实验一　　统计数字问题

1、问题描述：

一本书的页码从自然数1 开始顺序编码直到自然数 n。书的页码按照通常的习惯编排，每个页码都不含多余的前导数字0。例如，第6 页用数字6 表示，而不是06 或006 等。数字

计数问题要求对给定书的总页码 n，计算出书的全部页码中分别用到多少次数字0，1， 2，…，9。

2、题目分析：

考虑由0,1,2,…,9组成的所有 n 位数。从 n 个0到 n 个9共有个 n 位数，在这些 n 位数中，0,1,2,…，9每个数字使用次数相同，设为。

满足如下递归式：

由此可知，。

据此，可从低位向高位进行统计，再减去多余的0的个数即可。

3、算法设计：

定义数组 a[10]存放0到9这10个数出现的次数，个位为第0位，第 j 位的数字为 r。采用 while 循环从低位向高位统计：

a. 统计从个位算起前 j 位0~9个数；

b. 如果 j+1位为0，去掉第 j+1位补0个数；

c. 统计第 j+1位出现1~(r-1)个数；

d. 统计第 j+1位出现 r 个数。

4、源程序：

```
#include <iostream.h>
int main()
```

```
{
    long int sn[10];
    int i,n,c,k,s,pown;
    for(i=0;i<10;i++)
        sn[i]=0;
 cin>>n;
    for(k=s=0,pown=1;n>0;k++,n/=10,pown*=10)
    {
        c=n%10;
        for(i=0;i<10;i++)
            sn[i]+=c*k*(pown/10);
        for(i=0;i<c;i++)
            sn[i]+=pown;
        sn[c]+=1+s;
        sn[0] -=pown;
        s+=c*pown;
    }
    for(i=0;i<10;i++)
  cout<<sn[i]<<'\n';
}
```

5、算法分析：

函数 count()的复杂度为 O(1)，主函数调用 count()，故该

算法的时间复杂度为 O(1)。

实验二　　最大间隙问题

1、问题描述：

最大间隙问题：给定 n 个实数 x1，x2，...，xn,求这 n 个数在实轴上相邻2 个数之间的最大差值。假设对任何实数的下取整函数耗时 O(1)，设计解最大间隙问题的线性时间算法。对于给定的 n 个实数 x1，x2，...，xn，编程计算它们的最大间隙。

2、题目分析：

考虑到实数在实轴上按大小顺序排列，先对这 n 个数排序，再用后面的数减去前面的数，即可求出相邻两数的差值，找出差值中最大的即为最大差值。

3、算法设计：

a. 用快速排序算法对这 n 个数排序，快速排序算法是基于分治策略的一个排序算

法。其基本思想是，对于输入的子数组 a[p:r]，按以下三个步骤进行排序：

①分解：以 a[p]为基准元素将 a[p:r]划分为3段 a[p:q-1]，a[q]和 a[q+1:r]，使 a[p:q-1]中任何一个元素小于等于 a[p]，而 a[q+1:r]中任何一个元素大于等于 a[q]。下标 q 在

划分过程中确定。

②递归求解：通过递归调用快速排序算法分别对 a[p:q-1]和 a[q+1:r]进行排序。

③合并：由于对 a[p:q-1]和 a[q+1:r]的排序是就地进行的，所以在 a[p:q-1]和

a[q+1:r]都已排好的序后，不需要执行任何计算，a[p:r]就已排好序。

b. 用函数 maxtap()求出最大差值。

4、源程序：

```
#include<iostream.h>
#include<stdio.h>
double a[1000000];
template<class Type>
void swap(Type &x,Type &y)
{
 Type temp=x;
 x=y;
 y=temp;
}
template<class Type>
int Partition(Type *a,int low,int high)
{
```

```
 Type pivotkey;
 int  mid=(low+high)/2;

if((a[low]<a[mid]&&a[mid]<a[high])||(a[low]>a[mid]&&
a[mid]>a[high]))
  swap(a[low],a[mid]);

if((a[low]<a[high]&&a[mid]>a[high])||(a[low]>a[high]
&&a[mid]<a[high]))
  swap(a[low],a[high]);
 pivotkey=a[low];
 int i=low;
 int j=high+1;
 while(true)
 {
  while(a[++i]<pivotkey);
  while(a[--j]>pivotkey);
  if(i>=j) break;
  swap(a[i],a[j]);
 }
 a[low]=a[j];
 a[j]=pivotkey;
```

```
 return j;
}
template<class Type>
void Quicksort(Type *a,int low,int high)
{
 if(low<high)
 {
  int q=Partition(a,low,high);
  Quicksort(a,low,q-1);
  Quicksort(a,q+1,high);
 }
}
template<class Type>
Type maxtap(Type *a,int n)
{
 Type maxtap=0;
 for(int i=0;i<n-1;i++)
  maxtap=(a[i+1]-a[i])>maxtap?(a[i+1]-a[i]):maxtap;
 return maxtap;
}
 main()
{
```

```
int i,n;
scanf("%d",&n);
for(i=0;i<n;i++)
  scanf("%lf",&a[i]);
Quicksort(a,0,n-1);
printf("%lf",maxtap(a,n));
return 0;
```

5、算法分析:

快速排序的运行时间与划分是否对称有关，其最坏情况发生在划分过程产生的两个区域分别包含 n-1个元素和1个元素的时候。由于函数 Partition 的计算时间为 O（n)，所以如果算法 Partition 的每一步都出现这种不对称划分，则其计算时间复杂性 T（ n ）满足:

解此递归方程可得。

在最好情况下，每次划分所取得基准都恰好为中值，即每次划分都产生两个大小为 n / 2 的区域，此时，Partition 的计算时间 T （ n ）满足:

其解为。

可以证明，快速排序算法在平均情况下的时间复杂性也是。

实验三　　众数问题

1、问题描述：

给定含有 n 个元素的多重集合 S，每个元素在 S 中出现的次数称为该元素的重数。多重集 S 中重数最大的元素称为众数。例如，S={1，2，2，2，3，5}。多重集 S 的众数是2，其重数为3。 对于给定的由 n 个自然数组成的多重集 S，编程计算 S 的众数及其重数。

2、题目分析：

题目要求计算集合中重复出现最多的数以及该数出现的次数，若两个数出现的次数相同，则取较小的数。先对集合中的元素从小到大排序，再统计重数，找出最大的重数以及它对应的元素。

3、算法设计：

a. 建立数组 a[n]存储集合中的元素；

b. 调用库函数 sort(a,a+n)对集合中的元素从小到大排序；

c. 采用 for 循环依次对排序后的元素进行重数统计：

①用 m 标记元素，用 s 统计该元数的重数，分别用 max 和 t 标记出现的最大重数和该重数对应的元素。

②若当前元素不同于前一个元素，且前一个元素的重数

s>max,更新 max 和 t。

4、源程序：

```
#include<iostream>
using?namespace?std;
#include<algorithm>
main()
{
 int?n,i,t,m=0,s,max=0,*a;
 scanf("%d",&n);
 a=new?int[n];
 for(i=0;i<n;i++)
  scanf("%d",&a[i]);
 sort(a,a+n);
 for(i=0;i<n;i++)
 {
  if(m!=a[i])
   s=1;
  else
   s++;
  m=a[i];
  if(s>max)
```

```
   {
    t=m;
    max=s;
   }
 }
 printf("%d\n%d\n",t,max);
 return?0;
}
```

5、算法分析：

主函数内调用的库函数 sort(a,a+n)的时间复杂度为，for 循环的时间杂度为 O(n)，故该算法的时间杂度为。

实验四　　半数集问题

1、问题描述：

?给定一个自然数 n，由 n 开始可以依次产生半数集 set(n)中的数如下：(1) n∈set(n)；(2) 在 n 的左边加上一个自然数，但该自然数不能超过最近添加的数的一半；(3) 按此规则进行处理，直到不能再添加自然数为止。

例如：set(6)={6,16,26,126,36,136}，半数集 set(6)中有6个元素。注意半数集
是多重集，对于给定的自然数 n，编程计算半数集 set(n)中的元素个数。

2、题目分析：

题目要求输入有多行，每行给出一个整数 n (0 < n < 1000)，输入以文件结束标志 EOF 结束。半数集 set(n)中的元素个数即为所有半数集 set(j) (j<=n/2)的元素个数之和加1（即将其自身添加到所求元素中)。

3、算法设计：

a. 用数组 count[i]计算 set(i)中的元素个数，即计算所有 j<=i/2的 set(j) 中的元素加

其自身的个数之和;

b. 用数组 count_sum[i]计算所有 j<=i 的 set(j) 中的元素个数之和;

c. 采用 for 循环分别对 count[i]、count_sum[i]进行累加，即可求出半数集 set(n)中的

元素个数。

4、源程序：

```
#include <stdio.h>
int set(int n)
{
 int i,count[1001],count_sum[1001];
 count[1]=1;
 count_sum[1]=1;
```

```
 for(i=2;i<=n;i++)
 {
  count[i]=count_sum[i/2]+1;
 count_sum[i]=count_sum[i-1]+count[i];
 }
 return count[n];
}
main()
{
 int n;
 while(scanf("%d",&n)!=EOF)
  printf("%d\n",set(n));
 return 0;
}
```

5、算法分析：

函数 set(n)的时间复杂度为 O(n)，主函数调用函数 set(n)，故该算法的时间复杂度为 O(n)。

实验五　　集合划分问题

2、题目分析：

题目要求计算 n 个元素的集合共有多少个划分（其中每个划

分都由不同的非空子集组成），n 个元素的集合划分为 m 个块的划分数为 F(n,m)=F(n-1,m-1)+m*F(n-1,m),m 从1到 n 的划分个数相加即可求得总的划分数。

3、算法设计：

a. 这一题的结果数很大，需要使用64位长整型：__int64;

b. 函数 div()采用递归的方式计算 n 个元素的集合划分为 i 个块的划分数：

①div (n,1)=1, div (n,n)=1;

②div (n,i)= div (n-1,i-1)+i*div (n-1,i)

c. 主函数采用 for 循环调用函数 div(n,i)（1≤i≤n）对划分数进行累加统计。

4、源程序：

```
#include<stdio.h>
__int64?div(__int64?n,__int64?i)
{
 if(i==1||i==n)
  return?1;
 else?return?div(n-1,i-1)+i*div(n-1,i);
}
main()
{
```

```
 __int64?i,n,s=0;
 scanf("%I64d",&n);
 for(i=1;i<=n;i++)
  s=s+div(n,i);
 printf("%I64d",s);
 return?0;
}
```

5、算法分析：

函数 div()的时间复杂度为，主函数内 for 循环的时间复杂度为 O(n)，函数 div()嵌套在 for 循环内，故该算法的时间复杂度为。

实验七　　编辑距离问题

1、问题描述：

?设 A 和 B 是2 个字符串。要用最少的字符操作将字符串 A 转换为字符串 B。这

里所说的字符操作包括 (1)删除一个字符；(2)插入一个字符；(3)将一个字符改为另

一个字符。将字符串 A 变换为字符串 B 所用的最少字符操作数称为字符串 A 到 B 的编

辑距离，记为 d(A,B)。试设计一个有效算法，对任给的2 个

字符串 A 和 B，计算出它

们的编辑距离 d(A,B)。

2、题目分析：

题目要求计算两字符串的编辑距离，可以采用动态规划算法求解，由最优子结构性质可建立递归关系如下：

其中数组 d[i][j] 存储长度分别为 i、j 的两字符串的编辑距离；用 edit 标记所比较

的字符是否相同，相同为0，不同为1；用 m、n 存储字符串 a、b 的长度。

3、算法设计：

a. 函数 min()找出三个数中的最小值；

b. 函数 d()计算两字符串的编辑距离：

①用 edit 标记所比较的字符是否相同，相同为0，不同为1；

②分别用 m、n 存储字符串 a、b 的长度，用数组 d[i][j] 存储长度分别为 i、j 的两字符串的编辑距离，问题的最优值记录于 d[n][m]中；

③利用递归式写出计算 d[i][j]的递归算法。

4、源程序：

```
#include <iostream>
using namespace std;
int min(int a, int b, int c)
{
    int temp = (a < b ? a : b);
    return (temp < c? temp : c);
}
int d(char* a, char* b)
{
    int m = strlen(a);
 int n = strlen(b);
 int i,j ,temp;
 int **d;
 d=(int  **)malloc(sizeof(int *)*(m+1));
 for(i=0;i<=m;i++)
 {
       d[i]=(int *)malloc(sizeof(int)*(n+1));
 }
 d[0][0]=0;
 for(i=1;i<=m;i++)
  d[i][0]=i;
 for(j=1;j<=n;j++)
```

```
  d[0][j]=j;
     for( i=1;i<=m;i++)
  {
   for(j=1;j<=n;j++)
    {
     int edit=( a[i-1] == b[j-1] ? 0 : 1);
    d[i][j]=min((d[i-1][j-1]+edit),
 (d[i][j-1]+1),(d[i-1][j]+1));
    }
  }
  temp = d[m][n];
  free(d);
     return temp;
}
main()
{
 char a[10000],b[10000];
 scanf("%s\n%s",a,b);
 printf("%d",d(a,b));
 return 0;
}
```

5、算法分析：

函数 d()采用了双重循环，里层的 for 循环复杂度为 O(n)，外层的 for 循环复杂度为 O(m)，主函数调用了函数 d()，故该算法的时间复杂度为 O(mn)。

实验八　　程序存储问题

1、问题描述：

设有 n 个程序{1,2,…, n }要存放在长度为 L 的磁带上。程序 i 存放在磁带上的长度是 i l ， 1 ≤i ≤n。程序存储问题要求确定这 n 个程序在磁带上的一个存储方案，使得能够在磁带上存储尽可能多的程序。对于给定的 n 个程序存放在磁带上的长度，编程计算磁带上最多可以存储的程序数。

2、题目分析：

题目要求计算给定长度的磁带最多可存储的程序个数，先对程序的长度从小到大排序，再采用贪心算法求解。

3、算法设计：

a. 定义数组 a[n]存储 n 个程序的长度，s 为磁带的长度；

b. 调用库函数 sort(a,a+n)对程序的长度从小到大排序；

c. 函数 most()计算磁带最多可存储的程序数，采用 while 循环依次对排序后的程序

长度进行累加，用 i 计算程序个数，用 sum 计算程序累加长度（初始 i=0,sum=0）：

① sum=sum+a[i];

② 若 sum<=s，i 加1，否则 i 为所求；

③ i=n 时循环结束；

d. 若 while 循环结束时仍有 sum<=s，则 n 为所求。

4、源程序：

```
#include<iostream>
using namespace std;
#include<algorithm>
int a[1000000];
int most(int *a,int n,int s)
{
 int i=0,sum=0;
 while(i<n)
 {
  sum=a[i]+sum;
  if(sum<=s)
   i++;
  else return i;
 }
```

```
 return n;
}
main()
{
 int i,n,s;
 scanf("%d %d\n",&n,&s);
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 sort(a,a+n);
 printf("%d",most(a,n,s));
 return 0;
}
```

5、算法分析：

函数 most()的时间杂度为 O(n)，主函数调用的库函数 sort(a,a+n)的时间复杂度为，且主函数调用函数 most()，故该算法的时间杂度为。

实验九　　最优服务次序问题

1、问题描述：

设有 n 个顾客同时等待一项服务。顾客 i 需要的服务时间为 ti ，1 ≤i≤n。应如何安排 n 个顾客的服务次序才能使平均等待时间达到最小?平均等待时间是 n 个顾客等待服务时间

的总和除以 n。对于给定的 n 个顾客需要的服务时间，编程计算最优服务次序。

2、题目分析：

考虑到每个顾客需要的服务时间已给定，要计算最优服务次序，可以先对顾客需要的服务时间由短到长排序，再采用贪心算法求解。

3、算法设计：

a. 定义数组 a[n]存储 n 个顾客需要的服务时间；

b. 调用库函数 sort(a,a+n)对顾客需要的服务时间由短到长排序；

c. 函数 time()计算最小平均等待时间,采用 while 循环依次对顾客等待服务时间进

行累加，用 a[i]标记第 i+1个顾客需要的服务时间，用 b[i]计算第 i+1个顾客的等待服务

时间,用 t 计算前 i+1个顾客等待服务时间的总和（初始 i=1, t=a[0],b[0]=a[0]）：

① b[i]=a[i]+b[i-1]，t=b[i]+t；

② i++；

③ i=n 时循环结束；

d. t/n 为所求。

4、源程序:

```
#include<iostream>
using namespace std;
#include<algorithm>
int a[1000000],b[1000000];
double time(int *a,int n)
{
 int i=1,j=0;
 double t=a[0];
 b[0]=a[0];
 while(i<n)
  {
   b[i]=a[i]+b[i-1];
   t=b[i]+t;
   i++;
  }
 return t/n;
}
main()
{
 int i,n;
```

```
 scanf("%d",&n);
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 sort(a,a+n);
 printf("%0.2lf",time(a,n));
 return 0;
}
```

5、算法分析：

函数 time()的时间杂度为 O(n)，主函数调用的库函数 sort(a,a+n)的时间复杂度为，且主函数调用函数 time(),故该算法的时间杂度为。

实验十　　汽车加油问题

1、问题描述：

?一辆汽车加满油后可行驶 n 公里。旅途中有若干个加油站。设计一个有效算法，指出应在哪些加油站停靠加油，使沿途加油次数最少。并证明算法能产生一个最优解。对于给定的 n 和 k 个加油站位置，编程计算最少加油次数。

2、题目分析：

题目要求编程计算最少加油次数，若无法到达目的地，则输

出“No Solution”。该题 可以采用贪心算法求解，从出发地开始进行判别：油足够则继续行驶；油不够则加油，计算加油次数；油满仍不够则“No Solution”。

3、算法设计：

a. n 表示汽车加满油后可行驶 n 公里，k 表示出发地与目的地之间有 k 个加油站；

b. 定义数组 a[k+1]存储加油站之间的距离：用 a[i]标记第 i 个加油站与第 i+1个加

油站之间的距离（第0 个加油站为出发地，汽车已加满油；第 k+1 个加油站为目的地）；

c. 用 m 计算加油次数，用 t 标记在未加油的情况下汽车还能行驶 t 公里，采用 for

循环从出发地开始（即 i=0）依次计算加油次数：

① 若 a[i]>n，则输出“No Solution”；

② 若 t<a[i]，则加油一次：t=n,m++；

③ 行驶 a[i]公里后，汽车还能行驶 t- a[i]公里；

④ i=k+1时循环结束；

d. m 即为所求。

4、源程序：

```
#include<stdio.h>
main()
{
 int i,n,k,t,m,a[10000];
 scanf("%d%d",&n,&k);
 for(i=0;i<=k;i++)
  scanf("%d",&a[i]);
 m=0;
 t=n;
 for(i=0;i<=k;i++)
 {
  if(a[i]>n)
  {
   printf("No Solution");
   break;
  }
  else if(t<a[i])
  {
   t=n; m++;
  }
  t=t-a[i];
 }
```

```
 if(i==k+1)
 printf("%d",m);
 return 0;
}
```

5、算法分析：

主函数内 for 循环的时间复杂度为 O（k），故该算法的时间杂度为 O（k）。

实验十一　　工作分配问题

1、问题描述：

??设有 n 件工作分配给 n 个人。将工作 i 分配给第 j 个人所需的费用为 cij 。试设计一个算法，为每一个人都分配1 件不同的工作，并使总费用达到最小。设计一个算法，对于给定的工作费用，计算最佳工作分配方案，使总费用达到最小。

2、题目分析：

考虑到每个人对应的每项工作费用已给定，要计算最佳工作分配方案，使总费用达到最小，可以采用回溯法求解。由于回溯法是对解空间的深度优先搜索，因此此题可用排列树来表示解空间。

3、算法设计：

a. 用 c[i][j]存储将工作 i 分配给第 j 个人所需的费用，用 v[j] 标记第 j 个人是否已分
配工作；

b. 用递归函数 backtrack (i, total)来实现回溯法搜索排列树（形式参数 i 表示递归深
度，n 用来控制递归深度，形式参数 total 表示当前总费用，s 表示当前最优总费用）：

① 若 total>=s，则不是最优解，剪去相应子树，返回到 i-1 层继续执行；

② 若 i >n，则算法搜索到一个叶结点，用 s 对最优解进行记录，返回到 i-1层
继续执行；

③ 采用 for 循环针对 n 个人对工作 i 进行分配(1≤j≤n)：

1> 若 v[j]==1 ，则第 j 个人已分配了工作，找第 j+1个人进行分配；

2> 若 v[j]==0，则将工作 i 分配给第 j 个人（即 v[j]=1 ），对工作 i+1调用递归函数 backtrack(i+1,total+c[i][j])继续进行分配；

3> 函数 backtrack(i+1,total+c[i][j])调用结束后则返回

v[j]=0，将工作 i 对

第 j+1个人进行分配；

4> 当 j>n 时，for 循环结束；

④ 当 i=1时，若已测试完 c[i][j]的所有可选值，外层调用就全部结束；

c. 主函数调用一次 backtrack(1,0)即可完成整个回溯搜索过程，最终得到的 s 即为

所求最小总费用。

4、源程序：

```
#include<stdio.h>
#define MAX 1000;
int n,c[21][21],v[21],s,total;
void backtrack(int i,int total)
{
 int j;
 if(total>=s)
 return;
 if(i>n)
 {
 s=total;
 return;
```

```
 }
 else
 for(j=1;j<=n;j++)
 {
 if(v[j]==1)
  continue;
 if(v[j]==0)
 {
  v[j]=1;  backtrack(i+1,total+c[i][j]);
  v[j]=0;
 }
 }
}
main()
{
 int i,j;
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 for(j=1;j<=n;j++)
 {
 scanf("%d",&c[i][j]);
```

```
 v[j]=0;
 }
 }
 s=MAX;
 backtrack(1,0);
 printf("%d",s);
 return 0;
}
```

5、算法分析：

递归函数 backtrack (i, total)遍历排列树的时间复杂度为 0（n!），主函数调用递归函数 backtrack(1,0)，故该算法的时间复杂度为 0（n!）。

实验十二　　0-1背包问题

1、问题描述：

?给定 n 种物品和一个背包。物品 i 的重量是 wi ，其价值为 vi ，背包的容量为 C。应如何选择装入背包的物品，使得装入背包中物品的总价值最大? 在选择装入背包的物品时，对每种物品 i 只有2 种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i。0-1背包问题形式化描述：给定 C＞0，wi ＞0，vi ＞0，1≤i≤n，

要求 n 元0-1向量(x1 ， x2 ，…， xn)，xi = 0或1，1≤i≤n，使得，而且达到最大。

2、题目分析：

考虑到每种物品只有2 种选择，即装入背包或不装入背包，并且物品数和背包容量已给定，要计算装入背包物品的最大价值和最优装入方案，可用回溯法搜索子集树的算法进行求解。

3、算法设计：

a. 物品有 n 种，背包容量为 C，分别用 p[i]和 w[i]存储第 i 种物品的价值和重量，用
x[i]标记第 i 种物品是否装入背包，用 bestx[i]存储第 i 种物品的最优装载方案；

b. 用递归函数 Backtrack (i,cp,cw)来实现回溯法搜索子集树（形式参数 i 表示递归深
度，n 用来控制递归深度，形式参数 cp 和 cw 表示当前总价值和总重量，bestp 表示当前
最优总价值)：

① 若 i >n，则算法搜索到一个叶结点，判断当前总价值是否最优：

1> 若 cp>bestp，更新当前最优总价值为当前总价值（即 bestp=cp)，更新

装载方案（即 bestx[i]=x[i](1≤i≤n)）;

② 采用 for 循环对物品 i 装与不装两种情况进行讨论（0≤j≤1）：

1> x[i]=j;

2> 若总重量不大于背包容量（即 cw+x[i]*w[i]<=c），则更新当前总价值和总重量（即 cw+=w[i]*x[i],cp+=p[i]*x[i]），对物品 i+1调用递归函数 Backtrack(i+1,cp,cw) 继续进行装载;

3> 函数 Backtrack(i+1,cp,cw)调用结束后则返回当前总价值和总重量（即 cw-=w[i]*x[i],cp-=p[i]*x[i]）;

4> 当 j>1时，for 循环结束;

③ 当 i=1时，若已测试完所有装载方案，外层调用就全部结束;

c. 主函数调用一次 backtrack(1,0,0)即可完成整个回溯搜索过程，最终得到的 bestp 和 bestx[i]即为所求最大总价值和最优装载方案。

4、源程序：

```
#include<stdio.h>
int n,c,bestp;
```

```
int p[10000],w[10000],x[10000],bestx[10000];
void Backtrack(int i,int cp,int cw)
{
 int j;
 if(i>n)
 {
 if(cp>bestp)
 {
 bestp=cp;
 for(i=0;i<=n;i++)
 bestx[i]=x[i];
 }
 }
 else
 for(j=0;j<=1;j++)
 {
 x[i]=j;
 if(cw+x[i]*w[i]<=c)
 {
  cw+=w[i]*x[i];
  cp+=p[i]*x[i];
  Backtrack(i+1,cp,cw);
```

```
    cw-=w[i]*x[i];
    cp-=p[i]*x[i];
   }
   }
}
main()
 {
  int i;
  bestp=0;
  scanf("%d%d",&n,&c);
  for(i=1;i<=n;i++)
  scanf("%d",&p[i]);
  for(i=1;i<=n;i++)
  scanf("%d",&w[i]);
  Backtrack(1,0,0);
  printf("Optimal value is\n");
  printf("%d\n",bestp);
  for(i=1;i<=n;i++)
  printf("%d ",bestx[i]);
  return 0;
}
```

5、算法分析：

递归函数 Backtrack (i,cp,cw)遍历子集树的时间复杂度为，主函数调用递归函数 Backtrack(1,0,0)，故该算法的时间复杂度为。

实验十三　　最小重量机器设计问题

1、问题描述：

？　　　　设某一机器由 n 个部件组成，每一种部件都可以从 m 个不同的供应商处购得。设 wij 是从供应商 j 处购得的部件 i 的重量，cij 是相应的价格，给出总价格不超过 d 的最小重量机器设计。

2、题目分析：

考虑到从每一处供应商购得每一种部件的重量和相应价格以及总价格的上限已给定，要设计最小重量机器方案计算最小重量，可用回溯法搜索排列树的算法进行求解。

3、算法设计：

a. 部件有 n 个，供应商有 m 个，分别用 w[i][j]和 c[i][j]存储从供应商 j 处购得的部件 i 的重量和相应价格，d 为总价格的上限。

b. 用递归函数 Knapsack(i,cs,ws)来实现回溯法搜索排列树（形式参数 i 表示递归深度，n 用来控制递归深度，形式参数 cs 和 ws 表示当前总价格

和总重量，bestw 表示当前

最优总重量)：

① 若 cs>d，则为不可行解，剪去相应子树，返回到 i-1层继续执行；

② 若 ws>=bestw，则不是最优解，剪去相应子树，返回到 i-1层继续执行；

③ 若 i >n，则算法搜索到一个叶结点，用 bestw 对最优解进行记录，返回到

i-1层继续执行；

④ 采用 for 循环对部件 i 从 m 个不同的供应商购得的情况进行讨论(1≤j≤m)：

1> 调用递归函 Knapsack(i+1,cs+c[i][j],ws+w[i][j])对部件 i+1进行购买；

2> 当 j>m 时 for 循环结束；

⑤ 当 i=1时，若已测试完所有购买方案，外层调用就全部结束；

c. 主函数调用一次 Knapsack(1,0,0)即可完成整个回溯搜索过程，最终得到的 bestw

即为所求最小总重量。

4、源程序：

```
#include<stdio.h>
```

```
#define MIN 1000
int n,m,d,cs,ws,bestw;
int w[100][100],c[100][100];
void Knapsack(int i,int cs,int ws)
{
 int j;
 if(cs>d)
  return;
 else if(ws>=bestw)
  return;
 else if(i>n)
 {
  bestw=ws;
  return;
 }
 else
 {
  for(j=1;j<=m;j++)
Knapsack(i+1,cs+c[i][j],ws+w[i][j]);
 }
}
main()
```

```
{
 int i,j;
 scanf("%d%d%d",&n,&m,&d);
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=m;j++)
   scanf("%d",&c[i][j]);
 }
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=m;j++)
   scanf("%d",&w[i][j]);
 }
 bestw=MIN;
 Knapsack(1,0,0);
 if(bestw==MIN)
  printf("No Solution!");
 else
  printf("%d",bestw);
 return 0;
}
```

5、算法分析：

递归函数 Knapsack(i,cs,ws)遍历排列树的时间复杂度为 O（n!）；主函数的双重 for

循环的时间复杂度为 O(mn)， 且主函数调用递归函数 Knapsack(1,0,0)，故该算法的时

间复杂度为 O（n!）。

实验十四　　最小权顶点覆盖问题

1、问题描述：

？　　　　给定一个赋权无向图 G=(V,E)，每个顶点 v∈V 都有一个权值 w(v)。如果 U 包含于 V，且对于(u,v)∈E 有 u∈U 且 v∈V-U，则有 v∈K. 如：U = {1}，若有边（1，2）， 则有 2属于 K. 若有集合 U 包含于 V 使得 U + K = V, 就称 U 为图 G 的一个顶点覆盖。G 的最小权顶点覆盖是指 G 中所含顶点权之和最小的顶点覆盖。

2、题目分析：

考虑到每个顶点有在与不在顶点覆盖集中两种情况，并且每个顶点的权值已给定，

要计算最小权顶点覆盖的顶点权之和，可用回溯法搜索子集树的算法进行求解。

3、算法设计：

a. 给定的图 G 有 n 个顶点和 m 条边，用 w[i]存储顶点 i 的权值，用 e[i][j]标记两顶

点为 i 和 j 的边是否存在，用 c[i]标记顶点 i 是否在顶点覆盖集中；

b. 用函数 cover()判断图 G 是否被顶点覆盖（用 t 标记）：

① 初始 t=0；

② 采用 while 循环对每个顶点 i（1≤i≤n）进行讨论：

1> 若顶点 i 不在顶点覆盖集中（即 c[i]==0），则查找与之有边连接的顶点 j（即 e[i][j]==1），判断所有顶点 j：

若存在顶点 j 在顶点覆盖集中（即 c[j]==0），则 t=1；

若所有顶点 j 都不在顶点覆盖集中（即 t==0），则图 G 未被顶点

覆盖（return 0）；

2> 当 i>n 时循环结束；

③ return 1；

c. 用递归函数 cut(i, s) 来实现回溯法搜索子集树（形式参数 i 表示递归深度，n 用

来控制递归深度，形式参数 s 表示当前顶点权之和）：

① 若 s>=bestw，则不是最优解，剪去相应子树，返回到 i-1 层继续执行；

② 若 i >n，则算法搜索到一个叶结点，调用函数 cover() 对图 G 进行判断：

若 cover()为真，则用 bestw 对最优解进行记录，返回到 i-1 层继续执行；

③ 对顶点 i 分在与不在顶点覆盖集中两种情况进行讨论：

1> 若顶点 i 不在顶点覆盖集中（即 c[i]==0），则调用函数 cut(i+1,s)；

2> 若顶点 i 在顶点覆盖集中（即 c[i]==1），则调用函数 cut(i+1,s+w[i])；

④ 当 i=1时，若已测试完所有顶点覆盖方案，外层调用就全部结束；

d. 主函数调用一次 cut(1,0)即可完成整个回溯搜索过程，最终得到的 bestw 即为所

求最小顶点权之和。

4、源程序：

```
#include<stdio.h>
#define MIN 100000
int m,n,u,v,bestw;
int e[100][100],w[100],c[100];
int cover()
{
 int i,j,t;
 i=1;
```

```
while (i<=n)
{
t=0;
if(c[i]==0)
{
j=1;
while(j<i)
{
if(e[j][i]==1&&c[j]==1)
 t=1;
 j++;
}
j++;
while(j<=n)
{
if(e[i][j]==1&&c[j]==1)
 t=1;
 j++;
}
if(t==0)
 return 0;
}
```

```
 i++;
 }
 return 1;
}
void cut(int i,int s)
{
 if(s>=bestw)
 return;
 if(i>n)
  {
   if(cover())
   bestw=s;
 return;
  }
 c[i]=0;
 cut(i+1,s);
 c[i]=1;
 cut(i+1,s+w[i]);
}
main()
{
 int i,j,k;
```

```
 scanf("%d%d",&n,&m);
 for(i=1;i<=n;i++)
 {
 scanf("%d",&w[i]);
 c[i]=0;

 }
 for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
 e[i][j]=0;
 for(k=1;k<=m;k++)
 {
 scanf("%d%d",&u,&v);
 e[u][v]=1;
 }
 bestw=MIN;
   cut(1,0);
 printf("%d",bestw);
 return 0;
}
```

5、算法分析：

函数cover()的双重while循环的时间复杂度为，递归函数cut(i, s)遍历子集
树的时间复杂度为，且函数cover()嵌套在函数cut(i, s)内，所以递归函数cut(i, s)
的时间复杂度为；主函数的双重for循环的复杂度为，且主函数调用
递归函数cut(1, 0)，故该算法的时间复杂度为。

实验十五　　集合相等问题

1、问题描述：

？　　给定2个集合S和T，试设计一个判定S和 T是否相等的蒙特卡罗算法。

2、题目分析：

题目要求用蒙特卡罗算法进行求解，随机选择集合S中的元素与集合T中的元素进行比较，若随机选择很多次都能从集合T中找到与之对应的相等，则集合S和T相等。

3、算法设计：

a. 蒙特卡罗算法Majority对从集合S中随机选择的数组元素x,测试集合T中是否有
与之相等的元素：若算法返回的结果为true，则集合T中有

与 x 相等的元素；若返回 false，

则集合 T 中不存在与 x 相等的元素，因而集合 S 和 T 不相等。

b. 算法 MajorityMC 重复调用次算法 Majority，调用过程中若 Majority

返回 true 则继续调用，否则可以判定集合 S 和 T 不相等（MajorityMC 返回 false）。

c. 主函数调用算法 MajorityMC：若返回 true，则集合 T 和集合 S 相等；若返回 false，

则集合 S 和 T 不相等。

4、源程序：

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <time.h>
bool Majority(int *S,int *T,int n)
{
 int i,j,x;
 bool k;
 time_t t;
 //利用随机函数 rand 求0—n-1的随机数 i
```

```
    srand((unsigned)time(&t));
 i=rand()%(n)-1;
 x=S[i];
 k=0;
 for(j=0;j<n;j++)
 {
  if(T[j]==x)
   k=1;
 }
 return k;
}
bool MajorityMC(int *S,int *T,int n)
{//重复次调用算法Majority
 int k;
 double e;
 e=0.00001;
//利用函数ceil求
 k=(int)ceil(log(1/e));
 for(int i=1;i<=k;i++)
 {
  if(!Majority(S,T,n))
   return 0;
```

```
 }
 return 1;
}
main()
{
 int i,n;
 int S[100000],T[100000];
 scanf("%d",&n);
 for(i=0;i<n;i++)
  scanf("%d",&S[i]);
 for(i=0;i<n;i++)
  scanf("%d",&T[i]);
 if(MajorityMC(S,T,n))
  printf("YES");
 else printf("NO");
 return 0;
}
```

5、算法分析：

蒙特卡罗算法 Majority 的时间复杂度为 O(n)；算法 MajorityMC 重复调用

次算法 Majority，时间复杂度为；主函数调用算法 MajorityMC，故该算

法的时间复杂度为。

实验十六　　战车问题

1、问题描述：

？　　　　在 n×n 格的棋盘上放置彼此不受攻击的车。按照国际象棋的规则，车可以攻击与之处在同一行或同一列上的棋子。在棋盘上的若干个格中设置了堡垒，战车无法穿越堡垒攻击别的战车。对于给定的设置了堡垒的 n×n 格棋盘，设计一个概率算法，在棋盘上放置尽可能多彼此不受攻击的车。

2、题目分析：

从第一行开始随机产生一个位置，看战车在该位置上能否放置，分三种情况讨论：

a. 该随机位置对应在棋盘上是一个堡垒，则不能放置;

b. 该位置与前面放置的战车相冲突，即在受前面放置战车攻击的位置上，则

也不能放置;

c. 该随机位置不受攻击也不是堡垒，则可以放置；

如果不能放置,则重新产生一个随机位置再判断，如果可以放置，则放在该位置上并记录战车个数和该战车可能攻击的位置。以这种方法从第一行开始放置，第一行不能放置战车

后再放第二行，直至第 n 行结束。

3、算法设计：

a. 建立随机数类 CRandomNumber；

b. 函数 CheckPlace 判断是否可以放入战车，同时查看所放战车的攻击位置；

c. 函数 MaxChes 产生随机位置，放置并记录战车数。

4、源程序：

```
#include<iostream.h>
#include<fstream.h>
#include<time.h>
#include<stdlib.h>
const?unsigned?long?multiplier=1194211693L;
const?unsigned?long?adder=12345L;
class?CRandomNum???
{
private:
 unsigned?long?nSeed;
public:
 CRandomNum(unsigned?long?s=0);????
unsigned?short?Random(unsigned?long?n);?
?};
```

```
CRandomNum::CRandomNum(unsigned long?s)
{
 if(s==0)
  nSeed=time(0);
 else?
  nSeed=s;
}
unsigned?short?CRandomNum::Random(unsigned?long?n)
{
 nSeed=multiplier*nSeed+adder;
return?(unsigned?short)((nSeed>>16)%n);
}
bool?CheckPlaceChe(int?**b,char?**a,int?x,int?y,int?
n)??
{
 if(b[x][y]==1)return?false;
 else
 {
 b[x][y]=1;
 if((y>=1)&&(y<=n))
 {??//上搜索?
 for(int?i=y-1;i>=0;i--)
```

```
{
if?((b[x][i]==1)?&&(a[x][i]=='X'))
?break;
else? ?b[x][i]=1;
}
}

?????????if((y<=n-1)&&(y>=0))
{??//下搜索
for(int?i=y+1;i<n;i++)
{
????if((b[x][i]==1)&&(a[x][i]=='X'))
?break;
?? else  b[x][i]=1;
}
}
???? if((x>=1)&&(x<=n))
{???//左搜索
for(int?j=x-1;j>=0;j--)
{
if((b[j][y]==1)?&&(a[j][y]=='X'))
break;
```

```
 else  b[j][y]=1;
 }
 }
 if((x>=0)&&(x<=n-1))
 {??//右搜索
 for(int?j=x+1;j<n;j++)
 {
 ???if((b[j][y]==1)?&&(a[j][y]=='X'))
 ??break;
 ?else ??b[j][y]=1;
 }
 }
 return?true;
 }
 }
int?MaxChes(int?**b,char**a,int?*x,int?n)
{
 int?max1=0;
 static?CRandomNum?rnd;
 for(int?i=0;i<n;i++)????
 {
 bool?flag=false;
```

```
 do{
????????????for(int?j=0;j<n;j++)
 {
 ????if(b[i][j]==0){?flag=true;break;}
else?{flag=false;continue;}
 }
 if(flag)
 {
 ?x[i]=rnd.Random(n);?
 ??if(CheckPlaceChe(b,a,i,x[i],n))
 ?++max1;
 } ?????
 }while(flag);
???   }
 return?max1;
 }
int??main()
{
 int?n,i,j;
 cin>>n;
 char?**a=new?char*[n];???
 for(i=0;i<n;i++)
```

```
 a[i]=new?char[n];
 for(i=0;i<n;i++)
 for(j=0;j<n;j++)
 ??cin>>a[i][j];
 ???int?**b=new?int*[n];????
 for(i=0;i<n;i++)
 b[i]=new?int[n];
 int?max=0;
 int?*x=new?int[n];
 for(i=0;i<n;i++)
 x[i]=0;
????
 int?t=0;
 while(t<15000)
 {
????????for(i=0;i<n;i++)
 {??for(int?j=0;j<n;j++)
 ?{
 ?if(a[i][j]=='X')
 b[i][j]=1;
 else
 b[i][j]=0;
```

```
 ?  }
 }
???
 ???int??max2=MaxChes(b,a,x,n);
 ???if(max<max2)?max=max2;
 ?t++;
 }
 cout<<max;
 delete?[]x;
 for(i=0;i<n;i++)????
 delete[]a[i];
 delete?[]a;
 ????for(i=0;i<n;i++)????
 delete[]b[i];
 delete?[]b;
 return?0;
 }
```

5、算法分析：

算法的时间主要在判断是否可以放入战车和产生随机位置上，若重复的次数为 K，则时间复杂度为 O(Kn2)。