第 2 章　枚举

应用计算机求解实际问题往往都是从枚举设计起步的。在当今计算机的运算速度非常快的背景下，应用枚举设计可简明而快捷地解决一般数量规模的许多实际应用问题。

本章介绍统计求和、整数搜索、解方程与不等式等基础案例的枚举求解，并应用枚举设计求解诸如数式、数组、数列与数阵等许多丰富有趣的案例。

2.1　枚举概要

1. 枚举的概念

枚举法（Enumerate）也称为列举法、穷举法。枚举是蛮力策略的具体体现，又称为蛮力法。枚举法是一种简单而直接地解决问题的方法，其基本思想是逐一列举问题的所有情形，并根据问题提出的条件逐一检验哪些是问题的解。

枚举法常用于解决“是否存在”或“有多少种可能”等问题。其中许多实际应用问题靠人工推算求解是不可想象的，而应用枚举设计可以充分发挥计算机运算速度快、擅长重复操作的特点，简洁明了。

枚举法的特点是算法设计比较简单，只要一一列举问题所涉及的所有情形即可。应用枚举时，应注意对问题所涉及的有限种情形进行一一列举，既不能重复，又不能遗漏。重复列举浪费时间，还可引发增解，影响解个数的准确性；而列举的遗漏可直接导致问题解的遗漏。

2. 枚举模式

实施枚举通常应用循环结构来实现，常用的枚举模式有以下两个。

（1）区间枚举

对于有明确范围要求的实际案例，通过枚举循环的上下限控制枚举区间；在循环体中完成各个运算操作，然后根据所求解的具体条件，应用选择结构实施判别与筛选，求得所要求的解。

区间枚举设计的框架描述：

```
n=0;
for(k=<区间下限>;k<=<区间上限>;k++)      // 根据实际控制枚举范围
   { <运算操作序列>;
     if(<约束条件>)                     // 根据约束条件实施筛选
     { printf(<满足要求的解>);          // 逐一输出问题的解
       n++;                             // 统计解的个数
     }
   }
printf(<解的个数>);                     // 输出解的个数
```

（2）递增枚举

有些问题没有明确的范围限制，可根据问题的具体实际，试探性地从某一起点开始增值枚举，对每一个数进行操作与判别，若满足条件即输出结果。

递增枚举设计的框架描述：

```
k=0;
while(1)                                    // 设置递增循环
   { k++;                                   // 枚举变量 k 递增
     <运算操作序列>;
     if(<约束条件>)                         // 根据约束条件实施筛选与结束
     { printf(<满足要求的解>);              // 输出问题的解
       return;                              // 返回结束
     }
   }
```

递增枚举往往得到一个解后即结束。

尽管枚举比较简单，在应用枚举设计求解实际问题时要认真分析，准确设置枚举循环，并确定约束与筛选条件。

3. 枚举的实施步骤

应用枚举设计通常分以下几个步骤：

（1）确定枚举策略，根据枚举路线设置枚举量（简单变量或数组）；

（2）根据问题的具体范围，设计枚举循环；

（3）根据问题的具体要求，确定筛选（约束）条件；

（4）设计枚举程序并运行、调试，对运行结果进行分析与讨论。

当问题所涉及数量规模较大时，枚举的工作量也就相应较大，程序运行时间也就相应较长。为此，应用枚举求解时，应根据问题的具体实际进行分析归纳、寻求简化规律、优化枚举策略、精简枚举循环、降低枚举复杂度。

4. 枚举设计的意义

虽然巧妙和高效的算法很少来自枚举，但枚举法作为一种常规的基础算法，不能受到冷漠与忽视，更不能被认为无关紧要、可有可无。

（1）理论上，枚举可以解决计算领域中的各种问题。尤其处在计算机运算速度非常高的今天，枚举设计的应用领域是非常广阔的。

（2）在实际应用中，如果要解决的问题规模不大，应用枚举设计求解的速度是可以接受的。此时，设计一个更高效率的算法在代价上不值得。

（3）枚举可作为某类问题时间性能的底限，用来衡量同样问题的高效率算法。

本章将通过若干典型案例的求解，说明枚举设计的实际应用。

2.2　统计求和

统计计数与求和求积是计算机应用的基础课题，通常只要合理设计枚举循环即可简捷地求解。

本节介绍同码小数求和与三角网格统计这两个有一定难度的案例枚举设计，以提高对求和设计技巧与运用枚举实施统计的掌握。

2.2.1　同码小数

整数部分为零、小数部分各位数字相同的小数称为同码小数，例如 0.3,0.33,0.333,…是

同数码 3 的小数，记这些小数的前 5 项之和 0.3+0.33+0.333+0.3333+0.33333 为 s(3,5)，记前 5 项的加权和 0.3+2×0.33+3×0.333+4×0.3333+5×0.33333 为 w(3,5)，一般地记：

s(d,n)= 0.d+0.dd+0.ddd+…+0.dd…d

w(d,n)= 0.d+2×0.dd+3×0.ddd+…+n×0.dd…d

两和式为 n 项之和，其中第 k 项小数点后有 k 个数字 d，加权和第 k 项的权系数为 k。

依次输入整数 d(1≤d≤9)和 n(1≤n＜10000)，计算并输出和 s(d,n)与 w(d,n)（四舍五入精确到小数点后 6 位）。

枚举设计要点如下：

（1）求和的精度并不高，只要设置双精实变量 s,w 实施累加即可。

（2）设置 j(1～n)循环枚举和式的每一项，设 s 的当前项为 t，其后一项显然为 t=t/10+0.1*d。

（3）加权和 w 的每一项在 t 的基础上乘加权系数 j，即累加 t*j。

1. 枚举设计描述

```
// 求 s(d,n)与 w(d,n)
main()
{ int d,j,n; double t,s,w;
  printf("   请输入整数 d,n: ");
  scanf("%d,%d",&d,&n);
  t=s=w=0;                                  // t、s、w 清零
  for(j=1;j<=n;j++)
    {  t=t/10+0.1*d;                        // t 为 s 的第 j 项
       s+=t;                                // 求和 s
       w+=t*j;                              // 求加权和 w
    }
  printf("  s(%d,%d)=%.6f\n",d,n,s);        // 输出和 s
  printf("  w(%d,%d)=%.6f\n",d,n,w);        // 输出加权和 w
}
```

2. 算法测试与分析

```
请输入整数 d,n: 7,2014
s(7,2014)=1566.358025
w(7,2014)=1578192.681756
```

注意：在算法描述中省略了有关对 C 头文件的调用（下同），上机测试时需按规定加上。输出的小数点后第 6 位数字是四舍五入的结果。

枚举通过一重循环实现，时间复杂度为 O(n)。

3. 问题引申

对于给定的同码小数的和

s(d,n)= 0.d+0.dd+0.ddd+…+0.dd…d

w(d,n)= 0.d+2×0.dd+3×0.ddd+…+n×0.dd…d

输入正整数 d(1≤d≤9)，n(1≤n＜10000)，试精确求和 s(d,n)与 w(d,n)。同时统计：在 s(d,n)与 w(d,n)的 n 个小数位中，共有多少个小数位 s 与 w 对应位的数字相同？

（1）枚举设计要点

问题引申增加了求和的难度，同时增加了统计。准确求和是统计的基础，要求精确到所有 n 位小数。

设置一维数组 s，s[j]表示和的小数点后第 j 位，s[0]为和的整数部分；同理，设置一维数组 w，w[j]表示加权和的小数点后第 j 位，w[0]为加权和的整数部分。

1）对应位累加求和

注意到 s(d,n)小数点后第 n 位为 d，赋值给 s[n]；

小数点后第 n-1 位为 2*d，赋值给 s[n-1]；

……

小数点后第 1 位为 n*d，赋值给 s[1]。

而加权和 w(d,n)的小数点后第 n 位为 n*d，赋值给 w[n]；

小数点后第 n-1 位为((n-1)+n)*d，赋值给 w[n-1]；

……

小数点后第 1 位为(1+2+…+n)*d，赋值给 w[1].

循环中应用变量 t 累加权数。

2）从后向前进位

对应位累加完成后，从 s[n]开始逐位往前进位（j=n,n-1,…,1）：

```
s[j-1]=s[j-1]+s[j]/10;
s[j]=s[j]%10;
```

所得 s[0]即为所求和的整数部分。

加权和 w 进位类似。

3）按对应位比较 s 与 w 数字相同

设置比较 j（1～n）循环，若 s[j]=w[j]，即两个和对应位数字相同，用 m 统计位数。

4）输出和 s 与 w

设置输出 j（1～n）循环，从整数部分 s[0]、w[0]开始，依次输出和 s 与 w 的各位数字。

（2）枚举设计描述

```
// 求 s(d,n)与 w(d,n)，并比较和 s 与 w 共有多少个小数位对应数字相同
main()
{ int d,j,m,n; long t,s[10000],w[10000];
  printf("   请输入整数 d,n: ");
  scanf("%d,%d",&d,&n);
  for(j=0;j<=n;j++)
    { s[j]=0;w[j]=0;}                    // s,w 数组清零
  for(t=0,j=n;j>=1;j--)
    { t=t+j;                             // t 实施对权系数 j 累加
      s[j]=(n+1-j)*d;                    // 和 s 小数点后第 j 位共 n+1-j 个 d 之和
      w[j]=t*d;                          // 加权和 w 小数点后第 j 位共 t 个 d 之和
    }
 for(j=n;j>=1;j--)                       // 从后往前逐一进位
   { s[j-1]=s[j-1]+s[j]/10;
     s[j]=s[j]%10;
     w[j-1]=w[j-1]+w[j]/10;
     w[j]=w[j]%10;
   }
m=0;
for(j=1;j<=n;j++)                        // 比较相应小数位上 s 与 w 数字相同的位数
```

```
    if(s[j]==w[j]) m++;
printf("   s(%d,%d)=%ld.",d,n,s[0]);       // 输出和 s、w
for(j=1;j<=n;j++) printf("%ld",s[j]);
printf("\n   w(%d,%d)=%ld.",d,n,w[0]);
for(j=1;j<=n;j++) printf("%ld",w[j]);
printf("\n   和 s 与 w 共有%d 个小数位对应数字相同.\n",m);
}
```

（3）算法测试与复杂性分析

```
请输入整数 d,n: 8,50

s(8,50)=44.34567901234567901234567901234567901234567901234568

w(8,50)=1133.22359396433470507544581618655692729766803840877920
  和 s 与 w 共有 4 个小数位对应数字相同.
```

算法设计中，除数组清零外，设计有分位累加、进位、小数对应位比较与和的输出 4 个枚举循环。各个循环的数量级都是 n，可知算法的时间复杂度为 O(n)。

设计中有些循环当然可以合并，以上分开设计可使算法描述更为清晰。

变通：若要求具体求出 s 与 w 哪几个小数对应位数字相同，应如何处理？

2.2.2 三角网格

把一个正三角形的三边分成 n 等分，分别与各边平行连接各分点，得 n-三角网格。例如 n=6 时，6-三角网格如图 2-1 所示。

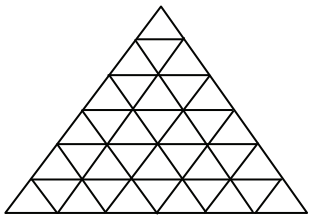

图 2-1 6-三角网格

对指定正整数 n，试求 n-三角网格中所有不同三角形（大小不同或方位不同）的个数，以及所有这些三角形的面积之和（约定网格中最小的单位三角形的面积为 1）。

输入整数 n(1<n<120)，输出 n-三角网格中不同三角形的个数，以及所有这些三角形的面积之和。

1. 统计求解要点

（1）设 n-三角网格中所含单位三角形数为 p(n),显然从最上层开始的第 1 层为 1 个，第 2 层 3 个，…，底层为 2n-1 个。因而有

$$p(n)=1+3+\cdots+(2*n-1)$$

一般地，设 k-三角网格中所含单位三角形数为 p(k),则

$$P(k)=1+3+\cdots+(2*k-1) \quad (k=1,2,3,\cdots,n)$$

计算出 p(1),p(2),…,p(n)，为后续计算面积和时使用。

（2）统计三角形数与面积和时，设三角形的水平边为底，顶角在上称为“正立”，顶角在下称为“倒立”。以下分“正立”与“倒立”两类分别统计求和。

设正立三角形的个数为 s1,其面积之和为 ss1。

正立三角形从大到小统计：

边为 n 的三角形 1 个，其面积为 p(n)；

边为 n-1 的三角形 1+2 个，每个面积为 p(n-1)；

……

边为 1 的三角形 1+2+…+n 个，每个面积为 p(1)。

s1=1+(1+2)+(1+2+3)+…+(1+2+…+n)

ss1=1*p(n)+(1+2)*p(n-1)+…+(1+2+…+n)*p(1)

（3）设倒立三角形的个数为 s2,其面积之和为 ss2。

倒立三角形从小到大统计：

边为 1 的三角形 1+2+…+(n-1)个，每个面积为 p(1)；

边为 2 的三角形 1+2+…+(n-3)个，每个面积为 p(2)；

……

① 当 n 为偶数时，边为 n/2 的三角形 1 个，每个面积为 p(n/2)；

s2=1+(1+2+3)+…+(1+2+…+(n-1))

ss2=1*p(n/2)+(1+2+3)*p(n/2-1)+…+(1+2+…+(n-1))*p(1)

② 当 n 为奇数时，边为(n-1)/2 的三角形 1+2 个，每个面积为 p((n-1)/2)；

s2=(1+2)+(1+2+3+4)+…+(1+2+…+(n-1))

ss2=(1+2)*p((n-1)/2)+(1+2+3+4)*p((n-1)/2)+…+(1+2+…+(n-1))*p(1)

（4）所求 n-三角网格中不同三角形的个数为 s1+s2，所有这些三角形的面积之和（即所含单位三角形的个数之和）为 ss1+ss2。

2. 枚举描述

```
// n-三角网格中的不同三角形个数及面积之和
main()
 { int k,m,n,u,p[1000];
   long t,t1,t2,s1,s2,ss1,ss2;
   printf("  请输入正整数 n: ");
   scanf("%d",&n);
   for(t=0,k=1;k<=n;k++)
      {t=t+(2*k-1);p[k]=t;}
   t1=t2=s1=s2=ss1=ss2=0;
   for(k=1;k<=n;k++)              // 求正立三角形个数及其面积之和
     { t1=t1+k;
       s1=s1+t1;ss1=ss1+t1*p[n+1-k];
     }
   m=(n%2==0?1:2);
   for(k=m;k<=n-1;k=k+2)          // 求倒立三角形个数及其面积之和
     { t2=t2+(k-1)+k;u=(n+1-k)/2;
       s2=s2+t2;ss2=ss2+t2*p[u];
     }
  printf("三角网格中共有三角形个数为：%ld \n",s1+s2);
```

```
  printf("三角网格中所有三角形面积之和为:%ld \n",ss1+ss2);
}
```

3. 算法测试与分析

```
请输入正整数 n:100
三角网格中共有三角形个数为:256275
三角网格中所有三角形面积之和为: 201941895
```

本题求解难点在于统计“倒立”三角形时，需对奇数与偶数两种情形分别总结规律并实施求和。

另外，p 数组的建立大大简化了求三角形面积之和的计算。

以上枚举设计的时间复杂度为 O(n)。

2.3 整数搜索

搜索某些特定的整数是枚举设计的基本应用课题，其中不乏有趣的经典难题。

2.3.1 整数对

设 b 是正整数 a 去掉一个数字后的正整数，对于给出的正整数 n，寻求满足和式 a+b=n 的所有正整数对 a,b。

例如，n=34，满足和式 a+b=n 的正整数对有 3 对：(27,7)、(31,3)、(32,2)。

输入正整数 n（n>10），输出满足要求的所有正整数对 a,b（若没有，则输出“0”）。

1. 设计要点

（1）根据给出的 n 设置整数 a 的枚举循环，对每一个 a，计算 b=n-a。

对于给定的 n，确定枚举 a 的取值范围必须慎重：范围太小，可能造成遗解；范围太大，造成无效操作太多。

注意到 b≥1（有可能取 1），因而取 a 循环终点为 n-1。

由 a＞b，a+b=n，可知 a＞n/2，因而取 n/2 为 a 循环起点。

（2）设计赋值表达式 d=a/(t*10)*t+a%t; 生成 a 的去数字数。

事实上，当 t=1 时，表达式为 d=a/10; d 即为 a 的去个位数字后的数。

当 t=10 时，表达式为 d=a/100*10+a%10; d 即为 a 的去十位数字后的数。

……

依此类推，设 a 是一个 m 位数，应用 t 可实现生成 a 的 m 个去数字数。这些去数字数逐个与 b=n-a 进行比较并决定取舍。

2. 枚举描述

```
// 整数对
main()
{ long a,b,d,n,t,k;
  printf("  请输入整数 n: "); scanf("%ld",&n);
  k=0;
  for(a=n/2;a<=n-1;a++)
  { b=n-a;t=1;
    while(a>t)                    // 应用 t 控制去数字循环次数
```

```
        { d=a/(t*10)*t+a%t;          // d 为 a 的一个去数字数
          if(d==b)
            { k++;
              printf(" (%ld,%ld)",a,b);
              break;
            }
          t=t*10;
        }
      }
  printf("\n    %ld 共有以上%ld 个解\n",n,k);
}
```

3. 算法测试与分析

```
 请输入整数 n: 2014
(1507,507)  (1827,187)  (1831,183)  (1832,182)  (1857,157)
(1907,107) (2007,7)
  2014 共有以上 7 个解
```

以上枚举设计中，去数字表达式的建立是巧妙的。枚举的时间复杂度为 O(n)。

2.3.2　基于 s 的双和数组

把一个偶数 2s 分解为 6 个互不相等的正整数 a,b,c,d,e,f，然后把这 6 个正整数分成(a,b,c)与(d,e,f)两个三元组，若这两组数具有以下两个相等特性：

$$a+b+c=d+e+f$$

$$\frac{1}{a}+\frac{1}{b}+\frac{1}{c}=\frac{1}{d}+\frac{1}{e}+\frac{1}{f}$$

则把数组(a,b,c)与(d,e,f)称为基于 s 的双和数组(约定 a<b<c,d<e<f,a<d)。

输入正整数 s（10＜s＜1000），搜索并输出基于 s 的所有双和数组。

1. 枚举设计要点

因 6 个不同正整数之和至少为 21，因而可知正整数 s>10。

（1）设置枚举 a,b 与 d,e 循环。

注意到 a+b+c=s，且 a＜b＜c，因而 a,b 循环取值为：

a: 1～(s-3)/3（因 b 比 a 至少大 1，c 比 a 至少大 2）；

b: a+1～(s-a-1)/2（因 c 比 b 至少大 1）；

c=s-a-b。

设置 d,e 循环时基本同上，只是注意到 d＞a，因而 d 起点为 a+1。

（2）比较倒数和相等。

为了比较倒数和相等

$$1/a+1/b+1/c=1/d+1/e+1/f \qquad (2.1)$$

把式（2.1）转化为整数式

$$d*e*f*(b*c+c*a+a*b)=a*b*c*(e*f+f*d+d*e) \qquad (2.2)$$

若上式不成立，即倒数和不相等，则返回。

（3）排除 6 个正整数是否存在相等

注意到两个和相等的三元组中，若部分数相同，部分数不同，则不可能有倒数和相等。

因而可省略排除以上 6 个正整数中是否存在相同整数的检测。

（4）输出双和数组解

在设置的枚举循环中，确保了两个三元组和相等。若式（2.2）成立，即倒数和也相等，满足双和相等条件，则打印输出基于 s 的双和数组，并用 x 统计解的个数。

2. 算法描述

```
// 双和数组探索
main()
 {double a,b,c,d,e,f,x,s;
  printf("  请输入 s: ");
  scanf("%lf",&s);
  x=0;
  for(a=1;a<=(s-3)/3;a++)                    // 设置枚举循环
  for(b=a+1;b<=(s-a-1)/2;b++)
  for(d=a+1;d<=(s-3)/3;d++)
  for(e=d+1;e<=(s-d-1)/2;e++)
    { c=s-a-b; f=s-d-e;                      // 确保两组和等于 s
      if(a*b*c*(e*f+f*d+d*e)!=d*e*f*(b*c+c*a+a*b))
         continue;                           // 排除倒数和不相等
      x++;
      printf("%.0f: (%.0f,%.0f,%.0f)  ",x,a,b,c);
      printf("(%.0f,%.0f,%.0f)\n",d,e,f);
    }
  if(x==0) printf("  无解! \n");
}
```

3. 算法测试与分析

```
s=26:
1: ( 4,10,12)  ( 5, 6,15)
```

若输入小于 26 的整数 s，则没有双和数组输出。可见，存在基于 s 的双和数组的整数 s 最小值为 26。

```
s=98:
1: ( 2,36,60)  ( 3, 5,90)
2: ( 7,28,63)  ( 8,18,72)
3: ( 7,35,56)  ( 8,20,70)
4: (10,33,55)  (12,20,66)
```

算法设置了关于 s 的 4 重循环，时间复杂度为 $O(n^4)$。因而当 s 比较大时，搜索基于 s 的双和数组是困难的。

变通：修改以上算法，实现基于 s 的和积（两个三元组和相等且积也相等）数组的搜索。

2.3.3 最小连续 m 个合数

素数是上帝用来描写宇宙的文字(伽俐略语)。

素数，又称为质数，是不能被 1 与本身以外的其他整数整除的整数。如 2,3,5,7,11,13,17 是前几个素数。

与此相对应，一个整数如果能被除 1 与本身以外的整数整除，该整数称为合数或复合数。例如，15 能被除 1 与 15 以外的整数 3 和 5 整除，则 15 是一个合数。

对于指定的正整数 m（m≤200），搜索最小连续 m 个合数，输出该区间的起始与终止数。

例如 m=5，最小连续 5 个合数为：24～28。

本问题搜索最小连续 m 个合数，与筛选素数紧密关联。

筛选素数常用试商法与筛法，试商法是依据素数的定义来实施的。

1. 试商法设计

（1）试商设计要点

应用试商法来探求区间[c,d]中的奇数 i（只有唯一偶素数 2，不作试商判别）是不是素数，用奇数 j(取 3,5,…,直至 sqrt(i))去试商。若存在某个 j 能整除 i，说明 i 能被 1 与 i 本身以外的整数 j 整除，i 不是素数。若上述范围内的所有奇数 j 都不能整除 i，则 i 为素数。

这里设计为双重枚举：对指定区间[c,d]中的所有奇数 i 的枚举；对每一个奇数 i 的试商奇数 j 的枚举。

顺便指出，把试商奇数 j 的取值上限定为 i/2 或 i-1 也是可行的，但并不是可取的，这样无疑会增加许多无效试商。理论上说，如果 i 存在一个大于 sqrt(i)且小于 i 的因数，则必存在一个与之对应的小于 sqrt(i)且大于 1 的因数，因而从判别功能来说，取到 sqrt(i)已足够了。

判别 j 整除 i，常用表达式 i%j==0 实现。

（2）试商设计描述

```
// 求最小的连续 m 个合数
main()
{ long i,j,f,t,m;
  printf("请输入 m:"); scanf("%ld",&m);
  printf("最小的连续%ld 个合数为:",m);
  i=f=3;
  while(1)
     {i+=2;
      for(t=0,j=3;j<=sqrt(i);j+=2)          // 试商判别素数
         if(i%j==0) {t=1;break;}
      if(t==0)                              // t 为 0 表明 i 为素数
        { if(i-f>m)
            {printf("%ld,%ld",f+1,f+m);return;}
          f=i;
        }                                   // f 为 i 的前一个素数
     }
}
```

（3）算法改进

把以上算法中检测 i 是否为素数的试商程序段去除，设计一个判定素数的检测函数，供检测 i 时调用。

2. 应用筛法设计

当 m 比较大时，搜索范围相应较大,采用效率较高的筛法求素数是适宜的。

（1）筛选设计要点

求出区间[c,d]内的所有素数（区间起始数 c 可由小到大递增），只需相邻的两素数之差大于 m 即可。

应用筛法求指定区间[c,d]（约定起始数 c 为奇数）上的所有素数，对于该区间内总共

e=[(d-c)/2]（[x]表示取不大于 x 的最大整数,下同）个奇数，把从 3 开始至 d 的所有奇数 i 的倍数 gi,(g+2)i,…全部作筛除标记-1。这里，g=2[c/2i]+1 是使 gi 接近下限 c 的奇数，从而尽可能减少无效筛选操作。最后，凡未作删除标记-1 的即为区间[c,d]中的素数。

区间[c,d]的起始数 c 初始化为 3,而 d=c+20000。以后,c 赋给该区间最大素数 b,而 d 仍取 c+20000,继续分段搜索，既不重复，也无遗漏。

为了扩大程序的应用范围，设计的程序可求最小 m（2～200）个连续合数（整数 m 由键盘输入）。在分段区间[c,d]（每段后递增：c=b;d=c+20000;）中求得两相邻素数 f 与 b，若其差满足条件 b-f>=m+1，打印最小的 m 个连续合数[f+1,f+m]。

（2）筛选设计描述

```
// 求最小的连续 m 个合数
main()
{ long c,d,f,g,i,j,k,b,a[11000];
  int e,m,u,t;
  printf("输入 m:"); scanf("%d",&m);
  c=3;d=c+20000;u=1;f=2;t=0;
  while(u)                                  // 在[c,d]中筛选素数
     {for(i=0;i<=10999;i++) a[i]=0;
      e=(d-c)/2;i=1;
      while (i<=sqrt(d))
         {i=i+2;g=2*(c/(2*i))+1;
          if(g*i>d) continue;
          if(g==1) g=3;
          j=i*g;
          while (j<=d)
             {if(j>=c) a[(j-c)/2]=-1;       // 筛去标记-1
              j=j+2*i;
             }
         }
       for(u=1,k=0;k<=e;k++)
         {if(a[k]!=-1)
            {b=c+2*k;                       // b 即筛选所得素数
             if(b-f>=m+1 && t==0)           // 寻求最小的 m 个连续素数
               { printf("  最小%d 个连续合数区间为:",m);
                 printf("%ld,%ld\n",f+1,f+m);return;
               }
             f=b;
            }
         }
       c=b;d=c+20000;                       // 最大素数 b 赋给 c,继续探求
     }
}
```

3. 算法测试与分析

```
请输入 m: 200
最小 200 个连续合数区间为: 20831324,20831523
```

两个设计的时间复杂度比较：设区间中整数量为 n，试商法的时间复杂度为 $O(n\sqrt{n})$，而

筛法为直接删除，时间复杂度比试商法低。

当 m 较大时（例如 m=200），涉及区间中整数量 n 相应也大，筛法的效率明显高于试商法。但试商法在设计方面比筛法要简单。当 m 较小时（例如 m≤150），试商法的搜索速度还是可以接受的。

2.4　解方程与不等式

解方程与解不等式是算法设计新颖的基础课题之一。有些不定方程或较为复杂的不等式用常规的推理方法求解难以实现时，可考虑运用枚举设计有效求解。

2.4.1　佩尔方程

试求关于 x,y 的不定方程 $x^2 - n \cdot y^2 = 1$（其中 n 为非平方正整数）的正整数解。

最早求解这类不定方程的是印度数学家婆什伽罗。这个方程传到欧洲后，欧洲人称为佩尔（Pell）方程，这是数学家欧拉的误会引起的。实际上，佩尔并未解过这类方程，倒是费尔马解过，因此有人把这一方程称为费尔马方程。

佩尔（Pell）方程是关于 x,y 的二次不定方程。当 x=1 或 x=-1，y=0 时，显然满足方程。常把 x,y 中有一个为零的解称为平凡解，我们要求佩尔方程的非平凡解。

佩尔方程的非平凡解很多，这里只要求出它的最小解，即 x,y 为满足方程的最小正数的解，又称基本解。求出了基本解，其他解可由基本解推出。

对于给定的非平方正整数 n，试求出佩尔方程的基本解。

1. 枚举设计要点

对 y 从 1 开始递增 1 取值枚举，每一个 y 值计算 a=n*y*y 后判别：

若 a+1 不是平方数，则 y 增 1 后再试。

若 a+1 为某一整数 x 的平方，则(x,y)即为所求方程的解。因为 y 是从 1 开始递增的，所得到的解无疑是方程的基本解。

2. 枚举设计描述

```
// 解 Pell 方程: x^2-ny^2=1
main()
{ int n,m; double a,x,y;
  printf(" 请输入非平方整数 n: ");
  scanf("%d",&n);
  m=(int)sqrt(n);
  if(m*m==n)
     { printf(" n 须为非平方数!\n"); return; }
  y=0;
  while(1)
    { y++;                          // 对 y 递增枚举
      a=n*y*y;x=floor(sqrt(a+1));
      if(x*x==a+1)                  // 检测是否满足方程
          { printf(" 方程 x^2-%dy^2=1 的基本解为: ",n);
            printf(" x=%.0f, y=%.0f\n",x,y);
```

```
            break;
          }
      if(x>1000000000)
         { printf(" 此算法不能解方程 x^2-%dy^2=1\n",n);
           return;
         }
   }
}
```

3. 算法测试与说明

```
请输入非平方整数 n：73
方程 x^2-73y^2=1 的基本解为：x=2281249，y=267000
```

对于某些非平方数 n，例如 n=991，方程的解高达 30 位，其位数大大超过计算机语言有效数字的范围，枚举不可能给出正确的解，因此算法中加了另一个结束算法的条件“x＞1000000000”（此结束条件可根据具体情形而定）。

对于解的位数超范围的佩尔方程求解，必须应用其他专业算法（如连分数法等）才能进行准确求出。

2.4.2 分数不等式

试解以下关于正整数 n 的不等式

$$m<1+\frac{1}{2}-\frac{1}{3}+\frac{1}{4}+\frac{1}{5}-\frac{1}{6}+\cdots\pm\frac{1}{n}$$

其中 m 为从键盘输入的正整数，式中符号为两个“+”号后一个“-”号，即分母能被 3 整除时符号为“-”。

1. 枚举设计要点

式中出现减运算，导致不等式的解可能分段。

为叙述方便，记

$$s(n)=1+\frac{1}{2}-\frac{1}{3}+\frac{1}{4}+\frac{1}{5}-\frac{1}{6}+\cdots\pm\frac{1}{n}$$

（1）设置条件循环，每三项一组（包含两正一负）累加求和：

s=s+1.0/k+1.0/(k+1)-1.0/(k+2)（k=1,4,…）

若累加到某一组时 s＞m，退出循环，d=k+1，可得区间解：n≥d；

因 s(d+1)＞m，显然 s(d)＞m；

而 n=d+2 时，1.0/(n+3)为“+”，可得 s(d+2)＞m；

以后各项中，“-”项小于其前面的“+”项，可知对于 n＞d+2 有 s(n)＞m 成立。

（2）在 n<d 时是否有解，逐个求和检验，确定离散解。

因而有必要回过头来，在 n=1～d 中一项项求和，得个别离散解。这一步不能省，否则出现遗解。

2. 枚举描述

```
// 解不等式：m<1+1/2-1/3+1/4+1/5-1/6+...+-1/n
main()
{ long d,m,k; double s;
  printf(" 请输入 m: "); scanf("%d",&m);
```

```
  k=-2;s=0;
  while(s<=m)
     { k=k+3;s=s+1.0/k+1.0/(k+1)-1.0/(k+2); }
  d=k+1;                                  // 可确定区间解 n≥d
  for(s=0,k=1;k<=d-1;k++)
    { if(k%3>0) s=s+1.0/k;                // 逐项累加求和
      else s=s-1.0/k;
      if(s>m)
         printf("  n=%ld,",k);            // 逐个输出离散解
    }
  printf("  n>=%ld \n",d);                // 最后输出区间解
}
```

3. 算法测试与思考

```
请输入 m: 4
n=10151, n=10153, n>=10154
```

注意：要特别注意，不要把前面的离散解遗失。

变通：如果把后一个离散解写入区间解中，能否简化逐项求和找出离散解？

4. 枚举改进

（1）每三项一组（包含两正一负）累加求和：

s=s+1.0/k+1.0/(k+1)-1.0/(k+2)（k=1,4,…）

若累加到某一组时 s＞m，退出循环，d=k+1, 可得区间解：n≥d；

（2）此时，s(d-1)有可能大于 m。

为得到 s(d-1)，在原 s(d+1)基础上实施-1.0/d+1.0/(d+1)得 s(d-1)：

若 s(d-1)＞m，合并得区间解：n≥d-1；

若 s(d-1)＜m，区间解为：n≥d；

（3）当 s(d-1)＞m 时，s(d-3)还有可能大于 m。

因而在 s(d-1)的基础上实施 s+1.0/(d-2)-1.0/(d-1),得 s(d-3)：

若 s(d-3)＞m，得一个离散解：n＝d-3；

若 s(d-3)＜m，没有离散解。

（4）改进枚举描述

```
// 解不等式：m<1+1/2-1/3+1/4+1/5-1/6+...+-1/n
main()
{ long d,k,t,m; double s;
  printf("\n  请输入 m: "); scanf("%d",&m);
  k=-2;s=0;
  while(s<=m)
     { k=k+3;s=s+1.0/k+1.0/(k+1)-1.0/(k+2); }
  d=k+1;                                  // 可确定区间解 n≥d
  s=s-1.0/d+1.0/(d+1);                    // 得 s(d-1)
  if(s>m) t=d-1;
  else t=d;                               // 得区间解 n≥t
  s=s+1.0/(d-2)-1.0/(d-1);                // 得 s(d-3)
  if(s>m) printf("  n=%ld,",d-3);         // 输出一个离散解
  printf("  n≥%ld \n",t);                 // 输出区间解
}
```

（5）算法测试与分析

```
请输入 m: 7
 n=82273511,  n≥82273513
```

原枚举设计与改进后的枚举设计的时间复杂度都是 O(n)，深入分析可知，改进后枚举所需时间只有原枚举时间的 1/4。

2.5 数式与运算

应用枚举不仅可以搜索整数与求和统计，也可以构建某类数式并实施运算。

2.5.1 奇数序列运算式

试在由指定相连奇数组成的序列的每相邻两项中插入运算符号：

若相邻两项都是合数，则两项中插入“-”号；

若相邻两项中一项是合数，另一项是素数，则两项中插入“+”号；

若相邻两项都是素数，则两项中插入乘号“*”号；

输入奇数 b,c（1＜b＜c），根据以上规则插入运算符号，完成区间[b,c]中奇数序列的运算式，计算并输出该式的运算结果。

例如 b=31,c=45，完成运算式并计算得：31+33-35+37+39+41*43+45=1913。

1. 设计要点

序列各项是否为素数决定了运算符号，直接关系到运算式的结果。

（1）首先枚举区间[b,c]中的奇数，应用试商法确定每一个奇数是否为素数：

标注 a[k]=1,表示区间[b,c]中的第 k 个奇数 2k+(b-2)为素数；

标注 a[k]=0,表示区间[b,c]中的第 k 个奇数 2k+(b-2)非素数；

（2）根据相邻两项决定两项中的运算符号。

若 a[i-1]+a[i]==0，两项都是合数，插入“-”号；

若 a[i-1]+a[i]==1，两项中一项是素数，一项是合数，插入“+”号；

若 a[i-1]+a[i]==2，两项都是素数，插入“*”号；

（3）完成运算，这是设计的关键环节。

运算必须遵循先乘后加减的运算规则。

在枚举每个奇数的 i(1～n)循环中：

1）把第 i 个奇数值赋给变量 t:t=2*i+b-2；同时记下位置 i:f=i；

2）先实施乘运算：

```
    while(a[i]+a[i+1]==2)
       {i++;t=t*(2*i+b-2);}
```

这里应用条件循环，是因为注意到有连乘现象，例如 3*5*7。

3）然后实施加减。

2. 算法描述

```
// 奇数序列运算式
main()
{int b,c,f,n,k,i,j,a[3000];long t,s;
 printf(" 请输入首尾奇数 b,c(b<c): ");
```

```
 scanf("%d,%d",&b,&c);
 n=(c-b+2)/2;                    // 计算奇数序列 n 项
 for(k=1;k<=n+1;k++) a[k]=0;
 for(k=1;k<=n;k++)
  { for(t=0,j=3;j<=sqrt(2*k+b-2);j+=2)
      if((2*k+b-2)%j==0) {t=1;break;}
    if(t==0) a[k]=1;             // 标记第 k 个奇数 2k+b-2 为素数
  }
 printf("\n %d",b);
 for(i=2;i<=n;i++)               // 完成表达式
      {if(a[i-1]+a[i]==0) printf("-%d",2*i+b-2);      // 插入减号
       if(a[i-1]+a[i]==1) printf("+%d",2*i+b-2);      // 插入加号
       if(a[i-1]+a[i]==2) printf("*%d",2*i+b-2);      // 插入乘号
      }
 s=0; a[0]=1-a[1];                       // 确保第一项前为“+”
 a[n+1]=0;                               // 确保最后一项不“*”
 for(i=1;i<=n;i++)                       // 计算表达式结果
    { t=2*i+b-2;f=i;
      while(a[i]+a[i+1]==2)
          {i++;t=t*(2*i+b-2);}           // 先实施乘
      if(a[f-1]+a[f]==0)  s=s-t;         // 后实施加减
      if(a[f-1]+a[f]==1)  s=s+t;
    }
 printf("=%d.\n",s);
}
```

3. 算法测试与分析

```
请输入首尾奇数 b,c(b<c): 3,31
 3*5*7+9+11*13+15+17*19+21+23+25-27+29*31=1536
```

以上算法设计中，先乘后加减的处理是巧妙的。

本设计有三个枚举循环：素数判别；完成运算式；完成运算。其中素数判别循环中，含试商因数枚举的内循环。设区间中的奇数个数为 n，算法的时间复杂度为 $O(n\sqrt{n})$。

2.5.2　完美综合运算式

以下含乘方（a^b 即为 a 的 b 次幂）、加、减、乘、除的综合运算式（2.3）的右边为一位非负整数 f，请把数字 0～9 这 10 个数字中，不同于数字 f 的 9 个数字不重复地填入式（2.3）左边的 9 个□中（约定数字“1”、“0”不出现在式左边的一位数中，且“0”不为首位），使得该综合运算式成立。

$$□^{□}+□□÷□-□□□×□=f \tag{2.3}$$

满足上述要求的式（2.3）称为完美综合运算式。

输入非负整数 f（0≤f≤9），输出相应的综合运算式。

1. 按双精度型设计

（1）设计要点

设置 a,b,c,d,e,z 变量，所求的综合运算式为

$$a^{b}+z/c-d*e=f \tag{2.4}$$

注意到 a^b 属双精度计算，可把变量设计为 double 型。

同时设置 a,b,c,z,d,e 循环，所有量设置在整数范围内枚举。式右数字 f 从键盘输入。

注意到式中有 z/c,即式中 z 必须是 c 的整数倍，c 循环可设置为

for(z=2*c;z<=98;z=z+c)

1）若等式不成立，即 pow(a,b)+z/c!=d*e+f, 则返回继续；

2）检测式中 10 个数字是否存在相同数字：

对 7 个整数共 10 个数字进行分离，分别赋值给数组 g[0]～g[9]。共 10 个数字在二重循环中逐个比较：

若存在相同数字，t=1，不作输出。

若不存在相同数字，即式中 10 个数字为 0～9 不重复，保持标记 t=0, 则输出所得的完美综合运算式，并设置 n 统计解的个数。

（2）算法描述

```
// □^□+□□/□-□□□*□=f
// 式左的一位数不能为 0 或 1, 式左的二位数首位不能为 0
main()
{double a,b,c,d,e,f,z,g[10];
 int j,k,t,n;
 printf("请输入式右非负数字 f:");
 scanf("%lf",&f);
 n=0;
 for(a=2;a<=9;a++)
 for(b=2;b<=9;b++)
 for(c=2;c<=9;c++)
 for(z=2*c;z<=98;z=z+c)          // 各数实施枚举，确保 z 为 c 的倍数
 for(d=102;d<=987;d++)
 for(e=2;e<=9;e++)
    { if(pow(a,b)+z/c!=d*e+f) continue;        // 检验等式是否成立
      t=0;
      g[0]=f;g[1]=a;g[2]=b;g[3]=c;g[4]=e;      // 10 个数字赋给 g 数组
      g[5]=fmod(d,10);g[6]=fmod(floor(d/10),10);
      g[7]=floor(d/100);
      g[8]=fmod(z,10);g[9]=floor(z/10);
      for(k=0;k<=8;k++)
          for(j=k+1;j<=9;j++)
      if(g[k]==g[j])
           {t=1; break;}          // 检验数字是否有重复
      if(t==0)
         { n++;                   // 统计并输出一个解
           printf("%2d: %.0f^%.0f+%.0f/%.0f",n,a,b,z,c);
           printf("-%.0f*%.0f=%.0f   \n",d,e,f);
         }
    }
 }
```

（3）算法测试与说明

```
请输入式右非负数字 f:0
1: 4^6+72/9-513*8=0
2: 5^4+78/6-319*2=0
```

该题只限于 10 个数字处理，算法的运行流畅。

2. 按整形改进设计

（1）设计要点

尽管式中有 a^b，可改进为整形处理，a^b 用 a 自乘 b 次实现。

同时设置 a,b,c,d,e 循环，所有量设置在整数范围内枚举，式右数字 f 从键盘输入。

把运算式（2.4）变形为

$$z=(d*e+f-a\hat{}b)*c \tag{2.5}$$

对每一组 f,a,b,c,d,e，按式（2.5）计算 z。

检测 z 是否为二位数。若计算所得 z 非二位数，则返回。

然后分别对 7 个整数进行数字分离，设置 g 数组对 7 个整数分离的共 10 个数字进行统计，g(x)即为数字 x(0～9)的个数。

若某一 g(x)不为 1，不满足数字 0～9 这 10 个数字都出现一次且只出现一次，标记 t=1。

若所有 g(x)全为 1，满足数字 0～9 这 10 个数字都出现一次且只出现一次，保持标记 t=0，则输出所得的完美综合运算式。

（2）算法描述

```
// 运算式 □^□+□□/□-□□□*□=f
// 式左的一位数不能为 0 或 1, 0 不能为二位或三位数首位
main()
{int a,b,c,d,e,f,k,t,m,n,x,z,g[10];
 n=0;
 printf("  请输入式右非负数字 f:");
 scanf("%d",&f);
 for(a=2;a<=9;a++)
 for(b=2;b<=9;b++)
 for(c=2;c<=9;c++)
 for(d=102;d<=987;d++)                          // 实施枚举
 for(e=2;e<=9;e++)
   { for(t=1,k=1;k<=b;k++) t=t*a;               // 计算乘方 a^b
     z=(d*e+f-t)*c;
 if(z<10 || z>98)  continue;
     for(x=0;x<=9;x++) g[x]=0;
  g[a]++;g[b]++;g[c]++;g[e]++;g[f]++;           // g 数组统计
  g[d%10]++;g[d/100]++;m=(d/10)%10;g[m]++;
  g[z%10]++;g[z/10]++;
     for(t=0,x=0;x<=9;x++)
        if(g[x]!=1) {t=1;break;}                // 检验数字 0～9 各出现一次
     if(t==0)
       { n++;                                   // 统计并输出一个解
         printf("%2d: %d^%d+%d/%d",n,a,b,z,c);
```

```
            printf("-%d*%d=%d   \n",d,e,f);
          }
        }
      }
    }
```

（3）算法测试

```
请输入式右非负数字 f:5
1: 2^9+78/6-130*4=5
2: 9^3+64/2-108*7=5
```

3. 两种设计的比较与变通

不要局限于 a^b 是双精度计算就必须设置双精度变量处理，第 2 个设计应用 a 自乘 b 次计算 a^b 是可取的。

为了检测是否存在相同数字，两个设计中都设置了 g 数组，但两个设计中 g 数组的意义并不相同：前一个设计中每一个 g 数组元素存储一个数字，例如 g[0]=f,g[1]=a,…，然后通过二重循环比较不同的元素是否存在相同；而后一个设计中，10 个数字分别作为 g 数组的下标进行统计,只要通过一重循环检测 g 数组元素是否都为 1 即可。例如 g[f]++;g[a]++;…，如果 f=5,a=5,则 g[5]=2,说明数字“5”有重复。

同样是枚举设计，按整形改进设计可省略 z 循环，同时省略 z 是否能被 c 整除、等式是否成立的检测，显然枚举效率会高一些。

变通：把数字 0～9 这 10 个数字分别填入以下含加、减、乘、除与乘方（^）的综合运算式中的 10 个□中，使得该式成立

□^□+□□÷□□-□□×□=□

要求数字 0～9 这 10 个数字在式中出现一次且只出现一次，且约定数字“0”与“1”不出现在式左的一位数中，数字“0”不能为高位数字。

2.6 数列与数阵

探究某些有着特殊要求的数列与数阵（矩阵与方阵）是枚举设计的应用课题之一。

2.6.1 H 形数序列

定义形如 ab…bc 的数叫做 H 形数，其中数字 a 为高位，数字 c 为低位，数字 b 为中间段，中间段的位数为 H 形数位数减 2（至少有 1 位）。

显然，所有 3 位数都是 H 形数（其中 100 为最小的 H 形数），把所有 3 位数的中间数字多次重复可得 4 位及 4 位以上的 H 形数。

把 H 形数从小到大排序，构成 H 形数数列，试求 H 形数数列的第 n 项（约定 n≤1000000）与前 n 项之和。

1. 枚举设计要点

作为 H 形数的特例，当 a=b=c 时，H 形数即为全码相同数。

（1）项数与位数的关系

设第 n 项为 m 位 H 形数，注意到 3 位 H 形数共 900 个（100～999），4 位 H 形数也为 900 个（1000～9999），……，因而可归纳得

m=[(n-1)/900]+3　　（这里[x]为正数 x 取整）

（2）考察 H 形数列的排列规律

首先按其位数 m 升序排列，位数少的在前，位数多的在后；

当位数 m 相同时，按高位数字 a 升序排列，a 小在前，a 大在后；

当位数 m 与高位数字 a 相同时，按中位数字 b 升序排列，b 小在前，b 大在后；

当位数 m 与数字 a、b 都相同时，按低位数字 c 升序排列，c 小在前，c 大在后。

因而设置 4 重循环从小到大枚举 H 形数：

1）H 形数的位数 m：（3——），步长为 1 递增；

2）高位数字 a：（1～9）；

3）中位数字 b：（0～9）；

4）低位数字 c：（0～9）。

（3）实施求和

设置 s 数组求和，s[1]为和的个位数字，s[2]为和的十位数字，依此类推。

每生成一个 m 位 H 形数，即实施高、中、低三部分别按对应位求和：

```
s[1]+=c;                          // 个位求和
for(j=2;j<=m-1;j++) s[j]+=b;      // 中间段各位求和
s[m]+=a;                          // 高位求和
```

当达到 n 项时，和数组 s 从个位开始逐步向高位进位。即 s[j]的十位以上部分进位到它的高一位，s[j]的个位部分留在本位：

```
s[j+1]+=s[j]/10; s[j]=s[j]%10;(j=1,2,…,m-1)
```

（4）结果输出

输出 m 位 H 形数：高位数字 a；m-2 个中位数字 b；最后为低位数字 c。

输出和：从 s[m]开始，依次输出至 s[1]。

注意：此时 s[m]可能是一个多位数，而 s[m-1],…,s[1]均为一位数字。

2. 枚举描述

```
// 枚举求 H 形数列的第 n 项
main()
{ int a,b,c,j;  long k,m,n,s[2000];
  printf("  请输入n(<1000000): ");  scanf("%ld",&n);
  m=(n-1)/900+3;
  for(j=1;j<=m+1;j++) s[j]=0;
  m=2;k=0;
  while(1)
    { m++;                                  // 从m=3位开始
      for(a=1;a<=9;a++)
      for(b=0;b<=9;b++)
      for(c=0;c<=9;c++)
        { k++;s[1]+=c;s[m]+=a;              // 对应位求和
          for(j=2;j<=m-1;j++) s[j]+=b;
          if(k==n)                          // 到达数列的第n项时输出
            { printf("  H形数列的第%ld项为:%d",n,a);
              for(j=2;j<=m-1;j++)  printf("%d",b);
```

```
                printf("%d\n",c);
                printf("  数列前%d项之和为:",n);
                for(j=1;j<=m-1;j++)        // 完成进位
                  {s[j+1]+=s[j]/10;s[j]=s[j]%10;}
                for(j=m;j>=1;j--) printf("%ld",s[j]);
                printf("\n");
                return;
              }
          }
      }
}
```

3. 算法测试与分析

```
请输入 n(<1000000): 20136
H 形数列的第 20136 项为: 433333333333333333333335
数列前 20136 项之和为: 144933333333333333333268380
```

算法的操作“k++;”次数即循环次数为 n，对每一个数实施 m 次求和。注意到 m 是变化的，且 m<n，可知算法的时间复杂度低于 $O(n^2)$。

注意：为实现 H 形数的升序枚举，4 重循环 m,a,b,c 的嵌套结构不能随意变更。

算法中进位循环的循环体操作能否交换顺序？

s[j]=s[j]%10; s[j+1]+=s[j]/10; (j=1,2,…,m-1)

2.6.2 三阶素数幻方

通常的 n 阶幻方由 1,2,...,n^2 填入 n×n 方格，构成 n 行、n 列与两对角线之和均相等的方阵。素数幻方是由素数构成的各行、各列与两对角线之和均相等的方阵。

试在指定区间[c,d]找出 9 个素数，构成一个三阶素数幻方，使得该方阵中 3 行、3 列与两对角线上的 3 个数之和均相等。

输入区间 c,d，输出由该区间中素数构建的所有三阶素数幻方。

1. 数学建模

设正中间数为 n，每行、每列与每对角线之和为 s。注意到：

$$(\text{中间一行})+(\text{中间一列})+(\text{两对角线})=4s$$

$$\text{方阵所有 9 个数之和}=3s$$

两式相减即得：

$$3n=s \rightarrow n=s/3$$

这意味着凡含 n 的行、列或对角线的 3 个数中，除正中数 n 之外的另两数与 n 相差等距。为此，设方阵为：

$$\begin{matrix} n-x & n+w & n-y \\ n+z & n & n-z \\ n+y & n-w & n+x \end{matrix}$$

为避免解的重复，约定两对角线的 3 个数为大数在下（即 x,y>0），下面一行 3 个数为大数在右（即 x>y）。

显然，上述 3×3 方阵的中间一行、中间一列与两对角线上 3 数之和均为 3n。要使左右两列、上下两行的 3 数之和也为 3n，当且仅当

z=x-y

w=x+y　（x>y）

同时易知 9 个素数中不能有偶素数 2，因而 x、y、z、w 都只能是正偶数。

2. 枚举设计

首先枚举区间[c,d]中的奇数 k，在 a 数组中赋值 a[k]=0 的基础上，应用试商法找出素数 k，同时赋值 a[k]=1。

建立 n 循环枚举[c,d]中的奇数，若 n 非素数（a[n]=0）则返回。

对于每一个素数 n，枚举 y、x，并按上述两式得 z、w：

①若出现 x=2y，将导致 z=y，方阵中出现两对相同的数，显然应排除。

②显然，n-w 是 9 个数中最小的，n+w 是 9 个数中最大的。若 n-w<c 或 n+w>d，已超出[c,d]界限，应予以排除。

③检测方阵中其他 8 个数 n-x、n+w、n-y、n+z、n-z、n+y、n-w、n+x 是否同时为素数，引用变量 t1,t2,t1*t2 为 8 个数的 a 标记之积。若 t1*t2=0，即 8 个数中存在非素数，返回。

④否则，已找到一个三阶素数幻方解，按方阵格式输出三阶素数幻方并用变量 m 统计解的个数。

这样处理能较快地找出所有解，既无重复，也无遗漏。

3. 算法描述

```
// 三阶素数幻方
main()
{ int c,d,j,k,n,t,t1,t2,w,x,y,z,m;
  int a[3000];
  m=0;
  printf("  请确定区间下限 c,上限 d: ");
  scanf("%d,%d",&c,&d);
  if(c%2==0) c=c+1;
  for(k=c;k<=d;k++) a[k]=0;
  for(k=c;k<=d;k+=2)
    { for(t=0,j=3;j<=sqrt(k);j+=2)
        if(k%j==0) {t=1;break;}
      if(t==0) a[k]=1;                  // [c,d]中的奇数 k 为素数，标注 1
    }
  for(n=c;n<=d-8;n=n+2)
   { if(a[n]==0) continue;              // 排除正中数 n 为非素数
     for(y=2;y<=n-3;y+=2)
     for(x=y+2;x<=n-1;x+=2)
      { z=x-y;w=x+y;
        if(x==2*y || n-w<c || n+w>d)
             continue;                  // 控制幻方的素数范围
        t1=a[n-w]*a[n+w]*a[n-z]*a[n+z];
        t2=a[n-x]*a[n+x]*a[n-y]*a[n+y];
        if(t1*t2==0) continue;          // 控制其余 8 个均为素数
        m++;
        printf("  NO %d:\n",m);         // 统计并输出三阶素数幻方
        printf("%5d%5d%5d\n",n-x,n+w,n-y);
```

```
            printf("%5d%5d%5d\n",n+z,n,n-z);
            printf("%5d%5d%5d\n",n+y,n-w,n+x);
          }
        }
    printf("共 %d 个素数幻方.\n",m);
}
```

4. 算法测试与分析

```
请确定区间下限 c,上限 d: 3,120
  NO 1:
    17   113    47
    89    59    29
    71     5   101
  NO 2:
    41   113    59
    89    71    53
    83    29   101
 共 2 个素数幻方.
```

设指定区间中奇数个数为 n，本算法的时间复杂度为 $O(n^3)$。

变通：请修改程序，构建指定幻和的三阶素数幻方。

2.7 表格与图形

探索制作具有某类特性的表格与图形是枚举设计最具魅力的应用课题之一。本节试应用枚举设计制作新颖的 p 进制乘法表，构建奇妙的和积三角形，颇具启发性。

2.7.1 p 进制乘法表

设计十进制九九乘法表在许多程序设计资料中很常见。本节在九九乘法表设计的基础上，应用枚举设计创建新颖的一般 p（2～16）进制乘法表。

输入 p（2～16），构建并输出 p 进制乘法表。

1. 设计要点

当 p＞10 时，因涉及数字 A、B、C、D、E、F（分别对应数字 10、11、12、13、14、15），设置字符数组 d[17]="0123456789ABCDEF"。

设置两个乘数 k、j 的枚举循环，k（1～p-1），j（1～k）。两个乘数相乘得积为 t=k*j。

对其乘积 t 分为两类输出：

①t＜p 时，乘积只有一位，输出字符串数组中的第 t 个字符 d[t]即可；

②t≥p 时乘积为两位，输出高位为 d[(t/p)]，低位为 d[t%p]。

2. 2～16 进制乘法表程序设计。

```
// p(2～16)进制乘法表
main()
{ int k,j,t,p;
  char d[17]="0123456789ABCDEF";
  printf("input p: ");
  scanf("%d",&p);
```

```
  printf(" %d 进制乘法表:\n",p);
  for(k=1;k<=p-1;k++)
    { printf("  %c ",d[k]);            // 打印左竖列乘数
      for(j=1;j<=k;j++)
        { t=k*j;                       // 对乘积 t 作分别输出
          if(t<p)
             printf("  %c ",d[t]);
          else
             printf(" %c%c ",d[(t/p)],d[t%p]);
        }
      printf("\n");
    }
  printf("  * ");
  for(k=1;k<=p-1;k++)
      printf("  %c ",d[k]);            // 打印下横行乘数
  printf("\n");
}
```

3. 算法测试与说明

输入 p=16，得十六进制乘法表，如图 2-2 所示。

十六进制乘法表：

1	1													
2	2	4												
3	3	6	9											
4	4	8	C	10										
5	5	A	F	14	19									
6	6	C	12	18	1E	24								
7	7	E	15	1C	23	2A	31							
8	8	10	18	20	28	30	38	40						
9	9	12	1B	24	2D	36	3F	48	51					
A	A	14	1E	28	32	3C	46	50	5A	64				
B	B	16	21	2C	37	42	4D	58	63	6E	79			
C	C	18	24	30	3C	48	54	60	6C	78	84	90		
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4
F	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1
*	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

图 2-2　十六进制乘法表

由该表可知，在十六进制中，C*E=A8。

若输入 p=10，即得常见的十进制九九乘法表。

2.7.2　基于 s 的和积三角形

1. 问题提出

把给定的正整数 s（s≥45）分解为 9 个互不相等的正整数，把这 9 个整数不重复地填入 9 数字三角形（如图 2-3 所示）中的圆圈，若三角形三边上的 4 个数字之和相等（s1），且三边上的 4 个数字之积也相等（s2），则该三角形称为基于 s 的和积三角形。

对于指定正整数 s，探索并输出基于 s 的所有和积三角形。

2. 设计要点

把和为 s 的 9 个正整数存储于 b 数组 b(1),…,b(9)中，分布如图 2-4 所示。为避免重复，

不妨约定三角形中数字“下小上大、左小右大”，即三顶角数 b(1)＜b(7)＜b(4)，三边的中间二数 b(2)＜b(3)，b(6)＜b(5)，b(9)＜b(8)。

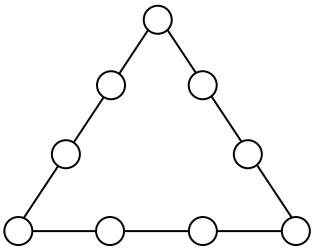

图 2-3　9 数字三角形

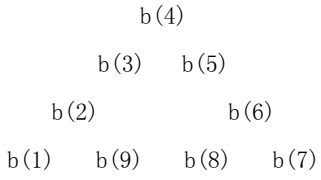

图 2-4　b 数组分布示意图

（1）三顶角数枚举探索

根据约定对 b(1)、b(7)和 b(4)的大小关系进行枚举探索。

b(1)的取值范围：1～(s-21)/3（因其他 6 个数之和至少为 21）。

b(7)的取值范围：b(1)+1～(s-b(1)-21)/2。

b(4)的取值范围：b(7)+1～(s-b(1)-b(7)-21)。

（2）判断 s1

注意到计算三边和时，三顶角数各重复了一次，即有关系式

s+b(1)+b(7)+b(4)=3*s1

若(s+b(1)+b(7)+b(4))%3≠0，则返回探索；否则，记 s1=(s+b(1)+b(7)+b(4))/3。

（3）各边中间数枚举探索

根据各边 4 数之和为 s1，对 b(3)、b(5)和 b(8)的值进行枚举探索。

b(3)的取值范围：(s1-b(1)-b(4))/2+1～s1-b(1)-b(4)。

b(5)的取值范围：(s1-b(4)-b(7))/2+1～s1-b(4)-b(7)。

b(8)的取值范围：(s1-b(1)-b(7))/2+1～s1-b(1)-b(7)。

然后计算出 b(2)、b(6)和 b(9)：

b(2)=s1-b(1)-b(4)-b(3)

b(6)=s1-b(4)-b(5)-b(7)

b(9)=s1-b(1)-b(7)-b(8)

（4）检测 b 数组

设计二重循环，检测 b 数组是否存在相同数，若 b 数组存在相同正整数，则返回探索；若不存在相同正整数，则继续以下检测。

（5）检测三边之积

设 s2=b(1)*b(2)*b(3)*b(4)，若另两边 4 数之积不为 s2，则返回探索；否则探索成功，打印输出一个结果。

所有枚举循环完成，基于 s 的和积三角形探索完毕。

3. 算法描述

```
// 基于 s 的数字三角形
main()
{ int k,j,t,s,s1,s2,n,b[10];
  printf(" 请输入正整数 s:");
```

```
    scanf("%d",&s);
    n=0;
    for(b[1]=1;b[1]<=(s-21)/3;b[1]++)
    for(b[7]=b[1]+1;b[7]<=(s-b[1]-21)/2;b[7]++)
    for(b[4]=b[7]+1;b[4]<=s-b[1]-b[7]-21;b[4]++)
    {
      if((s+b[1]+b[4]+b[7])%3!=0) continue;
      s1=(s+b[1]+b[4]+b[7])/3;
      for(b[3]=(s1-b[1]-b[4])/2+1;b[3]<s1-b[1]-b[4];b[3]++)
      for(b[5]=(s1-b[4]-b[7])/2+1;b[5]<s1-b[4]-b[7];b[5]++)
      for(b[8]=(s1-b[1]-b[7])/2+1;b[8]<s1-b[1]-b[7];b[8]++)
       {
         b[2]=s1-b[1]-b[4]-b[3];
         b[6]=s1-b[4]-b[7]-b[5];
         b[9]=s1-b[1]-b[7]-b[8];
         t=0;
         for(k=1;k<=8;k++)
         for(j=k+1;j<=9;j++)
           if(b[k]==b[j]) {t=1;k=8;break;}
           if(t==1) continue;
           s2=b[1]*b[2]*b[3]*b[4];
           if(b[4]*b[5]*b[6]*b[7]!=s2) continue;
           if(b[1]*b[9]*b[8]*b[7]!=s2) continue;
           n++;
           printf(" %3d: %2d",n,b[1]);
          for(k=2;k<=9;k++)
             printf(", %2d",b[k]);
          printf("  s1=%d, s2=%d \n",s1,s2);
       }
     }
    printf("共%d 个解.",n);
}
```

4. 算法测试与分析

```
请输入正整数 s:73
1:  3,  4, 14, 10, 12,  2,  7, 16,  5  s1=31, s2=1680
共 1 个解.
```

输入小于 73 的整数无输出，说明存在基于 73 的和积三角形是最小的和积三角形。解的图示如图 2-5 所示。

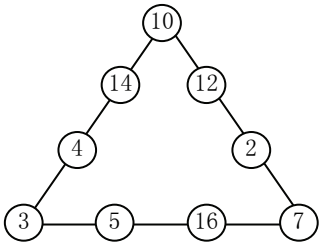

图 2-5　基于 73 的和积三角形

```
请输入正整数 s：90
    1：  2，10，12，18，  5，  3，16，15，  9   s1=42，s2=4320
    2：  3，  5，18，12，15，  2，  9，20，  6   s1=38，s2=3240
    3：  3，  8，10，21，  5，  4，12，20，  7   s1=42，s2=5040
共 3 个解.
```

该算法设计了六重枚举循环，算法设计中，判定“s+b[1]+b[4]+b[7]是否为 3 的倍数”是一个缩减无效循环的优化处理，比简单的六重循环要快捷得多。尽管如此，算法的时间复杂度为 $O(n^6)$，因而难以胜任 s 数量较大的和积三角形探索。

变通：是否存在 8 数字的和积三角形（一边 3 个数，两边各 4 个数）？是否存在 7 数字的和积三角形（一边 4 个数，两边各 3 个数）？

2.8 枚举设计的改进与优化

本章应用枚举设计简明地解决诸如统计求和、整数搜索、解方程、解不等式与数式、数列等常规问题，同时通过枚举探求数阵、表格与图形等有一定难度的实际案例，可见枚举设计的应用领域是广阔的。

本章所列举的各枚举案例中，有些是因为问题本身限制了数量不会太大，例如“完美综合运算式”中，十进制数字最多 10 个；有些是因为问题比较简单，例如“解不等式”，通过简单的一重循环即可求解，其时间复杂度为 O(n)。对于需应用多重循环枚举求解的案例，在 n 不是太大的实际应用范围内，以上各案例的枚举求解所需的时间是可以接受的。

求解这些基本实例时，应该说设计高效率算法的价值不大，就是有时想设计高效率的算法也并不容易实现。

应用枚举求解在设计上比较简单，不存在太多难点，但决不可太随意。从本章的枚举设计可以看出，枚举策略的确定、枚举路线的选择、枚举结构的设置与枚举参量的细化都有一定的技巧运用，自然也存在许多改进与优化的空间。

2.8.1 选择枚举路线

应用枚举设计求解实际案例往往存在若干不同的枚举路线。在缜密审题的基础上，根据求解实例的具体实际确定合适的枚举策略、选择合适的枚举路线，对缩减枚举的时间复杂度至关重要。

例 2-1 全排列数中的平方数

统计由 1～9 这 9 个数字全排列的 9!个 9 位数中平方数的个数，并指出其中最大的平方数。

解：设全排列的 9 位平方数 d=a*a，显然存在两个不同的枚举策略。

1. 枚举 9 位数 d

在区间[123456789，987654321]中枚举 d，对 d 实施以下两步检测：

（1）检测 d 是不是平方数，如果不是平方数，则返回；

（2）检测 d 是否存在重复数字，是否存在数字“0”，若存在，则返回。

设置 f 数组，统计 d 中各个数字的个数。如果 f[3]=2，即平方数 d 中有 2 个“3”。

检测若 f[k]!=1(k=1～9)，说明 d 中存在重复数字或存在数字“0”。

经以上两步检测后，d 满足题意要求，统计并通过赋值求最大的增方数。

```
// 枚举 9 位数 d
main()
{int k,m,n,t,f[10];
long a,b,c,d,w;
n=0;
for(d=123456789;d<=987654321;d++)
{a=(int)sqrt(d);
if(a*a!=d) continue;                    // 确保 d 为平方数
for(k=0;k<=9;k++) f[k]=0;
w=d;
while(w>0)
  { m=w%10;f[m]++;w=w/10;}
    for(t=0,k=1;k<=9;k++)
       if(f[k]!=1) {t=1;break;}      // 测试平方数是否有重复数字
    if(t==0) {n++;b=a;c=d;}
  }
printf("  排列数中共有%d 个平方数.\n",n);
printf("  其中最大者为%ld=%ld^2.\n",c,b);
}
```

算法测试：

```
排列数中共有 30 个平方数.
其中最大者为 923187456=30384^2.
```

2. 枚举整数 a

求出相应最小 9 位数的平方根 b 及最大 9 位数的平方根 c。

用变量 a 枚举[b,c]中的所有整数，计算 d=a*a，以确保 d 为平方数。

检测 d 是否存在重复数字或数字“0”后,作相应处理。

```
// 枚举整数 a
main()
{ int k,m,n,t,f[10];
  long a,b,c,d,e1,e2,w;
  n=0;
  e1=sqrt(123456789);e2=sqrt(987654321);
  for(a=e1;a<=e2;a++)
    { d=a*a; w=d;                       // 确保 d 为平方数
      for(k=0;k<=9;k++) f[k]=0;
      while(w>0)
        { m=w%10;f[m]++;w=w/10;}
      for(t=0,k=1;k<=9;k++)
         if(f[k]!=1) {t=1;break;}      // 测试平方数是否有重复数字或 0
      if(t==0)
        { n++;b=a;c=d;}
    }
  printf("  排列数中共有%d 个平方数.\n",n);
  printf("  其中最大者为%ld=%ld^2.\n",c,b);
}
```

3. 两个枚举策略比较

全排列数的个数为 9!,前一个枚举设计的枚举数量级为 9!，是一个非常庞大的数值。而后者的枚举数量级仅为$\sqrt{9!}$，而且可省去 d 是否为平方数的检测。

这两个枚举策略（即枚举路线）相比较，选用后者是合适的。

2.8.2 精简枚举结构

从算法的时间复杂度考虑，当 n 非常大时，枚举所需时间相应地会长。在进行枚举设计时，精简枚举结构是减少重复操作、降低枚举的时间复杂度的重要一环。

例 2-2 设 n 为正整数，求和

$$s(n)=1-\frac{1}{1+1/2}+\frac{1}{1+1/2+1/3}-\cdots\pm\frac{1}{1+1/2+\cdots+1/n}$$ （和式中各项符号一正一负）

算法 1：二重枚举设计

和式中各项的分母也为和，自然想到在项数枚举循环中，设置求各项分母的枚举内循环。

二重枚举设计描述：

```
  printf("  请输入正整数 n：");
  scanf("%d",&n);
  s=1;
  for(k=2;k<=n;k++)                  // 枚举和式各项
  { t=0;
    for(j=1;j<=k;j++)                // 枚举计算各项分母
       t=t+1.0/j;
    if(k%2==0) s=s-1/t;
    else s=s+1/t;
  }
printf("%lf",s);
```

算法 2：一重枚举设计

在项数 k 枚举循环中，应用 t=t+1/k 直接求出各分数的分母，也就是说，在计算第 k 项分母时直接用到第 k-1 项分母的结果，这样可减少重复计算，省去内循环，把循环结构优化为一重枚举。

一重枚举描述：

```
printf("  请输入正整数 n：");
scanf("%d",&n);
s=1;t=1;
for(k=2;k<=n;k++)                    // 枚举各项
   { t=t+1.0/k;                      // 计算第 k 项的分母
     if(k%2==0) s=s-1/t;
     else s=s+1/t;
   }
printf("%lf",s);
```

算法 1 二重循环的枚举设计的时间复杂度为 $O(n^2)$，而精简枚举结构的算法 2 精简为一重循环，算法 2 枚举设计的时间复杂度为 O(n)。可见这一并不复杂的改进优化了枚举设计的时间复杂度。

2.8.3　优化枚举参数

在枚举结构确定后，枚举循环的参数设置是否合适，直接关系到算法效率的高低。

例 2-3　求解四元二次不定方程 $x^2+y^2+z^2=w^2$ 在指定区间[a,b]的正整数解。

输入正整数 a 和 b（1≤a＜b＜10000），输出方程在区间[a,b]内的正整数解 x、y、z、w（约定 a≤x＜y＜z＜w≤b）。

解：输入指定区间[a,b]，一般设置四重循环在指定区间内枚举 x、y、z、w（x＜y＜z＜w），若满足方程式，则输出解并用 n 统计解数。

（1）基本枚举设计 1

```
// 四重循环基本枚举设计
main()
{long a,b,n,x,y,z,w;
 printf("  请输入区间[a,b]的上下限 a,b: ");
 scanf("%ld,%ld",&a,&b);
 n=0;
 for(x=a;x<=b-3;x++)          // 四重循环枚举
 for(y=x+1;y<=b-2;y++)
 for(z=y+1;z<=b-1;z++)
 for(w=z+1;w<=b;w++)
     if(x*x+y*y+z*z==w*w)     // 满足不定方程式时统计输出
       { n++;
         printf("  %ld: %ld,%ld,%ld,%ld  \n",n,x,y,z,w);
        }
 if(n==0)  printf("  方程在该区间内没有解.\n");
 else  printf("  共有%ld 组解.\n",n);
}
```

（2）优化循环参数设计 2

注意到 x＜y＜z＜w≤b，而 x 为最小的，显然 x<sqrt(b*b/3)；

当 x 选取之后，y 为次小的，显然 y<sqrt((b*b-x*x)/2；

当 x,y 选取之后，显然 z<=sqrt(b*b-x*x-y*y)。

优化循环参数后枚举描述：

```
// 优化循环参数设计
main()
{long a,b,n,x,y,z,w;
 printf("  请输入区间[a,b]的上下限 a,b: ");
 scanf("%ld,%ld",&a,&b);
 n=0;
 for(x=a;x<sqrt(b*b/3);x++)
 for(y=x+1;y<sqrt((b*b-x*x)/2);y++)
 for(z=y+1;z<=sqrt(b*b-x*x-y*y);z++)
for(w=z+1;w<=b;w++)
     if(x*x+y*y+z*z==w*w)     // 满足不定方程式时统计
        { n++;
```

```
            printf("  %ld: %ld,%ld,%ld,%ld  \n",n,x,y,z,w);
          }
 if(n==0)  printf("  方程在该区间内没有解.\n");
 else  printf("  共有%ld组解.\n",n);
}
```

（3）枚举结构与参数综合优化设计3

设指定区间为[a,b]，精简w循环，设置三重循环在指定区间内枚举x，y，z（x＜y＜z），应用方程式计算d=x*x+y*y+z*z，w=sqrt(d)：若w>b或w*w≠d，即不满足方程，返回；否则用n统计并输出解。

```
// 精简循环设计
main()
{long a,b,d,n,x,y,z,w;
 printf("  请输入区间[a,b]的上下限a,b: ");
 scanf("%ld,%ld",&a,&b);
 n=0;
 for(x=a;x<sqrt(b*b/3);x++)
 for(y=x+1;y<sqrt((b*b-x*x)/2);y++)
 for(z=y+1;z<=sqrt(b*b-x*x-y*y);z++)
     { d=x*x+y*y+z*z;
       w=(long)sqrt(d);        // w为x、y、z的平方和开平方
       if(w>b || w*w!=d) continue;
       n++;
       printf("  %ld: %ld,%ld,%ld,%ld  \n",n,x,y,z,w);
     }
 if(n==0)  printf("  方程在该区间内没有解.\n");
 else  printf("  共有%ld组解.\n",n);
}
```

（4）三个枚举设计比较

设区间[a,b]中的整数规模为n，对以上三个枚举设计的时间复杂度作粗略分析：

①基本设计1的时间复杂度显然为$O(n^4)$；

②优化参数的设计2仍设置了四重循环，时间复杂度仍为$O(n^4)$，但其系数已大大缩减；

③综合优化设计3只需三重循环实现，其时间复杂度优化为$O(n^3)$。

为了比较以上三个枚举设计的优劣，建议用同一组数据（例如a=1000,b=2013）对以上三个设计进行现场测试比较，实际效果非常明显。由此可见，即使是最基础的枚举算法，其改进与优化的空间也是非常大的。

习题 2

2-1 广义同码小数之和

$$s(d,n)=0.d+0.dd+0.ddd+\cdots+0.dd\cdots d\ (n\text{个}d)$$

其中正整数d、n从键盘输入（1＜d,n＜10000）。

例如：s(301,4)=0.301+0.301301+0.301301301+0.301301301301

依次输入整数d、n（1＜d,n＜10000），输出和s(d,n)的整数部分与小数点后前10位。

2-2　解不等式

设 n 为正整数，解不等式

$$2013 < 1 + \frac{1}{1+1/2} + \frac{1}{1+1/2+1/3} + \cdots + \frac{1}{1+1/2+\cdots+1/n} < 2014$$

2-3　解不定方程

试求解三元不定方程

$$\frac{1}{a} - \frac{1}{b \times c} = \frac{1}{a+b+c}$$

满足条件 $x \leqslant a, b, c \leqslant y$，$b < c$ 的所有正整数解。

2-4　合数世纪探索

定义：一个世纪的 100 个年号中不存在一个素数，即 100 个年号全为合数的世纪称为合数世纪。

试探索第 m（约定 m＜100）个合数世纪。

2-5　分解质因数

对给定区间[m,n]的正整数分解质因数，每一整数表示为质因数从小到大顺序的乘积形式。如果被分解的数本身是素数，则注明为素数。

例如，2012=2*2*503，2011=(素数!)。

2-6　因数比的最大值

设整数 a 的小于其本身的因数之和为 s，定义 p(a)=s/a 为整数 a 的因数比。

事实上，a 为完全数时，p(a)=1。例如，p(6)=1。

有些资料还介绍了因数之和为数本身 2 倍的整数，例如 p(120)=2。

试搜索指定区间[x,y]中因数比最大的整数。

2-7　基于素数代数和的最值

定义和：

$$s(n) = 1 \times 3 + 3 \times 5 + 5 \times 7 + 7 \times 9 \pm \cdots \pm (2n-1) \times (2n+1)$$

和式中第 k 项±(2k-1)*(2k+1)的符号识别：当(2k-1)与(2k+1)中至少有一个素数，取“+”；其余取“-”。例如和式中第 13 项取“-”，即为-25*27。

（1）求 s(2013)。

（2）设 1≤n≤2013，当 n 为多大时，s(n)最大？

（3）设 1≤n≤2013，当 n 为多大时，s(n)最小？

2-8　完美四则运算式

把数字 1～9 这 9 个数字填入以下含加、减、乘、除的综合运算式中的 9 个□中，使得该式成立。

□□×□+□□□÷□-□□=0

要求数字 1～9 这 9 个数字在各式中都出现一次且只出现一次，并约定数字“1”不出现在数式的一位数中（即排除各式中的各个 1 位数为 1 这一平凡情形）。

2-9　区间内的最大连续合数区间

搜索指定区间[c,d]内的最大连续合数区间。

输入：键盘输入区间范围[c,d]。

输出：区间内最多连续合数的个数，连续合数的起始与终止数。

例如：输入 c,d：10,100

最多连续合数的个数为：7

连续合数区间为：[90,96]

2-10　三角形双分线

设一块木板为非等腰△ABC，其三边长分别为 BC=a,CA=b,AB=c，寻求三角形木板上的分割线，把该三角形木板分割为周长相等且面积相等的两块。

试确定双分线的具体位置。

依次输入正整数 a,b,c（约定 a,b,c＜100），输出各分割线的位置（四舍五入，精确到小数点后第 4 位）。