

第3章 动态规划

3.3 最长公共子序列

• 若给定序列 $X=\{x_1, x_2, \dots, x_m\}$ ，则另一序列 $Z=\{z_1, z_2, \dots, z_k\}$ 是 X 的子序列是指存在一个严格递增下标序列 $\{i_1, i_2, \dots, i_k\}$ 使得对于所有 $j=1, 2, \dots, k$ 有： $z_j = x_{i_j}$ 。

例如，给定序列 $X=\{A, B, C, B, D, A, B\}$ ，序列 $Z=\{B, C, D, B\}$ 是 X 的子序列，相应的递增下标序列为 $\{2, 3, 5, 7\}$ 。

- 给定2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。
- 问题：给定2个序列 $X=\{x_1, x_2, \dots, x_m\}$ 和 $Y=\{y_1, y_2, \dots, y_n\}$ ，找出 X 和 Y 的最长公共子序列。

最长公共子序列的结构

设序列 $X=\{x_1, x_2, \dots, x_m\}$ 和 $Y=\{y_1, y_2, \dots, y_n\}$ 的最长公共子序列为 $Z=\{z_1, z_2, \dots, z_k\}$ ，从后向前推理，则

- (1) 若 $x_m = y_n$ ，则 $z_k = x_m = y_n$ ，且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的最长公共子序列。
- (2) 若 $x_m \neq y_n$ 且 $z_k \neq x_m$ ，则 Z 是 X_{m-1} 和 Y 的最长公共子序列。
- (3) 若 $x_m \neq y_n$ 且 $z_k \neq y_n$ ，则 Z 是 X 和 Y_{n-1} 的最长公共子序列。

由此可见，2个序列的最长公共子序列包含了这2个序列的前缀的最长公共子序列。因此，最长公共子序列问题具有最优子结构性质。

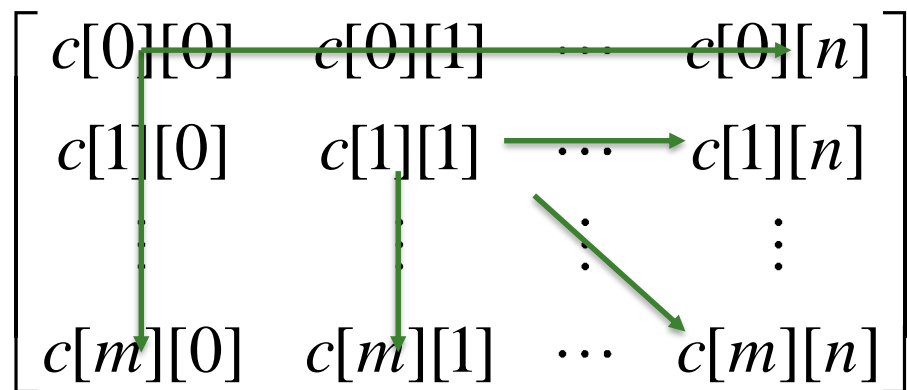
子问题的递归结构

由最长公共子序列问题的最优子结构性质建立子问题最优值的递归关系。用 $c[i][j]$ 记录序列 X_i 和 Y_j 的最长公共子序列的长度。其中， $X_i=\{x_1, x_2, \dots, x_i\}$;
 $Y_j=\{y_1, y_2, \dots, y_j\}$ 。当 $i=0$ 或 $j=0$ 时，空序列是 X_i 和 Y_j 的最长公共子序列。故此时 $c[i][j]=0$ 。其它情况下，由最优子结构性质可建立递归关系如下：

$$c[i][j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c[i-1][j-1] + 1 & i, j > 0; x_i = y_j \\ \max\{c[i][j-1], c[i-1][j]\} & i, j > 0; x_i \neq y_j \end{cases}$$

计算最优值

由于在所考虑的子问题空间中，总共有 $\theta(mn)$ 个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。



- 要得到 $c[i][j]$ ，需要其左侧、上侧和左上方的值
- 因此，先对 $c[i][0]$ 和 $c[0][j]$ 赋初值 0
- 然后计算 $c[1][1] \dots c[m][1]$ ，再计算 $c[1][2] \dots c[m][2]$
- 以此类推，最终得到 $c[m][n]$

计算最优值

由于在所考虑的子问题空间中，总共有 $\theta(mn)$ 个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。

```
void LCSLength(int m, int n, char *x, char *y, int **c, int **b)
```

```
{
    int i, j;
    for (i = 1; i <= m; i++) c[i][0] = 0;
    for (i = 1; i <= n; i++) c[0][i] = 0;
    for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++) {
            if (x[i]==y[j]) {
                c[i][j]=c[i-1][j-1]+1;
                b[i][j]=1; //(1) “左上角”
            }
            else if (c[i-1][j]>=c[i][j-1]) {
                c[i][j]=c[i-1][j];
                b[i][j]=2; //(2) “上侧”
            }
            else {
                c[i][j]=c[i][j-1];
                b[i][j]=3; //(3) “左侧”
            }
        }
}
```

构造最优解

```
void LCS(int i, int j, char *x, int **b)
```

```
{
    if (i ==0 || j==0) return;
    if (b[i][j]== 1){
        LCS(i-1, j-1, x, b);
        cout<<x[i];
    }
    else if (b[i][j]== 2) LCS(i-1, j, x, b);
    else LCS(i, j-1, x, b);
}
```

■ $b[i][j]$ 表示 $c[i][j]$ 取值的三种情况，用来构造最优解

LCS的回溯构造最优解过程

- $c[i][j]$ 存储最长公共子序列的长度
- $b[i][j]$ 记录 $c[i][j]$ 是由哪个子问题的解得到的
- 没有单独考虑 $c[i-1][j]==c[i][j-1]$ 的情况，对 $b[i][j]$ 二维数组的取值添加一种可能，等于4，来表示 $c[i-1][j]==c[i][j-1]$ 。这样通过路径回溯可以找出**所有**最长公共子序列。不做这项更改则只能找出一个最优解，不算错。

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
1	A	0(0)	0(4)	0(4)	0(4)	1(1)	1(1)
2	B	0(0)	1(1)	1(3)	1(3)	1(4)	2(1)
3	C	0(0)	1(2)	1(4)	2(1)	2(3)	2(4)
4	B	0(0)	1(1)	1(4)	2(2)	2(4)	3(1)
5	D	0(0)	1(2)	2(1)	2(4)	2(4)	3(4)
6	A	0(0)	1(2)	2(2)	2(4)	3(1)	3(4)
7	B	0(0)	1(1)	2(2)	2(4)	3(2)	4(1)

■ 指 $c[i][j]=4, b[i][j]=2$

比如序列 X 为 ABCBDAB，序列 Y 为 BDCABA，
则其公共最长子序列为 BCBA \ BCAB \ BDAB

算法的改进

- 在算法 **lcsLength** 和 **lcs** 中，可进一步将数组 **b** 省去。事实上，数组元素 $c[i][j]$ 的值仅由 $c[i-1][j-1]$ ， $c[i-1][j]$ 和 $c[i][j-1]$ 这3个数组元素的值所确定。对于给定的数组元素 $c[i][j]$ ，可以不借助于数组 **b** 而仅借助于三个值的关系在 $O(1)$ 时间内来确定 $c[i][j]$ 的值是由 $c[i-1][j-1]$ ， $c[i-1][j]$ 和 $c[i][j-1]$ 中哪一个值所确定的。
- 如果只需要计算最长公共子序列的长度，则算法的空间需求可大大减少。事实上，在计算 $c[i][j]$ 时，只用到数组 **c** 的第 i 行和第 $i-1$ 行。因此，用2行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至 $O(\min(m,n))$ 。

算法的改进

■ 比如:

$$\begin{bmatrix} c[0][0] & c[0][1] & \cdots & c[0][n] \\ c[1][0] & c[1][1] & \cdots & c[1][n] \\ \vdots & \vdots & \vdots & \vdots \\ c[m][0] & c[m][1] & \cdots & c[m][n] \end{bmatrix}$$

■ 只准备 $(m+1)*2$ 大小的空间

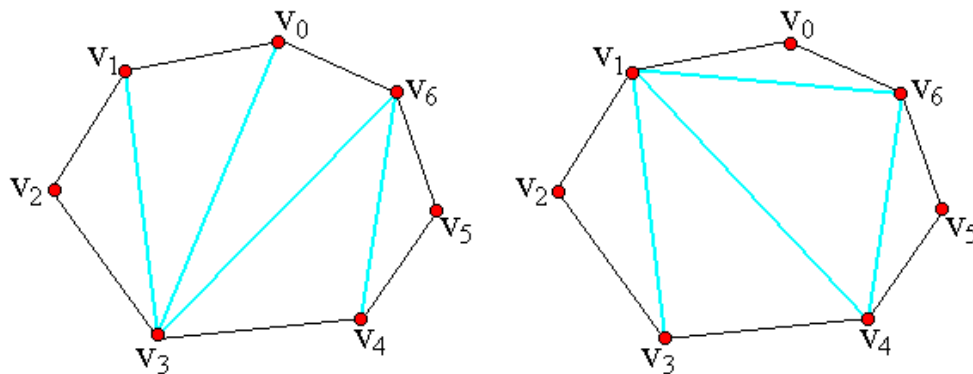
$$\begin{bmatrix} c[0][0] & c[0][1] \\ c[1][0] & c[1][1] \\ \vdots & \vdots \\ c[m][0] & c[m][1] \end{bmatrix} \xrightarrow{\text{shift}} \begin{bmatrix} c[0][1] & c[0][2] \\ c[1][1] & c[1][2] \\ \vdots & \vdots \\ c[m][1] & c[m][2] \end{bmatrix} \xrightarrow{\text{update}} \begin{bmatrix} c[0][n-1] & c[0][n] \\ c[1][n-1] & c[1][n] \\ \vdots & \vdots \\ c[m][n-1] & c[m][n] \end{bmatrix}$$

密码脱落问题

- X星球的考古学家发现了一批古代留下来的密码。公这些密码是由 A、B、C、D 四种植物的种子串成的序仔细分析发现，这些密码串当初应该是前后对称的，由于年代久远，其中许多种子脱落了，因而可能会失去镜像的特征。
- 你的任务是: 给定一个现在看到的密码串，计算一下从当初的状态它要至少脱落多少个种子，才可能会变成现在的样子。输入一行，表示现在看到的密码串 (长度不大于100(要求输出一个正整数，表示至少脱落了多少个种子。
- ABCBA ----- 0

3.5 凸多边形最优三角剖分

- 用多边形顶点的逆时针序列表示凸多边形，即 $P=\{v_0, v_1, \dots, v_{n-1}\}$ 表示具有 n 条边的凸多边形。
- 若 v_i 与 v_j 是多边形上不相邻的2个顶点，则线段 $v_i v_j$ 称为多边形的一条弦。弦将多边形分割成2个多边形 $\{v_i, v_{i+1}, \dots, v_j\}$ 和 $\{v_j, v_{j+1}, \dots, v_i\}$ 。
- 多边形的三角剖分是将多边形分割成互不相交的三角形的弦的集合 T 。
- 给定凸多边形 P ，以及定义在由多边形的三个顶点组成的三角形上的权函数 $w(v_i v_k v_j)$ 。要求确定该凸多边形的三角剖分，使得在该三角剖分中诸三角形上权之和为最小。

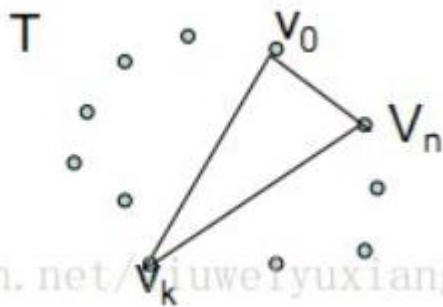


最优子结构性质

- 凸多边形的最优三角剖分问题有最优子结构性质。
- 事实上，若凸 $(n+1)$ 边形 $P=\{v_0, v_1, \dots, v_n\}$ 的最优三角剖分 T 包含三角形 $v_0 v_k v_n$ ， $1 \leq k \leq n-1$ ，则 T 的权为3个部分权的和：三角形 $v_0 v_k v_n$ 的权，子多边形 $\{v_0, v_1, \dots, v_k\}$ 和 $\{v_k, v_{k+1}, \dots, v_n\}$ 的权之和。可以断言，由 T 所确定的这2个子多边形的三角剖分也是最优的。因为若有 $\{v_0, v_1, \dots, v_k\}$ 或 $\{v_k, v_{k+1}, \dots, v_n\}$ 的更小权的三角剖分将导致 T 不是最优三角剖分的矛盾。

最优三角剖分的递归结构

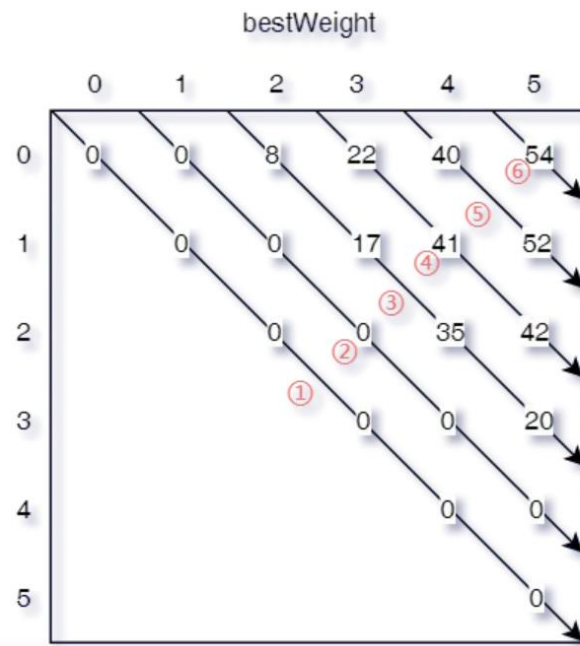
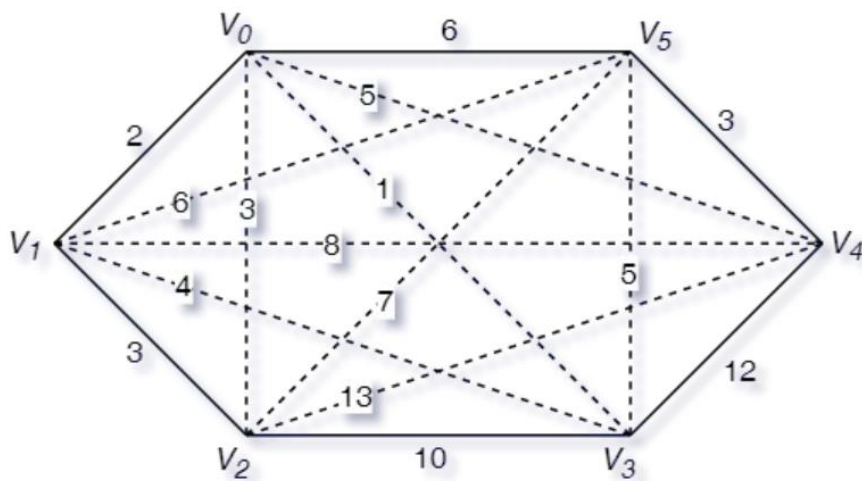
- 定义 $t[i][j]$, $1 \leq i < j \leq n$ 为凸子多边形 $\{v_{i-1}, v_i, \dots, v_j\}$ 的最优三角剖分所对应的权函数值, 即其最优值。为方便起见, 设退化的多边形 $\{v_{i-1}, v_i\}$ 具有权值 0。据此定义, 要计算的凸 $(n+1)$ 边形 P 的最优权值为 $t[1][n]$ 。
- $t[i][j]$ 的值可以利用最优子结构性质递归地计算。当 $j-i \geq 1$ 时, 凸子多边形至少有 3 个顶点。由最优子结构性质, $t[i][j]$ 的值应为 $t[i][k]$ 的值加上 $t[k+1][j]$ 的值, 再加上三角形 $v_{i-1}v_kv_j$ 的权值, 其中 $i \leq k \leq j-1$ 。由于在计算时还不知道 k 的确切位置, 而 k 的所有可能位置只有 $j-i$ 个, 因此可以在这 $j-i$ 个位置中选出使 $t[i][j]$ 值达到最小的位置。



最优三角剖分的递归结构

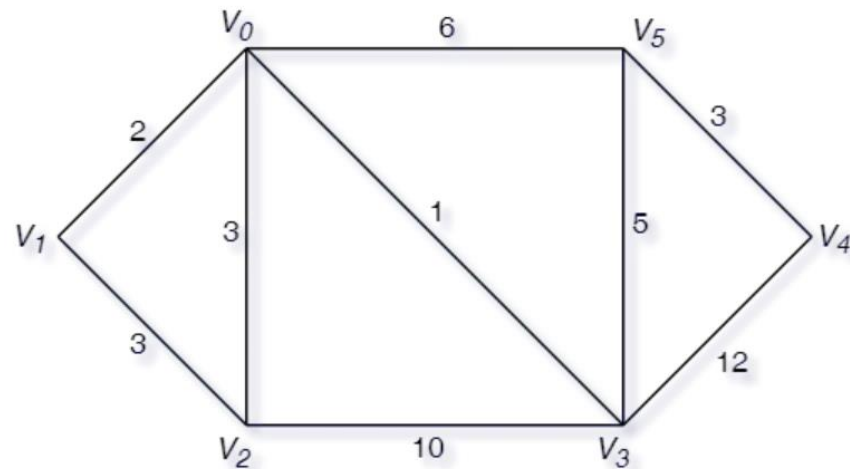
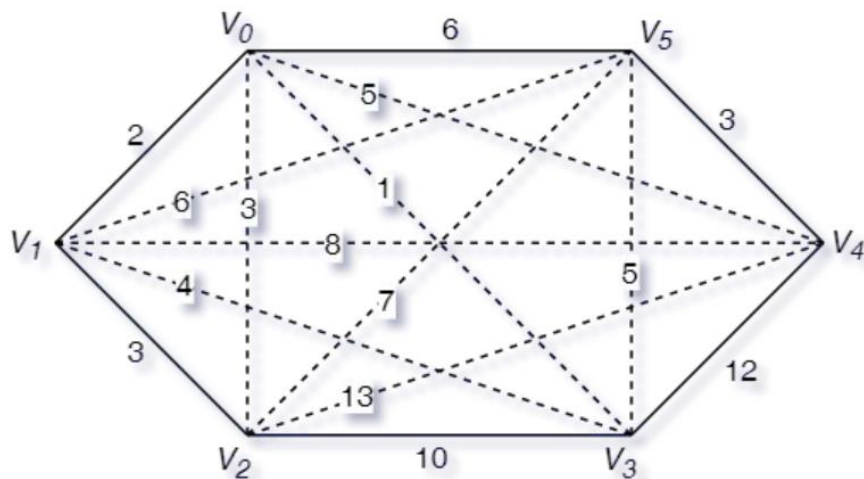
- 由此， $t[i][j]$ 可递归地定义为：

$$t[i][j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{t[i][k] + t[k+1][j] + w(v_{i-1}v_kv_j)\} & i < j \end{cases}$$



■ 得到该递归式就可以写出动态规划算法

最优三角剖分



- 此凸 6 边形的最优三角剖分的权值之和为 $(2 + 3 + 3) + (3 + 10 + 1) + (1 + 5 + 6) + (5 + 12 + 3) = 54$ 。


```

public static void minWeightTriangulation(int n) {
    for (int i = 1; i <= n; i++) {
        t[i][i] = 0;
    }
    for (int r = 2; r <= n; r++) { //i与j的差值
        for (int i = 1; i <= n - r + 1; i++) {
            int j = i + r - 1;
            m[i][j] = m[i + 1][j] + w(i-1,i,j); //k==i的情况
            s[i][j] = i;
            for (int k = i + 1; k < i+r-1; k++) {
                int u = m[i][k] + m[k + 1][j] + w(i-1,k,j);
                if (u < m[i][j]) {
                    m[i][j] = u;
                    s[i][j] = k; }}}}

```

3.6 多边形游戏

多边形游戏是一个单人玩的游戏，开始时有一个由 n 个顶点构成的多边形。每个顶点被赋予一个整数值，每条边被赋予一个运算符“+”或“*”。所有边依次用整数从1到 n 编号。

游戏第1步，将一条边删除。

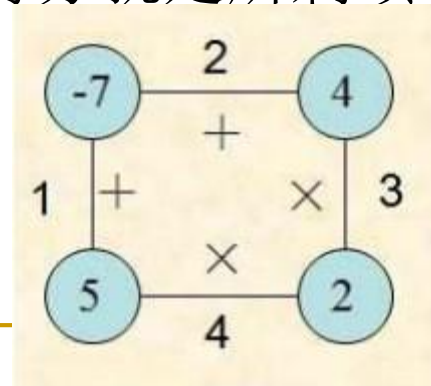
随后 $n-1$ 步按以下方式操作：

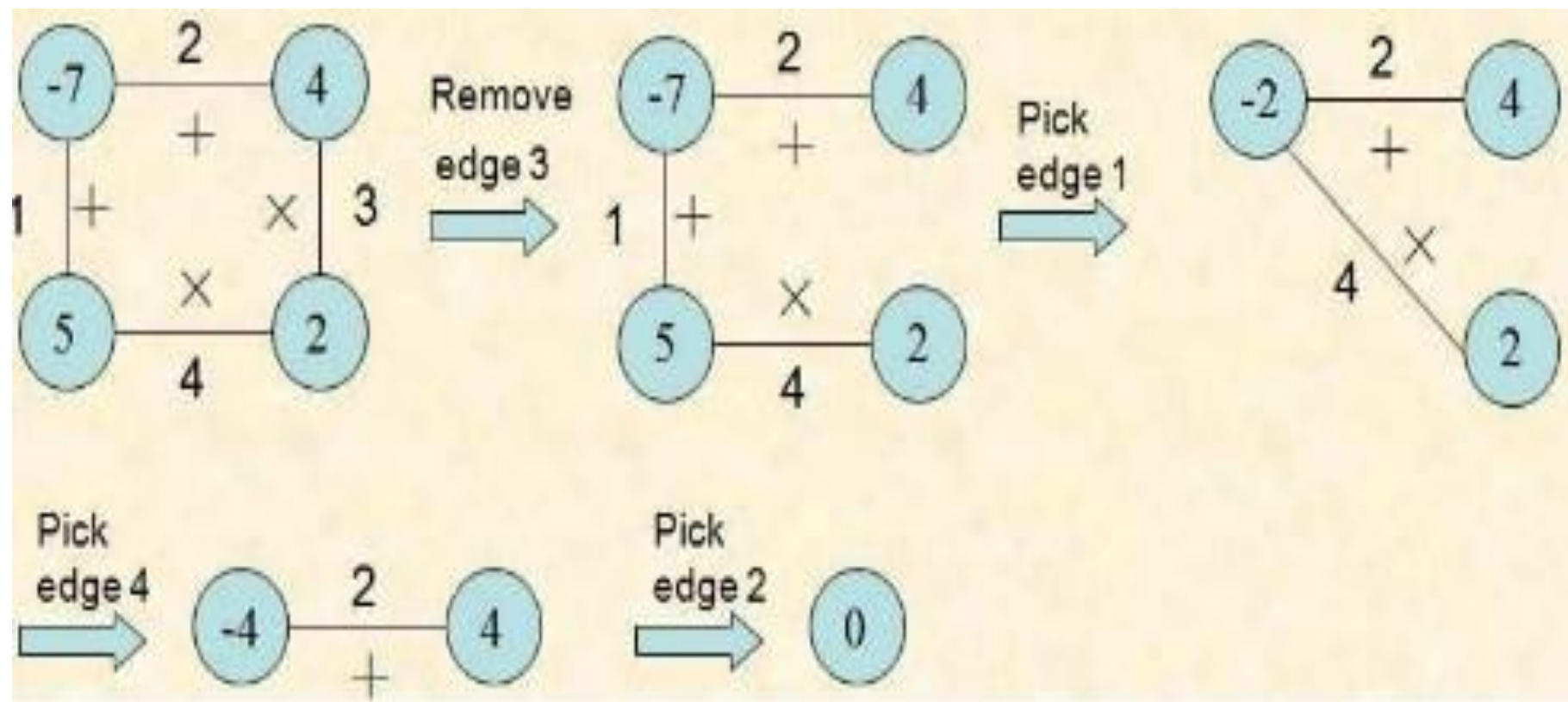
(1)选择一条边 E 以及由 E 连接着的2个顶点 V_1 和 V_2 ；

(2)用一个新的顶点取代边 E 以及由 E 连接着的2个顶点 V_1 和 V_2 。将由顶点 V_1 和 V_2 的整数值通过边 E 上的运算得到的结果赋予新顶点。

最后，所有边都被删除，游戏结束。游戏的得分就是所剩顶点上的整数值。

问题:对于给定的多边形，计算最高得分。

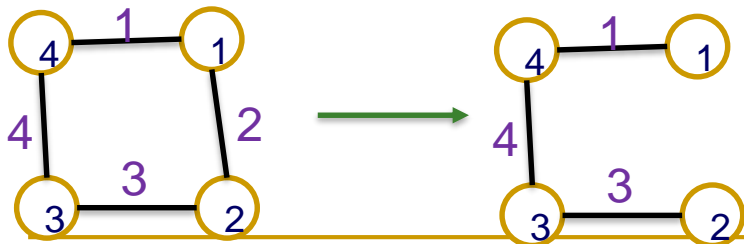




多边形游戏规则

最优子结构性质

- 设所给的多边形的边和顶点的顺时针序列为 $op[1], v[1], op[2], v[2], op[3], \dots, op[n], v[n]$ 其中, $op[i]$ 表示第 i 条边所对应的运算符, $v[i]$ 表示第 i 个顶点上的数值, $i=1 \sim n$ 。
- 在所给多边形中, 从顶点 $i (1 \leq i \leq n)$ 开始, 长度为 j (链中有 j 个顶点) 的顺时针链 $p(i, j)$ 可表示为 $v[i], op[i+1], \dots, v[i+j-1]$ 。则将一条边 $op[i]$ 删去后, 得到的链为 $p(i, j)$ 。
- 如果这条链的最后一次合并运算在 $op[i+s]$ 处发生 ($1 \leq s \leq j-1$), 则可在 $op[i+s]$ 处将链分割为 2 个子链 $p(i, s)$ 和 $p(i+s, j-s)$ 。
- 这里的加法需要进行取模运算, 即 $\text{mod } n$



$$p(2, 4) = v(2), op(3), \dots, v(1)$$

最优子结构性质

- 设 m_1 是对子链 $p(i, s)$ 的任意一种合并方式得到的值，而 a 和 b 分别是在所有可能的合并中得到的最小值和最大值。 m_2 是 $p(i+s, j-s)$ 的任意一种合并方式得到的值，而 c 和 d 分别是在所有可能的合并中得到的最小值和最大值。依此定义有 $a \leq m_1 \leq b$, $c \leq m_2 \leq d$

(1) 当 $op[i+s]='+'$ 时，显然有 $a+c \leq m \leq b+d$

(2) 当 $op[i+s]='*'$ 时，有 $\min\{ac, ad, bc, bd\} \leq m \leq \max\{ac, ad, bc, bd\}$

- 换句话说，主链的最大值和最小值可由子链的最大值和最小值得到。

多边形游戏的递归求解

■ 设 $m[i,j,0]$ 是链 $p(i,j)$ 合并的最小值，而 $m[i,j,1]$ 是最大值。若最优合并是在 $op[i+s]$ 处将 $p(i,j)$ 分为两个长度小于 j 的子链 $p(i, s)$ 和 $p(i+s, j-s)$ 的最大值和最小值均已计算出。即：

$$a = m[i, s, 0],$$

$$b = m[i, s, 1]$$

$$c = m[i + s, j - s, 0],$$

$$d = m[i + s, j - s, 1]$$

(1) 当 $op[i+s]='+'$ 时

$$m[i, j, 0] = a + c$$

$$m[i, j, 1] = b + d$$

(2) 当 $op[i+s]='*'$ 时

$$m[i, j, 0] = \min\{ac, ad, bc, bd\}$$

$$m[i, j, 1] = \max\{ac, ad, bc, bd\}$$

多边形游戏的递归求解

- 综合 (1)和(2),将 $p(i, j)$ 在 $op[i+s]$ 处断开的最大值记为 $maxf(i, j, s)$, 最小值记为 $minf(i, j, s)$

- $$minf(i, j, s) = \begin{cases} a + c & op[i+s]='+' \\ \min\{ac, ad, bc, bd\} & op[i+s]='*' \end{cases}$$

- $$maxf(i, j, s) = \begin{cases} b + d & op[i+s]='+' \\ \max\{ac, ad, bc, bd\} & op[i+s]='*' \end{cases}$$

- 由于最优断开位置 s 有 $1 \leq s \leq j-1$ 的 $j-1$ 中情况。

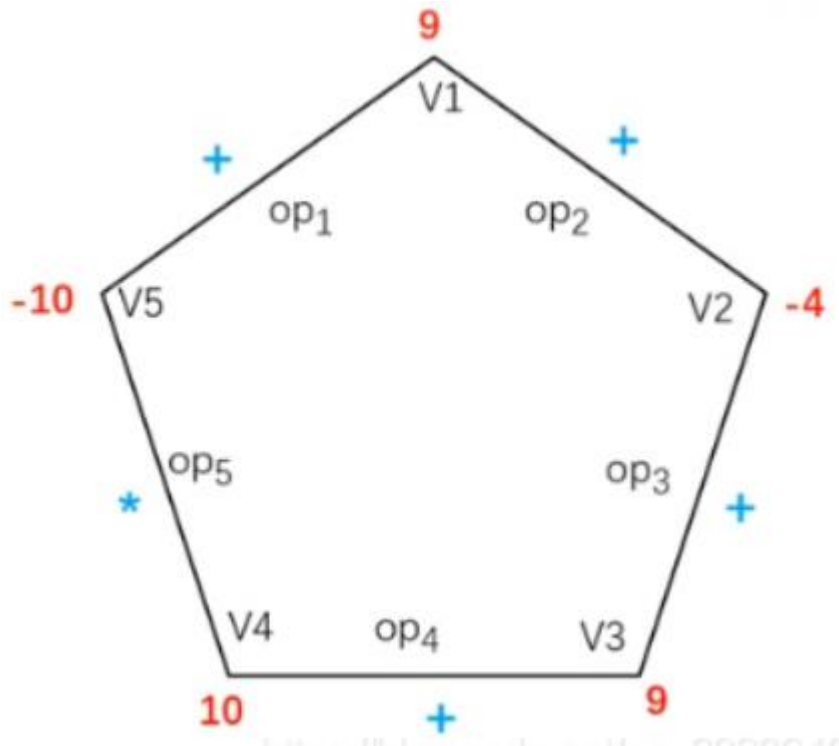
$$m[i, j, 0] = \min_{1 \leq s \leq j} \{minf(i, j, s)\}, 1 \leq i, j \leq n$$

- $$m[i, j, 1] = \max_{1 \leq s \leq j} \{maxf(i, j, s)\}, 1 \leq i, j \leq n$$

- 初始边界值为 $m[i, 1, 0]=v[i]$ $m[i, 1, 1]=v[i]$ $1 \leq i \leq n$

- 多边形是封闭的, 当 $i+s>n$ 时, 顶点 $i+s$ 实际编号为 $(i+s)\%n$ 。
 $m[i, n, 1]$ 记为游戏首次删除第 i 条边后得到的最大得分。

多边形游戏样例



$$m(1,1,1) = 9$$

$$m(1,1,0) = 9$$

$$m(2,1,1) = -4$$

$$m(2,1,0) = -4$$

$$m(3,1,1) = 9$$

$$m(3,1,0) = 9$$

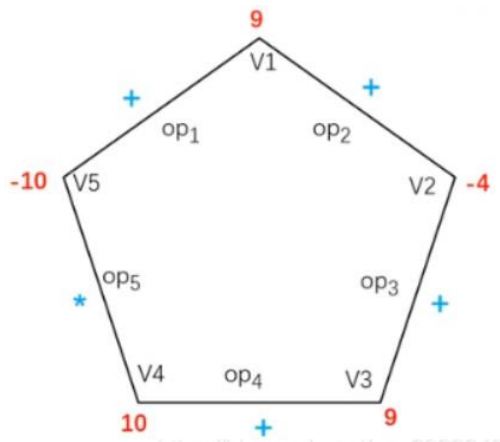
$$m(4,1,1) = 10$$

$$m(4,1,0) = 10$$

$$m(5,1,1) = -10$$

$$m(5,1,0) = -10$$

多边形游戏样例



$$m(1,2) = m(1,1)op(2)m(2,1); op(2) = "+"$$

$$m(1,2,1) = m(1,1,1) + m(2,1,1) = 5$$

$$m(1,2,0) = m(1,1,0) + m(2,1,0) = 5$$

$$m(2,2) = m(2,1)op(3)m(3,1); op(3) = "+"$$

$$m(2,2,1) = m(2,1,1) + m(3,1,1) = 5$$

$$m(2,2,0) = m(2,1,0) + m(3,1,0) = 5$$

$$m(3,2) = m(3,1)op(4)m(4,1); op(4) = "+"$$

$$m(3,2,1) = m(3,1,1) + m(4,1,1) = 19$$

$$m(3,2,0) = m(3,1,0) + m(4,1,0) = 19$$

$$m(4,2) = m(4,1)op(5)m(5,1); op(5) = "*"$$

$$m(4,2,1) = \max\{m(4,1,1)*m(5,1,1), m(4,1,1)*m(5,1,0), m(4,1,0)*m(5,1,1), m(4,1,0)*m(5,1,0)\} = -100$$

$$m(4,2,0) = \min\{m(4,1,1)*m(5,1,1), m(4,1,1)*m(5,1,0), m(4,1,0)*m(5,1,1), m(4,1,0)*m(5,1,0)\} = -100$$

$$m(5,2) = m(5,1)op(1)m(1,1)$$

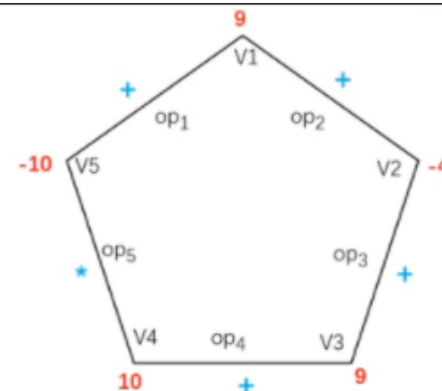
$$m(5,2,1) = m(5,1,1) + m(1,1,1) = -1$$

$$m(5,2,0) = m(5,1,0) + m(1,1,0) = -1$$

$$\max = \max\{m(1,5,1), m(2,5,1), m(3,5,1), m(4,5,1), m(5,5,1)\} = 40$$

$$\min = \min\{m(1,5,0), m(2,5,0), m(3,5,0), m(4,5,0), m(5,5,0)\} = -240$$

	1	2	3	4	5
1	9	5	14	24	-86
2	-4	5	15	-95	-5
3	9	19	-91	-1	-5
4	10	-100	-10	-14	40
5	-10	-1	-5	4	14



所以
 $\max = 40$
 $\min = -240$

练习题：股票买卖问题

- 给定一个数组 `prices`，它的第 i 个元素 `prices[i]` 表示一支给定股票第 i 天的价格。
- 你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票。设计一个算法来计算你能获取的最大利润。
- 返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0。

练习题：股票买卖问题

- 输入：[7,1,5,3,6,4]
 - 输出：5
 - 解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = $6 - 1 = 5$ 。
 - 注意利润不能是 $7 - 1 = 6$ ，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。
-
- 输入：prices = [7,6,4,3,1]
 - 输出：0
 - 解释：在这种情况下，没有交易完成，所以最大利润为 0。

练习题：股票买卖问题

```
public class Solution {  
    public int maxProfit(int[] prices) {  
        int len = prices.length;  
        if (len < 2) {  
            return 0;  
        }  
        // 有可能不发生交易，因此结果集的初始值设置为 0  
        int res = 0;  
        // 枚举所有发生一次交易的股价差  
        for (int i = 0; i < len - 1; i++) {  
            for (int j = i + 1; j < len; j++) {  
                res = Math.max(res, prices[j] - prices[i]);  
            }  
        }  
        return res;  
    }  
}
```

暴力求解 $O(n^2)$

练习题：股票买卖问题(动态规划)

买卖股票有约束，根据题目意思，有以下两个约束条件：

条件 1：你不能在买入股票前卖出股票；

条件 2：最多只允许完成一笔交易。

因此 当天是否持股 是一个很重要的因素，而当前是否持股和昨天是否持股有关系，为此我们需要把 是否持股 设计到状态数组中。

状态定义：

$dp[i][j]$ ：下标为 i 这一天结束的时候，手上持股状态为 j 时，我们持有的现金数。换种说法： $dp[i][j]$ 表示天数 $[0, i]$ 区间里，下标 i 这一天状态为 j 的时候能够获得的最大利润。其中：

$j = 0$ ，表示当前不持股；

$j = 1$ ，表示当前持股。

注意：下标为 i 的这天的计算结果包含了区间 $[0, i]$ 所有的信息，因此最后输出 $dp[len - 1][0]$ 。

练习题：股票买卖问题(动态规划)

- 使用「现金数」这个说法主要是为了体现 买入股票手上的现金数减少，卖出股票手上的现金数增加；
- 「现金数」等价于题目中说的「利润」，即先买入这只股票，后买入这只股票的差价；
- 因此在刚开始的时候，我们的手上肯定是有了一定现金数能够买入这只股票，即刚开始的时候现金数肯定不为 0，但是写代码的时候可以设置为 0。极端情况下（股价数组为 [5, 4, 3, 2, 1]），此时不发生交易是最好的。

推导状态转移方程：

$dp[i][0]$ ：规定了今天不持股，有以下两种情况：

- 昨天不持股，今天什么都不做；
- 昨天持股，今天卖出股票（现金数增加），

$dp[i][1]$ ：规定了今天持股，有以下两种情况：

- 昨天持股，今天什么都不做（现金数与昨天一样）；
- 昨天不持股，今天买入股票（注意：只允许交易一次，因此手上的现金数就是当天的股价的相反数）。

练习题：股票买卖问题(动态规划)

知识点：

- 多阶段决策问题：动态规划常用于求解多阶段决策问题；
- 无后效性：每一天是否持股设计成状态变量的一维。状态设置具体，推导状态转移方程方便。

Solution of MDP

