下角再以不交叉的线路返回到出发点，算法应该做怎样的修改？
原来的状态已经不能保证"不交叉"，在这种状况下，我们用增加一维状态量的方法，以讨论是否交叉，比较方便的思路——从左上角出发两条路，再向右下方递推的过程中，两条路的最末端始终不能是同一点，必须分布在这一阶段的两点上，直到最后在右下角汇合。
由于增加了一维状态量，时间复杂度变为O(n^3)。

f.0-1背包：巧妙地选取状态量

有时，选定状态时发现不能使用动态规划，或问题从表面上看让人无所适从，我们不妨从另一个角度看问题。
上一章我们介绍了部分背包问题，下面我们介绍更经典的0-1背包问题。
问题描述：有N件物品和一个最大载重为M的背包，每件物品都有相应的重量和价值。现要求给出一个存放方案，使背包中物品总价值最大。所涉及到的数字均为整 数。
数据规模：1≤N≤100，1≤M≤10000。
这个问题与部分背包问题的区别在于，物品只能整件放入，不能只取一部分。这样，背包很可能会有部分载重量未利用。若用贪心法取单位重量价值高的，可能会使 背包未利用的载重升高，得不到最优方案。
考虑到数据规模，选用动态规划法。
似乎只有"每考虑一件物品为一阶段"的划分方法。可是如何确定状态？很多人想不到这个方法：以一个最大载重值为一个状态。
完整递推思路——计算将考虑了前i个物品后，最大载重为j的背包能取得的最大价值f(i,j)，这个过程是可由f(i-1,j)和f(i-1,j- g[i])递推而得。最后求出f(N,M)。
动态转移方程——f(i,j)=max{f(i-1,j),f(i-1,j-g[i])+v[i]},1≤i≤N,1≤j≤M（实现时注意对边界情形的处 理），其中g[i]、v[i]分别表示第i件物品的重量、价值。
时间复杂度为O(N*M)，可以接受。
实现时要注意空间复杂度（N*M的数组显然是开不出来的）。
动态规划的空间消耗是很大的，往往比深度优先搜索大得多。
动态规划的算法，往往空间复杂度比时间复杂度少一维。
下面是常见的一种降低储存的方法（在第二章提到过）——
动态规划实现时往往可表示成矩阵形式。若递推矩阵中每一行的值只与上一行有关，输出只和最末行有关，则可将奇数行储存在第一行，偶数行储存在第二行，降低 空间复杂度。
这里还有一种优化方法2——
若递推矩阵中每一行中某一列的值只与上一行中这一列及位于其右的列的有关，输出只和最末行有关，则可按从左到右的顺序计算该行每一列的值，并覆盖掉原值即 可。
上面的0-1背包问题，实现时，面临的情况与"优化方法2"中的类似但又有区别——每一行中某一列的值只与上一行中这一列及位于其左的列的有关。这 样，我们需要按从右到左的顺序计算。空间复杂度O(M)。
附件中给出0-1背包问题的源代码。
背包问题还有两种。
一种是完全背包问题：在0-1背包的基础上，每种物品可放入很多个。这个问题也用动态规划法解决。你会发现，这个问题实现时只与0-1背包差一点点——它 按从左到右的顺序计算，而0-1背包按从右到左的顺序计算。
另一种有时也称作完全背包问题：在0-1背包的基础上，要求背包恰好达到载重。这个问题留给读者思考。

g.Bitonic 旅行：最佳的状态转化方式

历史上有一个著名的"货郎担问题"（也叫"中国邮路问题"）：已知 n 个点两两间的距离，给出过这些定点的最短闭合回路。（有时把这 n 个定点间距离定 义为欧氏几何平面上距离，称"欧几里德旅行商问题"。）

这个问题是 NP-难问题，只能用搜索解决。

后来，J. L. Bentley 提出了这个问题的变形──Bitonic Tour 问题（又称双调旅程问题）。

Bitonic 旅行：已知地图上 n 个旅行须到达的城市，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。

数据规模：1≤n≤1000

你可能会这么想：按旅行顺序每过一个城市为一阶段。可是这样，状态量需要记录由东向西每一步所经过的城市，时间复杂度 O(2^n)，与搜索无异！

我们需要换一种状态转化方式。

递推思路──从最东端开始，找两条到最西端的路径。将地点按从东到西编号，每加入一个地点为一个阶段。计算从地点 i 到最东再到地点 j 路程的最小值 f(i,j)，这个过程是递推的。最后求出 min{f(n,k)+dist[n,k]}。

动态转移方程──

f(i,j)=f(i-1,j)+dist[i,i-1],1≤j≤i-2,i≤n

f(i,j)=f(i,j-1)+dist[j,j-1],1≤j=i-1,i≤n

（实现时注意对边界情形的处理），其中 dist[i,j]表示 i 与 j 间距离。

时间复杂度为 O(n^2)，很好。附件中给出源代码。

h.最长非降子序列模型

NOIP 命题者还是比较仁慈的，考察动态规划时降低了难度，大多考察"模仿"，很少"构造"──NOIP 的动态规划题，很多都是在经典问题基础上略 作变化而成。下面这个问题是我接触到的变形题最泛滥的经典问题。

最长非降子序列：给定一串数，从中删去一些数，使剩下的序列是非降的。求剩下的序列的最大长度。

数据规模：1≤n≤1000

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。

动态转移方程──f(i)=max{f(j)+1},n[j]≤n[i],1≤j<i(实现时注意对边界情形的处理),其中 n[i]表示序列中第 i 个数的值。

时间复杂度为 O(n^2)。

说明：实际上最长非降子序列的最佳解法是二分法（但 NOIP 要求的数据规模较小，往往可以用动态规划解）。最长非降子序列的二分法会在第七章介绍。

看看下面这两个问题，你是不是觉得一下子就有了思路？

合唱队形（NOIP2004）──N 位同学站成一排，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。合唱队形是指这样的一种 队形：设 K 位同学从左到右依次编号为1，2…，K，他们的身高分别为 T1，T2，…，TK，则他们的身高满足 T1<...Ti+1>…>TK(1<=i& lt;=K)。你的任务是，已知所有 N 位同学的身高，计算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。

2≤N≤100，130≤Ti≤230。

船（摘自《奥赛经典》）──PALMIA 国家被一条河流分成南北两岸，南北两岸上各有 N 个村庄。北岸的每一个村庄有一个唯一的朋友在南岸，且他们的朋友村 庄彼此不同。每一对朋友村庄想要一条船来连接他们，他们向政府提出申请以获得批准。由于河面上常常有雾，政府决定禁止船只航线相交（如果相交，则很可能导 致碰船）。你的任务是编写一个程序，帮助政府官员决定批准哪些船只航线，使得不相交的航线数目最大。1≤N≤5000。

对于"合唱队形"，我们只需讨论第 i 人作最高点时，左边升、右边降序列长度最大值。时间复杂度为

O(n^3)。
对于"船"，我们记 a[i]为北岸第 i 村对应的南岸村庄的位置，这个问题就变成了求 a[i]最长非降子序列的问题。这里最好使用二分法解答。(请读者好好 想想这是为什么。)
可见，一个问题可以衍生出很多类似的问题。要抓好经典问题的解决策略，这在 NOIP 中大有用处。

i.构造动态规划算法

不过，少数时候经典算法是帮不上忙的，我们自己要依据动态规划原理，动脑子构造算法。
有一些小技巧：
1.抓住"递推"这一本质，理清思路。
2.如果你对自己的数学相当有自信，你可以尝试数学推导动态转移方程。
3.有时，构造需要一些反常规的思维，可以试着摆脱常规思路的束缚，或者借鉴一下经典算法是如何打破思维局限的。
4.分析数据规模，如果有更简单的满足数据规模要求的算法，不必强求动归；适用动归的数据规模一般是30至1000。

j.动态规划、递推、广度优先搜索的区别与转化

动态规划是递推思想的典型应用。广度优先搜索有时也会用到递推思想。
有时，我们会用递推的方法来分析问题、尝试归纳问题。这通常会把我们引向动态规划或广度优先搜索。我们往往并不需要在开始时决定使用动态规划还是广度优先 搜索，而是先理清递推思路。
更深入地说，动态规划可看作是对图的一种遍历；在这种遍历中，后扩展的节点的当前值不会影响先扩展的节点的当前值。这就是它与广度优先搜索的区别所在—— 广度优先搜索中，后扩展的节点的当前值会影响先扩展的节点的当前值。
动态规划越来越被 NOIP 提高组命题者青睐，成为每年必考点。动归思考难度大，而实现较容易，这需要我们多看经典问题，积累经验。

习题

动 态规划的习题 ——感谢 深蓝评测系统 提供习题

6.常用数学方法

a.排列组合

在解决问题时，我们常常使用一些比较固定的"工具式"的算法，将它们作为算法的一部分。这样的工具常常包括排序、高精度运算、数学方法、二分查找等 等。
其中的一些在前面已经介绍，接下来我们还会介绍一些。

NOIP 要求我们会运用一些数学方法。这些方法往往是离散数学中的经典方法、定理。现在我们介绍一些。
数学方法包括求素数（用不大于 sqrt(n)的正整数除一下），求最大公约数（欧几里得算法，第一章讲过），排列组合等。
最基本的方法就是排列组合。高中数学已经介绍了排列组合，最有用的结论如下：
* n 个球中选出 m 个有序地排成一列，排法种数为 A(n,m)，在这里1≤m≤n。
A(n,m)=n!/(n-m)!=n(n-1)(n-2)…(n- m+1)
* n 个球中选出 m 个无序地分成一组，分法种数为 C(n,m)，在这里1≤m≤n。
C(n,m)=n!/[(n-m)!m!]=A(n,m)/m!
* 组合数性质：C(n,m)=C(n-1,m)+C(n-1,m-1)
* 插空法：n 个相同的 A 球和 m 个相同的 B 球排成一行（m≤n+1），要求 B 球两两不相邻，排法种数为 C(n+1,m)
* 杨辉三角的组合数形式：
1
1 1
1 2 1
1 3 3 1
…
上面的杨辉三角可写成如下形式
1
C(1,0) C(1,1)
C(2,0) C(2,1) C(2,2)
C(3,0) C(3,1) C(3,2) C(3,3)
…
* 球盒问题和第二类 Stirling 数
有一类问题可以用"n 个球放入 m 个盒子的放法种数"来描述。这里讨论盒子均不留空的情况，盒子可留空时的情形可据此推导。
相同球放入不同盒子的放法种数为 C(n+1,m)（运用插空法）。
不同球放入相同盒子的放法种数 L(n,m)=L(n-1,m-1)+m*L(n-1,m),L(n,1)=L(n,n)=1（L(i,j),i≥j 组成的 一系列数称作第二类 Stirling 数）。
相同球放入不同盒子的放法种数为 m!L(n,m)。
* Catalan 数
h(1)＝1,h(n)=h(1)h(n-1)+h(2)h(n-2)+…+h(n-1)h(1)的通项为 h(n)=C(2n-2,n-1)/n。
注意以下的结论——
n 个数连乘，根据结合律，用括号把它们分成两个因子相乘、不可再添加括号的形式，如 abcde 写成((ab)((cd)e))。这样的写法种数为 h(n)。
一个足够大的栈入栈序列为1,2,…,n，可能的出栈序列有 h(n)种。
将凸 n 边形区域划分成三角形区域的方法数为 h(n)。

在分析问题时，注意看清问题的本质，把实际问题抽象成数学模型，看看有没有数学方法可用。

b.递推与通项的选用

先来看一个问题——2^k 进制数（NOIP2006）。
设 r 是个2^k 进制数，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那一位。
（3）将 r 转换为2进制数 q 后，则 q 的总位数不超过 w。
在这里，正整数 k（1≤k≤9）和 w（k<W≤30000）是事先给定的。问：满足上述条件的不同的 r 共有多少个？
分析——
如果你的基础知识过关，那么你应该可以从（3）中知道，q 的总位数是 r 的总位数的 k 倍。（如果你没反应过来，请翻翻以前学习进制转换时的资料吧。）
现在命 c=[w/k]（[]是取整数部分函数），即 r 的位数上限是 c。当然，若 c<w/k，会有首位数字比较小的情形，这可在之后分开讨论。
下面仔细分析 c=w/k 时的情况。
现在，把 r 看成由 i 个数字组成（2≤i≤c）。i 个数字可从1至2^k-1任取并进行组合，每种组合方式对应一个数，即有 C(2^k-1,i)个数。
那么，r 共有 n=C(2^k-1,2)+C(2^k-1,3)+…+C(2^k-1,c)个。
c<w/k，读者自己想想吧～这里给出源代码。
现在的问题是计算。显然，高精度运算是不可避免的。这里还要算 C(2^k-1,i)。
如果用公式 C(n,m)=n!/[(n-m)!m!]，往往会有高精度除法，很难办。这里的好办法是——递推，用 C(n,m)=C(n- 1,m)+C(n-1,m-1)吧！
高精度加法很容易实现，虽然要做的加法比较多，但 CPU 算加法比乘法快得多，所以总时间比使用通项 C(n,m)=n!/[(n-m)!m!]还可能短一 些！
事实上，执行时间惊人的短。这里给出源代码。

```
// C Program "Digital" written by LastLeaf

#include <stdio.h>

#define H_Bit 51

int times(m,n)
{
    int i,a=1;
    for(i=1;i<=n;i++)a*=m;
    return a;
}

unsigned *HPrint(unsigned *N)
{
    FILE *fp;
    int i;

    fp=fopen("digital.out","w");

    for(i=H_Bit-N[0]+1;i<=H_Bit;i++)
```

```
        if(N[i]!=0)
        {
            fprintf(fp,"%u",N[i]);
            break;
        }
    for(i++;i<=H_Bit;i++)
        fprintf(fp,"%u%u%u%u",N[i]/1000,N[i]/100%10,N[i]/10%10,N[i]%10);
    fprintf(fp,"\n");

    fclose(fp);

    return N;
}

unsigned *HInitialize(unsigned *N,int value)
{
    N[0]=1;
    N[H_Bit]=value;
    return N;
}

unsigned *HAdd(unsigned *N1,unsigned *N2)
{
    int up=0,i;

    if(N2[0]>N1[0])
    {
        for(i=N1[0];i<N2[0];i++)N1[i]=0;
        N1[0]=N2[0];
    }
    for(i=0;i<N2[0];i++)
    {
        N1[H_Bit-i]+=N2[H_Bit-i];
        if(N1[H_Bit-i]>=10000)N1[H_Bit-i]-=10000,up=1;
        else up=0;
    }
    for(i=H_Bit-i;i>H_Bit-N1[0] && up==1;i--)
    {
        N1[i]++;
        if(N1[i]==10000)N1[i]=0;
        else up=0;
    }
    if(i!=0 && up==1)
    {
        N1[0]--;
        N1[i]=1;
    }
```

```
    return N1;
}

int main()
{
    int i,k,c,j,imax,hmax,bmax,w;
    unsigned A[513][H_Bit],Ans[H_Bit];
    FILE *fp;

    fp=fopen("digital.in","r");
    fscanf(fp,"%d %d",&k,&w);
    fclose(fp);

    bmax=times(2,k)-1;
    if(w%k==0)
        c=w/k,hmax=bmax;
    else
        c=w/k+1,hmax=times(2,w%k)-1;
    HInitialize(Ans,0);
    for(i=1;i<=bmax;i++)HInitialize(A[i],1);
    imax=bmax;
    for(j=2;j<=c;j++,imax--)
        for(i=2;i<=imax;i++)
            HAdd(A[i],A[i-1]);
    for(i=bmax-1;i>imax-hmax && i>0;i--)
        HAdd(Ans,A[i]);

    HPrint(Ans);

    return 0;
}
```

高中数学经常求通项，不过对于计算机，往往递推式更有用！去寻找递推式吧～

7.分治

a.子问题与母问题的相似性

在第三章"搜索"中我们讲过，要擅长利用相似性。现在我们介绍利用相似性的另一条思路：分治。

分治主要用来处理一维的数据结构。
分治，就是把母问题拆分为几个（一般是两个或三个）子问题，使每个子问题与母问题相似。这样就把大问题化成了相似的小问题，问题性质不变、规模成倍减小， 最后很容易解决。
由于利用了子问题的相似性，分治采用递归实现。
实现时递归函数要完成两步——"分"、"合"。
"分"是指分拆成小问题、递归调用，这一步比较容易。
"合"是指对分拆得到的小问题的结论进行合并处理，有时这一步很困难。
若"分"出的子问题有两个，又可称该方法为"二分法"。类似地，也有"三分法"之说。
在第二章讨论过快速排序和归并排序，这都是对分治的经典应用。
还有一些经典算法也运用了分治思想，常常被用作解题工具。下面介绍一下。

b.二分查找

有时我们要判断一串数组成的序列中是否存在某数字 N。
最普通的方法是，将 N 与该序列中每个数逐个比较。这种方法时间复杂度为 O(n)。
如果我们预先得知该序列是有序（非增或非降）的，我们可以采用二分查找法（以序列非降为例）：
1.首先，将该序列作为保留的序列。
2.将 N 与保留的序列中最中间的数 A 比较。
2.若 N

c.分析算式

有时会碰到这种问题：输入一个表示算式的字符串（一般含四则运算、乘方等运算符），要求输出计算结果。
对于这样的问题，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，先将其左右两边分别运算（递归地）、求值，再将得到的值进行 A 运算。
有一个类似的问题"删除多余括号"：输入一个表示算式的字符串（含四则运算、乘方、括号），要求尽可能多地删除其中多余的括号而不改变算式的运算过程和结 果。
我们用可以类似的方法解决。下面给出源代码，以便读者更好地理解这个递归地分析算式的过程。

```
// Program "Delete()" 删除多余括号
// This is a C source file written by LastLeaf

// 该程序已在gcc(DEV-C++)上编译通过

#include <stdio.h>
#include <string.h>

int a[1024];
char s[1024];

int cal(int st,int stp,int prev)
```

```
{
    int t,i,min=4,mini;

    for(i=st;i<=stp;i++)
    {
        switch(s[i])
        {
            case '^':
                if(min>3)
                {
                    min=3;
                    mini=i;
                }
                break;
            case '*': case '/':
                if(min>2)
                {
                    min=2;
                    mini=i;
                }
                break;
            case '+': case '-':
                if(min>1)
                {
                    min=1;
                    mini=i;
                }
                break;
            case '(':
                i++;
                for(t=1;t!=0;i++)
                {
                    if(s[i]=='(')t++;
                    if(s[i]==')')t--;
                }
                i--;
                break;
        }
    }

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

```
        }
        return 4;
    }

    if(min==4)
        return 4;
    cal(st,mini-1,min);
    if(s[mini]=='+' || s[mini]=='*')
        cal(mini+1,stp,min);
    else
        cal(mini+1,stp,min+1);

    return min;
}

int main()
{
    FILE *fp;
    int i,sc;

    fp=fopen("delete.in","r");
    fscanf(fp,"%s",s);
    fclose(fp);

    fp=fopen("delete.out","w");
    sc=strlen(s);
    cal(0,sc-1,0);

    for(i=0;i<sc;i++)
        if(!a[i])fprintf(fp,"%c",s[i]);
    fprintf(fp,"\n");
    fclose(fp);

    return 0;
}
```

NOIP2005中出现了"等价表达式"问题：明明进了中学之后，学到了代数表达式。有一天，他碰到一个很麻烦的选择题。这个题目的题干中首先给出 了一个代数表达式，然后列出了若干选项，每个选项也是一个代数表达式，题目的要求是判断选项中哪些代数表达式是和题干中的表达式等价的。

这个题目手算很麻烦，因为明明对计算机编程很感兴趣，所以他想是不是可以用计算机来解决这个问题。假设你是明明，能完成这个任务吗？

这个选择题中的每个表达式都满足下面的性质：

1. 表达式只可能包含一个变量'a'。
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'+'（加），'-'（减），'*'（乘），'^'（乘幂），以及小括号'('，')'。小括号的优先级最高，其次是'^'，然 后是'*'，最后是'+'和'-'。'+'和'-'的优先级是相同的。相同优先级的运算从左到右进行。（注意：运算符 '+'，'-'，'*'，'^'以及小括号'('，')'都是英文字符）

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。
下面是一些合理的表达式的例子：
((a^1) ^ 2)^3，a*a+a-a，((a+a))，9999+(a-a)*a，1 + (a -1)^3，1^10^9……
思路分析：对于这个问题，聪明的做法是，给未知数设一个值，然后计算每个代数式的结果（分治法），看看与原式是否匹配。不匹配即被排除。这里可以不使用高 精度运算，这可以理解为：我们只计算高精度数的最后一位。若最后一位都不能匹配，就肯定要排除。我们多代几个比较随机的数之后，便基本上可以把干扰项排除 完（除非一个极小概率事件发生了）。剩下的应该就是正确答案。

d.最长非降子序列的二分法

介绍"动态规划"时我们接触过最长非降子序列问题。这里有一个更好的解决方案，它基于贪心和分治：
1.另设置序列A[]，A[i]表示若非降子序列长度为i，其最后一项可能的最小数值。
2.一次考虑原序列中每一项。对于某项的数值n，若A[i]中存在数值大于n的项，则用二分查找的方法找出A[i]中数值大于n的项中最前的一项 A[m]，令A[m]=n；否则在A[]末尾添加一项，值为n。
3.最后，最长非降子序列长度就是A[]的项数，A[]就是最长非降子序列之一。
有很多经典问题用分治解决。但是分治在NOIP提高组中出现不多。读者若想多了解一些，请参阅下方提供的资料链接。

8.图论思想

a.图论基础

我认为图论问题应当是 NOIP 中最难的问题，因为命题者往往把图论题设计得很繁琐，提高了编写和调试的难度。当然，图论题也有简单的，而 且，NOIP 提高组并不是每年都会涉及到图论题。这里，我们的介绍也就不那么详尽了。

首先要弄清楚两个问题：什么是图？什么是图论？
图论中所指的图是由若干点和两点之间的线（称作"边"）构成的。图论是数学的重要分支，主要研究图的性质。
要注意的是，图论只研究两点间的关联，不关心位置、长度、距离等。也就是说，移动图上任何点的位置，或者把边画得弯弯曲曲，仍然未改变这个图。只有为两点 之间增减点或边、改变边的方向或权值才使图发生改变。
这样，我们可以把图记作 G=(V,E)，V 是顶点集，E 是边集。
一些图中，边是有方向的，这样的图叫有向图。
对于无向图和有向图，边都可以有权值，有时权值还可为负。
两点间有一条边连接，称这两点"相邻"。
任何两点间都有路径连通（即可从一点沿边到达另一点），则称这个图是连通图。
树是一种特殊的连通图。一个连通图是树的充要条件是边数=顶点数-1；一个图是
从一点出发可沿一条路径回到出发点，这条路径称为"回路"。

b.图的表示方法

图有很多中表示方法。把图上的点都标上序号，就可以用以下三种方法来表示图。
要注意正确选择表示方法，因为不同方法的时间复杂度、实现困难程度很可能有区别。
输入时常常采用邻接表。邻接表是这样的一个表：每列表示一条边；有两到三行，含义见下图。（NOIP 输入时往往采用"每行表示一条边"的形式。）

```
1 2 2 ——有向图的边的起点，无向图的边的一点
3 3 4 ——有向图的边的终点，无向图的边的另一点
3 5 9 ——边的权值
```

处理数据时比较方便的方法是邻接矩阵。邻接矩阵是一个二维表：x 行 y 列表示顶点 x 到 y 的边(x,y)的权值（无边则记为"0"或"∞"，无权值图的 边记为"1"）。

```
- 1 2 3 4
1 - 5 0 9
2 5 - 8 3
3 0 8 - 0
4 9 3 0 -
```

（对于无向图，(x,y)=(y,x)）
有时，表示图的最佳方式是链表。链表的每一个节点包含几个部分：所代表的顶点序号，它相邻顶点的指针、对应权值的列表。

c.经典图论算法

解决图论问题，我们需要一些基本的工具。这些就是图论的经典算法。
最小生成树问题：Prim 算法。
问题描述——找出一个无向有权值连通图 G=(V,E)的连通子图 G'=(V,E')，使 G'是树且边的权值和最小。

算法——
1.选择一个顶点作起点，记为“已到达”，其他顶点“未到达”。先置 E′为空。
2.找出所有连接一个“已到达”顶点和一个“未到达”顶点的边，从这些边中找出权值最小的边(i,j)。(i,j)加入 E′。将(i,j)所连接的“未到 达”顶点标记为“已到达”。
3.重复2，直到所有顶点均“已到达”。此时 G′=(V,E′)为所求的连通子图。
算法实现时有很多要注意的问题，要恰当处理使时间复杂度为 O(v^2)（v 表示顶点数）。
如果用邻接矩阵处理，可设数组记录“未到达”顶点 i 与所有“已到达”顶点的边中权值最小的边(i,j)，每标记一个“已到达”顶点就调整一次该数组。这样 做时间复杂度为 O(v^2)。
另外，Kruskal 算法也用来解决最小生成树问题，时间复杂度 O(ve)（e 表示边数）。其思路是——
按边的权值由小到大考虑每条边：若选取这条边后这条边不会与其他边构成回路，则选取这条边；否则抛弃这条边。最后已选取的 v-1条边组成最小生成树。
具体实现请读者思考。
单源点最短路径问题：Dijkstra 算法。
问题描述——找出一个有向有权值（非负）连通图 G=(V,E)中某顶点 a 分别到其余的顶点的最短路径长度。
算法——
Dijkstra 算法是一个贪心算法，时间复杂度为 O(n^2)。读者可以想想为什么它是正确的。
这个算法用邻接矩阵实现，开始时，邻接矩阵中没有边的位置记为∞。
1.将 a 记为“已到达”，其他顶点“未到达”。开辟数组 dist[]，用于储存 a 到其余各顶点的最短路径长度。
2.开辟数组 cur[]，cur[i]表示当前“未到达”顶点 i 与 a 距离（如果 i 已到达，这个值是无意义的，记作∞）。
3.选取 cur[]中最小的一项及其对应的顶点 n。此时可得 dist[n]=cur[n]。将 n 记为“已到达”，将 cur[]中的该项记为∞。
4.对每个“未到达”顶点 i，判断路径(a,n,i)的距离是否小于当前距离，即比较 dist[n]+(n,i)和 cur[i]，cur[i]取较小值。
5.重复3、4，直到全部顶点“已到达”。
另外，还有一种类似的问题，叫做“各顶点对最短路径问题”。我们同样可以用 Dijkstra 算法解决，但也可以用更为方便的 Floyed 算法解决。至于 Floyed 算法的具体步骤，请读者自行查阅。
AOV 网问题：拓扑排序算法。
问题描述——AOV 网（Active On Vertex network）是一种有向无权值图，它可以表示各顶点所代表的事件的承接关系。现在要给出一种可能的事件发生顺序，满足：
对于每条边，终点所代表事件的发生必须以起点所代表的事件已发生为前提；换句话说，若想到达某个顶点 i，则 i 的所有前趋顶点必须都到达过。
算法——
1.找出所有无前趋的且未到达顶点，依次到达这些顶点并删除以它们中的某个为起点的所有边。
2.重复1，直到找不出任何符合要求的顶点。此时若所有顶点已到达，则先前到达顶点的顺序就是拓扑排序的一种结果；若还有顶点未到达，则说明存在 AOV 网 存在回路，不能拓扑排序。
拓扑排序实际应用很广。比如在课程安排上，某些课程的学习必须建立在其他课程已学过的基础上，而某些课程之间先后顺序却可以调换。这样，通过拓扑排序，就 可以得到几种合适的课程安排顺序。
AOE 网问题：关键路径算法。
AOE 网（Active On Edge network）是一种有向有权值图，一般情况下，它有一个源点和一个结束点。
有关 AOE 网的问题往往是实际生产生活中遇到的问题，其关键是求“关键路径”。这条关键路径往往是源点与结束点间权值和最大或最小的路径。由这条路径我们 往往很容易推出很多结论。
关键路径的求法分两步：
1.拓扑排序。如果失败，则说明关键路径不存在；如果成功，也就为下一步划好了阶段。

2.动态规划。
具体解决办法读者可在实际问题中自己摸索。附件中给出一个求关键路径的源程序。
还有一些经典图论算法（如网络流算法）由于过于生僻或繁琐，这里不再介绍。这些在 NOIP 中的实际应用也极少。

d.构造图论模型

其实图论思想中价值最高的不是经典算法，而是图论建模。
下面介绍几个经典的图论模型应用。
构造回路——
篝火晚会问题（NOIP2005）：
佳佳刚进高中，在军训的时候，由于佳佳吃苦耐劳，很快得到了教官的赏识，成为了"小教官"。在军训结束的那天晚上，佳佳被命令组织同学们进行篝火晚会。一 共有 n 个同学，编号从1到 n。一开始，同学们按照1，2，……，n 的顺序坐成一圈，而实际上每个人都有两个最希望相邻的同学。如何下命令调整同学的次序， 形成新的一个圈，使之符合同学们的意愿，成为摆在佳佳面前的一大难题。
佳佳可向同学们下达命令，每一个命令的形式如下：
(b1, b2,… bm -1, bm)
这里 m 的值是由佳佳决定的，每次命令 m 的值都可以不同。这个命令的作用是移动编号是 b1，b2，…… bm -1，bm 的这 m 个同学的位置。要求 b1换到 b2的位置上，b2换到 b3的位置上，……，要求 bm 换到 b1的位置上。
执行每个命令都需要一些代价。我们假定如果一个命令要移动 m 个人的位置，那么这个命令的代价就是 m。我们需要佳佳用最少的总代价实现同学们的意愿，你能帮 助佳佳吗？
输出最小的总代价。如果无论怎么调整都不能符合每个同学的愿望，则输出-1。
思路：
把每个人和他最喜欢相邻的人连起来，如果能形成一个大圈（回路），这个大圈就是我们所要移动到的最终目标；如果形不成一个大圈，肯定就不能同时满足所有人 的愿望，输出-1即可。
下面构造一个有向图：每个人作为一个顶点，从这个顶点仅出发一条边，指向这个人最终应该到的位置。
最终目标之一：1 2 3 4 5 6 初始：2 3 1 5 4 6 图：2→3→1 5→4 6

注意：最终目标也可以是 2 3 4 5 1，5 4 3 2 1……
这样，图上会形成若干回路（有时有的顶点不在回路中），一个命令对应一个回路，恰使这个回路上的所有人归位。
可是哪个最终目标需要的总代价最少？一个命令的价值对应一个回路的顶点数，命令的总价值对应所有回路顶点数之和。总代价最小，就意味着回路里的人最少，在 原位不需移动的人最多！
我不想再说下去了，剩下的已经很明白了。如果你还不懂，只好——下载附件，看源代码吧！
构造路径、二部图——
狼羊菜问题：一位渔民带了狼、羊、菜，准备过河。可是小船每次只能容下渔民和一件物品。渔民不在时狼会吃羊、羊会吃菜。请设计一种最佳渡河方案。
有的人可能会笑我——三岁小孩的智力题放在这里做什么？
那你看看这一个变形：三个白人、三个黑人叫来你的小船，准备渡河。小船每次只能容下你和另两个人。要注意，你不在时某个岸边若黑人比白人多或白人比黑人 多，白人和黑人就会打架。请给出所有安全的渡河方法。
这个问题就很复杂了。或许你绞尽脑汁可以得到一个解，可是怎么证明解的个数呢？
还是回到狼羊菜问题。我们考虑某一时刻的所有可能情形：（表示为河的出发岸/对岸）
狼羊菜人/┐ /狼羊菜人
狼羊人/菜 | 菜/狼羊人
狼菜人/羊 | 羊/狼菜人
羊菜人/狼 | 狼/羊菜人

羊人/狼菜　┗狼菜/羊人
有些状态可以相互转化，我们用边连接起来（图中连了其中一条）。连完后成了一个无向图。
题目要求找出"狼羊菜人/"到"/狼羊菜人"的路径，也就变得很容易了吧？狼羊菜的变形问题当然也可以用类似的方法解决（如果我没记错，上面给出的变形问 题是有四个解的）。
另外，像上面这样的图叫做"二部图"（有的地方称"二分图"）。它的特点是图顶点可以被分为两组，每组内部的顶点两两间均无边相连。
一个无向图是二部图的充要条件是图中无奇数边组成的回路。
还有一些很经典的图论建模问题，这里不再详述。你可以看看下面"未提及的经典问题"一栏。
终于完成这个系列了～如果你还向更深入地学习，请参考 Programet 的其他相关文章，谢谢你的支持～

习题

图 论的习题 —感谢 深蓝评测系统 提供习题

附件：关键路径算法、篝火晚会问题解法源文件