

P5(part 2),P6,P7, OJ4~6

2024/6/13

## 11.1[兑换硬币]

- 直觉，优先尽量换面额最大的，换不干净再选面额第二大的，以此类推；
- 最终答案就是硬币金额 $S$ 的 $c$ 进制位串中，所有位串数字的和；
- 具体而言，证明最优方案一定是 $S$ 的 $c$ 进制分解结果，先说明：除了最高位，每一位必小于 $c$ . 能满足该性质只可能是 $c$ 进制的表示.

## 11.3[设立基站]

- 算法思路：
  - 从左往右选择基站，第一个基站 $b_1$ 选择建在 $x_1$ 右边 $t$ 距离的位置；
  - 假设 $b_1$ 能覆盖的房子范围为 $\{x_1, x_2, \dots, x_{k_1}\}$ ，则第二个基站 $b_2$ 选择建在 $x_{k_1+1}$ 右边 $t$ 距离的位置；
  - 重复上述过程，直到所有房子都能被至少一个基站覆盖；
- 算法正确性证明：
  - 同Prim理，对算法的步骤数 $k$ 进行归纳
  - 假设对于任意正整数 $k$ ，存在最优解集包含上述算法前 $k$ 步选择的基站位置；

也即：假设按照上述算法设立 $k$ 个基站，能够覆盖（从左至右）最多的点. 然后归纳 $k+1$ .

## 12.1[记录Floyd最短路的前驱后继]

1)

- 初始化时:  $GO[i][j] = j$ ;
- 更新时:  $GO[i][j] = GO[i][k]$ ;

2)

- 初始化时:  $FROM[i][j] = i$ ;
- 更新时:  $FROM[i][j] = FROM[k][j]$ ;

理解Floyd的工作原理  $dis_k[i][j] = \min(dis_{k-1}[i][j], dis_{k-1}[i][k] + dis_{k-1}[k][j])$

## 12.2[点对间最大瓶颈路] (最大化两点路径上的最小边)

1) 从源点 $s$ 到终点 $t$ 的最大吞吐率是 $s$ 到 $t$ 所有路径上，最小管道吞吐率最大的那条，所对应的最小管道吞吐率；

- 只需要修改Dijkstra的更新步骤:  $cap(s, v) = \max\{cap(s, v), f(cap(s, u), c(u, v))\}, e = (u, v);$
- 其中 $f$ 的具体形式为:  $f = \min\{cap(s, u), c(u, v)\};$

2) 同理，只需要修改Floyd-Warshall算法的更新步骤:

- $D^{(k)}[i][j] = \max\{D^{(k-1)}[i][j], f(D^{(k-1)}[i][k], D^{(k-1)}[k][j])\};$
- 其中 $f$ 的具体形式为:  $f = \min\{D[i][k], D[k][j]\};$

## 12.2[点对间最大瓶颈路] (最大化两点路径上的最小边)

- 1) 从源点 $s$ 到终点 $t$ 的最大吞吐率是 $s$ 到 $t$ 所有路径上，最小管道吞吐率最大的那条，所对应的最小管道吞吐率；
- 只需要修改Dijkstra的更新步骤： $cap(s, v) = \max\{cap(s, v), f(cap(s, u), c(u, v))\}$ ,  $e = (u, v)$ ;
  - 其中 $f$ 的具体形式为： $f = \min\{cap(s, u), c(u, v)\}$ ;

其中(1)经过优化可以做到 $O((n + m) \log n)$ , (2)需要 $O(n^3)$

(2)也可以以每一个点为源点跑Dijkstra, 优化后 $O(n(n + m) \log n)$

也可以使用最大瓶颈生成树的思路：两个点间最大的瓶颈边一定在最大生成树上；

这时候先求出MST, 然后点对查询, 需要 $O(m \log m + n^2 \cdot n) = O(n^3)$

! 优化查询过程可以做到 $O(n^2 \log n)$  (e.g.倍增查询 or 链剖分)

## 12.4[点集间的最短路]

- 添加超级节点 $s$ , 其仅与 $S$ 中的所有节点相邻, 且权值均为0;
- 添加超级节点 $t$ , 其仅与 $T$ 中的所有节点相邻, 且权值均为0;
- 以 $s$ 为源点, 在增广的图 $G'$ 上运行Dijkstra, 答案就是 $\text{dist}_s[t]$ ;

## 12.7[经过特定某点的最短路]

- 以 $v_0$ 为源点，在原图 $G$ 上运行Dijkstra算法，得到 $\text{dist}_{v_0}[t]$ ；
- 以 $v_0$ 为源点，在转置图 $G^T$ 上运行Dijkstra算法，得到 $\text{dist}_t[v_0]$ ；
- 因此，任意节点对 $(u, w)$ 之间的最短路径为： $\text{dist}[u][w] = f(\text{dist}_{v_0}[u], \text{dist}_{v_0}[w], \text{dist}_u[v_0], \text{dist}_w[v_0])$ ；
- 其中 $f$ 的具体形式为： $f = \text{dist}_u[v_0] + \text{dist}_{v_0}[w]$ ；

# 13.1: (最大相容任务集合)

注意理解贪心算法和动态规划的异同

- 初始话:

```
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to n do
            if f[i] <= s[k] < f[k] <= s[j] then
                S[i, j].add(k)
```

- 状态转移: (注意状态转移顺序: 行序从下往上, 列序从左往右)

```
for i := n downto 1 do
    for j := 1 to n do
        if S[i, j].empty() then
            c[i, j] = 0
        else
            for k in S[i, j] do
                c[i][j] = max(c[i][j], c[i][k]+c[k][j]+1)
```

- 时间复杂度:  $O(n^3)$ ; 空间复杂度:  $O(n^3)$

## 13.2

**13.2（整数子集合问题）** 给定一个自然数集合  $A = \{s_1, s_2, \dots, s_n\}$  和自然数  $S$ ，要判断是否存在  $A$  的子集，其中元素的和恰好是  $S$ 。请设计一个动态规划算法来解决该问题。

类比背包问题。

设计  $f_{i,j}$  表示当考虑前  $i$  个数时，和能不能凑到  $j$  ( $j \leq S$ )，用 bool 值来表示。

转移方程： $f_{i,j} = f_{i-1,j} \vee f_{i-1,j-a_i}$

初始化  $f_{0,0} = \text{true}$ 。按  $n, S$  两维枚举，最后答案是  $f_{n,S}$ 。可以把数组第一维优化掉，做成滚动数组。

空间  $O(n)$  时间  $O(nS)$

## 13.4[LIS(Longest increasing subsequence)问题]

设计 $f_i$ 表示以*i*结尾的最长LIS (当然本题是非严格递增) 长度。

$$\text{于是 } f_i = \max_{j < i, A_j \leq A_i} \{f_j\} + 1$$

枚举*i, j*( $j < i$ )两维，初始化*f*全零，复杂度 $O(n^2)$ 。

! LIS可以优化到 $O(n \log n)$ . 把上述找 $A_j \leq A_i$ 且 $j < i$ 的枚举过程改成在范围 $[1, A_i - 1]$ 内维护*f*最大值，再把 $A_i$ —— $f_i$ 加入维护，可以先对*A*数组离散化，再用树状数组优化枚举变成 $O(\log n)$ 。

! 另一种 $O(n \log n)$ 方法是，维护 $d[i]$ 表示LIS长度是*i*时序列结尾最小的数。

每次 $A_k$ 二分一个最大的 $j(d[j] \leq A[k])$ 来转移，然后再去 $d[f[k]]$ 中更新自己。

## 13.5: (序列的k划分)

经典划分问题，注意“连续子序列”就是子串，同时这也是一个*Minimax*优化问题

令 $dp[i, j]$ 表示前 $i$ 个数的 $j$ -划分最低代价，并预处理：求出 $X$ 的前缀和 $S[1..n]$

- 初始化： $dp[i, 1] = S[i]$ ,  $f[1, j] = S[1]$
- 考虑最后一次划分的位置，有状态转移方程：

$$dp[i, j] = \min_{k=1}^{i-1} \{ \max\{dp[k, j-1], S[i] - S[k]\} \}$$

- 时间复杂度 $O(kn^2)$ , 空间复杂度 $O(kn)$

改进：

- $dp[k, j-1]$ 是关于 $k$ 的增函数，而 $S[i] - S[k]$ 是关于 $k$ 的减函数，因此*Minimax*问题可以转化为“找函数单峰交点”问题，利用二分法，时间复杂度降到 $O(kn \log n)$
- $dp[\cdot, j]$ 只与 $dp[\cdot, j-1]$ 有关，空间可状态压缩到 $O(n)$

## 13.6: (最大乘积子数组)

注意与“最大和”的区别在于正负号

类似于“最大和子数组”，令 $dp[i]$ 表示以 $A[i]$ 结尾的子数组的最大乘积，分两种情况讨论：

1) 假设均为正数，则易得状态转移方程：

$$dp[i] = \begin{cases} \max(A[i], dp[i - 1] \times A[i]), & i > 0 \\ A[i], & i = 0 \end{cases}$$

2) 假设有正有负，则需要两个 $dp$ 数组，第一个是最大乘积数组 $dp_1$ ，第二个是最小乘积数组 $dp_2$ ，而状态转移方程为：

$$dp_1[i] = \begin{cases} \max(A[i], dp_1[i - 1] \times A[i], dp_2[i - 1] \times A[i]), & i > 0 \\ A[i], & i = 0 \end{cases}$$

$$dp_2[i] = \begin{cases} \min(A[i], dp_1[i - 1] \times A[i], dp_2[i - 1] \times A[i]), & i > 0 \\ A[i], & i = 0 \end{cases}$$

- 优化：由于状态转移方程只依赖 $dp$ 数组的相邻两项，因此可以用变量代替数组，将空间压缩为 $O(1)$

## 13.8: (公共子序列问题) - 1

dp经典的双串匹配问题

1) 令 $dp[i, j]$ 表示 $X$ 的前 $i$ 个字符组成的子串和 $Y$ 的前 $j$ 个字符组成的子串的最长公共子序列的长度

- 初始化:  $dp[i, j] = 0, i = 0 \parallel j = 0$

- 状态转移:

$$dp[i, j] = \max \{ dp[i, j - 1], dp[i - 1, j], dp[i - 1, j - 1] + I(X[i] = Y[j]) \}$$

$\Downarrow$  or

$$dp[i, j] = \begin{cases} dp[i - 1, j - 1] + 1, & X[i] = Y[j] \\ \max\{dp[i, j - 1], dp[i - 1, j]\}, & X[i] \neq Y[j] \end{cases}$$

## 13.8: (公共子序列问题) - 2

LCS问题的扩展

2) 给定两个字符串  $X$  和  $Y$ 。请给出计算最长子序列的算法，要求在最长子序列中  $X$  中的字符可以重复出现但是  $Y$  中的不可以。该问题的一个例子如图 13.5 所示。

2) 令  $dp[i, j]$  表示  $X$  的前  $i$  个字符组成的子串和  $Y$  的前  $j$  个字符组成的子串的最长公共子序列的长度，且其中  $X[1..i]$  中的字符可以重复出现

- 初始化:  $dp[i, j] = 0, i = 0 \parallel j = 0$
- 状态转移:

$$dp[i, j] = \max\{dp[i, j - 1], dp[i - 1, j], dp[i, j - 1] + I(X[i] = Y[j])\}$$

$\Downarrow$  or

$$dp[i, j] = \begin{cases} dp[i, j - 1] + 1, & X[i] = Y[j] \\ \max\{dp[i, j - 1], dp[i - 1, j]\}, & X[i] \neq Y[j] \end{cases}$$

## 13.8: (公共子序列问题) - 3

LCS 问题的扩展

3) 除  $X$ 、 $Y$  两个字符串之外, 给定一个正整数  $k$ 。问题和上一小题一样, 但是子序列中  $X$  中的字符重复出现的次数不超过  $k$ ,  $Y$  中的字符不可重复出现。例如, 假设  $X$ 、 $Y$  如图 13.5 所示,  $k = 2$ , 那么在最长子序列中  $X$  中的字符可以出现 2 次。

(为了避免误解, 用  $K$  表示题干中规定的  $X$  中字符可以出现的最大次数)

3) 令  $dp[i, j, k]$  表示  $X$  的前  $i$  个字符组成的串和  $Y$  的前  $j$  个字符组成的串的最长公共子序列的长度, 且其中  $X[i]$  中的字符重复出现的次数不超过  $k$  次 ( $1 \leq k \leq K$ ), 其余  $X[1..i]$  中的字符重复出现的次数也不超过  $K$  次

- 初始:  $dp[i, j, k] = 0$

- 状态转移:

- 若  $X[i] = Y[j]$ :

$$dp[i, j, 1] = dp[i - 1, j - 1, K] + 1$$

$$dp[i, j, k] = dp[i, j - 1, k - 1] + 1, \quad 1 < k \leq K$$

- 若  $X[i] \neq Y[j]$ :

$$dp[i, j, k] = \max\{dp[i, j - 1, k], dp[i - 1, j, K]\}, \quad 1 \leq k \leq K$$

## 13.8: (公共子序列问题) - 3

LCS 问题的扩展

3) 除  $X$ 、 $Y$  两个字符串之外, 给定一个正整数  $k$ 。问题和上一小题一样, 但是子序列中  $X$  中的字符重复出现的次数不超过  $k$ ,  $Y$  中的字符不可重复出现。例如, 假设  $X$ 、 $Y$  如图 13.5 所示,  $k = 2$ , 那么在最长子序列中  $X$  中的字符可以出现 2 次。

(为了避免误解, 用  $K$  表示题干中规定的  $X$  中字符可以出现的最大次数)

3) 令  $dp[i, j, k]$  表示  $X$  的前  $i$  个字符组成的串和  $Y$  的前  $j$  个字符组成的串的最长公共子序列的长度, 且其中  $X[i]$  中的字符重复出现的次数不超过  $k$  次 ( $1 \leq k \leq K$ ), 其余  $X[1..i]$  中的字符重复出现的次数也不超过  $K$  次

- 初始:  $dp[i, j, k] = 0$

- 状态转移:

另一种方法: 第一个串每个字符暴力复制  $k$  遍。复杂度也是一样的。

- 若  $X[i] = Y[j]$ :

$$dp[i, j, 1] = dp[i - 1, j - 1, K] + 1$$

$$dp[i, j, k] = dp[i, j - 1, k - 1] + 1, \quad 1 < k \leq K$$

- 若  $X[i] \neq Y[j]$ :

$$dp[i, j, k] = \max\{dp[i, j - 1, k], dp[i - 1, j, K]\}, \quad 1 \leq k \leq K$$

## 13.9: (最长前后向连续子串)

dp经典的单串自匹配问题

13.9 请设计一个高效的算法，找到字符串  $T[1..n]$  中前向和后向相同的最长连续子串的长度。前向和后向的子串不能够重叠。下面是几个例子：

- 给定输入字符串 ALGORITHM，你的算法返回 0。
- 给定输入字符串 RECURRSION，你的算法返回 1，子串是 R。
- 给定输入字符串 REDIVIDE，你的算法返回 3，子串是 EDI（前向和后向的字符串不能够重叠）。

令  $dp[i, j]$  表示以  $T[i]$  开头和以  $T[j]$  结尾的两个相同连续字串的长度

- 初始话：  $dp[i, j] = 0$
- 状态转移：  $dp[i, j] = dp[i + 1, j - 1] + 1, T[i] = T[j]$
- 最终答案为：  $\max(dp[i, j])$
- 注意：
  - 状态转移顺序：  $i : n \rightarrow 1, j : 1 \rightarrow n$
  - 不能直接用  $T[i..j]$  作为子问题

## 13.10

13.10 令  $A[1..m]$  和  $B[1..n]$  是两个任意的序列。 $A$ 、 $B$  的公共超序列（supersequence）是一个序列，它包含  $A$  和  $B$  为其子序列。请设计一个高效算法找到  $A$ 、 $B$  的最短公共超序列。

设  $f_{i,j}$  表示  $A$  串前  $i$  个， $B$  串前  $j$  个字符的 LCSS。

$$f_{i,j} = \begin{cases} \min(f_{i-1,j}, f_{i,j-1}) + 1 & (A_i \neq B_j) \\ f_{i-1,j-1} + 1 & (A_i = B_j) \end{cases}$$

初始边界  $f_{i,0} = i$ ,  $f_{0,j} = j$ 。  $O(n^2)$ 。

长度分别为  $|X| = m$ 、 $|Y| = n$ 、 $|Z| = k$ ，现在的问题是判断序列  $X$  和  $Y$  是否可以合并为一个新的序列  $Z$ ，并且不改变其中任何一个序列中元素的相对顺序。例如， $X = \langle ABC \rangle$ 、 $Y = \langle BACA \rangle$

## 13.11: (最长公共子序列变体)

注意分析完整三个字符串之间的各种匹配关系

1) 不正确，因为删去的可能是  $Y$  中的字符，反例如下：

$$X = \langle ABC \rangle, Y = \langle BACA \rangle, Z = \langle A_X B_Y A_Y C_Y B_X A_Y C_X \rangle, Z' = \langle ABAC \rangle$$

2) 令  $dp[i, j]$  表示  $X[1..i]$  和  $Y[1..j]$  能否合成为  $Z[1..i+j]$ ，则有状态转移方程：

$$dp[i, j] = (dp[i-1, j] \wedge Z[i+j] = X[i]) \quad || \quad (dp[i, j-1] \wedge Z[i+j] = Y[j])$$

3) 令  $dp[i, j, k]$  为  $X[1..i]$  和  $Y[1..j]$  合成为  $Z[1..k]$  需要删除的最小元素集合，则有状态转移方程：

$$dp[i, j, k] = \min \begin{cases} dp[i-1, j, k-1], & Z[k] = X[i] \\ dp[i, j-1, k-1], & Z[k] = Y[j] \\ dp[i-1, j, k].add(X[i]), \\ dp[i, j-1, k].add(Y[j]), \\ dp[i, j, k-1].add(Z[k]) \end{cases}$$

## 13.12

13.12 给定包含  $n$  个字符的字符串  $s[1..n]$ , 该字符串可能来自一本年代久远的书籍, 只是由于纸张朽烂的缘故, 文档中所有的标点符号都不见了 (因此该字符串看起来就像这样: “itwasthebestoftimes...” )。现在你希望在字典的帮助下重建这个文档。在此, 字典表示为一个布尔函数  $dict(\cdot)$ , 对于任意的字符串  $w$ ,

$$dict(w) = \begin{cases} \text{TRUE} & w \text{ 是合法单词} \\ \text{FALSE} & \text{其他情况} \end{cases}$$

- 1) 请给出一个动态规划算法, 判断  $s[1..n]$  是否能重建为由合法单词组成的序列。假设调用  $dict$  每次只需一个单位的时间, 该算法运行时间要求不超过  $O(n^2)$ 。
- 2) 若  $s[1..n]$  是由合法单词组成的, 请输出对应的单词序列。

$$f_i = \bigvee_{0 \leq j < i} (f_j \wedge dict(s[j + 1..i])), \text{ 其中 } f \text{ 表示对于字符串前 } i \text{ 个字符是否可以组成合法序列。}$$

关于输出方案, 只需要每次用  $G_i$  记录是从前面哪个  $j$  转移来的 (也就是  $f_j = 1$  且  $dict(s[j + 1..i]) = 1$ ) 即可, 从  $G_n$  开始回溯, 每次记录单词  $s[G_x + 1..x]$  再令  $x = G_x$ 。最后把记录的单词翻转输出即可。

## 13.13[最长回文子序列]

请设计一个算法，计算对给定字符串可以拆分的最少回文数量，并分析算法的时间、空间复杂度。例如，给定输入字符串“BUBBASEESABANANA”，你的算法给出的结果应该是 3。

(1)  $f_{i,j}$  表示串  $S[i \dots j]$  内的最长回文子序列。则

$$f_{i,j} = \begin{cases} \max(f_{i+1,j}, f_{i,j-1}) & (S_i \neq S_j) \\ f_{i+1,j-1} + 1 + [i \neq j?] & (S_i = S_j) \end{cases}$$

边界  $f_{j,i} = 0 (j < i)$ ，上述方括号内是条件判断，true 则加 1（对应着长度总共加 2）。答案是  $f_{1,n}$ 。

(2) 显然设计  $f_i$  表示串前  $i$  位最少回文划分数。 $f_i = \min_{j < i \text{ 且 } S[j+1 \dots i] \text{ 是回文串}} \{f_j + 1\}$

$O(n^2)$  DP 即可。其中回文串的判断可以  $O(n^2)$  预处理，枚举每一个中心  $i$  并向两边拓展。总复杂度  $O(n^2)$ 。判断是否是回文串也可以利用问(1)的  $[f_{j+1,i} == i - j?]$

! 有一种使用回文自动机的复杂做法可以优化到  $O(n \log n)$ ，此处略。

## 13.15: (零钱兑换问题)

经典背包问题类型

- 令 $dp[i, j]$  表示  $x[1..i]$  能否兑换金额  $j$ ，由于硬币数量无限，有状态转移方程：

$$dp[i, j] = dp[i - 1, j] \ || \ dp[i, j - x_i], j \geq x_i$$

- 令 $dp[i, j]$  表示  $x[1..i]$  能否兑换金额  $j$ ，由于硬币数量唯一，有状态转移方程：

$$dp[i, j] = dp[i - 1, j] \ || \ dp[i - 1, j - x_i], j \geq x_i$$

- 令 $dp[i, j, k]$  表示仅使用不超过  $k$  枚硬币的情况下， $x[1..i]$  能否兑换金额  $j$ ，有状态转移方程：

$$dp[i, j, k] = dp[i - 1, j, k] \ || \ dp[i, j - x_i, k - 1], j \geq x_i, 1 \leq k \leq K$$

- 时间复杂度为  $O(nvK)$ ，其中设  $K$  为题干中最多使用的硬币数量

## 13.16: (最小顶点覆盖)

经典树形dp问题

**13.16** 图  $G = (V, E)$  的一个顶点覆盖  $S$  是  $V$  的子集, 满足:  $E$  中的每条边都至少有一个端点属于  $S$ 。请给出如下问题的一个线性时间的算法:

- 令  $dp_1[i]$  表示顶点  $i$  所在子树的最小顶点覆盖的大小, 且顶点  $i$  在该最小顶点覆盖中
- 令  $dp_2[i]$  表示顶点  $i$  所在子树的最小顶点覆盖的大小, 且顶点  $i$  不在该最小顶点覆盖中
- 状态转移方程:

$$dp_1[i] = 1 + \sum_{parent[j]=i} \min(dp_1[j], dp_2[j])$$
$$dp_2[i] = \sum_{parent[j]=i} dp_1[j]$$

- 时间复杂度为  $O(V)$
- 注意: 状态转移顺序是从叶子结点逐层往上

## 13.18

13.18 假设你准备开始一次长途旅行。以 0 公里为起点，一路上共有  $n$  座旅店，距离起点的公里数分别为  $a_1 < a_2 < \dots < a_n$ 。旅途中，你只能在这些旅店停留。最后一座旅店  $a_n$  为你的终点。理想情况下，你每天可以行进 200 公里，不过考虑到旅店间的实际距离，有时候可能还达不到这么远。如果你某天走了  $x$  公里，那么你将受到  $(200 - x)^2$  的惩罚。你需要计划好行程，以使总惩罚（每天所受惩罚的总和）最小。请设计一个高效的算法，计算一路上最优停留位置序列。

数据保证间隔不超过 200。则设  $f_i$  表示某天停留在  $i$  号点位的总最小惩罚。

$$\text{则 } f_i = \min_{a_i - a_j \leq 200} \{f_j + (200 - (a_i - a_j))^2\}$$

复杂度  $O(n^2)$ 。

！考虑高效的优化：拆开转移式去掉  $\min$  符号得，

$$f_i = f_j + 40000 + a_i^2 + a_j^2 - 2a_i a_j - 400a_i + 400a_j$$

$$\text{移项, } a_j^2 + f_j = (2a_i - 400)a_j + f_i + 400a_i - a_i^2 - 40000$$

以  $a_j^2 + f_j$  为纵坐标， $a_j$  为横坐标， $(2a_i - 400)$  为斜率，发现斜率在过程中是递增的，于是采用斜率优化，维护一个斜率单调不减的下凸包，用单调队列来保存斜率递增的决策点  $j$ ，每次只需要取队首开始第一个斜率  $k \geq (2a_i - 400)$  的决策点  $j$ 。复杂度  $O(n)$ 。

## 13.20: (最小代价进油计划)

注意题干中的若干限制条件

令  $dp[i, j]$  表示第  $i$  天结束时，油库里剩余  $j$  升油的最小总代价( $0 \leq j \leq L$ )，则有状态转移方程：

- $dp[i, j] = c \cdot j + \min_{k=0}^L \{ dp[i - 1, k] + P \times cost(i, j, k) \}$
- 其中： $cost(i, j, k) = \begin{cases} 1, & 0 < j - k + g_i \leq L \\ 0, & j - k + g_i = 0 \\ \infty, & others \end{cases}$
- 由于要求油库里的油必须卖完，因此最终未来  $n$  天的最小总代价为  $dp[n, 0]$
- 另外，为了输出进油计划，还需要一个备忘录数组  $memo[i, j]$  记录最终转移到  $dp[i, j]$  的  $dp[i - 1, k^*]$  的下标  $k^*$ ，则第  $i$  天的进油量为  $j - k^* + g_i$
- 分析：
  - 时间复杂度： $O(nL^2)$ ；空间复杂度： $O(nL)$
  - 注意：由于  $dp$  数组的状态转移仅依赖相邻两项，因此  $dp$  数组可作状态压缩到一维，但  $memo$  数组需要回溯无法压缩，故最终空间复杂度仍然是  $O(nL)$

## 13.23

**13.23** 假设你需要为一家公司策划一次宴会。公司所有人员组成一个层级关系，即所有人按照上下级关系组成一棵树，树的根节点为公司董事长。公司的人力资源部门为每个员工评估了一个实数值的友好度评分。为了聚会能够更轻松地进行，公司不希望一名员工和他的直接领导共同出现在宴会上。请设计一个算法决定宴会邀请人员名单，使得所有参加人员的友好度评分总和最大。

$$f_{i,0} = \sum_{j \in child_i} \max(f_{j,0}, f_{j,1})$$

$$f_{i,1} = \sum_{j \in child_i} f_{j,0} + a_i$$

边界：叶子  $f_{i,1} = a_i$

答案  $\max(f_{root,0}, f_{root,1})$

$f_{i,0/1}$  是  $i$  号参与 (1) 或者不参与 (0) 时候的最大答案。树形DP，初始化全0。 $O(n)$ . 关于方案，记录转移过程即可，然后从根开始向下回溯。

## 13.23

**13.23** 假设你需要为一家公司策划一次宴会。公司所有人员组成一个层级关系，即所有人按照上下级关系组成一棵树，树的根节点为公司董事长。公司的人力资源部门为每个员工评估了一个实数值的友好度评分。为了聚会能够更轻松地进行，公司不希望一名员工和他的直接领导共同出现在宴会上。请设计一个算法决定宴会邀请人员名单，使得所有参加人员的友好度评分总和最大。

$$f_{i,0} = \sum_{j \in child_i} \max(f_{j,0}, f_{j,1})$$

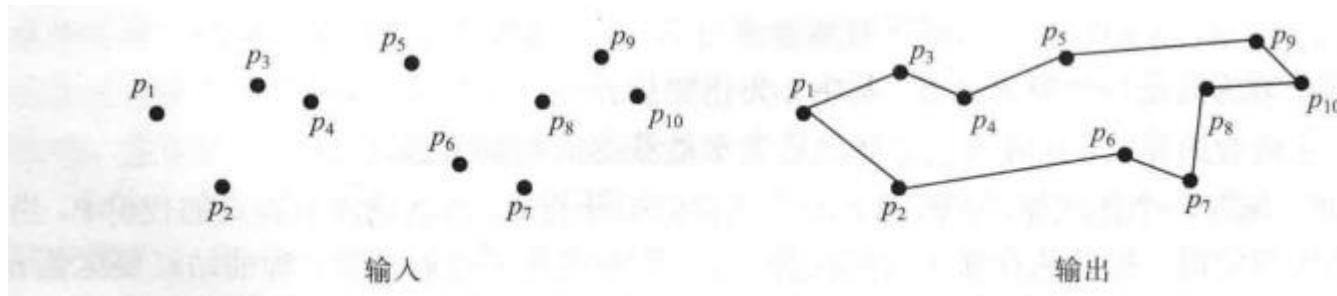
$$f_{i,1} = \sum_{j \in child_i} f_{j,0} + a_i$$

边界：叶子  $f_{i,1} = a_i$

答案  $\max(f_{root,0}, f_{root,1})$

$f_{i,0/1}$  是  $i$  号参与 (1) 或者不参与 (0) 时候的最大答案。树形DP，初始化全0。 $O(n)$ . 关于方案，记录转移过程即可，然后从根开始向下回溯。

13.24



实际就是规划2条中途点不重合的路径，使这2条路径之和最短。所以看成两条路都从左往右走即可。

设 $f_{x,y}$ 表示1号线路走到了 $x$ 号点，2号线路走到了 $y$ 号点。若 $x > y$ ，表示1号线路走了 $y+1 \dots x$ 这些点，2号线路还停留在 $y$ 点。反之亦然。于是枚举 $x, y \in [1, n]$ ：

*if*  $x > y$  *then* :  $f_{x+1,y} = \min(f_{x+1,y}, f_{x,y} + \text{dist}(x, x+1))$ ,  $f_{x,x+1} = \min(f_{x,x+1}, f_{x,y} + \text{dist}(y, x+1))$

*if*  $x < y$  *then* :  $f_{x,y+1} = \min(f_{x,y+1}, f_{x,y} + \text{dist}(y, y+1))$ ,  $f_{y+1,y} = \min(f_{y+1,y}, f_{x,y} + \text{dist}(x, y+1))$

即每次最领先的那条线路的下一个点，轮到谁来走，可能是1号也可能是2号。

最终会得到 $f_{n,i \neq n}$ ,  $f_{i \neq n,n}$ 的所有最优答案。枚举找这里面 $f_{n,i} + \text{dist}(i, n)$ 或 $f_{i,n} + \text{dist}(i, n)$  (实际上是  
对称的) 的最小值即可。复杂度 $O(n^2)$ .

进一步地改进，因为答案的对称性，不妨直接保证 $f_{x,y}$ 中必须 $x > y$ .

## 20.1: (经典NP问题的证明) - 1

1)

问题	优化问题	判定问题
CLIQUE	输入无向图，输出最大团大小	输入无向图和 $k$ ，判断图中是否有大小为 $k$ 的团
KNAPSACK	输入 $n$ 个物品、每个物品大小和价值、背包大小，输出背包能装物品的最大价值	输入 $n$ 个物品、每个物品大小和价值、背包大小和价值 $k$ ，输出背包能否装价值不小于 $k$ 的物品
INDEPENDENT-SET	输入无向图，输出最大独立集	输入无向图和 $k$ ，输出图中是否存在大小为 $k$ 的独立集
VERTEX-COVER	输入无向图，输出最小顶点覆盖	输入无向图和 $k$ ，输出图中是否存在大小为 $k$ 的顶点覆盖

## 20.1: (经典NP问题的证明) - 2

2)

问题	优化 $\Rightarrow$ 判定	优化 $\leq$ 判定
CLIQUE	多项式内解出优化问题，再把优化结果和 $k$ 比较即可	对 $k$ 的所有取值进行判定，即可解决优化问题，因为 $k$ 不超过 $ V $ ，故还是多项式时间
KNAPSACK	多项式内解出优化问题，再把优化结果和 $k$ 比较即可	对 $k$ 的所有取值进行判定，即可解决优化问题，因为 $k$ 不超过价值总和，故还是多项式时间
INDEPENDENT-SET	多项式内解出优化问题，再把优化结果和 $k$ 比较即可	对 $k$ 的所有取值进行判定，即可解决优化问题，因为 $k$ 不超过 $ V $ ，故还是多项式时间
VERTEX-COVER	多项式内解出优化问题，再把优化结果和 $k$ 比较即可	对 $k$ 的所有取值进行判定，即可解决优化问题，因为 $k$ 不超过 $ V $ ，故还是多项式时间

## 20.1: (经典NP问题的证明) - 3

3)

问题	证明
CLIQUE	给定无向图和 $k$ ，验证一个大小为 $k$ 的点集是否为团，只需验证是否所有点对之间存在边，时间为 $O(k^2) = O(n^2)$ ，所以是NP问题
KNAPSACK	给定 $n$ 个物品、每个物品大小和价值、背包大小、价值 $k$ ，验证一个物品集合是否能放入背包且价值不小于 $k$ ，验证大小之和不大于背包容量需要 $O(n)$ 时间，验证价值之和不小于 $k$ 也是 $O(n)$ 时间，所以是NP问题
INDEPENDENT-SET	给定无向图和 $k$ ，验证一个大小为 $k$ 的点集是否为独立集，只需验证点和点之间是否相邻，时间为 $O(k^2) = O(n^2)$ ，所以是NP问题
VERTEX-COVER	给定无向图和 $k$ ，验证一个大小为 $k$ 的点集是否为点覆盖，只需验证每个边的两个端点是不是至少有一个在点覆盖中 $O(kE) = O(n^3)$ ，所以是NP问题

## 20.2: (多项式归约关系的传递性)

假设  $P \leq_P Q$  且  $Q \leq_P R$

- $P \leq_P Q$ : 存在一个转换函数  $T_1(x)$ , 且  $T_1(x)$  为多项式时间, 使得对于  $P$  的合法输入  $x$ , 转换为  $T_1(x)$  是  $Q$  的合法输入, 且两者输出相同
- $Q \leq_P R$ : 存在一个转换函数  $T_2(x)$ , 且  $T_2(x)$  为多项式时间, 使得对于  $Q$  的合法输入  $x$ , 转换为  $T_2(x)$  是  $R$  的合法输入, 且两者输出相同
- 对于  $P$  的合法输入  $x$ ,  $T_1(x)$  为  $Q$  的合法输入,  $T_2(T_1(x))$  是  $R$  的合法输入, 且  $T_2(T_1(x))$  为多项式时间, 三者输出相同

则  $T_2(T_1(x))$  为  $P \leq_P R$  的转换函数

所以  $\leq_P$  是一个传递关系

## 20.3: (归约复杂度的计算)

**20.3** 假设  $A \leq_P B$ , 归约可以在  $O(n^2)$  时间内完成,  $B$  可以在  $O(n^4)$  时间内解决。请计算解决  $A$  问题所需的时间。

- 由于  $B$  自身开销:  $O(n^4)$
- 归约的开销, 即  $A$  的输入  $x$  转换为  $B$  的输入  $T(x)$  的开销:  $O(n^2)$
- 因此  $A$  的总开销为两部分之和:  $O(n^2) + O(n^4) = O(n^4)$

## 20.4: (排序和选择之间的归约)

问题间的归约：如果  $P$  问题能在多项式时间内归约到  $Q$  问题，说明  $P$  问题可以通过多项式时间的输入转换以及多项式次黑盒式地调用  $Q$  问题的算法来解决

- 排序问题多项式归约到选择问题：排序算法只需  $n$  次调用选择算法，每次选择阶为  $i$  的元素， $i : 1 \rightarrow n$
- 选择问题多项式归约到排序问题：选择算法只需调用 1 次排序算法然后遍历一遍，即可完成选择
- 想起了选择排序

**20.4** 给定“排序”与“选择”这两个问题，请从两个方向给出它们相互之间的归约。归约的过程是否使你想起某个排序算法？（提示：本题和后面两题所讨论的归约并不是判定问题之间的归约。你需要泛化 20.2.1 节中对于判定问题间归约的定义。简单而言，问题间的归约就是“黑盒”地调用一个问题的算法去解决另一个问题。）

## 20.5: (选择问题之间的归约)

问题间的归约：如果  $P$  问题能在多项式时间内归约到  $Q$  问题，说明  $P$  问题可以通过多项式时间的输入转换以及多项式次黑盒式地调用  $Q$  问题的算法来解决

- 问题1到问题2的归约：向问题2输入  $k = \frac{|S|}{2}$  即可
- 问题2到问题1的归约：类似于快速选择，每次调用问题1的算法，找到中位数，并利用其进行 *partition*，都能排除一半规模，使得阶  $k$  的元素在另外一半，则最后当规模为1时，即找到了阶为  $k$  的元素

20.5 给定下列两个问题：

- 问题 1：找出集合  $S$  中所有数的中位数。
- 问题 2：找出集合  $S$  的所有数中阶为  $k$  (第  $k$  小) 的数。

请给出问题 1 到问题 2 的归约和问题 2 到问题 1 的归约。

## 21.1: (P/NP/NPC 的关系)

NPC问题首先属于NP问题，且是NP问题中最难的问题；而NPH问题不一定是NP问题，且不比所有NP问题容易

### 21.1 请证明：

- 1) 如果任意一个 NP 完全问题可以在多项式时间解决，则所有 NP 问题均可以在多项式时间解决，即  $P = NP$ 。
  - 2) 如果任意一个 NP 完全问题不存在多项式时间的解，则所有 NP 完全问题均不可能在多项式时间解决。
- 证明：如果存在一个  $NPC$  问题可以在多项式时间内解决，而所有  $NP$  问题都可在多项式时间归约到该  $NPC$  问题，从而多项式时间内解决，从而  $P = NP$
- 2)
- 逆否命题：“如果存在一个  $NPC$  问题有多项式时间解，则所有  $NPC$  问题都有多项式时间解”
  - 证明：由于  $NPC$  问题也是  $NP$  问题，根据1) 的结论可以直接得证

## 21.3: (伪最大团问题)

1)

- 算法：从  $n$  个点中选  $k$  个点，验证该  $k$  个点是否是团
- 时间复杂度：

- 选取开销： $O(\binom{n}{k}) = O\left(\frac{n!}{k!(n-k)!}\right) = O\left(\frac{n^n}{k^k(n-k)^{n-k}}\right) = O\left(\left(\frac{n}{k}\right)^k\right) = O(n^k)$
- 验证开销： $O(k^2) = O(n^2)$

2) 伪最大团问题和最大团问题的区别在于  $k$  的性质：

- 伪最大团问题  $k$  是一个常数，不依赖于  $n$
- 最大团问题  $k$  是一个参数，依赖于  $n$  e.g.  $k = \frac{n}{2}$

所以该多项式算法不能证明  $P = NP$

## 21.4: (DNF-SAT问题)

注意不要默认归约代价一定是多项式时间

1) 证明: 对于一个析取范式  $C_1 \vee C_2 \vee \cdots \vee C_n$ , 其中  $C_i = q_1 \wedge q_2 \wedge \dots \wedge q_m$

- 只需要一个子句  $C_i$  为真即可满足, 外循环次数为  $O(n)$
- 对于每一个子句  $C_i$  为真, 当且仅当  $\forall l \in C_i, \neg l \notin C_i$ , 内循环开销为  $O(m)$
- 时间复杂度为:  $O(nm)$ , 因此是  $P$  问题

2)

- “CNF-SAT的输入转化为DNF-SAT的输入”这一步暂未发现多项式级别算法
- 因此转换函数  $T(x)$  不是多项式时间算法, 于是题干的归约不是多项式时间归约
- 从而无法证明  $P = NP$

## 21.5: (背包问题)

要证明某问题是 $NPC$ 问题，只需证明：① 该问题是 $NP$ 问题；② 已知的某个 $NPC$ 问题能够在多项式时间内归约到该问题，即该问题是一个 $NPH$ 问题

1) 证明：

- ① 背包问题是 $NP$ 问题：20.1已证；
- ② 子集和问题能多项式时间归约到背包问题：给定自然数  $S$  和  $A = \{s_1, s_2, \dots, s_n\}$ ，令背包大小为  $S$ ，且  $k = S$ ，物品大小和价值都是  $A = \{s_1, s_2, \dots, s_n\}$ ，使用背包问题的算法，输出是否存在大小和不超过  $S$ ，且价值和不低于  $k = S$  的放法，输出结果就是子集和问题的结果

2)  $dp[i, j]$  表示前  $i$  个物品装进大小为  $j$  背包的最大价值，则有状态转移方程：

$$dp[i, j] = \max\{dp[i - 1, j], dp[i - 1, j - weight[i]] + value[i]\}$$

3) 时间复杂度和空间复杂度都是  $O(nW)$ ，但输入规模是  $\log W$ ，因此严格写法为  $O(n2^{\log W})$ ，是指数级算法，因此不代表找到了一个多项式时间的算法

## 21.6: (支配集问题和集合覆盖问题)

要证明某问题是  $NPC$  问题，只需证明：① 该问题是  $NP$  问题；② 已知的某个  $NPC$  问题能够在多项式时间内归约到该问题，即该问题是一个  $NPH$  问题

- 先证明  $SET\text{-}COVER$  是  $NP$  问题：
  - 对于一个给定解，判定其大小是否为  $k$ ，再检查全集中每个元素是否都在这个解中，代价为  $O(|U|)$
- 再证明  $DOMINATION\text{-}SET$  可以多项式归约到  $SET\text{-}COVER$ ：
  - 对于一个  $DOMINATION\text{-}SET$ ，给定无向图  $G$ ，令全集  $U$  为顶点集合  $V$ ，对于每个顶点  $v_i$ ，将  $v_i$  和其所有邻居  $neighbor(v_i)$  组成集合  $S_i$ ，调用集合覆盖算法，判定是否存在大小为  $k$  的集合覆盖
  - 集合覆盖算法的判定结果就是支配集问题的判定结果，即是否存在大小为  $k$  的支配集
    - 1)  $DOMINATION\text{-}SET$ : 对于无向图  $G$ ，其中是否有大小为  $k$  的支配集<sup>⊕</sup>？
    - 2)  $SET\text{-}COVER$ : 给定全集  $U$  以及  $U$  的  $n$  个子集  $S_1, S_2, \dots, S_n$  满足  $\bigcup_{i=1}^n S_i = U$ 。是否存在大小为  $k$  的集合覆盖<sup>⊕</sup>？
    - 3) 已知  $DOMINATION\text{-}SET$  是  $NP$  完全问题，请证明  $SET\text{-}COVER$  是  $NP$  完全问题。
- 所以  $SET\text{-}COVER$  是  $NPC$  问题

<sup>⊕</sup> 图  $G = (V, E)$  的支配集  $D$  的定义为：任意  $V \setminus D$  中的点均和  $D$  中的某个点有边相连。

<sup>⊖</sup> 集合覆盖是若干给定的子集组成的子集族，其中所有子集的并为全集。集合覆盖的大小为其中子集的个数。

## 21.10: (回避路径问题)

要证明某问题是 *NPC* 问题，只需证明：**21.10 (回避路径问题)** 在导航应用中，用户有时希望在选择路径时，避免一个地方走多次，由此产生了回避路径问题 (Evasive Path Problem, EPP)。给定有向图  $G = (V, E)$ ，指定约到该问题，即该问题是一个 *NPH* 问题的起点  $s$  和终点  $t$ ，另外还指定了一组没有重叠的区域  $Z_1, Z_2, \dots, Z_k$ ，其中  $Z_i \subseteq V (1 \leq i \leq k)$ 。

问题是判断是否存在从  $s$  到  $t$  的路径，满足每个区域中至多访问一个节点。请证明 EPP 问题

- 首先证明 *EPP* 问题是 *NP* 问题：是 *NP* 完全的。(可以假设已知有向哈密顿路径问题是 *NP* 完全的。)
  - 给定一条从  $s$  到  $t$  的路径，检验该路径上的所有点所踏足的区域，如果存在一个区域  $Z_i$  被访问超过一次，则该解非法，否则是合法解，时间开销为路径  $O(|V|)$
- 其次证明有向哈密顿路径问题 (*DHP*) 能够多项式时间归约到 *EPP* 问题：
  - 给定图  $G = (V, E)$ ，其中  $V = \{v_1, v_2, \dots, v_n\}$ ，遍历节点对  $\{(s, t) | s, t \in V, s \neq t\}$ ，重复以下过程：
  - 首先建立一个“ $n$  层图”  $G'$ ，第 1 层只有源点  $s$ ，第  $n$  层只有终点  $t$
  - 中间  $n - 2$  层，每层都有  $n - 2$  个节点，分别对应  $V / \{s, t\}$  的  $n - 2$  个节点  $\{u_1, u_2, \dots, u_{n-2}\}$
  - 只有相邻层之间的节点有边相连，即层内没有边相连，且跨层之间也没有边相连
  - 第  $i$  层的第  $j$  个节点  $u_j^{(i)}$  会有指向第  $i + 1$  层的第  $l$  个节点  $u_l^{(i+1)}$  的边，当且仅当  $(u_j^{(i)}, u_l^{(i+1)}) \in E$
  - 设计回避区域  $Z_1, Z_2, \dots, Z_{n-2}$ ，其中  $Z_j = \{u_j^{(1)}, u_j^{(2)}, \dots, u_j^{(n-2)}\}$ ，将该  $G'$  和  $Z_{1 \sim n-2}$  输入 *EPP* 算法
  - 则图  $G$  存在给定起点  $s$  和终点  $t$  的哈密顿路径，当且仅当  $G'$  存在从  $s$  到  $t$  的回避路径
  - 转换代价为  $O(n^3)$ ，遍历  $(s, t)$  对代价为  $O(n^2)$ ，因此 *DHP* 可以多项式时间归约到 *EPP* 问题

## OJ 4.A[分层图-最短路]

题意：给定无向图求1到N最短路，有 $K$ 次免费跳过某条边的机会，但是免费跳过一次后每走一条边代价要多1。

类似上题的思路，建立分层的图，图有 $k + 1$ 层，若原图中有无向边 $\langle x, y \rangle = w$ ，则新的分层图中，建边：

$$\langle x_k, y_k \rangle = \langle y_k, x_k \rangle = w + k, 0 \leq k \leq K$$

$$\langle x_k, y_{k+1} \rangle = \langle y_k, x_{k+1} \rangle = 0, 0 \leq k \leq K - 1$$

从点 $[1, 0]$ 开始跑优化Dijkstra，最后的答案就是 $\min_k(\text{dist}_{N,k})$

复杂度 $O(KM \log KN)$

# OJ 4.B[拓扑排序-DAG上递推 / 收缩强连通片]

问题分析：对于一个有向图 $G$ ，找到顶点1和顶点n的路径条数

- 结果为无限条的条件：顶点1到顶点n的路径中存在环
- 对于统计图上可叠加性属性的问题，通常都是要转换到DAG收缩图 $G'$ 来进行快速求解
- 转换成 $G'$ 后，可以通过拓扑排序/记忆化搜索在线性时间内解决：
  - 设 $f[u]$ 为 $u$ 到 $n$ 的路径条数，则：
$$f[u] = \sum_{(u,v) \in E} f[v]$$
  - 而结果为无限条的条件也很简单了：在拓扑/记忆化搜索的过程中判断当前强连通片大小是否为1，如果不是则打上是无限条的标记
  - 整个的时间复杂度是 $O(n + m)$

## OJ 4.C[最小生成树(并查集运用)]

题意：有一张 $n$ 个点 $m$ 条边的连通图。有 $Q$ 次询问。每次询问给出 $k[i]$ 条边，问是否存在某个最小生成树同时包含这些边。 $n, m \leq 500000, \sum k_i \leq 700000, Q \leq 100000$

利用Kruskal算法的特性理解最小生成树性质：在大小相同的边中，有的边是**必须**选入的边，有的边是**不可能**入选的边，有的边是可能入选的边。

在排过序的边中进行Kruskal连边操作时，对于边权都是 $w$ 的边，它们连接着一个个连通块（回想算法的并查集操作，并查集已经利用小于 $w$ 的边将点合并成一个个连通块了），这些已有的「连通块里含有什么点」对于这些 $w$ 边是透明的，大小为 $w$ 的边们只负责将看到的这些块尽所能合并和连通起来，不会影响后续大于 $w$ 的边们的连边操作。不管如何连边 $w$ ，操作完，任意两点间的连通情况是不会变的。

故而在进行Kruskal时，对于每条边两端点所属的集合代表（也就是这条边能够看到的「连通块」）记录下来。处理询问时，相同大小的边排序到一起处理，每次将询问边对应的两个块合并，直到相同边的一波次结束（没有问题）或者出现冲突（边的两端连着同一个块）。

复杂度考虑Kruskal过程和询问里排序边大小以及并查集的合并/撤销操作 $O(M \log M + Q \log Q)$

## OJ 5.A[背包]

有一组非负整数 $x_1, x_2, \dots, x_N$ 和两个正整数 $M, K$ , 满足等式

$x_1 + x_2 + \dots + x_N = M$ . 其中 $x_i \in [0, K]$  for any  $1 \leq i \leq N$

分组的背包方案数。直接代码演示。

```
while(~scanf("%d%d%d",&n,&m,&k)){
    mst(f);f[0]=1;
    for(int i=1;i<=n;++i)
        for(int j=m;j;--j)
            for(int x=1;x<=k&&x<=j;++x)
                f[j]=(f[j]+f[j-x])%MOD;
    printf("%d\n",f[m]);
}
```

# OJ 5.B[区间DP-前后缀优化]

问题分析：注意本游戏先后手的得分总和不变，因此先手的最佳策略就只需要最小化后手的最佳策略得分

- 预处理：求出 $a[1..n]$ 前缀和 $S[1..n]$ ，则任意子牌组 $a[i..j]$ 的得分和： $sum[i..j] = S[j] - S[i - 1]$
- 令 $dp[i][j]$ 表示在拿到子牌组 $a[i..j]$ 后，先手的最佳策略得分，则有状态转移方程：

$$dp[i][j] = sum[i..j] - \min\{0, \min_{k=i+1}^j dp[k][j], \min_{k=i}^{j-1} dp[i][k]\}$$

- 注意到状态转移中的前后缀性质，可进一步优化状态转移方程：

$$dp[i][j] = sum[i..j] - \min\{0, suf[i + 1][j], pre[i][j - 1]\}$$

$$suf[i][j] := \min_{k=i}^j dp[k][j] = \min\{dp[i][j], suf[i + 1][j]\}$$

$$pre[i][j] := \min_{k=i}^j dp[i][k] = \min\{dp[i][j], pre[i][j - 1]\}$$

- 最终结果为 $2 \times dp[1][n] - S[n]$ ；时间复杂度： $O(n^3) \xrightarrow{\text{优化}} O(n^2)$

# OJ 6.A[最长公共子串]

形式化的描述，有 $k$ 个非空串 $q_1, q_2, \dots, q_k$ ，满足：

$s$ 可以表记为串 $a_1 q_1 a_2 q_2 \dots a_k q_k a_{k+1}$ , 其中 $a_1, a_2, \dots, a_{k+1}$ 是一系列可能为空的串；

$t$ 可以表记为串 $b_1 q_1 b_2 q_2 \dots b_k q_k b_{k+1}$ , 其中 $b_1, b_2, \dots, b_{k+1}$ 是一系列可能为空的串；

求出最大的 $L = |q_1| + |q_2| + \dots + |q_k|$

设 $f[i][j][k][0/1]$ 表示 $s$ 串前 $i$ 位， $t$ 串前 $j$ 位，匹配了最多 $k$ 个子串，并且末尾的字母是否被选择了时的最大长度。

$$f[i][j][k][1] = \max(f[i-1][j-1][k][1], f[i-1][j-1][k-1][0]) + 1, \text{ 当 } s[i] = t[j]$$

$$f[i][j][k][0] = \max(\max(f[i-1][j][k][0], f[i][j-1][k][0]), \max(f[i-1][j][k][1], f[i][j-1][k][1]))$$

答案为 $\max(f[n][m][K][0], f[n][m][K][1])$

# OJ 6.B[线性DP-单调性优化]

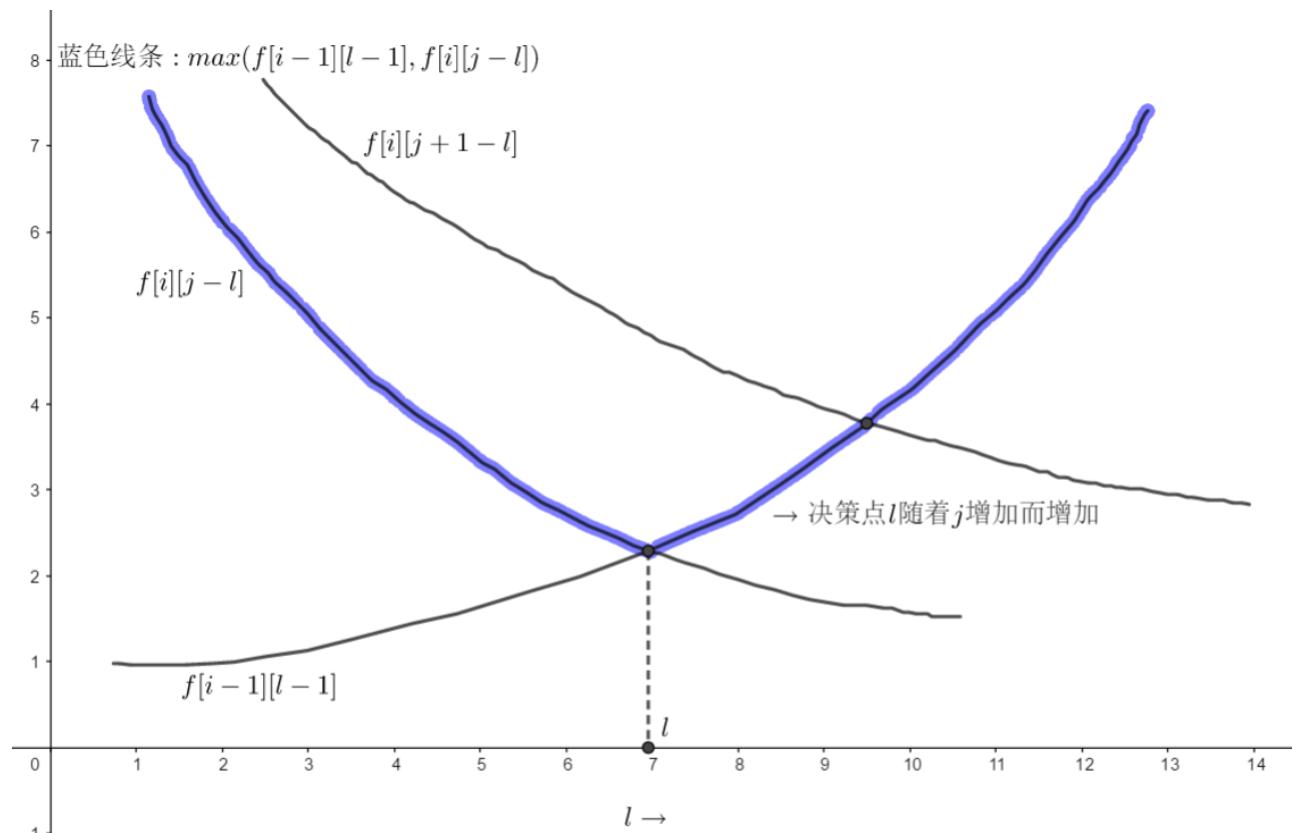
令 $f[i][j]$ 表示剩余*i*条命，牌组还剩*j*张牌最少需要几轮。则：

$$f[i][j] = \min_{l=1}^j \{ \max(f[i][j-l], f[i-1][l-1]) \}$$

最终答案则是两方取较小者。复杂度 $O(kn^2)$

优化：通过 $f$ 单调性和峰值来确定决策点。

复杂度 $O(kn)$



## OJ 6.B[线性DP-单调性优化]

!

- 令 $dp[i][t]$ 表示在剩余*i*个生命、还有*t*轮游戏时，能保证找到带标记小王的最大牌组数，则有状态转移方程：

$$dp[i][t] = 1 + dp[i][t - 1] + dp[i - 1][t - 1]$$

- 于是 $T^* = \arg \min_t dp[k][t], \ s.t. dp[k][t] \geq n$ ; 最终答案为： $\min\{T_X^*, T_Y^*\}$
- 优化：注意到状态转移方程的形式是一个二阶线性递推式，可以尝试着用生成函数法直接求出 $dp[k][t]$ 的通项公式，那么 $T^*$ 就可以直接进行在 $[0, n]$ 中二分查找，具体的通项公式（计算代价为 $O(k)$ ）推导见下一页
- 时间复杂度为： $O(kn) \xrightarrow{\text{优化}} O(k \log n)$

由于 $dp$ 数列有递推式:  $dp[k][t] = dp[k][t - 1] + dp[k - 1][t - 1] + 1$ ,  $dp[0][t] = dp[k][0] = 0$

设 $dp$ 数列的生成函数为:  $g_t(x) = \sum_{k=0}^{\infty} dp[k][t]x^k = \sum_{k=1}^{\infty} dp[k][t]x^k$ , 结合上述递推式, 可展开得:

$$\begin{aligned}
g_t(x) &= \sum_{k=1}^{\infty} dp[k][t]x^k = \sum_{k=1}^{\infty} \{dp[k][t - 1] + dp[k - 1][t - 1] + 1\}x^k \\
&= (1 + x)g_{t-1}(x) + \sum_{k=1}^{\infty} x^k = (1 + x)g_{t-1}(x) + \frac{x}{1 - x} \\
&= (1 + x)\left[(1 + x)g_{t-2}(x) + \frac{x}{1 - x}\right] + \frac{x}{1 - x} = \dots = (1 + x)^t \cdot g_0(x) + \frac{x}{1 - x} \cdot \sum_{i=0}^t (1 + x)^i \\
&= \frac{x}{1 - x} \cdot \frac{(1 + x)^t - 1}{x} = \frac{(1 + x)^t - 1}{1 - x} = [1 + x + x^2 + \dots] \cdot [C_t^1 x + C_t^2 x^2 + \dots] \\
&= C_t^1 x + [C_t^1 + C_t^2]x^2 + \dots = \sum_{k=1}^{\infty} \left[ \sum_{i=1}^k C_t^i \right] x^k \Rightarrow dp[k][t] = \sum_{i=1}^k C_t^i, \text{ 计算开销为 } O(k)
\end{aligned}$$

Thank You