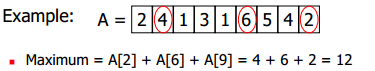
算法整理四

Chapter11 动态规划Dynamic Programming贪心算法得到的活动选择并不都是正确的。

数据选择问题

在数组A之中选择n个元素的子集，元素之间至少间隔两个元素，并且所选元素总和的值最大。



解题思路：

我们不知道A[n]是否在subset之中，但是我们知道A[n]要么在，要么不在。所以

1. 当A[n]在subset之中的时候

A[n-1]和A[n-2]必然不在subset之中

1. 当A[n]不在subset之中

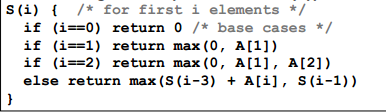
next step to consider A[1..n–1]



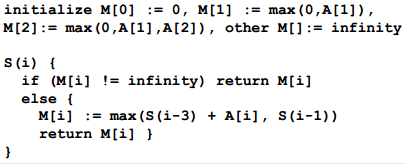
递归公式



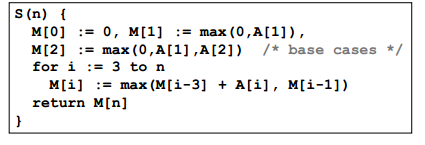
伪代码：



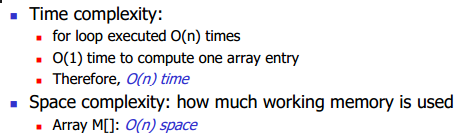
发现在递归调用的时候，S(i)被计算了好几次，所以用数组存储之前计算的结果能够提高算法效率。



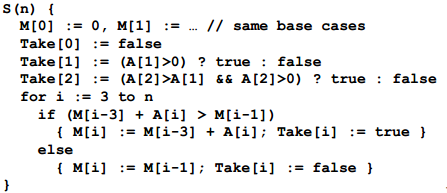
等效的迭代代码：



迭代的空间和时间复杂度：



但是以上的代码，只返回了max sum，没有记录下最优解，所以我们需要一个数组来记录最优解。



动态规划的规则

1. 递归公式A recursive formulation
2. 最优子结构Optimal substructure
3. Overlapping subproblems：子问题有重叠，比如上面的代码，因为S(i)重复了计算数次，所以才需要新的数组存储中间值。重复计算的S(i)就是子问题重叠部分。

动态规划解题步骤（四大步骤）

Step 1: Identify the structure of the optimal solution

Step 2: Define a recursive formulation

Step 3: Compute the optimal cost using a table.

Two approaches:

 Top-down: use recursion to call smaller subproblems. Stored results used if possible

 Bottom-up: iterate all subproblems starting from smaller ones

 Step 4: Construct the actual solution

Chapter 12 Dynamic Programming IILongest Common Subsequence(LCS)最长公共子序列问题

A longest common subsequence is a common subsequence with maximum length



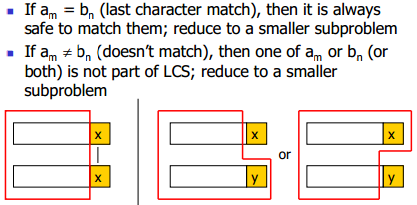
根据动态规划的一般解题步骤，我们可以分成四步考虑。

S1：确定最优答案的结构Identify the structure of the optimal solution

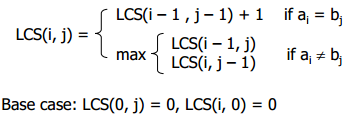
假定两个序列：



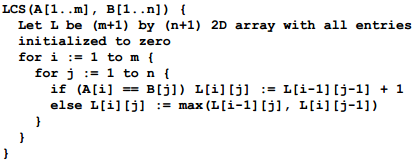
在A,B两个序列之中取最后两个字母比较，有相等和不相等两个结果。



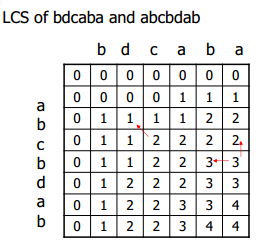
S2:递归公式recursive formulation



S3：此处使用自上而下的算法Bottom-up Algorithm伪代码



用table来计算



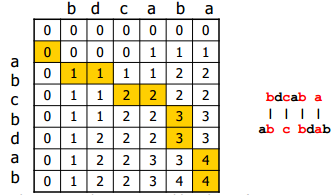
先根据base case填表，然后根据递推计算。

时间复杂度：O(mn)

空间复杂度: O(mn)

Step 4: 得到最优解Retrieving the Actual LCS

我们可以检索table知道最优解



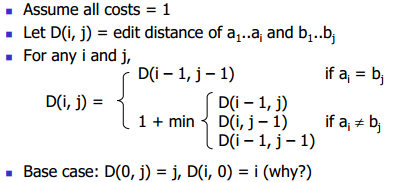
我们从L[m][n]开始倒着往前搜寻。可能有多个解。像上表，就有两个解，一个是如图所示，还有一个是bcab。

相似问题：

有两个序列A,B，用最少的花费，把A转变成B

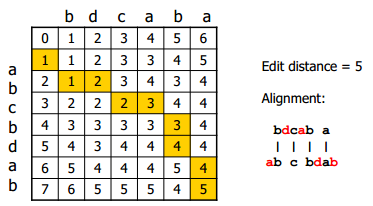
可用的操作有插入，删除，替换（每个operation都有a cost）

思路是计算距离edit distance



如果字母是相等的，就不必要有操作，所以也就没有花费。如果不相等，就是最优子结构的花费再加上一次cost，因为我们使用了distance这个概念，也就是把a cost抽象为1..

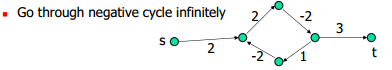
e.g



Chapter 13 Dynamic Programming IIIDirected Acyclic Graph (DAG)有向无环图

Dijkstra’s algorithm是求最短路径的算法，但是当边的权值是负数的时候，不适合使用这个算法。所以需要一个新的算法求边权值为负数的最短路径。

如果一个图，有negative cycles，则不一定有最短路径。



所以我们在计算最短路径的时候，假设没有negative cycles。

最短路径只有n-1条边，如果大于n-1条边，则有的点就访问了两次，也就是说有环。

Bellman-Ford Algorithm是可以求有负边的最短路径算法

算法解释：

S(i,v)表明从s到v的最短路径，经过了i条边。

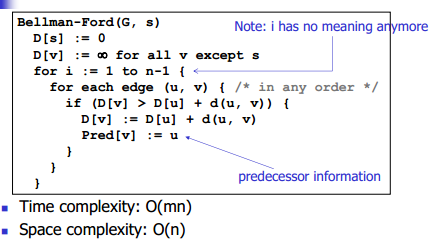
假设一条边e(w,v)，所以



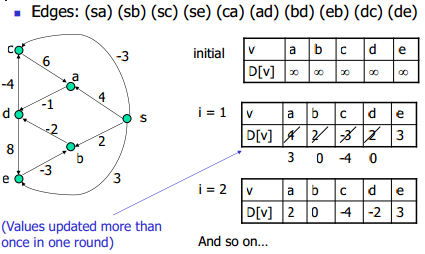
如果从s到v，只经过i-1条边，而e(w,v)的存在不影响最短路径的大小，所以最短路径就不经过e(w,v)。如果e(w,v)的存在影响最短路径的大小，所以此条路径是先经过w,再经过v。因为我们一开始并不知道w是哪个节点，所以我们从v开始搜索和v相邻的所有节点。

伪代码：

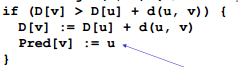
Principle: apply the edge update procedure to all edges. We don’t know the right order, so choose an arbitrary one and do it a “large enough” number of times



e.g



先写出所有的边，然后对每条边执行一次

。但是样例中间有点错误，不过不影响解答方式。一次迭代之中，数组D的元素可能被改动过多次。

All-Pairs Shortest Paths所有顶点对之间的时间复杂度：

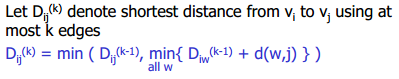
之前求的都是单源的最短路径，现在我们需要求所有顶点的最短路径，就是应用n次最短路径算法。

Positive edge weights正值边: Dijkstra’s algorithm. O(m log n) x n =O(mn log n)

General edge weights: Bellman-Ford algorithm. O(mn) x n =O(mn2)

Using Bellman-Ford in APSP

将Bellman-ford算法写成类似动态规划DP的递归公式：



用二维数组存储两点之间的distance

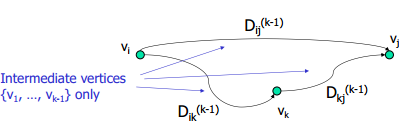
此算法的时间复杂度



更快的算法是动态规划DP------ Floyd-Warshall Algorithm



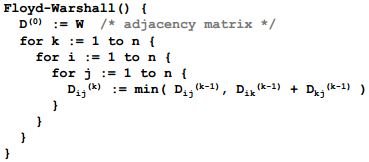
指数k表示中间节点的最大个数小于等于k-1



要么在最短路径之中，要么不在。

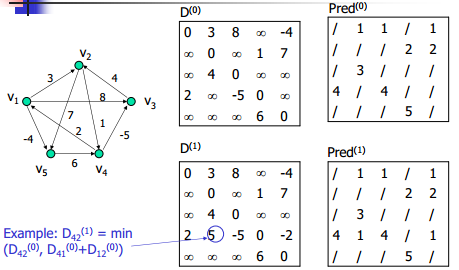


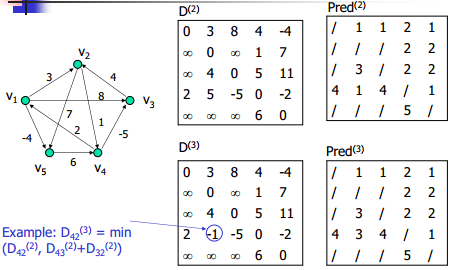
伪代码：

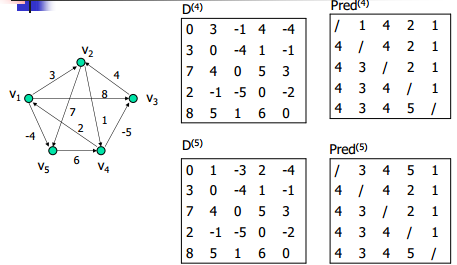


e.g 自己去算一遍

Pred这个数组是记录路径，最后用来输出完整路径。而D这个数组只记录最短路径长度。







时间和空间复杂度

时间复杂度：

空间复杂度：，二维数组

给出完整路径的Floyd-Warshall Algorithm伪代码

