Day 3

分治法

步骤：

分解（Divide）：将原问题分解为若干子问题，这些子问题都是原问题规模较小的实例。
解决（Conquer）：递归地求解各子问题。如果子问题规模足够小，则直接求解。
合并（Combine）：将所有子问题的解合并为原问题的解

快速幂

$$a^n=\begin{cases} a^{\frac{n}{2}}\times a^{\frac{n}{2}}(n\text{ 为偶数}) \\ a^{\frac{n}{2}}\times a^{\frac{n}{2}}\times a(n\text{ 为奇数}) \end{cases}$$

时间复杂度为 O(log*n*)。

归并排序

```
void Merge(int a[],int s,int m, int e,int tmp[]){
 //将数组a的局部a[s,m]和a[m+1,e]合并到tmp,并保证tmp有序，然后再拷贝回
//a[s,m]
 //归并操作时间复杂度：O（e-m+1),即O（n)
  int pb = 0;
  int p1 = s,p2 = m+1;
  while( p1 <= m && p2 <= e) {
    if( a[p1] < a[p2])
        tmp[pb++] = a[p1++];
    else
        tmp[pb++] = a[p2++];
 }
    while( p1 <= m)
        tmp[pb++] = a[p1++];
    while( p2 <= e)
        tmp[pb++] = a[p2++];
    for(int i = 0;i < e-s+1; ++i)
        a[s+i] = tmp[i];
}
void MergeSort(int a[],int s,int e,int  tmp[]) {
    if( s < e) {
        int m = s + (e-s)/2;
        MergeSort(a,s,m,tmp);
        MergeSort(a,m+1,e,tmp);
        Merge(a,s,m,e,tmp);
    }
}
```

求逆序对个数

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

```
int cnt=0;
// 逆序对个数
int a[100002], c[100002];
void MergeSort(int l, int r){
// r＝右边界索引＋1
    int mid, i, j, tmp;
    if (r>l+1){
        mid = (l+r)/2;
```

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

快速排序

```
void QuickSort(int a[],int s,int e) {
    if( s >= e)
        return;
    int k = a[s];
    int i = s,j = e;
    while( i != j ) {
        while( j > i &&  a[j] >= k )
            --j;
        swap(a[i],a[j]);
        while( i < j && a[i] <= k )
            ++i;
        swap(a[i],a[j]);
    } //处理完后，a[i] = k
    QuickSort(a,s,i-1);
    QuickSort(a,i+1,e);
}
```

输出前 m 大的数

把前 m 大的都弄到数组最右边，然后对这最右边 m 个元素排序，再输出
关键 ： O(n)时间内实现把前 m 大的都弄到数组最右边
引入操作 arrangeRight(k): 把数组(或数组的一部分）前 k 大的都弄到最右边
如何将前 k 大的都弄到最右边

1）设 key=a[0], 将 key 挪到适当位置，使得比 key 小的元素都在 key 左边，比 key 大的元素都在 key 右边（线性时间完成）

2) 选择数组的前部或后部再进行 arrangeRight 操作（a＝n - key 所在的位置，b＝key 所在的位置）

a＝k　　done

a＞k　　对此 a 个元素再进行 arrangeRigth(k)

a＜k　　对左边 b 个元素再进行 arrangeRight(k-a)

贪心法

贪心选择性质

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。

动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。

对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解，否则得到的是近优解。

最优子结构性质

当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。

但是，需要注意的是，并非所有具有最优子结构性质的问题都可以采用贪心策略来得到最优解。

装载问题

简单的装载问题

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

解：将物品重量从小到大进行排序，优先挑重量轻的装入背包。

部分背包问题

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

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

乘船问题

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

解：让最轻的人和能与他同船的最重的人乘一条船。如果办不到，就一人一条船。

区间问题

选择不相交区间

数轴上有 n 个开区间(ai, bi)。选择尽量多的区间，使这些区间两两没有公共点。

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

区间选点问题

数轴上有 n 个闭区间[ai, bi]。取尽量少的点，使得每个区间内都至少有一个点（不同区间内含有的点可以是同一个）。

解：把所有区间按 bi 从小到大排序（bi 相同时，a 从大到小排序），然后一定取第一个区间的最后一个点。

区间覆盖问题

数轴上有 n 个闭区间[ai, bi]。选择尽量少的区间来覆盖指定线段[s, t]。

解：

1. 预处理：扔掉不能覆盖[s, t]的区间。

2. 把各区间按 a 从小到大排序。如果区间 1 的起点不是 s，无解，否则选择从 *s* 到终点的最长区间。
3. 选择此区间[ai, bi]后，问题转化了覆盖[bi, t]，于是返回①，直到[s, t]被完全覆盖为止。

删数问题

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

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

工序问题

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

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

种树问题

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

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

马的哈密尔顿链

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

解：每次往出度最小的点上跳。

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

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

int board[8][8];

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

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

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

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

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

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

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

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

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

三值的排序

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

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

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

【分析】

为了使交换次数最小，我们想到了以下贪心策略：

① 能不交换就不交换；

② 如果能只用一次交换就完成归位，就不用两次交换。

由于排序之后是 11……1122……2233……33 的形式，我们不妨按照排序之后的结果对原数据分区。令 $w(i,j)$表示数字 i 在 j 区的数量。例如 122 312 13 中 $w(2,1)=2$。

按照①的说法，在一区的 1、二区的 2、三区的 3 就不应该再被交换了。

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

设这一次交换的步数为 A，则

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

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

设这一次交换的步数为 B，则

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

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

最后将 A 和 B 相加，即最终结果。

田忌赛马

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

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

【分析】

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

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

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

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

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

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

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

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

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

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

动态规划

最优子结构性质

如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质。

在分析问题的最优子结构性质时，所用的方法具有普遍性：首先假设由问题的最优解导出的子问题的解不是最优的，然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。

利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。最优子结构是问题能用动态规划算法求解的前提（同一个问题可以有多种方式刻划它的最优子结构，有些表示方法的求解速度更快，即：空间占用小，问题的维度低）。

无后效性

当前的若干个状态值一旦确定，则此后过程的演变就只和这若干个状态的值有关，和之前是采取哪种手段或经过哪条路径演变到当前的这若干个状态，没有关系。

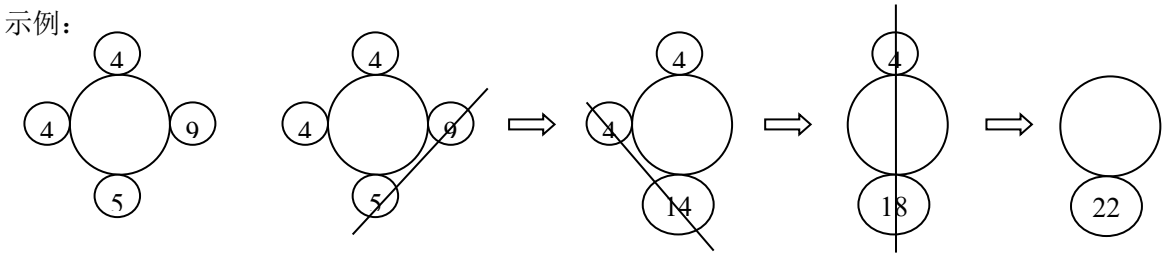
步骤

① 划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。
② 确定状态和状态变量：注意状态必须满足无后效性。
③ 确定决策：找到子问题是进行动态规划的重要一步。动态规划和递推更多应考虑本问题由哪些已解决子问题构成，而不是考虑本问题对将来哪些问题有贡献。
④ 确定边界条件，写出状态转移方程。
⑤ 编程
如果某一个问题是从已知的动态规划问题变形而来的，你可以考虑在原有的状态转移方程的基础上加一维。

区间问题：石子合并

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

示例：



这种合并方法的总得分＝14＋18＋22＝54

(1) 环的处理方法

以某一点作为起点，按顺时针或逆时针的顺序把环上的元素复制两遍。处理时注意起点范围（0～$n-1$）和最大长度（n）。
例如上面的示例，可以变成：5 9 4 4 5 9 4 4，这样就包含了分别以 5、9、4、4 为起点的 4 个环。

(2) 求解

递推思路：计算将第 i 堆至第 j 堆完全合并所能获得的最大得分。这是此题的关键。考虑模拟每种合并后的具体情形是行不通的。把问题变成这样后就好解决了。
划分阶段：以合并的次数作为标准划分阶段。
确定状态：$f_1(i,j)$表示第 i 堆至第 j 堆合并所能获得的最大价值，$f_2(i,j)$表示第 i 堆至第 j 堆合并所能获得的最小价值。
状态转移方程：
$$f_1(i,j)=\max\{f_1(i,k)+f_1(k+1,j)+d(i,j)\}$$
$$f_2(i,j)=\min\{f_2(i,k)+f_2(k+1,j)+d(i,j)\}，\text{其中 } 1\leqslant i\leqslant k<j\leqslant n$$
边界状态：$f_1(i,i)=f_2(i,i)=0$
$d(i,j)$表示当次合并的得分，即从 i 到 j 的石子数量，$d(i,j)=a[i]+a[i+1]+\cdots+a[j]$，可用前序和求出。
递推时注意，循环的最外层不是 i，也不是 j，而是 $j-i$！
最后从 $f_1(1,n)$到 $f_1(n,n+n)$，从 $f_2(1,n)$到 $f_2(n,n+n)$，找出最值即可。

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

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

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

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

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

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

时间复杂度：$O(n^3)$

能量项链

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

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

【分析】

思路是相似的——阶段、状态都一样，而得分的方式变了。

设第 i 个珠子的头标记是 value[i]，规定所有下标都是从 1 开始。

状态转移方程：设 $f(i,j)$为从 i 到 j 这一条链的最大值，那么

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

边界条件 $f(k,k)=0$。

坐标问题

单向取数问题

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

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

【分析】

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

划分阶段：每走一步为一个阶段。

状态表示：$f(r,c)$表示从起点出发走到第 r 行第 c 列时经过数字总和的最大值。

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

边界处理：让方格的下标从 1 开始，这样方格就可以多出一圈——0。

变式问题

① 必须经过某点

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

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

② 不能经过某点

一个简单的办法是：把那个点的值设置成－INF（负无穷大）。由于经过此点要“付

出巨大的代价”，所以状态不会从这点转移而来。

③ 竖直方向上最多有连续两个点相邻

解决方法：在状态中加一维，表示“已经连续向下走的次数”。

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

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

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

输出：$f(m,n,0)$和$f(m,n,1)$中的最大值。

④ 传送门#1

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

解决方法：分两种情况讨论，即经过传送门、不经过传送门。

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

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

不经过传送门

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

经过传送门

⑤ 传送门#2

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

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

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

不经过传送门

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

先进入左侧的传送门

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

先进入右侧的传送门

传纸条

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

【分析】

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

思路 1：

划分阶段：每走一步为一个阶段（一次只移动一张纸条）。

状态表示：设两张纸条分别位于第 r_1 行第 c_1 列、第 r_2 行第 c_2 列，那么 $f(r_1,c_1,r_2,c_2)$ 表示两张纸条从起点出发，分别到达这两个点时经过的数字和的最大值。

为了保证道路不交叉，应有 $x_1<r_2$。

状态转移方程：$f(r_1,c_1,r_2,c_2)=\max\begin{cases}f(r_1-1,c_1,r_2,c_2)+\text{map}(r_1,c_1) & (\downarrow\times)\\ f(r_1,c_1-1,r_2,c_2)+\text{map}(r_1,c_1) & (\rightarrow\times)\\ f(r_1,c_1,r_2-1,c_2)+\text{map}(r_2,c_2) & (\times\downarrow)\\ f(r_1,c_1,r_2,c_2-1)+\text{map}(r_2,c_2) & (\times\rightarrow)\end{cases}$

一个优化：因为 $r_1+c_1=r_2+c_2$，所以可以去掉一维。这样时间可由四次方降到三次方。

输出：f(m, n-1, m-1, n)

思路 2：

可以发现，只要知道移动步数和两个横坐标，那么两个纵坐标就可以计算出来。

划分阶段：每走一步为一个阶段。

与思路 1 不同的是，这一次两张纸条要同时移动。

状态表示：设两张纸条分别位于第 r_1 行第 c_1 列、第 r_2 行第 c_2 列，i 为移动步数，那么 $f(i,r_1,r_2)$表示两张纸条从起点出发，经过 i 步分别到达这两个点时经过的数字和的最大值。

为了保证道路不交叉，应有 $r_1>r_2$。

状态转移方程：$f(i,r_1,r_2)=\text{map}(r_1,c_1)+\text{map}(r_2,c_2)+\max\begin{cases}f(i\text{-}1,r_1-1,r_2-1) & (\downarrow\downarrow)\\ f(i\text{-}1,r_1-1,r_2) & (\downarrow\rightarrow)\\ f(i\text{-}1,r_1,r_2-1) & (\rightarrow\downarrow)\\ f(i\text{-}1,r_1,r_2) & (\rightarrow\rightarrow)\end{cases}$

$c_1=i+2-r_1$，$c_2=i+2-r_2$，同时 $r_1+c_1=r_2+c_2$。

输出：$f(m+n-3, m, m-1)$

$m+n-3$ 是从左上角出发，到右下角的移动步数。

背包问题

部分背包问题

部分背包问题是贪心算法问题。

0/1 背包问题

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

(1) 二维数组表示

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

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

时间复杂度、空间复杂度：都是 $O(NC)$

(2) 一维数组表示

定义状态：由于递推的过程和雪天环形路上的扫雪车类似，所以可以把 i 省略。

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

递推时注意 c 要从 C 开始，倒着推。

时间复杂度：$O(NC)$

空间复杂度：降到了 $O(C)$

(3) 一维之下的一个常数优化

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

(4) 初始化时的细节

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

如果不用“恰好”，那么应该让 f 的所有元素都置 0。

完全背包问题

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

(1) 基本算法

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

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

一个简单的优化：如果物品 i、j 满足 $w[i]\leqslant w[j]$且 $v[i]\geqslant v[j]$，就可以把物品 j 去掉。
不过它不能改善最坏情况的复杂度（最坏情况：根本没有可以去掉的东西）。
另一种优化：首先将重量大于 C 的物品去掉，然后使用类似桶排序或计数排序的方法，计算出费用相同的物品中哪个价值最高。

(2) 更优的算法

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

时间复杂度：$O(NC)$

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

多重背包问题

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

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

```
for (int i=1;i<n;i++){
    if (w[i]*a[i]>C){                          // 如果物品够多，问题其实就是完全背包问题
        for (int c=0;c<=C;c++)                 // 完全背包
            if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + v[i]);
    }
    else{
        int k=1, amount=a[i];
        while (k<amount){
            // 是否取一个重量为k×w[i]，价值为k×v[i]的物品？
            for (int c=k*w[i];c>=0;c--)
                if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + k*v[i]);
            // 继续分割
            amount-=k;
            k+=k;
        }
        // 把剩下的作为单独一个物品
        for (int c=amount*w[i];c>=0;c--)
```

```
            if (c>=w[i]) f[c] = max(f[c], f[c-w[i]] + amount*v[i]);
    }
}
```

时间复杂度：O($V\times\sum\log w[i]$)

二维费用的背包问题

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

(1) 0/1 背包的表示方法
费用加了一维，只需把状态也加一维。
状态表示：设 $f[i][c][u]$为前 i 件物品付出两种代价分别为 c 和 u 时可以获得的最大价值。
状态转移方程：$f[i][c][u]=\max\begin{cases}f[i-1][c][u]\\f[i-1][c-w[i]][u-u[i]]+v[i]\end{cases}$
当然，为了节省空间，可以把 i 去掉。
一个启示：当发现由熟悉的动态规划题目变形而来的题目时，在原来的状态中加一维以满足新的限制，这是一种比较通用的方法。

(2) 限制物品总个数的 0/1 背包
有 n 件物品和一个容量为 C 背包。第 i 件物品的重量是 $w[i]$，价值是 $v[i]$。现在要求**转入背包的物品个数不超过 M**。求解将哪些物品装入背包可使价值总和最大。

只需把问题变一下——有 N 件物品和一个容量为 C、容积为 M 的背包。第 i 件物品的重量是 $w[i]$，体积是 1，价值是 $v[i]$。求解将哪些物品装入背包可使价值总和最大。
把最大个数看做一种容积就可以了。

(3) 二维费用的完全背包和多重背包问题
循环时仍然按照完全背包（顺序循环）和多重背包（分割）的方法操作，只不过比完全背包和多重背包多了一维。

(4) 二维费用背包的另一种解法
把背包的容量和费用看作一个复数。详见《背包九讲》。

分组的背包问题

有 n 件物品和一个容量为 C 的背包。第 i 件物品的重量是 $w[i]$，价值是 $v[i]$。**这些物品被划分为 K 组，每组中的物品互相冲突，最多选一件。**求解将哪些物品

装入背包可使价值总和最大。

状态表示：设 $f[k][c]$ 为前 k 组物品花费 c 时可以获得的最大价值。

状态转移方程：$f[k][c]=\max\begin{cases}f[k-1][c]\\f[k-1][c-w[i]]+v[i]\end{cases}$ 物品i属于第k组

注意循环的顺序。

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

有依赖的背包问题

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

第一种思想：将每个主件及其附件集合转化为物品组。不过，由于附件可能还有附件，就不能将每个附件都看作一个一般的 01 背包中的物品了。若这个附件也有附件集合，则它必定要被先转化为物品组，然后用分组的背包问题，解出主件及其附件集合所对应的附件组中各个费用的附件所对应的价值。

第二种思想：每个父结点都需要对它的各个儿子的属性进行一次 DP 以求得自己的相关属性。

第三种思想：这已经触及到了“泛化物品”的思想，你会发现这个“依赖关系树”每一个子树都等价于一件泛化物品，求某结点为根的子树对应的泛化物品相当于求其所有儿子的对应的泛化物品之和。

泛化物品

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

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

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

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

可以看到，f 也是一个由泛化物品 h 和 l 决定的定义域为 $0\sim C$ 的函数，也就是说，f 是一个由泛化物品 h 和 l 决定的泛化物品，f 是 h 与 l 的和。这个运算的

时间复杂度取决于背包的容量，是 $O(V^2)$。

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

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

混合背包问题

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

由于我们按物品划分阶段，所以没有必要想太多。下面给出伪代码：

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

特殊要求

(1) 输出字典序最小的最优方案

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

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

(2) 求方案总数

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

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

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

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

(3) 最优方案的总数

这里的最优方案是指物品总价值最大的方案。以 01 背包为例。

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

代码如下：

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

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

(4) 求次优解、第 K 优解

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

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

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

然后可以说：$f[i][c]$这个有序队列是由①、②这两个有序队列合并得到的。有序队列①即 $f[i-1][c][K]$，②则理解为在 $f[i-1][c-w[i]][K]$的每个数上加上 $v[i]$后得到的有序队列。合并这两个有序队列并将结果的前 K 项储存到 $f[i][c][K+1]$

中的复杂度是 O(K)。最后的答案是 f[N][C][K]。总的复杂度是 O(CNK)。

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

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

编号问题

最长非降子序列（LIS）

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

【分析】

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

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

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

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

最长公共子序列（LCS）

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

【分析】

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

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

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

递归结构问题

乘积最大

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

【分析】

划分阶段：以一个乘号为一个阶段。

状态表示：$f(i,l)$表示前 i 个数字插入 l 个乘号之后的最大乘积。

状态转移方程：$f(i,l)=\max\{f(j,l-1)\times s(j+1,n)\}$，$l<j<i$，$l\leqslant\min\{k,i-1\}$

边界条件：$f(i,0)=s(1,i)$

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

```
#include <iostream>
#include <cstring>
using namespace std;
struct hp {……};          // 见 124 页“11.7  高精度算法（压位存储）!”。
int n, k;
hp f[51][21], s[51][51];

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

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

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

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

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

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

加分二叉树

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

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

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

【分析】

本题中的树是无根树，需要枚举节点作为根的情况，重复有根树的动态规划过程。

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

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

边界条件：$f(i,i)=d_i$

递推时注意，循环的最外层不是 i，也不是 j，而是 $j-i$！

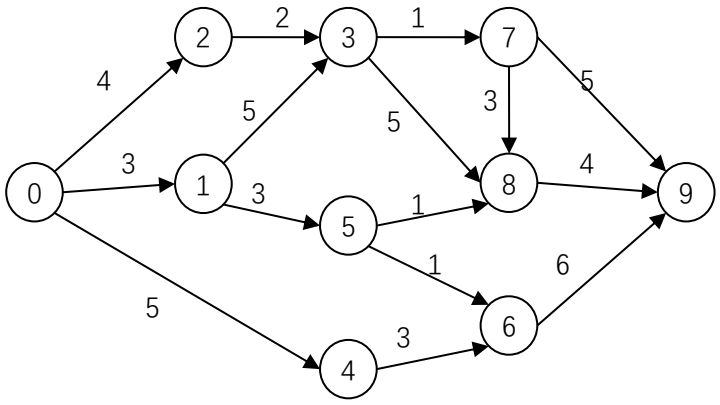
```cpp
#include <iostream>
#include <cstring>
using namespace std;

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

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

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

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

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

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

DAG 上的最短路径

如果用动态规划求 DAG 上的最短路径，应该先进行拓扑排序。

特殊 DAG 的最短路径

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



邻接矩阵（未填充部分为 ∞）

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

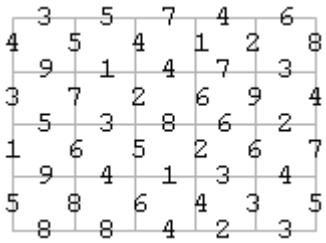
【分析】
状态表示：设 $f(x)$表示结点 0 到结点 x 的最短路径长度。
状态转移方程：$f(x)=\min(f(i)+G[i][x])$，其中结点 i 是结点 x 的前趋。
边界条件：$f(0)=0$

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

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

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

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

街道

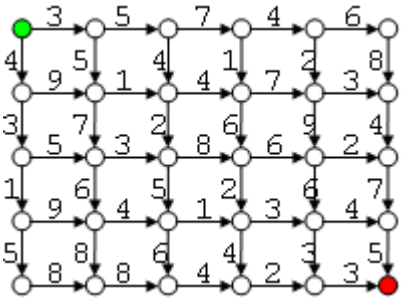
下图是一个 $m\times n$ 的街区。每条马路（最短的边算一条马路）上有一个数字。从左上角出发到右下角，路上只能往右或往下走。问经过的数字的和最大可以达到

多少。



【分析】

转化思路：构造出一个 DAG。



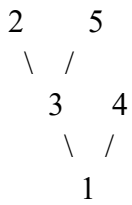
由于这道题道路整齐，所以求解起来更容易一些。

树形动态规划

苹果树

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

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



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

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

【分析】

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

状态转移方程：f(i,n)＝max{f(i 的左儿子,k)＋f(i 的右儿子,n－k)＋i→num}，其中 num 指该结点与父亲结点连接成的树枝上的苹果数，0≤k≤j
实现：由于要把儿子的信息传递给父亲，所以用后序遍历（下面的代码是后序遍历，虽然看起来不像）。

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

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

选课

某大学有 m 门功课，每门课有个学分，每门课有一门或没有直接先修课（若课程 a 是课程 b 的先修课，那么只有学完了课程 a，才能学习课程 b）。一个学生要从这些课程里选择 n 门课程学习，他能获得的最大学分是多少？
【输入格式】第一行有两个整数 m，n 用空格隔开。(1≤n≤m≤1000)
接下来的 m 行：第 I＋1 行包含两个整数 k_i 和 s_i，k_i 表示第 I 门课的直接先修课，s_i 表示第 I 门课的学分。若 k_i＝0 表示没有直接先修课（1≤k_i≤m，1≤s_i≤10）。
【输出格式】只有一个数，是选 n 门课程的最大得分。

【分析】
状态表示：f(i,c)表示在以 i 为根结点的二叉树中取 c 门课程后得到的学分的最大值。
状态转移方程：
f(i,c)＝max{f(i->leftchild, k－1)＋i->v＋f(i->rightchild, c－k)}，0≤k≤j
实现：
把 0 作为顶点。
需要把多叉树转化为二叉树（左儿子右兄弟）。
后序遍历

```
#include <iostream>
#include <cstring>
using namespace std;
// -1 表示没有结点。
```

```
struct node{
    int value, leftchild, rightchild;
} a[1002];
int F[1002][152], parent[1002];
int m,n;
#define f(x,y) F[(x)+1][(y)+1]

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

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

    // 树变二叉树
    int k,s;
    for (int i=1; i<=m; i++){
        cin>>k>>s;
        a[i].value=s;
        if (parent[k]==0)
            a[k].leftchild=i;
        else
            a[parent[k]].rightchild=i;
        parent[k]=i;
    }
    // 递推的边界处理
    for (int i=-1; i<=m; i++)
        for (int j=-1; j<=n; j++)
            f(i,j) = (i==-1 || j==0) ? 0: (-1);

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

状态压缩类问题：过河

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

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

【分析】

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

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

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

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

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

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

接下来就可以用动态规划了。

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

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

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

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

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

```
        if(l<k) k=l;
        p+=k;
        map[p]=1;
    }

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

Bitonic 旅行

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

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

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

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

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

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