算法设计与分析

➢ LECTURE 9

Lecture 9

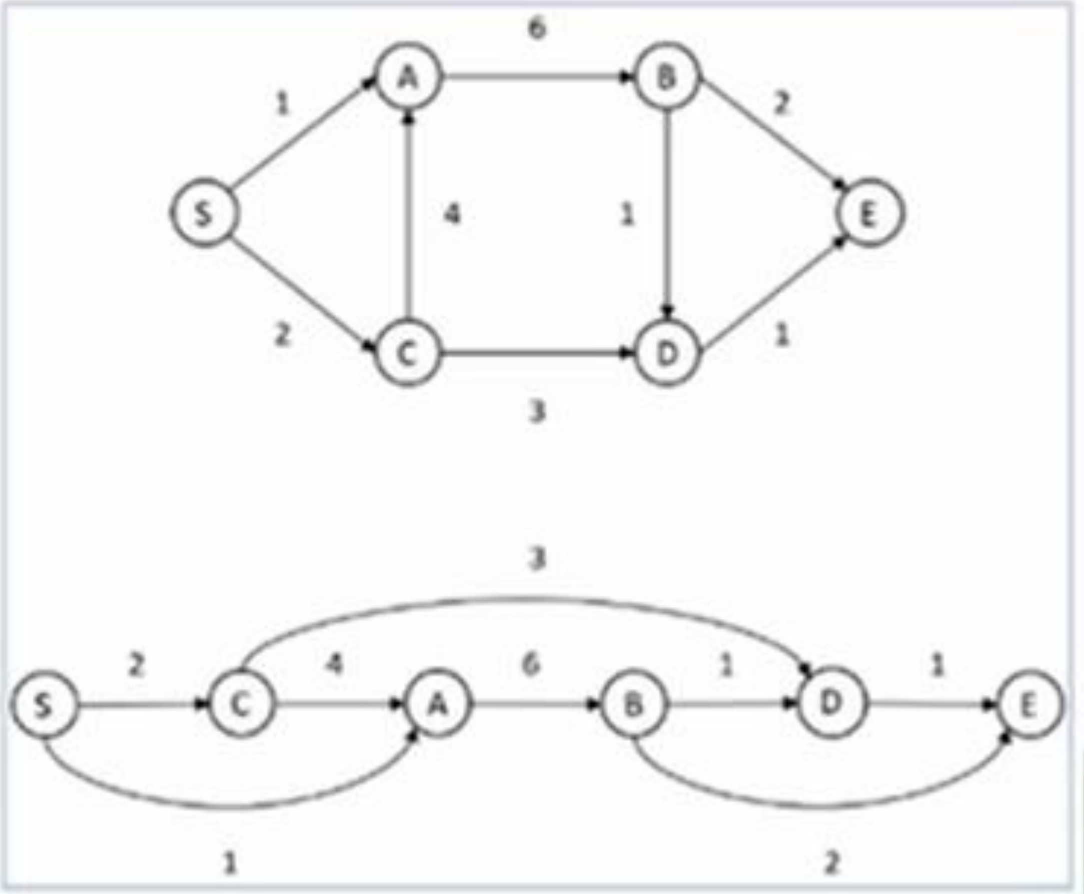
图优化-动态规划

Outline

- 多源最短路径问题
  - Floyd 算法
- 动态规划
  - 矩阵相乘
  - 编辑距离
  - 硬币兑换问题
  - 最大和连续子序列

有向无环图的给定源点最短路径

- 有向无环图
  - 拓扑排序
- D.dis = min{B.dis+1, C.dis+3)
- 思考？
  - 最长路径、路径上边权值乘积的最小值，etc.

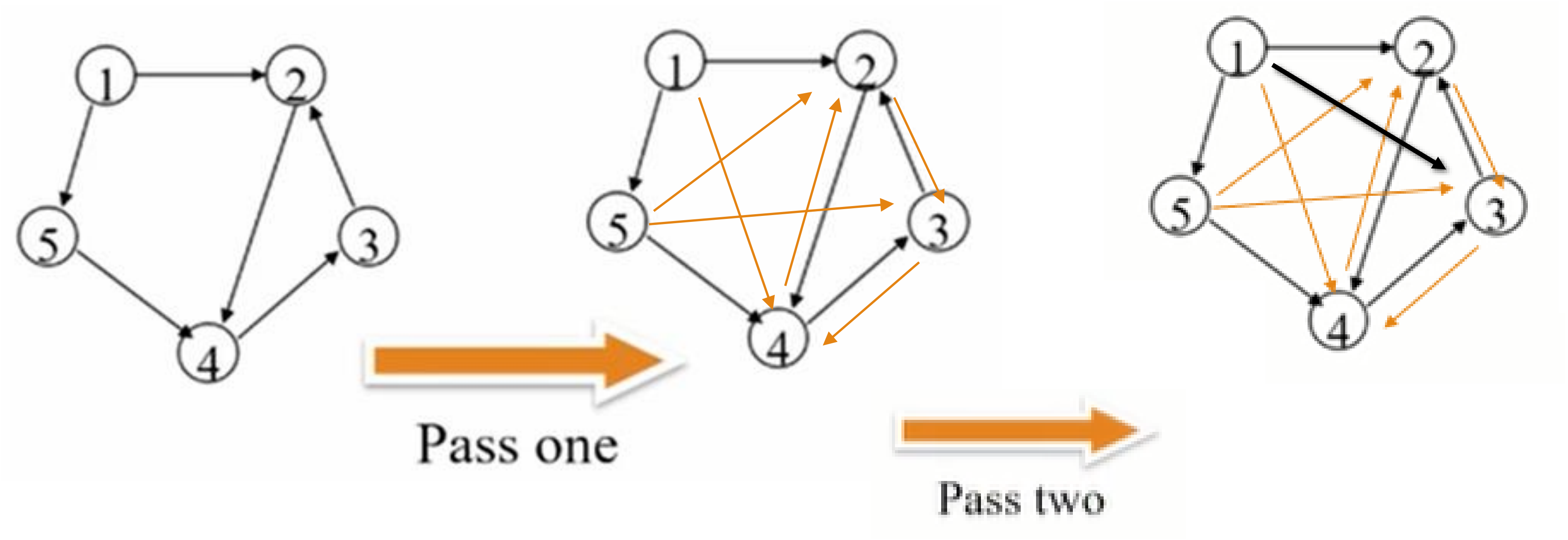


所有点对最短路径

- 对于图上的所有点对 u, v:
  - 是否存在一条从 u 到 v 的路径？
  - 最短路径是多少？
- 问题核心：二元关系的传递闭包
  - 蛮力算法
  - 基于路径长度递归的最短路径
  - 基于动态规划策略的 Floyd 算法

传递闭包问题和 Shortcut 算法

- 对于三个点 i, j, k，如果i->j且j->k，则i⇝k, 成为 shortcut
- 对所有可能的点对反复尝试添加 shortcut，得到最终的传递闭包图



传递闭包问题和 Shortcut 算法

- **void** simpleTransitiveClosure(**boolean**[ ][ ] A, **int** n, **boolean**[ ][ ] R)
- **int** i,j,k;
- Copy A to R;
- Set all main diagonal entries, r_{ii}, to *true*;
- **while** (any entry of R changed during one complete pass)
- **for** (i=1; i≤n; i++)
- **for** (j=1; j≤n; j++)
- **for** (k=1; k≤n; k++)
- $r_{ij}=r_{ij}\vee(r_{ik}\wedge r_{kj})$

$O(n^4)$

The order of (i,j,k) matters

基于路径长度的递归

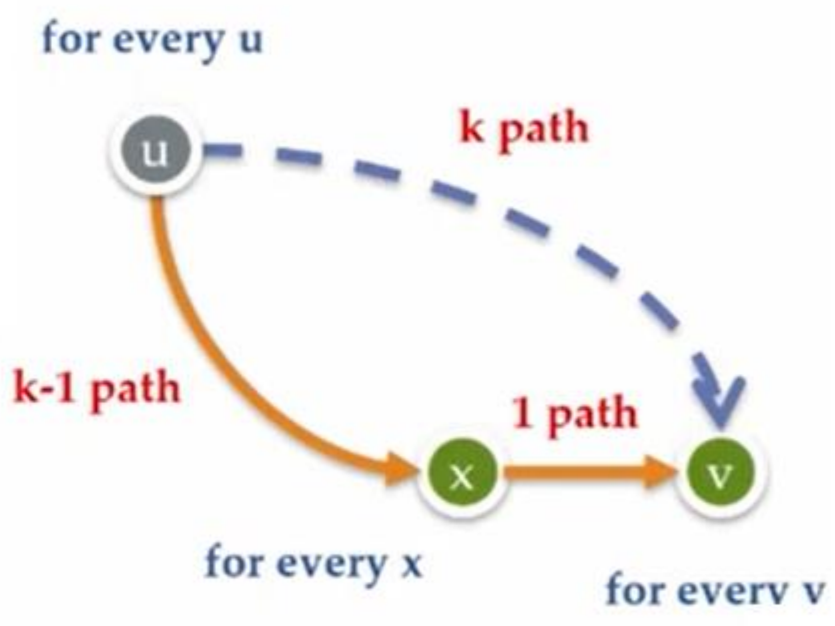
- 递归：最长 k 边可达
- 枚举：
  - 所有的路径长度
  - 所有的源点和终点

```
for k=1 to n-1
    for all vertices u
        for all vertices v
            for all vertices x pointing to v
                r^k_uv = r^(k-1)_uv ∨ (r^(k-1)_ux ∧ r_xv)
```

$$r^{k}_{uv} = r^{k-1}_{uv} \vee (r^{k-1}_{ux} \wedge r_{xv})$$

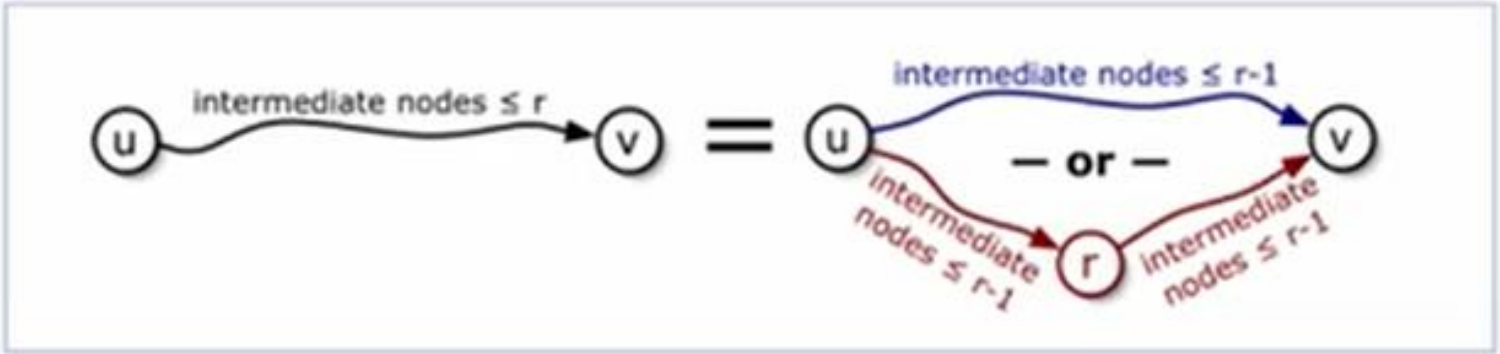
$r_{xv} = 1$

$O(n^4)$



Floyd 算法

基本思想



高效“聪明”的递归

- 定义子问题 d(i,j,k) 表示仅使用 I_k 的点作为中继节点，点 v_i到v_j 的最短路径的长度

计算距离矩阵

- $D^{(0)}[i][j] = w_{ij}$
- $D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$

```
Void allPairsShortestPaths(float [][] W, int n, float [][] D)
  int i, j, k;
  Copy W into D;
  for (k=1; k≤n; k++)
    for (i=1; i≤n; i++)
      for (j=1; j≤n; j++)
        D[i][j] = min (D[i][j], D[i][k]+D[k][j]);
```

Lecture 9

图优化-动态规划

Outline

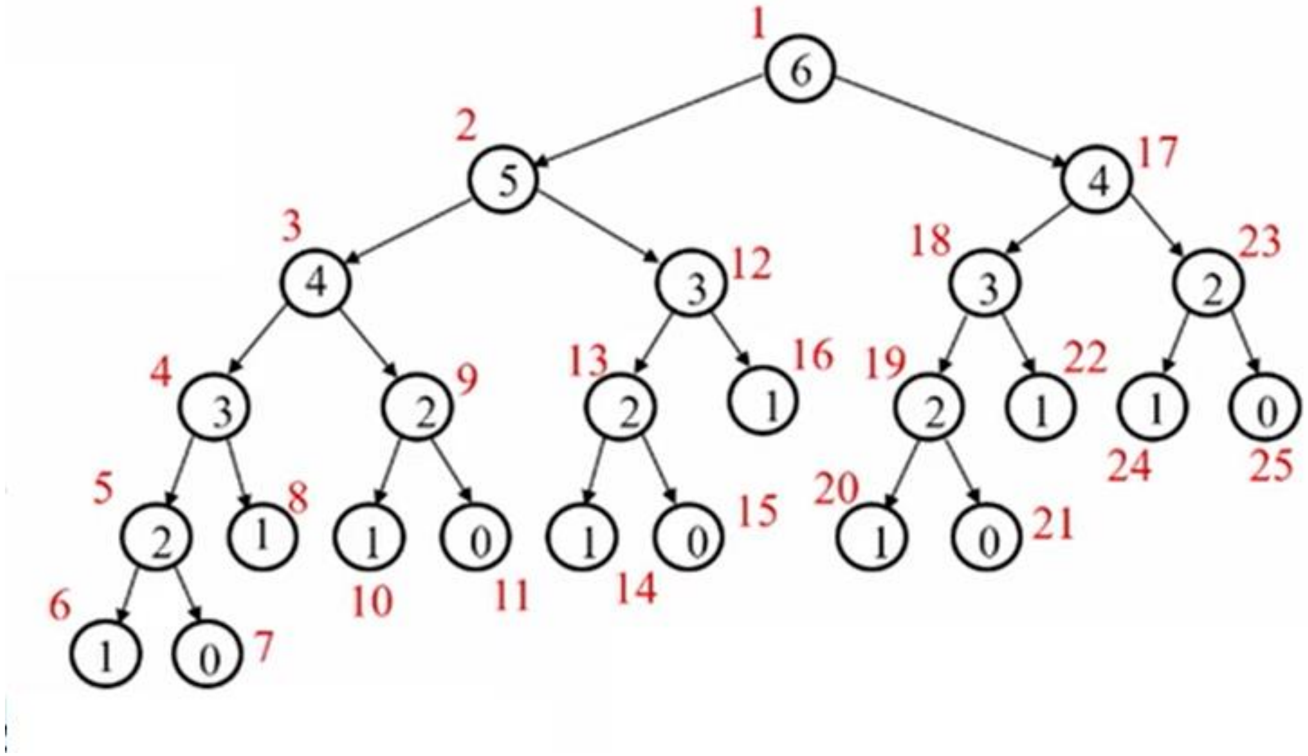
- 动态规划
  - 最优子结构

- 动态规划
  - 矩阵相乘
  - 编辑距离
  - 硬币兑换问题
  - 最大和连续子序列

斐波那契数列

- f(n)=f(n-1)+f(n-2)

- 递归实现代价：
- T(n)=T(n-1)+T(n-2)



https://leetcode-cn.com/problems/fibonacci-number/

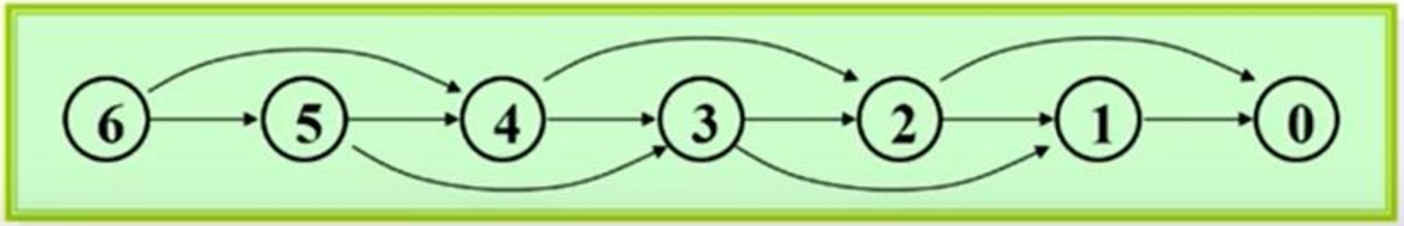
斐波那契数列

- 自顶向下（DFS）
- 借助额外数组（查表）

```
memo = {}
def fib2(n):
    if n in memo:                              # 查表
        return memo[n]
    else:
        if n <= 2:                             # 边界条件
            f = 1
        else:
            f = fib2(n-1) + fib2(n-2)          # 递归调用
        memo[n] = f                            # 将结果存储于表中
        return f
```

斐波那契数列

- 自底向上
- 利用拓扑排序

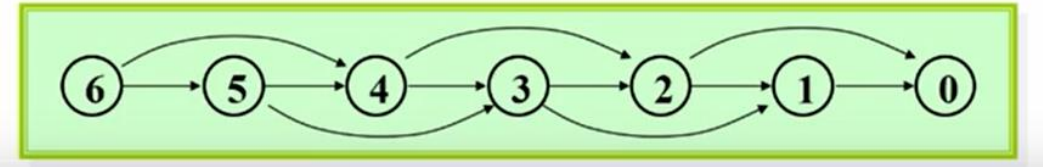



```python
def fib_bottom_up(n):
    fib = {}                              # 存储结果的字典
    for k in range(n+1):
        if k<=2:                          # 边界条件
            f = 1
        else:
            f = fib[k-1]+fib[k-2]         # 自底向上填表
        fib[k] = f
    return fib[n]
```

动态规划策略

- 递归+记忆+猜测
  - “记忆”高效地实现递归
  - “猜测”构造递归

- 动态规划一般求解过程：
  - 定义子问题（语言描述、参数）
  - 建立子问题关系（递归）
    - 通过猜测构造子问题求解
    - 子问题求解过程——节点遍历（满足拓扑排序）
  - 明确边界条件
  - 得到原问题的解



矩阵相乘

- 给定一个矩阵序列，现要计算它们的乘积 $A_1A_2\cdots A_n$
- 计算相乘的最低代价及相应的相乘次序

- 例：$A_1(\in R^{30\times 1})\times A_2(\in R^{1\times 40})\times A_3(\in R^{40\times 10})\times A_4(\in R^{10\times 25})$
- 不同相乘次序的代价：
  - $((A_1\times A_2)\times A_3)\times A_4)$：*20700*
  - $A_1\times(A_2\times(A_3\times A_4))$：*11750*
  - $(A_1\times A_2)\times(A_3\times A_4)$：*41200*
  - $A_1\times((A_2\times A_3)\times A_4)$：*1400*

蛮力递归

- BF1:
  - 考虑第一次相乘的位置
  - $W(n) = (n-1)W(n-1) + n$

- BF2:
  - 考虑最后一次相乘的位置
  - $W(n) \geq 2W(n-1) + n$

采用动态规划的递归

- 子问题依赖—递归顺序

```
matrixOrder(n, cost, last)
•    for (low=n-1; low≥1; low--)
•      for (high=low+1; high≤n; high++)

          Compute solution of subproblem (low,
          high) and store it in cost[low][high] and
          last[low][high]

•    return cost[0][n]
```

采用动态规划的递归

- 子问题计算：
  - 枚举所有的k

```
float matrixOrder(int[] dim, int n, int[]
    multOrder)
 <initialization of last,cost,bestcost,bestlast...>
  for (low=n-1; low≥1; low--)
    for (high=low+1; high≤n; high++)
      if (high-low==1) <base case>
      else bestcost=∞;
      for (k=low+1; k≤high-1; k++)
        a=cost[low][k];
        b=cost[k][high]
        c=multCost(dim[low], dim[k],
   dim[high]);
        if (a+b+c<bestCost)
          bestCost=a+b+c; bestLast=k;
      cost[low][high]=bestCost;
      last[low][high]=bestLast;
 extrctOrderWrap(n, last, multOrder)
```

编辑距离

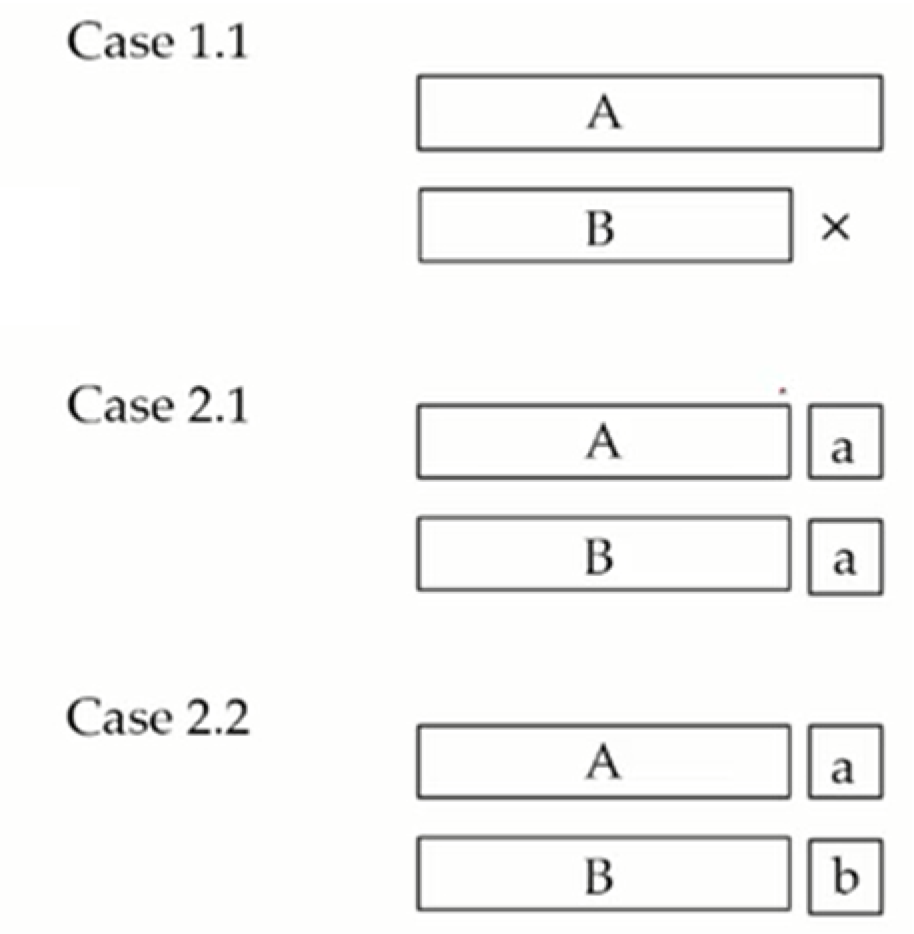
- 给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。
- 你可以对一个单词进行如下三种操作：
  - 插入一个字符
  - 删除一个字符
  - 替换一个字符

- 输入： word1 = "horse", word2 = "ros“
- 输出： 3
- 解释： horse -> rorse (将 'h' 替换为 ‘r’) rorse -> rose (删除 'r')rose -> ros (删除 'e')

https://leetcode-cn.com/problems/edit-distance

编辑距离

- 递归
  - 找到子问题
  - 枚举所有可能
- 子问题调度
  - 依赖关系



编辑距离

```python
class Solution(object):
    def minDistance(self, word1, word2):
        m, n = len(word1)+1, len(word2)+1
        dp = [[0]*n for _ in range(m)]
        for i in range(m):
            dp[i][0] = i
        for j in range(n):
            dp[0][j] = j

        for i in range(1, m):
            for j in range(1, n):
                if word1[i-1] == word2[j-1]:
                    dp[i][j] = min(dp[i-1][j-1], dp[i-1][j]+1, dp[i][j-1]+1)
                else:
                    dp[i][j] = min(dp[i-1][j-1]+1, dp[i-1][j]+1, dp[i][j-1]+1)
        return dp[m-1][n-1]
```

硬币兑换问题

- 假设有多种面值的硬币 $d_1 = 1,\ d_1 < d_2 < \cdots < d_n$，每种硬币有充分多个，最少使用多少个硬币可以兑换金额 N.

- 子问题：
  - coin[i, j]: 使用前 i 种面值的硬币，兑换金额为 j 的钱，最少使用的硬币数

$$coin[i,j] = \begin{cases} coin[i-1,j], & \text{没有使用}d_i\text{的硬币} \\ coin[i,j-d_i]+1, & \text{至少使用了1个面值为}d_i\text{的硬币} \end{cases}$$

最大和连续子序列问题

- 定义子问题（前缀）
  - DP[i] 表示到第 i 个元素子序列累加和最大
- 建立子问题关系
  - $\mathrm{DP[i]} = \mathrm{MAX}\begin{cases}\mathrm{DP[i-1]+A[i]} \\ A[i]\end{cases}$
- 边界条件：DP[0]=0
- 子问题求解顺序（拓扑排序）

动态规划关键特征

- 动态规划
  - 重叠子问题（表面）
  - 最优子结构

- 如何使用动态规划
  - “蛮力”递归
    - 重叠子问题
  - “聪明且高效”地实现
    - 子问题拓扑排序

Lecture 9

动态规划

Outline

- 动态规划
  - 相容任务调度
  - 单词排版
  - 0-1 背包问题
  - 整数子集和问题

相容任务调度

- 假设有一组任务需要解决，每个任务有指定的起始、终止时间，同时，多个任务不能同时进行。
- 如果两个任务的时间区间不重叠，称这两个任务是相容的。
- 问题：找出最大的相容任务集。

- 贪心策略：
  - 最早开始任务
  - 最短任务
  - 最少冲突任务

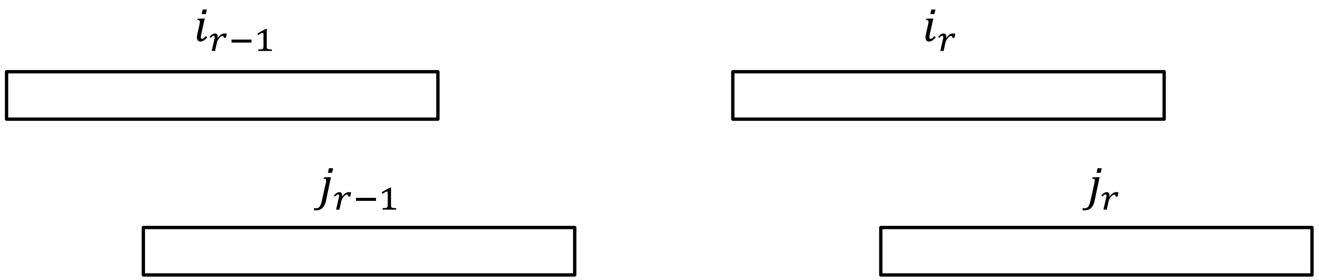
相容任务调度-贪心

基于任务结束时间的贪心算法

- 将所有任务按照结束时间的先后进行排序
- 从前往后依次扫描，如果一个任务不与已选择任务冲突，则选择它；否则忽略

算法正确性证明

- 数学归纳法



贪心解始终先于最优解

相容任务调度-动态规划

定义子问题

- c[i,j] 表示任务集合$S_{i,j}$中最大相容任务的个数

建立子问题关系

- c[i,j] = MAX $\begin{cases} 0, & S_{i,j} = \emptyset \\ \max\limits_{a_k \in S_{i,j}} \{c[i,k] + c[k,j] + 1\}, & S_{i,j} \neq \emptyset \end{cases}$

单词排版

- Input : w = [‘book’, ‘face’,...](e.g. [5,5,5,5,14])，p=18
- Output: 把 w 排版在宽度为 p的页面上，尽量美观
- 美观程度定义为 B = $(p - \sum w_i)^3$

- 定义子问题（前缀）
  - DP[i] 表示前 i 个单词的最优解
- 建立子问题关系（猜测j，蛮力枚举）
  - DP[i] = MIN $\begin{cases} \text{DP[j]}+B(j,i), & j \in [1,i] \\ B(j,i) = \infty, p < \sum w_k \end{cases}$

0-1背包问题

- Input:
  - 物品：(s1=3, v1=4)、(s2=4,v2=5)、(s3=5,v3=6)， 总容量 S=10
- Output:
  - 累加价值最多的物品

- 定义子问题
  - DP[i][j] 表示前 i 个物品放置包容量为 j 的累加最大价值
- 建立子问题关系
  - $$DP[i,j] = MAX\begin{cases} DP[i-1][j] \\ DP[i-1, j-s(i)] + v(i) \end{cases}$$

整数子集和问题

- 给定 X={1,2,,,,,N},从中选出子集使得子集和等于 $\frac{s_n}{2}$, s_n为X的累加和。

- 定义子问题
  - DP[i,j] 表示前 i 个元素子序列累加和为j的个数
- 建立子问题关系
  - $$DP[i,j] = \begin{cases} DP[i-1,j], & j < i \\ DP[i-1,j] + DP[i-1][j-1], & j \geq i \end{cases}$$

Thanks