算法设计与分析

计科2102 梅炳寅 202108010206

第一次作业

<1> 算法分析题1-6

（1）

F(N)	G(N)	ANSWER	备注
f(n)=log(n^2)	g(n)=logn+5	f(n)=θ(g(n))	同阶
f(n)=log(n^2)	g(n)=sqrt(n)	f(n)=O(g(n))	sqrt(n)更高阶
f(n)=n	g(n)=(log^2)n	f(n)=Ω(g(n))	n更高阶
f(n)=nlogn+n	g(n)=logn	f(n)=Ω(g(n))	nlogn+n更高阶
f(n)=10	g(n)=log10	f(n)=θ(g(n))	同阶
f(n)=(log^2)n	g(n)=logn	f(n)=Ω(g(n))	(log^2)n更高阶
f(n)=2^n	g(n)=100n^2	f(n)=Ω(g(n))	2^n更高阶
f(n)=2^n	g(n)=3^n	f(n)=O(g(n))	3^n更高阶

<2> 算法实现题1-1 统计数字问题

洛谷P2602 [ZJOI2010] 数字计数的简化版

▲解题思路

参考思路：https://acmachineoier.blog.luogu.org/solution-p2602

数据量较大，显然不好直接暴力。

分别统计十个数字出现的个数，对于每个数字，再分别统计其在每一位上的出现情况。综合来看，对于每个数字在每一位上出现情况的统计可以称为一次操作。用循环来遍历数字i，用模板变量m（m为1或10的倍数）来控制操作的位置。

对于一次操作，使用取模和整除将原数n分割成三个部分a/b/c。b为操作的位置，a为操作位置之前的数字，c为操作位置之后的数字。例如372869在m=100时，分割的结果为a=372，b=8，c=69。

在i不为0的情况下：

下面我们将用数字n=372869举例。

若b比i大，也就是说这个位置取i之后，前面可以取到a，后面是可以任意取数字的。在m=100时，b=8，若i取6，那么在8前面我可以从0取到372这一共373个数，后面我可以从00到99共100（即m）个。所以这种情况下结果应该有`(a + 1) * m`。

若b和i相等，也就是说这个位置取i之后，如果前面取到a-1，后面是可以任意取数字的，但前面取到a时，后面有一部分是不可以的。若i取8，那么前面从0到371这一共372个数都是没有问题的，此时后面可以从00到99共100（即m）个，这部分的结果是`a*m`。但是前面取372时，后面就只能从00到69共70（即c+1）个了，这部分的结果是c+1。合起来一共是`a*m+c+1`。

若b比i小，也就是说前面只能取到比a小，后面可以随便取。若i取9，那么前面只能从0取到371，372都不行。但是好在这样取时，后面是可以从00取到99的。故`a*m`。

在i为0的情况下：

下面我们用数字372069举例。

若b也为0，即m=100，b=0，i=0的这个操作。前面可以从1取到371，此时后面可以随便取。注意为什么前面不能取0！前面取372的时候，后面只能取c+1个。故`(a - 1) * m + c + 1`

若b不为0，假设m=1000，b=2，i=0的这个操作。前面其实是可以从0取到36的，此时后面可以随便取。但是由于b>0=i，所以若前面取到37，后面肯定不符合了。故`a*m`

最后只要把上述情况分别讨论，结果总和就可以了。

▲代码

```cpp
#include<cstdio>
using namespace std;

int main()
{
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
    long long n;
    scanf("%lld",&n);
    int ans[10];
    for (int i = 0; i <= 9; i++)
        ans[i] = 0;
```

```
    for (int i = 0; i <= 9; i++)
    {
        long long m = 1;
        while (m < n)
        {
            long long a = n / (m * 10);
            long long b = n / m % 10;
            long long c = n % m;
            if (i)
            {
                if (b > i)
                    ans[i] += ((a + 1) * m);
                else if (b == i)
                    ans[i] += (a * m + c + 1);
                else if (b < i)
                    ans[i] += (a * m);
            }
            else
            {
                if (b == 0)
                    ans[i] += ((a - 1) * m + c + 1);
                else if (b)
                    ans[i] += (a * m);
            }
            m *= 10;
        }
    }
    for (int i = 0; i <= 9; i++)
        printf("%d\n",ans[i]);
}
```

▲验证

测试样例

```
input 99
output 9 20 20 20 20 20 20 20 20 20
input 11
output 1 4 1 1 1 1 1 1 1 1
```

<3> 算法实现题1-2 字典序问题

参考思路：https://zhuanlan.zhihu.com/p/112912233

▲解题思路

- 方法1： 首先可以简单的知道，要计算以一个确定的首字母字母i在确定的层数l中，要判断共有多少个这样的升序字符串，就要对第二个字母，进行遍历，从i+1遍历到26，得到其对应的确定层数l-1对应的升序字符串，再配合上第一个确定的首字母i，即得到我们所要求的字符串。 例如：对以b开头的有3层的字符串，要计算其总个数。第二个字母一定要比b大，那么可能是$c, d, \ldots$，把这里面的每个可能字母都进行遍历，得到其对应的2层字符串的个数，然后进行累加求和，就可以得到所求的总个数。 因此可以得到递推关系式，若首字母i在确定的层数l中共有$f(i,l)$个升序字符串，那么$f(i,l)=\sum_{j=i+1}^{26} f(j,l-1)$,又因为显然对于任意的$f(i,1)=1$，因此，可以进行递归求解$f(i,l)$ 既然能够确定$f(i,l)$的值，那么要求任意一个字符串的次序，就只要计算出，在它之前共有多少个，再+1就是所要求的数目。设其深度为$depth$，然后其字母分别为$a_0, a_1, \ldots, a_{depth-1}$，那么显然比$depth$小的层数都在该字符串之前，因此要加上$\sum_{l=1}^{i<depth} \sum_{i=1}^{i\leq 26} f(i,l)$；同时还有首字母比$a_0$小的，层数也为$depth$的字符串，其数目为$\sum_{i=1}^{i<a_0} f(i,depth)$；最后对于首字母为$a_0$的深度为$depth$的字符串，还要进行排序，其数目为$\sum_{i=1}^{i<depth} \sum_{j=a_{i-1}+1}^{j<a_i} f(j,depth-i)$ 故总数目

$$sum=\sum_{l=1}^{i<depth} \sum_{i=1}^{i\leq 26} f(i,l)+\sum_{i=1}^{i<a_0} f(i,depth)+\sum_{i=1}^{i<depth} \sum_{j=a_{i-1}+1}^{j<a_i} f(j,depth-i)$$

▲代码

```
#include <cstdio>
#include <iostream>
#include <string>
using namespace std;
int n;
string s[1000];
int ans[1000];
int f[27][7];
int g[7];

int func(int i, int k)
{
    if (k == 1)
    {
        f[i][k] = 1;
```

```
16          return 1;
17      }
18      int total = 0;
19      for (int j = i + 1; j <= 26; j++)
20          total += (f[j][k - 1]) ? f[j][k - 1] : func(j, k - 1);
21          //如果已经存在，就不用再算一遍了，记忆化搜索
22      f[i][k] = total;
23      return total;
24  }
25
26  int main()
27  {
28      // build table
29      //初始化
30      for (int i = 0; i < 27; i++)
31          for (int j = 0; j < 7; j++)
32              f[i][j] = 0;
33      for (int i = 0; i < 7; i++)
34          g[i] = 0;
35
36      //填表
37      for (int i = 1; i <= 26; i++)
38      {
39          for (int j = 1; j <= 6; j++)
40              if (f[i][j] == 0)
41                  func(i, j);
42      }
43      // 计算每种长度字符串的个数放在g[i]
44      for (int k = 1; k <= 6; k++)
45          for (int i = 1; i <= 26; i++)
46              g[k] += f[i][k];
47
48      // check
49      /*
50      for (int i = 1; i <= 26; i++)
51      {
52          for (int j = 1; j <= 6; j++)
53              cout << f[i][j] << " ";
54          cout << endl;
55      }
56      cout << endl;
57      for (int i = 1; i <= 6; i++)
```

```
58          cout << g[i] << " ";
59      */
60
61      // process problem
62      cin >> n;
63      for (int i = 0; i < n; i++)
64          ans[i] = 0;
65      for (int i = 0; i < n; i++)
66          cin >> s[i];
67      for (int i = 0; i < n; i++)
68      {
69          int length = s[i].size();
70          int total = 0;
71          int a0 = s[i][0] - 'a' + 1;
72
73          // 比该字符串长度小的字符串个数直接加上（之前计算过每种长度字符串的个数放在g[i]）
74          for (int j = 1; j < length; j++)
75              total += g[j];
76
77          // 与该字符串长度相同，但是首字母不一样的，加上
78          // 例如该串首字母d开头，则前面可能有a,b,c开头的，这些需要加上去
79          for (int j = 1; j < a0; j++)
80              total += f[j][length];
81
82          // 与该字符串相同长度，相同首字母，但每个位置都有可能因为不是紧挨着的而留出别的可能
83          // 例如abcde和afghi和afjkl，abcde和afghi之间在第二个位置上留有可能，要考虑b/c/d/e的可能，
84          // 故要加上f[2][4],f[3][4],f[4][4],f[5][4],是length-k而不是k因为第二位置引导4位长度的字符串。
85          // afghi只有第二个位置存在问题，而afjkl在第二个位置和第三个位置都存在问题，
86          // afjkl还需要额外加上f[7][3],f[8][3],f[9][3]这三个，即ghi在第三个位置产生的问题
87          // 后面的位置也是同理可得
88          for (int k = 1; k < length; k++)
89          {
90              for (int j = (s[i][k - 1] - 'a' + 1) + 1; j < (s[i][k] - 'a' + 1); j++)
91                  total += f[j][length - k];
92          }
```

```
93
94          //前面统计的是到目前为止已有的序号，求自身的序号要+1
95          total += 1;
96
97          ans[i] = total;
98      }
99      for (int i = 0; i < n; i++)
100     {
101         cout << ans[i] << " ";
102     }
103 }
```

▲验证

测试样例

```
1  input:
2  5
3  ab
4  ac
5  abcde
6  xyz
7  abcdef
8
9  output:
10 27 28 17902 2951 83682
```

经过参考代码对拍发现结果完全一致

第二次作业【分治算法】

<1>算法实现题 2-2 马的Hamilton周游路线问题

▲题目重述

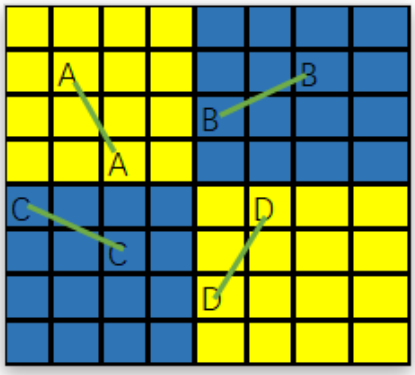
`8*8`的国际象棋棋盘上的一只马，恰好走过除起点外的其他63个位置各一次，最后回到起点。这条路线称为马的一条Hamiltion周游路线。对于给定的`m*n`的国际象棋盘，m,n均为大于5的偶数，且|m-n|<=2,试分析分治算法找出马的一条Hamilton周游路线。

算法设计：对于给定的偶数m,n>=6,且|m-n|<=2,计算m*n的国际象棋盘上马的一条Hamilton周游路线。
数据输入：由文件input.txt给出输入数据。第1行有两个正整数m和n,表示给固定的国际象棋棋盘有m行，每行有n个格子组成
结果输出：将计算的马的Hamilton周游路线用下面两种表达方式输出到文件output.txt.

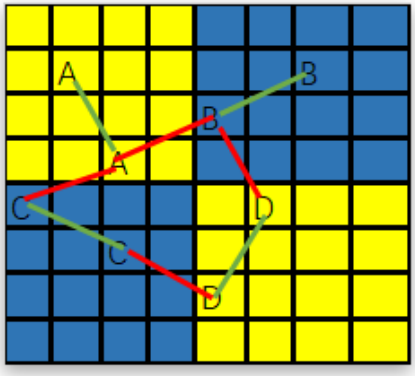
▲解题思路

首先打表数据：*66,68,88,810,1010*,10*12的结构化棋盘，其中是小规模的子结构，其中包含了该规模的棋盘走法。

分治划分规模：将棋盘尽可能平均地分割成4块。当m,n=4k时，分割为2个2k；当m,n=4k+2时，分割为1个2k和1个2k+2，如图：



合并为:



▲代码

```
1  #include <iostream>
2  #include <fstream>
```

```
using namespace std;
struct grid
{
    //表示坐标
    int x;
    int y;
};
class Knight{
    public:
        Knight(int m,int n);
        ~Knight(){};
        void out0(int m,int n,ofstream &out);
        grid
*b66,*b68,*b86,*b88,*b810,*b108,*b1010,*b1012,*b1210,link[20]
[20];
        int m,n;
        int pos(int x,int y,int col);
        void step(int m,int n,int a[20][20],grid *b);
        void build(int m,int n,int offx,int offy,int col,grid
*b);
        void base0(int mm,int nn,int offx,int offy);
        bool comp(int mm,int nn,int offx,int offy);
};
Knight::Knight(int mm,int nn){
    int i,j,a[20][20];
    m=mm;
    n=nn;
    b66=new grid[36];b68=new grid[48];
    b86=new grid[48];b88=new grid[64];
    b810=new grid[80];b108=new grid[80];
    b1010=new grid[100];b1012=new grid[120];
    b1210=new grid[120];
    //cout<<"6*6"<<"\n";
    ifstream in0("66.txt",ios::in); //利用文件流读取数据
    ifstream in1("68.txt",ios::in); //利用文件流读取数据
    ifstream in2("88.txt",ios::in); //利用文件流读取数据
    ifstream in3("810.txt",ios::in); //利用文件流读取数据
    ifstream in4("1010.txt",ios::in); //利用文件流读取数据
    ifstream in5("1012.txt",ios::in); //利用文件流读取数据
    for(i=0;i<6;i++)
    {
        for(j=0;j<6;j++)
```

```
        {
            in0>>a[i][j];
        }
    }
    step(6,6,a,b66);
    //cout<<"6*8"<<"\n";
    for(i=0;i<6;i++)
    {
        for(j=0;j<8;j++)
        {
            in1>>a[i][j];
        }
    }
    step(6,8,a,b68);
    step(8,6,a,b86);
    //cout<<"8*8"<<"\n";
    for(i=0;i<8;i++)
    {
        for(j=0;j<8;j++)
        {
            in2>>a[i][j];
        }
    }
    step(8,8,a,b88);
    for(i=0;i<8;i++)
    {
        for(j=0;j<10;j++)
        {
            in3>>a[i][j];
        }
    }
    step(8,10,a,b810);
    step(10,8,a,b108);
    //cout<<"10*10"<<"\n";
    for(i=0;i<10;i++)
    {
        for(j=0;j<10;j++)
        {
            in4>>a[i][j];
        }
    }
    step(10,10,a,b1010);
```

```
    for(i=0;i<10;i++)
    {
        for(j=0;j<12;j++)
        {
            in5>>a[i][j];
        }
    }
    step(10,12,a,b1012);
    step(12,10,a,b1210);
}
//将读入的基础棋盘的数据转换为网格数据
void Knight::step(int m,int n,int a[20][20],grid *b)
{
    int i,j,k=m*n;
    if(m<n)
    {
        for(i=0;i<m;i++)
        {
            for(j=0;j<n;j++)
            {
                int p=a[i][j]-1;
                b[p].x=i;
                b[p].y=j;
            }
        }
    }
    else
    {
        for(i=0;i<m;i++)
        {
            for(j=0;j<n;j++)
            {
                int p=a[j][i]-1;
                b[p].x=i;
                b[p].y=j;
            }
        }
    }
}
//分治法的主体部分
bool Knight::comp(int mm,int nn,int offx,int offy)
{
```

```
    int mm1,mm2,nn1,nn2;
    int x[8],y[8],p[8];
    if(mm%2||nn%2||mm-nn>2||nn-mm>2||mm<6||nn<6) return 1;
    if(mm<12||nn<12)
    {
        base0(mm,nn,offx,offy);
        return 0;
    }
    mm1=mm/2;
    if(mm%4>0)
    {
        mm1--;
    }
    mm2=mm-mm1;
    nn1=nn/2;
    if(nn%4>0)
    {
        nn1--;
    }
    nn2=nn-nn1;
    //分割
    comp(mm1,nn1,offx,offy);//左上角
    comp(mm1,nn2,offx,offy+nn1);//右上角
    comp(mm2,nn1,offx+mm1,offy);//左下角
    comp(mm2,nn2,offx+mm1,offy+nn1);//右下角
    //合并
    x[0]=offx+mm1-1; y[0]=offy+nn1-3;
    x[1]=x[0]-1;     y[1]=y[0]+2;
    x[2]=x[1]-1;     y[2]=y[1]+2;
    x[3]=x[2]+2;     y[3]=y[2]-1;
    x[4]=x[3]+1;     y[4]=y[3]+2;
    x[5]=x[4]+1;     y[5]=y[4]-2;
    x[6]=x[5]+1;     y[6]=y[5]-2;
    x[7]=x[6]-2;     y[7]=y[6]+1;
    for(int i=0;i<8;i++)
    {
        p[i]=pos(x[i],y[i],n);
    }
    for(int i=1;i<8;i+=2)
    {
        int j1=(i+1)%8,j2=(i+2)%8;
        if(link[x[i]][y[i]].x==p[i-1])
```

```
            link[x[i]][y[i]].x=p[j1];
        else
            link[x[i]][y[i]].y=p[j1];
        if(link[x[j1]][y[j1]].x==p[j2])
            link[x[j1]][y[j1]].x=p[i];
        else
            link[x[j1]][y[j1]].y=p[i];
    }
    return 0;
}
////根据基础解构造子棋盘的Hamilton回路
void Knight::base0(int mm,int nn,int offx,int offy)
{
    if(mm==6&&nn==6)
        build(mm,nn,offx,offy,n,b66);
    if(mm==6&&nn==8)
        build(mm,nn,offx,offy,n,b68);
    if(mm==8&&nn==6)
        build(mm,nn,offx,offy,n,b86);
    if(mm==8&&nn==8)
        build(mm,nn,offx,offy,n,b88);
    if(mm==8&&nn==10)
        build(mm,nn,offx,offy,n,b810);
    if(mm==10&&nn==8)
        build(mm,nn,offx,offy,n,b108);
    if(mm==10&&nn==10)
        build(mm,nn,offx,offy,n,b1010);
    if(mm==10&&nn==12)
        build(mm,nn,offx,offy,n,b1012);
    if(mm==12&&nn==10)
        build(mm,nn,offx,offy,n,b1210);
}
void Knight::build(int m,int n,int offx,int offy,int col ,grid
*b)
{
    int i,p,q,k=m*n;
    for(i=0;i<k;i++)
    {
        int
x1=offx+b[i].x,y1=offy+b[i].y,x2=offx+b[(i+1)%k].x,y2=offy+b[(i
+1)%k].y;
        p=pos(x1,y1,col);
```

```
207         q=pos(x2,y2,col);
208         link[x1][y1].x =q;
209         link[x2][y2].y =p;
210     }
211 }
212 //计算方格的编号
213 int Knight::pos(int x,int y,int col)
214 {
215     return col*x+y;
216 }
217 void Knight::out0(int m,int n,ofstream &out)
218 {
219     int i,j,k,x,y,p,a[20][20];
220     if(comp(m,n,0,0))
221         return;
222     for(i=0;i<m;i++)
223     {
224         for(j=0;j<n;j++)
225         {
226             a[i][j]=0;
227         }
228     }
229     i=0;j=0;k=2;
230     a[0][0]=1;
231     out<<"(0,0)"<<"";
232     for(p=1;p<m*n;p++)
233     {
234         x=link[i][j].x;
235         y=link[i][j].y;
236         i=x/n;j=x%n;
237         if(a[i][j]>0)
238         {
239             i=y/n;
240             j=y%n;
241         }
242         a[i][j]=k++;
243         out<<"("<<i<<","<<j<<")";
244         if((k-1)%n==0)
245         {
246             out<<"\n";
247         }
248     }
```

```
    out<<"\n";
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            out<<a[i][j]<<"   ";
        }
        out<<"\n";
    }
}
int main()
{
    int m,n;
    ifstream in("input.txt",ios::in); //利用文件流读取数据
    ofstream out("output.txt",ios::out);//利用文件流将数据存到文件中
    in>>m>>n;
    Knight k(m,n);
    k.comp(m,n,0,0);
    k.outO(m,n,out);
    in.close();
    out.close();
}
```

▲验证

书上案例验证通过

<2> 算法实现题 2-3 半数集问题

▲问题重述

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。

编程任务：对于给定的自然数 n，编程计算半数集 set(n)中的元素个数。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。每个文件只有 1 行，给出整数 n。(0<n<1000)

结果输出：程序运行结束时，将计算结果输出到文件 output.txt 中。输出文件只有 1 行，给出半数集 set(n)中的元素个数。

▲解题思路

问题有些复杂的时候可以先举例子。

例如.set(6) 6，16，26，126，36，136。半数集set(6)中有6个元素。

例如.set(10) 10，510，2510，12510，1510，410，2410，12410，1410，310，1310，210，1210，110，半数集set(10)中有14个元素。

设set(n)中的元素个数为f(n)。如：6的前面可以加上1、2、3，而2、3的前面又都可以加上1，也就是f(6)=1+f(3)+f(2)+f(1)。

则显然有递归表达式：f(n)=1+∑f(i)，i=1,2.....n/2。

可以先写一个递归函数如下

```
int f(int n)
{
int temp=1;
if(n>1)
for (int  i=1; i < = n/2; i ++)
            temp+=f(i);
return temp;
}
```

时间复杂度：n/2个相加，每一个需要计算1+...+i/2，因此时间复杂度为O(n^2)缺点：这样会有很多的重复子问题计算。

这个问题显然存在重叠子问题：例如，当n=4时，f(4)=1+f(1)+f(2)，而f(2)=1+f(1)，在计算f(2)的时候又要重复计算一次f(1)。如果这样的话时间复杂度会很大，我们要想办法简化重叠部分的计算。

可以采用“备忘录”的方法，来避免重叠部分的计算。

▲代码

```cpp
#include <algorithm>
#include <stdio.h>
using namespace std;

int solve(int f[], int n)
{
    if (n == 1)
        return 1;
    if (f[n] != 0)
        return f[n];
    int sum = 1;
    for (int i = 1; i <= n / 2; i++)
        sum += solve(f, i);
    f[n] = sum;
    return sum;
}

int main()
{
    int n;
    scanf("%d", &n);
    int f[n + 1];
    for (int i = 1; i <= n; i++)
        f[i] = 0;
    printf("%d\n", solve(f, n));
}

```

▲验证

P1028 [NOIP2001 普及组] 数的计算（https://www.luogu.com.cn/problem/P1028）

题目描述

复制Markdown 展开

给出正整数 n，要求按如下方式构造数列：

1. 只有一个数字 n 的数列是一个合法的数列。
2. 在一个合法的数列的末尾加入一个正整数，但是这个正整数不能超过该数列最后一项的一半，可以得到一个新的合法数列。

请你求出，一共有多少个合法的数列。两个合法数列 a, b 不同当且仅当两数列长度不同或存在一个正整数 $i \leq |a|$，使得 $a_i \neq b_i$。

输入格式

输入只有一行一个整数，表示 n。

输出格式

输出一行一个整数，表示合法的数列个数。

输入输出样例

输入 #1	复制	输出 #1	复制
6		6	

测试结果如下：

▲算法分析

使用到记忆化剪枝，本质上仍然是对每个f[i]都去遍历了f[0]到f[n/2]，所以时间复杂度应该是O(n^2)。

由于开了备忘录数组，空间复杂度O(n)。

<3>算法实现题 2-6 排列的字典序问题

▲题目重述

问题描述：n个元素1 , 2 , ⋯ , n 有 n ! {1,2,\cdots,n}有n!1,2,⋯,n有n!个不同的排列。将这n ! n!n!个排列按字典序排列，并编号为0 , 1 , ⋯ , n ! − 1 0,1,\cdots,n!-10,1,⋯,n!−1。每个排列的编号为其字典序值。例如，当n=3时，6个不同排列的字典序值如下：

字典序值	**0**	**1**	**2**	**3**	**4**	**5**
排列	123	132	213	231	312	321

算法设计：给点n及n个元素的一个排列，计算出这个排列的字典序值，以及按字典序排列的下一个排列
数据输入：由文件input.txt提供输入数据。文件的第1行是元素个数n。接下来的一行是n个元素的一个排列
结果输出：将计算出的排列的字典序值和按字典序排列的下一个排列输出到文件output.txt。文件的第1行是字典序值，第2行是按字典序排列的下一个排列。

▲解题思路

1. 阶乘函数 (`factorial`)： 通过递归计算阶乘，并使用数组 `factorial_array` 进行记忆化，以避免在相同参数上的重复计算。
2. 排列函数 (`permutation`)： 通过调用阶乘函数计算排列的总数，该函数返回以输入整数为长度的排列的数量。
3. 字典序索引计算函数 (`dictNum`)： 该函数通过模拟字典序生成的过程，计算给定整数序列在字典序中的索引。它使用一个辅助链表 `sup_array` 来存储部分已经排好序的数字，然后遍历输入数组，计算每个数字在当前序列中的相对位置，从而得到总的字典序索引。
4. 主函数 (`main`)：
    - 读取输入文件中的整数 `n` 和整数数组 `data`。
    - 调用 `dictNum` 函数计算字典序索引，并将结果写入输出文件。
    - 尝试找到下一个字典序排列，如果找到，将其写入输出文件。

▲代码

```cpp
#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define rep(i, s, n) for (int i = s; i < n; i++)
#define reb(i, d, n) for (int i = d; i >= n; i--)
const int N = 100;
ll factorial_array[N];
list<int> sup_array; // 辅助数组
ll factorial(int n)
{
    if (factorial_array[n] != 0)
        return factorial_array[n];
    ll res = 1;
    for (int i = 1; i <= n; i++)
        res *= i;
    factorial_array[n] = res; // 记忆，避免重复计算
    return res;
}
ll permutation(int n)
{
    ll res = 1;
    rep(i, 2, n)
        res *= factorial(i);
    return res;
}

ll dictNum(int *data, int size)
{
    // 将数字插入辅助数组中，判断有多少个比其小的数
    int index = 1;
    ll res = 0;
    sup_array.push_back(data[0]);
    res += (data[0] - 1) * factorial(size - 1);
    for (int i = 1; i < size; i++)
    {
        list<int>::iterator p = sup_array.begin();
        int min_nums;
        int j;
        for (j = 0; p != sup_array.end(); p++, j++)
        {
```

```
            if (data[i] < *p) // 升序排列 找到位置
            {
                min_nums = (data[i] - j - 1);
                sup_array.insert(p, data[i]);
                break;
            }
        }
        if (p == sup_array.end())
        {
            min_nums = (data[i] - j - 1);
            sup_array.push_back(data[i]);
        }
        res += (min_nums)*factorial(size - i - 1);
    }

    return res;
}

int main()
{
    int n;
    ifstream fin;
    fin.open("input.txt", ios::in);
    if (!fin.is_open())
    {
        cerr << "input.txt not found" << endl;
        return -1;
    }
    ofstream fout;
    fout.open("output.txt", ios::out);

    fin >> n;
    int data[N];
    // cout << factorial(7) + 4 * factorial(6) + 2 *
factorial(5) + 2 * factorial(4) + 3 * factorial(3) + 1 << endl;
    rep(i, 0, n)
    {
        fin >> data[i];
    }
    fout << dictNum(data, n) << endl;
    // 寻找下一个，倒着遍历 直到找到第一个比最后一个小的数，然后交换，再把后
面排序即可
```

```
    bool flag = 0;
    reb(i, n - 2, 0)
    {
        if (data[i] < data[n - 1])
        {
            swap(data[i], data[n - 1]);
            sort(data + i + 1, data + n);
            flag = 1;
            break;
        }
    }
    if (!flag)
        sort(data, data + n);
    rep(i, 0, n)
            fout
        << data[i] << " ";
    fin.close();
    fout.close();

    return 0;
}

```

▲验证

书上验证案例通过。

<4> 算法实现题 2-7 集合划分问题

▲问题重述

n 个元素的集合{1,2,....., n }可以划分为若干个非空子集。例如，当 n=4 时，集合{1，2，3，4}可以划分为 15 个不同的非空子集如下：

```
{{1}，{2}，{3}，{4}}，
{{1，2}，{3}，{4}}，
{{1，3}，{2}，{4}}，
{{1，4}，{2}，{3}}，
{{2，3}，{1}，{4}}，
{{2，4}，{1}，{3}}，
{{3，4}，{1}，{2}}，
```

```
{{1，2}，{3，4}}，
{{1，3}，{2，4}}，
{{1，4}，{2，3}}，
{{1，2，3}，{4}}，
{{1，2，4}，{3}}，
{{1，3，4}，{2}}，
{{2，3，4}，{1}}，
{{1，2，3，4}}
```

编程任务：
给定正整数 n，计算出 n 个元素的集合{1,2,....., n }可以划分为多少个不同的非空子集。

数据输入：
由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n。

结果输出:
程序运行结束时，将计算出的不同的非空子集数输出到文件 output.txt 中。

▲解题思路

这道题在实验里做过了，在实验里我给出了三种方法解决这道题，详细可以参考实验报告2。

这里作为作业，为了简单考虑，我采取先求出第二类斯特林数再加和得到贝尔数的方法来解决。

【贝尔数】

B[n]的含义是基数为 n的集合划分成非空集合的划分数。

贝尔数自身递推关系： $B[n+1] = \sum C(n,k)\ B[k]$ ，其中k从0到n。

其中定义B[0] = 1

【第二类斯特林数】

第二类斯特林数实际上是集合的一个拆分，表示将 n个不同的元素拆分成m个集合间有序（可以理解为集合上有编号且集合不能为空）的方案数，记为 S(n,m) （这里是大写的）或者 {n m} (n在上m在下) 。

和第一类斯特林数不同的是，这里的集合内部是不考虑次序的，而圆排列圆的内部是有序的。常常用于解决组合数学中的几类放球模型。描述为：将n个不同的球放入m个无差别的盒子中，要求盒子非空，有几种方案。

S(n,k)的值可以递归的表示为：S(n+1, k) = kS(n, k) + S(n, k-1)。

递推边界条件：

S(n,n) = 1, n>=0

S(n,0) = 0, n>=1

为什么会这样表示呢？当我们将第（n + 1）个元素添加到k个划分集合时，有两种可能性。

- 第n + 1个元素作为一个单独的集合参与到划分成k个集合，有S(n,k-1)个。
- 将第n + 1个元素添加到已经划分的k个集合中，一共有k*S(n,k)种。

▲代码

```
#include <bits/stdc++.h>
#include <iostream>
using namespace std;
typedef long long ll;
typedef int itn;
const int N = 5000;
const int mod = 998244353;
int n, m, k;
ll S[N][N];
ll B[N];

int solve_S(int n, int k)
{
    S[0][0] = 1;
    S[n][0] = 0;
    for (int i = 1; i <= n; ++i)
    {
        for (int j = 1; j <= k; ++j)
        {
            S[i][j] = (S[i - 1][j - 1] + 1ll * j * S[i - 1][j])
% mod; // 公式中的 k 是当前的 k
```

```
        }
    }
    return 0;
}

ll Bell(int n)
{
    if (n == 0)
        return 1;
    ll ans = 0;
    for (int k = 0; k <= n; k++)
    {
        if (!S[n][k])
            solve_S(n, k);
        ans += S[n][k];
        ans = ans % mod;
    }
    return ans;
}
int main()
{
    for (int i = 0; i < N; i++)
        B[i] = 0;
    B[0] = 1;

    int n, T;
    scanf("%d", &T);
    for (int i = 0; i < T; i++)
    {
        scanf("%d", &n);
        // printf("ans = %lld\n", Bell(n));
        printf("%lld\n", Bell(n));
    }
    return 0;
}

```

▲验证

洛谷P5748 集合划分计数（https://www.luogu.com.cn/problem/P5748）（与书本问题有细微差异：数据多组，要求取模）

题目描述

复制Markdown 展开

一个有 n 个元素的集合，将其分为任意个非空子集，求方案数。
注意划分出的集合间是无序的，即 $\{\{1,2\},\{3\}\}$ 和 $\{\{3\},\{2,1\}\}$ 算作一种方案。

由于答案可能会很大，所以要对 998244353 取模。

输入格式

第一行一个正整数 T，表示数据组数。
接下来 T 行，每行一个正整数 n。

输出格式

输出 T 行，每行一个整数表示答案。

输入输出样例

输入 #1 复制

```
5
2
3
7
9
233
```

输出 #1 复制

```
2
5
877
21147
53753544
```

第三次作业【动态规划】

<1>算法实现题 3-1 独立任务最优解问题

▲问题重述

用两台处理机A 和 B处理n 个作业。设第个作业交给机器A处理时需要时间ai，若由机器 B 来处理，则需要时间 b。由于各作业的特点和机器的性能关系，可能对于某些，有azb，而对于某些jjfi)，有ab。既不能将一个作业分开由两台机器处理也没有一台机器能同时处理2个作业。设计一个动态规划算法，使得这两台机器处理完这n
个作业的时间最短(从任何一台机器开工到最后一台机器停工的总时间)。研究一个实例:
(a1,a2,a3,a4,as,a6)=(2,5,7,10,5,2)，(b, b2,b, b4, bs, b6)=(3,8,4,11,3,4)。

算法设计:对于给定的两台处理机A 和 B 处理个作业，找出一个最优调度方案，使2台机器处理完这n个作业的时间最短。

数据输入:由文件 inputtxt 提供输入数据。文件的第1行是1个正整数n，表示要处理n个作业。在接下来的2行中，每行有n个正整数，分别表示处理机A和B 处理第i个作业需要的处理时间。

案例：

```
input.txt
6
2 5 7 10 5 2
3 8 4 11 3 4

output.txt
15
```

▲解题思路

状态转移方程：`dp[i][j]=min(dp[i-1][j]+b[i],dp[i-1][j-a[i]])`;

`dp[i][j]`代表做完前i个任务，A机器花几分钟情况下，B机器所花的时间，也就是说`dp[i][j]`就是表示B机器所花时间。

`dp[i][j] = dp[i-1][j]+b[i]`代表第i个任务交给B来做，所以做完前i个任务的时候,A机器和前i - 1的任务一样，还是花了j分钟，而B机器则花`dp[i-1][j]+b[i]`分钟；

`dp[i][j] = dp[i-1][j-a[i]]`代表第i个任务交给A来做，现在的A机器花费时间是j，所以在前i - 1个任务完成的时候，A机器是花了j-a[i]分钟的，所以现在B机器还是花了`dp[i-1][j-a[i]]`分钟；

一直到`dp[n][i]`:代表所有的任务都做完了，B机器所花费的时间，那么最迟的时间就是B的时间和A的时间求最大值；

最后这个循环`for(int i=0; i<=sum; i++)ans=min(ans,max(dp[n][i],i));//max(dp[n][i],i)` 表示完成前n个作业A机器花i分钟 B机器花`dp[n][i]`分钟情况下，最迟完工时间

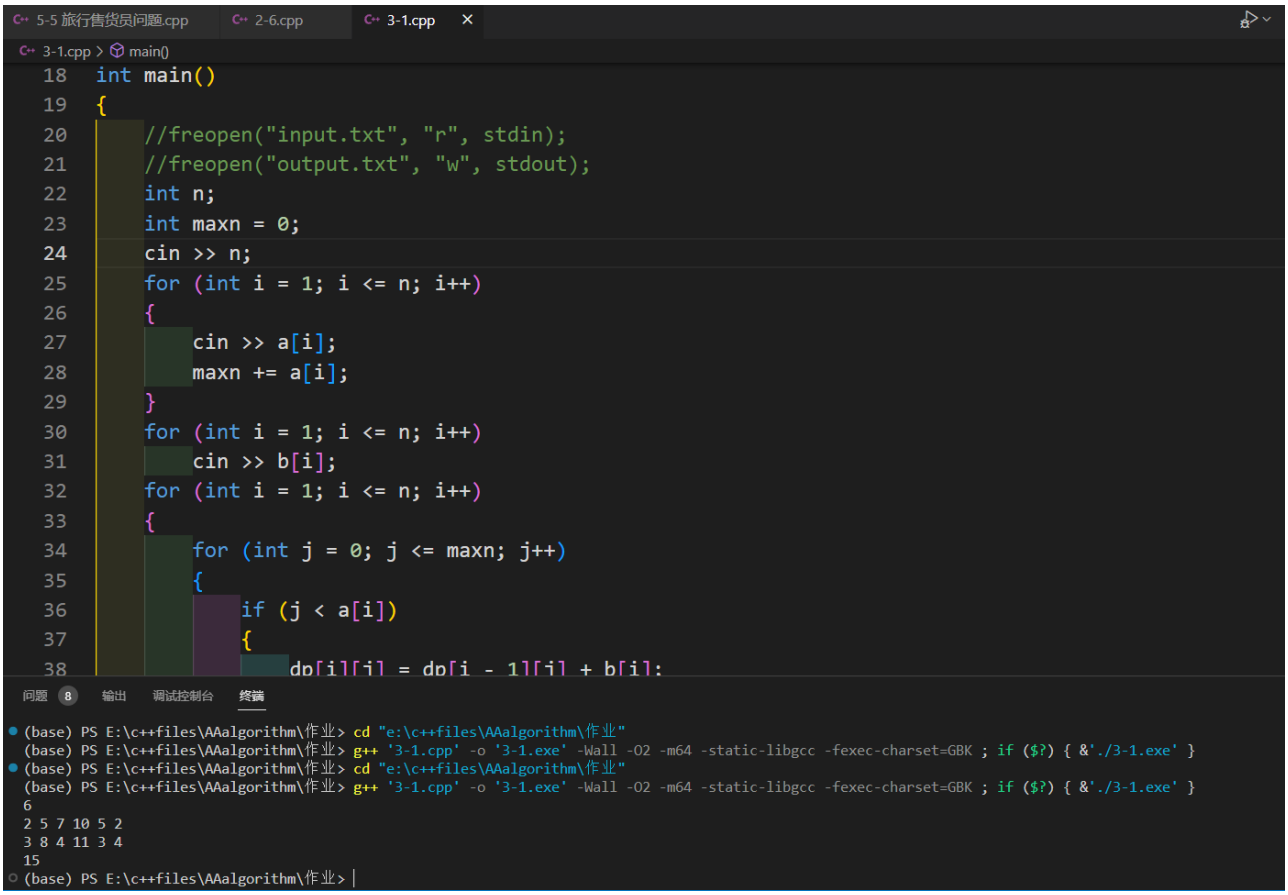
▲代码

```
#include <iostream>
#include <algorithm>
```

```cpp
#include <cstring>
#include <cstdio>
#include <vector>
#include <cmath>
#include <queue>
#include <stack>
#include <map>
using namespace std;
typedef long long ll;
const int INF = 0x3f3f3f3f;
const int MAXN = 1e6 + 100;
const double eps = 1e-6;
int Data[MAXN];
int a[MAXN], b[MAXN];
int dp[500][1000];
int main(){
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
    int n;
    int maxn = 0;
    cin >> n;
    for(int i=1;i<=n;i++){
        cin >> a[i];
        maxn += a[i];
    }
    for(int i=1;i<=n;i++) cin >> b[i];
    for(int i=1;i<=n;i++){
        for(int j=0;j<=maxn;j++){
            if(j < a[i]){
                dp[i][j] = dp[i - 1][j] + b[i];
            }else{
                dp[i][j] = min(dp[i - 1][j - a[i]], dp[i - 1][j] + b[i]);
            }
        }
    }
    int ans = INF;
    for(int i=0;i<=maxn;i++){
        if(i < dp[n][i]){
            ans = min(ans, dp[n][i]);
        }else{
            ans = min(ans, i);
```

```
        }
    }
    cout << ans;
    return 0;
}

```

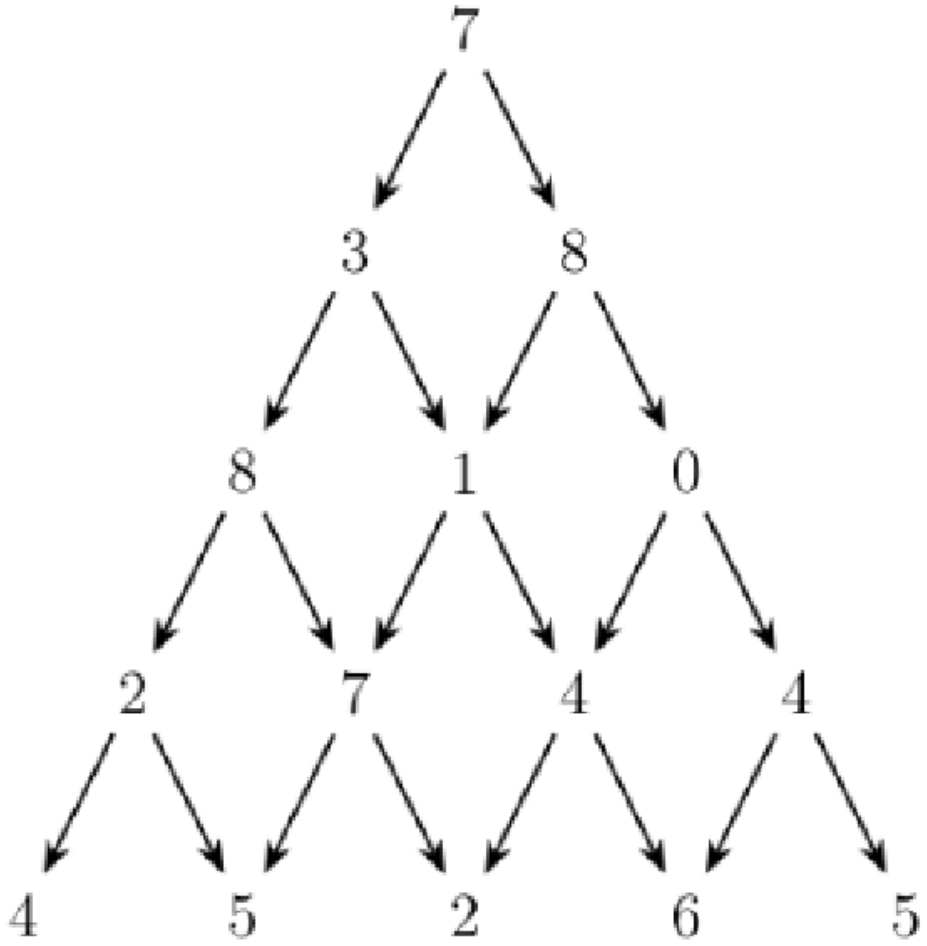
▲验证



<2>算法实现题 3-4 数字三角形问题

▲问题重述

定一个由n行数字组成的数字三角形，试设计一个算法，计算出从三角形的顶至底的一条路径，使该路径经过的数字总和最大。数据输入： 由文件input.txt提供输入数据。文件的第1行是数字三角形的行数n，(1≤n≤100)。接下来n行是数字三角形各行中的数字。所有数字在0~99之间。结果输出：将计算结果输出到文件output.txt。文件第1行中的数是计算出的最大值。示例：如右图所示，从 7→3→8→7→5的路径产生了最大权值30。



▲解题思路

这道题可以用动态规划来做，重点是表示状态和写状态转移方程设 v[i,j] 为点 (i,j) 上的值，f[i,j] 表示由 (1,1) 到 (i,j) 的路径最大总和，则 f[i,j]=Max{f[i-1,j-1],f[i-1,j]}+v[i,j]。

此处注意“左上角点”对应的是点 (i-1,j-1)，右上角对应的是点 (i-1,j)。边界条件更加容易想到，是 f[i][0]=`f[0][j]`=0 (0<=i,j<=n)。最后还需注意一点。如果浅学过DP可能下意识输出 `f[n][n]`，但根据题目中“到底部任意处结束”可以看出，总和最大的路径可能终结于 (n,1) 到 (n,n) 的任意一点，最后还需要再从中寻找最大的 f[n,j]。

整理思路，首先二重循环输入三角形存储于 a[][] 中，再设一 f[][] 用二重循环求解后，从 `f[n][1]` 至 `f[n][n]` 中找到最大值输出。

▲代码

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[1010][1010],f[1010][1010];
int main(){
    int n,s=0;
    cin>>n;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=i;j++)
            cin>>a[i][j];
    for(int i=1;i<=n;i++)
        for(int j=1;j<=i;j++)
            f[i][j]=max(f[i-1][j-1],f[i-1][j])+a[i][j];
    for(int j=1;j<=n;j++) s=max(s,f[n][j]);
    cout<<s;
    return 0;
}

```

▲验证：

洛谷P1216 [USACO1.5] [IOI1994]数字三角形 Number Triangles（https://www.luogu.com.cn/problem/P1216）

<3>算法实现题 3-8 最小m段和问题

▲问题重述

给定n个整数组成的序列，现在要求将序列分割成m段，每段子序列中的数在原数列中连续排列。如何分割才能使m段子序列的和的最大值达到最小？

数据读入/读出：

第一行中有2个正整数n和m。正整数n是序列的长度，正整数m是分割的段数。接下来的一行中有n个整数。

▲解题思路

使用`dp[i][j]`放置将前i个数分成j段的最大最小值。
所以`dp[i][1]`就是前i个数的和，`dp[i][1] = dp[i-1][1] + a[i]`;
当j>1的时候，假设前k个数为j-1段，从k~i为第j段,所以前j-1段的最大最小值为:`dp[k][j-1]` (前k个数分为j-1段).
最后一段为：`dp[i][1]-dp[k][1]` (前i个数的和减去前k个数的和)
这两个值中选取一个最大值，当所有情况讨论结束后，选出结果中最小的作为`dp[i][j]`的值。

因此，状态转移方程为

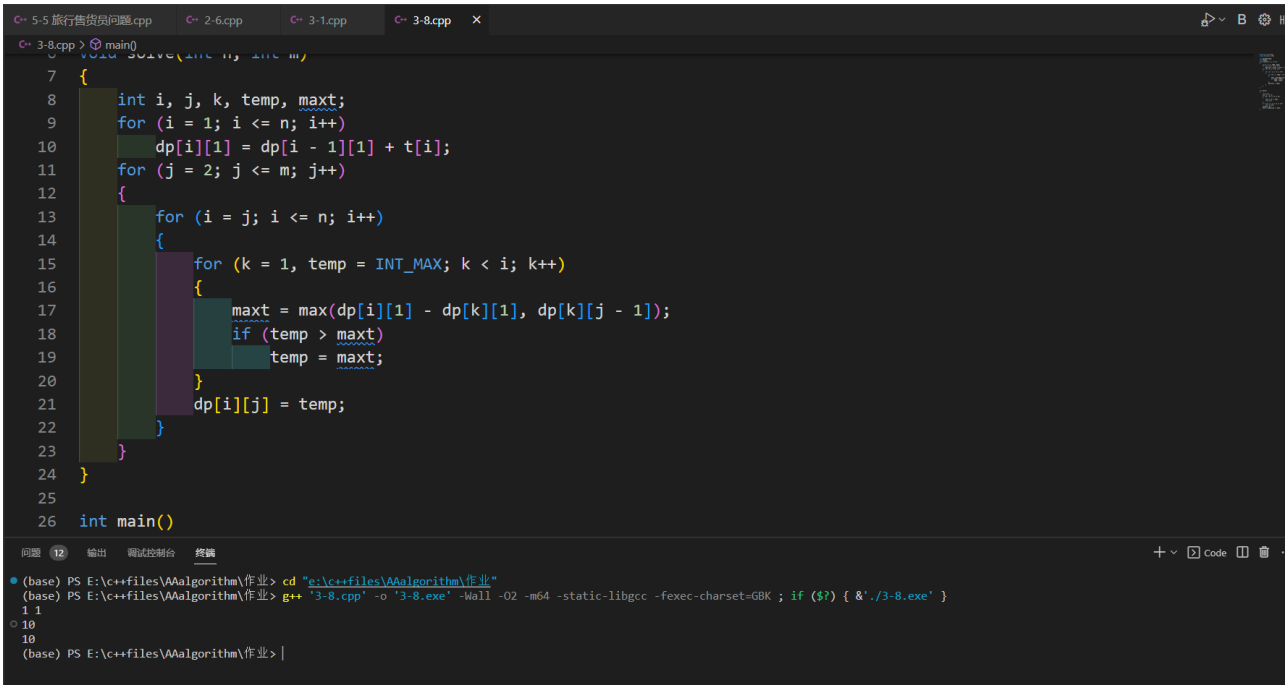
```
dp[i][j] = min{ max{ dp[k][j-1], dp[i][1]-dp[k][1] } } (0<k<i)
```

▲代码

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

int dp[500][500];
int t[500];
void solve(int n, int m)
{
    int i, j, k, temp, maxt;
    for (i = 1; i <= n; i++)
        dp[i][1] = dp[i - 1][1] + t[i];
    for (j = 2; j <= m; j++)
    {
        for (i = j; i <= n; i++)
        {
```

```
            for (k = 1, temp = INT_MAX; k < i; k++)
            {
                maxt = max(dp[i][1] - dp[k][1], dp[k][j - 1]);
                if (temp > maxt)
                    temp = maxt;
            }
            dp[i][j] = temp;
        }
    }
}

int main()
{
    int n, m;
    cin >> n >> m;
    if ((n < m) || (n == 0))
    {
        cout << 0 << endl;
        return 0;
    }
    for (int i = 1; i <= n; i++)
        cin >> t[i];
    solve(n, m);
    cout << dp[n][m] << endl;
}

```

▲验证



<4>算法实现题 3-25 m处理器问题

▲问题重述

在一个网络通信系统中，要将 n 个数据包依次分配给 m 个处理器进行数据处理，并要求处理器负载尽可能均衡。

设给定的数据包序列为:$\{\sigma_0, \sigma_1, \ldots, \sigma_{n-1}\}$ $\{\sigma_0, \sigma_1, \ldots, \sigma_{n-1}\}$。

m处理器问题要求的是$r_0 = 0 \le r_1 \le \ldots \le r_{m-1} \le n = r_m$ $r_0 = 0 \le r_1 \le \ldots \le r_{m-1} \le n = r_m$ ，将数据包序列划分为m段: $\{\sigma_0, \ldots, \sigma_{r1-1}\}$ $\{\sigma_0, \ldots, \sigma_{r_1-1}\}$, $\{\sigma_{r1}, \ldots, \sigma_{r2-1}\}$ $\{\sigma_{r_1}, \ldots, \sigma_{r_2-1}\}$, … , $\{\sigma_{r_{m-1}}, \ldots, \sigma_{n-1}\}$ $\{\sigma_{r_{m-1}}, \ldots, \sigma_{n-1}\}$, 使 $\max_{i=0}^{m-1} f(r_i, r_{i+1})$ $\max_{i=0}^{m-1} f(r_i, r_{i+1})$达到最小。式中，$f(i,j) = \sqrt{\sigma_i^2 + \ldots + \sigma_j^2}$ $f(i,j) = \sqrt{\sigma_i^2 + \ldots + \sigma_j^2}$是序列$\{\sigma_i, \ldots, \sigma_j\}$ $\{\sigma_i, \ldots, \sigma_j\}$的负载量。

$\max_{i=0}^{m-1} f(r_1, r_{i+1})$ $\max_{i=0}^{m-1} f(r_1, r_{i+1})$ $\max_{i=0}^{m-1} f(r_1, r_{i+1})$ $\max_{i=0}^{m-1} f(r_1, r_{i+1})$的最小值称为数据包序列$\{\sigma_0, \sigma_1, \ldots, \sigma_{n-1}\}$ $\{\sigma_0, \sigma_1, \ldots, \sigma_{n-1}\}$的均衡负载量。

对于给定的数据包序列$\{\sigma_0, \sigma_1, \ldots, \sigma_{n-1}\}$ $\{\sigma_0, \sigma_1, \ldots, \sigma_{n-1}\}$，编程计算 m 个处理器的均衡负载量。

数据输入：

第 1 行有 2 个正整数 n 和 m。n 表示数据包个数，m 表示处理器数。接下来的 1 行中有 n 个整数，表示 n 个数据包的大小。

▲解题思路

分析与解答： 设 $g(i, k)$是将$\{\sigma_i, \cdots, \sigma_{n-1}\}$划分为 k 段的均衡负载量，所求的最优值为 $g(0, m)$。

$g(i, k)$具有最优子结构性质，且满足如下递归式：

当 $2 \le k < n-i$ 时，$g(i,k) = \min_{i \le j \le n-k} \max\{f(i,j), g(j+1, k-1)\}$。

当 $n-i < k \le m$ 时，$g(i,k) = g(i, n-i)$。

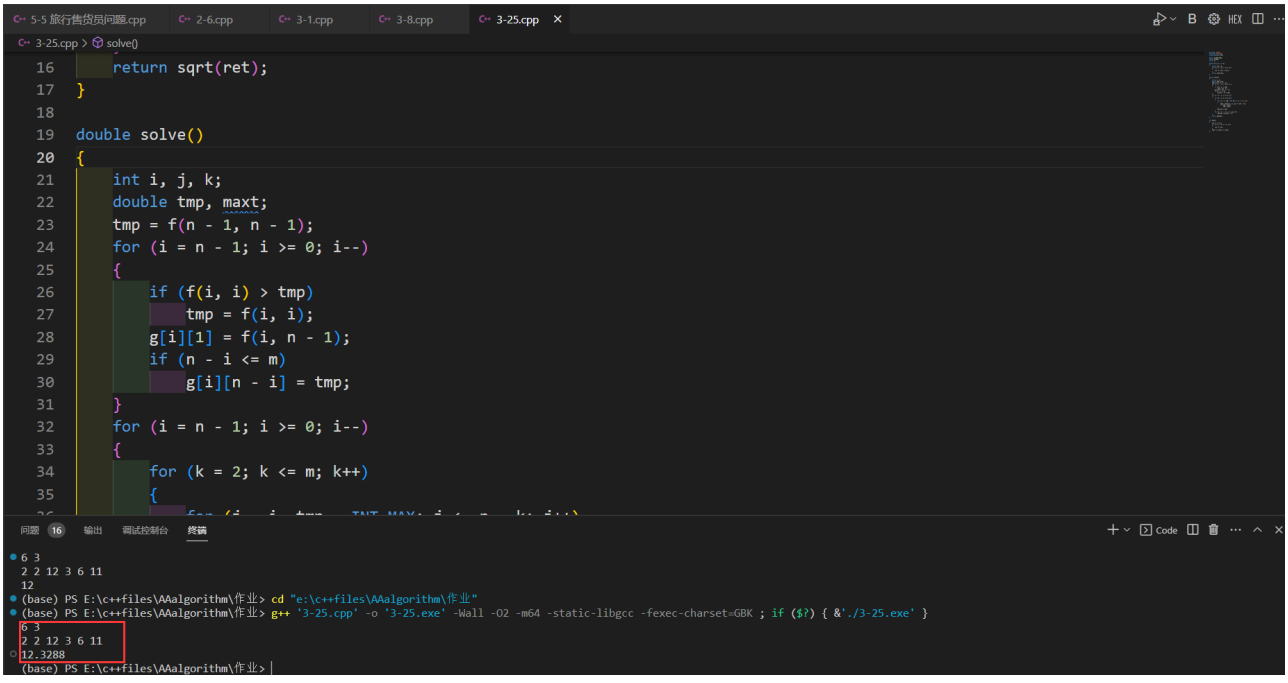
边界条件是：$g(i,1) = f(i, n-1)$，$g(i, n-i) = \max_{i \le j \le n} f(j,j)$。

▲代码

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

double g[500][500];
double t[500];
int n, m;

double f(int a, int b)
{
    double ret = 0;
    for (int i = a; i <= b; i++)
    {
        ret += (t[i] * t[i]);
    }
    return sqrt(ret);
}

double solve()
{
    int i, j, k;
    double tmp, maxt;
    tmp = f(n - 1, n - 1);
    for (i = n - 1; i >= 0; i--)
    {
        if (f(i, i) > tmp)
            tmp = f(i, i);
        g[i][1] = f(i, n - 1);
        if (n - i <= m)
            g[i][n - i] = tmp;
    }
    for (i = n - 1; i >= 0; i--)
    {
        for (k = 2; k <= m; k++)
        {
            for (j = i, tmp = INT_MAX; j <= n - k; j++)
            {
                maxt = max(f(i, j), g[j + 1][k - 1]);
                if (tmp > maxt)
                    tmp = maxt;
            }
            g[i][k] = tmp;
```

```
43          }
44          for (k = n - i + 1; k <= m; k++)
45              g[i][k] = g[i][n - i];
46      }
47      return g[0][m];
48  }
49
50  int main()
51  {
52      cin >> n >> m;
53      for (int i = 0; i < n; i++)
54      {
55          cin >> t[i];
56      }
57      cout << solve() << endl;
58  }
59
```

▲验证



第四次作业【贪心算法】

<1> 算法分析题4-1 会场安排问题

▲问题重述

假设要在足够多的会场里安排一批活动，并希望使用尽可能少的会场。设计一个有效的贪心算法进行安排。（这个问题实际上是著名的图着色问题。若将每一个活动作为图的一个顶点，不相容活动间用边相连。使相邻顶点着有不同颜色的最小着色数，相应于要找的最小会场数。） 对于给定的k个待安排的活动，计算使用最少会场的时间表。

Input
输入数据的第一行有1 个正整数k（k≤10000），表示有k个待安排的活动。接下来的k行中，每行有2个正整数，分别表示k个待安排的活动开始时间和结束时间。时间以0 点开始的分钟计。

Output
输出一个整数，表示最少会场数。

```
Sample Input
5
1 23
12 28
25 35
27 80
36 50
Sample Output
3
```

▲想法

先按照开始时间递增排序，若开始时间相同则按照结束时间递增排序。每次都优先安排序号靠前的活动，除非不相容，一遍扫完之后再扫第二遍，同时开第二个会场，直至所有活动都被安排。

▲代码

```cpp
// -*- coding:utf-8 -*-

// File    :   4-1 会场安排问题.cpp
// Time    :   2023/12/29
// Author  :   wolf

#include <algorithm>
#include <iostream>

using namespace std;
struct activity
{
    int start;
    int end;
    bool if_arrange;
} a[10001];

bool cmp(struct activity a, struct activity b)
{
    if (a.start < b.start)
        return 1;
    else if (a.start == b.start)
    {
        if (a.end <= b.end)
            return 1;
    }
    return 0;
}

int main()
{
    int k;
    cin >> k;
    for (int i = 0; i < k; i++)
    {
        cin >> a[i].start >> a[i].end;
        a[i].if_arrange = 0;
    }
    sort(a, a + k, cmp);
    int ok_num = 0;  // 已安排成功的会场数量
    int now_end = 0; // 当前安排的活动的结尾时间
    int ans = 0;     // 要开的会场数目
```

```
    while (ok_num < k)
    {
        now_end = 0;
        ans++;
        for (int i = 0; i < k; i++)
        {
            if (a[i].if_arrange == 0)
            {
                if (a[i].start >= now_end) // 相容，可加入
                {
                    a[i].if_arrange = 1;
                    ok_num++;
                    now_end = a[i].end;
                }
            }
        }
    }
    cout << ans << endl;
    return 0;
}

```

▲验证

测试数据一（样例）

输入

```
5
1 23
12 28
25 35
27 80
36 50
```

输出

```
3
```

测试数据二

输入

```
14
34 99
53 57
19 75
11 95
50 76
33 76
61 74
22 69
16 28
7 44
23 92
14 65
7 19
36 52
```

输出

```
9
```

测试数据三

输入

```
57
71 73
23 41
1 78
39 74
6 76
31 63
0 74
64 82
15 46
65 69
5 19
64 71
```

```
15 71
23 29
10 42
4 77
15 16
29 52
50 71
60 63
32 87
16 47
50 61
42 94
1 29
86 91
9 22
10 71
31 87
13 88
44 49
7 43
62 96
20 71
6 57
22 24
68 92
66 94
32 83
18 77
79 97
53 83
6 8
31 81
42 98
72 90
27 42
13 64
33 93
2 34
17 29
29 70
18 88
7 54
```

```
11 23
3 46
30 89
```

输出

```
30
```

<2> 算法实现题4-9 虚拟汽车加油问题

▲问题重述

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。对于给定的n(n <= 5000)和k(k <= 1000)个加油站位置，编程计算最少加油次数。并证明算法能产生一个最优解。

算法设计：对于给定的n和k个加油站的位置，计算最少的加油次数。

数据输入：第一行有两个正整数n和k，表示汽车加油后可行驶n km，且旅途中有k个加油站。接下来的一行中，有k+1个整数，表示第k个加油站和第k-1个加油站之间的距离。
结果输出：将计算的最少加油次数输出，如果无法到达目的地，则输出“No Solution”。

▲想法

对于到达每一个加油站时的处理：如果目前的油够到达下一站，就不再加油了，把油留到后面加，如果目前的油不够到达。

▲代码

```
// -*- coding:utf-8 -*-

// File    :   4-9 虚拟汽车加油问题.cpp
// Time    :   2023/12/29
// Author  :   wolf

#include <iostream>
```

```cpp

using namespace std;

int main()
{
    int n, k, temp;
    cin >> n >> k;
    int rest_oil = n;
    cin >> temp; // 读入第一个加油站，不做处理
    int ans = 0;
    for (int i = 0; i < k; i++)
    {
        int now_input;
        cin >> now_input;
        if (now_input > n)
        {
            cout << "No Solution" << endl;
            return 0;
            // 到下一加油站的距离超过加满油能行驶的距离，时不可行的。
        }
        if (rest_oil - now_input >= 0)
        {
            rest_oil -= now_input;
        }
        else
        {
            rest_oil = n;
            rest_oil -= now_input;
            ans++;
        }
    }
    cout << ans << endl;

    return 0;
}

```

▲验证

输入

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

输出

```
4
```

<3> 算法实现题4-13 非单位时间任务安排问题

▲问题重述：

问题描述和要求
具有截止时间和误时惩罚的任务安排问题可描述如下：
（1）给定 n 个任务的集合 S={1,2,...,n}；
（2）完成任务 i 需要 ti时间，1≤i≤n；
（3）任务 i 的截止时间 di，1≤i≤n，即要求任务 i 在时间 di之前结束；
（4）任务 i 的误时惩罚 wi，1≤i≤n，即任务 i 未在时间 di之前结束将招致 wi 的惩罚；若按时完成则无惩罚。

算法设计：设计一个贪心选择思想与动态规划思想相结合的算法，解决以上任务 安排问题，确定 S 的一个最优时间表，使得总误时惩罚达到最小。

数据输入：首先输入一个正整数 n 表示任务的个数。接下来的 n 行中，每行输入 三个整数，分别代表相应任务需要的时间 ti，截止时间 di，以及误时惩罚 wi。

结果输出：输出任务安排结果以及总的误时惩罚。

案例：

```
input
7
1 4 70
2 2 60
1 4 50
1 3 40
1 1 30
1 4 20
3 6 80

output
110
```

▲想法

首先将仍无依照其截止时间非递减排序

设对于任务1，2，……，i。截止时间为d的最小误时惩罚为`p[i][d]`，则`p[i][d]`符合最优子结构性质，且有递推式

```
p[i][d] = min{ p[i-1][d] + w[i], p[i-1][min{d,d[i]} - t[i]] }
```

边界条件有：

考虑`p[1][d]`

对于 `t[1] <= d` 时，截止时间大于完成任务的时间，而且只有一个任务，该任务是可以完成的，所以没有惩罚。

对于 `t[1] > d` 时，由于第一个工作无法完成，必然导致惩罚，罚时为w[1]。

▲代码

```
// -*- coding:utf-8 -*-

// File    :   4-13 非单位时间任务安排问题.cpp
// Time    :   2023/12/30
```

```cpp
// Author  :   wolf

#include <algorithm>
#include <iostream>

using namespace std;
int const MAXINT = 99999;
struct task
{
    int t; // cost time
    int d; // deadline
    int w; //  weight
} task[1000];

bool cmp(struct task a, struct task b)
{
    return a.d < b.d;
}

int main()
{
    int n;
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        cin >> task[i].t >> task[i].d >> task[i].w;
    }
    sort(task, task + n, cmp);
    int maxd = task[n - 1].d;
    int dp[n][maxd + 1]; // dp[i][j] 到第i个任务，截止时间为j时的最小惩罚
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j <= maxd; j++)
        {
            dp[i][j] = MAXINT;
        }
    }
    for (int j = 0; j <= maxd; j++)
    {
        if (j < task[0].d)
            dp[0][j] = task[0].w;
```

```
        else
            dp[0][j] = 0;
    }
    for (int i = 1; i < n; i++)
    {
        for (int j = 0; j <= maxd; j++)
        {
            int drop = dp[i - 1][j] + task[i].w;
            dp[i][j] = drop;
            if (min(j, task[i].d) - task[i].t >= 0)
            {
                int fetch = dp[i - 1][min(j, task[i].d) -
task[i].t];
                dp[i][j] = min(drop, fetch);
            }
        }
    }
    cout << dp[n - 1][maxd] << endl;
    // for (int i = 0; i < n; i++)
    // {
    //     for (int j = 0; j <= maxd; j++)
    //     {
    //         cout << dp[i][j] << " ";
    //     }
    //     cout << endl;
    // }

    return 0;
}

```

▲验证

```
input
7
1 4 70
2 2 60
1 4 50
1 3 40
1 1 30
1 4 20
3 6 80

output
110
```

<4> 算法实现题4-14 多元Huffman编码问题

▲想法

（1）求最大费用时只需将石堆排列后从大到小，两两进行合并即可。
（2）求最小费用时，将石堆排列后，每次合并的石堆数为K堆。但有时会出现特例，即每次合并K堆，最后一次合并时无法以K堆进行合并，这样合并的结果就不是最小费用了。所以就要先判断原总堆数是否能使每次合并的堆数都为K堆，如果不能的话就要在原堆数前面加上 X 个“0”堆来补齐缺少的堆数.。
例如共7堆最大合并5堆：
石子堆：45 13 12 5 9 22 16
这时排序后：5 9 12 13 16 22 45
如果先合并前5堆，这样结果就为177
如果补零的话，得到 0 0 5 9 12 13 16 22 45，这时正好每次都合并5堆，最后合并为1堆，这时结果为148

编程思路：

（1）分别创建一个递增的优先队列和一个递减的优先队列。创建队列的优点是可以对数字自动排序、取数等一些操作方便，如q.top();表示取队首元素。
（2）对于递增队列：① 先判断是否需要加“0”堆，需要就加；② 只要队列中的元素个数大于1（等价于队列中还至少有k堆石子，当然剩余石子堆数为k的整数倍），就对队列前k个元素进行处理。每次将队首加入到sum中，加完后就把队首弹出队列，继续处理下一个队首，直至处理了k个队首（即合并了k堆石子）。得到n个sum后，将sum累加起来，得到的就是最小费用。
（3）对于递减队列：方法同二，只是这里是连续处理2个队首。

▲代码

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n, k, x;                                        // x为每堆石子数
    long long sum1 = 0, sum2 = 0;                       // sum1、sum2分别为最小总费用和最大总费用
    priority_queue<int, vector<int>, greater<int>> q1; // 创建递增的优先队列
    priority_queue<int, vector<int>, less<int>> q2;    // 创建递减的优先队列
    cin >> n >> k;
    for (int i = 0; i < n; i++) // 输入每堆石子数并压入优先队列中
    {
        cin >> x;
        q1.push(x);
        q2.push(x);
    }
    while (q1.size() % (k - 1) != 1) // 判断q1能否每次合并k堆且恰好合并完
        q1.push(0);                  // 不能合并完，就要压入“0”
    while (q1.size() > 1)
    {
        long long sum = 0;
        for (int i = 0; i < k; i++) // 每次都是队首前k堆合并
        {
            sum += q1.top(); // 前k堆每堆的费用加入到临时变量sum中
```

```
            q1.pop();         // 队首的这堆石子合并后，将它弹出队列
        }
        sum1 += sum;  // 将前k堆的费用加入到总费用中
        q1.push(sum); // 这个前k堆的费用压入队列中
    }

    while (q2.size() > 1)
    {
        long long sum = 0;
        for (int i = 0; i < 2; i++)
        {
            sum += q2.top();
            q2.pop();
        }
        sum2 += sum;
        q2.push(sum);
    }
    cout << sum2 << " " << sum1 << endl;
    return 0;
}

```

▲验证

输入

```
7 3
45 13 12 16 9 5 22
```

输出

```
593 199
```

第五次作业【回溯算法】

<1> 算法分析题5-3 回溯法重写0-1背包

▲问题重述

一共有N件物品，第i（i从0开始）件物品的重量为weight[i]，价值为value[i]。在总重量不超过背包承载上限maxw的情况下，求能够装入背包的最大价值是多少？并要求输出选取的物品编号。

（要求使用回溯法求解）

▲解题思路

使用回溯法。构造解空间树，从第0层到第n-1层，每层表示对于背包内某个物品的“取”或“不取”。第n层为答案层，在第n层进行判定结果是否是想要的（即能不能获得更优的解），若是就做出相应的处理。

这是一个万能的解空间树图，借来用用。

剪枝想法：

（1）如果在第n层之前，就出现了总和大于的maxw情况，那么此时已经超重了。之后无论是否取，都不可能再得到总和小于maxw的结果了。这种情况以及它的子树直接删去即可。

（2）如果在第n层之前，目前已有的价值，即使加上剩余可取的最大价值，也不能达到已经达到的bestv，那么之后即使全部取也不能达到bestv了。这种情况及它的子树直接删去即可。

剪枝代码可以删去，不影响结果，但会降低效率。

▲代码

```
1 // -*- coding:utf-8 -*-
2
3 // File    :   01背包问题（回溯）.cpp
```

```cpp
// Time    :   2023/12/14
// Author  :   wolf

#include <iostream>
using namespace std;

int w[5000];
int v[5000];
bool flag[5000];
bool ans[5000];
int now_w = 0, now_v = 0;
int n, maxw, bestv = 0;
int rest_v;

void backtrace(int depth)
{
    if (depth == n) // 到达第n层：答案
    {
        if (now_v > bestv && now_w <= maxw) // 答案是需要打印的
        {
            bestv = now_v;
            for (int i = 0; i < n; i++)
            {
                ans[i] = flag[i];
            }
        }
        return;
    }
    if (depth < n && now_w > maxw)
        return; // 剪枝：此时背包已经过重
    if (now_v + rest_v <= bestv)
        return; // 剪枝：此时剩余价值即使全部拾取也无法达到最大价值
    rest_v -= v[depth];
    // 取这个物品
    now_v += v[depth];
    now_w += w[depth];
    flag[depth] = 1;
    backtrace(depth + 1);
    now_v -= v[depth];
    now_w -= w[depth];
    flag[depth] = 0;
    // 不取这个物品
```

```
46      backtrace(depth + 1);
47      rest_v += v[depth];
48      return;
49  }
50
51  int main()
52  {
53      cin >> maxw >> n;
54      for (int i = 0; i < n; i++)
55      {
56          cin >> w[i] >> v[i];
57          ans[i] = 0;
58          flag[i] = 0;
59          rest_v += v[i];
60      }
61      backtrace(0);
62      for (int i = 0; i < n; i++)
63      {
64         if (ans[i])
65              cout << i << " ";
66      }
67      cout << endl;
68      cout << bestv << endl;
69      return 0;
70  }
71
```

▲验证

洛谷P1048（https://www.luogu.com.cn/problem/P1048）

【验证时把输出最优解向量的for循环删去，题目要求不一样】

回溯法解决背包问题的O(2^n)还是从数量级上显著不如动态规划的O(n^2)。

故在数据量很大的时候，不能通过测评，显示超时。

所以01背包问题还是得用动态规划解，本题只是练习一下回溯法。

<2> 算法分析题5-5 旅行商问题（剪枝）

▲题目重述

5-5 旅行售货员问题的费用上界。

设 G 是一个有 n 个顶点的有向图，从顶点 i 发出的边的最大费用记为 $\max(i)$。

（1）证明旅行售货员回路的费用不超过 $\sum_{i=1}^{n}\max(i)+1$。

（2）在旅行售货员问题的回溯法中，用上面的界作为 bestc 的初始值，重写该算法，并尽可能地简化代码。

▲解答

（1）任一旅行售货员回路可表示为 n 个顶点的一个排列 $(\pi(1), \pi(2), \cdots, \pi(n))$。这个回路的费用为

$$h(\pi)=\sum_{i=1}^{n}a(\pi(i)), \pi(i \bmod n+1))$$

由此可知

$$h(\pi)=\sum_{i=1}^{n}a(\pi(i), \pi(i \bmod n+1)) \leqslant \sum_{i=1}^{n}\max(\pi(i))$$

$$=\sum_{i=1}^{n}\max(i)<\sum_{i=1}^{n}\max(i)+1$$

▲代码

```cpp
// -*- coding:utf-8 -*-

// File    :   5-5 旅行售货员问题.cpp
// Time    :   2023/12/30
// Author  :   wolf

#include <iostream>

using namespace std;
int const MAXINT = 99999;
int map[1000][1000];
int v, e, best_cost = MAXINT, now_cost = 0;
int now_order[1000]; // 储存当前排列解向量
int ans[1000];       // 储存结果排列解向量
int upbound;

void swap(int a, int b)
{
    int temp = now_order[a];
    now_order[a] = now_order[b];
    now_order[b] = temp;
};

void backtrack(int depth)
{
    if (depth == v - 1) // 到达答案层
    {
        if (map[now_order[depth]][now_order[0]] != -1) // 能跟第一个相连
        {
            now_cost += map[now_order[depth]][now_order[0]];
            if (now_cost < best_cost) // 更优，保存结果
            {
                best_cost = now_cost;
                for (int i = 0; i < v; i++)
                {
                    ans[i] = now_order[i];
                }
            }
            now_cost -= map[now_order[depth]][now_order[0]];
```

```
        }
    }
    else
    {
        for (int i = depth; i < v; i++) // 生成排列树当前向量中depth位置的节点编号（第depth个访问哪个节点）
        {
            if (map[now_order[depth - 1]][now_order[i]] != -1) // 前一节点到当前的这个节点有可达边
            {
                if (now_cost + map[now_order[depth - 1]][now_order[i]] <= upbound)
                // 剪枝：若当前步骤使得费用和就大于上界了，不必继续
                {
                    swap(i, depth);
                    now_cost += map[now_order[depth - 1]]
                                   [now_order[depth]];
                    backtrack(depth + 1);
                    now_cost -= map[now_order[depth - 1]]
                                   [now_order[depth]];
                    swap(i, depth);
                }
            }
        }
    }
    return;
}
int main()
{
    cin >> v >> e;
    for (int i = 0; i < v; i++)
        for (int j = 0; j < v; j++)
            map[i][j] = -1;
    for (int i = 0; i < e; i++)
    {
        int a, b, weight;
        cin >> a >> b >> weight;
        map[a][b] = weight;
        map[b][a] = weight;
    }
    // 剪枝预备：
    for (int i = 0; i < v; i++)
```

```
    {
        int i_out_max = 0; // 存储从i节点向外
        for (int j = 0; j < v; j++)
        {
            i_out_max = max(i_out_max, map[i][j]);
        }
        upbound += i_out_max;
    }

    // 指定从0号节点开始遍历
    // 反正最后每一个节点都需要遍历过
    for (int i = 0; i < v; i++)
        now_order[i] = i; // 初始按照递增顺序（初始顺序是无所谓的，只要保证都能遍历一遍就可以）
    backtrack(1);          // 0号节点已固定，搜索从1号节点开始的全排列
    cout << best_cost << endl;
    return 0;
}

```

▲验证

```
input
4 5
0 1 10
0 2 15
1 2 35
1 3 25
2 3 30

output
50
```

<3> 算法实现题5-2 最小长度电路板排列问题

▲问题重述

5-2　最小长度电路板排列问题。

问题描述：最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是，将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B=\{1, 2, \cdots, n\}$ 是 n 块电路板的集合。集合 $L=\{N_1, N_2, \cdots, N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集，且 N_i 中的电路板用同一根导线连接在一起。在最小长度电路板排列问题中，连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如，设 $n=8$，$m=5$，给定 n 块电路板及其 m 个连接块如下：

$$B=\{1, 2, 3, 4, 5, 6, 7, 8\}；\ L=\{N_1, N_2, N_3, N_4, N_5\}$$

$$N_1=\{4, 5, 6\}；\ N_2=\{2, 3\}；\ N_3=\{1, 3\}；\ N_4=\{3, 6\}；\ N_5=\{7, 8\}$$

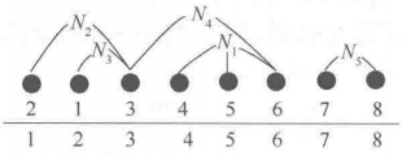


图 5-1　电路板排列

这 8 块电路板的一个可能的排列如图 5-1 所示。

在最小长度电路板排列问题中，连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如，在图 5-1 所示的电路板排列中，连接块 N_4 的第 1 块电路板在插槽 3 中，它的最后 1 块电路板在插槽 6 中，因此 N_4 的长度为 3。同理 N_2 的长度为 2。图 5-1 中的连接块最大长度为 3。

试设计一个回溯法找出所给 n 块电路板的最佳排列，使得 m 个连接块中的最大长度达到最小。

算法设计：对于给定的电路板连接块，设计一个算法，找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m（$1 \leqslant m, n \leqslant 20$）。接下来的 n 行中，每行有 m 个数。第 k 行的第 j 个数为 0 表示电路板 k 不在连接块 j 中，为 1 表示电路板 k 在连接块 j 中。

结果输出：将计算的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第一行是最小长度；接下来的 1 行是最佳排列。

▲解题思路

排列树解决问题。创建以下函数/类

1. **Board** 类：
   - `class Board` 定义了一个名为 `Board` 的类，用于处理电路板排列问题。
   - 私有成员变量包括 `n`（电路板数）、`m`（连接块数）、`x`（当前解的数组）、`bestx`（当前最优解的数组）、`bestd`（当前最优密度）、`low`（辅助数组，存储连接块的最小高度）、`high`（辅助数组，存储连接块的最大高度）、`B`（连接块数组）。
2. **len** 函数：
   - `int len(int ii)` 用于计算排列中的某一部分的长度，即连接块之间的最大高度差。

- 该函数首先将 `low` 和 `high` 数组初始化为合适的值，然后根据当前排列计算连接块的最小和最大高度，最后计算最大高度差并返回。

3. **Backtrack** 函数：

- `void Backtrack(int i)` 是一个递归函数，用于在排列树上进行回溯搜索，寻找最优解。
- 当达到排列树的终点（`i == n`）时，计算当前排列的长度，并更新最优解。
- 否则，对于当前位置 `i`，尝试选择不同的电路板进行交换，然后继续递归搜索。

4. **ArrangeBoards** 函数：

- `int ArrangeBoards(int **B, int n, int m, int *bestx)` 是主要的调用函数。
- 在该函数中，创建一个 `Board` 类的实例 `X`，并通过初始化设置其成员变量。
- 利用回溯算法调用 `Backtrack(1)` 来找到最优解。
- 返回最优解的密度。

5. **main** 函数：

- `main` 函数读取输入，调用 `ArrangeBoards` 函数来解决问题，并输出结果。
- 动态分配二维数组 `B` 以存储连接块信息。
- 读取输入的电路板连接信息。
- 创建数组 `bestx` 用于存储最优解。
- 输出最优解的密度和排列。

▲代码

```
// 电路板排列问题
#include <bits/stdc++.h>
using namespace std;
class Board
{
    friend int ArrangeBoards(int **, int, int, int *);

private:
    void Backtrack(int i);
    int len(int ii);
    int n,      // 电路板数
        m,      // 连接块数
        *x,     // 当前解
        *bestx, // 当前最优解
```

```
        bestd,  // 当前最优密度
        *low,   //
        *high,  //
        **B;    // 连接块数组
};
int Board::len(int ii)
{
    for (int i = 0; i <= m; i++)
    {
        high[i] = 0;
        low[i] = n + 1;
    }
    for (int i = 1; i <= ii; i++)
    {
        for (int k = 1; k <= m; k++)
        {
            if (B[x[i]][k])
            {
                if (i < low[k])
                    low[k] = i;
                if (i > high[k])
                    high[k] = i;
            }
        }
    }
    int tmp = 0;
    for (int k = 1; k <= m; k++)
    {
        if (low[k] <= n && high[k] > 0 && tmp < high[k] -
low[k])
            tmp = high[k] - low[k];
    }
    return tmp;
}
void Board::Backtrack(int i) // 回溯搜索排列树
{
    if (i == n) // 到达排列树终点
    {
        int tmp = len(i);
        if (tmp < bestd)
        {
            bestd = tmp;
```

```
            for (int j = 1; j <= n; j++)
                bestx[j] = x[j];
        }
    }
    else
    {
        for (int j = i; j <= n; j++) // 选择x[j]为下一块电路板
        {
            swap(x[i], x[j]);
            int ld = len(i);
            if (ld < bestd)
                Backtrack(i + 1);
            swap(x[i], x[j]);
        }
    }
}
int ArrangeBoards(int **B, int n, int m, int *bestx)
{
    Board X;
    // 初始化X
    X.x = new int[n + 1];
    X.low = new int[m + 1];
    X.high = new int[m + 1];
    X.B = B;
    X.n = n;
    X.m = m;
    X.bestx = bestx;
    X.bestd = n + 1;
    // 初始化total和now
    for (int i = 1; i <= n; i++)
    {
        X.x[i] = i;
    }
    X.Backtrack(1);
    delete[] X.x;
    delete[] X.low;
    delete[] X.high;
    return X.bestd;
}
int main()
{
    int n, m;
```

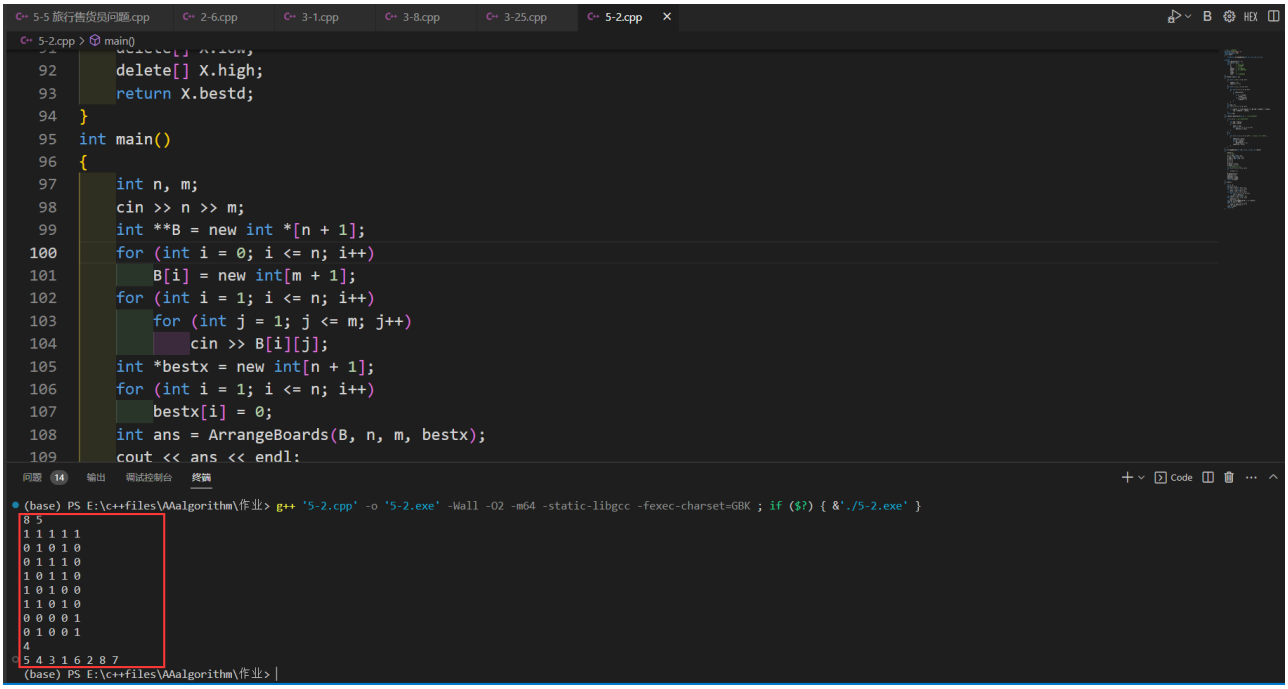
```
    cin >> n >> m;
    int **B = new int *[n + 1];
    for (int i = 0; i <= n; i++)
        B[i] = new int[m + 1];
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            cin >> B[i][j];
    int *bestx = new int[n + 1];
    for (int i = 1; i <= n; i++)
        bestx[i] = 0;
    int ans = ArrangeBoards(B, n, m, bestx);
    cout << ans << endl;
    for (int i = 1; i <= n; i++)
        cout << bestx[i] << " ";
    cout << endl;
    return 0;
}

```

▲验证

测试案例

```
8 5
1 1 1 1 1
0 1 0 1 0
0 1 1 1 0
1 0 1 1 0
1 0 1 0 0
1 1 0 1 0
0 0 0 0 1
0 1 0 0 1
```

测试结果：



<4> 算法实现题5-7 n色方柱问题

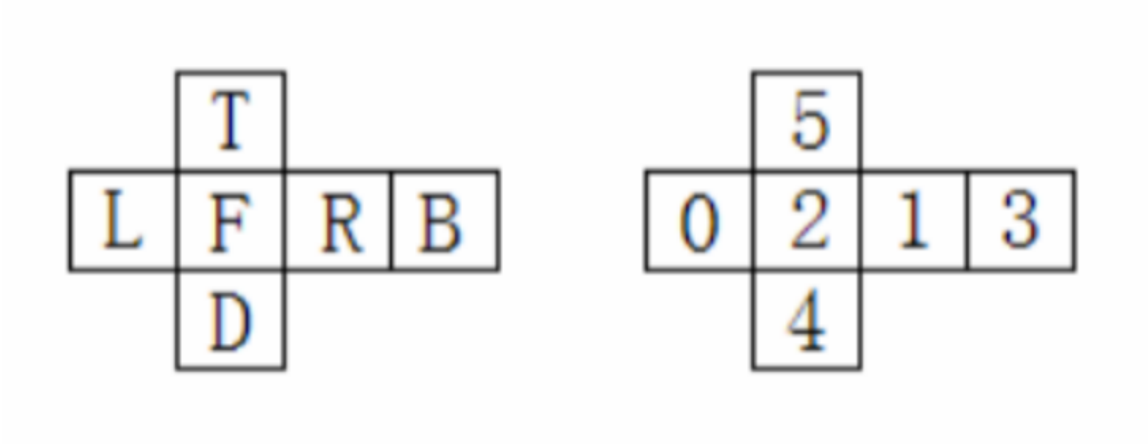
▲问题重述

设有 n 个立方体，每个立方体的每一面用红、黄、蓝、绿等 n 种颜色之一染色。要把这n 个立方体叠成一个方形柱体，使得柱体的 4 个侧面的每一侧均有 n 种不同的颜色。试设计一个回溯算法，计算出 n 个立方体的一种满足要求的叠置方案。

对于给定的 n 个立方体以及每个立方体各面的颜色，计算出 n 个立方体的一种叠置方案，使得柱体的 4 个侧面的每一侧均有 n 种不同的颜色。

数据输入：

第一行有 1 个正整数 n，0< n< 27，表示给定的立方体个 数和颜色数均为 n。第 2 行是 n 个大写英文字母组成的字符串。该字符串的第 k(0≤ k< n) 个字符代表第 k 种颜色。接下来的 n 行中，每行有 6 个数，表示立方体各面的颜色。立方体各面的编号如下图所示。



图中 F 表示前面，B 表示背面，L 表示左面，R 表示右面，T 表示顶面，D 表示底面。相 应地，2 表示前面，3 表示背面，0 表示左面，1 表示右面，5 表示顶面，4 表示底面。
例如，在示例输出文件中，第3行的6个数0 2 1 3 0 0分别表示第1个立方体的左面的颜色为R, 右面的颜色为B, 前面的颜色为G, 背面的颜色为Y, 底面的颜色为R, 顶面的颜色为 R。

案例：

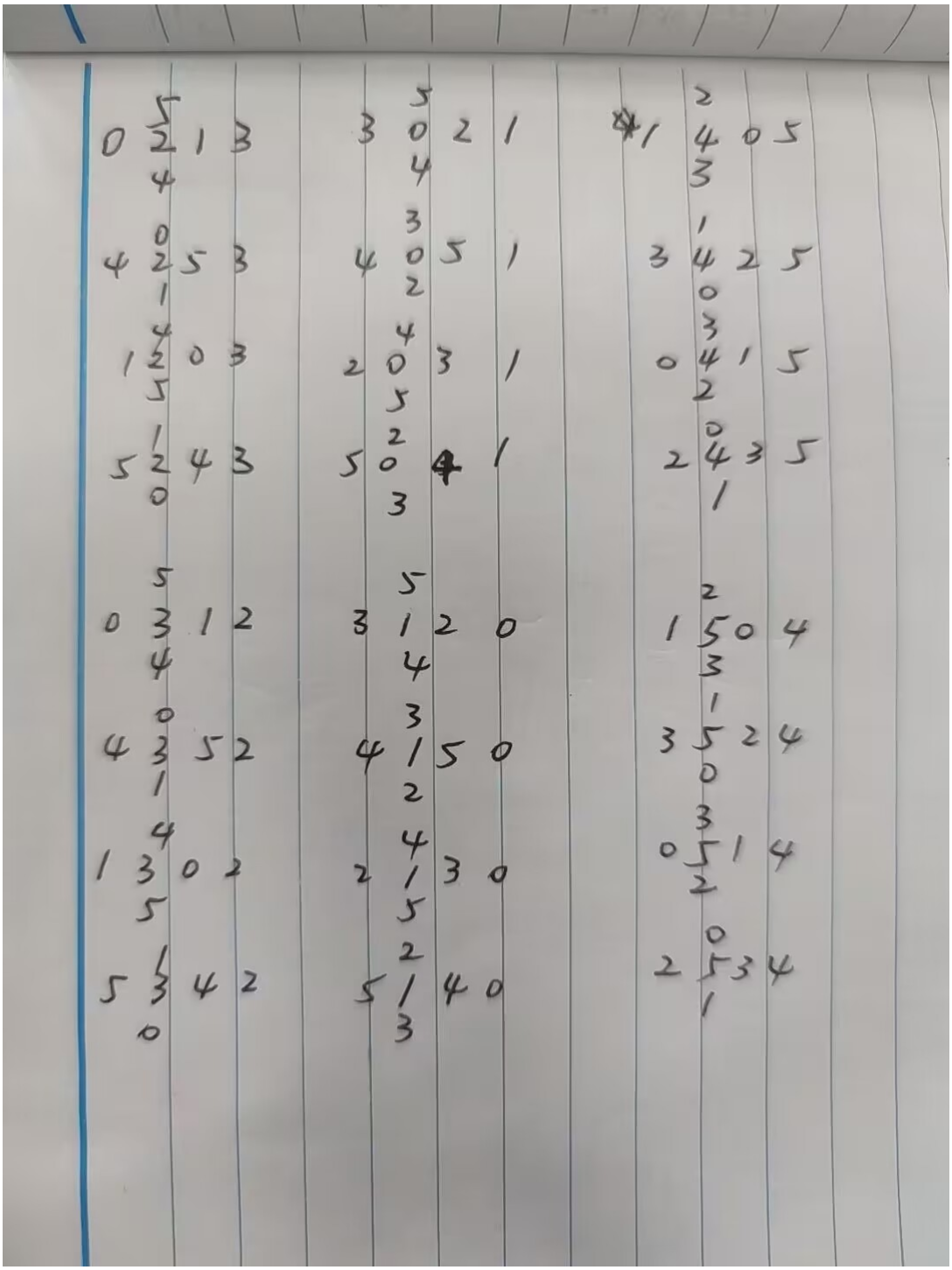
```
input
4
RGBY
0 2 1 3 0 0
3 0 2 1 0 1
2 1 0 2 1 3
1 3 3 0 2 2

output
RBGYRR
YRBGRG
BGRBGY
GYYRBB
```

▲解题思路

在下面的代码中使用注释讲解。使用回溯法的思路套用回溯法的模板，但有改动。

本题比较晦涩难懂，在理解上需要花很多时间。

关于代码部分的映射方式



▲代码

```
// -*- coding:utf-8 -*-

// File    :   5-7 n色方柱问题.cpp
// Time    :   2023/12/27
// Author  :   wolf

```

```cpp
#include <iostream>

using namespace std;
char color[28]; // 用来储存输入的数字对应的颜色（只在输出的时候转换为字符，在做题时使用数字存储）
int box[28][6]; // box[i][j]用来储存第i个立方体各个面（即第j面）的颜色
int n;
int count = 1; //  标记这是第几个输出的；可能结果
const int place[24][6] = {
    // 转动立方体的映射函数，使用box[depth+1][j]=origin[place[method][j]]来获取第method种方法下下一层的摆放方式
    // place[i][j]为第i种转换方法下该立方体该层的第j面应该变换为place[i][j]
    {0, 1, 2, 3, 4, 5},
    {4, 5, 2, 3, 1, 0},
    {1, 0, 2, 3, 5, 4},
    {5, 4, 2, 3, 0, 1}, // 2为底面，3为顶面

    {0, 1, 3, 2, 4, 5},
    {4, 5, 3, 2, 1, 0},
    {1, 0, 3, 2, 5, 4},
    {5, 4, 3, 2, 0, 1}, // 3为底面，2为顶面

    {3, 2, 0, 1, 4, 5},
    {4, 5, 0, 1, 2, 3},
    {2, 3, 0, 1, 5, 4},
    {5, 4, 0, 1, 3, 2}, //  0为底面，1为顶面

    {3, 2, 1, 0, 4, 5},
    {4, 5, 1, 0, 2, 3},
    {2, 3, 1, 0, 5, 4},
    {5, 4, 1, 0, 3, 2}, //  1为底面，0为顶面

    {1, 0, 4, 5, 3, 2},
    {3, 2, 4, 5, 0, 1},
    {0, 1, 4, 5, 2, 3},
    {2, 3, 4, 5, 1, 0}, //  4为底面，5为顶面

    {1, 0, 5, 4, 3, 2},
    {3, 2, 5, 4, 0, 1},
    {0, 1, 5, 4, 2, 3},
    {2, 3, 5, 4, 1, 0}, //  5为底面，4为顶面
```

```
};

void backtrack(int depth)
{
    // cout << "depth=" << depth << endl;
    if (depth == n - 1) // 到达答案层
    {
        cout << "Possible Solution " << count << " : " << endl;
        count++;
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < 6; j++)
            {
                cout << color[box[i][j]] << " ";
            }
            cout << endl;
        }
        cout << endl;
    }
    else
    {
        int process = depth + 1;
        int origin[6];
        // 【使用origin[]数组先保存原始情况，可以省去整体恢复原状的步骤，24次for循环每次都是新的开始】
        for (int i = 0; i < 6; i++)      // 保存待处理层初始存放的颜色
            origin[i] = box[process][i]; // origin[i]表示该层初始第i个面存放的颜色
        // cout << "origin=" << endl;
        // for (int i = 0; i < 6; i++)
        //     cout << origin[i] << " ";
        // cout << endl;
        for (int i = 0; i < 24; i++) // 列举子集树，每个立方体有24种放法,第i种方法
        {
            // cout << "method = " << i << endl;
            for (int j = 0; j < 6; j++) // 按照该种方案的映射，把处理的该层立方体先摆好
            {
                box[process][j] = origin[place[i][j]];
            }
```

```
            // 接下来看这种摆法是否可行
            int flag = 1; // 初始标记，表示可行
            // 表示遍历某个侧面时，是否出现重复颜色，used_i[j]标记第i个侧面第j号颜色是否被用过，初始清零
            int used_0[n];
            int used_1[n];
            int used_2[n];
            int used_3[n];
            for (int i = 0; i < n; i++) //
            {
                used_0[i] = 0;
                used_1[i] = 0;
                used_2[i] = 0;
                used_3[i] = 0;
            }
            for (int i = 0; i <= process; i++) // 遍历到现在所有已经放好的立方体，查看每个侧面是否有重复的颜色
            {
                used_0[box[i][0]]++;
                used_1[box[i][1]]++;
                used_2[box[i][2]]++;
                used_3[box[i][3]]++;
                if (used_0[box[i][0]] > 1 || used_1[box[i][1]] > 1 || used_2[box[i][2]] > 1 || used_3[box[i][3]] > 1)
                // 题目要求是最后摆好的整个立方体条的每个侧面都要有n种颜色
                // 但因为n个立方体摆出的，该侧面条有n种颜色，所以相当于每个立方体在该侧面的颜色都不一样
                // 即某个侧面条上每种颜色只能使用一次，这里转换了题目的条件
                // 某个侧面条上某个颜色用了不止一次，不符合条件，该方案以及该方案的子树剪掉
                {
                    flag = 0;
                    // cout << used_0[box[i][0]] << " " << used_1[box[i][1]] << " " << used_2[box[i][2]] << " " << used_3[box[i][3]] << endl;
                    break;
                }
            }
            if (flag == 1) // 目前所有已经摆好的立方体的每个侧面都没有重复的颜色，符合要求，可以继续放下一个立方体
```

```
                backtrack(depth + 1);
            }
        }
        return;
}

int main()
{
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        cin >> color[i];
    }
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < 6; j++)
        {
            cin >> box[i][j];
        }
    }
    cout << endl
         << "ans = " << endl;
    backtrack(-1);
    return 0;
}

```

▲验证

未找到在线测评，使用题目给出的案例

```
input
4
RGBY
0 2 1 3 0 0
3 0 2 1 0 1
2 1 0 2 1 3
1 3 3 0 2 2

```

输出结果如下：

```
ans =
Possible Solution 1 :
R B G Y R R
Y R B G R G
B G R B G Y
G Y Y R B B

Possible Solution 2 :
B R G Y R R
R Y B G G R
G B R B Y G
Y G Y R B B

Possible Solution 3 :
R B Y G R R
Y R G B R G
B G B R G Y
G Y R Y B B

Possible Solution 4 :
B R Y G R R
R Y G B G R
G B B R Y G
Y G R Y B B

Possible Solution 5 :
Y G R B R R
G B Y R R G
B R B G G Y
R Y G Y B B

Possible Solution 6 :
G Y R B R R
B G Y R G R
R B B G Y G
Y R G Y B B

Possible Solution 7 :
Y G B R R R
G B R Y R G
```

```
B R G B G Y
R Y Y G B B

Possible Solution 8 :
G Y B R R R
B G R Y G R
R B G B Y G
Y R Y G B B
```

可见答案不唯一（显然可能），题目给定的答案在其中之一。

我们这种算法的时间复杂度O(n*24^n)，数据量大了之后可能会很慢。

<5> 算法实现题5-13 任务分配问题

▲问题重述

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。

算法设计：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。

数据输入：第一行有1 个正整数n (1≤n≤20)。接下来的n行，每行n个数，表示工作费用。

样例：

```
input
3
10 2 3
2 3 4
3 4 5

output
9
```

▲解题思路

假设人不动，将工作分发给不同人。

解向量x[i]为第i个人被分配第x[i]个工作，解空间树为排列树。从第0层开始，第0层就有n个节点（可以认为第-1层有一个节点），答案层在第n层（第n-1层将自己与自己交换，实际上不影响最终结果）。在第n层比较当前总费用是不是比目前保存的方案总费用更少，如果是就将这个更小的存下来，否则不做处理。

遍历完整个排列树后得到最优解。

▲代码

```
// -*- coding:utf-8 -*-

// File    :   5-13.cpp
// Time    :   2023/12/25
// Author  :   wolf

#include <iostream>

using namespace std;
const int MAXNUM = 9999999;
int c[21][21];
int x[21];
int nowcost = 0, mincost = MAXNUM, n;

void swap(int a, int b)
{
    int temp = x[a];
    x[a] = x[b];
    x[b] = temp;
}

void backtrack(int depth)
{
    if (depth == n) // 答案层
    {
        mincost = min(mincost, nowcost);
        // cout << nowcost << endl;
```

```
28          // 不需要输出解向量，故不用保存解向量
29      }
30      else
31      {
32          for (int i = depth; i < n; i++)
33          {
34              // depth当前人，i为待分发物品序号，人不动，发物品
35              nowcost += c[x[i]][depth];
36              swap(i, depth);
37              backtrack(depth + 1);
38              swap(i, depth);
39              nowcost -= c[x[i]][depth];
40          }
41      }
42  }
43
44  int main()
45  {
46      cin >> n;
47      for (int i = 0; i < n; i++)
48      {
49          for (int j = 0; j < n; j++)
50          {
51              cin >> c[i][j];
52          }
53      }
54      for (int i = 0; i < n; i++)
55          x[i] = i;
56      backtrack(0);
57      cout << mincost << endl;
58
59      return 0;
60  }
61
```

▲验证

测试数据可通过。

第六次作业【分支限界法】

<1> 算法实现题6-2 最小权顶点覆盖问题

▲问题重述

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⊆VU⊆V，且对任意(u,v)∈E 有 u∈U 或 v∈U，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。

算法设计：
对于给定的无向图 G，设计一个优先队列式分支限界法，计算 G 的最小权顶点覆盖。

数据输入：
由文件input.txt给出输入数据。第 1 行有 2 个正整数 n 和 m，表示给定的图 G 有 n 个顶点和 m 条边，顶点编号为 1，2，...，n。第 2 行有 n 个正整数表示 n 个顶点的权。接下来 的 m 行中，每行有 2 个正整数 u,v，表示图 G 的一条边(u,v)。

结果输出：
将计算的最小权顶点覆盖的顶点权之和以及最优解输出到文件output.txt。文件的第1行是最小权顶点覆盖顶点权之和；第2行是最优解xi（1<=i<=n）,xi=0表示顶点i不在最小权顶点覆盖中，xi=1表示顶点i在最小权顶点覆盖中。

```
输入文件示例
input.txt
7 7
1 100 1 1 1 100 10
1 6
2 4
2 5
3 6
4 5
4 6
6 7

输出文件示例
```

```
output.txt
13
1 0 1 0 1 0 1
```

▲解题思路

1. 定义一个最小堆 `MinHeap` 类，用于实现堆操作。
2. `HeapNode` 类表示图中的一个顶点。`DealNode` 类包含一些操作，主要是用于处理堆中结点的操作。
3. `DealNode::BBVC()` 方法是该算法的核心部分。通过不断地加入和不加入某个顶点，并通过堆来遍历所有可能的情况，找到图的最小顶点覆盖。
4. `MinCover` 函数是对 `DealNode::BBVC()` 方法的封装，用于获取最终的最小顶点覆盖权重。
5. 在 `main` 函数中，用户输入了图的顶点数 `vertexNum` 和边数 `edgeNum`。然后输入每个顶点的权值，并通过边的信息构建了图的邻接矩阵。
6. 调用 `MinCover` 函数得到最小顶点覆盖权重，并输出结果。

▲代码

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

template <class Type>
class MinHeap // 最小堆类;
{
public:
    MinHeap(Type a[], int n);    // 带两参数的构造函数，在此程序中没有应用;
    MinHeap(int ms);             // 构造函数重载，只初始化堆的大小，对堆中结点不初始化；另外，堆元素的存储是以数组
    ~MinHeap();                  // 形式，且无父、子指针，访问父亲结点，利用数组标号进行;
    bool Insert(const Type &x); // 插入堆中一个元素;
    bool RemoveMin(Type &x);     // 删除堆顶最小结点;
    void MakeEmpty();            // 使堆为空
    bool IsEmpty();
    bool IsFull();
    int Size();

protected:
```

```
20      void FilterDown(const int start, const int endOfHeap); //
    自顶向下构造堆
21      void FilterUp(const int start);                          //
    自底向上构造堆
22  private:
23      Type *heap;
24      int maxSize;
25      const int defaultSize;
26      int currentSize; // 堆当前结点个数大小
27  };
28
29  template <class Type>
30  MinHeap<Type>::MinHeap(int ms) : defaultSize(100)
31  {
32      maxSize = (ms > defaultSize) ? ms : defaultSize;
33      heap = new Type[maxSize];
34      currentSize = 0;
35  }
36
37  template <class Type>
38  MinHeap<Type>::MinHeap(Type a[], int n) : defaultSize(100)
39  {
40      maxSize = (n > defaultSize) ? n : defaultSize;
41      heap = new Type[maxSize];
42      currentSize = n;
43      for (int i = 0; i < n; i++)
44          heap[i] = a[i];
45      int curPos = (currentSize - 2) / 2;
46      while (curPos >= 0)
47      {
48          FilterDown(curPos, currentSize - 1);
49          curPos--;
50      }
51  }
52
53  template <class Type>
54  MinHeap<Type>::~MinHeap()
55  {
56      delete[] heap;
57  }
58
59  template <class Type>
```

```cpp
void MinHeap<Type>::FilterDown(const int start, const int endOfHeap)
{
    int i = start, j = i * 2 + 1;
    Type temp = heap[i];
    while (j <= endOfHeap)
    {
        if (j < endOfHeap && heap[j] > heap[j + 1])
            j++;
        if (temp < heap[j])
            break;
        else
        {
            heap[i] = heap[j];
            i = j;
            j = 2 * i + 1;
        }
    }
    heap[i] = temp;
}

template <class Type>
void MinHeap<Type>::FilterUp(const int start)
{
    int i = start, j = (i - 1) / 2;
    Type temp = heap[i];
    while (i > 0)
    {
        if (temp >= heap[j])
            break;
        else
        {
            heap[i] = heap[j];
            i = j;
            j = (i - 1) / 2;
        }
    }
    heap[i] = temp;
}

template <class Type>
bool MinHeap<Type>::RemoveMin(Type &x)
```

```
{
    if (IsEmpty())
    {
        cerr << "Heap empty!" << endl;
        return false;
    }
    x = heap[0];
    heap[0] = heap[currentSize - 1];
    currentSize--;
    FilterDown(0, currentSize - 1);
    return true;
}

template <class Type>
bool MinHeap<Type>::Insert(const Type &x)
{
    if (IsFull())
    {
        cerr << "Heap Full!" << endl;
        return false;
    }
    heap[currentSize] = x;
    FilterUp(currentSize);
    currentSize++;
    return true;
}

template <class Type>
bool MinHeap<Type>::IsEmpty()
{
    return currentSize == 0;
}

template <class Type>
bool MinHeap<Type>::IsFull()
{
    return currentSize == maxSize;
}

template <class Type>
void MinHeap<Type>::MakeEmpty()
{
```

```
143     currentSize = 0;
144 }
145
146 template <class Type>
147 int MinHeap<Type>::Size()
148 {
149     return currentSize;
150 }
151
152 // 最小堆结点
153 class HeapNode // 堆结点类;
154 {
155     friend class DealNode;
156
157 public:
158     operator int() const { return cn; }
159
160 private:
161     int i,  // i标示堆中结点号
162         cn, // cn标示当前加入的覆盖顶点中权重之和
163         *x, // x数组标示那些顶点加入了覆盖顶点的行列
164         *c; // c数组标示X中的覆盖顶点中所有的邻接顶点
165 };
166
167 // VC类用来对堆中结点内部的的操作
168 class DealNode
169 {
170     friend int MinCover(int **, int[], int);
171
172 private:
173     void BBVC();
174     bool cover(HeapNode E);
175     void AddLiveNode(MinHeap<HeapNode> &H, HeapNode E, int cn,
    int i, bool ch);
176     int **a, n, *w, *bestx, bestn;
177 };
178
179 void DealNode::BBVC()
180 {
181     // 建立初始空堆
182     MinHeap<HeapNode> H(1000);
183     HeapNode E;
```

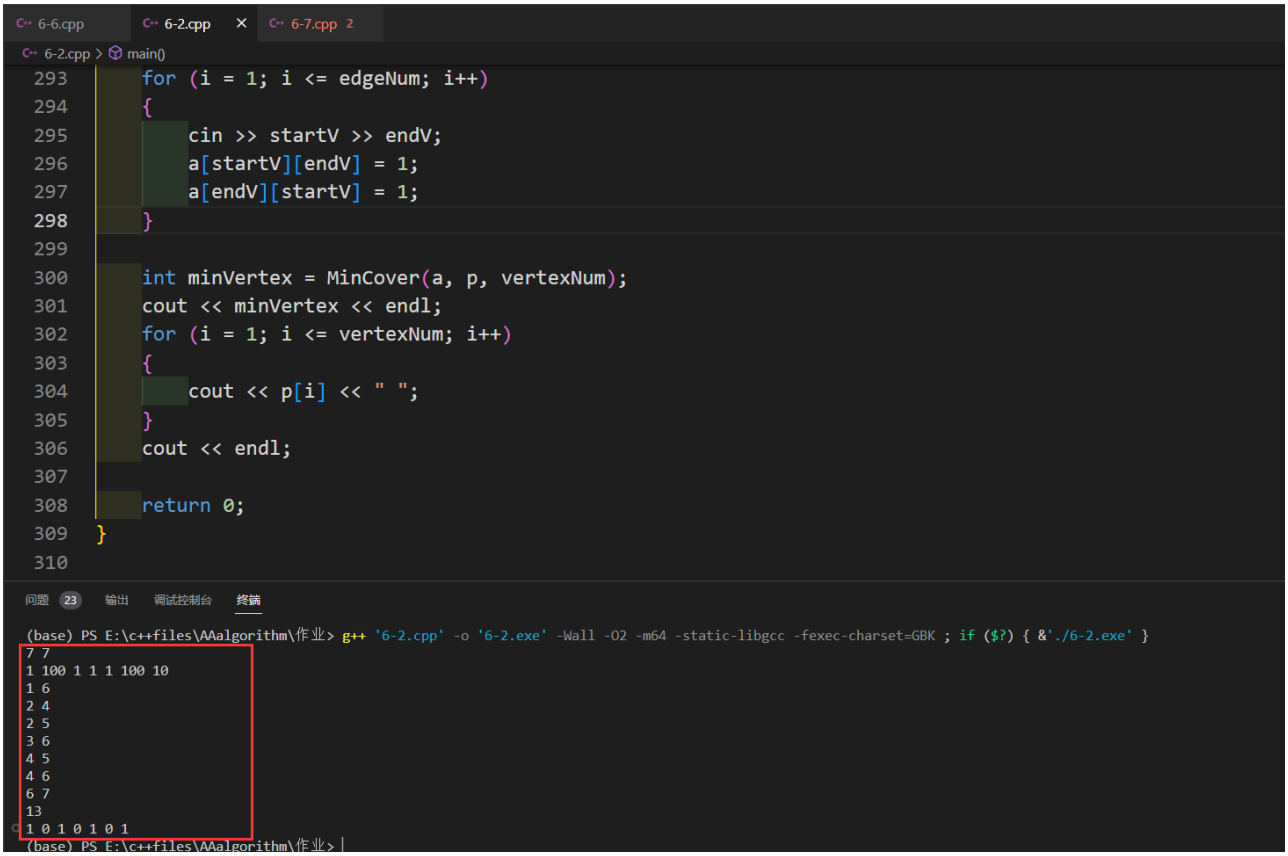
```
    E.x = new int[n + 1];
    E.c = new int[n + 1];
    for (int j = 1; j <= n; j++)
    {
        E.x[j] = E.c[j] = 0;
    }

    int i = 1, cn = 0;
    while (true)
    {
        if (i > n)
        {
            if (cover(E))
            {
                for (int j = 1; j <= n; j++)
                    bestx[j] = E.x[j];
                bestn = cn;
                break;
            }
        }
        else
        {
            if (!cover(E))
                AddLiveNode(H, E, cn, i, true); // 加入结点标号为i的结点到顶点覆盖集中，并把更新后的结点再插入堆中
            AddLiveNode(H, E, cn, i, false);    // 不把结点标号为i 的结点加入到顶点覆盖集中，并把更新后的结点插入堆中
        }
        if (H.IsEmpty())
            break;
        H.RemoveMin(E); // 取堆顶点赋给E
        cn = E.cn;
        i = E.i + 1;
    }
}

// 检测图是否被覆盖
bool DealNode::cover(HeapNode E)
{
    for (int j = 1; j <= n; j++)
    {
```

```
223         if (E.x[j] == 0 && E.c[j] == 0) // 存在任意一条边的两个顶点都为0的情况下，为未覆盖情况
224             return false;                // X[j]记录覆盖顶点，c[j]记录与覆盖顶点相连的顶点 0表征未覆盖，1表征已覆盖
225     }
226     return true;
227 }
228
229 void DealNode::AddLiveNode(MinHeap<HeapNode> &H, HeapNode E, int cn, int i, bool ch)
230 {
231     HeapNode N;
232     N.x = new int[n + 1];
233     N.c = new int[n + 1];
234     for (int j = 1; j <= n; j++)
235     {
236         N.x[j] = E.x[j];
237         N.c[j] = E.c[j];
238     }
239     N.x[i] = ch ? 1 : 0;
240
241     if (ch)
242     {
243         N.cn = cn + w[i]; // 记录i顶点是否加入覆盖的行列中；
244         for (int j = 1; j <= n; j++)
245             if (a[i][j] > 0) // 如果i,j相邻，刚把j顶点加入覆盖邻接顶点集中；
246                 N.c[j]++;
247     }
248     else
249     {
250         N.cn = cn;
251     }
252     N.i = i;
253     H.Insert(N); // 插入堆中
254 }
255
256 int MinCover(int **a, int v[], int n)
257 {
258     DealNode Y;
259     Y.w = new int[n + 1];
260     for (int j = 1; j <= n; j++)
```

```
261     {
262         Y.w[j] = v[j]; // 初始化DealNode类对象Y;
263     }
264     Y.a = a;
265     Y.n = n;
266     Y.bestx = v; // 将地址赋予bestx，
267     Y.BBVC();
268     return Y.bestn; // bestn是最后的最小顶点覆盖集权重;
269 }
270 
271 int main()
272 {
273     int startV, endV;        // 一条边的起始节点，终止节点
274     int vertexNum, edgeNum; // 顶点数，边数
275     int i;
276 
277     cin >> vertexNum >> edgeNum;
278 
279     int **a; // 图的邻接矩阵表示，1表示有边
280     a = new int *[vertexNum + 1];
281 
282     for (int k = 0; k <= vertexNum; k++)
283         a[k] = new int[vertexNum + 1];
284     for (int i = 0; i <= vertexNum; i++)
285         for (int j = 0; j <= vertexNum; j++)
286             a[i][i] = 0;
287 
288     int *p; // 顶点的权值数组
289     p = new int[vertexNum + 1];
290     for (i = 1; i <= vertexNum; i++)
291         cin >> p[i];
292 
293     for (i = 1; i <= edgeNum; i++)
294     {
295         cin >> startV >> endV;
296         a[startV][endV] = 1;
297         a[endV][startV] = 1;
298     }
299 
300     int minVertex = MinCover(a, p, vertexNum);
301     cout << minVertex << endl;
302     for (i = 1; i <= vertexNum; i++)
```

```
303     {
304         cout << p[i] << " ";
305     }
306     cout << endl;
307 
308     return 0;
309 }
310 
```

▲验证



<2> 算法实现题6-6 n后问题

▲问题重述

设计一个解n后问题的队列式分支限界法，计算在n × n n\times nn×n个方格上放置彼此不受攻击的n个皇后的一个放置方案。
案例

```
input
5
output
1 3 5 2 4
```

▲解题思路

1. 定义一个结构体`node`，表示棋盘上的每一个可能的位置，以及记录了当前状态的一些信息，如列、左右对角线等的占用情况。
2. 使用优先队列`priority_queue`来存储搜索过程中的状态，按照结构体中的`x`值进行排序。这里的`x`表示当前放置的皇后所在的行数。
3. 在主循环中，初始化棋盘的初始状态，将第一行的每一个位置作为起点，生成相应的初始状态，并加入优先队列中。
4. 进入主循环，每次从优先队列中取出一个状态，判断是否达到了目标状态（即放置了所有皇后），如果是则输出解，并结束程序（因为只需要找到一个可行解即可）。
5. 如果当前状态不是目标状态，继续在下一行尝试放置皇后。遍历每一列，对于每一个可行的位置，生成新的状态并加入优先队列中。
6. 在生成新状态时，进行剪枝操作，检查当前位置是否与之前的皇后冲突，如果冲突则跳过该位置。
7. 重复以上步骤，直到找到一个解或者队列为空。由于采用优先队列，搜索时会先尝试最有希望的位置，加速找到解的过程。

▲代码

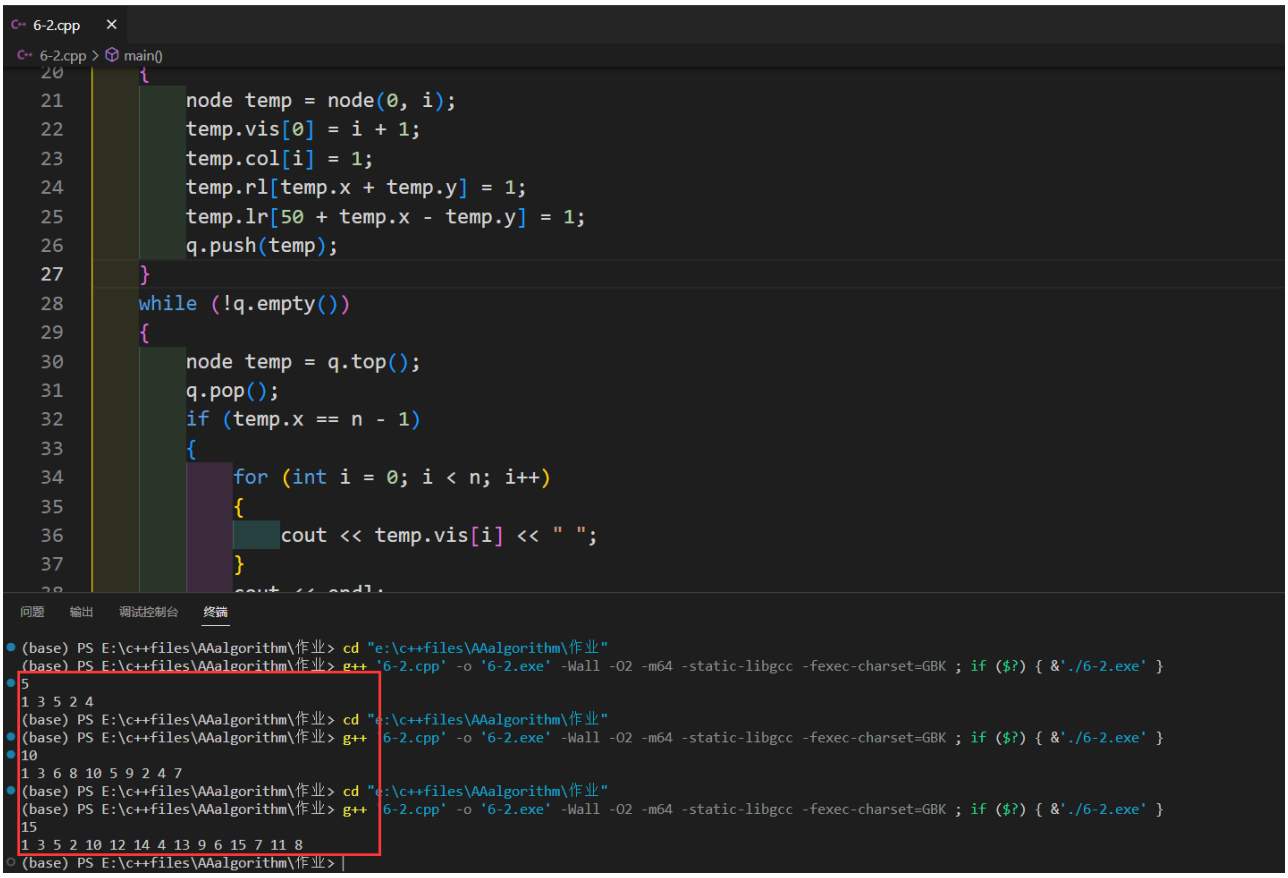
```cpp
#include <bits/stdc++.h>
using namespace std;
#define N 100
int n;
struct node
{
    int vis[N] = {0}, col[N] = {0}, lr[N] = {0}, rl[N] = {0};
    int x, y;
    node(int a, int b) : x(a), y(b) {}
    bool operator<(const node &a) const
    {
        return x < a.x;
    }
};
priority_queue<node> q;
int main()
{
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        node temp = node(0, i);
        temp.vis[0] = i + 1;
```

```
        temp.col[i] = 1;
        temp.rl[temp.x + temp.y] = 1;
        temp.lr[50 + temp.x - temp.y] = 1;
        q.push(temp);
    }
    while (!q.empty())
    {
        node temp = q.top();
        q.pop();
        if (temp.x == n - 1)
        {
            for (int i = 0; i < n; i++)
            {
                cout << temp.vis[i] << " ";
            }
            cout << endl;
            break; // 只需要给出一个答案即可
        }
        if (temp.x < n - 1)
        {
            for (int i = 0; i < n; i++)
            {
                node next = node(temp.x + 1, i);
                if (temp.col[next.y] || temp.lr[50 + next.x -
next.y] || temp.rl[next.x + next.y])
                { // 剪枝
                    continue;
                }
                for (int i = 0; i < N; i++)
                {
                    next.lr[i] = temp.lr[i];
                    next.rl[i] = temp.rl[i];
                    next.col[i] = temp.col[i];
                }
                next.col[next.y] = 1;
                next.lr[50 + next.x - next.y] = 1;
                next.rl[next.x + next.y] = 1;
                for (int i = 0; i < next.x; i++)
                {
                    next.vis[i] = temp.vis[i];
                }
                next.vis[next.x] = i + 1;
```

```
                q.push(next);
            }
        }
    }
    return 0;
}

```

▲验证

验证了n=5,10,15三种情况。



<3> 算法实现题6-7 布线问题

▲问题重述

6-7　布线问题。

问题描述：假设要将一组元件安装在一块线路板上，为此需要设计一个线路板布线方案。各元件的连线数由连线矩阵 conn 给出。元件 i 和元件 j 之间的连线数为 conn(i,j)。如果将元件 i 安装在线路板上位置 r 处，而将元件 j 安装在线路板上位置 s 处，则元件 i 和元件 j 之间的距离为 dist(r,s)。确定了所给的 n 个元件的安装位置，就确定了一个布线方案。与此布线方案相应的布线成本为 $\text{dist}(r,s)\times\sum_{1\leqslant i\leqslant j\leqslant n}\text{conn}(i,j)$。试设计一个优先队列式分支限界法，找出所给 n 个元件的布线成本最小的布线方案。

算法设计：对于给定的 n 个元件，设计一个优先队列式分支限界法，计算最佳布线方案，使布线费用达到最小。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（$1\leqslant n\leqslant 20$）。接下来的 $n-1$ 行，每行 $n-i$ 个数，表示元件 i 和元件 j 之间连线数（$1\leqslant i<j\leqslant 20$）。

结果输出：将计算的最小布线费用以及相应的最佳布线方案输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	10
2 3	1 3 2
3	

▲解题思路

1. `MinHeap` 类定义了最小堆，用于存储待处理的状态。该堆的元素是 `BoardNode` 类型的对象。
2. `BoardNode` 类表示电路板的一种摆放方式，包含了一些必要的信息。`len` 方法用于计算电路板摆放的长度。
3. `BBArrangeBoards` 函数是基于分支限界法的核心算法。它通过不断生成摆放状态，使用最小堆来搜索可能的最优解。`HeapSize` 为堆的大小。
4. `Make2DArray` 函数用于动态创建二维数组。
5. 在 `main` 函数中，用户输入了电路板数量 `n`。通过 `Make2DArray` 创建了二维数组 `B`，表示电路板之间的连接关系。然后调用 `BBArrangeBoards` 函数求解问题，并输出最小长度和对应的摆放方式。

▲代码

```
#include <array>
#include <bits/stdc++.h>
#include <queue>
using namespace std;
int n, *p;
template <class Type>
class MinHeap // 最小堆类;
{
public:
```

```
10      MinHeap(Type a[], int n);    // 带两参数的构造函数，在此程序中没有
    应用；
11      MinHeap(int ms);             // 构造函数重载，只初始化堆的大小，对
    堆中结点不初始化；另外，堆元素的存储是以数组
12      ~MinHeap();                  // 形式，且无父、子指针，访问父亲结
    点，利用数组标号进行；
13      bool Insert(const Type &x); // 插入堆中一个元素；
14      bool RemoveMin(Type &x);     // 删除堆顶最小结点；
15      void MakeEmpty();            // 使堆为空
16      bool IsEmpty();
17      bool IsFull();
18      int Size();
19
20  protected:
21      void FilterDown(const int start, const int endOfHeap); //
    自顶向下构造堆
22      void FilterUp(const int start);                          //
    自底向上构造堆
23  private:
24      Type *heap;
25      int maxSize;
26      const int defaultSize;
27      int currentSize; // 堆当前结点个数大小
28  };
29
30  template <class Type>
31  MinHeap<Type>::MinHeap(int ms) : defaultSize(100)
32  {
33      maxSize = (ms > defaultSize) ? ms : defaultSize;
34      heap = new Type[maxSize];
35      currentSize = 0;
36  }
37
38  template <class Type>
39  MinHeap<Type>::MinHeap(Type a[], int n) : defaultSize(100)
40  {
41      maxSize = (n > defaultSize) ? n : defaultSize;
42      heap = new Type[maxSize];
43      currentSize = n;
44      for (int i = 0; i < n; i++)
45          heap[i] = a[i];
46      int curPos = (currentSize - 2) / 2;
```

```
47      while (curPos >= 0)
48      {
49          FilterDown(curPos, currentSize - 1);
50          curPos--;
51      }
52  }
53
54  template <class Type>
55  MinHeap<Type>::~MinHeap()
56  {
57      delete[] heap;
58  }
59
60  template <class Type>
61  void MinHeap<Type>::FilterDown(const int start, const int
    endOfHeap)
62  {
63      int i = start, j = i * 2 + 1;
64      Type temp = heap[i];
65      while (j <= endOfHeap)
66      {
67          if (j < endOfHeap && heap[j] > heap[j + 1])
68              j++;
69          if (temp < heap[j])
70              break;
71          else
72          {
73              heap[i] = heap[j];
74              i = j;
75              j = 2 * i + 1;
76          }
77      }
78      heap[i] = temp;
79  }
80
81  template <class Type>
82  void MinHeap<Type>::FilterUp(const int start)
83  {
84      int i = start, j = (i - 1) / 2;
85      Type temp = heap[i];
86      while (i > 0)
87      {
```

```
88          if (temp >= heap[j])
89              break;
90          else
91          {
92              heap[i] = heap[j];
93              i = j;
94              j = (i - 1) / 2;
95          }
96      }
97      heap[i] = temp;
98  }
99
100 template <class Type>
101 bool MinHeap<Type>::RemoveMin(Type &x)
102 {
103     if (IsEmpty())
104     {
105         cerr << "Heap empty!" << endl;
106         return false;
107     }
108     x = heap[0];
109     heap[0] = heap[currentSize - 1];
110     currentSize--;
111     FilterDown(0, currentSize - 1);
112     return true;
113 }
114
115 template <class Type>
116 bool MinHeap<Type>::Insert(const Type &x)
117 {
118     if (IsFull())
119     {
120         cerr << "Heap Full!" << endl;
121         return false;
122     }
123     heap[currentSize] = x;
124     FilterUp(currentSize);
125     currentSize++;
126     return true;
127 }
128
129 template <class Type>
```

```
130 bool MinHeap<Type>::IsEmpty()
131 {
132     return currentSize == 0;
133 }
134 
135 template <class Type>
136 bool MinHeap<Type>::IsFull()
137 {
138     return currentSize == maxSize;
139 }
140 
141 template <class Type>
142 void MinHeap<Type>::MakeEmpty()
143 {
144     currentSize = 0;
145 }
146 
147 template <class Type>
148 int MinHeap<Type>::Size()
149 {
150     return currentSize;
151 }
152 
153 class BoardNode
154 {
155     friend int BBArrangeBoards(int **, int, int *&);
156 
157 public:
158     operator int() const { return cd; }
159     int len(int **, int ii);
160 
161 private:
162     int *x, s, cd;
163 };
164 
165 int BoardNode::len(int **conn, int ii)
166 {
167     int sum = 0;
168     for (int i = 1, sum = 0; i <= ii; i++)
169     {
170         for (int j = i + 1; j <= ii; j++)
171         {
```

```
            int dist = x[i] > x[j] ? x[i] - x[j] : x[j] - x[i];
            sum += conn[i][j] * dist;
        }
    }
    return sum;
}

int BBArrangeBoards(int **conn, int n, int *&bestx)
{
    int HeapSize = 10;
    MinHeap<BoardNode>
        H(HeapSize);
    BoardNode E;
    E.x = new int[n + 1];
    E.s = 0;
    E.cd = 0;
    for (int i = 1; i <= n; i++)
        E.x[i] = i;
    int bestd = INT_MAX;
    bestx = 0;
    while (E.cd < bestd)
    {
        if (E.s == n - 1)
        {
            int ld = E.len(conn, n);
            if (ld < bestd)
            {
                delete[] bestx;
                bestx = E.x;
                bestd = ld;
            }
            else
                delete[] E.x;
        }
        else
        {
            for (int i = E.s + 1; i <= n; i++)
            {
                BoardNode N;
                N.x = new int[n + 1];
                N.s = E.s + 1;
                for (int j = 1; j <= n; j++)
```

```
                    N.x[j] = E.x[j];
                N.x[N.s] = E.x[i];
                N.x[i] = E.x[N.s];
                N.cd = N.len(conn, N.s);
                if (N.cd < bestd)
                    H.Insert(N);
                else
                    delete[] N.x;
            }
        }
        delete[] E.x;
    }
    try
    {
        H.RemoveMin(E);
    }
    catch (...)
    {
        return bestd;
    }
    while (true)
    {
        delete[] E.x;
        try
        {
            H.RemoveMin(E);
        }
        catch (...)
        {
            break;
        }
    }
    return bestd;
}

template <class T>
void Make2DArray(T **&x, int rows, int cols)
{
    x = new T *[rows];
    for (int i = 0; i < rows; ++i)
    {
        x[i] = new T[cols];
```

```
256     }
257 }
258
259 int main()
260 {
261     cin >> n;
262     p = new int[n + 1];
263     int **B;
264     Make2DArray(B, n + 1, n + 1);
265     for (int i = 1; i <= n - 1; i++)
266         for (int j = i + 1; j <= n; j++)
267             cin >> B[i][j];
268     cout << BBArrangeBoards(B, n, p) << endl;
269     for (int i = 1; i <= n; i++)
270         cout << p[i] << " ";
271     cout << endl;
272     return 0;
273 }
274
```

▲验证

书上案例验证通过。