第5章 动态规划

动态规划(Dynamic Programming,DP)是算法竞赛的必考题型,内容丰富多变[①]。本章将全面介绍动态规划的思想、模板、应用场景和优化等内容。

动态规划是地道的“计算思维”,非常适合用计算机实现,可以说是独属于计算机学科的基础理论。与贪心法、分治一样,动态规划是一种解题的思路,而不是一个具体的算法知识点。动态规划是一种需要学习才能获得的思维方法。像贪心法、分治这样的方法,在生活中和其他学科中有很多类似的例子,很容易联想和理解。但是动态规划不同,它是一种生活中没有的抽象计算方法,没有学过的人很难自发产生这种思路。

本章详解竞赛相关的大部分 DP 知识点,具体如下。

(1) DP 的基本概念和编程方法。

(2) 常见的线性 DP 问题。

(3) 特殊场景的 DP 应用,包括数位统计 DP、状态压缩 DP、区间 DP、树形 DP 等。

(4) DP 优化,包括一般优化、单调队列优化、斜率优化、四边形不等式优化等。

① 1950 年,Richard Bellman 把多阶段决策过程命名为 Dynamic Programming。闫学灿在网上发布的算法指导视频中提出“闫氏 DP 分析法”,用画图辅助 DP 的建模过程,受到网友的推崇。

扫一扫

视频讲解

5.1 DP 概念和编程方法

动态规划(DP)是一种算法技术,它将大问题分解为更简单的子问题,对整体问题的最优解决方案取决于子问题的最优解决方案。动态规划常用于求解计数问题(求方案数)和最值问题(最大价值、最小花费)等。

本节介绍 DP 的基础知识,包括 DP 的特征、DP 的编程方法、DP 状态的设计和状态方程的推导,以及 DP 的空间优化滚动数组。

5.1.1 DP 的概念

DP 是求解多阶段决策问题最优化的一种算法思想,它用于解决具有重叠子问题、最优子结构特征的问题。

下面以斐波那契数列为例说明 DP 的概念。

斐波那契数列是一个递推数列,它的每个数字是前面两个数字的和,如 1,1,2,3,5,8…计算第 n 个斐波那契数,递推公式为

$$\text{fib}(n)=\text{fib}(n-1)+\text{fib}(n-2)$$

斐波那契数列的应用场景是走楼梯问题:一次可以走一个或两个台阶,问走到第 n 个台阶时,一共有多少种走法?走楼梯问题的数学模型是斐波那契数列。要走到第 n 级台阶,分为两种情况,一种是从 $n-1$ 级台阶走一步过来,另一种是从 $n-2$ 级台阶走两步过来。

用递归编程求斐波那契数列,代码如下。

```
1  int fib (int n){
2      if (n == 1 || n == 2)       return 1;
3      return (fib (n-1) + fib (n-2));   //递归以 2 的倍数递增
4  }
```

代码中的递归以 2 的倍数递增,复杂度为 $O(2^n)$,非常差。用 DP 可以优化复杂度。

为了解决总体问题 fib(n),将其分解为两个较小的子问题,即 fib($n-1$)和 fib($n-2$),这就是 DP 的应用场景。

一些问题具有两个特征:重叠子问题、最优子结构。用 DP 可以高效率地处理具有这两个特征的问题。

1. 重叠子问题

首先,子问题是原大问题的小版本,计算步骤完全一样;其次,计算大问题时,需要多次重复计算小问题。这就是重叠子问题。以斐波那契数列为例,递归计算 fib(5),分解为如图 5.1 所示的子问题。

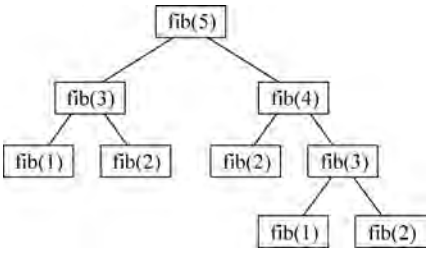

图 5.1 fib(5)分解

其中,fib(3)计算了两次,其实只计算一次就够了。

一个子问题的多次重复计算，耗费了大量时间。用 DP 处理重叠子问题，每个子问题只计算一次，从而避免了重复计算，这就是 DP 效率高的原因。具体的做法是首先分析得到最优子结构，然后用递推或带记忆化搜索的递归进行编程，从而实现高效的计算。

DP 在获得时间高效率的同时，可能耗费更多的空间，即时间效率高，空间耗费大。滚动数组是优化空间效率的一个办法。

2. 最优子结构

首先，大问题的最优解包含小问题的最优解；其次，可以通过小问题的最优解推导出大问题的最优解。这就是最优子结构。在斐波那契数列问题中，把数列的计算构造成 $fib(n)=fib(n-1)+fib(n-2)$，即把原来为 n 的大问题，减小为 $n-1$ 和 $n-2$ 的小问题，这是斐波那契数列的最优子结构。

在 DP 的概念中，还常常提到“**无后效性**”。简单地说，就是“未来与过去无关”。此概念不太容易理解，下面以走楼梯问题为例进行解释。要走到第 n 级台阶，有两种方法，一种是从第 $n-1$ 级台阶走一步过来，另一种是从第 $n-2$ 级台阶走两步过来。但是，前面是如何走到第 $n-1$ 级或第 $n-2$ 级台阶，$fib(n-1)$ 和 $fib(n-2)$ 是如何计算得到的，并不需要知道，只需要它们的计算结果就行了。换句话说，只关心前面的结果，不关心前面的过程，在计算 $fib(n)$ 时，直接使用 $fib(n)$ 和 $fib(n-1)$ 的结果，不需要知道它们的计算过程，这就是无后效性。

无后效性是应用 DP 的必要条件，因为只有这样，才能降低算法的复杂度，应用 DP 才有意义。如果不满足无后效性，那么在计算 $fib(n)$ 时，还需要重新计算 $fib(n-1)$ 和 $fib(n-2)$，算法并没有优化。

从最优子结构的概念可以看出，它是满足无后效性的。

这里用斐波那契数列举例说明 DP 的概念，可能过于简单，不足以说明 DP 的特征。建议读者用后文的“0/1 背包”经典问题重新理解 DP 的特征。

5.1.2 DP 的两种编程方法

处理 DP 中的大问题和小问题，有两种思路：自顶向下（Top-Down，先大问题，再小问题）、自底向上（Bottom-Up，先小问题，再大问题）。

编码实现 DP 时，自顶向下用带记忆化搜索的递归编码，自底向上用递推编码。两种方法的复杂度是一样的，每个子问题都计算一遍，而且只计算一遍。

1. 自顶向下与记忆化

先考虑大问题，再缩小到小问题，递归很直接地体现了这种思路。为避免递归时重复计算子问题，可以在子问题得到解决时就保存结果，再次需要这个结果时，直接返回保存的结

果就可以了。这种存储已经解决的子问题的结果的技术称为记忆化(Memoization[①])。

以斐波那契数列为例,记忆化代码如下。

```
1  int memoize[N];                                          //保存结果
2  int fib (int n){
3      if (n == 1 || n == 2)  return 1;
4      if(memoize[n] != 0)    return memoize[n];          //直接返回保存的结果,不再递归
5      memoize[n] = fib (n - 1) + fib (n - 2);           //递归计算结果,并记忆
6      return memoize[n];
7  }
```

在这段代码中,一个斐波那契数列只计算一次,所以总复杂度为 $O(n)$。

2. 自底向上与制表递推

这种方法与递归的自顶向下相反,避免了用递归编程。自底向上的方法先解决子问题,再递推到大问题,通常通过填写多维表格来完成,编码时用若干 for 循环语句填表,根据表中的结果,逐步计算出大问题的解决方案。

用制表法计算斐波那契数列,维护一张一维表 dp[],记录自底向上的计算结果,更大的数是前面两个数的和,如下所示。

dp[1]	dp[2]	dp[3]	dp[4]	dp[5]	dp[6]	dp[7]	dp[8]	…
1	1	2	3	5	8	13	21	…

代码如下。

```
1  const int N = 255;
2  int dp[N];
3  int fib (int n){
4      dp[1] = dp[2] =1;
5      for (int i=3;i<=n;i++)  dp[i] = dp[i-1] +dp[i-2];
6      return dp[n];
7  }
```

代码的复杂度显然也为 $O(n)$。

制表递推编程时,超过 4 维(dp[][][][])的表格也是常见的,表格可以用滚动数组优化空间。

对比自顶向下和自底向上这两种方法,自顶向下的优点是能更宏观地把握问题、认识问题的实质;自底向上的优点是编码更直接。两种编码方法都很常见。

① 这个英文单词没有写错。https://www.diffbt.com/memoization-vs-tabulation/解释:In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

5.1.3 DP的设计和实现

本节以0/1背包问题为例，详细解释与DP的设计、编程有关的内容。滚动数组也应是本节的内容，但是因为比较重要，所以后面单独用一节介绍。

背包问题在DP中很常见，其中0/1背包问题是最基础的，其他背包问题都由它衍生出来[①]。

0/1背包问题：给定n种物品和一个背包，第i个物品的体积为c_i，价值为w_i，背包的总容量为C。把物品装入背包时，第i种物品只有两种选择：装入背包或不装入背包，称为0/1背包问题。如何选择装入背包的物品，使装入背包中的物品的总价值最大？

设x_i表示物品i装入背包的情况：$x_i=0$时，不装入背包；$x_i=1$时，装入背包。定义：

约束条件：$\sum_{i=1}^{n} c_i x_i \leqslant C, x_i = 0,1$

目标函数：$\max \sum_{i=1}^{n} w_i x_i$

下面给出一道0/1背包的模板题，以此题为例进行基本DP的讲解。

例5.1 Bone collector(hdu 2602)

问题描述：骨头收集者带着体积为C的背包去捡骨头，已知每块骨头的体积和价值，求能装进背包的最大价值。

输入：第1行输入测试数量；后面每3行为一个测试，其中第1行输入骨头数量N和背包体积C，第2行输入每块骨头的价值w，第3行输入每块骨头的体积c。

输出：最大价值。

数据范围：$N \leqslant 1000, C \leqslant 1000$。

1. DP状态的设计

引入一个$(N+1)\times(C+1)$的二维数组dp[][]，称为DP状态，dp[i][j]表示把前i个物品(从第1个到第i个)装入容量为j的背包中获得的最大价值。可以把每个dp[i][j]都看作一个背包：背包容量为j，装1～i这些物品。最后的dp[N][C]就是问题的答案——把N个物品装进容量C的背包。

2. DP转移方程

用自底向上的方法计算，假设现在递推到dp[i][j]，分两种情况：

(1) 第i个物品的体积比容量j还大，不能装进容量j的背包。那么直接继承前$i-1$个物品装进容量j的背包的情况即可，即dp[i][j]=dp[$i-1$][j]。

① 崔添翼《背包问题九讲》(https://github.com/tianyicui/pack)介绍了一些背包问题。读者可以阅读这篇文章，如果有费解的地方，请对照本章节对各种背包问题的解释。

(2) 第 i 个物品的体积比容量 j 小，能装进背包。又可以分为两种情况：装或不装第 i 个物品。

① 装第 i 个物品。从前 $i-1$ 个物品的情况推广而来，前 $i-1$ 个物品的价值为 dp[$i-1$][j]。第 i 个物品装进背包后，背包容量减少 $c[i]$，价值增加 $w[i]$，有 dp[i][j]=dp[$i-1$][$j-c[i]$]+$w[i]$。

② 不装第 i 个物品，有 dp[i][j]=dp[$i-1$][j]。

取两种情况中的最大值，状态转移方程为

$$dp[i][j]=\max(dp[i-1][j],dp[i-1][j-c[i]]+w[i])$$

总结上述分析，0/1 背包问题的重叠子问题是 dp[i][j]，最优子结构是 dp[i][j]的状态转移方程。

算法复杂度：算法需要计算二维矩阵 dp[][]，二维矩阵的大小为 $O(NC)$，每项计算时间为 $O(1)$，总时间复杂度为 $O(NC)$，空间复杂度为 $O(NC)$。

0/1 背包问题的**简化版**：一般物品具有体积(或重量)和价值两个属性，求满足体积约束条件下的最大价值。如果再简单一点，只有一个体积属性，求能放到背包的最多物品，那么，只要把体积看作价值，求最大体积就好了。状态转移方程变为

$$dp[i][j]=\max(dp[i-1][j],dp[i-1][j-c[i]]+c[i])$$

3. 详解 DP 的转移过程

初学者可能对上面的描述仍不太清楚，下面用一个例子详细说明。有 4 个物品，其体积分别为{2,3,6,5}，价值分别为{6,3,5,4}，背包的容量为 9。

填写 dp[][]表的过程，按照只装第 1 个物品，只装前 2 个物品，只装前 3 个物品…的顺序，一直到装完，这就是从小问题扩展到大问题的过程。表格横向为 j，纵向为 i，按先横向递增 j，再纵向递增 i 的顺序填表。dp[][]矩阵如图 5.2 所示。

步骤 1：只装第 1 个物品。

由于物品 1 的体积为 2，所以背包容量小于 2 的，都放不进去，即 dp[1][0]=dp[1][1]=0。

若物品 1 的体积等于背包容量，能放进去，背包价值等于物品 1 的价值，即 dp[1][2]=6。

容量大于 2 的背包，多余的容量用不到，所以价值与容量为 2 的背包一样，如图 5.3 所示。

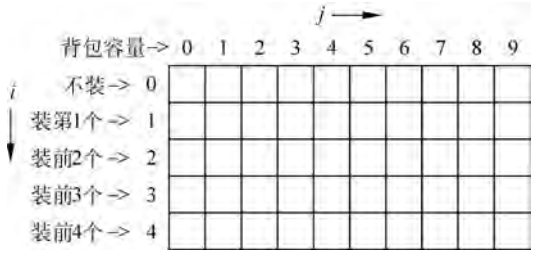

图 5.2　dp[][]矩阵

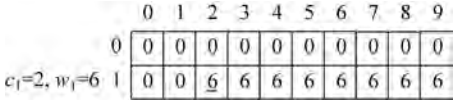

图 5.3　装第 1 个物品

步骤 2：只装前 2 个物品。

如果物品 2 体积比背包容量大，那么不能装物品 2，情况与只装第 1 个物品一样。

dp[2][0]=dp[2][1]=0,dp[2][2]=6。

下面填写 dp[2][3]。物品 2 体积等于背包容量,那么可以装物品 2,也可以不装。

如果装物品 2(体积为 3,价值也为 3),那么可以变成一个更小的问题,即只把物品 1 装到容量为 $j-3$ 的背包中,如图 5.4 所示。

如果不装物品 2,那么相当于只把物品 1 装到背包中,如图 5.5 所示。

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
c_1=2, w_1=6 1	0	0	6	6	6	6	6	6	6	6
c_2=3, w_2=3 2	0	0	6	3+0						

图 5.4 装第 2 个物品

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
c_1=2, w_1=6 1	0	0	6	6	6	6	6	6	6	6
c_2=3, w_2=3 2	0	0	6	6						

图 5.5 不装第 2 个物品

取两种情况的最大值,得 dp[2][3]=max{3,6}=6。

后续步骤:继续以上过程,最后得到如图 5.6 所示的 dp 矩阵(图中的箭头是几个例子)。

最后的答案是 dp[4][9],把 4 个物品装到容量为 9 的背包,最大价值为 11。

4. 输出背包方案

现在回头看具体装了哪些物品。需要倒过来观察:

dp[4][9]=max{dp[3][4]+4,dp[3][9]}=dp[3][9],说明没有装物品 4,用 $x_4=0$ 表示;

dp[3][9]=max{dp[2][3]+5,dp[2][9]}=dp[2][3]+5=11,说明装了物品 3,$x_3=1$;

dp[2][3]=max{dp[1][0]+3,dp[1][3]}=dp[1][3],说明没有装物品 2,$x_2=0$;

dp[1][3]=max{dp[0][1]+6,dp[0][3]}=dp[0][1]+6=6,说明装了物品 1,$x_1=1$。

如图 5.7 所示,实线箭头标识了方案的转移路径。

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
c_1=2, w_1=6 1	0	0	6	6	6	6	6	6	6	6
c_2=3, w_2=3 2	0	0	6	6	6	9	9	9	9	9
c_3=6, w_3=5 3	0	0	6	6	6	9	9	9	11	11
c_4=5, w_4=4 4	0	0	6	6	6	9	9	10	11	11

图 5.6 完成 dp 矩阵

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
c_1=2, w_1=6 1	0	0	6	6	6	6	6	6	6	6
c_2=3, w_2=3 2	0	0	6	6	6	9	9	9	9	9
c_3=6, w_3=5 3	0	0	6	6	6	9	9	9	11	11
c_4=5, w_4=4 4	0	0	6	6	6	9	9	10	11	11

图 5.7 背包方案

5. 递推代码和记忆化代码

下面的代码分别用自底向上的递推和自顶向下的记忆化递归实现。

1) 递推代码

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  const int N = 1011;
```

```
int w[N], c[N];                                          // 物品的价值和体积
int dp[N][N];
int solve(int n, int C){
    for(int i = 1; i <= n; i++)
        for(int j = 0; j <= C; j++){
            if(c[i] > j)  dp[i][j] = dp[i-1][j];     //第 i 个物品比背包还大,装不了
            else  dp[i][j] = max(dp[i-1][j],dp[i-1][j-c[i]]+w[i]);//第 i 个物品能装
    }
    return dp[n][C];
}
int main(){
    int T; cin>>T;
    while(T--){
        int n,C;   cin >> n >> C;
        for(int i = 1;i <= n;i++) cin>>w[i];
        for(int i = 1;i <= n;i++) cin>>c[i];
        memset(dp,0,sizeof(dp));                         //清零,置初值为 0
        cout << solve(n, C) << endl;
    }
    return 0;
}
```

2）记忆化代码

记忆化代码只改动递推代码中的 solve()函数。

```
int solve(int i, int j){                    //前 i 个物品,放进容量 j 的背包
    if (dp[i][j] != 0) return dp[i][j];     //记忆化
    if(i == 0) return 0;
    int res;
    if(c[i] > j) res =   solve(i-1,j);    //第 i 个物品比背包还大,装不了
    else  res = max(solve(i-1,j), solve(i-1,j-c[i])+w[i]);   //第 i 个物品可以装
    return dp[i][j] = res;
}
```

本节的 0/1 背包问题，物品价值都大于 0，且背包容量有最大限制。在 6.6 节“0/1 分数规划”给出的例题 Talent show G(洛谷 P4377)中，物品的价值可以为负数，且背包容量有最小限制，请思考这种背包问题并参考第 6 章的代码。

5.1.4 滚动数组

滚动数组是 DP 最常使用的空间优化技术。

DP 的状态方程常常是二维和二维以上，占用了太多空间。例如，5.1.3 节的代码使用了二维矩阵 int dp[N][C]，设 $N=10^3$，$C=10^4$，都不算大，但 int 型占据 4B，矩阵需要的空间为 $4\times10^3\times10^4\approx40$MB，已经超过一般竞赛题的空间限制。

用滚动数组可以大大减少空间。它能把二维状态方程 $O(n^2)$的空间复杂度优化到一维的 $O(n)$，更高维的数组也可以优化后减少一维。

从状态转移方程 dp[i][j]=max(dp[$i-1$][j],dp[$i-1$][$j-c[i]$]+$w[i]$)可以看出,dp[i][]只与 dp[$i-1$][]有关,和前面的 dp[$i-2$][],dp[$i-3$][],…都没有关系。从前面的图表也可以看出,每行是通过上面一行算出来的,与更前面的行没有关系。那些用过的已经无用的 dp[$i-2$][],dp[$i-3$][],…多余了,那么干脆就复用这些空间,用新的一行覆盖已经无用的一行(滚动),只需要两行就够了。

下面给出滚动数组的两种实现方法[①],两种实现方法都很常用。

1. 交替滚动

定义 dp[2][j],用 dp[0][]和 dp[1][]交替滚动。这种方法的优点是逻辑清晰,编码不易出错,**建议**初学者采用这种方法。

下面的代码中,now 始终指向正在计算的最新的一行,old 指向已计算过的旧的一行。对照原递推代码,now 相当于 i,old 相当于 $i-1$。

```
// hdu 2602(滚动数组代码 1)
int dp[2][N];                          //替换 int dp[][];
int solve(int n, int C){
    int now = 0, old = 1;              //now 指向当前正在计算的一行,old 指向旧的一行
    for(int i = 1; i <= n; i++){
        swap(old,now);                 //交替滚动,now 始终指向最新的一行
        for(int j = 0; j <= C; j++){
            if(c[i] > j)  dp[now][j] = dp[old][j];
            else          dp[now][j] = max(dp[old][j], dp[old][j - c[i]] + w[i]);
        }
    }
    return dp[now][C];                 //返回最新的行
}
```

注意,j 循环是 0～C,其实**反过来也可以**。但是在下面的“自我滚动”代码中,必须反过来循环,即 C～0。

2. 自我滚动

用两行做交替滚动在逻辑上很清晰,但是还能继续精简:一维 dp[]就够了,自己滚动自己。

```
// hdu 2602(滚动数组代码 2)
int dp[N];
int solve(int n, int C){
    for(int i = 1; i <= n; i++)
        for(int j = C; j >= c[i]; j--)       //反过来循环
            dp[j] = max(dp[j],dp[j - c[i]] + w[i]);
    return dp[C];
}
```

注意,j 应该反过来循环,即从后向前覆盖。下面说明原因,用 dp[j]′表示旧状态,dp[j]

① “交替滚动”和“自我滚动”是本书提出的名词。

表示滚动后的新状态。

(1) j 从小到大循环是错误的。例如，$i=2$ 时，图 5.8 左侧的 dp[5]，经计算得到 dp[5]=9，把 dp[5]更新为 9。继续计算，当计算 dp[8]时，得 dp[8]=dp[5]′+3=9+3=12，这个答案是错的。错误的产生是由动数组重复使用同一个空间引起的。

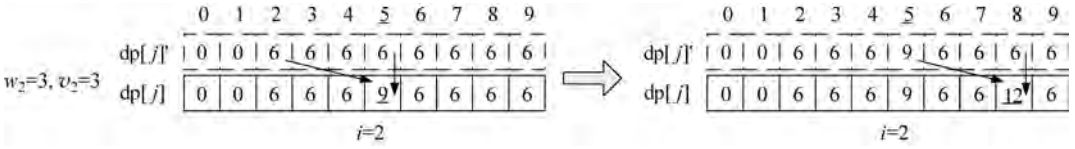

图 5.8 j 从小到大循环，是错误的

(2) j 从大到小循环是对的。例如，$i=2$ 时，首先计算最后的 dp[9]=9，它不影响前面状态的计算，如图 5.9 所示。

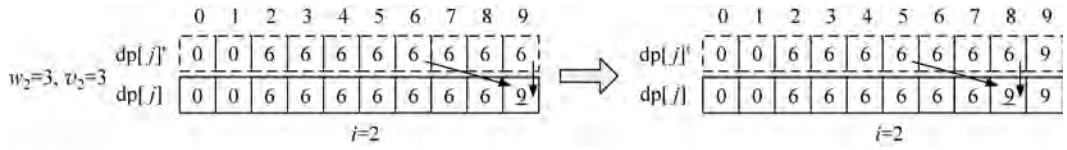

图 5.9 j 从大到小循环，是正确的

经过交替滚动或自我滚动的优化，DP 的空间复杂度从 $O(N\times C)$ 降低到 $O(C)$。

> 提示 滚动数组也有缺点。它覆盖了中间转移状态，只留下了最后的状态，所以损失了很多信息，导致无法输出具体的方案。

二维以上的 dp 数组也能优化。例如，求 dp[t][][]，如果它只和 dp[$t-1$][][]有关，不需要 dp[$t-2$][][]、dp[$t-3$][][]等，那么可以把数组缩小为 dp[2][][]或 dp[][]。

扫一扫

视频讲解

5.2 经典线性 DP 问题

本节介绍一些经典 DP 问题，这些问题比较简单，常常出现在面试中[①]。请大量练习这种题目，熟练掌握 DP 的算法思想。

1. 分组背包

有一些物品，把物品分为 n 组，其中第 i 组第 k 个物品的体积为 $c[i][k]$，价值为 $w[i][k]$；每组内的物品冲突，每组内最多只能选出一个物品装进背包；给定一个容量为 C 的背包，问如何选物品，使装进背包的物品的总价值最大。

解题思路与 0/1 背包问题很相似。回顾 0/1 背包问题的状态定义 dp[i][j]，它表示把前 i 个物品(第 1～i 个)装入容量为 j 的背包中获得的最大价值。

① 更多动态规划问题可参考 https://www.geeksforgeeks.org/dynamic-programming/。

类似地，在分组背包中定义状态 dp[i][j]，它表示把前 i 组物品装进容量 j 的背包（每组最多选一个物品）可获得的最大价值。状态转移方程为

$$dp[i][j]=\max\{dp[i-1][j],dp[i-1][j-c[i][k]]+w[i][k]\}$$

其中，dp[$i-1$][j]表示第 i 组不选物品；dp[$i-1$][$j-c[i][k]$]表示第 i 组选第 k 个物品。求解方程，需要做 i、j、k 的三重循环。

如果用滚动数组实现，状态转移方程变为

$$dp[j]=\max\{dp[j],dp[j-c[i][k]]+w[i][k]\}$$

下面是一道简单例题，用来演示代码。

例 5.2　ACboy needs your help（hdu 1712）

问题描述：ACboy 这学期可以选 N 门课，他只想学 M 天。每门课的学分不同，问这 M 天如何安排 N 门课，才能得到最多学分？

输入：有多个测试。每个测试的第 1 行输入 N 和 M。后面有 N 行，每行输入 M 个数字，表示一个矩阵 $A[i][j]$，$1\leqslant i\leqslant N\leqslant 100$，$1\leqslant j\leqslant M\leqslant 100$，表示第 i 门课学 j 天能得到 $A[i][j]$学分。若 $N=M=0$，表示测试结束。

输出：对每个测试，输出最多学分。

以第 1 个测试为例，$N=M=2$，第 1 门课学 1 天得 1 分，学 2 天得 2 分；第 2 门课学 1 天得 1 分，学 2 天得 3 分；ACboy 的选择是用 2 天学第 2 门课。本题可以建模为分组背包，每门课是一组。

物品分组：第 i 门课是第 i 组，天数是物品的体积，$A[i][j]$是第 j 个物品的价值。

背包容量：总天数 M 就是背包容量。

下面是用滚动数组实现的代码，注意它是**“自我滚动”**，所以 j 是从大到小循环的。

```
#include<bits/stdc++.h>
using namespace std;
const int N=105;
int w[N][N],c[N][N];                          //物品的价值、体积
int dp[N];
int n,m;
int main(){
    while(scanf("%d%d",&n,&m) && n && m){//输入：n门课，即n组；m天，即容量为m
        for(int i=1;i<=n;i++)                 //第i门课，即第i组
            for(int k=1;k<=m;k++){            //m也是第i组的物品个数
                scanf("%d",&w[i][k]);         //第i组第k个物品的价值
                c[i][k] = k;      //第i组第k个物品的体积，学k天才能得分，体积就是k
            }
        memset(dp,0,sizeof(dp));
        for(int i=1;i<=n;i++)                 //n门课，即n组；遍历每门课，即遍历每个组
            for(int j=m;j>=0;j--)             //容量为m
                for(int k=1;k<=m;k++)         //用k遍历第i组的所有物品
                    if(j>= c[i][k])           //第k个物品能装进容量j的背包
                        dp[j] = max(dp[j], dp[j-c[i][k]] + w[i][k]); //第i组第k个
```

```
        printf("%d\n",dp[m]);
    }
}
```

2. 多重背包

给定 n 种物品和一个背包，第 i 种物品的体积为 c_i，价值为 w_i，并且有 m_i 个，背包的总容量为 C。如何选择装入背包的物品，使装入背包中物品的总价值最大？

例 5.3　宝物筛选（洛谷 P1776）

输入：第 1 行输入整数 n 和 C，分别表示物品种数和背包的最大容量。接下来 n 行中，每行输入 3 个整数 w_i、c_i、m_i，分别表示第 i 个物品的价值、体积、数量。

输出：输出一个整数，表示背包不超载的情况下装入物品的最大价值。

下面给出 3 种解法。

1）简单方法

下面给出两种思路。

第 1 种思路是转换为 0/1 背包问题。把相同的 m_i 个第 i 种物品看作独立的 m_i 个物品，共有 $\sum_{i=1}^{n} m_i$ 个物品，然后按 0/1 背包问题求解，复杂度为 $O(C\sum_{i=1}^{n} m_i)$。

第 2 种思路是直接求解。定义状态 dp[i][j]表示把前 i 个物品装进容量 j 的背包，能装进背包的最大价值。第 i 个物品分为装或不装两种情况，得到多重背包的状态转移方程为

$$\mathrm{dp}[i][j]=\max\{\mathrm{dp}[i-1][j],\mathrm{dp}[i-1][j-k\cdot c[i]]+k\cdot w[i]\},$$
$$1\leqslant k\leqslant \min\{m[i],j/c[i]\}$$

直接写 i、j、k 三重循环，复杂度与第 1 种思路一样。下面用滚动数组编码，提交判题后会超时。

```
//洛谷 P1776: 滚动数组版本的多重背包(超时)
#include<bits/stdc++.h>
using namespace std;
const int N=100010;
int n,C,dp[N];
int w[N],c[N],m[N];                          //物品 i 的价值、体积、数量
int main(){
    cin>>n>>C;                               //物品数量,背包容量
    for(int i=1;i<=n;i++) cin>>w[i]>>c[i]>>m[i];
//以下是滚动数组版本的多重背包
    for(int i=1;i<=n;i++)                    //枚举物品
        for(int j=C;j>=c[i];j--)             //枚举背包容量
            for(int k=1; k<=m[i] && k*c[i]<=j; k++)
```

```
14                  dp[j] = max(dp[j],dp[j-k*c[i]]+k*w[i]);
15      cout << dp[C] << endl;
16      return 0;
17  }
```

2）二进制拆分优化

这是一种简单而有效的技巧。在上述简单方法的基础上加入这个优化，能显著改善复杂度。原理很简单，例如，第 i 种物品有 $m_i=25$ 个，这 25 个物品放进背包的组合有 0～25 的 26 种情况。不过，要组合成 26 种情况，其实并不需要 25 个物品。根据二进制的计算原理，任何一个十进制整数 X 都可以用 1，2，4，8 等 2 的倍数相加得到，如 $25=16+8+1$，这些 2 的倍数只有 $\log_2 X$ 个。题目中第 i 种物品有 m_i 个，用 $\log_2 m_i$ 个数就能组合出 0～m_i 种情况。总复杂度从 $O(C\sum_{i=1}^{n} m_i)$ 优化到 $O(C\sum_{i=1}^{n} \log_2 m_i)$。

注意拆分的具体实现，不能全部拆成 2 的倍数，而是先按 2 的倍数从小到大拆，最后是一个小于或等于最大倍数的余数。对 m_i 这样拆分非常有必要，能够保证拆出的数相加在 $[1,m_i]$ 范围内，不会大于 m_i。例如，$m_i=25$，把它拆成 $1+2+4+8+10$，最后是余数 10，$10<16=2^4$，读者可以验证用这 5 个数能组合成 1～25 的所有数字，不会超过 25。

```
1   //洛谷 P1776: 二进制拆分 + 滚动数组
2   #include <bits/stdc++.h>
3   using namespace std;
4   const int N = 100010;
5   int n,C,dp[N];
6   int w[N],c[N],m[N];
7   int new_n;                                  //二进制拆分后的新物品总数量
8   int new_w[N],new_c[N],new_m[N];             //二进制拆分后的新物品
9   int main(){
10      cin >> n >> C;
11      for(int i = 1;i <= n;i++)  cin >> w[i]>> c[i]>> m[i];
12  //以下是二进制拆分
13      int new_n = 0;
14      for(int i = 1;i <= n;i++){
15          for(int j = 1;j <= m[i];j <<= 1) {   //二进制枚举: 1,2,4,...
16              m[i] -= j;                       //减去已拆分的
17              new_c[++new_n] = j * c[i];       //新物品
18              new_w[new_n]   = j * w[i];
19          }
20          if(m[i]){                            //最后一个是余数
21              new_c[++new_n] = m[i] * c[i];
22              new_w[new_n]   = m[i] * w[i];
23          }
24      }
25  //以下是滚动数组版本的 0/1 背包
26      for(int i = 1;i <= new_n;i++)            //枚举物品
27          for(int j = C;j >= new_c[i];j-- )    //枚举背包容量
28              dp[j] = max(dp[j],dp[j - new_c[i]] + new_w[i]);
29      cout << dp[C] << endl;
30      return 0;
31  }
```

这种解法可以看作多重背包问题的标准解法，不过，还有更优的解法——单调队列优化。

3）单调队列优化

这种方法的复杂度为 $O(nC)$，是最优的解法。详情见 5.8 节。

洛谷 P2347“砝码称重”是一道类似的多重背包问题，请读者练习用二进制拆分优化解决。

3. 最长公共子序列（Longest Common Subsequence，LCS）

一个给定序列的子序列，是在该序列中删去若干元素后得到的序列。例如，$X=\{A,B,C,B,D,A,B\}$，它的子序列有$\{A,B,C,B,A\}$、$\{A,B,D\}$、$\{B,C,D,B\}$等。子序列和子串是不同的概念，子串的元素在原序列中是连续的。

给定两个序列 X 和 Y，当另一序列 Z 既是 X 的子序列，又是 Y 的子序列时，称 Z 是序列 X 和 Y 的公共子序列。最长公共子序列是长度最长的公共子序列。

问题描述：给定两个序列 X 和 Y，找出 X 和 Y 的一个最长公共子序列。

用暴力法找最长公共子序列，需要先找出 X 的所有子序列，然后一一验证是否为 Y 的子序列。如果 X 有 m 个元素，那么 X 有 2^m 个子序列；Y 有 n 个元素；总复杂度大于 $O(n2^m)$。

用动态规划求 LCS，复杂度为 $O(nm)$。用 dp[i][j]表示序列 X_i（表示 $x_1,x_2,\cdots,x_i$ 这个序列，即 X 的前 i 个元素组成的序列；这里用小写的 x 表示元素，用大写的 X 表示序列）和 Y_j（表示 $y_1,y_2,\cdots,y_j$ 这个序列，即 Y 的前 j 个元素组成的序列）的最长公共子序列的长度。dp[n][m]就是答案。

分解为以下两种情况。

（1）当 $x_i=y_j$ 时，已求得 X_{i-1} 和 Y_{j-1} 的最长公共子序列，在其尾部加上 x_i 或 y_j，即可得到 X_i 和 Y_j 的最长公共子序列。状态转移方程为 dp[i][j]=dp[$i-1$][$j-1$]+1。

（2）当 $x_i\neq y_j$ 时，求解两个子问题：X_{i-1} 和 Y_j 的最长公共子序列、X_i 和 Y_{j-1} 的最长公共子序列。取其中的最大值，状态转移方程为 dp[i][j]=max{dp[i][$j-1$],dp[$i-1$][j]}。

习题：hdu 1159(Common subsequence)。

4. 最长递增子序列（Longest Increasing Subsequence，LIS）

问题描述：给定一个长度为 n 的数组，找出一个最长的单调递增子序列。

例如，一个长度为 7 的序列 $A=\{5,6,7,4,2,8,3\}$，它最长的单调递增子序列为{5,6,7,8}，长度为 4。

定义状态 dp[i]表示以第 i 个数为结尾的最长递增子序列的长度，那么有

$$\text{dp}[i]=\max\{\text{dp}[j]\}+1,\quad 0<j<i,A_j<A_i$$

最终答案是 max{dp[i]}。

复杂度分析：j 在 0～i 滑动，复杂度为 $O(n)$；i 的变化范围也为 $O(n)$；总复杂度为 $O(n^2)$。

提示 DP 并不是 LIS 问题的最优解法，有复杂度为 $O(n\log_2 n)$ 的非 DP 解法[①]。

习题：hdu 1257(最少拦截系统)。

5. 编辑距离(Edit Distance)

问题描述：给定两个单词 word1 和 word2，计算出将 word1 转换为 word2 所需的最小操作数。一个单词允许进行 3 种操作：①插入一个字符；②删除一个字符；③替换一个字符。

为方便理解，把长度为 m 的 word1 存储在数组 word1[1]～word1[m]，把长度为 n 的 word2 存储在数组 word2[1]～word2[n]，不使用 word1[0]和 word2[0]。

定义二维数组 dp，dp[i][j]表示从 word1 的前 i 个字符转换到 word2 的前 j 个字符所需要的操作步骤，dp[m][n]就是答案。图 5.10 所示为 word1="abcf"，word2="bcfe"的 dp 转移矩阵。

i=	0		a	b	c	f
		0	1	2	3	4
j=0	b	1	1	1	2	3
1	c	2	2	2	1	2
2	f	3	3	3	2	1
3	e	4	4	4	3	2
4						

图 5.10 编辑距离的 dp 转换矩阵

若 word1[i]=word2[j]，则 dp[i][j]=dp[$i-1$][$j-1$]，如图 5.10 中 dp[2][1]处的箭头所示。

其他情况，dp[i][j]=min{dp[$i-1$][$j-1$]，dp[$i-1$][j]，dp[i][$j-1$]}+1，如图 5.10 中 dp[4][2]处的箭头所示。dp[i][j]是它左侧、左上、上方的 3 个值中的最小值加 1，分别对应以下 3 种操作：

(1) dp[$i-1$][j]+1，删除，将 word1 的最后字符删除；

(2) dp[i][$j-1$]+1，插入，在 word2 的最后插入 word1 的最后字符；

(3) dp[$i-1$][$j-1$]+1，替换，将 word2 的最后一个字符替换为 word1 的最后一个字符。

算法复杂度为 $O(mn)$。

习题：洛谷 P2758。

6. 最小划分(Minimum Partition)

问题描述：给出一个正整数数组，把它分成 S_1 和 S_2 两部分，使 S_1 的数字和与 S_2 的数字和的差的绝对值最小。最小划分的特例是 S_1 和 S_2 的数字和相等，即差为 0。

例如，数组[1,6,11,5]，最小划分是 S_1=[1,5,6]，S_2=[11]；S_1 的数字和减去 S_2 的数字和，绝对值为|11−12|=1。

最小划分问题可以转化为 0/1 背包问题。求出数组的和 sum，把问题转换为：背包的容量为 sum/2，把数组中的每个数字看作物品的体积，求出背包最多可以放 res 体积的物品，返回结果|res−(sum−res)|。

① 参考《算法竞赛入门到进阶》7.1.4 节“最长递增子序列”的详细讲解。

习题：lintcode 724(Minimum partition)[①]。

7. 行走问题(Ways to Cover a Distance)

问题描述：给定一个整数 n，表示距离的步骤，一个人每次能走 1～3 步，问要走到 n，有多少种走法？例如，$n=3$，有 4 种走法：{1,1,1}、{1,2}、{2,1}、{3}。

这个问题和爬楼梯问题差不多。爬楼梯问题是每次能走 1 级或 2 级台阶，问走到第 n 级有多少种走法。爬楼梯问题的解实际上是一个斐波那契数列。

定义行走问题的状态 dp[i]为走到第 i 步的走法数量，那么有

$$\mathrm{dp}[0]=1,\mathrm{dp}[1]=1,\mathrm{dp}[2]=2$$

$$\mathrm{dp}[i]=\mathrm{dp}[i-1]+\mathrm{dp}[i-2]+\mathrm{dp}[i-3],\quad i>2$$

8. 矩阵最长递增路径(Longest Path in Matrix)

问题描述：给定一个矩阵，找一条最长路径，要求路径上的数字递增。矩阵的每个点可以向上、下、左、右 4 个方向移动，不能沿对角线方向移动。

例如，矩阵 $\begin{bmatrix}9 & 9 & 3\\ 7 & 5 & 7\\ 3 & 1 & 1\end{bmatrix}$ 的一个最长递增路径为 1-3-7-9，长度为 4。

下面给出两种解法。

(1) 暴力 DFS。设矩阵有 $m\times n$ 个点，以每个点为起点做 DFS，搜索递增路径，在所有递增路径中找出最长的路径。每个 DFS 都是指数级时间复杂度的，复杂度非常高。

(2) 记忆化搜索。在暴力 DFS 的基础上，用记忆化进行优化。把每个点用 DFS 得到的最长递增路径记下来，后面再搜索到这个点时，直接返回结果。由于每个点只计算一次，每条边也只计算一次，虽然做了 $m\times n$ 次 DFS，但是总复杂度仍然为 $O(V+E)=O(mn)$，其中 V 为点数，E 为边数。这也算是动态规划的方法。

习题：leetcode 329(矩阵中的最长递增路径)[②]。

9. 子集和问题(Subset Sum Problem)

问题描述：给定一个非负整数的集合 S，一个值 M，问 S 中是否有一个子集，其子集和等于 M。

例如，$S=\{6,2,9,8,3,7\}$，$M=5$，存在一个子集{2,3}，子集和等于 5。

用暴力法求解，即检查所有的子集。共有 2^n 个子集，为什么？用二进制辅助理解：若一个元素被选中，标记为 1；若没有选中，标记为 0；空集是 n 个 0，所有元素都被选中是 n 个 1，从 n 个 0 到 n 个 1，共有 2^n 个。

用 DP 求解，定义二维数组 dp。当 dp[i][j]=1 时，表示 S 的前 i 个元素存在一个子集和等于 j。问题的答案就是 dp[n][M]。

用 S[1]～S[n]记录集合 S 的 n 个元素。

---

① https://www.lintcode.com/problem/minimum-partition/description

② https://leetcode-cn.com/problems/longest-increasing-path-in-a-matrix/

分析状态转移方程，分为两种情况。

(1) 若 $S[i]>j$，则 $S[i]$不能放在子集中，有 $dp[i][j]=dp[i-1][j]$。

(2) 若 $S[i]\leqslant j$，有两种选择：不把 $S[i]$放在子集中，则 $dp[i][j]=dp[i-1][j]$；把 $S[i]$放在子集中，则 $dp[i][j]=dp[i-1][j-S[i]]$。

这两种情况，只要其中一个为 1，那么 $dp[i][j]$就为 1。

读者可以用图 5.11 的例子进行验证。

如果已经确定问题有解，即 $dp[n][M]=1$，如何输出子集内的元素？按推导状态转移方程的思路，从 $dp[n][M]$开始，沿着 dp 矩阵倒推回去即可。

$i=0$		1	2	3	4	5	6
$S[]$->		6	2	9	8	3	7
$j=0$	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0
2	0	0	1	1	1	1	1
3	0	0	1	1	1	1	1
4	0	0	0	1	1	1	1
5	0	0	0	1	1	1	1

图 5.11 子集和问题的 dp 矩阵

10. 最优游戏策略(Optimal Strategy for a Game)

问题描述[①]：有 n 堆硬币排成一行，它们的价值分别是 $v_1,v_2,\cdots,v_n$，n 为偶数；两人交替拿硬币，每次只能在剩下的硬币中拿走第 1 堆或最后一堆。如果你是先手，你能拿到的最大价值是多少？

例如，$v=\{8,15,3,7\}$，先手这样拿可以获胜：先手拿 7；对手拿 8；先手再拿 15；对手再拿 3，结束。先手拿到的最大价值为 7+15=22。

本题不能用贪心法。例如，在样例中，如果先手第 1 次拿 8，那么对手接下来肯定拿 15，先手失败。

定义二维数组 dp，$dp[i][j]$表示从第 i 堆到第 j 堆硬币区间内，先手能拿到的最大值。

在硬币区间$[i,j]$，先手有以下两个选择。

(1) 拿 i。接着对手也有两个选择：拿 $i+1$，剩下$[i+2,j]$；或拿 j，剩下$[i+1,j-1]$。在这两个选择中，对手必然选择对先手不利的拿法。

(2) 拿 j。接着对手也有两个选择：拿 i，剩下$[i+1,j-1]$；拿 $j-1$，剩下$[i,j-2]$。

得到 dp 转移方程[②]如下。

$$dp[i][j]=\max\{V[i]+\min(dp[i+2][j],dp[i+1][j-1]),$$
$$V[j]+\min(dp[i+1][j-1],dp[i][j-2])\}$$
$$dp[i][j]=V[i],\quad j=i$$
$$dp[i][j]=\max(V[i],V[j]),\quad j=i+1$$

11. 矩阵链乘法(Matrix Chain Multiplication)

先了解背景知识。

(1) 矩阵乘法。如果矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 能相乘，那么 $\boldsymbol{A}$ 的列数等于 $\boldsymbol{B}$ 的行数。设 $\boldsymbol{A}$ 为 m 行 n 列(记为 $m\times n$)，$\boldsymbol{B}$ 为 $n\times u$，那么乘积 $\boldsymbol{AB}$ 的尺寸为 $m\times u$，矩阵乘法 $\boldsymbol{AB}$ 需要做 $m\times n\times u$ 次乘法运算。

(2) 矩阵乘法的结合律：$(\boldsymbol{AB})\boldsymbol{C}=\boldsymbol{A}(\boldsymbol{BC})$。括号体现了计算的先后顺序。

① 问题和样例来自 https://www.geeksforgeeks.org/optimal-strategy-for-a-game-dp-31/。

② 还有一种 DP 方案，参考 https://www.geeksforgeeks.org/optimal-strategy-for-a-game-set-2/。

(3) 括号位置不同，矩阵乘法需要的乘法操作次数不同。以矩阵 $\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$ 的乘法为例，设 $\boldsymbol{A}$ 的尺寸为 $m \times n$，$\boldsymbol{B}$ 的尺寸为 $n \times u$，$\boldsymbol{C}$ 的尺寸为 $u \times v$，以下两种计算方法，需要的乘法次数分别为

$$(\boldsymbol{AB})\boldsymbol{C}, \text{乘法次数是 } m \times n \times u + m \times u \times v$$

$$\boldsymbol{A}(\boldsymbol{BC}), \text{乘法次数是 } m \times n \times v + n \times u \times v$$

两者的差是 $|m \times n \times (u-v) + u \times v \times (m-n)|$，它可能是一个巨大的值。如果能知道哪个括号方案是最优的，就能够大大减少计算量。

下面给出**矩阵链乘法问题**的定义：给定一个数组 $p[]$，其中 $p[i-1] \times p[i]$ 表示矩阵 $\boldsymbol{A}_i$ 的尺寸，输出最少的乘法次数，并输出此时的括号方案。

例如，$p[]=\{40,20,30,10,30\}$，它表示 4 个矩阵，尺寸分别为 40×20，20×30，30×10，10×30。4 个矩阵相乘，当括号方案为 $(\boldsymbol{A}(\boldsymbol{BC}))\boldsymbol{D}$ 时，有最少乘法次数(26000)。

如果读者学过区间 DP，就会发现这是一个典型的区间 DP 问题。设链乘的矩阵为 $\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_j$，即区间 $[i,j]$，那么按结合率，可以把它分成两个子区间 $[i,k]$ 和 $[k+1,j]$，分别链乘，有

$$\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_j = (\boldsymbol{A}_i\cdots\boldsymbol{A}_k)(\boldsymbol{A}_{k+1}\cdots\boldsymbol{A}_j)$$

必定有一个 k，使乘法次数最少，记这个 k 为 $k_{i,j}$。并且记 $\boldsymbol{A}_{i,j}$ 为此时 $\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_j$ 通过加括号后得到的一个最优方案，它被 $k_{i,j}$ 分开。

那么子链 $\boldsymbol{A}_i\boldsymbol{A}_{i+1}\cdots\boldsymbol{A}_k$ 的方案 $\boldsymbol{A}_{i,k}$、子链 $\boldsymbol{A}_{k+1}\boldsymbol{A}_{k+2}\cdots\boldsymbol{A}_j$ 的方案 $\boldsymbol{A}_{k+1,j}$ 也都是最优括号子方案。

这样就形成了递推关系，即

$$\boldsymbol{A}_{i,j} = \min\{\boldsymbol{A}_{i,k} + \boldsymbol{A}_{k+1,j} + p_{i-1}p_kp_j\}$$

用二维矩阵 dp[i][j] 表示 $\boldsymbol{A}_{i,j}$，得到转移方程为

$$\text{dp}[i][j] = \begin{cases} 0, & i = j \\ \min\{\text{dp}[i][k] + \text{dp}[k+1][j] + p_{i-1}p_kp_j\}, & i \leqslant k < j \end{cases}$$

dp[1][n] 就是答案，即最少乘法次数。

dp[i][j] 的编码实现，可以套用区间 DP 模板，遍历 i、j、k，复杂度为 $O(n^3)$。

区间 DP 常常可以用四边形不等式优化，但是本题不可以，因为它不符合四边形不等式优化所需要的单调性条件。

习题：poj 1651(Multiplication puzzle)。

12. 布尔括号问题(Boolean Parenthesization Problem)

问题描述[①]：布尔变量有两种取值：T(true)和 F(false)。定义 3 种布尔逻辑操作：&(与)、|(或)、^(异或)。现在输入：n 个取值符号，$n-1$ 个逻辑操作。加上括号可以改变执行顺序。要使结果是 T，共有多少种括号方案？

① https://www.geeksforgeeks.org/boolean-parenthesization-problem-dp-37/

输入样例：symbol[]={T,F,T},operator[]={^,&}

输出样例：2

提示：两种方案为((T ^ F) & T)和(T ^(F & T))。

13. 最短公共超序列(Shortest Common Supersequence)

问题描述[①]：给定两个字符串 str1 和 str2，找一个最短的字符串，使 str1 和 str2 是它的子序列。

例如，str1="AGGTAB"，str2="GXTXAYB"，输出"AGXGTXAYB"。

【习题】

(1) 洛谷 P1216/P1020/P1091/P1095/P1541/P1868/P2679/P2501/P3336/P3558/P4158/P5301。

(2) poj 1015/3176/1163/1080/1159/1837/1276/1014。

扫一扫

视频讲解

5.3 数位统计 DP

数位统计 DP 用于数字的数位统计，是一种比较简单的 DP 套路题。

一个数字的数位有个位、十位、百位，等等，如果题目和数位统计有关，那么可以用 DP 思想，把低位的统计结果记录下来，在高位计算时直接使用低位的结果，从而提高效率。

用下面一道例题详解数位统计 DP 的建模和两种实现：递推、记忆化搜索。

数位统计有关的题目，基本内容是处理“前导 0”和“数位限制”。

例 5.4 数字计数(洛谷 P2602，poj 2282)

问题描述：给定两个正整数 a 和 b，求在[a,b]的所有整数中，每个数码(digit)各出现了多少次。

输入：输入两个整数 a 和 b。

输出：输出 10 个整数，分别表示 0～9 在[a,b]中出现了多少次。

数据范围：$1 \leqslant a \leqslant b \leqslant 10^{12}$。

首先转换一下题目的要求。区间[a,b]等于[1,$a-1$]与[1,b]的差分，把问题转换为在区间[1,x]内，统计数字 0～9 各出现了多少次。

由于 a 和 b 的值太大，用暴力法直接统计[a,b]内的每个整数显然会超时，需要设计一个复杂度约为 $O(\log_2 n)$的算法。如何加快统计？容易想到 DP——把对低位数的统计结果用于高位数的统计。例如，统计出三位数之后，继续统计一个四位数时，对这个四位数的后 3 位的统计直接引用前面的结果。以统计[0,324]内数位 2 出现了多少次为例，搜索过程如

① https://www.geeksforgeeks.org/shortest-common-supersequence/

图 5.12 所示。其中，下画线的数字区间是前面已经计算过的，记录在状态 dp[]中，不用再重算；符号 * 表示区间中有数位 2，需要特殊处理。

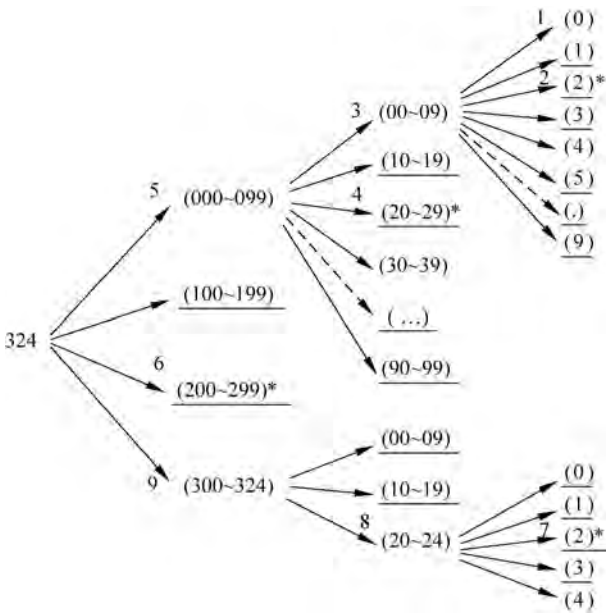

图 5.12　统计[0,324]内数位 2 出现的次数

把[0,324]区间分解为 4 个区间：000～099、100～199、200～299、300～324。其中，000～099、100～199、200～299 能够沿用 00～99 的计算结果。至于 300～324，最高位 3 不是要找的数位 2，等价于计算 00～24。

称数字前面的 0 为“前导 0”，如 000～099 中的 0。称每位的数字为“数位限制”，如 324 中最高位的 3、次高位的 2、最低位的 4。计数统计时需要特判前导 0 和数位限制，后面有详细解释。

下面分别用递推和记忆化搜索两种编码方法实现。

5.3.1　数位统计 DP 的递推实现

定义状态 dp[]，dp[i]为 i 位数的每种数字有多少个，说明如下。

(1) 一位数 0～9，每种数字有 dp[1]=1 个。

(2) 二位数 00～99，每种数字有 dp[2]=20 个。注意，这里是 00～99，不是 0～99。如果是 0～99，0 只出现了 11 次。这里把 0 和其他数字一样看待，但编程时需要特殊处理，因为按照习惯写法，数字前面的 0 应该去掉，如 043 应该写成 43。前导 0 在 0～9、00～99、000～999 等所有情况下都需要特殊处理。

(3) 三位数 000～999，每种数字有 dp[3]=300 个。

(4) 四位数 0000～9999，每种数字有 dp[4]=4000 个，依此类推。

dp[]有两种计算方法。

(1) $dp[i]=dp[i-1]\times 10+10^{i-1}$，这是从递推的角度分析得到的。以数字 2 为例，计算 dp[2]时，2 在个位上出现了 $dp[i-1]\times 10=dp[1]\times 10=10$ 次，即 2，12，22，…，92；在十

位上出现了 $10^{i-1}=10^{2-1}=10$ 次，即 20,21,22,…,29。计算 dp[3]时，2 在个位和十位上出现了 dp[2]×10=200 次，在百位上出现了 $10^{3-1}=100$ 次。

(2) dp[i]=$i\times10^i/10$，这是按排列组合的思路得到的。因为从 i 个 0 递增到 i 个 9，所有的字符共出现了 $i\times10^i$ 次，0～9 每个数字出现了 $i\times10^i/10$ 次。

下面考虑如何编程。以[0,324]为例，先从 324 的最高位开始，每种数字的出现次数 cnt 计算如下。

(1) 普通情况。例如，00～99 共出现了 3 次，分别出现在 000～099、100～199、200～299 的后两位上，每个数字在后两位上共出现 dp[$i-1$]×num[i]=dp[2]×3=60 次。对应代码第 23 行。

(2) 特判当前的最高位，即“数位限制”。第 3 位上的数字有 0、1、2、3，3 是最高位的数位限制。数字 0、1、2 分别在 000～099、100～199、200～299 的最高位上出现了 100 次，对应代码第 24 行；数字 3 在 300～324 中出现了 25 次，对应代码第 25～27 行。

(3) 特判前导 0。在前面的计算中，都把 0 和其他数字一样看待，但前导 0 应该去掉。对应代码第 28 行。

```
1  //改写自 https://www.luogu.com.cn/blog/mak2333/solution-p2602
2  #include<bits/stdc++.h>
3  using namespace std;
4  typedef long long ll;
5  const int N=15;
6  ll ten[N],dp[N];
7  ll cnta[N],cntb[N];                          //cnt[i],统计数字 i 出现了多少次
8  int num[N];
9  void init(){                                 //预计算 dp[]
10     ten[0] = 1;                              //ten[i]: 10 的 i 次方
11     for(int i=1;i<=N;i++){
12         dp[i]  = i*ten[i-1];                //或者写成 dp[i]  = dp[i-1]*10+ten[i-1];
13         ten[i] = 10*ten[i-1];
14     }
15 }
16 void solve(ll x, ll *cnt){
17     int len = 0;                             //数字 x 有多少位
18     while(x){                                //分解 x,num[i]为 x 的第 i 位数字
19         num[++len] = x%10;
20         x=x/10;
21     }
22     for(int i=len;i>=1;i--){                 //从高到低处理 x 的每位
23         for(int j=0;j<=9;j++)       cnt[j] += dp[i-1]*num[i];
24         for(int j=0;j<num[i];j++)  cnt[j] += ten[i-1]; //特判最高位比 num[i]小的数字
25         ll num2 = 0;
26         for(int j=i-1;j>=1;j--)     num2 = num2*10+num[j];
27         cnt[num[i]] += num2+1;                //特判最高位的数字 num[i]
28         cnt[0] -= ten[i-1];                   //特判前导 0
29     }
30 }
31 int main(){
```

```
    init();
    ll a,b;   cin >> a >> b;
    solve(a - 1, cnta), solve(b, cntb);
    for(int i = 0;i <= 9;i++)   cout << cntb[i] - cnta[i] <<" ";
}
```

分析代码的复杂度，solve()函数有两层 for 循环，只循环了 10×len 次。

5.3.2 数位统计 DP 的记忆化搜索实现

回顾记忆化搜索，其思路是在递归函数 dfs()中搜索所有可能的情况，遇到已经计算过的记录在 dp 中的结果，就直接使用，不再重复计算。

用递归函数 dfs()实现上述记忆化搜索，执行过程如下。如图 5.12 所示，以[0,324]为例，从输入 324 开始，一直递归到最深处的(0)，然后逐步回退，图中用箭头上的数字标识了回退的顺序。

记忆化搜索极大地减少了搜索次数。例如，统计 000～099 中 2 的个数，因为用 dp[]进行记忆化搜索，计算 5 次即可；如果去掉记忆化部分，需要检查每个数字，共 100 次。

下面设计 dp 状态。和前面递推的编码类似，记忆化搜索的代码中也需要处理前导 0 和每位的最高位。编码时，每次统计 0～9 中的一个数字，代码中用变量 now 表示这个数字。下面的解释都以 now＝2 为例。

定义状态 dp[][]，用来记录 0～9、00～99、000～999 这样的无数位限制情况下 2 的个数，以及 20～29、200～299、220～229、2200～2299 这种前面带有 2 的情况下 2 的个数。

dp[pos][sum]表示最后 pos 位范围是[$\underbrace{0\cdots0}_{pos个}$,$\underbrace{99\cdots9}_{pos个}$]，前面 2 的个数为 sum 时，数字 2 的总个数。例如，dp[1][0]＝1 表示 00～09，10～19，30～39，…区间内 2 的个数为 1；dp[1][1]＝11 表示 20～29 区间内 2 的个数为 11；dp[1][2]＝21 表示 220～229 区间内 2 的个数为 21；dp[1][3]＝31 表示 2220～2229 区间内 2 的个数为 31；dp[2][0]＝20 表示 000～099，100～199，300～399，400～499，…区间内 2 的个数为 20；dp[2][1]＝120 表示 200～299 区间内 2 的个数为 120；dp[2][2]＝220 表示 2200～2299 区间内 2 的个数为 220；等等。

用 lead 标识是否有前导 0，lead＝false 表示没有前导 0，lead＝true 表示有前导 0。

用 limit 标识当前最高位的情况，即“数位限制”的情况。如果是 0～9，limit＝false；否则 limit＝true。例如[0,324]，计算 324 的最高位时，范围是 0～3，此时 limit＝true。再如，从最高位数字 1 递归到下一位时，下一位的范围是 0～9，此时 limit＝false。如果不太理解，请对比前面递推实现的解释。

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 15;
ll dp[N][N];
int num[N],now;                                    //now: 当前统计 0～9 的哪一个数字
```

```
ll dfs(int pos,int sum,bool lead, bool limit){     //pos: 当前处理到第 pos 位
    ll ans = 0;
    if(pos == 0) return sum;                       //递归到 0 位数,结束,返回
    if(!lead && !limit && dp[pos][sum]!= -1)  return dp[pos][sum]; //记忆化搜索
    int up = (limit ? num[pos] : 9);     //这一位的最大值,如 324 的第 3 位是 up = 3
    for(int i = 0;i <= up;i++){                    //下面以 now = 2 为例
        if(i == 0 && lead)  ans += dfs(pos - 1, sum,  true, limit&&i == up);
                                                   //计算 000～099
        else if(i == now) ans += dfs(pos - 1, sum + 1,false,limit&&i == up);
                                                   //计算 200～299
        else if(i != now) ans += dfs(pos - 1, sum,  false,limit&&i == up);
                                                   //计算 100～199
    }
    if(!lead && !limit) dp[pos][sum] = ans;        //状态记录: 无前导 0
    return ans;
}
ll solve(ll x){
    int len = 0;                                   //数字 x 有多少位
    while(x){
        num[++len] = x % 10;
        x/ = 10;
    }
    memset(dp, - 1,sizeof(dp));
    return dfs(len,0,true,true);
}
int main(){
    ll a,b;     cin >> a >> b;
    for(int i = 0;i < 10;i++)   now = i, cout << solve(b) - solve(a - 1)<<" ";
    return 0;
}
```

代码中的 dp[pos][sum]，有队员习惯把 limit 和 lead 也加进去，定义为 dp[pos][sum][lead][limit]。

具体实现见下面的代码。

```
//用下面的代码替换上面的部分代码。只有 3 行不同
ll dp[N][N][2][2];                                 //这一行不同
ll dfs(int pos, int sum, bool lead, bool limit){
    ll ans = 0;
    if (pos == 0) return sum;
    if (dp[pos][sum][limit][lead] != -1) return dp[pos][sum][limit][lead]; //这一行不同
    int up = (limit ? num[pos] : 9);
    for(int i = 0;i <= up;i++){
        if(i == 0 && lead)  ans += dfs(pos - 1, sum,  true, limit&&i == up);
        else if(i == now) ans += dfs(pos - 1, sum + 1,false,limit&&i == up);
        else if(i != now) ans += dfs(pos - 1, sum,  false,limit&&i == up);
    }
    dp[pos][sum][limit][lead] = ans;               //这一行不同
    return ans;
}
```

记忆化搜索代码的复杂度和递推代码一样。

本题是数位统计 DP 的套路题，记忆化搜索的代码可以当作模板，下面的几道题套用了模板，关键是处理前导 0 和数位限制。

5.3.3 数位统计 DP 例题

下面用两道例题巩固对数位统计 DP 的理解。

1. 洛谷 P2657

例 5.5 Windy 数（洛谷 P2657）

问题描述：不含前导 0 且相邻两位数字之差至少为 2 的正整数称为 Windy 数。问在 a 和 b 之间（包括 a 和 b），总共有多少个 Windy 数？（特例：把一位数 0～9 看成 Windy 数）

输入：输入两个整数 a 和 b，$1 \leqslant a \leqslant b \leqslant 2 \times 10^9$。

输出：输出一个整数，表示答案。

输入样例：	输出样例：
25 50	20

在输入样例[25,50]区间中，32、33、34、43、44、45 不是 Windy 数，如数字 32，3－2＝1＜2。

求区间[a,b]内的 Windy 数，可以转换为分别求[1,a－1]和[1,b]区间。问题转换为给定一个数 x，求[0,x]内有多少个 Windy 数。

这是一道明显的数位统计 DP 题。检查一个很大的数时，对高位部分的检查可以沿用低位部分的检查结果。例如，计算 0～342 的 Windy 数，分为统计 000～099、100～199、200～299、300～342 内的 Windy 数。这与前面的洛谷 P2602 模板题的思路一样。

定义状态 dp[pos][last]，表示数字长度为 pos 位，前一位是 last 的情况下（包括前导 0）的无数位限制的 Windy 数总数。

例如，dp[1][0]＝8 表示 00～09 区间内 Windy 数有 8 个，数字长度 pos＝1，前一位 last＝1。注意此时前导 0 也是合法的，其中 00 和 01 不是 Windy 数，02～09 是 Windy 数，共 8 个。如果不算前导 0，0～9 内应该有 10 个 Windy 数。

dp[1][1]＝7 表示 10～19 区间内 Windy 数有 7 个；dp[1][2]＝7 表示 20～29 区间内 Windy 数有 7 个；dp[2][0]＝57 表示 000～099 区间内 Windy 数有 57 个，此时前导 0 也是合法的，如果不算前导 0，0～99 区间内应该有 74 个 Windy 数；dp[2][1]＝50 表示 100～199 区间内 Windy 数有 50 个；dp[2][2]＝51 表示 200～299 区间内 Windy 数有 51 个；dp[2][3]＝51 表示 300～399 区间内 Windy 数有 51 个；dp[3][1]＝362 表示 1000～1999 区间内 Windy 数有 362 个。

本题的代码与洛谷 P2602 模板题相似。其中，lead 和 limit 变量的含义也相同，分别标识前导 0 和数位限制。

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

```
int dp[15][15], num[15];
int dfs(int pos, int last, bool lead, bool limit){
    int ans = 0;
    if(pos == 0) return 1;
    if(!lead && !limit && dp[pos][last]!= -1) return dp[pos][last];
    int up = (limit?num[pos]:9);
    for(int i = 0;i <= up;i++){
        if(abs(i - last)< 2)  continue;   //不是 Windy 数
        if(lead && i == 0)    ans += dfs(pos - 1, -2,true, limit&&i == up);
        else                  ans += dfs(pos - 1,i,false, limit&&i == up);
    }
    if(!limit && !lead)   dp[pos][last] = ans;
    return ans;
}
int solve(int x){
    int len = 0;
    while(x) { num[++len] = x % 10; x/ = 10;}
    memset(dp, -1,sizeof(dp));
    return dfs(len, -2,true,true);
}
int main(){
    int a,b; cin >> a >> b;
    cout << solve(b) - solve(a - 1);
    return 0;
}
```

2. 洛谷 P4124

例 5.6　手机号码(洛谷 P4124)

问题描述：选手机号码，号码必须同时包含两个特征：手机号码中至少要出现 3 个相邻的相同数字；号码中不能同时出现 8 和 4。例如，满足条件的号码有 13000988721、23333333333、14444101000；而不满足条件的号码有 1015400080、10010012022。手机号码一定是 11 位数，不含前导 0。给出两个数 a 和 b，统计出 $[a,b]$ 区间内所有满足条件的号码数量。a 和 b 也是 11 位的手机号码。

输入：两个整数 a 和 b，$10^{10} \leqslant a \leqslant b \leqslant 10^{11}$。

输出：输出一个整数表示答案。

输入样例：	输出样例[①]：
12121284000 12121285550	5

本题也是数位统计 DP 的套路题。

本题可以回避前导 0。因为号码是 11 位的，最高位为 1～9，只要限定最高位不为 0 即可。

① 样例解释：满足条件的号码有 12121285000、12121285111、12121285222、12121285333、12121285550。

定义状态 dp[pos][u][v][state][n8][n4]，其中 pos 表示当前数字长度，u 表示前一位数字，v 表示再前一位数字，state 标识是否出现 3 个连续相同数字，n8 标识是否出现 8，n4 标识是否出现 4。

其他代码和前面模板题相似。

```
//改写自 www.luogu.com.cn/blog/yushuotong-std/solution--4124
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
ll dp[15][11][11][2][2][2];
int num[15];
ll dfs(int pos,int u,int v,bool state,bool n8,bool n4,bool limit){
    ll ans=0;
     if(n8 && n4) return 0;                    //8 和 4 不能同时出现
     if(!pos) return state;
     if(!limit && dp[pos][u][v][state][n8][n4]!=-1) return dp[pos][u][v][state][n8][n4];
     int up = (limit?num[pos]:9);
     for(int i=0;i<=up;i++)
         ans += dfs(pos-1,i,u,state||(i==u&&i==v),n8||(i==8),n4||(i==4),
         limit&&(i==up));
     if(!limit)  dp[pos][u][v][state][n8][n4]=ans;
     return ans;
}
ll solve(ll x){
     int len = 0;
     while(x){num[++len]=x%10; x/=10;}
     if(len!=11) return 0;
     memset(&dp,-1,sizeof(dp));
     ll ans = 0;
     for(int i=1;i<=num[len];i++)          //最高位 1～9，避开前导 0 问题
         ans+= dfs(len-1,i,0,0,i==8,i==4,i==num[len]);
     return ans;
}
int main(){
     ll a,b;   cin>>a>>b;
     cout<<solve(b)-solve(a-1);
}
```

【习题】

(1) 洛谷 P4798/P3281/P2518/P3286/P4999。

(2) poj 3252。

扫一扫

视频讲解

5.4 状态压缩 DP

状态压缩是 DP[①] 的一个小技巧，一般应用在集合问题中。当 DP 状态是集合时，把集

① “状态压缩 DP、区间 DP、树形 DP”这些名词可能是中国算法竞赛选手创造的，英文可以翻译为：DP on Subsets（状态压缩 DP）；DP over Intervals（区间 DP）；DP on Trees（树形 DP）。

合的组合或排列用一个二进制数表示，这个二进制数的0/1组合表示集合的一个子集，从而把对DP状态的处理转换为二进制的位操作，让代码变得简洁易写，同时提高算法效率。从二进制操作简化集合处理的角度看，状态压缩也是一种DP优化方法。

5.4.1 引子

1. Hamilton问题

状态压缩DP常常用Hamilton(旅行商)问题作为引子。

例5.7 最短Hamilton路径[①]

问题描述：给定一个有权无向图，包括n个点，标记为$0\sim n-1$，以及连接n个点的边，求从起点0到终点$n-1$的最短路径。要求必须经过所有点，而且只经过一次。$1\leqslant n\leqslant 20$。

输入：第1行输入整数n。接下来n行中，每行输入n个整数，其中第i行第j个整数表示点i到j的距离，记为$a[i,j]$。$0\leqslant a[i,j]\leqslant 10^7$。

对于任意x,y,z，数据保证$a[x,x]=0$，$a[x,y]=a[y,x]$且$a[x,y]+a[y,z]\geqslant a[x,z]$。

输出：输出一个整数，表示最短Hamilton路径的长度。

时间限制：3s。

Hamilton问题是NP问题，没有多项式复杂度的解法。

先尝试暴力解法，枚举n个点的全排列。共有$n!$个全排列，一个全排列就是一条路径，计算每个全排列的路径长度，需要做n次加法。在所有路径中找最短的路径，总复杂度为$O(n\times n!)$。

2. DP求解Hamilton问题

如果用状态压缩DP求解，能把复杂度降低到$O(n^2\times 2^n)$。当$n=20$时，$O(n^2\times 2^n)\approx 4$亿，比暴力法好很多。下面介绍状态压缩DP的做法。

首先定义dp状态。设S为图的一个子集，用dp[S][j]表示集合S内的最短Hamilton路径，即从起点0出发经过S中的所有点，到达终点j时的最短路径(集合S中包括j点)。然后根据DP的思路，让S从最小的子集逐步扩展到整个图，最后得到的dp[N][$n-1$]就是答案，N表示包含图上所有点的集合。

如何求dp[S][j]？可以从小问题$S-j$递推到大问题S。其中，$S-j$表示从集合S中去掉j，即不包含j点的集合。

如何从$S-j$递推到S？设k为$S-j$中的一个点，把$0\sim j$的路径分为两部分：0-1-…-k和k-($k+1$)-…-j。以k为变量枚举$S-j$中所有点，找出最短的路径，状态转移方

① https://www.acwing.com/problem/content/description/93/

程为

$$dp[S][j]=\min\{dp[S-j][k]+dist(k,j)\},\quad k\in S-j$$

集合 S 初始时只包含起点 0，然后逐步将图中的点包含进来，直到最后包含所有点。这个过程用状态转移方程实现。

用图 5.13 可解释上述原理。读者可以体会为什么用 DP 遍历路径比用暴力法遍历路径更有效率。其关键在于已经遍历过的点的顺序对以后的决策没有影响，这体现了 DP 的无后效性。

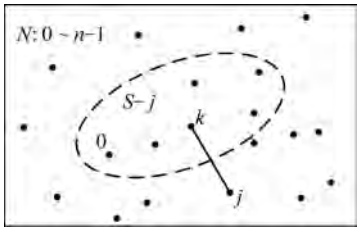

图 5.13　枚举集合 $S-j$ 中所有点

3. 用状态压缩 DP 编码

以上是 dp 状态设计，接下来是编程实现，最重要的是如何操作集合 S。这就是状态压缩 DP 的技巧：用一个二进制数表示集合 S，把 S“压缩”到一个二进制数中。这个二进制数的每位表示图上的一个点，等于 0 表示 S 不包含这个点，等于 1 表示包含。例如，S=0011 0101，其中有 4 个 1，表示集合中包含 5、4、2、0 共 4 个点。本题最多有 20 个点，那么就定义一个 20 位的二进制数表示集合 S。

下面给出代码，第 1 个 for 循环有 2^n 次，加上后面两个各 n 次的 for 循环，总复杂度为 $O(n^2\times 2^n)$。

第 1 个 for 循环实现了从最小的集合扩展到整个集合。最小的集合为 S=1，它的二进制数只有最后一位是 1，即包含起点 0；最大的集合是 $S=(1<<n)-1$，它的二进制数中有 n 个 1，包含了所有点。

算法最关键的部分是“枚举集合 $S-j$ 中所有点”，是通过代码中的两个 if 语句实现的：

if((S>>j) & 1)，判断当前的集合 S 中是否含有 j 点；

if((S^(1<<j))>>k & 1)，其中 S^(1<<j)的作用是从集合中去掉 j 点，得到集合 $S-j$，然后>> k & 1 表示用 k 遍历集合中的 1，这些 1 就是 $S-j$ 中的点，这样就实现了“枚举集合 $S-j$ 中所有的点”。注意，S^(1<<j)也可以写为 S-(1<<j)。

这两个语句可以写在一起：if(((S>>j) & 1) &&((S^(1<<j))>>k & 1))，不过分开写效率更高。

```
#include<bits/stdc++.h>
using namespace std;
int n, dp[1<<20][21];
int dist[21][21];
int main(){
    memset(dp,0x3f,sizeof(dp));          //初始化最大值
    cin>>n;
    for(int i=0; i<n; i++)               //输入图
        for(int j=0; j<n; j++)
            cin>>dist[i][j];             //输入点之间的距离
    dp[1][0]=0;                          //开始：集合中只有点 0,起点和终点都是 0
    for(int S=1; S<(1<<n); S++)          //从小集合扩展到大集合,集合用 S 的二进制表示
```

```
            for(int j = 0; j < n; j++)                    //枚举点 j
                if((S >> j) & 1)                          //(1)这个判断与下面的(2)同时起作用
                    for(int k = 0; k < n; k++)            //枚举到达 j 的点 k,k 属于集合 S - j
                        if((S^(1 << j)) >> k & 1)         //(2)k 属于集合 S - j,S - j 用(1)保证
                    //把(1)和(2)写在一起,更容易理解,但是效率低一点
                    //if( ((S >> j) & 1) && ((S^(1 << j)) >> k & 1) )
                            dp[S][j] = min(dp[S][j],dp[S^(1 << j)][k] + dist[k][j]);
        cout << dp[(1 << n) - 1][n - 1];                  //输出:路径包含了所有的点,终点是 n - 1
        return 0;
    }
```

上述 Hamilton 问题指定了起点和终点,类似的题目有洛谷 P1433。这一题略有不同,它指定了起点 0,但是没有指定终点,把上面的代码略加修改即可。在上面的代码中,最后求得 dp[(1 << n) − 1][j],表示起点为 0,终点为 j,经过所有点的最短路径。查询所有 dp[(1 << n) − 1][j]中的最小值,就是洛谷 P1433 的答案。

5.4.2 状态压缩 DP 的原理

从 5.4.1 节可知,状态压缩 DP 的应用背景是以集合为状态,且集合一般用二进制表示,用二进制的位运算处理,所以又可以称为"集合 DP[①]"。

集合问题一般是指数复杂度的(NP 问题),例如:①子集问题,设 n 个元素无先后关系,那么共有 2^n 个子集;②排列问题,对所有 n 个元素进行全排列,共有 $n!$ 个全排列。

可以这样概况状态压缩 DP 的思想:如果用二进制表示集合的状态(子集或排列),并用二进制的位运算遍历和操作,又简单又快。当然,由于集合问题是 NP 问题,所以状态压缩 DP 的复杂度仍然是指数级的,只能用于小规模问题的求解。

一个问题能用状态压缩 DP 求解,时间复杂度主要取决于 DP 算法,与是否使用状态压缩关系不大。状态压缩只是 DP 处理集合的工具,也可以用其他工具处理集合,只是不太方便,时间复杂度也差一点。

C 语言的位运算符有 &、|、^、<<、>>等,下面给出一些例子。虽然数字是用十进制表示的,但位运算是按二进制处理的。

```
#include<bits/stdc++.h>
int main(){
    int a = 213, b = 21;                    // a = 1101 0101, b = 0001 1001
    printf("a & b = %d\n",a & b);           // AND  =  17, 二进制 0001 0001
    printf("a | b = %d\n",a | b);           // OR   = 221, 二进制 1101 1101
    printf("a ^ b = %d\n",a ^ b);           // XOR  = 204, 二进制 1100 1100
    printf("a << 2 = %d\n",a << 2);         // a * 4 = 852, 二进制 0011 0101 0100
    printf("a >> 2 = %d\n",a >> 2);         // a/4  =  53, 二进制 0011 0101
    int i = 5;                              //a 的第 i 位是否为 1
```

① 所以,一般把状态压缩 DP 翻译为 DP on Subsets,即"子集上的 DP"。

```
10      if((1 << (i-1)) & a)  printf("a[%d] = %d\n",i,1);   //a 的第 i 位是 1
11      else                   printf("a[%d] = %d\n",i,0);   //a 的第 i 位是 0
12      a = 43, i = 5;                        //把 a 的第 i 位改成 1,a = 0010 1011
13      printf("a = %d\n",a | (1 <<(i-1))); //a = 59, 二进制 0011 1011
14      a = 242;                              //把 a 最后的 1 去掉,a = 1111 0010
15      printf("a = %d\n", a & (a-1));        //去掉最后的 1,a = 240, 二进制 1111 0000
16      return 0;
17  }
```

用位运算可以简便地**对集合进行操作**，表 5.1 给出了几个操作，在上面的代码中已有演示。

表 5.1　用位运算对集合进行操作

说　明	操　作
判断 a 的第 i 位(从最低位开始数)是否等于 1	$1<<(i-1))\ \&\ a$
把 a 的第 i 位改成 1	$a\ |(1<<(i-1))$
把 a 的第 i 位改成 0	$a\ \&(\sim(1<<i))$
把 a 的最后一个 1 去掉	$a\ \&(a-1)$

在具体题目中需要灵活使用位运算。后面的例题给出了位运算操作集合的实际应用的例子，帮助读者更好地掌握。

5.4.3　状态压缩 DP 例题

1. poj 2411

本题是状态压缩 DP 的经典例题，其特点是“轮廓线[①]”。

例 5.8　Mondriaan's dream(poj 2411)

问题描述：给定 n 行 m 列的矩形，用 1×2 的砖块填充，有多少种填充方案？例如，$n\times m=2\times4$ 时，有 5 种方案；$n\times m=2\times3$ 时，有 3 种方案，如图 5.14 所示。

(a) 2×4　　(b) 2×3

图 5.14　填充方案案例

输入：每行是一个测试用例，输入两个整数 n 和 m。若 $n=m=0$，表示终止。$1\leqslant n, m\leqslant11$。

输出：对每个测试用例，输出方案数。

摆放砖块的操作步骤，可以从第 1 行第 1 列开始，从左向右、从上向下依次摆放。横砖只占一行，不影响下一行的摆放；竖砖占两行，会影响下一行。同一行内，前列的摆放决定

① 在“插头 DP”中大量使用了轮廓线的概念。

后列的摆放。例如，第 1 列放横砖，那么第 2 列就是横砖的后半部分；第 1 列放竖砖，那么就不影响第 2 列。上、下两行是相关的，如果上一行是横砖，则不影响下一行；如果上一行是竖砖，那么下一行的同一列是竖砖的后半部分。

读者可以先对比暴力搜索的方法。使用 BFS，从第 1 行第 1 列开始扩展到全局，每个格子的砖块有横放、竖放 2 种摆法，共 $m \times n$ 个格子，复杂度大约为 $O(2^{m \times n})$。

下面用 DP 求解。DP 的思想是从小问题扩展到大问题，本题中，是否能从第 1 行开始，逐步扩展，直到最后一行？本题的复杂性在于一个砖块可能影响连续的两行，而不是一行，必须考虑连续两行的情况。

如图 5.15 所示，用一条虚线把矩形分为两部分，上半部分已经填充完毕，下半部分未完成。把这条划分矩形的虚线称为轮廓线。

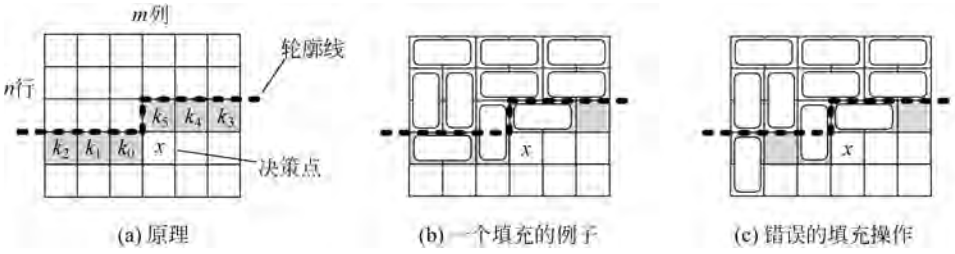

图 5.15　用轮廓线划分矩形

轮廓线下面的 6 个阴影方格 $k_5k_4k_3k_2k_1k_0$ 表示当前的砖块状态，它跨越了两行。从它们推广到下一个方格 x，即递推到新状态 $k_4k_3k_2k_1k_0x$。

$k_5k_4k_3k_2k_1k_0$ 有各种情况，用 0 表示没有填充砖块，用 1 表示填充了砖块，共有 000000～111111 共 2^6 种情况。图 5.15(b)是一个例子，其中 k_3 未填充，$k_5k_4k_3k_2k_1k_0=110111$。用二进制表示状态，这就是状态压缩的技术。

注意，根据 DP 递推的操作步骤，递推到阴影方格时，砖块只能填充到阴影格本身和上面的部分，不能填到下面。在图 5.15(c)中，把 k_2 填充到下面是错误的。

这 2^6 种情况中有些是非法的，应该去掉。在扩展到 x 时，分析 2^6 种情况和 x 的对应关系，根据 x 是否填充砖块，有 3 种情况。

(1) $x=0$(x 不放砖块)。如果 $k_5=0$(k_5 上没有砖块)，由于 k_5 只剩下和 x 一起填充的机会，现在失去了这一机会，所以这个情况是非法的；如果 $k_5=1$，则 $x=0$ 可以成立，递推到 $k_4k_3k_2k_1k_0x=k_4k_3k_2k_1k_00$。

(2) $x=1$(x 放竖砖)。x 只能和 k_5 一起放竖砖，要求 $k_5=0$，递推到 $k_4k_3k_2k_1k_0x=k_4k_3k_2k_1k_01$。

(3) $x=1$(x 放横砖)。x 只能和 k_0 一起放横转，要求 $k_0=0$，另外还应有 $k_5=1$，递推到 $k_4k_3k_2k_1k_0x=k_4k_3k_2k_111$。

经过上述讨论，对 $n \times m$ 的矩阵，可以得到状态定义和状态转移方程。

1) 状态定义

定义 DP 状态为 dp[i][j][k]，它表示递推到第 i 行第 j 列，且轮廓线处填充为 k 时的方案总数。

其中，k 是用 m 位二进制表示的连续 m 个方格，这 m 个方格中的最后一个方格就是

第 i 行第 j 列的方格。k 中的 0 表示方格不填充，1 表示填充。m 个方格前面的所有方格（轮廓线以上的部分）都已经填充为 1。dp[$n-1$][$m-1$][$(1<<m)-1$]就是答案，它表示递推到最后一行，最后一列，k 的二进制是 m 个 1（表示最后一行全填充）。时间复杂度为 $O(m\times n\times 2^m)$。

后面给出的代码用到了滚动数组，把二维[i][j]改为一维，状态定义变为 dp[2][k]。

2）状态转移

根据前面分析的 3 种情况，分别转移到新的状态。

（1）$x=0,k_5=1$。从 $k=k_5k_4k_3k_2k_1k_0=1k_4k_3k_2k_1k_0$ 转移到 $k=k_4k_3k_2k_1k_00$。转移代码为

dp[now][$(k<<1)$ &($\sim(1<<m)$)] +=dp[old][k];

其中，$\sim(1<<m)$表示原来的 $k_5=1$ 移到了第 $m+1$ 位，超出了 k 的范围，需要把它置 0。

（2）$x=1,k_5=0$。从 $k=k_5k_4k_3k_2k_1k_0=0k_4k_3k_2k_1k_0$ 转移到 $k=k_4k_3k_2k_1k_01$。转移代码为

dp[now][$(k<<1)$^1] +=dp[old][k];

（3）$x=1,k_0=0,k_5=1$。从 $k=k_5k_4k_3k_2k_1k_0=k_5k_4k_3k_2k_11$ 转移到 $k=k_4k_3k_2k_111$。转移代码为

dp[now][$((k<<1)\mid 3)$ &($\sim(1<<m)$)] +=dp[old][k];

其中，$(k<<1)\mid 3$ 表示末尾置 11；$\sim(1<<m)$表示原来的 $k_5=1$ 移到了第 $m+1$ 位，把它置 0。

下面是 poj 2411 的代码。

```
#include <iostream>
#include <cstring>
using namespace std;
long long dp[2][1<<11];
int now,old;                              //滚动数组,now 指向新的一行,old 指向旧的一行
int main(){
    int n,m;
    while( cin>>n>>m && n ){
        if(m>n)   swap(n,m);              //复杂度为 O(nm*2^m), m 较小有利
        memset(dp,0,sizeof(dp));
        now=0,old=1;                      //滚动数组
        dp[now][(1<<m)-1]=1;
        for(int i=0;i<n;i++)              //n 行
            for(int j=0;j<m;j++){         //m 列
                swap(now,old);            //滚动数组,now 始终指向最新的一行
                memset(dp[now],0,sizeof(dp[now]));
                for(int k=0;k<(1<<m);k++){                    //k 为轮廓线上的 m 格
                    if(k&1<<(m-1))                            //情况(1): 要求 k5=1
                      dp[now][(k<<1)&(~(1<<m))] += dp[old][k];
                              //原来的 k5=1 移到了第 m+1 位,置 0
                    if(i&&!(k&1<<(m-1)))                      //情况(2)
                              //i 不等于 0,即 i 不是第 1 行,另外要求 k5=0
                      dp[now][(k<<1)^1] += dp[old][k];
                    if(j&&(!(k&1))&&(k&1<<(m-1)))             //情况(3)
```

```
25                                  //j 不等于 0,即 j 不是第 1 列,另外要求 k0 = 0, k5 = 1
26                          dp[now][((k << 1) | 3) & (~(1 << m))] += dp[old][k];
27                                  //k 末尾置为 11,且原来的 k5 移到了第 m + 1 位,置 0
28                      }
29                  }
30              cout << dp[now][(1 << m) - 1]<< endl;
31          }
32          return 0;
33      }
```

2. hdu 4539

例 5.9　排兵布阵(hdu 4539)

问题描述：团长带兵来到 $n \times m$ 的平原作战。每个士兵可以攻击到并且只能攻击到与其曼哈顿距离为 2 的位置以及士兵本身所在的位置。当然，一个士兵不能站在另外一个士兵所能攻击到的位置；同时，因为地形，平原上也不是每个位置都可以安排士兵。现在，已知 n、m($n \leqslant 100$，$m \leqslant 10$)以及平原阵地的具体地形，请你帮助团长计算该阵地最多能安排多少士兵。

输入：包含多组测试数据。每组测试的第 1 行输入两个整数 n 和 m；接下来的 n 行中，每行输入 m 个数，表示 $n \times m$ 的矩形阵地，其中 1 表示该位置可以安排士兵，0 表示该地形不允许安排士兵。

输出：对每组测试，输出最多能安排的士兵数量。

输入样例：	输出样例：
6 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	2

图 5.16 的例子是一个合理的安排，图中的 1 表示一个站立的士兵，×表示曼哈顿距离为 2 的攻击点，不能安排其他士兵。

本题的思路比较容易。

首先考虑暴力法。对一种站立安排，如果任意两个士兵都没有站在曼哈顿距离为 2 的位置上，就是一个合法的安排。但是一共有 $2^{n \times m}$ 种站立安排，显然不能用暴力法判断。

下面考虑 DP 的思路。从第 1 行开始，一行一行地放士兵，在每行都判断合法性，直到最后一行。假设递推到了第 i 行，只需要看它和第 $i-1$、$i-2$ 行的情况即可。

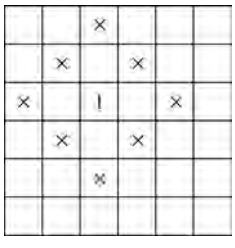

图 5.16　士兵和他的攻击点

(1) 判断第 i 行自身的合法性。这一行站立的士兵,不能站在曼哈顿距离为 2 的位置上。例如,$m=6$ 时,合法的士兵站立情况有 000010、000011、0000110、100011、110011 等。

(2) 判断第 i 行和第 $i-1$ 行的合法性。第 i 行任何一个士兵与第 $i-1$ 行士兵的曼哈顿距离不能为 2。

(3) 判断第 i 行和第 $i-2$ 行的合法性。

(4) 判断第 $i-1$ 行和第 $i-2$ 行的合法性。

定义 DP 状态 dp[i][j][k]表示递推到第 i 行时的最多士兵安排数量,此时第 i 行的士兵站立情况为 j,第 $i-1$ 行的士兵站立情况为 k。在 j 和 k 的二进制表示中,0 表示有士兵,1 表示无士兵。

从第 $i-1$ 行递推到第 i 行,状态转移方程为

$$dp[i][j][k]=\max(dp[i-1][k][p])+count\_line(i,sta[j])$$

其中,count_line(i,sta[j])计算第 i 行在合法的 j 状态下的士兵数量;用 p 遍历第 $i-2$ 行的合法情况。

下面给出代码。代码中有 4 个 for 循环,复杂度为 $O(nM^3)$。M 是预计算出的一行的合法情况数量,当 $m=10$ 时,$M=169$。用 init_line()函数预计算一行的合法情况。

```
1  //代码改写自 https://blog.csdn.net/jzmzy/article/details/20950205
2  #include<bits/stdc++.h>
3  using namespace std;
4  int mp[105][12];                    //地图
5  int dp[105][200][200];
6  int n,m;
7  int sta[200];                       //预计算一行的合法情况,m = 10 时只有 169 种合法情况
8  int init_line(int n){               //预计算出一行的合法情况
9      int M = 0;
10     for(int i = 0; i < n; i ++)
11         if((i&(i>>2)) == 0 && (i&(i<<2)) == 0)  //左右间隔 2 的位置没人,就是合法的
12            sta[M++] = i;
13     return M;                                   //返回合法情况有多少种
14 }
15 int count_line(int i, int x){                  //计算第 i 行的士兵数量
16     int sum = 0;
17     for(int j=m-1; j>=0; j--) {                 //x 是预计算过的合法安排
18         if(x&1) sum += mp[i][j];                //把 x 与地形匹配
19         x >>= 1;
20     }
21     return sum;
22 }
23 int main(){
24     while(~scanf("%d%d",&n,&m)) {
25         int M = init_line(1<<m);                //预计算一行的合法情况,有 M 种
26         for(int i = 0; i < n; i ++)
27             for(int j = 0; j < m; j ++)
28                 scanf("%d",&mp[i][j]);          //输入地图
29         int ans = 0;
30         memset(dp, 0, sizeof(dp));
31         for(int i = 0; i < n; i ++)             //第 i 行
32             for(int j = 0; j < M; j ++)         //枚举第 i 行的合法安排
```

```
33                  for(int k = 0; k < M; k ++) {      //枚举第 i-1 行的合法安排
34                      if(i == 0) {                   //计算第 1 行
35                          dp[i][j][k] = count_line(i, sta[j]);
36                          ans = max(ans, dp[i][j][k]);
37                          continue;
38                      }
39                      if((sta[j]&(sta[k]>>1)) || (sta[j]&(sta[k]<<1)))
40                          continue;                          //第 i 行和第 i-1 行冲突
41                      int tmp = 0;
42                      for(int p = 0; p < M; p ++){           //枚举第 i-2 行合法状态
43                          if((sta[p]&(sta[k]>>1)) || (sta[p]&(sta[k]<<1))) continue;
44                                                             //第 i-1 行和第 i-2 行冲突
45                          if(sta[j]&sta[p]) continue;        //第 i 行和第 i-2 行冲突
46                          tmp = max(tmp, dp[i-1][k][p]); //从第 i-1 行递推到第 i 行
47                      }
48                      dp[i][j][k] = tmp + count_line(i, sta[j]);
49                                                             //加上第 i 行的士兵数量
50                      ans = max(ans, dp[i][j][k]);
51                  }
52          printf("%d\n",ans);
53      }
54      return 0;
55  }
```

5.4.4 三进制状态压缩 DP

前面的状态压缩 DP 都利用了二进制运算，除了用二进制做状态压缩，也可以用其他进制，如三进制。用下面的例题说明三进制状态压缩 DP。

例 5.10　hdu 3001

问题描述：Acmer 先生决定访问 n 座城市。他可以空降到任意城市，然后开始访问，要求访问到所有城市，任何一座城市访问的次数不少于一次，不多于两次。n 座城市间有 m 条道路，每条道路都有路费。求 Acmer 先生完成旅行需要花费的最小费用。

输入：第 1 行输入 n 和 m，$1 \leqslant n \leqslant 10$。后面 m 行中，输入 3 个整数 a、b、c，表示城市 a 和 b 之间的路费为 c。

输出：最少花费，如果不能完成旅行，则输出 -1。

本题中 $n \leqslant 10$，数据很小，但是由于每座城市可以走两遍，可能的路线就有 $(2n)!$ 条，所以不能用暴力法。

本题是旅行商问题的变形，编程方法和 Hamilton 路径问题非常相似。阅读下面的题解时，请与 5.4.1 节的解释对照。

在普通路径问题中，一座城市只有两种情况：访问和不访问，分别用 1 和 0 表示，可以用二进制进行状态压缩。但是本题有 3 种情况：不访问、访问一次、访问两次，所以用三进制进行状态压缩分别用 0、1、2 表示。

当 $n=10$ 时，路径有 3^{10} 条，每种路径用三进制表示。例如，第 14 条路径，十进制 14 的三进制为 112_3，它的意思是：第 3 座城市走一次，第 2 座城市走一次，第 1 座城市走两次。

用 tri[i][j]定义路径，它表示第 i 条路径上的城市 j 的状态。在上面的例子中，tri[14][3]=1，tri[14][2]=1，tri[14][1]=2。make_trb()函数完成初始化计算，它把十进制 14 转换为三进制 112_3，并赋值给 tri[i][j]。

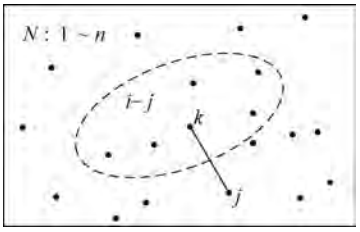

图 5.17　枚举路径 $i-j$ 中所有点

定义 DP 状态 dp[j][i]表示从城市 j 出发，按路径 i 访问 i 中所有城市的最小费用。

枚举路径 $i-j$ 中所有点，如图 5.17 所示。

图 5.17 中，$i-j$ 表示从路径 i 中去掉城市 j。从城市 j 开始访问路径 i，等于先走完路径 $i-j$，再走到城市 j。用 k 遍历 $i-j$ 中的所有城市，找到最少费用，状态转移方程为

$$dp[j][i]=\min(dp[j][i],dp[k][l]+graph[k][j])$$

其中，$l=i-bit[j]$，它涉及本题的**关键操作**：从路径 i 中去掉城市 j。这是三进制状态压缩的关键问题，如何从集合中去掉某个元素？

回顾 5.4.1 节的二进制状态压缩，是这样从集合 S 中去掉点 j 的：S^(1<<j)，它等价于 $S-(1<<j)$。类似地，在三进制中，从 i 中去掉 j 的代码写为 $i-bit[j]$，其中 bit[j]是三进制第 j 位的权值。

下面给出代码。comp_dp()函数中有 3 个 for 循环，第 1 个 for 循环 3^n 次，后两个分别循环 n 次，总复杂度为 $O(3^n n^2)$，当 $n=10$ 时，正好通过 OJ 测试。

```
#include<bits/stdc++.h>
const int INF = 0x3f3f3f3f;
using namespace std;
int n,m;
int bit[12]={0,1,3,9,27,81,243,729,2187,6561,19683,59049};
                //三进制每位的权值.与二进制的 0, 1, 2, 4, 8…对照理解
int tri[60000][11];
int dp[11][60000];
int graph[11][11];                          //存图
void make_trb(){                            //初始化,求所有可能的路径
     for(int i=0;i<59050;++i){              //共 3^10 = 59050 种路径状态
          int t=i;
          for(int j=1; j<=10; ++j){ tri[i][j]=t%3;  t/=3; }
     }
}
int comp_dp(){
         int ans = INF;
         memset(dp, INF, sizeof(dp));
         for(int j=0;j<=n;j++)
             dp[j][bit[j]]=0;               //初始化: 从第 j 座城市出发,只访问 j,费用为 0
         for(int i=0;i<bit[n+1];i++){ //遍历所有路径,每个 i 是一条路径
             int flag=1;                    //所有城市都遍历过一次以上
             for(int j=1;j<=n;j++){         //遍历城市,以 j 为起点
                 if(tri[i][j] == 0){        //是否有一座城市访问次数为 0
```

```
                    flag = 0;                    //还没有经过所有点
                    continue;
                }
                for(int k = 1; k <= n; k++){     //遍历路径 i - j 的所有城市
                    int l = i - bit[j];          //从路径 i 中去掉第 j 座城市
                    dp[j][i] = min(dp[j][i],dp[k][l] + graph[k][j]);

                }
            }
            if(flag)                             //找最小费用
               for(int j = 1; j <= n; j++)
                   ans = min(ans,dp[j][i]);      //路径 i 上最小的总费用
        }
        return ans;
}
int main(){
    make_trb();
    while(cin >> n >> m){
        memset(graph,INF,sizeof(graph));
        while(m -- ){
            int a,b,c;      cin >> a >> b >> c;
            if(c < graph[a][b])  graph[a][b] = graph[b][a] = c;
        }
        int ans = comp_dp();
        if(ans == INF) cout <<" - 1"<< endl;
        else           cout << ans << endl;
    }
    return 0;
}
```

【习题】

洛谷 P1433/P2831/P2704/P1879/P1896/P3092/P3694/P4925/P2157/P2167/P2396/P4363/P5005/P2150。

5.5 区 间 DP

扫一扫

视频讲解

区间 DP 是常见的 DP 应用场景。区间 DP 也是线性 DP，它把区间当成 DP 的阶段，用区间的两个端点描述状态和处理状态的转移。区间 DP 的主要思想是先在小区间进行 DP 得到最优解，然后再合并小区间的最优解求得大区间的最优解。解题时，先解决小区间问题，然后合并小区间，得到更大的区间，直到解决最后的大区间问题。合并的操作一般是把左右两个相邻的子区间合并。

区间 DP 的计算量比较大。一个长度为 n 的区间，编程时，区间 DP 至少需要两层 for 循环，第 1 层的 i 从区间的首部或尾部开始；第 2 层的 j 从 i 开始到结束，i 和 j 一起枚举出所有的子区间。所以，区间 DP 的复杂度大于 $O(n^2)$，一般为 $O(n^3)$。

很多区间 DP 问题可以用四边形不等式优化，把复杂度降为 $O(n^2)$。

5.5.1 石子合并问题和两种模板代码

区间 DP 的经典例子是石子合并问题，下面用这个例子解释区间 DP 的概念，并给出模板代码。

例 5.11 石子合并

问题描述：有 n 堆石子排成一排，每堆石子有一定的数量。将 n 堆石子合并成一堆，每次只能合并相邻的两堆石子，合并的花费为这两堆石子的总数。经过 $n-1$ 次合并后成为一堆，求最小总花费。

输入：第 1 行输入整数 n，表示有 n 堆石子。第 2 行输入 n 个数，分别表示这 n 堆石子的数目。

输出：最小总花费。

输入样例：	输出样例：
3 2 4 5	17

样例的计算过程是第 1 次合并 2+4=6；第 2 次合并 6+5=11；总花费为 6+11=17。

本题不能用贪心法求解，下面指出原因。先回顾贪心法，它的应用场景是能从局部最优扩展到全局最优。

(1) 如果石子堆没有顺序，可以任意合并，用贪心法每次选择最小的两堆合并。

(2) 本题要求只能合并相邻的两堆，不能用贪心法。贪心操作是每次合并时找石子数相加最少的两堆相邻石子。例如，环形石子堆，初始是{2,4,7,5,4,3}，用贪心法得到最小花费 64，但是另一种方法的结果为 63，如表 5.2 所示。

表 5.2 贪心法对比

步骤	贪心法(方法 1)		方法 2	
	花费	剩余石子堆	花费	剩余石子堆
初始	—	2,4,7,5,4,3	—	2,4,7,5,4,3
1	5	5,4,7,5,4	6	6,7,5,4,3
2	9	9,7,5,4	13	13,5,4,3
3	9	9,7,9	7	13,5,7
4	16	16,9	12	13,12
5	25	25	25	25
总花费	64		63	

下面用 DP 求解。

定义 dp[i][j]为合并第 i 堆到第 j 堆的最小花费。

状态转移方程为

$$\mathrm{dp}[i][j]=\min\{\mathrm{dp}[i][k]+\mathrm{dp}[k+1][j]+w[i][j]\},\quad i\leqslant k<j$$

dp[1][n]就是答案。方程中的 $w[i][j]$表示从第 i 堆到第 j 堆的石子总数。

按自顶向下的思路分析状态转移方程，很容易理解。计算大区间[i,j]的最优值时，合并它的两个子区间[i,k]和[$k+1$,j]，对所有可能的合并($i\leqslant k<j$，即 k 在 i 和 j 之间滑动)，采用最优的合并。子区间再分解为更小的区间，最小的区间[i,$i+1$]只包含两堆石子。

编程用自底向上递推的方法，先在小区间进行 DP 得到最优解，然后再逐步合并小区间为大区间。下面给出求 dp[i][j]的代码，其中包含 i、j、k 的 3 层循环，时间复杂度为 $O(n^3)$。第 1 个循环 j 是区间终点，第 2 个循环 i 是区间起点，第 3 个循环 k 在区间内滑动。注意，起点 i 应该从 $j-1$ 开始递减，也就是从最小的区间[$j-1$,j]开始，逐步扩大区间。i 不能从 1 开始递增，那样就是从大区间到小区间了。

```
1  for(int i = 1; i <= n; i++)   dp[i][i] = 0;//n 为石子堆数.初始化
2  for(int j = 2; j <= n; j++)                 // 区间[i,j]的终点 j,i<j<=n
3      for(int i = j - 1;i >= 1;i -- ) {       // 区间[i,j]的起点 i
4          dp[i][j] = INF;                     //初始化为极大值
5          for(int k = i;k < j;k++)       //大区间[i,j]从小区间[i,k]和[k+1,j]转移而来
6              dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + w[i][j]);
7      }
```

下面给出另一种写法，它的 i、j、k 都是递增的，更容易理解，推荐使用这种写法。代码中用了一个辅助变量 len，它等于当前处理的区间[i,j]的长度。dp[i][j]是大区间，它需要从小区间 dp[i][k]和 dp[$k+1$][j]转移而来，所以应该先计算出小区间，才能根据小区间计算出大区间。len 就起到了这个作用，从最小的区间 len=2 开始，此时区间[i,j]为[i,$i+1$]；最后是最大区间 len=n，此时区间[i,j]为[1,n]。

```
1  for(int i = 1; i <= n; i++)    dp[i][i] = 0;
2  for(int len = 2; len <= n; len++)         //len 为区间[i,j]的长度,从小区间扩展到大区间
3      for(int i = 1; i <= n - len + 1; i++){  // 区间起点 i
4          int j = i + len - 1;              // 区间终点 j,i<j<=n
5          dp[i][j] = INF;
6          for(int k = i; k < j; k++)
7              dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j] + w[i][j]);
8      }
```

上述代码的复杂度为 $O(n^3)$，不太好。区间 DP 常常可以用四边形不等式进行优化，把复杂度性能提升到 $O(n^2)$，详见 5.10 节。经典例题“石子合并”也在 5.10 节进行了更多讲解。当然，不是所有的区间 DP 都能做四边形不等式优化。

5.5.2 区间 DP 例题

下面给出几道经典区间 DP 例题。请读者在看题解之前，自己思考并写出代码，一定会大有收获。

1. hdu 2476

例 5.12　String painter(hdu 2476)

问题描述：给定两个长度相等的字符串 A、B，由小写字母组成。一次操作，允许把 A 中的一个连续子串(区间)都转换为某个字符(就像用刷子刷成一样的字符)。要把 A 转换为 B，问最少操作数是多少？

输入：第 1 行输入字符串 A，第 2 行输入字符串 B。两个字符串的长度不大于 100。

输出：一个表示答案的整数。

输入样例：	输出样例：
zzzzzfzzzzz abcdefedcba	6

提示　第 1 次把 zzzzzfzzzzz 转换为 aaaaaaaaaaa，第 2 次转换为 abbbbbbbbba，第 3 次转换为 abccccccccba…

这道经典例题能帮助读者深入理解区间 DP 是如何构造和编码的。

1) 从空白串转换到 B

先考虑一个简单的问题：从空白串转换到 B。为方便阅读代码，把字符串存储为 $B[1]\sim B[n]$，不用 $B[0]\sim B[n-1]$，编码时这样输入：scanf("%s%s",$A+1$,$B+1$)。

如何定义 DP 状态？可以定义 dp[i]表示在区间[1,i]内转换为 B 的最少操作次数。或者更进一步，定义 dp[i][j]表示在区间[i,j]内从空白串转换到 B 时的最少操作次数。重点是区间[i,j]两端的字符 $B[i]$ 和 $B[j]$，分析以下两种情况。

(1) $B[i]=B[j]$。第 1 次用 $B[i]$把区间[i,j]刷一遍，这个刷法肯定是最优的。如果分别去掉两个端点，得到两个区间[$i+1$,j]和[i,$j-1$]，这两个区间的最少操作次数相等，也等于原区间[i,j]的最少操作次数。例如，B="abbba"，先用"a"全部刷一遍，再刷一次"bbb"，共刷两次。如果去掉第 1 个"a"，剩下的"bbba"也是刷两次。

(2) $B[i]\neq B[j]$。因为两个端点不同，至少要各刷一次。用标准的区间操作，把区间分成[i,k]和[$k+1$,j]两部分，枚举最少操作次数。

2) 从 A 转换到 B

如何求 dp[1][j]？观察 A 和 B 相同位置的字符，分析以下两种情况。

(1) $A[j]=B[j]$。这个字符不用转换，有 dp[1][j]=dp[1][$j-1$]。

(2) $A[j]\neq B[j]$。仍然用标准的区间 DP，把区间分成[1,k]和[$k+1$,j]两部分，枚举最少操作次数。这里利用了从空白串转换到 B 的结果，当区间[$k+1$,j]内 A 和 B 的字符不同时，从 A 转换到 B 与从空白串转换到 B 是等价的。

下面给出 hdu 2476 的代码，其中，从空白串转换到 B 的代码完全套用了石子合并问题的编码。

```
#include <bits/stdc++.h>
using namespace std;
char A[105],B[105];
int dp[105][105];
const int INF = 0x3f3f3f3f;
int main() {
    while(~scanf("%s%s", A+1, B+1)){
          int n = strlen(A+1);                    //输入 A, B
          for(int i=1;i<=n;i++)   dp[i][i]=1;//初始化
//先从空白串转换到 B
          for(int len=2; len<=n; len++)
              for(int i=1; i<=n-len+1; i++){
                  int j = i + len-1;
                  dp[i][j] = INF;
              if(B[i] == B[j])                    //区间[i, j]两端的字符相同,B[i] = B[j]
                      dp[i][j] = dp[i+1][j];  //或者 dp[i][j] = dp[i][j-1])
               else                               //区间[i, j]两端的字符不同 B[i] ≠ B[j]
                   for(int k=i; k<j; k++)
                       dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j]);
            }
//下面从 A 转换到 B
          for(int j=1; j<=n; ++j){
              if(A[j] == B[j])  dp[1][j] = dp[1][j-1];     //字符相同,不用转换
              else
                 for(int k=1; k<j; ++k)
                 dp[1][j] = min(dp[1][j], dp[1][k] + dp[k+1][j]);
          }
          printf("%d\n",dp[1][n]);
    }
    return 0;
}
```

2. hdu 4283

例 5.13　You are the one(hdu 4283)

问题描述：n 个男孩去相亲，排成一队上场。大家都不想等，排队越靠后越愤怒。每人的耐心不同，用 D 表示火气，设男孩 i 的火气为 D_i，他排在第 k 个时，愤怒值为 $(k-1)D_i$。

主持人不想看到会场气氛紧张。他安排了一间黑屋，可以调整这排男孩上场的顺序，屋子很狭长，先进去的男孩最后出来(相当于一个**堆栈**)。例如，当男孩 A 排到时，如果他后面的男孩 B 火气更大，就把 A 送进黑屋，让 B 先上场。一般情况下，那些火气小的男孩要多等等，让火气大的占便宜。不过，零脾气的你也不一定吃亏，如果你原本排在倒数第 2 个，而最后一个男孩脾气最坏，主持人为了让这个坏家伙第 1 个上场，把其他人全赶进了黑屋，结果你就排在了黑屋的第 1 名，将第 2 个上场。

注意，每个男孩都要进出黑屋。

对所有男孩的愤怒值求和，求所有可能情况的最小总愤怒值。

输入：第 1 行输入一个整数 T，即测试用例的数量。对于每种情况，第 1 行输入 $n(0<n\leqslant100)$，后面 n 行中，输入整数 $D_1\sim D_n$ 表示男孩的火气($0\leqslant D_i\leqslant100$)。

输出：对每个用例，输出最小总愤怒值之和。

读者可以试试用栈来模拟，这几乎是不可能的。这是一道区间 DP 例题，巧妙地利用了栈的特性。

定义 dp[i][j]表示从第 i 个人到第 j 个人，即区间[i,j]的最小总愤怒值。

由于栈的存在，本题的区间[i,j]的分割点 k 比较特殊。分割时，总是用区间[i,j]的第 1 个元素 i 把区间分成两部分，让 i 第 k 个从黑屋出来上场，即第 k 个出栈。根据栈的特性，若第 1 个元素 i 第 k 个出栈，则第 $2\sim k-1$ 个元素肯定在第 1 个元素之前出栈，第 $k+1\sim j$ 个元素肯定在第 k 个之后出栈①。

例如，5 个人的排队序号是 1,2,3,4,5。如果第 1($i=1$)个人要第 3($k=3$)个出场，那么栈的操作是这样：1 号进栈→2 号进栈→3 号进栈→3 号出栈→2 号出栈→1 号出栈。2 号和 3 号在 1 号之前出栈，1 号第 3 个出栈，4 号和 5 号在 1 号后面出栈。

分割点 $k(1\leqslant k\leqslant j-i+1)$把区间划分成两段，即 dp[$i+1$][$i+k-1$]和 dp[$i+k$][$j$]。dp[$i$][$j$]的计算分为以下 3 部分。

(1) dp[$i+1$][$i+k-1$]。原来 i 后面的 $k-1$ 个人现在排到 i 前面了。

(2) $D[i](k-1)$。第 i 个人往后挪了 $k-1$ 个位置，愤怒值增加 D[i]($k-1$)。

(3) dp[$i+k$][j]$+k$(sum[j]$-$sum[$i+k-1$])。第 k 个位置后面的人，即区间[$i+k$,j]的人，由于都在前 k 个人之后，相当于从区间的第 1 个位置往后挪了 k 个位置，所以整体愤怒值要加上 k(sum[j]$-$sum[$i+k-1$])。其中，sum[j]$=\sum_{i=1}^{j}D_i$ 为 $1\sim j$ 所有人火气值的和，sum[j]$-$sum[$i+k-1$]是区间[$i+k$,j]内人的火气值的和。

代码完全套用石子合并的模板。其中，DP 方程给出了两种写法，对照看更清晰。

```
for(int len = 2; len <= n; len++)
    for(i = 1;i <= n - len + 1;i++) {
        j = len + i - 1;
        dp[i][j] = INF;
        for(int k = 1;k <= j - i + 1;k++)       //k: i往后挪了k位,这样写容易理解
            dp[i][j] = min(dp[i][j], dp[i + 1][i + k - 1] + D[i] * (k - 1)
                                + dp[i + k][j] + k * (sum[j] - sum[i + k - 1])); //DP方程
      //for(int k = i;k <= j;k++)              //或者这样写,k为整个队伍的绝对位置
      //    dp[i][j] = min(dp[i][j], dp[i + 1][k] + D[i] * (k - i)
      //                              + dp[k + 1][j] + (k - i + 1) * (sum[j] - sum[k]));
    }
```

5.5.3 二维区间 DP

前面例子中的区间[i,j]可以看作在一条直线上移动，即一维 DP。下面给出一道二维区间 DP 的例题，它的区间同时在两个方向移动。

① https://blog.csdn.net/weixin_41707869/article/details/99686868

例 5.14 Rectangle painting 1(CF1199F)[①]

问题描述：有一个 $n\times n$ 大小的方格图，某些方格初始为黑色，其余为白色。一次操作，可以选定一个 $h\times w$ 的矩形，把其中所有方格涂成白色，代价是 $\max(h,w)$。要求用最小的代价把所有方格涂成白色。

输入：第 1 行输入整数 n，表示方格的大小。后面有 n 行，每行输入长度为 n 的串，包含符号'.'和'#','.'表示白色，'#'表示黑色。第 x 行第 y 列的坐标是 (x,y)。$n\leqslant 50$。

输出：打印一个整数，表示把所有方格涂成白色的最小代价。

输入样例：	输出样例：
5 #...# #.#. ..... .#... #....	5

设矩形区域从左下角坐标 (x_1,y_1) 到右上角坐标 (x_2,y_2)。定义状态 dp[x_1][y_1][x_2][y_2]表示把这个区域涂成白色的最小代价。

这个区域可以分别按 x 轴或按 y 轴分割成两个矩形，遍历所有可能的分割，求最小代价。从 x 方向看，是一个区间 DP；从 y 方向看，也是一个区间 DP。

代码可以完全套用一维 DP 的模板，分别在两个方向上操作。

(1) x 方向，把区间分为[x_1,k]和[$k+1$,x_2]。状态转移方程为

dp[x_1][][x_2][] = min(dp[x_1][][x_2][],dp[x_1][][k][] + dp[$k+1$][][x_2][])

(2) y 方向，把区间分为[y_1,k]和[$k+1$,y_2]。状态转移方程为

dp[][y_1][][y_2] = min(dp[][y_1][][y_2],dp[][y_1][][k] + dp[][$k+1$][][y_2])

下面给出代码，有 5 层循环，时间复杂度为 $O(n^5)$。对比一维区间 DP，有 3 层循环，复杂度为 $O(n^3)$。

```
1  //代码改写自 https://www.cnblogs.com/zsben991126/p/11643832.html
2  #include<bits/stdc++.h>
3  using namespace std;
4  #define N 55
5  int dp[N][N][N][N];
6  char mp[N][N];                                    //方格图
7  int main(){
8      int n;  cin>>n;
9      for(int i=1;i<=n;i++)
10         for(int j=1;j<=n;j++)   cin>>mp[i][j];
```

① https://codeforces.com/contest/1199/problem/F，如果 CodeForces 很慢，用代理提交(https://vjudge.net/problem/CodeForces-1199F)。

```
    for(int i = 1;i <= n;i++)
        for(int j = 1;j <= n;j++)
            if(mp[i][j] == '.')  dp[i][j][i][j] = 0;      //白格不用涂
            else dp[i][j][i][j] = 1;                      //黑格(i,j)涂成白色,需一次
    for(int lenx = 1;lenx <= n;lenx++)       //len 从 1 开始,不是 2,因为有 x 和 y 两个方向
        for(int leny = 1;leny <= n;leny++)
            for(int x1 = 1;x1 <= n - lenx + 1;x1++)
                for(int y1 = 1;y1 <= n - leny + 1;y1++){
                    int x2 = x1 + lenx - 1;               //x1 为 x 轴起点,x2 为 x 轴终点
                    int y2 = y1 + leny - 1;               //y1 为 y 轴起点,y2 为 y 轴终点
                    if(x1 == x2 && y1 == y2) continue;    //lenx = 1 且 leny = 1 的情况
                    dp[x1][y1][x2][y2] = max(abs(x1 - x2),abs(y1 - y2)) + 1; //初始值
                    for(int k = x1;k < x2;k++) //枚举 x 方向,y 不变,区间[x1,k] + [k + 1,x2]
                        dp[x1][y1][x2][y2] = min(dp[x1][y1][x2][y2],
                                                 dp[x1][y1][k][y2] + dp[k + 1][y1][x2][y2]);
                    for(int k = y1;k < y2;k++) //枚举 y 方向,x 不变,区间[y1,k] + [k + 1,y2]
                        dp[x1][y1][x2][y2] = min(dp[x1][y1][x2][y2],
                                                 dp[x1][y1][x2][k] + dp[x1][k + 1][x2][y2]);
                }
    cout << dp[1][1][n][n];
}
```

【习题】

(1) 力扣：https://leetcode-cn.com/circle/article/NfHhXD/。

(2) 区间 DP(kuangbin 带你飞)：https://vjudge.net/contest/77874。

(3) 洛谷 P1880/P3146/P1063/P1005/P4170/P4302/P2466。

扫一扫

视频讲解

5.6 树 形 DP

在树这种数据结构上做 DP 很常见：给出一棵树，要求以最少代价(或最大收益)完成给定的操作。在树上做 DP 显得非常自然，因为树本身有“子结构”性质(树和子树)，具有递归性，符合 5.1.2 节提到的“记忆化递归”的思路，所以树形 DP 一般就这样编程。

基于树的解题步骤一般是先把树转换为有根树(如果是几棵互不连通的树，就加一个虚拟根，它连接所有孤立的树)，然后在树上做 DFS，递归到最底层的叶子节点，再一层层返回信息更新至根节点。显然，树上 DP 所操作的就是这一层层返回的信息。不同的题目需要灵活设计不同的 DP 状态和转移方程。

树需要用数据结构来存储。关于树的存储方式，请提前阅读本书 10.1 节“图的存储”。

5.6.1 树形 DP 的基本操作

先看一道简单的入门题，了解树的存储以及如何在树上设计 DP 和进行状态转移。请读者特别注意 DP 设计时的两种处理方法：二叉树、多叉树。

例 5.15　二叉苹果树(洛谷 P2015)

问题描述：有一棵苹果树，如果树枝有分叉，一定是分两叉。这棵树共有 n 个节点，编号为 1～n，树根编号为 1。用一根树枝两端连接的节点的编号描述一根树枝的位置，下面是一棵有 4 根树枝的树：

```
2   5
 \ /
  3   4
   \ /
    1
```

这棵树的枝条太多了，需要剪枝。但是一些树枝上长有苹果，最好别剪。给定需要保留的树枝数量，求出最多能留住多少个苹果。

输入：第 1 行输入两个数 n 和 q($1 \leqslant q \leqslant n, 1 < n \leqslant 100$)，$n$ 表示树的节点数，q 表示要保留的树枝数量。接下来 $n-1$ 行描述树枝的信息。每行输入 3 个整数，前两个数是它连接的节点的编号，第 3 个数是这根树枝上苹果的数量。每根树枝上的苹果不超过 30000 个。

输出：最多能留住的苹果的数量。

首先是树的存储，在计算之前需要先存储这棵树。树是图的一种特殊情况，树的存储和图的存储相似：一种方法是邻接表，用 vector 实现又简单又快；另一种方法是链式前向星，对空间要求很高时可以使用，编码也不算复杂。本题给出的代码用 vector 存储树。

本题的求解从根开始自顶向下，是典型的 DP 思路。

1. DP 状态和 DP 状态转移方程

定义状态 dp[u][j]表示以节点 u 为根的子树上留 j 条边时的最多苹果数量。dp[1][q]就是答案。

状态转移方程如何设计？下面给出两种思路：二叉树方法、多叉树(一般性)方法。

1) 二叉树

根据二叉树的特征，考虑 u 的左右子树，如果左儿子 lson 留 k 条边，右儿子 rson 就留 $j-k$ 条边，用 k 在[0,j]内遍历不同的分割。读者可以尝试用这种思路写出记忆化搜索的代码，主要步骤参考下面的代码。

```
int dfs(int u, int j){                                  //以 u 为根的子树,保留 j 个树枝时的最多苹果
    if(dp[u][j] >= 0)  return dp[u][j];                 //记忆化,如果已计算过,就返回
    for(int k = 0; k < j; k++)                          //用 k 遍历
        dp[u][j] = max(dp[u][j], dfs(u.lson,k) + dfs(u.rson, j - k)); //左右儿子合起来
    return dp[u][j];
}
```

2) 多叉树

由于局限于二叉树这种结构，下面用多叉树实现，这是**一般性**的方法。把状态转移方程写成如下形式。

```
for(int j = sum[u]; j >= 0; j -- )          //sum[u]为以 u 为根的子树上的总边数
    for(int k = 0; k <= j - 1; k++)         //用 k 遍历不同的分割
        dp[u][j] = max(dp[u][j], dp[u][j - k - 1] + dp[v][k] + w); //状态转移方程
```

其中，v 是 u 的一个子节点。dp[u][j]的计算分为以下两部分。

(1) dp[v][k]：在 v 上留 k 条边。

(2) dp[u][$j-k-1$]：除了 v 上的 k 条边，以及[u,v]边，那么以 u 为根的这棵树上还有 $j-k-1$ 条边，它们在 u 的其他子节点上。

下面给出代码。

```
//洛谷 P2015 的代码(邻接表存树)
#include<bits/stdc++.h>
using namespace std;
const int N = 200;
struct node{
    int v, w;                                   //v 是子节点,w 是边[u,v]的值
    node(int v = 0, int w = 0) : v(v), w(w) {}
};
vector<node> edge[N];
int dp[N][N], sum[N];                           //sum[i]记录以 i 为根的子树的总边数
int n, q;
void dfs(int u, int father){
    for(int i = 0; i < edge[u].size(); i++) {   //用 i 遍历 u 的所有子节点
        int v = edge[u][i].v, w =  edge[u][i].w;
        if(v == father) continue;               //不回头搜索父亲,避免循环
        dfs(v, u);                              //递归到最深的叶子节点,然后返回
        sum[u] += sum[v] + 1;                   //子树上的总边数
      //for(int j = sum[u];j >= 0;j -- )
      //    for(int k = 0;k <= j - 1;k++)       //两个 for 优化为以下代码,不优化也可以
        for(int j = min(q, sum[u]); j >= 0; j -- )
            for(int k = 0; k <= min(sum[v], j - 1); k++)
                dp[u][j] = max(dp[u][j], dp[u][j - k - 1] + dp[v][k] + w);
    }
}
int main(){
    scanf("%d%d", &n, &q);                      //n 个点,留 q 根树枝
    for(int i = 1; i < n; i++){
        int u, v, w;  scanf("%d%d%d", &u, &v, &w);
        edge[u].push_back(node(v,w));           //把边[u,v]存到 u 的邻接表中
        edge[v].push_back(node(u,w));           //无向边
    }
    dfs(1, 0);                                  //从根节点开始做记忆化搜索
    printf("%d\n", dp[1][q]);
    return 0;
}
```

2. 二叉树和多叉树的讨论

本题是二叉树应用，但是上面的代码是按多叉树处理的。代码中用 v 遍历了 u 的所有子树，并未限定是二叉树。状态方程计算 dp[u][j]时包含两部分：dp[u][$j-k-1$]和

dp[v][k]，其中 dp[v][k]是 u 的一棵子树 v，dp[u][$j-k-1$]是 u 的其他所有子树。

上面代码中最关键的是 dfs()函数中 j 的循环方向，它应该从 sum[u]开始递减，而不是从 0 开始递增。例如，计算 dp[u][5]，它用到了 dp[u][4]、dp[u][3]等，它们可能是有值的，原值等于以前用 u 的另一棵子树计算得到的结果，也就是排除当前的 v 这个子树时计算的结果。

(1) 让 j 递减循环是正确的。例如，先计算 $j=5$，dp[u][5]用到了 dp[u][4]、dp[u][3]等，它们都是正确的原值；下一步计算 $j=4$ 时，新的 dp[u][4]会覆盖原值，但是不会影响到对 dp[u][5]的计算。

(2) 让 j 递增循环是错误的。例如，先计算 $j=4$，得到新的 dp[u][4]；再计算 dp[u][5]，这时需要用到 dp[u][4]，而此时的 dp[u][4]已经不再是正确的原值了。

读者可以联想 5.1.4 节的“自我滚动”编程，它的循环也是从大到小递减的。两者的原理一样，即新值覆盖原值的问题。

k 的循环顺序则无所谓，它是[0，$j-1$]区间的分割，从 0 开始递增或从 $j-1$ 开始递减都可以。

两个 for 循环还可以做一些小优化，详情见代码。

复杂度分析：dfs()函数递归到每个节点，每个节点有两个 for 循环，总复杂度小于 $O(n^3)$。

3. 树形 DP 例题

例 5.16　没有上司的舞会(洛谷 P1352)

问题描述：某单位有 n 个职员，编号为 1～n。他们之间有从属关系，也就是说他们的关系就像一棵以老板为根的树，父节点就是子节点的直接上司。现在有一个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 r_i，但是如果某个职员的直接上司来参加舞会，那么这个职员就无论如何也不肯来参加舞会了。编程计算邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

输入：第 1 行输入一个整数 n。第 2～$n+1$ 行中，每行输入一个整数，第 $i+1$ 行的整数表示职员 i 的快乐指数 r_i。第 $n+2$～$2n$ 行中，每行输入两个整数 l 和 k，代表 k 是 l 的直接上司。

输出：输出一行一个整数代表最大的快乐指数。

这也是一道经典题。一个节点表示一个职员，如果一个节点代表的职员参加宴会，那么他的子节点就不能参加宴会；如果不参加宴会，他的子节点参不参加都行。根据 DP 的解题思路，定义状态：dp[i][0]表示不选择当前节点(不参加宴会)的最优解；dp[i][1]表示选择当前节点(参加宴会)的最优解。

状态转移方程有两种情况。

(1) 不选择当前节点，那么它的子节点可选可不选，取其中的最大值，即

$$dp[u][0]+=\max(dp[v][0],dp[v][1])$$

(2) 选择当前节点，那么它的子节点不能选，即

$$dp[u][1]+=dp[v][0]$$

代码包含 3 部分：①建树，用 STL vector 生成链表，建立关系树；②树的遍历，用 DFS 从根节点开始进行记忆化搜索；③DP。

复杂度分析：遍历每个节点，总复杂度为 $O(n)$。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 6005;
int val[N], dp[N][2], father[N];
vector<int> G[N];
void addedge(int from,int to){
    G[from].push_back(to);          //用邻接表建树
    father[to] = from;              //父子关系
}
void dfs(int u){
    dp[u][0] = 0;                   //赋初值：不参加宴会
    dp[u][1] = val[u];              //赋初值：参加宴会
  //for(int i=0;i<G[u].size();i++){ //遍历 u 的邻居 v,逐一处理这个父节点的每个子节点
  //    int v = G[u][i];
    for(int v : G[u]){              //这一行和上面两行的作用一样
        dfs(v);                     //DFS 子节点
        dp[u][1] += dp[v][0];       //父节点选择,子节点不选
        dp[u][0] += max(dp[v][0], dp[v][1]);     //父节点不选,子节点可选可不选
   }
}
int main(){
      int n; scanf("%d",&n);
      for(int i=1;i<=n;i++) scanf("%d",&val[i]);   //输入快乐指数
      for(int i=1;i<n;i++){
        int u,v;   scanf("%d%d",&u,&v);
        addedge(v,u);
      }
      int t = 1;
      while(father[t]) t = father[t];            //查找树的根节点
      dfs(t);                                    //从根节点开始,用 DFS 遍历整棵树
      printf("%d\n", max(dp[t][0], dp[t][1]));
    return 0;
}
```

5.6.2 背包与树形 DP

有一些树形 DP 问题可以抽象为背包问题，称为“树形依赖的背包问题”。例如，5.6.1 节的“二叉苹果树”问题，可以建模为“分组背包”（注意与普通分组背包的区别，这里的每个组可以选多个物品，而不是一个），具体如下。

（1）分组。根节点 u 的每个子树是一个分组。

（2）背包的容量。把以 u 为根的整棵树上的树枝数看作背包容量。

（3）物品。把每个树枝看作一个物品，体积为 1，树枝上的苹果数量看作物品的价值。

（4）背包目标。能放入背包的物品的总价值最大，就是留在树枝上的苹果数最多。

如果做个对比，会发现分组背包的代码和“二叉苹果树”的代码很像，下面给出两段代码

作为对比。

(1) 分组背包的代码。参考 5.2 节分组背包例题 hdu 1712。

```
for(int i = 1; i <= n; i++)                                    //遍历每个组
    for(int j = C; j >= 0; j -- )                              //背包总容量 C
        for(int k = 1; k <= m; k++)                            //用 k 遍历第 i 组的所有物品
            if(j >= c[i][k])                                   //第 k 个物品能装进容量 j 的背包
               dp[j] = max(dp[j], dp[j - c[i][k]] + w[i][k]);     //第 i 组第 k 个
```

(2) 树形 DP 代码。下面是洛谷 P2015 部分代码。

```
for(int i = 0; i < edge[u].size(); i++) {     //把 u 的每个子树看作一个组
    ...
    for(int j = sum[u]; j >= 0; j -- )        //把 u 的树枝总数看成背包容量
        for(int k = 0; k <= j - 1; k++)       //用 k 遍历每个子树的每个树枝
            dp[u][j] = max(dp[u][j], dp[u][j - k - 1] + dp[v][k] + w);
```

注意代码(1)和代码(2)的 j 循环都是从大到小,因为用到了“自我滚动”数组,在对应的章节中有详细解释。

树形背包问题的状态定义,一般用 dp[u][j]表示以点 u 为根的子树中选择 j 个点(或 j 条边)的最优情况。

下面给出一道经典题,请读者自己分析和编码。

例 5.17　有线电视网(洛谷 P1273,poj 1155)

问题描述:某收费有线电视网计划转播一场足球比赛。他们的转播网和用户终端构成一个树状结构,这棵树的根节点位于足球比赛的现场,叶子节点为各个用户终端,其他中转站为该树的内部节点。从转播站到转播站以及从转播站到所有用户终端的信号传输费用都是已知的,一场转播的总费用等于传输信号的费用总和。现在每个用户都准备了一笔费用用于观看这场精彩的足球比赛,有线电视网有权决定给哪些用户提供信号,不给哪些用户提供信号。写一个程序,找出一个方案,使有线电视网在不亏本的情况下使观看转播的用户尽可能多。

输入:第 1 行输入两个用空格隔开的整数 N 和 M,其中 $2 \leqslant N \leqslant 3000, 1 \leqslant M \leqslant N-1$,$N$ 为整个有线电视网的节点总数,M 为用户终端的数量。第 1 个转播站即树的根节点,编号为 1,其他转播站编号为 $2 \sim N-M$,用户终端编号为 $N-M+1 \sim N$。接下来的 $N-M$ 行中,每行的输入表示一个转播站的数据,第 $i+1$ 行表示第 i 个转播站的数据,其格式为

$$k\ A_1\ C_1\ A_2\ C_2 \cdots A_k\ C_k$$

其中,k 表示该转播站下接 k 个节点(转播站或用户),每个节点对应一对整数 A 与 C,A 表示节点编号,C 表示从当前转播站传输信号到节点 A 的费用。最后一行输入所有用户为观看比赛而准备支付的钱数。

输出:输出一个整数,表示符合条件的最大用户数。

本题与洛谷 P2015 类似。

定义 dp[u][j]表示以 u 为根的子树上有 j 个用户时的最小费用。计算结束后，使 dp[1][j]≤0 的最大 j 就是答案。

状态转移方程为 dp[u][j]=max(dp[u][j],dp[u][$j-k$]+dp[v][k]+w)，与洛谷 P2015 的状态转移方程几乎一样。

【习题】

本节用几道简单例题介绍了树形 DP 的基本操作。树形 DP 的题目往往较难，请读者多练习。

(1) 经典题：hdu 1520、hdu 2196(在《算法竞赛入门到进阶》中有详细讲解)。

(2) 树形背包：poj 1947/2486，hdu 1011/1561/4003。

(3) 洛谷 P4322/P1352/P1040/P1122/P1273/P2014/P2585/P3047/P3698/P5658/P2607/P3177/P4395/P4516。

扫一扫

视频讲解

5.7 一般优化

DP 是一种非常优秀的算法思想，它通过“重叠子问题、最优子结构、无后效性”实现了高效的算法。DP 的效率取决于 3 方面：①状态总数；②每个状态的决策数；③状态转移计算量。这 3 方面都可以进行优化。

(1) 状态总数的优化。相当于搜索的各种剪枝，去除无效状态；使用降维，设计 DP 状态时尽量用低维的 DP。

(2) 减少决策数量，即状态转移方程的优化，如四边形不等式优化、斜率优化。

(3) 状态转移计算量的优化，如用预处理减少递推时间；用 Hash 表、单调队列、线段树减少枚举时间等。

本节先介绍几种简单优化方法，后面几节再介绍几种高级优化方法。

1. 倍增优化

动态规划是多阶段递推，可以用倍增法将阶段性的线性增长加快为成倍增长。2.5 节的 ST 算法就是倍增优化的 DP。5.2 节中多重背包的二进制拆分优化也是典型的倍增优化。

2. 树状数组优化

如果题目与简单的区间操作有关，如区间查询、区间修改等，区间操作可以用树状数组或线段树来处理，把区间操作复杂度优化到 $O(\log_2 n)$，从而降低总复杂度。这种题目的基本思路是先定义 DP 状态和状态转移方程，然后思考如何对方程进行优化。

下面给出一道树状数组优化的例题。

例 5.18　方伯伯的玉米田(洛谷 P3287)

问题描述：有一个包含 n 个正整数的序列，做 0～k 次操作，每次任选一个区间，把区

间内的所有数加1。最后任意去掉一些数字。问最多能剩下多少数字，构成一个单调不下降序列？

输入：第一行输入两个整数 n 和 k，分别表示数字的数目和最多操作次数。第2行输入 n 个整数 $a_i(i=1,2,\cdots,n)$，表示序列中的 n 个数字。

输出：一个整数，表示最多剩下多少数字。

数据范围：$1<n<10000$，$1<k\leqslant 500$，$1\leqslant a_i\leqslant 5000$。

最长单调不下降序列是一个典型的DP问题，和5.2节中的最长递增子序列(LIS)相似。本题加上了区间操作，可以用高级数据结构提高区间操作的效率。

设序列存储于 $a[1]\sim a[n]$。区间操作可以进行简化：在区间 $[L,R]$ 内加1的操作，应该把右端点 R 设置成最后一个位置 n，也就是说，每次操作的区间是 $[L,n]$。因为：①如果 R 不是最后位置，对区间 $[L,R]$ 加1，会导致 R 后面的数相对变小，不利于形成更长的单调不下降序列；②如果 R 是最后位置，至少不会影响整个序列的单调不下降状态。

先回顾最长递增子序列(LIS)问题，定义状态 $dp[i]$ 表示以第 i 个数为结尾的最长递增子序列的长度，状态转移方程为 $dp[i]=\max\{dp[j]\}+1$，$0<j<i$，$a[j]<a[i]$，答案是 $\max\{dp[i]\}$。

本题结合了区间操作，定义 $dp[i][j]$ 表示做过 j 次区间操作，每次操作的起点都不超过 i，且以 i 为结尾的LIS的长度。状态转移方程为

$$dp[i][j]=\max\{dp[x][y]+1\}\quad x<i,y\leqslant j,a[x]+y\leqslant a[i]+j$$

这是一个二维区间问题，直接计算需要 i、j 二重循环。回顾4.2.3节二维树状数组例题对二维区间的处理，发现本题类似。树状数组可以把区间查询和更新复杂度优化到 $O(\log_2 n)$。

下面给出代码，是二维树状数组和LIS的结合。分析时间复杂度，在main()函数中有 n、k 两个循环，循环内用树状数组执行query()和update()，总复杂度约为 $O(nk\log_2 k\log_2 n)$。

```
1  //代码改写自 www.luogu.com.cn/blog/361308/solution-p3287
2  #include<bits/stdc++.h>
3  using namespace std;
4  #define lowbit(x)  ((x) & -(x))
5  int a[10005], dp[10005][505], t[505][5505], n, k;
6  void update(int x, int y, int d) {      //更新区间
7      for (int i=x; i<=k+1; i+=lowbit(i))
8          for (int j=y; j<=5500; j+=lowbit(j))
9              t[i][j] = max(t[i][j], d);
10 }
11 int query(int x, int y) {              //查区间最大值
12     int ans=0;
13     for (int i=x; i>0; i-=lowbit(i))
14         for (int j=y; j>0; j-=lowbit(j))
15             ans = max(ans, t[i][j]);
16     return ans;
17 }
18 int main() {
```

```
    scanf("%d%d", &n, &k);
    for (int i=1; i<=n; i++) scanf("%d", &a[i]);
    for (int i=1; i<=n; i++)
        for (int j=k; j>=0; j--){
            dp[i][j] = query(j+1, a[i]+j) + 1;
            update(j+1, a[i]+j, dp[i][j]);
        }
    printf("%d", query(k+1, 5500));
    return 0;
}
```

3. 线段树优化

用下面的例题说明线段树 DP 优化的做法。

例 5.19　基站选址(洛谷 P2605)

问题描述：有 n 个村庄位于一条直线上，第 i 个村庄距离第 1 个村庄的距离为 D_i。要在这 n 个村庄中建不超过 k 个基站，在第 i 个村庄建立基站的费用为 C_i。如果在距离第 i 个村庄不超过 S_i 的范围内建立了一个基站，那么村庄就被基站覆盖。如果第 i 个村庄没有被覆盖，需要补偿 W_i。请选择基站位置，使总费用最少。

输入：第 1 行输入两个整数 n 和 k。第 2 行输入 $n-1$ 个整数，分别表示 $D_2, D_3, \cdots, D_n$。第 3 行输入 n 个整数，分别表示 $C_1, C_2, \cdots, C_n$。第 4 行输入 n 个整数，分别表示 $S_1, S_2, \cdots, S_n$。第 5 行输入 n 个整数，分别表示 $W_1, W_2, \cdots, W_n$。

输出：最少总费用。

数据范围：$k \leqslant n$，$k \leqslant 100$，$n \leqslant 20000$，$D_i \leqslant 10^9$，$C_i \leqslant 10000$，$S_i \leqslant 10^9$，$W_i \leqslant 10000$。

定义状态 dp[i][j]表示前 i 个村庄内第 j 个基站建在 i 处的最小费用。

状态转移方程为

$$\mathrm{dp}[i][j]=\min\{\mathrm{dp}[k][j-1]+\mathrm{pay}[k][i]\},\quad j-1\leqslant k<i$$

其中，pay[k][i]表示区间[k,i]内没有被基站 i、k 覆盖的村庄的赔偿费用。

如果直接计算这个状态转移方程，需要做 i、j、k 三重循环，复杂度为 $O(n^3)$。如何优化？

(1) 滚动数组。发现状态转移方程中的 j 只与 $j-1$ 有关，那么可以用滚动数组优化 j，把复杂度降低为 $O(n^2)$。优化后的状态转移方程为

$$\mathrm{dp}[i]=\min\{\mathrm{dp}[k]+\mathrm{pay}[k][i]\},\quad 1\leqslant k<i$$

(2) 区间操作的优化。方程中的 pay[][]计算[k,i]区间内的赔偿费用，是一个区间求和问题，用线段树维护。

本题的线段树代码很长，请读者自己完成。

扫一扫

视频讲解

5.8 单调队列优化

单调队列是常见的 DP 优化技术，本节讲解基本的思路和方法。在 5.9 节中，单调队列也有关键的应用。

5.8.1 单调队列优化的原理

先回顾单调队列的概念，它具有以下特征。

(1) 单调队列的实现。用双端队列实现，队头和队尾都能插入和弹出。手写双端队列很简单。

(2) 单调队列的单调性。队列内的元素具有单调性，从小到大，或者从大到小。

(3) 单调队列的维护。每个新元素都能进入队列，它从队尾进入队列时，为维护队列的单调性，应该与队尾比较，把破坏单调性的队尾弹出。例如，一个从小到大的单调队列，如果要入队的新元素 a 比原队尾 v 小，那么把 v 弹走，然后 a 继续与新的队尾比较，直到 a 比队尾大为止，最后 a 进入队尾。

单调队列在 DP 优化中的基本应用，是对这样一类 DP 状态转移方程进行优化：

$$\mathrm{dp}[i]=\min\{\mathrm{dp}[j]+a[i]+b[j]\},\quad L(i)\leqslant j\leqslant R(i)$$

方程中的 min 也可以是 max。方程的特点是其中关于 i 的项 $a[i]$ 和关于 j 的项 $b[j]$ 是独立的。j 被限制在窗口 $[L(i),R(i)]$ 内，常见的如给定一个窗口值 k，$i-k\leqslant j\leqslant i$。这个 DP 状态转移方程的编程实现，如果简单地对 i 做外层循环，对 j 做内层循环，复杂度为 $O(n^2)$。如果用单调队列优化，复杂度可提高到 $O(n)$。

为什么单调队列能优化这个 DP 状态转移方程？

概括地说，单调队列优化算法能把内、外两层循环精简到一层循环。其本质原因是外层 i 变化时，不同的 i 所对应的内层 j 的窗口有重叠。如图 5.18 所示，$i=i_1$ 时，对应的 j_1 的滑动窗口(窗口内处理 DP 决策)范围是上方的阴影部分；$i=i_2$ 时，对应的 j_2 处理的滑动窗口范围是下方的阴影部分；两部分有重叠。当 i 从 i_1 增加到 i_2 时，这些重叠部分被重复计算，如果减少这些重复，就得到了优化。如果把所有重叠的部分都优化掉，那么所有 j 加起来只从头到尾遍历了一次，此时 j 的遍历实际上就是 i 的遍历了。

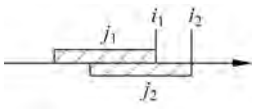

图 5.18 外层 i 和内层 j 的循环

在窗口内处理的这些决策，有以下两种情况。

(1) 被排除的不合格决策。内层循环 j 排除的不合格决策，在外层循环 i 增大时，需要重复排除。

(2) 未被排除的决策。内层循环 j 未排除的决策，在外层循环 i 增大时，仍然能按原来的顺序被用到。

那么可以用单调队列统一处理这些决策，从而精简到只用一个循环，得到优化。下面详细介绍单调队列的操作。

(1) 求一个 $\mathrm{dp}[i]$。i 是外层循环，j 是内层循环，在做内层循环时，可以把外层的 i 看

作一个定值。此时，$a[i]$可以看作常量，把 j 看作窗口$[L(i),R(i)]$内的变量，DP 状态转移方程等价于

$$dp[i]=\min\{dp[j]+b[j]\}+a[i]$$

问题转化为求窗口$[L(i),R(i)]$内的最优值 $\min\{dp[j]+b[j]\}$。记 $ds[j]=dp[j]+b[j]$，在窗口内，用单调队列处理 $ds[j]$，排除不合格的决策，最后求得区间内的最优值，最优值即队首。得到窗口内的最优值后，就可以求得 $dp[i]$。另外，队列中留下的决策，在 i 变化后仍然有用。

请注意，队列处理的决策 $ds[j]$只与 j 有关，与 i 无关，这是本优化方法的关键。如果既与 i 有关，又与 j 有关，它就不能在下一步求所有 $dp[i]$时得到应用。具体来说：①如果 $ds[j]$只与 j 有关，那么一个较小的 i_1 操作的某个策略 $ds[j]$与一个较大的 i_2 操作的某个策略 $ds[j]$是相等的，从而产生了重复性，可以优化；②如果 $ds[]$与 i、j 都有关，那么就没有重复性，无法优化。请结合后面的例题深入理解。

(2) 求所有 $dp[i]$。考虑外层循环 i 变化时的优化方法。一个较小的 i_1 所排除的 $ds[j]$，在处理一个较大的 i_2 时，也会被排除，重复排除其实没有必要；一个较小的 i_1 所得到的决策，仍能用于一个较大的 i_2。统一用一个单调队列处理所有 i，每个 $ds[j]$(提示：此时 j 不再局限于窗口$[L(i),R(i)]$，而是整个区间 $1\leqslant j\leqslant n$，那么 $ds[j]$实际上就是 $ds[i]$了)都进入队列一次，并且只进入队列一次，总复杂度为 $O(n)$。此时，内外层循环 i、j 精简为一个循环 i。

5.8.2 单调队列优化例题

1. 洛谷 P2627

例 5.20 Mowing the lawn(洛谷 P2627)

问题描述：有一个包括 n 个正整数的序列，第 i 个整数为 E_i，给定一个整数 k，找这样的子序列，子序列中的数在原序列连续不能超过 k 个。对子序列求和，问所有子序列中最大的和是多少？$1\leqslant n\leqslant 100000$，$0\leqslant E_i\leqslant 1000000000$，$1\leqslant k\leqslant n$。

例如，$n=5$，原序列为$\{7,2,3,4,5\}$，$k=2$，子序列$\{7,2,4,5\}$有最大和 18，其中的连续部分为$\{7,2\}$、$\{4,5\}$，长度都不超过 $k=2$。

由于 n 较大，算法的复杂度应该小于 $O(n^2)$，否则会超时。

用 DP 求解，定义 $dp[i]$为前 i 个整数的最大子序列和，状态转移方程为

$$dp[i]=\max\{dp[j-1]+sum[i]-sum[j]\},\quad i-k\leqslant j\leqslant i$$

其中，$sum[i]$为前缀和，即从 E_1 加到 E_i。

方程符合单调队列优化的标准方程：$dp[i]=\min\{dp[j]+b[j]\}+a[i]$。下面用这个例子详细讲解单调优化队列的操作过程。

把 i 看作定值，上述方程等价于

$$dp[i]=\max\{dp[j-1]-sum[j]\}+sum[i],\quad i-k\leqslant j\leqslant i$$

求 dp[i]，就是找到一个决策 j，$i-k\leqslant j\leqslant i$，使 dp[$j-1$]−sum[$j$]最大。

对这个方程编程求解，如果简单地做 i、j 循环，复杂度为 $O(nk)$，约为 $O(n^2)$。

如何优化？回顾单调队列优化的实质：外层 i 变化时，不同的 i 所对应的内层 j 的窗口有重叠。

内层 j 所处理的决策 dp[$j-1$]−sum[j]，在 i 变化时，确实发生了重叠。下面推理如何使用单调队列。

首先，对于一个固定的 i，用一个**递减**的单调队列求最大的 dp[$j-1$]−sum[j]。记 ds[j]=dp[$j-1$]−sum[j]，并记这个 i 对应的最大值为 dsmax[i]=max{ds[j]}。用单调队列求 dsmax[i]的步骤如下。

(1) 设从 $j=1$ 开始，首先让 ds[1]进入队列。此时窗口内的最大值 dsmax[i]=ds[1]。

(2) $j=2$，ds[2]进入队列，讨论两种情况。

若 ds[2]≥ds[1]，说明 ds[2]更优，弹走 ds[1]，ds[2]进入队列成为新队头，更新 dsmax[i]=ds[2]。这一步排除了不好的决策，留下更好的决策。

若 ds[2]<ds[1]，ds[2]进入队列。队头仍然是 ds[1]，保持 dsmax[i]=ds[1]。

这两种情况下 ds[2]都进入队列，是因为 ds[2]比 ds[1]更晚于离开窗口范围 k，即存活时间更长。

(3) 继续以上操作，让窗口内的每个 j($i-k\leqslant j\leqslant i$)都有机会进入队列，并保持队列是**从大到小的单调队列**。

经过以上步骤，求得了固定一个 i 时的最大值 dsmax[i]。

当 i 变化时，统一用一个单调队列处理，因为一个较小的 i_1 所排除的 ds[j]，在处理后面较大的 i_2 时，也会被排除，没有必要再重新排除一次；而且较小的 i_1 所得到的队列，后面较大的 i_2 也仍然有用。这样，每个 ds[j](1≤j≤n)都有机会进入队列一次，并且只进入队列一次，总复杂度为 $O(n)$。

如果对上述解释仍有疑问，请仔细分析洛谷 P2627 的代码。注意一个小技巧：虽然理论上在队列中处理的决策是 dp[$j-1$]−sum[j]，但是在编码时不用这么麻烦，队列只需要记录 j，然后在判断时用 dp[$j-1$]−sum[j]进行计算即可。

代码中去头和去尾的两个 while 语句是单调队列的常用写法，可以看作单调队列的特征。

```
//代码改写自 https://www.luogu.com.cn/blog/user21293/solution-p2627
#include<bits/stdc++.h>
using namespace std;
const int N=100005;
long long n,k,e[N],sum[N],dp[N];
long long ds[N];                                    //ds[j] = dp[j-1]-sum[j]
int q[N],head=0,tail=1;                             //递减的单调队列,队头最大
long long que_max(int j){
    ds[j] = dp[j-1]-sum[j];
    while(head<=tail && ds[q[tail]]<ds[j])  tail--;       //去掉不合格的队尾
    q[++tail]=j;                                          //j 进入队尾
```

```
    while(head<=tail && q[head]<j-k)   head++;              //去掉超过窗口 k 的队头
    return ds[q[head]];                                  //返回队头,即最大的 dp[j-1]-sum[j]
}
int main(){
    cin>>n>>k;   sum[0] = 0;
    for(int i=1;i<=n;i++){
        cin>>e[i];   sum[i] = sum[i-1] + e[i];           //计算前缀和
    }
    for(int i=1;i<=n;i++)   dp[i] = que_max(i) + sum[i]; //状态转移方程
    cout<<dp[n];
}
```

2. 多重背包

下面给出多重背包单调队列优化的解法,这是最优的方法。

多重背包问题:给定 n 种物品和一个背包,第 i 种物品的体积为 c_i,价值为 w_i,并且有 m_i 个,背包的总容量为 C。如何选择装入背包的物品,使装入背包的物品的总价值最大?

回顾用滚动数组实现的多重背包代码。

```
//洛谷 P1776,提交判题后返回 TLE
    for(int i=1;i<=n;i++)                                   //枚举每个物品
        for(int j=C;j>=c[i];j--)                            //枚举背包容量
            for(int k=1; k<=m[i] && k*c[i]<=j; k++)
                dp[j] = max(dp[j],dp[j-k*c[i]]+k*w[i]);
```

状态转移方程为 $dp[j]=\max\{dp[j-kc_i]+kw_i\},1\leqslant k\leqslant\min\{m_i,j/c_i\}$。

代码是 i、j、k 三重循环。其中,循环 i、j 互相独立,没有关系,不能优化;循环 j、k 是相关的,k 在 j 上有滑动窗口,所以目标是优化 j、k 这两层循环,此时可以把与 i 有关的部分看作定值。

对比单调队列的标准状态转移方程:$dp[i]=\max\{dp[j]+a[i]+b[j]\},L(i)\leqslant j\leqslant R(i)$,相差太大,似乎并不能应用单调队列。

回顾单调队列优化的实质,外层 i 变化时,不同的 i 所对应的内层 j 的窗口有重叠。状态转移方程 $dp[j]=\max\{dp[j-kc_i]+kw_i\}$ 的外层循环是 j,内层循环是 k,k 的滑动窗口是否重叠?下面观察 $j-kc_i$ 的变化情况。首先对比外层 j 和 $j+1$,让 k 从 1 递增,它们的 $j-kc_i$ 如下。

j:　$j-3c_i$　　$j-2c_i$　　$j-c_i$

$j+1$:　　$j+1-3c_i$　　$j+1-2c_i$　　$j+1-c_i$

可以看出,没有发生重叠。但是如果对比 j 和 $j+c_i$,发生了重叠。

j:　$j-3c_i$　　$j-2c_i$　　$j-c_i$

$j+c_i$:　$j+c_i-4c_i$　　$j+c_i-3c_i$　　$j+c_i-2c_i$

可以推理出，当 $j=j$，$j+c_i$，$j+2c_i$，…时有重叠；进一步推理出，当 j 除以 c_i 的余数相等时，这些 j 对应的内层 k 发生重叠。那么，如果把外层 j 的循环改为按 j 除以 c_i 的余数相等的值进行循环，就能利用单调队列优化了。

下面把原状态转移方程变换为可以应用单调队列的标准方程。原方程为

$$\mathrm{dp}[j]=\max\{\mathrm{dp}[j-kc_i]+kw_i\},\quad 1\leqslant k\leqslant \min\{m_i,j/c_i\}$$

令 $j=b+yc_i$，其中，$b=j\%c_i$，b 为 j 除以 c_i 得到的余数；$y=j/c_i$，y 为 j 整除 c_i 的结果。把 j 代入原方程，得①

$$\mathrm{dp}[b+yc_i]=\max\{\mathrm{dp}[b+(y-k)c_i]+kw_i\},\quad 1\leqslant k\leqslant \min\{m_i,y\}$$

令 $x=y-k$，代入得

$$\mathrm{dp}[b+yc_i]=\max\{\mathrm{dp}[b+xc_i]-xv_i+yw_i\},\quad y-\min(m_i,y)\leqslant x\leqslant y$$

与标准方程 $\mathrm{dp}[i]=\min\{\mathrm{dp}[j]+a[i]+b[j]\}$ 对比，几乎一样。

用单调队列处理决策 $\mathrm{dp}[b+xc_i]-xv_i$，下面给出代码，上述推理过程的原理详见代码中的注释。

```
//洛谷 P1776: 单调队列优化多重背包
#include<bits/stdc++.h>
using namespace std;
const int N=100010;
int n,C;
int dp[N],q[N],num[N];
int w,c,m;                                          //物品的价值 w,体积 c,数量 m
int main(){
    cin>>n>>C;                                      //物品数量 n,背包容量 C
    memset(dp,0,sizeof(dp));
    for(int i=1;i<=n;i++){
        cin>>w>>c>>m;
        if(m>C/c) m = C/c;                          //计算 min{m, j/c}
        for(int b=0;b<c;b++){                       //按余数 b 进行循环
            int head=1, tail=1;
            for(int y=0;y<=(C-b)/c;y++){            //y = j/c
                int tmp = dp[b+y*c]-y*w;            //用队列处理 tmp = dp[b + xc] - xw
                while(head<tail && q[tail-1]<=tmp)  tail--;
                q[tail] = tmp;
                num[tail++] = y;
                while(head<tail && y-num[head]>m)   head++;
                                                    //约束条件 y-min(mi,y)≤x≤y
                dp[b+y*c] = max(dp[b+y*c],q[head]+y*w); //计算新的 dp[]
            }
        }
    }
    cout<<dp[C]<<endl;
    return 0;
}
```

复杂度分析：外层 i 循环 n 次，内层的 b 和 y 循环总次数为 $c\times(C-b)/c\approx C$，且每次

① https://blog.csdn.net/qq_40679299/article/details/81978770

只进出队列一次，所以总复杂度为 $O(nC)$。

【习题】

（1）洛谷 P1725/P3957/P1776/P3089/P3572/P3522/P4544/P5665/P1973/P2569/P4852。

（2）poj 1821/2373/3017/3926。

（3）hdu 3401/3514/5945。

扫一扫

视频讲解

5.9 斜率优化/凸壳优化

有一类 DP 状态方程：

$$dp[i]=\min\{dp[j]-a[i]d[j]\},\quad 0\leqslant j<i,d[j]\leqslant d[j+1],a[i]\leqslant a[i+1]$$

它的特征是存在一个既有 i 又有 j 的项 $a[i]d[j]$，编程时，如果简单地对外层 i 和内层 j 循环，复杂度为 $O(n^2)$。

这里能用单调队列优化吗？单调队列所处理的策略，要求只能与内层 j 有关，与外层 i 无关，但是这个状态方程无法简单地得到只与 j 有关的部分。

用斜率（凸壳）模型，能够将方程转化，得到一个只与 j 有关的部分，即“斜率”，从而能够使用单调队列优化。这个算法称为斜率优化/凸壳优化（Convex Hull Trick），总时间复杂度为 $O(n)$，斜率优化的核心技术是斜率（凸壳）模型和单调队列。

斜率优化的数学建模在理解上有点困难，为方便读者理解，本节将详细论述算法的思想，然后用例题巩固理解，最后总结和扩展算法知识。

5.9.1 把状态转移方程变换为平面的斜率问题

观察 DP 状态转移方程，可以把 i 看作不变的常量，把关于 i 的部分（除了 dp[i]以外）看作常量，把 j 看作变量，目标是求 j 变化时 dp[i]的最优值。这里难以理解的是为什么要把 dp[i]看作变化的，请读完本节后再回头思考，dp[i]的变化对应了变化的直线截距。

去掉 min，状态转移方程转换为

$$dp[j]=a[i]d[j]+dp[i]$$

为方便观察，令 $y=dp[j],x=d[j],k=a[i],b=dp[i]$，状态转移方程变为

$$y=kx+b$$

斜率优化的数学模型，就是把状态转移方程转换为平面坐标系直线的形式 $y=kx+b$。

（1）变量 x、y 与 j 有关，并且只有 y 中包含 dp[j]。点(x,y)是题目中可能的决策。

（2）斜率 k、截距 b 与 i 有关，并且只有 b 中包含 dp[i]。最小的 b 包含最小的 dp[i]，也就是状态转移方程的解。

两点(x_1,y_1)、(x_2,y_2)连成的直线的斜率为$(y_2-y_1)/(x_2-x_1)$，它们只与 j 有关，这就是可以用单调队列处理的策略。

应用斜率优化的条件为 x 和 k 是单调增加的，即 x 随着 j 递增而递增，k 随着 i 递增而递增。

5.9.2 求一个 dp[i]

先考虑固定 i 的情况下求 dp[i]。由于 i 是定值，那么斜率 $k=a[i]$可以看作常数。当 j 在 $0\leqslant j<i$ 变化时，对于某个 j_r，产生一个点 $v_r=(x_r,y_r)$，这个点在一条直线 $y=kx+b_r$ 上，b_r 为截距，如图 5.19 所示。

对于 $0\leqslant j<i$ 中所有的 j，把它们对应的点都画在平面上，这些点对应的直线的斜率 $k=a[i]$都相同，只有截距 b 不同。在所有这些点中，有一个点 v'所在的直线有最小截距 b'，计算出 b'，由于 b'中包含 dp[i]，那么就算出了最优的 dp[i]，如图 5.20 所示。

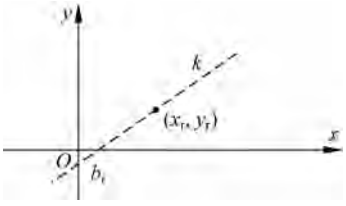

图 5.19 经过点(x_r,y_r)的直线

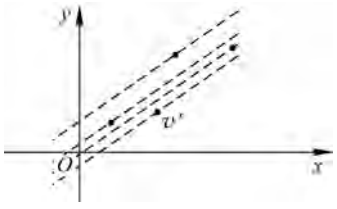

图 5.20 经过最优点 v'的直线

如何找最优点 v'？利用“下凸壳”。

前面提到，x 是单调增加的，即 x 随着 j 递增而递增。图 5.21(a)中给出了 4 个点，它们的 x 坐标是递增的。

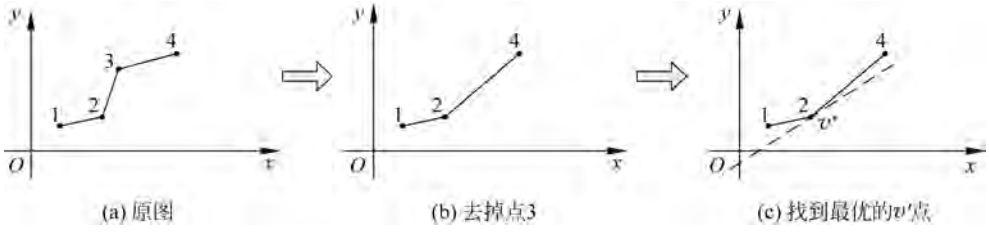

图 5.21 用下凸壳找最优点

图 5.21(a)中的点 1、2、3 构成了下凸壳，下凸壳的特征是线段 12 的斜率小于线段 23 的斜率。点 2、3、4 构成了上凸壳。经过上凸壳中间点 3 的直线，其截距 b 肯定小于经过点 2 或点 4 的有相同斜率的直线的截距，所以点 3 肯定不是最优点，去掉它。

去掉上凸壳后，得到图 5.21(b)，留下的点都满足下凸壳关系。最优点就在下凸壳上。例如，在图 5.21(c)中，用斜率为 k 的直线来切这些点，设线段 12 的斜率小于 k，线段 24 的斜率大于 k，那么点 2 就是下凸壳的最优点。

以上操作用单调队列编程很方便。

(1) 入队操作，队列处理的数据为斜率$(y_2-y_1)/(x_2-x_1)$。在队列内对这些斜率维护一个下凸壳，即每两个连续点连成的直线，其斜率是**单调**上升的。新的点进入队列时，确保它能与队列中的点一起仍然能够组成下凸壳。例如，队列尾部的两个点是 3 和 4，准备进入队列的新的点是 5。比较点 3、4、5，看线段 34 和线段 45 的斜率是否递增，如果斜率递增，那么点 3、4、5 形成了下凸壳；如果不递增，说明点 4 不对，从队尾弹走它；然后继续比较队列尾部的点 2、3 和 5；重复以上操作，直到 5 能进入队列为止。经过以上操作，队列内的点组成了一

个大的下凸壳，每两个连续点连成的直线斜率递增，队列保持为单调队列，如图 5.22 所示。

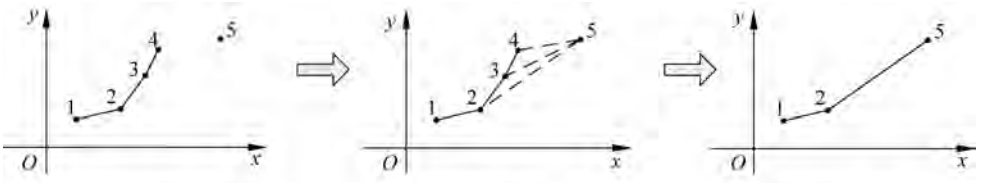

图 5.22　维护单调队列中的下凸壳

（2）出队列，找到最优点。设队头的两个点是 v_1 和 v_2，如果线段 v_1v_2 的斜率比 k 小，说明 v_1 不是最优点，弹走它，继续比较队头新的两个点，一直到斜率大于 k 为止，此时队头的点就是最优点 v'。

5.9.3　求所有 dp[i]

5.9.2 节求得了一个 dp[i]，复杂度为 $O(n)$。如果对于所有 i 都这样求 dp[i]，总复杂度仍然为 $O(n^2)$，并没有改变计算的复杂度。有优化的方法吗？

一个较小的 i_1，它对应的点是 $\{v_0, v_1, \cdots, v_{i1}\}$；一个较大的 i_2，对应了更多的点 $\{v_0, v_1, \cdots, v_{i1}, \cdots, v_{i2}\}$，其中包含了 i_1 的所有点。寻找 i_1 的最优点时，需要检查 $\{v_0, v_1, \cdots, v_{i1}\}$；寻找 i_2 的最优点时，需要检查 $\{v_0, v_1, \cdots, v_{i1}, \cdots, v_{i2}\}$。这里做了重复的检查，并且这些重复是可以避免的。这就是能优化的地方，仍然用下凸壳进行优化。

（1）每个 i 所对应的斜率 $k_i = a[i]$ 是不同的，根据约束条件 $a[i] \leqslant a[i+1]$，当 i 增大时，斜率递增，如图 5.23 所示。

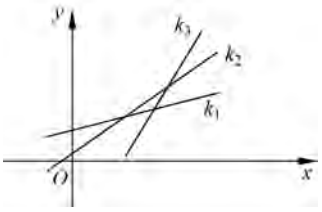

图 5.23　多个 i 对应的直线

（2）前面已经提到，对一个 i_1 找它的最优点时，可以去掉一些点，即那些斜率比 k_{i1} 小的点。这些被去掉的点，在后面更大的 i_2 时，由于斜率 k_{i2} 也更大，肯定也要被去掉。

根据上面的讨论，优化方法是对所有的 i，统一用一个单调队列处理所有点；被较小的 i_1 去掉的点被单调队列弹走，后面更大的 i_2 不再处理它们。

因为每个点只进入单调队列一次，总复杂度为 $O(n)$。

下面的代码演示了以上操作。

```
1  //q[]是单调队列,head 指向队首,tail 指向队尾
2  //队列中的元素是斜率,用 slope()函数计算两点组成的直线的斜率
3  for(int i = 1;i < = n;i++){
4      while(head < tail && slope(q[head],q[head + 1])< k)   //队头的两点线段斜率小于 k
5          head++;                                          //不合格,从队头弹出
6      int j = q[head];                                     //队头是最优点
7      dp[i] = ...;                                         //计算 dp[i]
8      while(head < tail && slope(i,q[tail - 1])< slope(q[tail - 1],q[tail]))   //入队操作
9          tail -- ;                                        //弹走队尾不合格的点
10     q[++tail] = i;                                       //新的点进入队列
11 }
```

为加深对上述代码的理解，考虑一个特例：进入队列的点都符合下凸壳特征，且这些点连成的直线的斜率大于所有斜率 k_i，那么结果是队头不会被弹出，入队的点也不会被弹出，队头被重复使用 n 次，如图 5.24 所示。

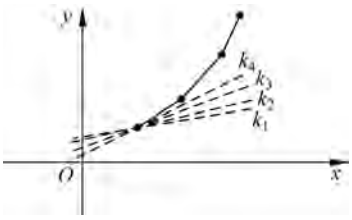

图 5.24　一个特例

5.9.4　例题

下面用一道例题给出典型代码。

例 5.21　Print article(hdu 3507)

问题描述：打印一篇包含 N 个单词的文章，第 i 个单词的打印成本为 C_i。在一行中打印 k 个单词的花费为 $\left(\sum_{i=1}^{k} C_i\right)^2 + M$，$M$ 为一个常数。如何安排文章，才能最小化费用？

输入：有很多测试用例。对于每个测试用例，第 1 行输入两个数字 N 和 M($0 \leqslant N \leqslant 500000$，$0 \leqslant M \leqslant 1000$)。在接下来的第 $2 \sim N+1$ 行中，输入 N 个数字。输入 EOF 终止。

输出：打印文章的最低费用。

题目的意思是有 N 个数和一个常数 M，把这 N 个数分成若干部分，每部分的计算值为这部分数的和的平方加上 M，总计算值为各部分计算值之和，求最小的总计算值。由于 N 很大，$O(N^2)$的算法超时。

设 dp[i]表示输出前 i 个单词的最小费用，DP 状态转移方程为

$$\mathrm{dp}[i] = \min\{\mathrm{dp}[j] + (\mathrm{sum}[i] - \mathrm{sum}[j])^2 + M\}, \quad 0 < j < i$$

其中，sum[i]表示前 i 个数字和。

看到这个转移方程，读者能想到它可以用斜率优化吗？虽然不可能一眼看出来，但是在阅读下面内容之前可以试试，看能不能套用 $y = kx + b$ 这种斜率优化 DP 的标准形式，也就是把状态转移方程转换为常数 k、b，以及变量 x、y。

下面把 DP 状态转移方程改写为 $y = kx + b$ 的形式。首先展开方程，得

$$\mathrm{dp}[i] = \mathrm{dp}[j] + \mathrm{sum}[i]\mathrm{sum}[i] + \mathrm{sum}[j]\mathrm{sum}[j] - 2\mathrm{sum}[i]\mathrm{sum}[j] + M$$

移项得

$$\mathrm{dp}[j] + \mathrm{sum}[j]\mathrm{sum}[j] = 2\mathrm{sum}[i]\mathrm{sum}[j] + \mathrm{dp}[i] - \mathrm{sum}[i]\mathrm{sum}[i] - M$$

对照 $y = kx + b$，有

$y=\mathrm{dp}[j]+\mathrm{sum}[j]\mathrm{sum}[j]$，$y$ 只与 j 有关；

$x=2\mathrm{sum}[j]$，x 只与 j 有关，且随着 j 递增而递增；

$k=\mathrm{sum}[i]$，k 只与 i 有关，且随着 i 递增而递增；

$b=\mathrm{dp}[i]-\mathrm{sum}[i]\mathrm{sum}[i]-M$，$b$ 只与 i 有关，且包含 $\mathrm{dp}[i]$。

下面给出代码。注意一个小技巧：虽然在理论上单调队列处理的是斜率 $(y(a)-y(b))/(x(a)-x(b))$，但是编码时不用这样算，队列只记录 a 和 b，判断时用 $(y(a)-y(b))/(x(a)-x(b))$ 进行计算即可。细节详见代码。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 500010;
int dp[N];
int q[N];                                                    //单调队列
int sum[N];
int X(int x){ return 2 * sum[x]; }
int Y(int x){ return dp[x] + sum[x] * sum[x]; }
//double slope(int a,int b){return (Y(a) - Y(b))/(X(a) - X(b));} //除法不好,改为下面的乘法
int slope_up  (int a,int b) { return Y(a) - Y(b);}           //斜率的分子部分
int slope_down(int a,int b) { return X(a) - X(b);}           //斜率的分母部分
int main(){
    int n,m;
    while(~scanf("%d%d",&n,&m)){
        for(int i = 1;i <= n;i++)  scanf("%d",&sum[i]);
        sum[0] = dp[0] = 0;
        for(int i = 1;i <= n;i++)  sum[i] += sum[i - 1];
        int head = 1,tail = 1;                               //队头/队尾
        q[tail] = 0;
        for(int i = 1;i <= n;i++){
            while(head < tail &&
                  slope_up(q[head + 1],q[head])<= sum[i] * slope_down(q[head + 1],
                  q[head]))
               head++;                                       //斜率小于k,从队头弹走
            int j = q[head];                                 //队头是最优点
            dp[i] = dp[j] + m + (sum[i] - sum[j]) * (sum[i] - sum[j]);     //计算 dp[i]
            while(head < tail &&
                           slope_up(i,q[tail]) * slope_down(q[tail],q[tail - 1])
                        <= slope_up(q[tail],q[tail - 1]) * slope_down(i,q[tail]))
                tail--;                                      //弹走队尾不合格的点
            q[++tail] = i;                                   //新的点进队尾
        }
        printf("%d\n",dp[n]);
    }
    return 0;
}
```

【习题】

(1) kuangbing 带你飞“专题二十斜率 DP”(https://vjudge.net/article/55)。

(2) 洛谷 P2900/P3628/P3648/P4360/P5468/P2305。

(3) 洛谷 P3195(玩具装箱)：DP 状态转移方程为 dp[i]=min{dp[j]+(sum[i]+i−sum[j]−j−L−1)^2}。

(4) 洛谷 P4072(征途)：二维斜率优化，DP 状态转移方程为 dp[i][p]=min{dp[j][p−1]+(s[i]−s[j])^2}。

扫一扫

视频讲解

5.10 四边形不等式优化

将四边形不等式(Quadrangle Inequality)应用于 DP 优化，起源于 1971 年 Knuth 的一篇论文[①]，用来解决最优二叉搜索树问题。1980 年，储枫(F. Frances Yao，姚期智的夫人)做了深入研究[②]，扩展为一般性的 DP 优化方法，把一些 DP 问题复杂度从 $O(n^3)$ 优化到 $O(n^2)$。所以，这个方法又称为 Knuth-Yao DP Speedup Theorem。

四边形不等式应用于 DP 优化的证明比较复杂，如果先给出定义和证明，会让人迷惑，所以本节按这样的顺序讲解：先用应用场合引导出四边形不等式的概念，再进行定义和证明，最后用例题巩固。四边形不等式优化虽然理论复杂，但是**编码很简单**。

5.10.1 应用场合

有一些常见的 DP 问题，通常是区间 DP 问题，它的状态转移方程为

$$\text{dp}[i][j]=\min\{\text{dp}[i][k]+\text{dp}[k+1][j]+w[i][j]\}$$

其中，$i\leqslant k<j$，初始值 dp[i][i]已知。min()也可以是 max()。

方程的含义如下。

(1) dp[i][j]表示从状态 i 到状态 j 的最小花费。题目一般是求 dp[1][n]，即从起点 1 到终点 n 的最小花费。

(2) dp[i][k]+dp[k+1][j]体现了递推关系。k 在 i 和 j 之间滑动，k 有一个最优值，使 dp[i][j]最小。

(3) w[i][j]的性质非常重要。w[i][j]是与题目有关的费用，如果它满足四边形不等式和单调性，那么计算时就能进行四边形不等式优化。

这类问题的经典的例子是“石子合并”[③]，在 5.5 节有介绍，它的状态定义就是上面的 dp[i][j]，表示区间[i,j]上的最优值，w[i][j]是从第 i 堆石子合并到第 j 堆石子的总数量。

在阅读后面的讲解时，读者可以对照“石子合并”这个例子来理解。注意，石子合并有多种情况和解法，详情见本节例题洛谷 P1880。

重新回顾 5.5 节求 dp[i][j]的代码，作为后面优化代码的对照。代码中有三重循环，复杂度为 $O(n^3)$。

---

① Knuth D E. Optimum Binary Search Trees[J]. Acta Informatica, 1971, 1: 14-25.

② Yao F F. Efficient Dynamic Programming Using Quadrangle Inequalities[C]//Proceedings of the 12th Annual ACM Symposium on Theory of Computing, 1980. http://www.cs.ust.hk/mjg_lib/bibs/DPSu/DPSu.Files/p429-yao.pdf.

③ 参考《算法竞赛入门到进阶》7.3 节“区间 DP”中的石子合并问题。

```
for(int i=1; i<=n; i++)  dp[i][i] = 0;          //初始值
for(int len = 2; len<= n; len++)     //len为区间[i,j]的长度.从小区间扩展到大区间
    for(int i = 1; i<= n-len+1; i++){          // 区间起点 i
        int j = i + len - 1;                   // 区间终点 j,i<=j<=n
        for(int k = i; k<j; k++)     //大区间[i,j]从小区间[i,k]和[k+1,j]转移而来
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j] + w[i][j]);
}
```

5.10.2 四边形不等式优化操作

只需一个简单的优化操作，就能把上面代码的时间复杂度降为 $O(n^2)$。这个操作就是把循环 $i \leqslant k < j$ 改为 $s[i][j-1] \leqslant k \leqslant s[i+1][j]$。$s[i][j]$记录 $i \sim j$ 的最优分割点，在计算 dp[i][j]的最小值时得到区间[i,j]的分割点 k，记录在 $s[i][j]$中，用于下一次循环。

这个优化称为**四边形不等式优化**。下面给出优化后的代码，优化见**加粗**部分。

```
for(i = 1;i<= n;i++){
    dp[i][i] = 0;
    s[i][i] = i;                                          //s[][]的初始值
}
for(int len = 2; len<= n; len++)
    for(int i = 1; i<= n-len+1; i++){
        int j = i + len - 1;
        for(k = s[i][j - 1]; k<= s[i + 1][j]; k++){       //缩小循环范围
            if(dp[i][j] > dp[i][k] + dp[k + 1][j] + w[i][j]){   //是否更优
                dp[i][j] = dp[i][k] + dp[k + 1][j] + w[i][j];
                s[i][j] = k;                              //更新最佳分割点
            }
    }
}
```

代码的**复杂度**如何？

代码中 i 和 k 这两个循环，优化前复杂度为 $O(n^2)$。优化后，每个 i 内部的 k 的循环次数为 $s[i+1][j]-s[i][j-1]$，其中 $j=i+\text{len}-1$。那么：

$i=1$ 时，k 循环 $s[2][\text{len}]-s[1][\text{len}-1]$次；

$i=2$ 时，k 循环 $s[3][\text{len}+1]-s[2][\text{len}]$次；

…

$i=n-\text{len}+1$ 时，k 循环 $s[n-\text{len}+2][n]-s[n-\text{len}+1][n+1]$次。

将上述次数相加，总次数为

$$s[2][\text{len}]-s[1,\text{len}-1]+s[3][\text{len}+1]-s[2,\text{len}]+\cdots+s[n+1,n]-s[n][n]$$
$$=s[n-\text{len}+2][n]-s[1][\text{len}-1]<n$$

i 和 k 循环的时间复杂度优化到了 $O(n)$，总复杂度从 $O(n^3)$优化到了 $O(n^2)$。

在后面的四边形不等式定理证明中，将更严谨地证明复杂度。

图 5.25 所示为四边形不等式优化效果，s_1 是区间[i,$j-1$]的最优分割点，s_2 是区间[$i+1$,j]的最优分割点。

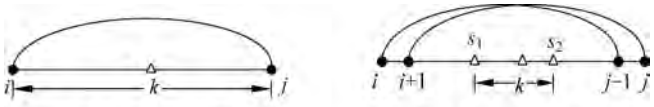

图 5.25　四边形不等式优化效果

读者对代码可能有两个疑问：

(1) 为什么能够把 $i \leqslant k < j$ 缩小到 $s[i][j-1] \leqslant k \leqslant s[i+1][j]$？

(2) $s[i][j-1] \leqslant s[i+1][j]$ 成立吗？

下面给出四边形不等式优化的正确性和复杂度的严谨证明，会解答这两个问题。

5.10.3　四边形不等式定义和单调性定义

在四边形不等式 DP 优化中，对于费用 w，有两个关键内容：四边形不等式定义和单调性。

(1) **四边形不等式定义 1**：设 w 是定义在整数集合上的二元函数，对于任意整数 $i \leqslant i' \leqslant j \leqslant j'$，如果有 $w(i,j)+w(i',j') \leqslant w(i,j')+w(i',j)$，则称 w 满足四边形不等式。

四边形不等式可以概况为：两个交错区间的 w 的和小于或等于小区间与大区间的 w 的和。

为什么称为“四边形”？如图 5.26 所示，把它变成一个几何图，画成平行四边形 $i'ijj'$。图中对角线长度和 $ij+i'j'$ 大于平行线长度和 $ij'+i'j$，这与四边形不等式的几何性质是相反的，所以可以理解成“**反**四边形不等式”。请读者注意，这个“四边形”只是一个帮助理解的示意图，并没有严谨的意义。图 5.26 这种四边形是储枫论文中的画法，也有其他的四边形画法。当中间两点 $i'=j$ 时(两点重合)，四边形变成了三角形。

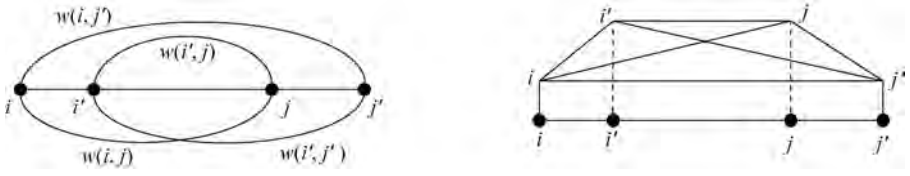

图 5.26　四边形不等式 $w(i,j)+w(i',j') \leqslant w(i,j')+w(i',j)$

定义 1 的特例是定义 2。

(2) **四边形不等式定义 2**：对于整数 $i < i+1 \leqslant j < j+1$，如果有 $w(i,j)+w(i+1,j+1) \leqslant w(i,j+1)+w(i+1,j)$，则称 w 满足四边形不等式。

定义 1 和定义 2 实际上等价，它们可以互相推导，请读者尝试证明。

(3) **单调性**：设 w 是定义在整数集合上的二元函数，如果对任意整数 $i \leqslant i' \leqslant j \leqslant j'$，有 $w(i,j') \geqslant w(i',j)$，称 w 具有单调性。

单调性可以形象地理解为：如果大区间包含小区间，那么大区间的 w 值大于或等于小区间的 w 值，如图 5.27 所示。

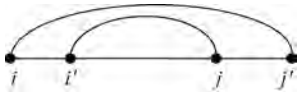

图 5.27　w 的单调性

在石子合并问题中，令 $w[i][j]$ 等于从第 i 堆石子合并到

第 j 堆石子的石子总数，它满足四边形不等式的定义、单调性。

(1) $w[i][j]+w[i'][j']=w[i][j']+w[i'][j]$，满足四边形不等式定义。

(2) $w[i][j']\geqslant w[i'][j]$，满足单调性。

利用 w 的四边形不等式、单调性的性质，可以推导出四边形不等式定理，用于 DP 优化。

5.10.4 四边形不等式定理

在储枫的论文中，提出并证明了四边形不等式定理[①]。

四边形不等式定理 如果 $w(i,j)$ 满足四边形不等式和单调性，则用 DP 计算 dp[][]的时间复杂度为 $O(n^2)$。

这个定理可以通过下面两个更详细的引理证明。

引理 1 状态转移方程 $dp[i][j]=\min(dp[i][k]+dp[k+1][j]+w[i][j])$，如果 $w[i][j]$ 满足四边形不等式和单调性，那么 $dp[i][j]$ 也满足四边形不等式。

引理 2 记 $s[i][j]=k$ 为 $dp[i][j]$ 取得最优值时的 k，如果 $dp[i][j]$ 满足四边形不等式，那么有 $s[i][j-1]\leqslant s[i][j]\leqslant s[i+1][j]$，即 $s[i][j-1]\leqslant k\leqslant s[i+1][j]$。

引理 2 直接用于 DP 优化，复杂度为 $O(n^2)$。

下面证明引理 1、引理 2、四边形不等式定理，部分内容翻译自储枫论文中对引理 1 和引理 2 的证明，并加上本书作者的解释。

定义方程：

$$\begin{cases} c(i,i)=0 \\ c(i,j)=w(i,j)+\min\{c(i,k-1)+c(k,j)\}, & i<k\leqslant j \end{cases} \tag{5.1}$$

前面例子中的 $dp[i][j]$ 和这里的 $c(i,j)$ 略有不同，$dp[i][j]=\min\{dp[i][k]+dp[k+1][j]+w[i][j]\}$，其中 $w[i][j]$ 在 min()内部。证明过程是一样的。

式(5.1)中的 w 要求满足四边形不等式和单调性，即

$$\begin{cases} w(i,j)+w(i',j')\leqslant w(i',j)+w(i,j'), & i\leqslant i'\leqslant j\leqslant j' \\ w(i',j)\leqslant w(i,j'), & [i',j]\subseteq[i,j'] \end{cases} \tag{5.2}$$

1. 证明引理 1

如果 $w(i,j)$ 满足四边形不等式和单调性，那么 $c(i,j)$ 也满足四边形不等式，即

$$c(i,j)+c(i',j')\leqslant c(i',j)+c(i,j'), \quad i\leqslant i'\leqslant j\leqslant j' \tag{5.3}$$

下面证明式(5.3)。

当 $i=i'$ 或 $j=j'$ 时，式(5.3)显然成立，下面考虑另外两个情况：$i<i'=j<j'$；$i<i'<j<j'$。

1) $i<i'=j<j'$

代入式(5.3)，得到一个“**反**三角形不等式”，如图 5.28 所示的三角形 ijj'，两边的和小于第 3 边。

$$c(i,j)+c(j,j')\leqslant c(i,j'), \quad i<j<j' \tag{5.4}$$

① 又称为 Knuth-Yao DP Speedup Theorem，DP 加速定理。

现在证明式(5.4)。

假设 $c(i,j')$ 在 $k=z$ 处有最小值，即 $c(i,j')=c_z(i,j')$。这里定义 $c_k(i,j)=w(i,j)+c(i,k-1)+c(k,j)$。

有两个对称情况：$z\leqslant j$；$z\geqslant j$。

$z\leqslant j$ 时，z 为 (i,j') 区间的最优点，不是 (i,j) 区间的最优点，所以有

$$c(i,j)\leqslant c_z(i,j)=w(i,j)+c(i,z-1)+c(z,j)$$

在两边加上 $c(j,j')$，有

$$\begin{aligned}c(i,j)+c(j,j')&\leqslant w(i,j)+c(i,z-1)+c(z,j)+c(j,j')\\&\leqslant w(i,j')+c(i,z-1)+c(z,j')\\&=c(i,j')\end{aligned}$$

上面的推导利用了：

(1) w 的单调性，有 $w(i,j)\leqslant w(i,j')$；

(2) 式(5.4)的归纳假设：假设 $z\leqslant j\leqslant j'$ 时成立，递推出 $i<j<j'$ 时式(5.4)也成立。观察图 5.28，有 $c(z,j)+c(j,j')\leqslant c(z,j')$，它满足**反**三角形不等式。

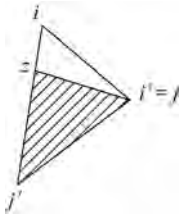

图 5.28 引理 1 证明 ($z\leqslant j$)①

$z\geqslant j$ 是上述 $z\leqslant j$ 的对称情况。

2) $i<i'<j<j'$

假设式(5.3)右边的小区间 $c(i',j)$ 和大区间 $c(i,j')$ 分别在 $k=y$ 和 $k=z$ 处有最小值，记为

$$c(i',j)=c_y(i',j)$$
$$c(i,j')=c_z(i,j')$$

同样有两个对称情况：$z\leqslant y$；$z\geqslant y$。

$z\leqslant y$ 时，有

$$c(i',j')\leqslant c_y(i',j')$$
$$c(i,j)\leqslant c_z(i,j)$$

将两式相加，得

$$\begin{aligned}&c(i,j)+c(i',j')\\&\leqslant c_z(i,j)+c_y(i',j')\\&=w(i,j)+w(i',j')+c(i,z-1)+c(z,j)+c(i',y-1)+c(y,j')\end{aligned}\tag{5.5}$$

式(5.5)的进一步推导利用了：

(1) 根据 w 的四边形不等式，有 $w(i,j)+w(i',j')\leqslant w(i',j)+w(i,j')$；

(2) 根据式(5.3)的归纳假设，即假设 $z\leqslant y<j<j'$ 时成立。观察图 5.29，有 $c(z,j)+c(y,j')\leqslant c(y,j)+c(z,j')$，满足反四边形不等式。

则式(5.5)变为

$$\begin{aligned}&c(i,j)+c(i',j')\\&\leqslant w(i',j)+w(i,j')+c(i,z-1)+c(i',y-1)+c(y,j)+c(z,j')\end{aligned}$$

① 图 5.28 和图 5.29 引自储枫论文。

$$\leqslant c_y(i',j)+c_z(i,j')$$
$$=c(i',j)+c(i,j')$$

$z\geqslant y$ 是 $z\leqslant y$ 的对称情况。

引理 1 证毕。

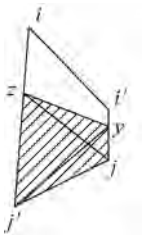

图 5.29　引理 1 的证明（$z\leqslant y$）

2. 证明引理 2

用 $K_c(i,j)$表示 $\max\{k\mid c_k(i,j)=c(i,j)\}$，也就是使 $c(i,j)$得到最小值的那些 k 中，最大的那个是 $K_c(i,j)$。定义 $K_c(i,i)=i$。$K_c(i,j)$就是前面例子中的 $s[i][j]$。

引理 2 可表述为

$$K_c(i,j)\leqslant K_c(i,j+1)\leqslant K_c(i+1,j+1) \tag{5.6}$$

下面证明引理 2。

$i=j$ 时，显然成立。下面假设 $i<j$。

先证明式(5.6)的第 1 部分 $K_c(i,j)\leqslant K_c(i,j+1)$。这等价于证明对于 $i<k\leqslant k'\leqslant j$，有

$$c_{k'}(i,j)\leqslant c_k(i,j)\Rightarrow c_{k'}(i,j+1)\leqslant c_k(i,j+1) \tag{5.7}$$

式(5.7)的意思是如果 $c_{k'}(i,j)\leqslant c_k(i,j)$成立，那么 $c_{k'}(i,j+1)\leqslant c_k(i,j+1)$也成立。$c_{k'}(i,j)\leqslant c_k(i,j)$的含义是在$[i,j]$区间，$k'$是比 k 更好的分割点，可以把 k'看作$[i,j]$的最优分割点。扩展到区间$[i,j+1]$时，有 $c_{k'}(i,j+1)\leqslant c_k(i,j+1)$，即 k'仍然是比 k 更好的分割点。也就是说，区间$[i,j+1]$的最优分割点肯定大于或等于 k'。

下面证明式(5.7)。

根据四边形不等式，$k\leqslant k'\leqslant j<j+1$ 时，有

$$c(k,j)+c(k',j+1)\leqslant c(k',j)+c(k,j+1)$$

两边加上 $w(i,j)+w(i,j+1)+c(i,k-1)+c(i,k'-1)$，得

$$c_k(i,j)+c_{k'}(i,j+1)\leqslant c_{k'}(i,j)+c_k(i,j+1)$$

移项，得

$$c_{k'}(i,j+1)\leqslant c_{k'}(i,j)+c_k(i,j+1)-c_k(i,j) \tag{5.8}$$

把式(5.7)中 $c_{k'}(i,j)\leqslant c_k(i,j)$的两边加上 $c_k(i,j+1)$，得

$$c_{k'}(i,j)+c_k(i,j+1)\leqslant c_k(i,j)+c_k(i,j+1)$$
$$c_{k'}(i,j)+c_k(i,j+1)-c_k(i,j)\leqslant c_k(i,j+1)$$

结合式(5.8)，得 $c_{k'}(i,j+1)\leqslant c_k(i,j+1)$，式(5.7)成立。

同样可以证明，式(5.6)的右半部分 $K_c(i,j+1)\leqslant K_c(i+1,j+1)$，在 $i<i+1\leqslant k\leqslant k'$ 时成立。

引理 2 说明当 i、j 增大时，$K_c(i,j)$是非递减的。

3. 证明四边形不等式定理

利用引理 2，可推论出四边形不等式定理，即用 DP 计算所有的 $c(i,j)$的时间复杂度为 $O(n^2)$。下面对这一结论进行说明。

用 DP 计算 $c(i,j)$时，是按 $\delta=j-i=0,1,2,\cdots,n-1$ 的间距逐步增加进行递推计算

的。具体过程请回顾 5.10.1 节中求 dp[i][j]的代码。从 $c(i,j)$递推到 $c(i,j+1)$时，只需要 $K_c(i+1,j+1)-K_c(i,j)$次最少限度的操作就够了。总次数是多少呢？对于一个固定的 δ，计算所有的 $c(i,j)$，$1\leqslant i\leqslant n-\delta$，$j=i+\delta$。

$i=1$ 时，$K_c(1+1,1+\delta+1)-K_c(1,\delta+1)=K_c(2,\delta+2)-K_c(1,\delta+1)$；

$i=2$ 时，$K_c(2+1,2+\delta+1)-K_c(2,\delta+2)=K_c(3,\delta+3)-K_c(2,\delta+2)$；

$i=3$ 时，$K_c(3+1,3+\delta+1)-K_c(3,\delta+3)=K_c(4,\delta+4)-K_c(3,\delta+3)$；

…

$i=n-\delta$ 时，$K_c(n-\delta+1,n-\delta+\delta+1)-K_c(n-\delta,\delta+n-\delta)=K_c(n-\delta+1,n+1)-K_c(n-\delta,n)$。

将以上式子相加，总次数为 $K_c(n-\delta+1,n+1)-K_c(1,\delta+1)<n$。

对于一个 δ，计算次数为 $O(n)$；有 n 个 δ，总计算复杂度为 $O(n^2)$。

以上证明了四边形不等式定理。

4. min()和 max()

前面讨论的都是 min()，如果是 max()，也可以进行四边形不等式优化。此时四边形不等式是“反”的，即

$$w(i,j)+w(i',j')\geqslant w(i',j)+w(i,j'),\quad i\leqslant i'\leqslant j\leqslant j'$$

定义

$$c(i,j)=w(i,j)+\max(a(i,k)+b(k,j)),\quad i\leqslant k\leqslant j$$

引理 3 若 w、a、b 都满足反四边形不等式，那么 c 也满足反四边形不等式。

引理 4 如果 a 和 b 满足反四边形不等式，那么

$$K_c(i,j)\leqslant K_c(i,j+1)\leqslant K_c(i+1,j+1),\quad i\leqslant j$$

引理 3 和引理 4 的证明与引理 1 和引理 2 的证明类似。

5.10.5 例题

拿到题目后，先判断 w 是否单调、是否满足四边形不等式，再使用四边形不等式优化 DP。下面给出两道经典题。

1. 石子合并

例 5.22 洛谷 P1880

问题描述：在一个圆形操场的四周摆放 N 堆石子，现要将石子有次序地合并成一堆。规定每次只能选相邻的两堆合并成新的一堆，并将新的一堆的石子数记为该次合并的得分。

试设计一个算法，计算出将 N 堆石子合并成一堆的最小得分和最大得分。

输入：第 1 行输入正整数 N，表示有 N 堆石子。第 2 行输入 N 个整数，第 i 个整数 a_i 表示第 i 堆石子的个数。

输出：共输出两行，第 1 行为最小得分，第 2 行为最大得分。

在5.5节中，已经指出石子合并不能用贪心法，需要用DP。状态转移矩阵dp[i][j]前文已有说明，这里不再赘述。

用四边形优化DP求解石子合并的最小值，复杂度为$O(n^2)$。最小值用四边形不等式优化DP求解，w在四边形不等式中取等号：$w[i][j]+w[i'][j']=w[i][j']+w[i'][j]$。

本题的石子堆是环状的，转换为线形更方便处理。复制和原来一样的数据，头尾接起来，使长度为n的数列转换为长度为$2n$的数列，变成线性的，这是一个**重要的技巧**。

本题除了求石子合并的最小值，还要求最大值。虽然最大值也用DP求解，但是它不满足**反**四边形不等式的单调性要求，不能优化。而且也没有必要优化，可以用简单的推理得到：区间[i,j]石子合并的最大值等于区间[i,$j-1$]和[$i+1$,j]中的石子合并最大值加$w(i,j)$。

石子合并问题的最优解是Garsia-Wachs算法，复杂度为$O(n\log_2 n)$。参考洛谷P5569，本题$N\leqslant 40000$，用DP会超时。

2. 最优二叉搜索树

最优二叉搜索树是Knuth提出的经典问题，是四边形不等式优化的起源。

例5.23　Optimal binary search tree(uva10304[①])

问题描述：给定有n个不同元素的集合$S=(e_1,e_2,\cdots,e_n)$，有$e_1<e_2<\cdots<e_n$，用S的元素建一棵二叉搜索树，希望查询频率越高的元素离根越近。访问树中元素e_i的成本cost(e_i)等于从根到该元素节点的路径数。给定元素的查询频率$f(e_1),f(e_2),\cdots,f(e_n)$，定义一棵树的总成本为

$$f(e_1)\text{cost}(e_1)+f(e_2)\text{cost}(e_2)+\cdots+f(e_n)\text{cost}(e_n)$$

总成本最低的树就是最优二叉搜索树。

输入：输入包含多个实例，每行一个。每行以输入$1\leqslant n\leqslant 250$开头，表示$S$的大小。在$n$之后输入$n$个非负整数，表示元素的查询频率$f(e_i)$，$0\leqslant f(e_i)\leqslant 100$。

输出：对于输入的每个实例，打印最优二叉搜索树的总成本。

输入样例：	输出样例：
1 5	0
3 10 10 10	20
3 5 10 20	20

① https://vjudge.net/problem/UVA-10304

二叉搜索树(BST)的特点是每个节点的值比它的左子树上所有节点的值大，比右子树上所有节点的值小。二叉搜索树的中序遍历是从小到大的排列。上述第 3 个样例的最优二叉搜索树的形状如图 5.30 所示，它的总成本为 $5\times2+10\times1=20$。

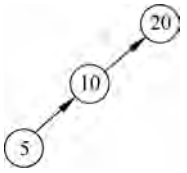

图 5.30　二叉搜索树

题目给的元素已经按照从小到大排列，可以方便地组成一棵 BST。

设 dp[i][j]是区间[i,j]的元素组成的 BST 的最小值。把区间[i,j]分成两部分：[i,$k-1$]和[$k+1$,j],k 在 i 和 j 之间滑动。在区间[i,j]建立的二叉树，k 是根节点。这是典型的区间 DP，状态转移方程为

$$dp[i][j]=\min\{dp[i][k-1]+dp[k+1][j]+w(i,j)-e[k]\}$$

其中，$w(i,j)$为区间和，$w(i,j)=f(e_i)+f(e_{i+1})+\cdots+f(e_j)$。当把左、右子树连在根节点上时，本身的深度增加 1，所以每个元素都多计算一次，这样就解决了 $\text{cost}(e_i)$的计算。最后，因为根节点 k 的层数是 0，所以减去根节点的值 $e[k]$。

$w(i,j)$符合四边形不等式优化的条件，所以 dp[i][j]可以用四边形不等式优化，算法的复杂度为 $O(n^2)$。

最优二叉搜索树问题的最优解也是 Garsia-Wachs 算法，复杂度为 $O(n\log_2 n)$。

【习题】

很多区间 DP 问题都能用四边形不等式优化。

(1) hdu 3516/2829/3506/3480。

(2) 洛谷 P1912/P4767/P4072。

小　结

DP 是典型的体现计算思维的知识点，与搜索一样，是算法竞赛最常见的考点之一，每次竞赛都会出 DP 类型题。

本章介绍了 DP 的一部分常用内容，还有一些内容请读者自己了解，如：

(1) DP 优化：带权二分优化、哈希表优化、分治优化等；

(2) DP 应用：概率 DP、DP 套 DP、动态 DP、插头 DP 等。

“插头 DP[①]”这个名词在中国算法竞赛选手中的流行，源于 2008 年信息学国家集训队陈丹琦[②]的论文《基于连通性状态压缩的动态规划问题》[③]。这篇文章的关键词有状态压缩、连通性、括号表示法、轮廓线、插头、棋盘模型，都与插头 DP 相关，参考模板题洛谷 P5056。

① 插头 DP，陈丹琦将其翻译为 Plug-like Dynamic Programming。

② https://www.cs.princeton.edu/~danqic/

③ https://www.cs.princeton.edu/~danqic/misc/dynamic-programming.pdf