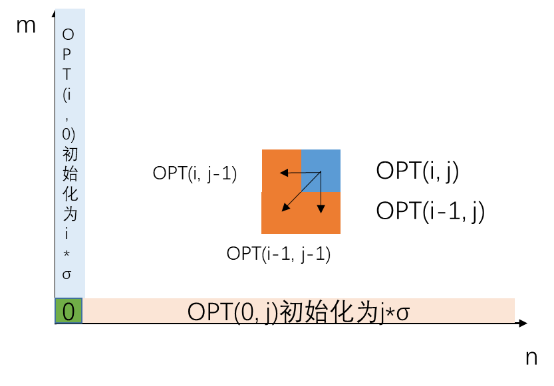
1. **字符串匹配问题补充：**

之前的背包问题可以看做二维动归，复杂度为O(n\*W)，而RNA结构问题可以看做二维动归加遍历复杂度为O(n3)，而字符串问题类似二维动归，不存在遍历，所以复杂度也为平方级别O(m\*n)。

* 1. **字符串匹配问题的依赖关系图：**



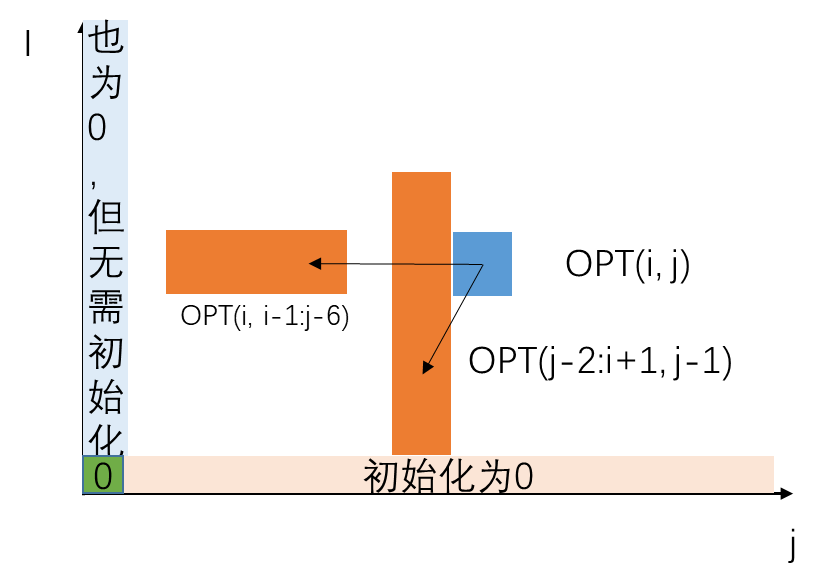
* 1. **空间压缩**

根据依赖关系图可以看出，当前行计算只与前一行和当前行的前一项有关，所以我们对空间压缩时只存储一行外加一个元素，并按照如下规则更新：

1. 对OPT(0)到OPT(j)初始化为j\*σ，对temp初始化为0，temp表示当前元素左下角的元素。
2. 计算第i行第0个元素时，直接将OPT(0)赋值给temp，然后将OPT(0)赋值为i\*σ；
3. 计算第i行时的第j个元素时，根据temp、OPT(j-1)和OPT(j)三项，算出新的OPT(j)，然后将原始OPT(j)赋值给temp，然后将新结果赋值给OPT(j)。

该思路的根本原理是，左下角的元素，在求完当前元素之后就不会再被用到，所以可以丢弃。

而RNA的二次结构问题中，空间复杂度就无法优化，原因是依赖项不止当前行，还有之前的行，所以无法压缩：



* 1. **最优解回溯：**

字符串匹配问题，如果要输出最优解，在计算表OPT时，加入一个新变量，表示当前项OPT是从之前三项中哪一项递推出的，之后OPT(m,n)开始往前进行DFS遍历，每次遍历到终点时，就输出一次栈的内容（逆序），就能找到所有符合条件的路径了。时间复杂度为O(m+n)，因为从OPT(m,n)每次坐标严格减少1，到OPT(0,0)最多需要O(m+n)次查找。

注意：之所以使用DFS，是因为有的项在递推时，多个依赖项会推出相同的结果，所以可能有多种结果。

* 1. **公共子序列问题：**

字符串匹配问题还能够转化为公共子序列问题，该问题一般给定两个字符串，求匹配的最大长度，这时与字符串匹配问题有一些小的区别：

1. 字符串匹配问题是求最小的开销，而公共子序列是求最大的长度，所以原问题的min应换成max；
2. 字符串匹配问题对于匹配错误和空的匹配都有惩罚，正确匹配惩罚是0。而该问题对于匹配错误和空匹配没有奖励，对正确匹配奖励为1；

所以，该问题的递推方程为：

OPT(i, j)= 0 (i==0 || j==0)

max{OPT(i-1, j), OPT(i, j-1)} , (s1[i]!=s2[j])

1+OPT(i-1, j-1) , (s1[i]==s2[j])

初始化都是0。

注意：当s1[i]==s2[j]时，为什么不是max{OPT(i-1, j), OPT(i, j-1), 1+OPT(i-1, j-1)}呢？

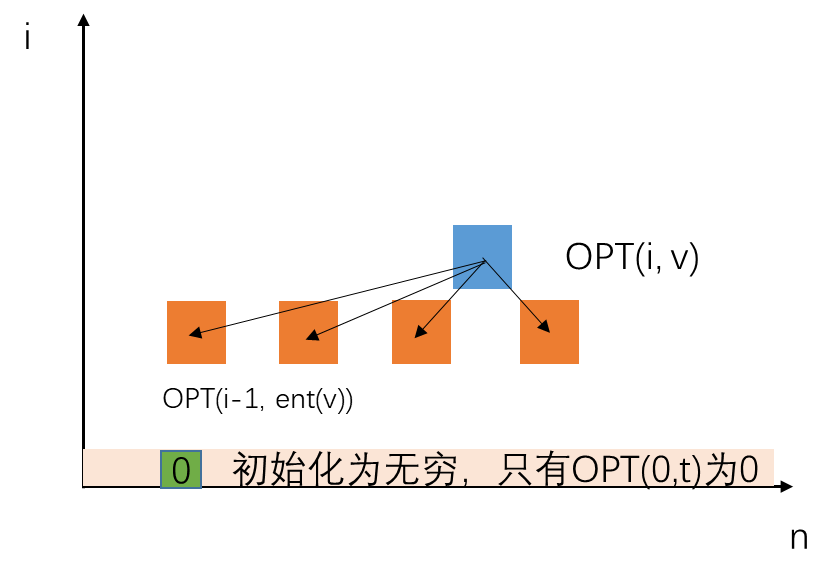
因为根据之前的定义，我们知道，根据不能交叉匹配的原则，如果当前字符串s1和s2的最后一个元素不匹配，那么这两个元素至多有一个元素被匹配。即如果s1[i]没有与s2[j]匹配，则只可能s1[i]与s2[m]匹配，或s2[j]与s1[n]匹配，其中m<j，n<i，二者不可能同时匹配。而上述两种可能匹配的长度都小于等于1+OPT(i-1, j-1)。即花括号中的三项中，最后一项是永远大于等于前两项的，所以我们只保留最后一项就能够求出最优结果了。

而原问题中，由于σ和α的影响，三者的大小关系不确定，所以不能省略前两项：

max{σ+OPT(i-1, j), σ+OPT(i, j-1), αij+OPT(i-1, j-1)}。

1. **Bellman-Ford算法的空间优化：**

首先看一下依赖关系图：



其中，横轴为各个节点，不再具有顺序性，所以依赖关系为一整行，我们至少需要O(n)的空间来存储，所以我们使用1维数组，长度n，其中d(v)表示，节点v到t的最短距离。

更新时，对于每一条 (u, v)的边，更新d(u)=min{d(u), cuv+d(v)}，如果后一项更小，我们将数组successor[u]设为v。最终输出时，只需要根据给定的起始点s，不断查找successor数组到t即可。

* 1. **跟原来方法的区别：**

原算法遍历的是v的出边，即OPT(i ,v) = min {OPT(i-1, v), OPT(i-1, w)+Cvw}，表示v到t的最短路径可以由v到w的最短路径，加上w到v的最短路径。而新算法则在该思想的基础上，对遍历顺序做了修改，在遍历第v个节点时，考虑它的入边(u, v)连接的点，更新d(u)。本质上，原算法也可以这样遍历，即在i轮v的遍历时，对每个u，更新OPT(i+1 ,u)= min{OPT(i, u), OPT(i, v)+Cuv}，此时i从0到n-2循环。

所以两种方法本质上没有区别，但我们说后一种更好，因为它可以少进行一些更新，即如果d(v)在上一轮没有被更新，则这一轮的d(v)的入边我们可以不用继续考虑，因为在d(v)轮时，我们计算的是u到v之后，再通过v到t的最优解，因为u到v是直连的，v到w为d(v)没有更新，所以这些路径都不会引起d(u)的更新。

另外，算法每次迭代至少能够计算出一个结点的最短路径。如果所有的d(v)在当前轮都没有被更新，则算法可以提前停止。

* 1. **原地更新的正确性**

首先，根据PPT我们知道，压缩后，第i轮循环得到的并不一定是最多i条边时的最优解，而可能大于i，本质原因时，我们并没有限制依赖的项一定是上一轮的值，也可能是这一轮刚刚计算出来的值，这就导致我们在第i轮循环时，它依赖的结果可能就是本轮刚刚计算出来的，也就相当于计算出的结果应该是i+1轮的值，所以如此传递下去可能轮数很少就能够得到结果。

比如PPT上一个线性的图，A->B->C，如果我们按照A、B、C的顺序考察，需要两轮求出A到C的最短距离，而如果按照C、B、A的顺序，一轮就能求出所有的d(v)值，并且不再变化。

**那么这样的依赖关系是否正确呢？**也就是说，每次对d(v)进行原地更新，而不是使用滚动数组，每次严格按照i-1轮的内容进行更新是否会影响正确性呢？（注意，新一轮迭代中，不是所有项都会被计算一遍，此时默认该节点的值与上一轮相等）

首先，原地更新的依赖关系一定不再符合我们原问题递推方程所示的那样了，但这并不影响结果的正确性。

假设，我们事先知道节点v到t的最短路径，并赋值给d(v)，尽管它可能不是第一轮就能计算得到的。那么我们是不是在第1轮就能使用它呢，显然是可以的，我们通过d(v)，不断更新所有先到v再到t的点，并且一定是正确的。

本质上，迭代过程中，数组d的值只会变得更小，而且越小越接近真实值，所以如果我们用合法的方式能让数组变得更小，那么一定不会破坏答案的正确性。

1. **基于BF算法的负环检测**

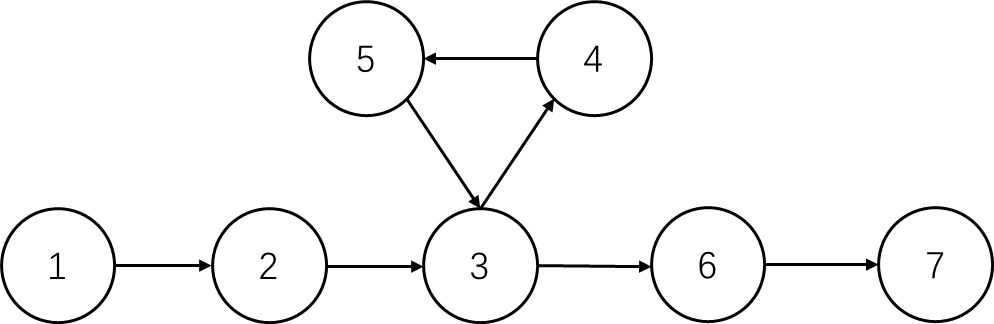
首先要明确PPT上的几个定理：

1)：如果后继数组中包含一个环，则该环是一个负环；

2）：如果循环到n轮还有d(v)值被更新，则说明图中有负环。

找负环的方式就是从被更新的d(v)向后找，一定可以找到一个负环，但v不一定在环上。

问题：对于下面的一个图



假设图中共有7个节点，3、4、5构成一个负环，当我们执行第7轮循环时，存不存在这种情况：边数从6增加到7，所有节点的最短路径不足以绕环一圈，所以OPT值没有被更改，进而证明检测算法是错误的。

答：上述情况是不存在的，虽然只加1条边，看似不足以绕大的负环走一圈，但每次循环仍然有一些点差一条边就能够绕负环一圈。比如上图中，节点1绕负环一圈走到7需要的边数就为7，两圈为10，那么当边数为8时，节点1不足以再绕一圈，但是对于节点3，边数为8时，刚好绕两圈，所以它的值会在第8轮迭代中更新。所以每次迭代，只要有负环，则一定节点多加一条边时，能够再绕负环一圈，更新自己的OPT值。

如果图本身就是一个大负环，则只有第一个点，在给定n条边时，足够绕负环一圈，同样也会更新。

1. **SPFA(Shortest Path Fast Algorithm):对环检测算法的进一步优化**

之前的BF算法时间复杂度为θ(m\*n)，但如果我们只扫描有用的边，而不是全部的边，则算法会更快一些。

本算法中，我们借助一个队列Q和一个布尔数组A来辅助更新d值。

初始时，将汇点t入队，同时更新A[t]为1，表示在队列中；d(t)=0，其余节点d值为正无穷。

While（队列为空时算法停止）

从队列中弹出一个元素v，更改该元素的布尔值为0，表示在队列外。

找到所有连入v的节点u，更新d(u)=d(v)+Cuv，如果右边更小。

将所有的u入队，更新这些节点的布尔值为1。

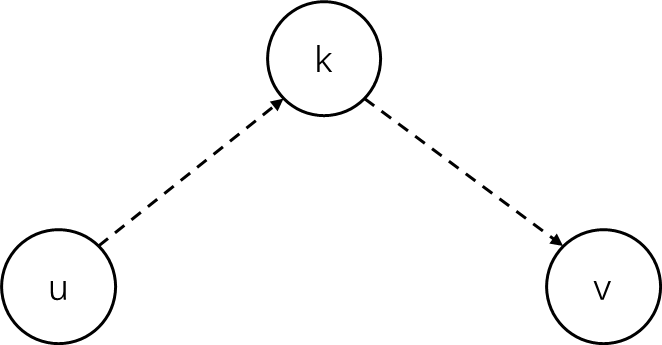
注意，入队时如果发现这些点已经在列队中（布尔值已经为1，则跳过这些节点）。

性质：

1. 一个结点可能进入多次，但如果一个结点进入了n次，说明原图中存在负环，此时还需要维护一个进入队列的计数器；
2. 如果不使用布尔数组，一个结点可能同时出现在队列中两次；
3. 只能使用队列，而不能用栈来实现。
4. **求图中任意两点最短路径：Floyd-Warshall算法（FW算法）**

首先，运行算法需要对所有节点进行排序，以保证每次循环遍历节点的顺序是一定的。

算法的思想很简单，如图：



U到v的最短路径一定是以下两种情况：1）u到v直连；2)u先到某个点k，之后再通过k到v，所以需要遍历所有可能的k值。转移方程：

fk(u, v) = min(fk-1(u, v), fk-1(u, k)+ fk-1(k, v))

观察状态转移方程，我们在求经过k时的路径时，用到了上一轮的值，那如果我们用二维数组存，是否存在第k轮用到的fk-1(u, k)实际上是本轮已经计算出来的呢？即fk(u, k)。答：不会，因为本轮中，更新的所有fk(u, v)中，如果u=k或者v=k时，fk(u, k)=min(fk-1(u, k), fk-1(u, k)+ fk-1(k, k))= fk-1(u, k)，所以不会更新，亦即，用到的值一定是上一轮的值，空间上可以将第k维省略掉。

数组初始化：

f0(u, v)= 如果uv之间存在直连边，则=w(u, v)，否则为无穷大；

注意，使用无穷大时还要注意溢出问题，即两个正无穷相加得到的是负数。所以还需要设置一个布尔数组，表示两点的可达性，初始化时，如果两点直连，为1，否则为0。更新时，如果用到了后面一项，则修改布尔数组的值为1。

实现代码：

for(int k=1; k<=n; k++)

{

for(int i=1; i<=n; i++)

{

for(int j=1; j<=n; j++)

{

if(is\_legal[i][k] && is\_legal[k][j] && dist[i][k]+dist[k][j]<dist[i][j])

{

dist[i][j]=dist[i][k]+dist[k][j];

is\_legal[i][j]=true;

}

}

}

}

算法复杂度为O(n^3)，并且不能压缩，所以仅适用于节点数较少的情况。

Floyd用于负环检测：

如果f(u, v) + f(v, u) <0，则说明有负环，但这样需要检查每个点对，复杂度高；

检查对角线f(k, k)，如果对角线小于零，则说明该点在一个负环中；

**如果需要输出最短路径**：

需要定义一个max数组，记录下一跳的节点，比如如果在更新max(u, v)时，使用到了第二项，经过了节点k，则更新max(u, v)= max(u, k)，注意不是k，因为下一个节点不一定是k，k只是路径中的一个点。初始化时，max(u, v)=如果u,v有边，则为v，否则为非法值。同时自己到自己下一跳也是自己。

**检测图中所有负环：**

定义一个超级汇点，连接图中所有的点，入度为n，出度为0，之后使用BF或FW算法，求点到超级汇点的最短距离，就能检测出图中所有的负环。

简要证明：

1. 引入超级汇点不会增加或减少环，因为新增的边都指向超级汇点，而汇点不可能在一个环上。
2. 不影响原图的可达性。

