hw6 刘佳璇 231220105

第十三章

13.1 最大相容任务集合问题

令 $dp[i][j]\ (1 \leq i, j \leq n)$ 表示在 a_i 结束之后、 a_j 开始之前的时间区间内最大相容任务的个数，状态转移方程为

$$dp[i][j] = \begin{cases} max(dp[i][k] + dp[k][j] + 1), & i < j \\ 0, & i \geq j \end{cases}$$

其中 k 满足 a_k 是在 a_i 结束之后、 a_j 开始之前的时间区间内的任务。 $dp[1][n]$ 即为问题的解。算法先从下到上、再从左到右遍历。

该算法共计算 $\mathrm{O}(n^2)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(n)$ ，故总的时间复杂度为 $\mathrm{O}(n^3)$ ，空间复杂度为 $\mathrm{O}(n^2)$ 。

13.2 整数子集合问题

令 $dp[i][j]\ (1 \leq i \leq n,\ 0 \leq j \leq S)$ 表示是否有仅由前 i 个元素构成的子集合的元素和为 j ，状态转移方程为

$$dp[i][j] = \begin{cases} dp[i-1][j] \vee dp[i-1][j-a_i], & i > 1 \text{且} j > 0 \\ true, & j = 0 \\ true, & i = 1 \text{且} j = a_1 \\ false, & i = 1 \text{且} j \neq 0,\ j \neq a_1 \end{cases}$$

$dp[n][S]$ 即为问题的解。算法先从上到下、再从左到右遍历。

该算法共计算 $\mathrm{O}(nS)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$ ，故总的时间复杂度为 $\mathrm{O}(nS)$ ，空间复杂度为 $\mathrm{O}(nS)$ 。

13.4 最长非递减子序列问题

令 $dp[i]\ (1 \leq i \leq n)$ 表示以第 i 个元素为结尾的最长非递减子序列的长度，状态转移方程为

$$dp[i] = \begin{cases} max(dp[k]) + 1, & i > 1 \\ 1, & i = 1 \end{cases}$$

其中 k 满足 $1 \leq k < i$ 且 $A[k] \leq A[i]$ 。 $max(dp[i])$ 即为算法的解。算法从左往右遍历。

该算法共计算 $\mathrm{O}(n)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(n)$ ，故总的时间复杂度为 $\mathrm{O}(n^2)$ ，空间复杂度为 $\mathrm{O}(n)$ 。

13.5 最低代价划分问题

令 $S[n]$ 为集合元素的前缀和， $dp[i][j]\ (1 \leq i \leq n,\ 1 \leq j \leq k)$ 表示由前 i 个元素组成的子集进行 j - 划分的最低代价，状态转移方程为

$$dp[i][j] = \begin{cases} min\{max(dp[k][j-1], S[i] - S[k])\}, & i > 1,\ j > 1 \\ S[n], & j = 1 \\ \text{不存在}, & i < j \end{cases}$$

其中 k 满足 $1 \leq k < i$ 。 $dp[n][k]$ 即为问题的解。算法先从上到下、再从左到右遍历。

该算法共计算 $\mathrm{O}(nk)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(n)$ ，故总的时间复杂度为 $\mathrm{O}(n^2k)$ ，空间复杂度为 $\mathrm{O}(nk)$ 。

13.6 最大乘积和子数组

1）元素皆为正数

令 $dp[i]\ (1 \leq i \leq n)$ 表示以第 i 个元素为结尾的最大乘积和，状态转移方程为

$$dp[i] = \begin{cases} max(A[i], A[i] \cdot dp[i-1]), & i > 1 \\ A[1], & i = 1 \end{cases}$$

$max(dp[i])$ 即为问题的解。算法从左到右遍历。

该算法共计算 $\mathrm{O}(n)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$ ，故总的时间复杂度为 $\mathrm{O}(n)$ ，空间复杂度为 $\mathrm{O}(n)$ 。

2）元素有正有负

令 $dp_+[i]\ (1 \leq i \leq n)$ 表示以第 i 个元素为结尾的最大乘积和， $dp_-[i]\ (1 \leq i \leq n)$ 表示以第 i 个元素为结尾的最小乘积和，状态转移方程为

$$dp_+[i] = \begin{cases} max(A[i], A[i] \cdot dp_+[i-1], A[i] \cdot dp_-[i-1]), & i > 1 \\ A[1], & i = 1 \end{cases}$$

$$dp_-[i] = \begin{cases} min(A[i], A[i] \cdot dp_-[i-1], A[i] \cdot dp_-[i-1]), & i > 1 \\ A[1], & i = 1 \end{cases}$$

$max(dp_+[i], dp_-[i])$ 即为问题的解。算法从左到右遍历。

该算法共计算 $\mathrm{O}(n)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$ ，故总的时间复杂度为 $\mathrm{O}(n)$ ，空间复杂度为 $\mathrm{O}(n)$ 。

13.8 最长公共子序列

1）

令 $dp[i][j]\ (0 \leq i \leq m,\ 0 \leq j \leq n)$ 表示子串 $X[1:i]$ 和 $Y[1:j]$ 的最长公共子序列，状态转移方程为

$$dp[i][j] = max \begin{cases} dp[i-1][j], \\ dp[i][j-1], \\ dp[i-1][j-1] + I\{x_i = y_j\} \end{cases}$$

其中 $dp[i][0] = dp[j][0] = 0$ 。 $dp[1][n]$ 即为问题的解。算法先从上到下、再从左到右遍历。

该算法共计算 $\mathrm{O}(mn)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$ ，故总的时间复杂度为 $\mathrm{O}(mn)$ ，空间复杂度为 $\mathrm{O}(mn)$ 。

2）

令 $dp[i][j]\ (0 \leq i \leq m,\ 0 \leq j \leq n)$ 表示子串 $X[1:i]$ 和 $Y[1:j]$ 的最长公共子序列，状态转移方程为

$$dp[i][j] = max \begin{cases} dp[i-1][j], \\ dp[i][j-1], \\ dp[i][j-1] + 1, \qquad x_i = y_j \end{cases}$$

其中 $dp[i][0] = dp[j][0] = 0$ 。 $dp[1][n]$ 即为问题的解。算法先从上到下、再从左到右遍历。

该算法共计算 $\mathrm{O}(mn)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$，故总的时间复杂度为 $\mathrm{O}(mn)$，空间复杂度为 $\mathrm{O}(mn)$。

3)

令 $dp[i][j]\ (0 \le i \le m,\ 0 \le j \le n)$ 表示子串 $X[1:i]$ 和 $Y[1:j]$ 的最长公共子序列，$A[i] := k$ 表示 x_i 剩余可用次数，状态转移方程为

$$dp[i][j] = max \begin{cases} dp[i-1][j], \\ dp[i][j-1], \\ dp[i][j-1]+1, \text{ do A[i]--}, \qquad x_i = y_j \text{ 且 } A[i] > 0 \end{cases}$$

其中 $dp[i][0] = dp[j][0] = 0$。$dp[1][n]$ 即为问题的解。算法先从上到下、再从左到右遍历。

该算法共计算 $\mathrm{O}(mn)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$，故总的时间复杂度为 $\mathrm{O}(mn)$，空间复杂度为 $\mathrm{O}(mn)$。

13.9 前向和后向相同的最长连续子串

令 $dp[i][j]\ (1 \le i < j \le n)$ 表示子串 $T[i:j]$ 中满足要求的最长连续子串的长度，$cur[i][j] := 0$ 表示在子串 $T[i:j]$ 中，T_i 和 T_j 是满足要求的前向子串的开头和后向子串的结尾且子串长度为 $cur[i][j]$（0 表示不是满足要求的子串）。状态转移方程为

$$dp[i][j] = \begin{cases} max \begin{cases} dp[i+1][j], \\ dp[i][j-1], \\ cur[i][j] = cur[i+1][j-1] + I\{T_i = T_j\} \end{cases} & \text{if} \quad i < j \\ 0, & \text{if} \quad i \ge j \end{cases}$$

$dp[1][n]$ 即为问题的解。算法先从下到上、再从左到右遍历。

该算法共计算 $\mathrm{O}(n^2)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$，故总的时间复杂度为 $\mathrm{O}(n^2)$，空间复杂度为 $\mathrm{O}(n^2)$。

13.10 最短公共超序列

原命题等价于求两个序列的最长公共子序列，并用队列 Q_x 和 Q_y 存储公共子序列在两个序列中的下标，然后合并两个序列。合并算法如下：

```
int i = 1, j = 1; // X和Y的下标指针
queue Q; // 存储最短公共超序列
while (i <= m && j <= n)
{
    while (i < Q_x.top)
    {
        Q.push(X[i]);
        i++;
    }
    while (j < Q_y.top)
    {
        Q.push(Y[j]);
        j++;
    }
```

```
    Q.push(X[Q_x.top]);
    Q_x.pop;
    Q_y.pop;
}

while (i <= m)
{
    Q.push(X[i]);
    i++;
}
while (j <= n)
{
    Q.push(Y[j]);
    j++;
}
```

13.11 最长公共子序列问题的变体

1）

不正确。删去的元素可能Y中的元素。比如令 $X =< ABC >, Y =< BACA >, Z =< ABABCCA >$ ，则 X 和 Z 的 LCS 为 $< ABC >$ 。令 $Z' =< AB \cancel{ABC} CA > \neq Y$ ，但实际上 Z 符合要求。

2）

令 $dp[i][j]\ (0 \leq i \leq m, 0 \leq j \leq n)$ 表示 X 和 Y 的子序列 $X[1:i]$ 和 $Y[1:j]$ 能否合并成 Z 的子序列 $Z[1:i+j]$ ，状态转移方程为

$$dp[i][j] = (dp[i-1][j] \wedge X_i = Z_{i+j}) \vee (dp[i][j-1] \wedge Y_j = Z_{i+j})$$

其中 $dp[i][0] = I\{X[1:i] = Z[1:i]\}$, $dp[0][j] = I\{Y[1:j] = Z[1:j]\}$ 。 $dp[1][n]$ 即为问题的解。算法先从上到下、再从左到右遍历。

该算法共计算 $\mathrm{O}(mn)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$ ，故总的时间复杂度为 $\mathrm{O}(mn)$ ，空间复杂度为 $\mathrm{O}(mn)$ 。

3）

令 $dp[i][j][p]\ (0 \leq i \leq m, 0 \leq j \leq n, 0 \leq p \leq k)$ 表示将 X 和 Y 的子序列 $X[1:i]$ 和 $Y[1:j]$ 合并成 Z 的子序列 $Z[1:p]$ 需要删除的最小元素集合，状态转移方程为

$$dp[i][j][p] = min \begin{cases} dp[i-1][j][p-1], & X_i = Z_p \\ dp[i][j-1][p-1], & Y_j = Z_p \\ dp[i-1][j][p], \text{ do } dp[i][j][p].push(X_i), & \\ dp[i][j-1][p], \text{ do } dp[i][j][p].push(Y_j), & \\ dp[i][j][p-1], \text{ do } dp[i][j][p].push(Z_p), & \end{cases}$$

$dp[m][n][k]$ 即为问题的解。算法按照 $i = 1 \ldots m$, $j = 1 \ldots n$, $p = 1 \ldots k$ 的顺序遍历。

该算法共计算 $\mathrm{O}(mnk)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$ ，故总的时间复杂度为 $\mathrm{O}(mnk)$ ，空间复杂度为 $\mathrm{O}(mnk)$ 。

13.12 重建合法单词序列

1）

将题目中的 $dict(w)$ 映射到 $dict(i,j)$，映射方法为 $w=s[i:j]$。令 $dp[i]\ (0\le i\le n)$ 表示子串 $s[1:i]$ 能否重建成合法单词组成的序列，状态转移方程为

$$dp[i]=\begin{cases}\bigvee(dp[k]\wedge dict(k+1,i))\ (0\le k<i), & i>0\\ 0, & i=0\end{cases}$$

$dp[n]$ 即为问题的解。算法从上到下遍历。

该算法共计算 $\mathrm{O}(n)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(n)$，故总的时间复杂度为 $\mathrm{O}(n^2)$，空间复杂度为 $\mathrm{O}(n)$。

2）

只需在算法中添加 $w[i]$ 记录前一个合法单词组成的序列，在计算 $dp[i]$ 时同步更新。状态转移方程为

$$w[i]=\begin{cases}s[k+1,i], & dp[k]\wedge dict(k+1,i)\\ \text{空}, & i=0\end{cases}$$

13.13 回文字符串

1）最长回文子序列

假设给定字符串为 $T[n]$，令 $dp[i][j]\ (1\le i\le j\le n)$ 表示子串 $T[i:j]$ 的最长回文子序列长度，状态转移方程为

$$dp[i][j]=\begin{cases}max\begin{cases}dp[i+1][j],\\ dp[i][j-1],\\ dp[i+1][j-1]+2I\{x_i=y_j\}\end{cases} & \text{if}\quad i<j\\ 0, & \text{if}\quad i=j\end{cases}$$

$dp[1][n]$ 即为问题的解。算法先从下到上、再从左到右遍历。

该算法共计算 $\mathrm{O}(n^2)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$，故总的时间复杂度为 $\mathrm{O}(n^2)$，空间复杂度为 $\mathrm{O}(n^2)$。

2）拆分最少回文数量

假设给定字符串为 $T[n]$，布尔函数 $isP(i,j)$ 用于判断子串 $T[i:j]$ 是否为回文，时间复杂度为 $\mathrm{O}(n)$。令 $dp[i][j]\ (1\le i<j\le n)$ 表示子串 $T[i:j]$ 可以拆分的最少回文数量，状态转移方程为

$$dp[i][j]=\begin{cases}min_{k=i+1}^{j-2}\{dp[i][k]+dp[k+1][j]\}, & isP(i,j)=false\\ 1, & isP(i,j)=true\end{cases}$$

$dp[1][n]$ 即为问题的解。算法先从下到上、再从左到右遍历。

该算法共计算 $\mathrm{O}(n^2)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(n)$，故总的时间复杂度为 $\mathrm{O}(n^3)$，空间复杂度为 $\mathrm{O}(n^2)$。

13.15 零钱兑换问题的变体

1）

令 $dp[i][j]\ (1 \le i \le n,\ 0 \le j \le v)$ 表示能否仅用面值为 $x[1:i]$ 的硬币兑换金额 j，状态转移方程为

$$dp[i][j] = \begin{cases} dp[i-1][j] \vee dp[i][j-x_i], & i > 1 \text{且} j > 0 \\ true, & j = 0 \\ true, & i = 1 \text{且} j = ax_1 (a \in Z^+) \\ false, & otherwise \end{cases}$$

$dp[n][v]$ 即为问题的解。算法先从上到下、再从左到右遍历。

该算法共计算 $\mathrm{O}(nv)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$，故总的时间复杂度为 $\mathrm{O}(nv)$，空间复杂度为 $\mathrm{O}(nv)$。

2）

原命题同 13.2 整数子集合问题。

3）

令 $dp[i][j][p]\ (1 \le i \le n,\ 0 \le j \le v,\ 0 \le p \le k)$ 表示能否用最多 p 枚面值为 $x[1:i]$ 的硬币兑换金额 j，状态转移方程为

$$dp[i][j][p] = \begin{cases} dp[i-1][j][p] \vee dp[i][j-x_i][p-1], & i > 1,\ j > 0,\ p > 0 \\ false, & p = 0 \\ true, & j = 0 \\ true, & i = 1 \text{且} j = ax_1 (0 \le a \le p) \\ false, & otherwise \end{cases}$$

$dp[n][v][k]$ 即为问题的解。法按照 $i = 1 \ldots m,\ j = 1 \ldots v,\ p = 1 \ldots k$ 的顺序遍历。

该算法共计算 $\mathrm{O}(nvk)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$，故总的时间复杂度为 $\mathrm{O}(nvk)$，空间复杂度为 $\mathrm{O}(nvk)$。

13.16 最小顶点覆盖

假设 $|V| = n$，令 $dp_+[i]\ (1 \le i \le n)$ 表示以 v_i 为根的子树中包含 v_i 的最小顶点覆盖的大小，$dp_-[i]\ (1 \le i \le n)$ 表示以 v_i 为根的子树中不包含 v_i 的最小顶点覆盖的大小，状态转移方程为

$$dp_+[i] = \begin{cases} 1 + min_{v_j \text{是} v_i \text{的子节点}}(dp_+[j], dp_-[j]), & v_i \text{不是叶节点} \\ 1, & v_i \text{是叶节点} \end{cases}$$

$$dp_-[i] = \begin{cases} min_{v_j \text{是} v_i \text{的子节点}}(dp_+[j]), & v_i \text{不是叶节点} \\ \text{不存在}, & v_i \text{是叶节点} \end{cases}$$

$min(dp_+[root], dp_-[root])$ 即为问题的解。算法从叶节点逐层向上遍历。

该算法共计算 $\mathrm{O}(n)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$，故总的时间复杂度为 $\mathrm{O}(n)$，空间复杂度为 $\mathrm{O}(n)$。

13.18 最小惩罚行程规划

令 $dp[i]\,(1 \leq i \leq n)$ 表示停留在 a_i 的最小总惩罚，$\Delta(i,j) = a_j - a_i\,(1 \leq i < j \leq n)$，状态转移方程为

$$dp[i] = \begin{cases} min_{\Delta(k,i)<=200}\{dp[k] + (200 - \Delta(k,i))^2\}, & i > 1 \\ (200 - a_1)^2, & i = 1 \end{cases}$$

$dp[n]$ 即为问题的解。算法从左到右遍历。

该算法共计算 $\mathrm{O}(n)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(n)$，故总的时间复杂度为 $\mathrm{O}(n^2)$，空间复杂度为 $\mathrm{O}(n)$。

13.20 最小代价进油方案

TODO

13.23 最友好宴会邀请名单

假设上下级关系树为 $G = (V, E)$，$|V| = n$，$a[1, \ldots, n]$ 表示 $v[1, \ldots, n]$ 的友好度。令 $dp_+[i]\,(1 \leq i \leq n)$ 表示以 v_i 为根的子树中包含 v_i 的友好度评分总和，$dp_-[i]\,(1 \leq i \leq n)$ 表示以 v_i 为根的子树中不包含 v_i 的友好度评分总和，状态转移方程为

$$dp_+[i] = \begin{cases} a_i + \sum_{v_j \text{ 是 } v_i \text{ 的子节点}}(dp_-[j]), & v_i \text{ 不是叶节点} \\ a_1, & v_i \text{ 是叶节点} \end{cases}$$

$$dp_-[i] = \begin{cases} \sum_{v_j \text{ 是 } v_i \text{ 的子节点}} max(dp_+[j], dp_-[j]), & v_i \text{ 不是叶节点} \\ 0, & v_i \text{ 是叶节点} \end{cases}$$

$max(dp_+[root], dp_-[root])$ 即为问题的解。算法从叶节点逐层向上遍历。

该算法共计算 $\mathrm{O}(n)$ 个子问题，每个子问题的时间复杂度为 $\mathrm{O}(1)$，故总的时间复杂度为 $\mathrm{O}(n)$，空间复杂度为 $\mathrm{O}(n)$。

13.24 最短外卖路线

TODO