动态规划

笔记源文件：Markdown，长图，PDF，HTML

1. 动态规划概述

> 1 核心思想：记表备查
>
> 1. 保存已解决的子问题的答案，而在需要时再找出已求得的答案
> 2. 避免大量重复计算，得到多项式时间算法
>
> 2 和分治法的区别
>
> 1. 相同：都是划分为子问题
> 2. 不同：分治的子问题是独立的，动态规划的子问题有重叠
>
> 3 解题步骤
>
> 1. 找出最优解的性质，并刻划其结构特征
> 2. 递归地定义最优值
> 3. 以自底向上的方式计算出最优值
> 4. 根据计算最优值时得到的信息，构造最优解
>
> 4 基本要素
>
> 1. 最优子结构：通常采用反证法来证明某个结构是子结构，全局最优⊆全局最优
> 2. 重叠子问题
>
> 5 两种基本形态：
>
> 1. 动态规划：记表备查，自下而上
> 2. 备忘录方法：动态规划变形，但是自上而下
>
> ⚠️
>
> 当所有子问题都需要至少解一次时，用动态规划更好
>
> 当子问题空间中的部分子问题可不必求解时，用备忘录方法好

2. 拆分类问题

2.1. 矩阵连乘$\{A_1, A_2, ..., A_n\}$

2.1.1. 矩阵计算次序

1 矩阵完全加括号：单个矩阵完全加括号，全加括号的矩阵相乘得全加括号的矩阵

2 定义：加括号方式$\xleftrightarrow{一一对应}$矩阵计算次序$\xrightarrow{密切影响}$计算量

例如$ABCD$，可以为$A(B(CD))$，$A((BC)D)$，$(AB)(CD)$，$(A(BC))D$，$((AB)C)D$

3 如何找到计算量最小的计算次序？穷举法or动态规划

2.1.2. 动态规划法

1 基本方法

1. 记$A[i:j]=A_iA_{i+1}\dots A_j$记所需最少乘次数为$m[i,j]$
2. 可将$A[i:j]$计算量分解为：$A[i:k]$计算量，$A[k+1:j]$计算量，二者乘积的计算量

2 分析最优结构

1. 计算$A[i:j]$的最有结构，包括计算$A[i:k]$和$A[k+1:j]$的最优结构
2. 最优子结构性质：动态规划的显著特征，即最优解包含着其子问题的最优解

3 建立递归关系

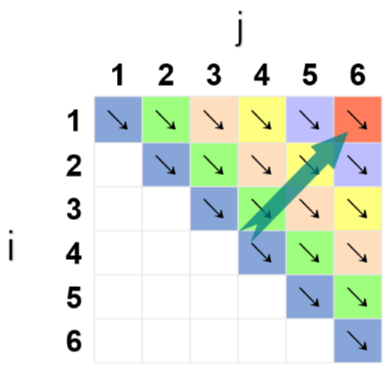
1. 假设A_1是$p_{i-1}\times p_i$矩阵，则$A[i:k]$是$p_{i-1}\times p_k$维，$A[k+1:j]$是$p_k\times p_j$维
2. $m[i:j]$可以分解为三部分
   - $m[i,k]$和$m[k+1,j]$
   - 还有$A[i:k]$和$A[k+1:j]$相乘的计算量，即$p_{i-1}\times p_k\times p_j$
3. 注意k游走在$j\to i$之间，有$j-i$种取值，所以得到以下

$$m[i,j]=\begin{cases}0 & \text{if } i=j\\ \min_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{if } i<j\end{cases}$$

当$k=i$时，$m[i,j]=m[i+1,j]+p_{i-1}p_ip_j$

4 计算最优值

1. 子问题总数：$1\le i\le j\le n$，要选取i,j并去掉一半重复的，故为$C_n^2/2$，可见递归是许多子问题被重算过
2. 动态规划：自下而上递归，保存已解决问题(再遇到时只简答检查)，避免重复计算



3. 代码实现：算法时空复杂度都为$O(n^3)$

```
void matrixMutiply(int[] p, int n, int[][] m, int[][] s)
{
    /*这里对应的是对角线上，所有i=j的情况，直接等于0就行了*/
    for (int i = 1; i <= n; i++) {m[i][i] = 0;}

    /*开始计算对角线以上*/
    //r表示连续乘积中矩阵的个数，它从2开始，因为长为1的情况已经考虑过了
    for (int r = 2; r <= n; r++)
    {
        /*i是子链起始矩阵索引，j是子链终止矩阵索引*/
        for (int i = 1; i <= n - r + 1; i++)
        {
```

```
            int j = r + i - 1;
            /*先强行初始化m[i][j],此时k=i,也就是分割为(Ai)+(Ai+1...Aj)*/
            m[i][j] = m[i + 1][j] + p[i - 1] * p[i] * p[j];
            s[i][j] = i;//记录分割点
            /*在ij之间移动k, 在k=i+1到k=j-1循环, 找到m[i][j]的最小值*/
            for (int k = i + 1; k < j; k++) //k为分割点
            {
                int temp = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] *
p[j];
                if (temp < m[i][j])
                {
                    m[i][j] = temp;
                    s[i][j] = k;
                }
            }
            //循环结束后得到的m[i][j]就是AiAi+1...Aj的最小乘积次数
            //更新i
        }
        //更新r
    }
}
```

最终`m[1][n]`的值将是矩阵链乘的最小乘法次数，然后得到最优时的分割点`s[i][j]`

5 构造最优解：$A[1:n]$的最优加括号方式为$A[1:s[1][n]]*A[s[1][n]+1:n]$

2.1.3. 备忘录方法

为每个**解过的子问题**建立了备忘，避免了相同子问题的重复求解

```
int MemoizedMatrixChain(int []p,int n,int [][]m,int [][]s)
{
 for(int i=1;i<=n;i++)
     for(int j=i;j<=n;j++)
         m[i][j]=0;
 //0表示相应的子问题还未被计算
 return LookupChain(1,n,p,m,s);
}
int LookupChain(int i,int j,int[]p,int[][]m,int [][]s)
{
 if(m[i][j]>0) //大于0表示其中存储的是所要求子问题的计算结果
     return m[i][j];  //直接返回此结果即可
 if(i==j)
     return 0;
 int u=LookupChain(i,i,p,m,s)+LookupChain(i+1,j,p,m,s)+p[i-
1]*p[i]*p[j];
 s[i][j]=i;
 for(int k=i+1;k<j;k++)
 {
     int t=LookupChain(i,k,p,m,s)+LookupChain(k+1,j,p,m,s)+p[i-
1]*p[k]*p[j];
     if(t<u)
     {
         u=t;
         s[i][j]=k;
```

```
        }
    }
    m[i][j]=u;
    return u;
}
```

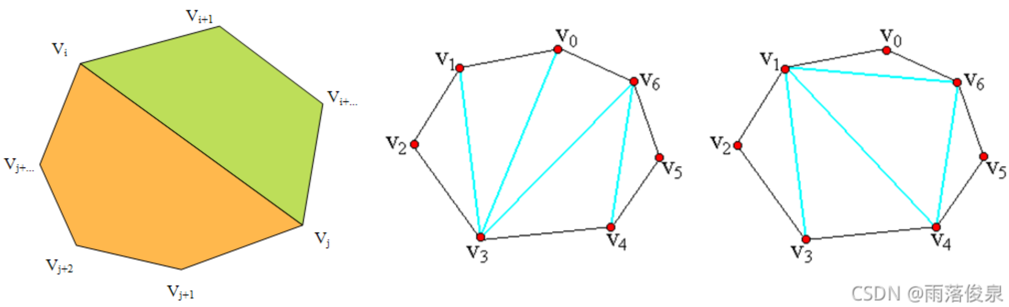
2.2. 凸多边形最优三角剖分

2.3.1. 问题描述

1 多边形与边的表示

1. $P=\{v_0,v_1,\ldots,v_{n-1}\}$表示具有$n$条边的凸多边形
2. 若v_i与v_j是多边形上不相邻的2个顶点，则线段v_iv_j称为多边形的一条弦
3. v_iv_j将原多边形分为：$\{v_i,v_{i+1},\ldots,v_j\}$和$\{v_j,v_{j+1},\ldots,v_i\}$

2 多边形的三角剖分：将多边形分割成三角形的弦集合T，T中弦互不相交，切T要达到最大



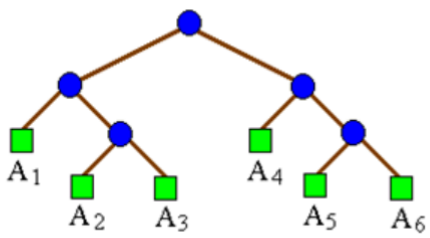
n顶点的凸多边形，恰有$n-3$条弦，和$n-2$个三角形

3 问题描述：给所有分割得到的三角形一共权函数w_i，要求确定一共剖分，使得$\sum w_i$最小

2.3.2. 语法树&三角剖分的结构

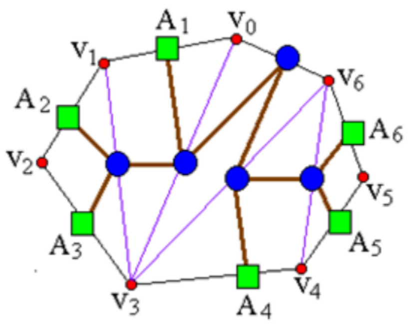
1 语法树的引入：

1. 一个表达式的完全加括号方式相应于一棵完全二叉树，即语法树
2. 示例：$((A1(A2A3))(A4(A5A6)))$



2 用语法树表示变形$P=\{v_0,v_1,\ldots,v_{n-1}\}$的三角剖分

1. 矩阵连乘中每个矩阵$A_i \iff$ 多边形的每条边$v_{i-1}v_i$
2. 矩阵连乘积$A[i+1:j] \iff$ 三角剖分中的一条弦v_iv_j，注意$i<j$



3 最优三角剖分$\overset{对应}{\longleftrightarrow}$矩阵链最优完全加括号

	最优三角剖分	矩阵链最优完全加括号
对应项1	$A_1A_2A_3A_4A_5A_6$	$P=\{v_0,v_1,v_2,v_3,v_4,v_5,v_6\}$
对应项2	A_i的维数为$p_{i-1}\times p_i$	$v_iv_jv_k$上的权函数值为$w(v_iv_jv_k)=p_ip_jp_k$

2.3.3. 最优子结构性质

1 设$P=\{v_0,v_1,...,v_{n-1}\}$的最优剖分为$T(n)$，其必定包含三角形$v_0v_kv_{n-1}$

2 剖分$T(n)$的权可分为：

1. 子多边形$\{v_0,v_1,...,v_k\}$和$\{v_k,v_{k+1},...,v_n\}$的权和
2. 三角形$v_0v_kv_{n-1}$的权

注意全局最优，局部也会是最优的

2.3.4. 最优三角形剖分的递归结构

1 $t[i][j]$：

1. 为子多边形$\{v_i,v_{i+1},...,v_j\}$的最优三角剖分所对应的权函数值，即最优值
2. $t[i-1][i]=0$
3. $P=\{v_0,v_1,...,v_{n-1}\}$的最优值为$t[1][n]$

2 将$\{v_i,v_{i+1},...,v_j\}$分为：$\{v_i,v_{i+1},...,v_k\}$和$\{v_{k+1},v_{k+2},...,v_j\}$，还有三角形$v_{i-1}v_kv_j$

$$t[i][j]=\begin{cases}0 & \text{if } i=j\\ \min\limits_{i\le k<j}\{t[i][k]+t[k+1][j]+w(v_{i-1},v_k)\} & \text{if } i<j\end{cases}$$

2.3.5. 计算最优值+构造最优三角形剖分

```
template<typename Type,typename E>
void MinWeightTriangulation(E *p,int n,Type **t,int **s)
{
 for(int i=1; i<=n; i++)
     t[i][i]=0;
 for(int r=2; r<=n; r++)
     for(int i=1; i<=n-r+1; i++)
     {
         int j=i+r-1;
         t[i][j]=t[i+1][j]+w(p,i-1,i,j);
```

```
            s[i][j]=i;  //k=i
            for(int k=i+1; k<i+r-1; k++)
            {
                int u=t[i][k]+t[k+1][j]+w(p,i-1,k,j);
                if(u<t[i][j])
                {
                    t[i][j]=u;
                    s[i][j]=k;
                }
            }
        }
}
```

$s[i][j]$记录了的顶点$+v_{i-1}+v_j$，一起构成三角形的第3个顶点的位置，据此可构造出最优三角剖分中所有三角形

```
template<typename E>
void Traceback(E *p,int i,int j,int **s)
{
 if(i==j)
     return;
 int k=s[i][j];
 cout<<p[i-1].v<<" "<<p[k].v<<" "<<p[j].v<<endl;
 Traceback(p,i,k,s);
 Traceback(p,k+1,j,s);
}
```

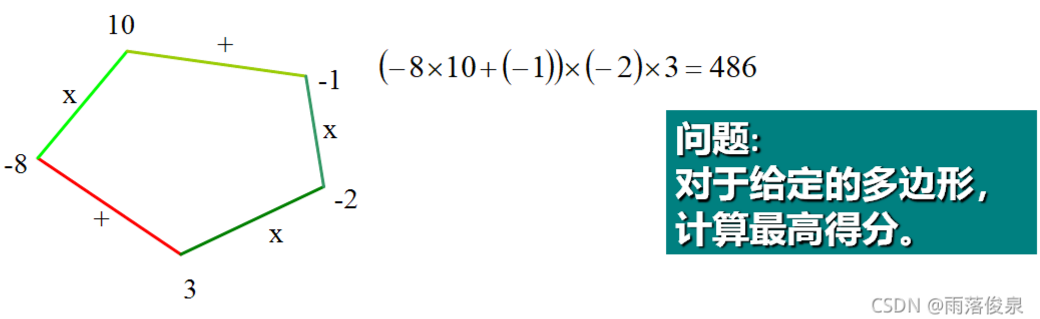
2.3. 多边形游戏

1 模型：多边形有n条边编号为$1 \to n$，每个顶点赋予一个整数，每条边赋予算符$+$或者$*$

2 计算规则：先删除一条边，然后循环执行以下操作

1. 选择边E，和E连接的两个顶点v_1, v_2
2. 用新顶点v取代E, v_1, v_2，并且$v = v_1 \xleftrightarrow{E中的运算} v_2$

3 问题：最终所有边都会被删除，游戏得分为所剩顶点值，如何的让游戏的分最高？



2.4. 公园游艇问题

1 问题描述

1. 公园上有n个游艇出租点，游客可在i接船，j还船，所需租金为$r(i,j)$

r(i,j)	2	3	4	5
1	13	15	24	44
2		16	18	8
3			7	26
4				12

2. 试设计一个算法，计算出从第一站→站n站所需最少租金

2 最优子结构+递归关系

1. 设$r(i,j)$为$i \xrightarrow{直达} j$的租金，$m(i,j)$为$i \xrightarrow{不论怎样只要到达就行} j$的租金
2. $m(i,j)$的最优解一定包含$m(i,k)$和$m(k,j)$的最优解
3. 建立递归

$$m[i][j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i][k] + m[k][j]\} & \text{if } i < j \end{cases}$$

3 计算最优+构造最优解

```
void cent(int[][] m, int n, int[][] s) {
    for (int i = 1; i <=n ; i++) {
        m[i][i]=0;
    }
    for (int r = 2; r <= n; r++)
        for (int i = 1; i <= n - r + 1; i++) {
            int j = i + r - 1;
            s[i][j] = i;
            for (int k = i; k <= j; k++) {
                int temp = m[i][k] + m[k][j];
                if (temp < m[i][j]) {
                    m[i][j] = temp;
                    s[i][j] = k;//在第k站下
                }
            }
        }
}
void traceBack(int i, int j, int[][] s) {
    if (i == j) {
        System.out.print(i);
        return;
    }
    System.out.print("[");
    traceBack(i, s[i][j], s);
    traceBack(s[i][j] + 1, j, s);
    System.out.print("]");
}
```

3. 移界类问题

3.1. 一维移界：最大子段和

1 问题描述：给定n个正负整数组成的序列，$a_1, a_2, ..., a_n$，求该序列子段和的最大值，注意当所有整数均为负数时定义最大子段和为0，即max $\left\{0, \max\limits_{1 \le i \le j \le n} \sum\limits_{k=i}^{j} a_k\right\}$

2 暴力破解：复杂度$T(n) = O(n^3)$

```
int MaxSum1(int n,int* a,int& besti,int& bestj){
int sum=0;
for(int i=1;i<=n;i++)      //注意：a[0]不用
{
for(int j=i;j<=n;j++) //框定i,j，在二者之间通过k游走，从而求和
{
    int thissum=0;
    for(int k=i;k<=j;k++)
        thissum+=a[k];
    if(thissum>sum)
    {
        sum=thissum;
        besti=i;
        bestj=j;
    }
}
return sum;
}
}
```

3 算法改进：注意到$\sum\limits_{k=i}^{j} a_k = a_j + \sum\limits_{k=i}^{j-1} a_k$，复杂度$T(n) = O(n^2)$

```
int MaxSum2(int n,int *a,int &besti,int &bestj){
int sum=0;
for(int i=1;i<=n;i++)
{
int thissum=0;
for(int j=i;j<=n;j++)
{
    thissum+=a[j];
    if(thissum>sum)
    {
        sum=thissum;
        besti=i;
        bestj=j;
    }
}
}
return sum;
}
```

4 分治算法

1. 将序列$a[1:n]$分为从长度相等的两段$a[1:\frac{n}{2}]$+$a[1:\frac{n}{2}-1]$，所以原序列的最大子段可能：
    - 与$a[1:\frac{n}{2}]$的最大子段相同
    - 与$a[\frac{n}{2}+1:n]$的最大子段相同
    - 横跨两段，为 $\max\limits_{1\leq i\leq n/2}\sum\limits_{k=i}^{n/2}a[k]+\max\limits_{n/2+1\leq j\leq n}\sum\limits_{k=n/2+1}^{j}a[k]$
2. 代码实现

```java
public static int soveByDivide(int[] a, int left, int right) {
    int sum = 0; // 初始化子数组的和为0
    if (left == right) { // 如果子数组只包含一个元素
        sum = a[left] > 0 ? a[left] : 0; // 如果该元素是正数，返回其值；否则返回0
    } else {
        int middle = (left + right) / 2; // 计算中点
        // 递归求解左半部分的最大子数组和
        int leftSum = soveByDivide(a, left, middle);
        // 递归求解右半部分的最大子数组和
        int rightSum = soveByDivide(a, middle + 1, right);

        int s1 = 0; // 初始化横跨两个子数组的最大子数组和的左半部分
        int lefts = 0; // 用于计算横跨左半边的连续子数组的临时和
        // 从中间向左扫描，寻找最大子数组和的左半部分
        for (int i = middle; i >= left; i--) {
            lefts += a[i]; // 累加当前元素
            if (lefts > s1) // 如果当前累加和大于已知的最大和
                s1 = lefts; // 更新最大和
        }

        int s2 = 0; // 初始化横跨两个子数组的最大子数组和的右半部分
        int rights = 0; // 用于计算横跨右半边的连续子数组的临时和
        // 从中间向右扫描，寻找最大子数组和的右半部分
        for (int j = middle + 1; j <= right; j++) {
            rights += a[j]; // 累加当前元素
            if (rights > s2) // 如果当前累加和大于已知的最大和
                s2 = rights; // 更新最大和
        }

        sum = s1 + s2; // 计算横跨中点的子数组的最大和

        // 比较左边、右边和横跨中点的子数组的和，取最大值
        if (sum < leftSum)
            sum = leftSum; // 如果左边子数组的和更大，更新sum
        if (sum < rightSum)
            sum = rightSum; // 如果右边子数组的和更大，更新sum
    }
    return sum; // 返回最大子数组和
}
```

3. 复杂度分析：$T(n)=O(n\log n)$

$$T(n)=\begin{cases}O(1) & \text{if } n\le c\\ 2T\left(\frac{n}{2}\right)+O(n) & \text{if } n>c\end{cases}$$

5 动态规划

1. 定义$b[j]=\max\limits_{1\le i\le j}\sum\limits_{k=i}^{j}a[k]\quad 1\le j\le n$，即前$j$个元素的子数组中的最大子段和

   则有$\max\limits_{1\le i\le j\le n}\sum\limits_{k=i}^{j}a[k]=\max\limits_{1\le j\le n}\max\limits_{1\le i\le j}\sum\limits_{k=i}^{j}a[k]=\max\limits_{1\le j\le n}b[j]$

2. 递推关系分析
   - $b[j-1]>0$时，$b[j]=b[j-1]+a[j]$
   - $b[j-1]<0$时$b[j]=a[j]$

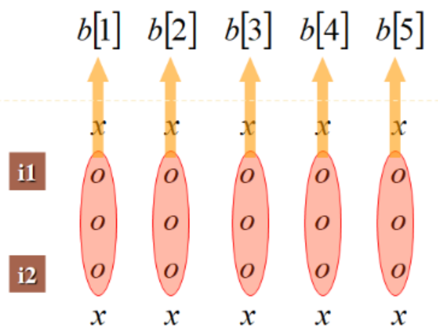
   所以$b[j]=\max\{b[j-1]+a[j],a[j]\},\quad 1\le j\le n$

3. 代码实现

```
public static int solveByDP(int[]a){
    int sum=0,b=0;
    for (int i = 0; i <a.length ; i++) {
        if (b>0)
            b+=a[i];
        else
            b=a[i];
        if (b>sum)
            sum=b;
    }
    return sum;
}
```

6 最大子段和问题推广：最大子矩阵和问题，给定一个A_{m*n}，试求A的一个子矩阵，使其各元素之和为最大

$$设b[j]=\sum_{i=i1}^{i2}a[i][j]$$



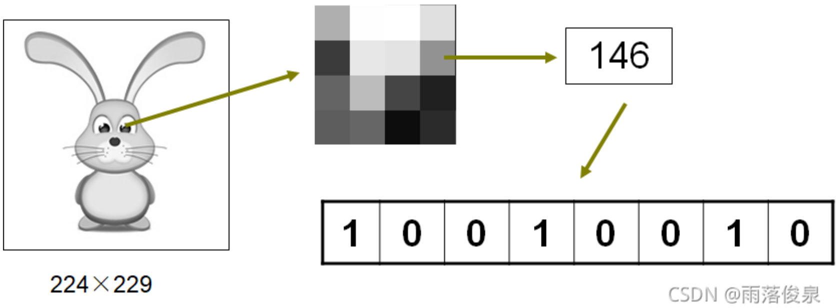

```
public static int MaxSum2(int m,int n,int[][]a){
    int sum=0;
    int[]b=new int[n];
    for(int i=0;i<m;i++){  //从第i行
        for(int k=0;k<n;k++)  //初始化数组b
            b[k]=0;
        for(int j=i;j<m;j++){  //到第j行
            for(int k=0;k<n;k++)
                b[k]+=a[j][k];//按列取值
```

```
                int max=solveByDP(b);
                if(max>sum)
                    sum=max;
            }
        }
        return sum;
    }
```

3.2. 一维：图像压缩

3.2.1. 图像压缩概述

1 计算机图像表示：用灰度值序列$\{p_1,p_2,...,p_n\}$表示图像，p_i表示像素点i的灰度值，灰度值用8位二进制数表示



2 计算机图像压缩

1. 将所有像素序列$\{p_1,p_2,...,p_n\}$分为**m个连续段**$\{S_1,S_2,...,S_m\}$，S_i为第i个像素段
2. 每个像素段S_i包含有$l[i]$个像素，S_i段中每个像素的位数为$b[i]$

3 像素序列的存储空间：以$1\le l[i]\le 255$，$0\le b[i]\le 7$为例

1. S_i像素段所占空间：
   - S_i像素段含有$l[i]$个像素，一共$l[i]*b[i]$位
   - 指定S_i中的一位需要8位控制信息，再指定这一位的灰度需要3位控制信息，故一共11位
2. 所有像素段所占信息：$\{\sum_{i=1}^{m} l[i]*b[i]\}+11*m$

3.2.2. 问题描述与解决

确定$\{p_1,p_2,...,p_n\}$的最优分段，使得依此分段所需的存储空间最小

1 满足最优子结构性质：设$l[i],b[i],1\le i\le m$是$\{p_1,p_2,...,p_n\}$最优分段，那么

1. $l[1],b[1]$是$\{p_1,p_2,...,p_{l[1]}\}$最优分段
2. $l[i],b[i],2\le i\le m$是$\{p_{l[1]},p_2,...,p_n\}$最优分段

2 递归计算最优值

$$s[i+k]=\min_{1\le k\le \min(i+k,256)}\{s[i]+k\cdot b_{\max}(i+1,i+k)\}+11$$

其中$b_{\max}(i,j)=\left\lceil \log\left(\max_{i\le k\le j}\{p_k\}+1\right)\right\rceil$

1. $s[i]$：前i个像素$p_0\to p_i$所需最小存储空间

2. $s[i+k]$：在原来前i个像素基础上增加k个像素
   - 保留原有的$s[i]$不动
   - $b_{max}(i+1, i+k)$是$(i+1) \to (i+k)$子段的最大灰度值的位数，在乘以k便是所新增子段的所占空间

3.3. 二维：最长公共子序列

1 什么是最长公共子序列：如下例子

$X = \{A, B, C, B, D, A, B\}$，$Y = \{B, D, C, A, B, A\}$，最长公共子序列为$\{B, C, A\}$

2 最长公共子序列的结构：

$X_m = \{x_1, x_2, \ldots, x_m\}$，$Y_n = \{y_1, y_2, \ldots, y_n\}$，最长公共子序列为 $Z_k = \{z_1, z_2, \ldots, z_k\}$

条件	结论
1.$x_m = y_n$	$z_k = x_m = y_n$，且Z_{k-1}是X_{m-1}和Y_{n-1}的最长公共子序列
2.$x_m \neq y_n$且$x_m \neq z_k$	Z_k是X_{m-1}和Y_n的最长公共子序列
3.$x_m \neq y_n$且$y_n \neq z_k$	Z_k是X_m和Y_{n-1}的最长公共子序列

3 子问题递归结构

1. $X_i = \{x_1, x_2, \ldots, x_i\}$，$Y_j = \{y_1, y_2, \ldots, y_j\}$，$c[i][j]$记录了最长公共子序列长度
2. 递归关系为：

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

- $c[i][0] = 0, c[0][j] = 0$，一个序列啥都没有的时候，自然公共子序列也啥都没有
- $x_i = y_j$时，X_{i-1}, Y_{i-1}必定含有一个长为$c[i][j] - 1$的公共子序列
- $x_i \neq y_j$时，现有最长公共子序列，必定是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，二者之一。为得到最长，所以二者取其大

4 计算最优值：算法耗时$O(mn)$

1. 输入两个序列x, y长分别为mn
2. $c[i][j]$存储$x[1:i], y[1:j]$的最长公共子序列长
3. $b[i][j] = 1/2/3$，表示$c[i][j]$的值是1/2/3哪个条件得到的

```
public  static void LCSLength(char[] x, char[] y, int[][] c, int[][] b)
{
    int m = x.length-1;
    int n = y.length-1;
    /*第一个条件*/
    for (int i = 1; i <= m; i++) {c[i][0] = 0;}
    for (int j = 1; j <= n; j++) {c[0][j] = 0;}
    /*其余条件*/
    for (int i = 1; i <= m; i++)
    {
```

```
        for (int j = 1; j <= n; j++)
        {
            //Xi和Yi的最长公共子序列是由Xi-1和Yi-1的最长公共子序列在尾部加上xi所得到
            if (x[i] == y[j])
            {
                c[i][j] = c[i - 1][j - 1] + 1;
                b[i][j] = 1;//表示通过情况1找到的
            }
            //Xi和Yi的最长公共子序列与Xi-1和Yi的最长公共子序列相同
            else if (c[i - 1][j] >= c[i][j - 1])
            {
                c[i][j] = c[i - 1][j];
                b[i][j] = 2;//表示通过情况2找到的
            }
            //Xi和Yi的最长公共子序列与Xi和Yj-1的最长公共子序列相同
            else
            {
                c[i][j] = c[i][j - 1];
                b[i][j] = 3;//表示通过情况3找到的
            }
        }
    }
}
```

5 构造最长公共子序列

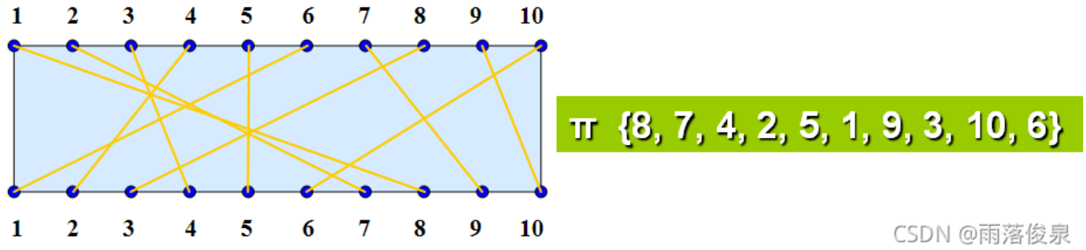
```
// 递归函数用于找到并打印最长公共子序列(LCS)
public static void LCS(int m, int n, char[] x, int[][] b) {
    // 基本情况：如果任一序列的长度为0，则无需进一步操作
    if (m == 0 || n == 0) {return;}

    // 检查b数组在位置[m][n]的值来确定当前步骤
    if (b[m][n] == 1) {
        // b[m][n]为1表示 x[m] 和 y[n] 匹配，是LCS的一部分
        // 在添加x[m]到LCS之前，先递归处理剩余的子序列
        LCS(m - 1, n - 1, x, b);
        // 递归完成后，打印当前匹配的字符
        System.out.print(x[m]);
    } else if (b[m][n] == 2) {
        // b[m][n]为2表示忽略x[m]，继续检查x序列的前一元素
        LCS(m - 1, n, x, b);
    } else {
        // b[m][n]为3表示忽略y[n]，继续检查y序列的前一元素
        LCS(m, n - 1, x, b);
    }
}
```

3.4. 二维：电路布线

1 问题描述：

1. 要求用导线$[i, \pi(i)]$将上下两排柱线连接，例如$[1, \pi(1) = 8]$表示将上1下8相连



2. 目的：不让导线相交的前提下，尽可能多地安排导线

2 递归获得

1. $Size(i,j)$表示 考虑上边$1 \to i$和下边$1 \to j$时，能够形成的最大不相交连线集合的大小
2. 当$i = 1$时

$$Size(1,j) = \begin{cases} 0 & j < \pi(1) \\ 1 & j \geq \pi(1) \end{cases}$$

3. 当$i > 1$时

$$Size(i,j) = \begin{cases} \text{Size}(i-1,j) & j < \pi(i) \\ \max\{\text{Size}(i-1,j), \text{Size}(i-1,\pi(i)-1)+1\} & j \geq \pi(i) \end{cases}$$

3.5. 背包0-1问题，背下来

1 问题描述

1. 给定n种物品+一个背包，背包容量为C，物品i重量和价格分别为w_i和v_i
2. 应如何选择装入背包的物品，使得装入背包中物品的总价值最大

$$\max \sum_{i=1}^{n} v_i x_i$$
$$\text{s.t.} \begin{cases} \sum\limits_{i=1}^{n} w_i x_i \leq C \\ x_i \in \{0,1\}, 1 \leq i \leq n \end{cases}$$

2 最优子结构：如何证明$(y_1, y_2, \ldots, y_n)$，是所给问题的一个最优解

1. 若$(y_1, y_2, \ldots, y_n)$是最优解，那么$(y_2, y_3 \ldots, y_n)$也应是相应子问题的最优解，满足以下

$$\max \sum_{i=2}^{n} v_i x_i$$
$$\text{s.t.} \begin{cases} \sum\limits_{i=2}^{n} w_i x_i \leq C - w_1 y_1 \\ x_i \in \{0,1\}, 2 \leq i \leq n \end{cases}$$

2. 若$(y_1, y_2, \ldots, y_n)$非最优解，那么假设$(z_2, z_3, \ldots, z_n)$是上述问题的最优解，则有：

$$\sum_{i=2}^{n} v_i z_i \geq \sum_{i=2}^{n} v_i y_i \quad \& \quad w_1 y_1 + \sum_{i=2}^{n} w_i z_i \leq C$$

3. 以上公式整理可得如下，则这又说明了$(y_1, z_2, \ldots, z_n)$是问题的一个更优解，矛盾

$$v_1y_1 + \sum_{i=2}^{n} v_iz_i \geq \sum_{i=1}^{n} v_iy_i$$

$$w_1y_1 + \sum_{i=2}^{n} w_iz_i \leq C$$

3 递归关系：令$m(i,j)$表示背包容量为j，可选择物品为$i, i+1, \dots, n$的最优值

不放物品i

放入物品i

$$m(i,j)=\begin{cases}\max\{m(i+1,j), m(i+1,j-w_i)+v_i\} & j \geq w_i \\ m(i+1,j) & 0 \leq j < w_i\end{cases}$$

$$m(n,j)=\begin{cases}v_n & j \geq w_n \\ 0 & 0 \leq j \leq w_n\end{cases}$$

只有物品n

4. 图形类：多段有向图

4.1. 什么是多段有向图

满足以下条件

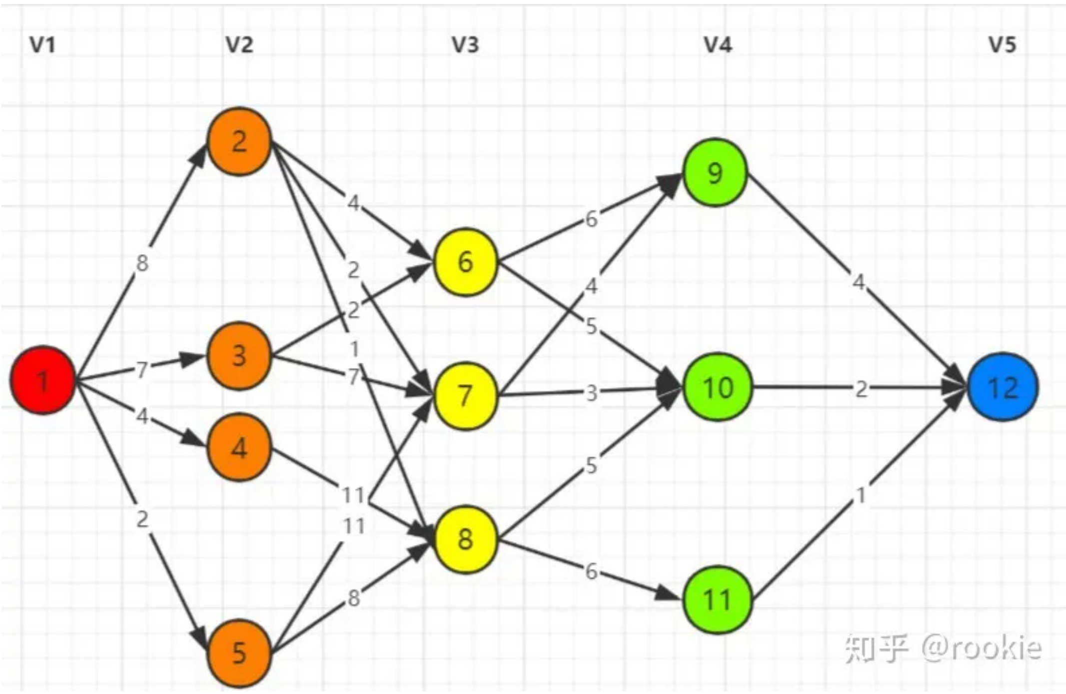
1 是一个带权有向图并且无环

2 有且仅有一个起点S和终点T

3 有n个阶段，每个阶段由特定的几个结点构成

4 每个结点都只能指向相邻阶段的点

示例：图中的节点被划为5个不相交的集合V_1, V_2, V_3, V_4, V_5，其中要求有V_1, V_5只能有一个结点



4.2. 求S到T的最小成本路径

> **1** 有关数据结构：
>
> 1. `cost[i]`：以i为起点，到终点T的距离
> 2. `d[i]`：记录从i到终点T最短路径中，所出现的结点
>
> **2** 向前处理的算法
>
> 1. 从最后一个节点开始，从后向前，依次计算该层中每个结点的`cost`值+`d`值
> 2. 一直计算到了开始节点时，根据`d`得到最短路径
>
> **3** 示例
>
> | 结点 | 结点所在层 | 结点cost[]值 | 结点d[]值 |
> | --- | --- | --- | --- |
> | 12 | v_5 | 0 | \ |
> | 11 | v_4 | $min\{c(11,12)+cost[12]\}=min\{1+0\}=1$ | 12 |
> | 10 | v_4 | $min\{c(10,12)+cost[12]\}=min\{2+0\}=2$ | 12 |
> | 9 | v_4 | $min\{c(9,12)+cost[12]\}=min\{4+0\}=4$ | 12 |
> | 8 | v_3 | $min\{c(8,11)+cost[11],c(8,10)+cost[10]\}=min\{7,7\}=7$ | 10 |
> | 7 | v_3 | $min\{c(7,10)+cost[10],c(7,9)+cost[9]\}=min\{5,8\}=7$ | 10 |
> | 6 | v_3 | $min\{c(6,10)+cost[10],c(6,9)+cost[9]\}=min\{7,10\}=7$ | 10 |
> | 5 | v_2 | $min\{c(5,8)+cost[8],c(5,7)+cost[7]\}=min\{15,16\}=15$ | 8 |
> | 4 | v_2 | $min\{c(4,8)+cost[8]\}=min\{11+7\}=18$ | 8 |
> | 3 | v_2 | $min\{c(3,7)+cost[7],c(3,6)+cost[6]\}=min\{12,9\}=9$ | 6 |
> | 2 | v_2 | $min\{c(2,8)+cost[8],c(2,7)+cost[7],c(2,6)+cost[6]\}=7$ | 7 |
> | 1 | v_1 | $min\{c(1,5)+cost[5],c(1,4)+cost[4],c(1,3)+cost[3],c(1,2)+cost[2]\}=15$ | 2 |