常见动态规划模型及其优化

长沙市雅礼中学　屈运华

区间DP，四边形不等式优化DP，状态压缩DP，单调队列优化DP，斜率优化的DP

动态规划基本模型——区间DP

区间DP是常见的动态规划模型。

我们用“石子合并”问题这个例子来解释区间DP。

区间DP——石子合并

题目描述：有n堆石子排成一排，每堆石子有一定的数量。将n堆石子并成为一堆，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，求总的最小花费。

输入：第一行是整数n，表示有n堆石子。第二行有n个数，分别表示这n堆石子的数目。

输出：总的最小花费。

输入样例：

3

2 4 5

输出样例：

17

提示：样例的计算过程是：第一次合并2+4=6；第二次合并6+5=11；总花费6+11=17。

区间DP——石子合并

定义dp[i][j]为合并第i堆到第j堆的最小花费。

状态转移方程是：

dp[i][j]=min{dp[i][k]+dp[k+1][j]+w[i][j]}　　　i≤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≤k<j，即k在i、j之间滑动），返回那个最优的合并。子区间再继续分解为更小的区间，最小的区间[i,i+1]只包含两堆石子。

区间DP——例题1

题目描述：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：第一行是字符串A，第二行是字符串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：

zzzzzfzzzzz

abcdefedcba

输出样例：

6

提示：第1次把zzzzzfzzzzz转换为aaaaaaaaaaa，第2次转为abbbbbbbbba，第3次转为abccccccccba...

题目网址

http://acm.hdu.edu.cn/showproblem.php?pid=2476

区间DP——例题1分析

（1）从空白串转换到B

先考虑简单一点的问题：从空白串转换到B。为方便阅读代码，把字符串存储为B[1]~B[n]，不从0开始，编码的时候这样输入：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的最少步数。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时的最少步数。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）若B[i] = B[j]。第一次刷用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2个区间[i+1,j]、[i,j−1]，这2个区间的最小步数相等，也等于原区间[i,j]的最小步数。例如B[i,j]= “abbba” ，先用"a"全部刷一遍，再刷1次"bbb"，共刷2次。如果去掉第一个"a"，剩下的"bbba"，也是刷2次。

2）若B[i] ≠ B[j]。因为两端点不等，至少要各刷1次。用标准的区间操作，把区间分成i,k]和[k+1,j]两部分，枚举最小步数。

区间DP——例题1分析

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

2）若A[j] ≠ 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，枚举最小步数。这里利用了上一步从空白转换到B的结果，当区间[k+1,j]内A和B的字符不同时，从A转到B，与从空白串转换到B是等价的。

区间DP——例题2

题目描述：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的火气是Di，他排在第k个时，愤怒值是(k – 1) * D i。导演不想看到会场气氛紧张。他安排了一个黑屋，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（黑屋就是一个堆栈）。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二个，而最后一个男孩脾气最坏，导演为了让这个刺头第一个上场，把其他人全赶进了黑屋，结果你就排在了黑屋的第1名，第二个上场相亲了。

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和。

输入：第一行包含一个整数T，即数据组数。对于每种情况，第一行是n（0 <n <= 100），后面n行，整数D1 ...Dn

输出：对每个用例，输出最小愤怒值之和。

题目网址

http://acm.hdu.edu.cn/showproblem.php?pid=4283

区间DP——例题2分析

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

由于栈的存在，这一题的区间[i,j]的分割点k比较特殊。分割时，总是用区间[i,j]的第一个元素i把区间分成两部分，让i第k个从黑屋出来上场相亲，即第k个出栈。根据栈的特性：若第一个元素i第k个出栈，则第二到k-1个元素肯定在第一个元素之前出栈，第k+1到最后一个元素肯定在第 k个之后出栈。

例如，5个人排队序号是1、2、3、4、5。如果要第1（i=1）个人第3（k=3）个出场，那么栈的操作是这样：1进栈、2进栈、3进栈、3出栈、2出栈，1出栈。2号、3号在1号之前出栈，1号第3个出栈，4号5号在1号后面出栈。

区间DP——例题2分析

分割点k（1≤ k≤j-i+1）把区间划分成了两段，即dp[i+1][i+k-1]和dp[i+k][j]。dp[i][j]的计算分为三部分：

（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）第i个人往后挪了k-1个位置，愤怒值加上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 s u m [j] = $\sum_{i=1}^{j} D_i$, 是1~j所有人D值的和，sum[j]-sum[i+k-1]是区间[i+k,j]内这些人的D值和。

二维区间DP

区间[i,j]可以看成在一条直线上移动，即一维DP。

下面我们给出一个二维区间DP的例题，它的区间同时在两个方向移动。

二维区间DP——例题3　CF1199 F

题目描述： 有一个n×n大小的方格图，某些方格初始是黑色，其余为白色。一次操作，可以选定一个h×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变成白色。

输入： 第1行是整数n，表示方格的大小。后面有n行，每行长度为n的串，包含符号’ . ‘和’ #’ ，’ . ‘表示白色，’ #'表示黑色。第i行的第j个字符是(i, j)。n ≤ 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：

5

#...#

.#.#.

...

.#...

#...

输出样例：

5

二维区间DP——例题3分析

设矩形区域从左下角坐标(x1,y1)到右上角坐标(x2,y2)。定义状态dp[x1][y1][x2][y2]，表示把这个区域内染成白色的最小代价。

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

代码可以完全套用前面一维DP的模板，分别在两个方向操作。

（1）x方向，区间分为[x1,k]和[k+1,x2]。状态转移方程是：

dp[x1][][x2][] = min(dp[x1][][x2][], dp[x1][][k][] + dp [k+1][][x2][])

（2）y方向，区间分为 [y1,k]和[k+1,y2]。状态转移方程是：

dp[][y1][][y2] = min(dp[][y1][][y2], dp[][y1][][k] + dp[k+1][][y2])

区间DP优化——四边形不等式

有一些常见的DP问题，通常是区间DP问题，它的状态转移方程是：

dp[i][j]=min(dp[i][k]+dp[k+1][j]+w[i][j])

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

方程的含义是：

（1） dp[i][j]表示从i状态到j状态的最小花费。题目一般是求dp[1][n]，即从起始点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，使得dp[i][j]最小。

（3） w[i][j]的性质非常重要。 w[i][j]是和题目有关的费用，如果它满足四边形不等式和单调性，那么用DP计算dp的时候，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” ，它的转移矩阵就是上面的 dp[i][j]， w[i][j]是从第i堆石子到第j堆石子的总数量。

区间DP优化——四边形不等式

只需一个简单的优化操作，就能把上面代码的复杂度变为O(n^2)。

这个操作就是把循环i≤k<j改为：s[i][j-1]≤k≤s[i+1][j]其中s[i][j]记录从i到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。

区间DP优化——四边形不等式

```
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
            }
        }
    }
```

区间DP优化——四边形不等式

代码的复杂度是多少？

代码i和k这2个循环，优化前是O(n^2)的。优化后，每个i内部的k的循环次数是 s[i+1][j]-s[i][j-1]，其中j=i+len-1。那么：

i=1时， k循环s[2][len]-s[1][len-1]次。

i=2时， k循环 s[3][len+1]-s[2][len]次。

i=n-len+1时，k循环s[n-len+2][n]-s[n-len+1][n+1]次。

上述次数相加，总次数：

s[2][len]-s[1,len-1]+s[3][len+1]-s[2,len]+...+s[n+1,n]-s[n][n]

=s[n-len+2][n]-s[1][len-1]

<n

区间DP优化——四边形不等式

下图给出了四边形不等式优化的效果， s1是区间[i,j−1]的最优分割点，s2是区间[i+1,j]的最优分割点。

(1)优化前 k 的滑动范围　　(2)优化后 k 的滑动范围

(1) 为什么能够把i<=k<j缩小到s[i][j−1]≤k≤s[i+1][j]？

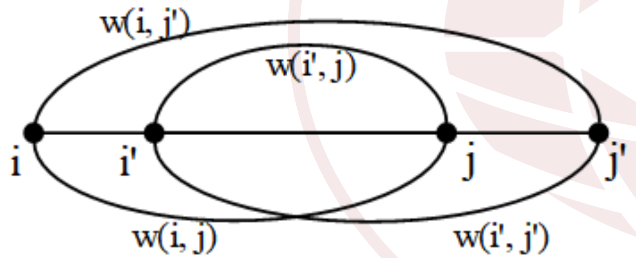
(2) s[i][j−1]≤s[i+1][j]成立吗？

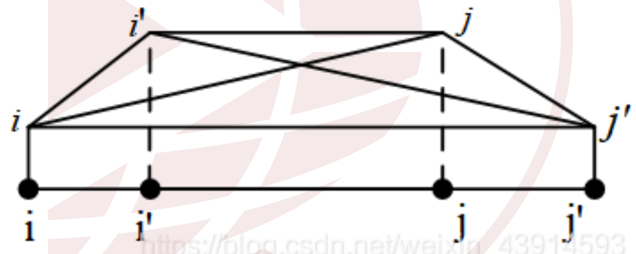
区间DP优化——四边形不等式定义和单调性定义

在四边形不等式DP优化中，对于w，有2个关键内容：四边形不等式定义、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 $i \le i' \le j \le j'$，如果有 $w(i, j) + w(i', j') \le w(i, j') + w(i', j)$,则称 w w w满足四边形不等式。

四边形不等式可以概括为：两个交错区间的w和，小于等于小区间与大区间的w和。





为什么被称为“四边形”？把它变成一个几何图，画成平行四边形，见上面图中的四边形i'ij'j，图中对角线长度和ij+i′j′大于平行线长度和ij′+i′j。

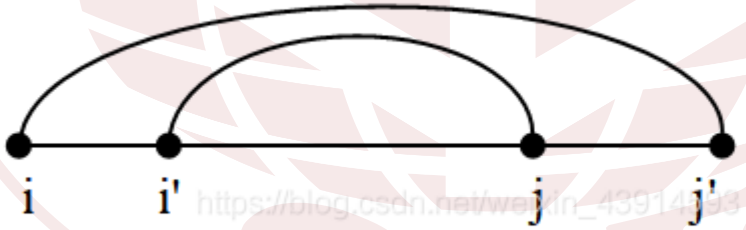
这个“四边形”只是一个帮助理解的示意图，并没有严谨的意义。也有其他的四边形画法，当中间两个点i′=j时，四边形变成了一个三角形。

区间DP优化——四边形不等式定义和单调性定义

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

（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$i \le i' \le j \le j'$，有 $w(i, j') \ge w(i', j)$，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值超过小区间的w值。



在石子合并问题中，令w[i][j]等于从第i堆石子加到第j堆石子的石子总数。它满足四边形不等式的定义、单调性：

$w[i][j']\ge w[i'][j]$，满足单调性；

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

利用 w w 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。

区间DP优化——四边形不等式定理（Knuth-Yao DP Speedup Theorem）

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[][] 的时间复杂度是O(n^2)的。

这个定理是通过下面2个更详细的引理来证明的。

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是dp[i][j]取得最优值时的k，如果dp满足四边形不等式，那么有s[i][j-1] ≤ s[i][j] ≤ s[i+1][j]，即s[i][j-1] ≤ k ≤ s[i+1][j]。

引理2直接用于DP优化，复杂度O(n^2)。

这两个引理的证明论文见下方的链接

http://www.cs.ust.hk/mjg_lib/bibs/DPSu/DPSu.Files/p429-yao.pdf

动态规划基本模型——状态压缩DP

提到状态压缩DP时，常常用Hamilton问题作为引例。

动态规划基本模型——状态压缩DP

最短Hamilton路径

时间限制：3s。

题目描述：给定一个有权无向图，包括n个点，标记为0 ~ 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，而且每个点只经过一次。$1 \le n \le 20$。

输入格式：第一行输入整数n。接下来n行每行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]）。$0 \le a[i, j] \le 10^7$

对于任意的x, y, z，数据保证 a[x, x]=0，a[x, y]=a[y, x] 并且 a[x, y]+a[y, z]>=a[x, z]。

输出格式：输出一个整数，表示最短Hamilton路径的长度。

动态规划基本模型——状态压缩DP

暴力解法：枚举n个点的全排列，共n!个全排列。一个全排列就是一条路径，计算这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，总复杂度是O(n×n!)。

Hamilton问题是NP问题，没有多项式复杂度的解法。不过，用状态压缩DP求解，能把复杂度降低到O(n^2×2^n) 。当n =20时，O(n^2×2^n) ≈ 4亿，比暴力法好很多。

首先定义DP。设S是图的一个子集，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中所有点，到达终点j时的最短路径；集合S中包括j点。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。

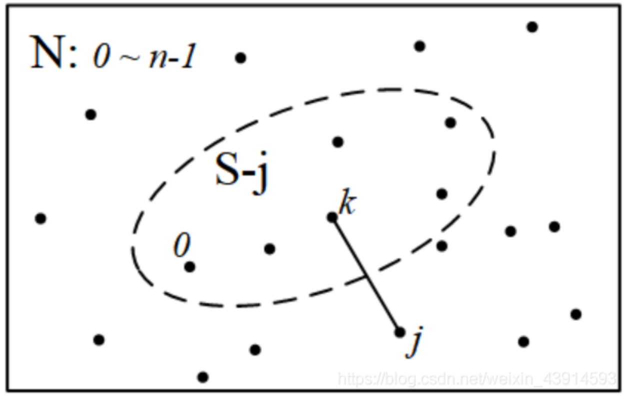
如何从S-j递推到S？设k是S-j中一个点，把从0到的路径分为两部分：(0→…→k) + (k→j)。以k为变量枚举S-j中所有的点，找出最短的路径，状态转移方程是：

dp[S][j]=min{dp[S−j][k]+dist(j,k)}

其中k属于集合S-j。

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

动态规划基本模型——状态压缩DP



上面是DP的设计，现在关键问题是如何操作集合S？
这就是状态压缩DP的技巧：用一个二进制数表示集合S，即把S“压缩”到一个二进制数中。 S的每一位表示图上的1个点，等于0表示S不包含这个点，等于1表示包含。
例如 S = 0000 0101，其中有两个1，表示集合中包含点2、0。
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。
后面给出了代码，第一个for循环有2^n次，加上后面2个各n次的for循环，总复杂度$O(n^2 \times 2^n)$。
第一个for循环，实现了从最小的集合扩展到整个集合。
最小的集合是S = 1，它的二进制数只有最后1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，包含了所有的点。
算法最关键的部分“枚举集合S-j中所有的点”，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)`也可以这样写： `S - (1<< j)`。

动态规划基本模型——状态压缩DP的原理

从上面的“引子”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来表示，用二进制的位运算来处理。

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

可以这样概况状态压缩DP的思想：集合的状态（子集或排列），如果用二进制表示状态，并用二进制的位运算来遍历和操作，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的，只能用于小规模问题的求解。

注意，一个问题用状态压缩DP求解，时间复杂度主要取决于DP算法，和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语言的位运算有 “&” ，"|"，"^"，"<<"，">>"等，下面是例子。虽然数字是用十进制表示的，但位运算是按二进制处理的。

动态规划基本模型——状态压缩DP的原理

用位运算可以简便地对集合进行操作，下表给出了几个例子，并在上面的代码中给出了示例。

(1)判断a的第i位（从最低位开始数）是否等于1:

1 << (i - 1)) & a

(2)把a的第i位改成1:

a | (1<<(i-1))

(3)把a的第i位改成0

a &(~(1<<i))

(4)把a的最后一个1去掉:

a & (a-1)

动态规划基本模型——轮廓线

Mondriaan’s Dream

题目描述：给定n行m列的矩形，用1×2的砖块填充，问有多少种填充方案。

输入格式：每一行是一个测试用例，包括两个整数：n和m。若n = m = 0表示终止。1 ≤ n, m ≤ 11。

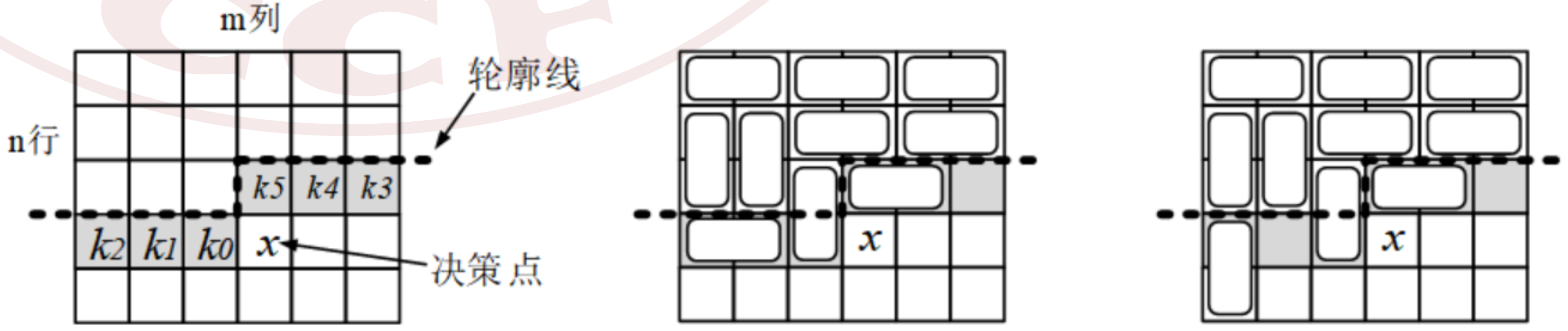
输出格式：对每个测试用例，输出方案数。

动态规划基本模型——轮廓线

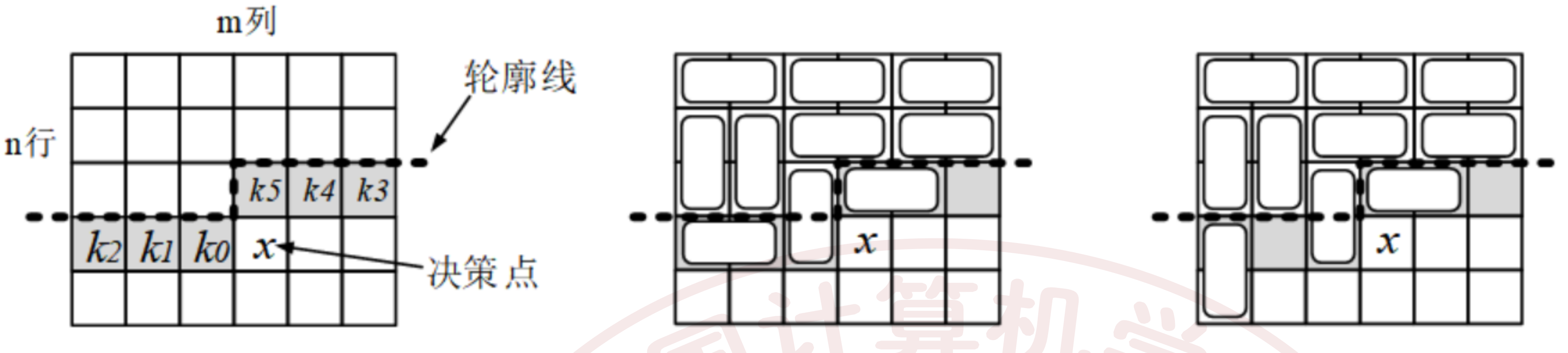
摆放砖头的操作步骤，可以从第一行第一列开始，从左往右、从上往下依次摆放。横砖只占1行，不影响下一行的摆放；竖砖占2行，会影响下一行。同一行内，前列的摆放决定后列的摆放，例如第1列放横砖，那么第2列就是横砖的后半部分；如果第1列放竖砖，那么就不影响第2列。上下两行是相关的，如果上一行是横砖，不影响下一行；如果上一行是竖砖，那么下一行的同一列是竖砖的后半部分。

用BFS搜索，从第一行第一列开始扩展到全局，每个格子的砖块有横放、竖放2种摆法，共m×n个格子，复杂度大约是O(2^{m×n})　　下面用DP解题。DP的思想是从小问题扩展到大问题，在这一题中，是否能从第一行开始，逐步扩展，直到最后一行？这一题的复杂性在于，一个砖块可能影响连续的2行，而不是1行，必须考虑连续2行的情况。

如下图所示，用一根虚线把矩形分为两半，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”。



(1)原理图　　(2)一个填充的例子　　(3)错误的填充操作



(1)原理图　(2)一个填充的例子　(3)错误的填充操作

轮廓线下面的6个阴影方格 k5k4k3k2k1k0表示当前的砖块状态，它跨越了2行。从它们推广到下一个方格x，即递推到新状态k4k3k2k1k0x 。k5k4k3k2k1k0有各种情况，用0表示没填砖块，用1表示填了砖块，有000000~111111共2^6种情况。图(2)是一个例子，其中 k3未填，k5k4k3k2k1k0= 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，砖块只能填到阴影格本身和上面的部分，不能填到下面去。在图(3)中，把k2的砖填到下面是错的。这2^6种情况，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，有三种情况：

（1） x = 0（ x不放砖块）。如果k5= 0（k5上没有砖块），由于k5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k5= 1，则x = 0可以成立。递推到 k4k3k2k1k0x=k4k3k2k1k00。

（2）x = 1（ x放竖砖），只能和k5一起放竖砖，要求 k5=0。递推到 k4k3k2k1k0x = k4k3k2k1k01。

（3） x= 1（ x放横砖），只能和k0一起放横转，要求k0=0，另外还应有 k5= 1。递推到k4k3k2k1k0x = k4k3k2k111。

经过上述讨论，对n行×m列的矩阵，可以得到状态定义和状态转移方程。

动态规划基本模型——状态转移方程

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个方格的最后一个方格是就是第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）。时间复杂度是O(m×n×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，状态定义改为dp[2][k]。

状态转移方程。根据前面分析的三种情况，分别转移到新的状态。

状态转移方程。根据前面分析的三种情况，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：

```
dp[now][(k<<1) & (~(1<<m))] += dp[old][k];
```

其中 ~(1<<m) 的意思是原来的$k_5 = 1$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：

```
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：

```
dp[now][((k<<1) | 3) & (~(1<<m))] += dp[old][k];
```

其中 (k<<1) | 3 的意思是末尾置11；~(1<<m)是原来的$k_5 = 1$移到了第m+1位，把它置0。

动态规划基本模型——单调队列优化DP

例.老徐真人秀

- 最美杭城人老徐参加一个真人秀，节目共录制N天，每一天节目组都提供同一种美味的苹果若干，老徐有个怪癖，每一天只会吃天数不超过$M\left(1 \le M \le N \le 10^6\right)$天的最美味的苹果。
- 老徐是编程达人，决定通过程序来快速选择。请老徐回答当时是怎么做到的?
- 老徐是大师，当然不会直接告诉你答案，当时随手扔过来下面这个：
- 比如，$N=5$，$M=4$，5天提供的苹果美味度分别为$(80,75,78,73,79)$，老徐的做法如下：

一.单调队列及应用

•五大要点：

· 区 最 ：

如上例中，设a_i表示第i天发放苹果的美味度，$f(i)$表示老徐第i天选择的苹果的美味度，

$f(i)=\max\{a_j\}(\max\{1,i-M+1\}\le j\le i)$

· 区 出 平移：

如上面例子中，区间左边界$l_i=\max\{1,i-M+1\}$，右边界$r_i=i$。随着i的增加，右边界r_i递增，左边界l_i非递减，当$i>M$时，l_i递增。查询区间是随着i右移向右平移的。

· 去除 · 余状 ：

如上例中，区间中两个元素a_j与a_k，如果满足$k>j$且$a_k\ge a_j$，a_k跟a_j比“既新鲜又美味”，a_j没有存在的必要，可以直接删除。

· 保持 列 ：

由③得队列中保留的元素是单调的，如例中是单调减的。

· 最 在 首：

如上例中维护单调减的队列，队列中的最大值在队首。

单调队列

队列中各个元素之间的关系是单调的(单调递增或递减）;在队首删除、队尾即可插入又可删除;

用于求解某个范围的最大值或最小值问题。

算法步骤：

一、把第一个元素进队，队头队尾指针指向第一个元素

二、对于其它的N-1个未入队的元素执行如下操作：

1、如果队头元素超过指定的区间范围，队头元素出队

2、当队尾指向的元素的值小于当前要入队的元素值，队尾元素出队

3、当前要入队的新元素入队

4、当前最优值为队头元素的值

代码实现方式：

1、STL中的双端队列（deque）

2、数组模拟

一.单调队列及应用

```
head=1; tail=1; que[head]=1; //第一个数进队
for(i=2;i<=n;i++)     //枚举其它N-1个元素
{     if(i-que[head]>=M) head++; //(1)
    //队首元素超过区间大小限制，则队首元素出队列。
     while(head<=tail)&&(a[q[tail]]<a[i])    tail--; //(2)
     //当队列非空且队尾元素小于未进队的数，队尾元素出队
     tail++;    q[tail]:=i;    //(3)
      //新元素入队
     cout<<a[q[head]]); //(4)
}
```

例1.[NOIP2010初赛]烽火传递

- **目·述：**

烽火台又称烽燧，是重要的军事防御设施，一般建在险要或交通要道上。一旦有敌情发生，白天燃烧柴草，通过浓烟表达信息；夜晚燃烧干柴，以火光传递军情，在某两座城市之间有N个烽火台，每个烽火台发出信号都有一定代价。为了使情报准确地传递，在连续M个烽火台中至少要有一个发出信号。

请计算总共最少花费多少代价，才能使敌军来袭之时，情报能在这两座城市之间准确出传递。

- **入格式：**

第一行：两个整数N,M。其中N表示烽火台的个数，M表示在连续M个烽火台中至少要有一个发出信号。

接下来N行，每行一个数W_i，表示第i个烽火台发出信号所需代价。

- **出格式：**

一行，表示答案。

- **例　入：**

5 3
1
2
5
6
2

- **例　出：**

4

- **数· ·　：**

对于50%的数据，$M \le N \le 1000$;

对于100%的数据，$M \le N \le 100{,}000, W_i \le 100$。

例1.[NOIP2010初赛]烽火传递

- **分・：**
- 动态规划：状态$f(i)$表示“在前i个烽火台传递情报且第i个烽火台一定发出信号”所需最小代价。
- 通过考虑“前一个烽火台的位置j”得到以下状态转移方程：

$$f(i)=\begin{cases} w_i & i \le M\text{时} \\ \min\{f(j)\}+w_i(i-M\le j\le i-1) & i>M\text{时} \end{cases}$$

- 答案$Ans=\min\{f(i)\}(\max\{N+1-M,1\}\le i\le N)$
- 上式计算$f(i)$时要计算区间最小值，而且区间是向右平移的，如果$f(k)\le f(j)$且$k>j$，可以删除$f(j)$，所以队列中的元素保持单调递增，最优决策在队首。
- 使用单调队列优化，时间复杂度为$O(N)$。

例2.修剪草坪

【题目描述】
在一年前赢得了小镇的最佳草坪比赛后，FJ 变得很懒，再也没有修剪过草坪。现在，新一轮的最佳草坪比赛又开始了，FJ 希望能够再次夺冠。 然而，FJ 的草坪非常脏乱，因此，FJ 只能够让他的奶牛来完成这项工作。FJ 有 N 只排成一排的奶牛，编号为 1 到 N。每只奶牛的效率是不同的，奶牛i的效率为Ei 。 靠近的奶牛们很熟悉，如果 FJ 安排超过 K 只连续的奶牛，那么这些奶牛就会罢工去开派对。因此，现在 FJ 需要你的帮助，计算 FJ 可以得到的最大效率，并且该方案中没有连续的超过K只奶牛。
【输入】
第一行：空格隔开的两个整数 N和 K； 第二到 N+1 行：第 i+1 行有一个整数 Ei 。
【输出】
一行一个值，表示 FJ 可以得到的最大的效率值。
【输入样例】
5 2
1
2
3
4
5
【输出样例】
12

例2.修剪草坪

分析：

设 dp[i][0] 表示以 i 结尾并且 i 不选所得的最大效率值

dp[i][1]表示以 i 结尾并且 i 要选所得的最大效率值

S[i]表示效率值的前缀和

转移方程为：

$dp[i][0]=\max(dp[i-1][0],dp[i-1][1])$

$dp[i][1]=\max(dp[j][0]+S[i]-S[j]);(i-K\le j<i)$

可以变形为$dp[i][1]=\max(dp[j][0]-S[j])+S[i];(i-K\le j<i)$

因此dp[i][1]可以用单调队列优化，维护队首为dp[i][0]-S[i]最大的单调队列

例3.背包问题——01背包

每种物品都有一个价值w和体积c.
你现在有一个背包容积为V,你想用一些物品装背包使得物品总价值最大.
多种物品,每种物品只有一个.求能获得的最大总价值.

分析

如果我们不选择第i件物品,那我们就相当于是用i-1件物品,填充了体积为v的背包所得到的最优解.
而我们选择第i件物品的时候,我们要得到体积为v的背包,我们需要通过填充用i-1件物品填充得到的体积为v-c[i]的背包得到体积为v的背包.

例3.背包问题——01背包

	V-c[i] 价值为A		V 哎.我好像能填充前面的得到更大价值.	

所以根据上面的分析,我们很容易设出01背包的二维状态

$f[i][v]$代表用i件物品填充为体积为v的背包得到的最大价值.

从而很容易的写出状态转移方程

$$f[i][v]=\max(f[i-1][v],f[i-1][v-c[i]]+w[i])$$

状态转移方程是如何得来的?
对于当前第ii件物品,我们需要考虑其是否能让我们得到更优解.
显然,根据上面的话
我们选择第i件物品的时候,我们要得到体积为v的背包,我们需要通过填充用i-1件物品填充得到的体积为v-c[i]的背包得到体积为v的背包.我们需要考虑到v−c[i]的情况.
当不选当前第i件物品的时候,就对应了状态转移方程中的$f[i-1][v]$,而选择的时候就对应了
$f[i-1][v-c[i]]+w[i]f[i-1][v-c[i]]+w[i]$.

例3.背包问题——01背包

考虑一维如何写!
仔细观察会发现,二维状态中,我们的状态每次都会传递给i(就是说我们的前几行会变得没用.)

我们设状态f[i]代表体积为i的时候所能得到的最大价值.

容易发现的是,我们的f[i]只会被i以前的状态影响.

代码写法↓

```
for(int i=1;i<=n;i++)//枚举 物品
        for(int j=V;j>=c[i];j--)//枚举体积
                f[j]=max(f[j],f[j-c[i]]+w[i]);//状态转移方程.
```

例4.背包问题——完全背包

每种物品都有一个价值w和体积c.
你现在有一个背包容积为V,你想用一些物品装背包使得物品总价值最大.

此类背包问题中,我们的每种物品有无限多个,可重复选取.
类似于01背包,我们依旧需要考虑前i-1件物品的影响.

分析

此时我们依旧可以设得二维状态

f[i][v]代表用i件物品填充为体积为v的背包得到的最大价值

依旧很容易写出状态转移方程

$$f[i][v]=\max(f[i-1][v],f[i-1][j-k*c[i]]+k*w[i])$$

//其中k是我们需要枚举的物品件数.而我们最多选取$\lfloor V/c[i] \rfloor$个

例4.背包问题——完全背包

同样地,我们去考虑一维状态实现
依旧设
f[i]代表体积为i的时候所能得到的最大价值

与01背包不同的是,我们可以重复选取同一件物品.

此时,我们就需要考虑到前面i-1件物品中是否有已经选取过(其实没必要

即,我们当前选取的物品,可能之前已经选取过.我们需要考虑之前物品对答案的贡献.

因此我们需要顺序枚举.

与01背包一维的写法类似.

```
for(int i=1;i<=n;i++)//枚举物品
        for(int j=c[i];j<=V;j++)//枚举体积.注意这里是顺序/
                f[j]=max(f[j,f[j-c[i]]]+w[i]);//状态转移.
```

例4.背包问题——多重背包

每种物品都有一个价值w和体积c.
你现在有一个背包容积为V,你想用一些物品装背包使得物品总价值最大.

此类问题与前两种背包问题不同的是,
这里的物品是有个数限制的.
(下面用num[i]表示物品i的个数.)
分析：
同样,我们最多可以放⌊V/c[i]⌋,但我们的物品数量可能不够这么多.
因此,我们枚举的物品个数是min(⌊V/c[i] ⌋,num[i])
多个物品,我们就可以看成为一个大的物品,再去跑01背包即可.
因此这个大物品的价值为k×w[i],体积为k×c[i]

```
for(int i=1;i<=n;i++)//枚举物品
        for(int j=V;j>=0;j--)//枚举体积
                for(int k=1;k<=num[i],k++)
                        //这个枚举到num[i]更省心
                         if(j-k*c[i]>=0)//判断能否装下.
                                 f[j]=max(f[j],f[j-k*c[i]]+k*w[i]);
```

中国计算机学会 China Computer Federation

例4.背包问题——多重背包的二进制拆分优化

二进制拆分的原理
我们可以用 1,2,4,8...2^n表示出11到 2^(n+1)-1的所有数.

考虑我们的二进制表示一个数。

根据等比数列求和,我们很容易知道我们得到的数最大就是2^(n+1)-1

而我们某一个数用二进制来表示的话,每一位上代表的数都是2的次幂.

就连奇数也可以,例如->19可以表示为10011(2)

二进制拆分的做法
因为我们的二进制表示法可以表示从1到num[i]的所有数,我们对其进行拆分,就得到好多个大物品(这里的大物品代表多个这样的物品打包得到的一个大物品).
(简单来讲,我们可以用一个大物品代表1,2,4,8..件物品的和。)
而这些大物品又可以根据上面的原理表示出其他不是2的次幂的物品的和.
因此这样的做法是可行的.
我们又得到了多个大物品,所以再去跑01背包即可.

例4.背包问题——多重背包的二进制拆分优化

```
for(int i=1;i<=n;i++)
{
        for(int j=1;j<=num[i];j<<=1)
  //二进制每一位枚举.
  //注意要从小到大拆分
        {
                num[i]-=j;//减去拆分出来的
                new_c[++tot]=j*c[i];//合成一个大的物品的体积
                new_w[tot]=j*w[i];//合成一个大的物品的价值
        }
        if(num[i])//判断是否会有余下的部分.
  //就好像我们某一件物品为13,显然拆成二进制为1,2,4.
  //我们余出来的部分为6,所以需要再来一份.
        {
                new_c[++tot]=num[i]*c[i];
                new_w[tot]=num[i]*w[i];
                num[i]=0;
        }
}
```

时间复杂度分析

我们拆分一种物品的时间复杂度为log(num[i]).

我们总共会有n种物品,再配上枚举体积的时间复杂度.

因此,二进制拆分做法的时间复杂度为

$$O(\sum_{i=1}^{n}\log(\mathrm{num[i]})\times \mathrm{V})$$

例4.背包问题——多重背包的单调队列优化

首先回想多重背包最普通的状态转移方程

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

其中$k\in[1,\min(\lfloor V/c[i]\rfloor,num[i])]$

下面用lim表示$\min(\lfloor V/c[i]\rfloor,num[i])$

容易发现的是$f[i][j-k*c[i]]$会被$f[i][j-(k+1)*c[i]]$影响 (很明显吧

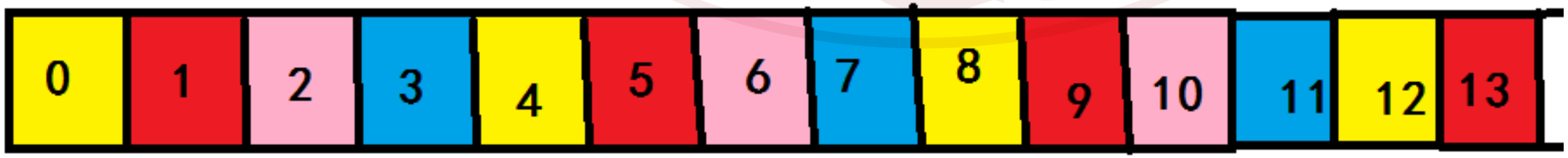
(我们通过一件体积为c[i]的物品填充体积为$j-(k+1)*c[i]$的背包,会得到体积为$j-k*c[i]$的背包.)

归纳来看的话

$f[i][j]$将会影响 $f[i][j+k*c[i]]$ $(j+k*c[i]<=V)$

栗子

c[i]=4

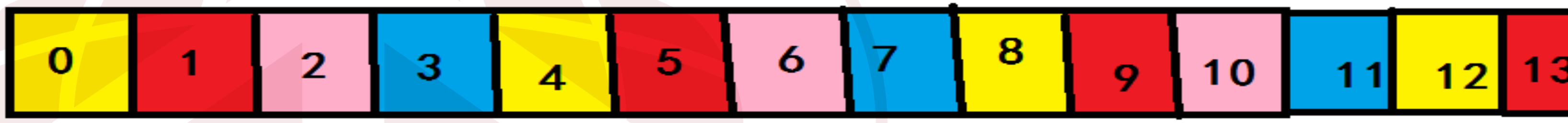


例4.背包问题——多重背包的单调队列优化

容易发现的是,同一颜色的格子,对c[i]取模得到的余数相同.
且,它们的差满足等差数列! (公差为c[i].)
通项公式为j=k*c[i]+取模得到的余数

所以我们可以根据对c[i]取模得到的余数进行分组.

即可分为1,2,3…c[i]−1 共c[i]组



且每组之间的状态转移不互相影响.(注意这里是组.相同颜色为一组)

相同颜色的格子,位置靠后的格子,将受到位置靠前格子的影响.

//但是这样的话,我们的格子会重复受到影响.
即f[9]可能受到f[5]的影响,也可能受到f[1]的影响

而f[5]也可能受到f[1]的影响.

所以我们考虑将原始状态转移方程变形.

例4.背包问题——多重背包的单调队列优化

令d=c[i],a=j/c[i],b=j%c[i]
其中a为全选状况下的物品个数.则j=a∗d+b
则代入原始的状态转移方程中
j−k∗d=a∗d+b−k∗d =(a−k)∗d+b
我们令(a−k)=k′
再回想我们最原始的状态转移方程中第二状态：f[i][j−k∗c[i]]+k∗w[i] 代表选择k个当前i物品.
根据容斥 ：全选−−不选=选.
因此 a−(a−k)=k而前面我们已经令(a−k)=k′
而我们要求的状态也就变成了
f[i][j]=max(f[i−1][k′∗d+b]+a∗w[i]−k′∗w[i])
而其中,我们的a∗w[i]为一个常量
所以我们的要求的状态就变成了
f[i][j]=max(f[i−1][k′∗d+b]−k′∗w[i])+a∗w[i]
根据我们的k∈[1,lim]容易推知k′∈[a−k,a]
那么当前的f[i][j]求解的就是为lim+1个数对应的f[i−1][k′∗d+b]−k′∗w[i]的最大值.
(之所以为lim+1个数,是包括当前这个j.)
将f[i][j]前面所有的f[i−1][k′∗d+b]−k′∗w[i]放入一个队列.
那我们的问题就是求这个最长为lim+1的队列的最大值+a∗w[i].

例4.背包问题——多重背包的单调队列优化

```
for(int i=1;i<=n;i++)//枚举物品种类
{
        cin>>c[i]>>w[i]>>num[i];//c,w,num分别对应 体积,价值,个数
        if(V/c[i] <num[i]) num[i]=V/c[i];//求lim
        for(int mo=0;mo<c[i];mo++)//枚举余数
        {
                head=tail=0;//队列初始化
                for(int k=0;k<=(V-mo)/c[i];k++)
                {
                        int x=k;
                        int y=f[k*c[i]+mo]-k*w[i];
                        while(head<tail && que[head].pos<k-num)head++;//限制长度
                        while(head<tail && que[tail-1].value<=y)tail--;
                        que[tail].value=y,que[tail].pos=x;
                        tail++;
                        f[k*c[i]+mo]=que[head].value+k*w[i];
      //加上k*w[i]的原因:
      //我们的单调队列维护的是前i-1种的状态最大值.
      //因此这里加上k*w[i].
                }
        }
}
```

斜率优化DP

有一类DP状态方程，例如：

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

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 i和 j循环，复杂度是O(n^2)

通过斜率优化（英文convex hull trick，凸壳优化），把时间复杂度优化到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DP

1. 把状态方程变换为平面的斜率问题

方程对某个固定的i，求 j变化时dp[i]的最优值，所以可以把关于i的部分看成固定值，把关于j的部分看成变量。把 min去掉，方程转化为：

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]，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。其中：

（1）变量 x、 y和 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]，也就是状态方程的解。

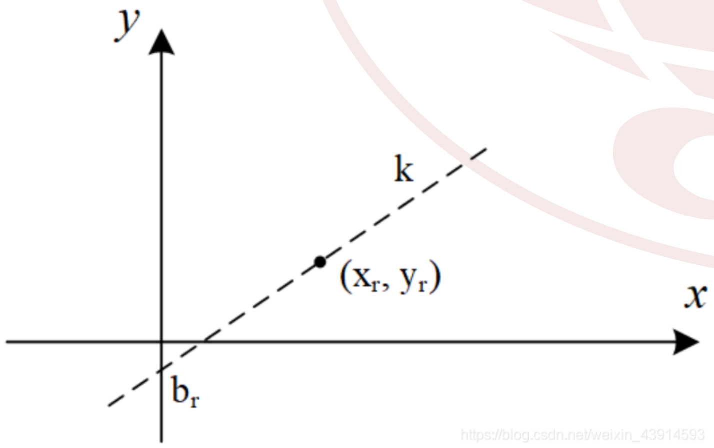
注意应用斜率优化的2个条件： x和 k是单调增加的，即x随着j递增而递增， k随着i递增而递增。

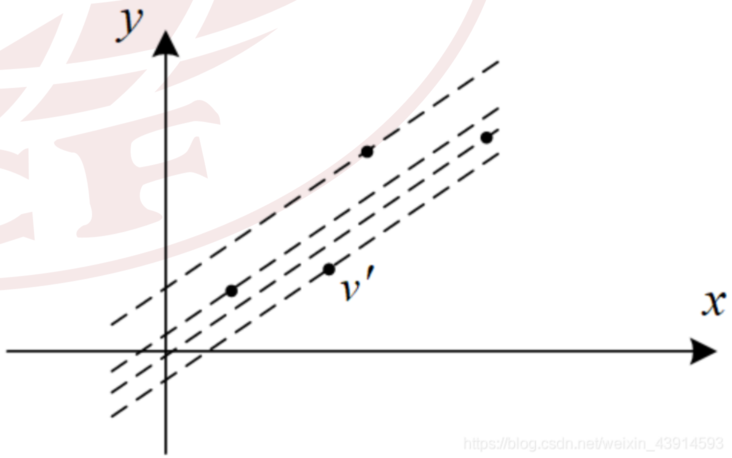
斜率优化DP

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]可以看成常数。当j在0≤j<i内变化时，对某个jr，产生一个点vr=(xr, yr)，这个点在一条直线y = kx + br上， br 是截距。如图。

对于0≤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'，算出 b'，由于b'中包含dp[i]，那么就算出了最优的dp[i]。如图。



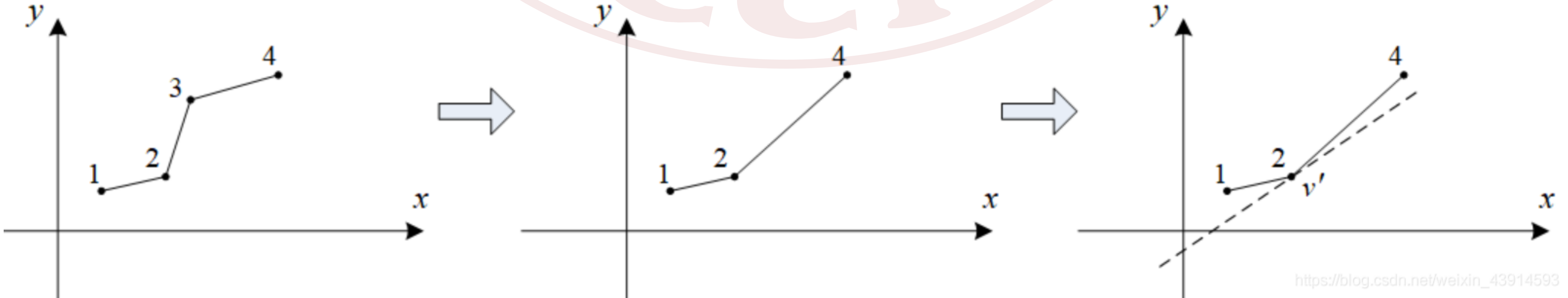


斜率优化DP

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。图中给出了了4个点，它们的x坐标是递增的。图（1）中的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。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2或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(2)，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(3)中，用斜率为k的直线来切这些点，设线段12的斜率小于k，24的斜率大于k，那么点2就是“下凸壳”的最优点。

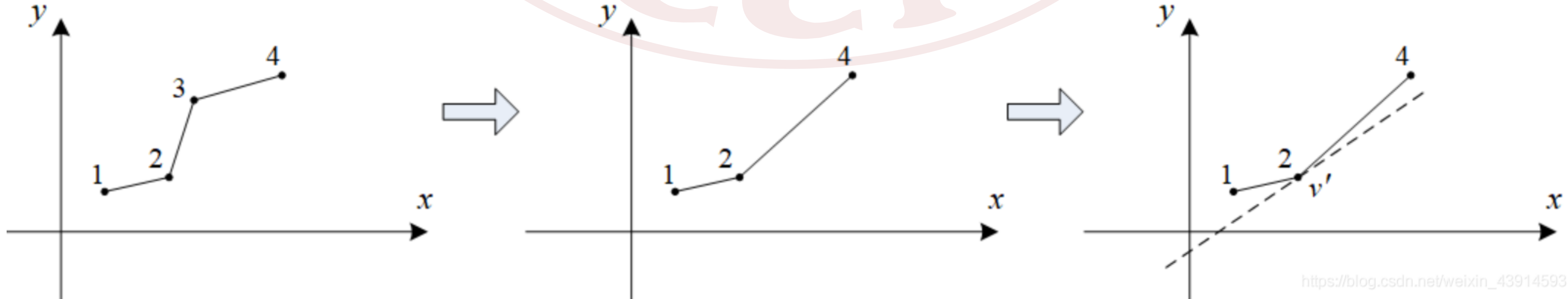


斜率优化DP

以上操作用单调队列编程很方便。

（1）进队操作，在队列内维护一个“下凸壳”，即每2个连续点组成的直线，其斜率是单调上升的。新的点进队列时，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2个点是v1、v2，准备加入队列的新的点是 v3 。比较v1、 v2、 v3，看线段v1v2和 v2v3的斜率是否递增，如果是，那么 v1、v2、v3 形成了“下凸壳”；如果斜率不递增，说明v2不对，从队尾弹走它；然后继续比较队列尾部的2个点和 v3；重复以上操作，直到v3能进队为止。经过以上操作，队列内的点组成了一个大的“下凸壳”，每2个点组成的直线，斜率递增，队列保持为单调队列。

（2）出队列，找到最优点。设队头的2个点是v1、 v2，如果线段 v1v2的斜率比 k小，说明v1不是最优点，弹走它，继续比较队头新的2个点，一直到斜率大于k为止，此时队头的点就是最优点v ′。



斜率优化DP

3. 求所有的dp[i]

以上求得了一个dp[i]，复杂度O(n)。如果对所有的i，每一个都这样求dp[i]，总复杂度仍然是O(n^2)的，并没有改变计算的复杂度。有优化的方法吗？

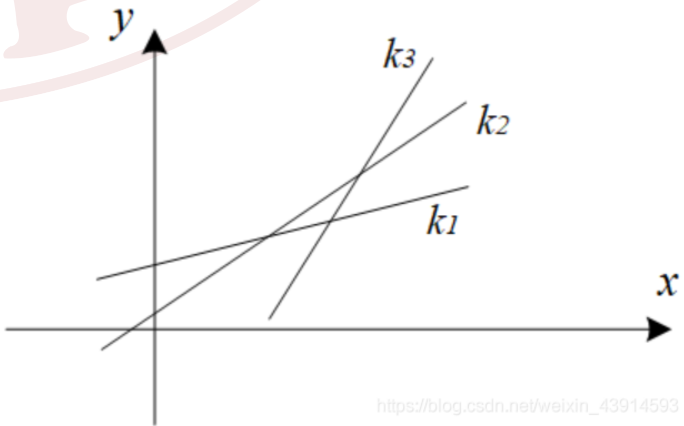
一个较小的 i1，它对应的点是{ v0，v1，. . . ，vi1}；一个较大的i2，对应了更多的点{v0, v1，. . . . ，vi1，. . . ，vi2 }，其中包含了i1的所有点。当寻找 i1的最优点时，需要检查{ v0，v1，. . . . ，vi1}；寻找i2的最优点时，需要检查{ v0，v1，. . . . ，vi1，. . . ，vi2}。这里做了重复的检查，并且这些重复是可以避免的。这就是能优化的地方，仍然用“下凸壳”进行优化。

（1）每一个i所对应的斜率 ki=a[i]是不同的，根据约束条件a[i]≤ a[i+1]，当i增大时，斜率递增。

（2）前面已经提到，对一个i1找它的最优点的时候，可以去掉一些点，即那些斜率比 ki小的点。这些被去掉的点，在后面更大的i2时，由于斜率ki2也更大，肯定也要被去掉。

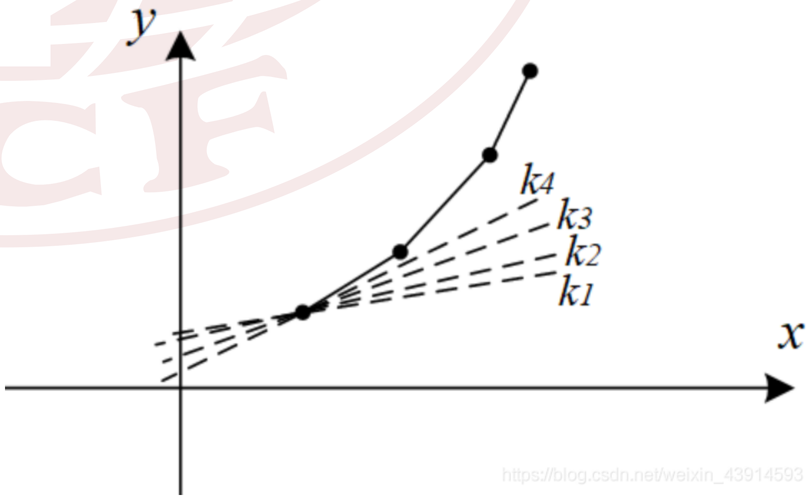
根据（1）和（2）的讨论，优化方法是：对所有的i，统一用一个单调队列处理所有的点；被较小的i1 去掉的点，被单调队列弹走，后面更大的i2不再处理它们。

因为每个点只进入一次单调队列，总复杂度O(n)。



斜率优化DP


```
//q[]是单调队列，head指向队首，tail指向队尾，slope()计算2个点组成的直线的斜率
for(int i=1;i<=n;i++){
    while(head<tail && slope(q[head],q[head+1])<k)  //队头的2个点斜率小于k
        head++;                                      //不合格，从队头弹出
    int j = q[head];    //队头是最优点
    dp[i] = ...;        //计算dp[i]
    while(head<tail && slope(i,q[tail-1])<slope(q[tail-1],q[tail]))   //进队操作
        tail--;         //弹走队尾不合格的点
    q[++tail] = i;      //新的点进队列
}
```



例题1：HDU 3507 Print Article——斜率优化DP

题目描述：

打印一篇包含N个单词的文章，第i个单词的打印成本为Ci。在一行中打印$(\sum_{i=1}^{k} C_i)^2 + M$

，M是一个常数。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，第一行中都有两个数字N和M（0≤n≤500000，0≤M≤1000）。然后，在接下来的2到N + 1行中有N个数字。输入用EOF终止。

输出：一个数字，表示打印文章的最低费用。

样例输入：

5 5

5

9

5

7

5

样例输出：

230

例题1：HDU 3507 Print Article——斜率优化DP

中国计算机学会 China Computer Federation

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，每一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转移方程：

dp[i] = min{dp[j] + (sum[i]-sum[j])^2 + M}　其中sum[i]表示前i个数字和。

下面把DP方程改写为y = kx + b的形式。首先展开方程：

dp[i] = dp[j] + sum[i]*sum[i] + sum[j]*sum[j] - 2*sum[i]*sum[j] + M

移项得：

dp[j] + sum[j]*sum[j] = 2*sum[i]*sum[j] + dp[i]-sum[i]*sum[i] - M

对照y = kx + b，有：

y = dp[j] + sum[j]*sum[j]， y只和 j有关。

x = 2*sum[j]， x只和j有关，且随着 j递增而递增。

k = sum[i]， k只和 i有关，且随着 i递增而递增。

b = dp[i] - sum[i]*sum[i] - M， b只和i有关，且包含dp[i]。

例题2：HNOI2008 玩具装箱——斜率优化DP

P教授要去看奥运，但是他舍不下他的玩具，于是他决定把所有的玩具运到北京。他使用自己的压缩器进行压缩，其可以将任意物品变成一堆，再放到一种特殊的一维容器中。P教授有编号为1...N的N件玩具，第i件玩具经过压缩后变成一维长度为Ci.为了方便整理，P教授要求在一个一维容器中的玩具编号是连续的。同时如果一个一维容器中有多个玩具，那么两件玩具之间要加入一个单位长度的填充物，形式地说如果将第i件玩具到第j个玩具放到一个容器中，那么容器的长度将为 x=j-i+Sigma(Ck) i<=K<=j 制作容器的费用与容器的长度有关，根据教授研究，如果容器长度为x,其制作费用为(X-L)^2.其中L是一个常量。P教授不关心容器的数目，他可以制作出任意长度的容器，甚至超过L。但他希望费用最小.

输入　　第一行输入两个整数N，L.接下来N行输入Ci.1<=N<=50000,1<=L,Ci<=10^7

输出　　输出最小费用

Sample Input

5 4

3

4

2

1

4

Sample Output

1

例题2：HNOI2008 玩具装箱——斜率优化DP

DP:

dp[i] = min(dp[j] + (sum[i] - sum[j - 1] + i - j - L)^2)

可以变成

dp[i] = min(dp[j] + (sum[i] - sum[j] + i - j - L - 1)^2)

令 a[i] = sum[i] + i, b[i]=sum[i]+i+L+1

dp[i] = min(dp[j] + (a[i]-b[j])^2)

dp[i] = dp[j] + a[i]^2+b[j]^2-2·a[i]·b[j]

dp[j]+b[j]^2 =2*a[i]*b[j]+dp[i]−a[i]^2

把b[j]看做 x，dp[j]+b[j]^2看做y

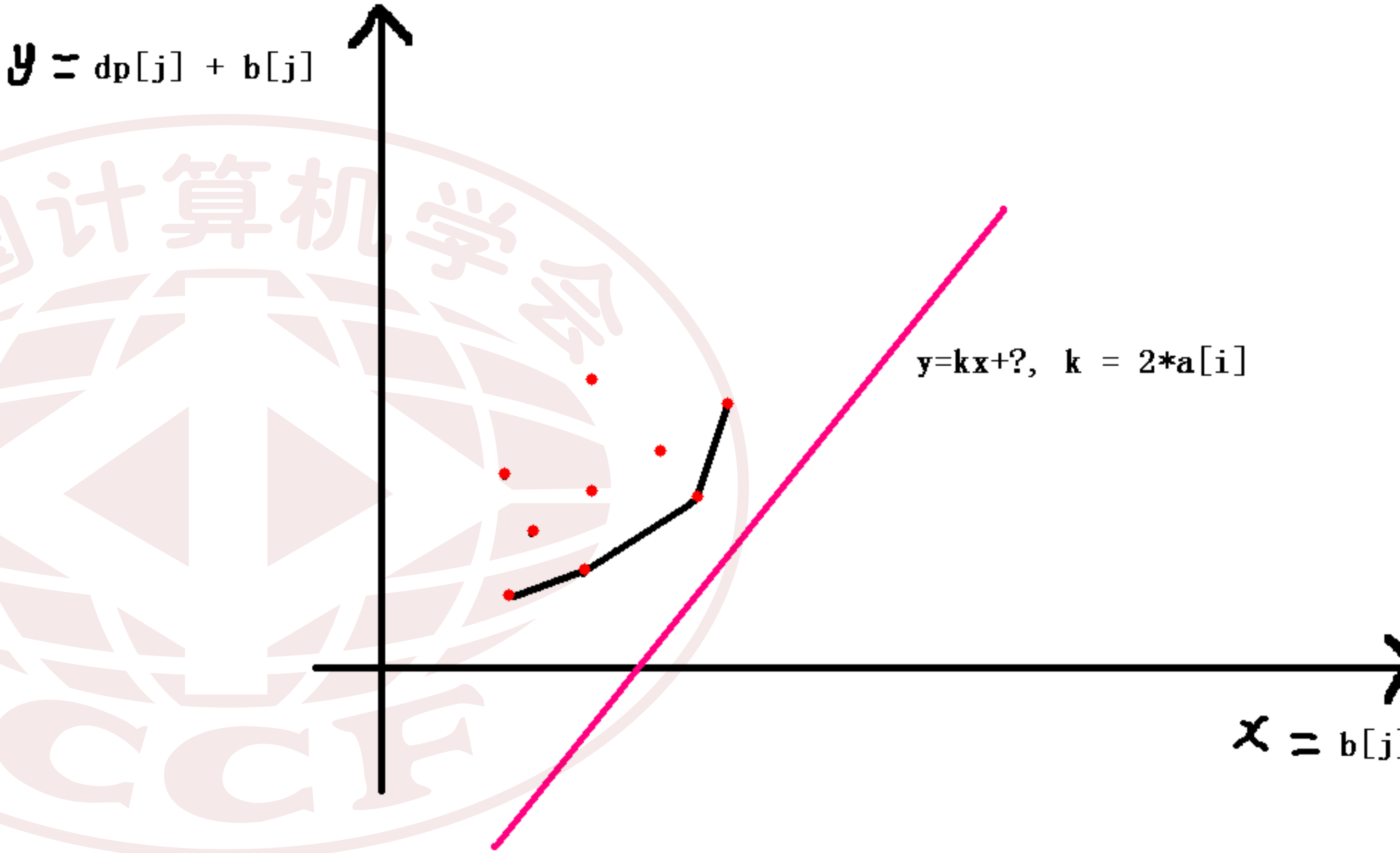
y=2·a[i]x+dp[i]-a[i]^2

这就是一条直线的解析式， y=kx+b, k=2*a[i], b=dp[i]-a[i]^2

要找一个点 P(b[j],dp[j]+b[j]^2) 使得上面的斜率为2*a[i]的直线过这个点且与y轴截距 (b=dp[i]-a[i]^2) 最小

例题2：HNOI2008 玩具装箱——斜率优化DP

就变成了用一条斜率为k=2*a[i] 的直线从
下到上扫过去，
碰到的第一个点然后把直线和y轴的截距求出来加上a[i]^2就是答案



Thanks!