计算机学院《算法设计与分析》第二次作业

October 30, 2024

1 不完美字符串问题

对于一个长度为 n 的仅由 0 和 1 构成的字符串 s，其不完美度定义为所有相邻字符对中互不相同的字符对数量的和。比如，长度为 4 的字符串 “0010” 的不完美度为 2，这是由 01 和 10 两个相邻数对贡献得到的。形式化地，字符串 s 的不完美度如下式：

$$f(s) = \sum_{i=1}^{n-1} [s_i \neq s_{i+1}]$$

（[] 运算的结果取决于其中布尔表达式的真值。若为假，其值为 0，若为真，其值为 1）

现给定一个长度为 n 的字符串 s，每一位仅由 0、1 和? 组成。你需要将其中的每一个? 替换成 0 或者 1 ，且最小化字符串 s 的不完美度 $f(s)$。

例如，给定的字符串为 “?1?0”，一种使不完美度最小的替换方案为 “1110”，其不完美度为 1，可以证明，没有其他替换方案能使不完美度更小。

请使用动态规划算法求解最小的不完美度，并给出一个替换方案。请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。

1.1 状态设计

令 $dp[i][j]$ 表示到第 i 个字符为止，且该字符为 j(0 或 1) 时的最小不完美度。

1.2 状态转移

由不完美度的计算公式可知，$dp[i]$ 的转移过程实际上只与 $s[i]$ 和 $s[i-1]$ 有关，类似于我们在概率与统计课程中学到的马尔可夫过程。由于部分位置字符串中字符确定，我们无法改变该字符，因此对于这种情况，我们采用以下处理方式:

$$s[i] = \text{‘0’}$$

$$dp[i][1] = \infty$$

由此，我们得到状态转移方程：

$$dp[i][0] = \begin{cases} \min(dp[i-1][0],\ dp[i-1][1]+1)\ ,\ s[i] \neq \text{‘1’} \\ \infty\ ,\ s[i] = \text{‘1’} \end{cases}$$

$$dp[i][1] = \begin{cases} \min(dp[i-1][0]+1,\ dp[i-1][1])\ ,\ s[i] \neq \text{‘0’} \\ \infty\ ,\ s[i] = \text{‘0’} \end{cases}$$

1.3 边界条件

字符串首个字符没有前置约束，因此有:

$$dp[i][1] = \begin{cases} 0\ ,\ s[i] \neq \text{‘0’} \\ \infty\ ,\ s[i] = \text{‘0’} \end{cases}$$

$$dp[i][0] = \begin{cases} 0\ ,\ s[i] \neq \text{‘1’} \\ \infty\ ,\ s[i] = \text{‘1} \end{cases}$$

1.4 目标状态

目标状态即字符串扫描完成时字符串末尾取 $0, 1$ 得到的 $\min(dp[n][0],\ dp[n][1])$ 此外，为了得到最终的替换字符串，我们需再遍历一遍 dp 数组，对于字符串 s，若 $s[i]$ 为 ‘?’，通过动态规划的各位置最优性，我们将 $s[i]$ 替换为 $res[i]$

$$res[i] = \begin{cases} \text{‘1’}\ ,\ dp[i][0] \geq dp[i][1] \\ \text{‘0}\ ,\ dp[i][0] < dp[i][1] \end{cases}$$

1.5 时间复杂度分析

通过递推式我们可知，总状态是 $O(n)$ 级别的，需遍历一次字符串，每次状态转移时间复杂度 $O(1)$，最终得到替换方案需遍历一遍字符串，遍历操作时间复杂度 $O(n)$，故总时间复杂度为 $O(n)$。

1.6 不完美字符串伪代码

Algorithm 1: imperfect($s[1..n]$)

Input: 输入的字符串 $s[n]$

Output: 一个正整数 sum，为最小的不完美度；替换后的字符串 $res[n]$，为一种可行的替换方案

```
1  global s[n]
2  初始化数组 dp[1..n][0..1]
3  dp[1][0] ← 0   dp[1][1] ← 0;
4  if s[1] = '0' then
5  |  dp[1][1] ← ∞;
6  end
7  if s[1] = '1' then
8  |  dp[1][0] ← ∞;
9  end
10 for i ← 2 to n do
11 |  if s[i] = '0' then
12 |  |  dp[i][0] ← min(dp[i − 1][0], dp[i − 1][1] + 1);
13 |  |  dp[i][1] ← ∞;
14 |  end
15 |  if s[i] = '1' then
16 |  |  dp[i][1] ← min(dp[i − 1][1], dp[i − 1][0] + 1);
17 |  |  dp[i][0] ← ∞;
18 |  else
19 |  |  dp[i][0] ← min(dp[i − 1][0], dp[i − 1][1] + 1);
20 |  |  dp[i][1] ← min(dp[i − 1][1], dp[i − 1][0] + 1);
21 |  end
22 end
23 sum ← min(dp[n][0], dp[n][1]);
24 for i ← 1 to n do
25 |  if s[i] = '?' then
26 |  |  if dp[i][0] > dp[i][1] then
27 |  |  |  res[i] ← '1';
28 |  |  else
29 |  |  |  res[i] ← '0';
30 |  |  end
31 |  else
32 |  |  res[i] ← s[i];
33 |  end
34 end
35 return sum, res
```

2 鲜花组合问题

花店共有 n 不同颜色的花，其中第 i 种库存有 a_i 枝，现要从中选出 m 枝花组成一束鲜花。

请设计算法计算有多少种组合一束花的方案，请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。（两种方案不同当且仅当存在一个花的种类 i，两种方案中第 i 种花的数量不同）

2.1 状态设计

该问题类似背包问题，对于每种花，最多可以选取 a_i 件，问题转化为总共购买 m 件商品的前提下，有多少种购买方案。

状态 $dp[i][j]$ 表示考虑前 i 种花，已经选取了 j 枝花的方案数。

2.2 状态转移及其优化

即只考虑了前 $i-1$ 种花的情况下，枚举选取多少枝当前种类的花，进行转移，有：

$$dp[i][j] = \sum_{k=0}^{\min(a_i,j)} dp[i-1][j-k]$$

我们发现这样的转移方法在状态转移时时间复杂度为 $O(m)$，时间复杂度过高

通过观察，我们不难发现，状态转移方程实际上是一个计算**前缀和**的过程，考虑引入前缀和数组 $sum[n]$，sum 迭代方程有：

$$sum[j] = dp[i-1][j] + sum[j-1]$$

由此改写状态转移方程为：

$$dp[i][j] = sum[j] - sum[j - \min(ai, j) - 1]$$

此时状态转移时间复杂度为 $O(1)$。

2.3 边界条件与目标状态

起始条件为 $dp[0][0] = 1$ 最终答案为 $dp[n][m]$，即考虑全部 n 种花，恰好选取了 m 支的方案数量。

2.4 时间复杂度分析

由前文分析知，状态数为 $n \times m$，每次状态转移时间复杂度为 $O(1)$，因此复杂度为 $O(nm)$。

2.5 作业题目 3 伪代码

Algorithm 2: flowers($A[1..n]$)

Input: 鲜花库存数量 $A[1..n]$，选取鲜花数量 m

Output: 鲜花组合方案数量

```
global A[1..n]
dp[1..n][1..m]
sum[1..m]
dp[0][0] ← 1;
for i ← 1 ton do
    sum[0] ← dp[i−1][0];
    for j ← 1 to m do 求前缀数组
        sum[j] ← sum[j−1] + dp[i−1][j];
    end
    for j ← 1 to m do
        if j − min(a_i, j) − 1 ≥ 0 then
            dp[i][j] ← sum[j]−sum[j − min(a_i, j) − 1];
        else
            dp[i][j] ← sum[j];
        end
    end
end
return dp[n][m]
```

3 最长公共上升子序列问题

对两个序列 A 和 B，序列 s 为 A 和 B 的公共上升子序列，当且仅当 s 是 A 和 B 的公共子序列，且 s 是上升子序列（$s_i < s_{i+1}, \forall 1 \le i < |s|$）。

对两个序列 A 和 B，若某个序列 s 是 A 和 B 的公共上升子序列，且对于任意的 A 和 B 的公共上升子序列 t，都有 $|t| \le |s|$，那么 s 称为 A 和 B 的最长公共上升子序列。

例如，给定两个序列 $\langle 2,3,1,6,5,4,6\rangle$ 和 $\langle 1,3,5,6\rangle$，其一个最长公共上升子序列为 $\langle 3,5,6\rangle$。

给定两个长度为 n 的序列 A 和 B。请设计一个动态规划算法，求它们的最长公共上升子序列的长度。

3.1 状态设计

我们假设两个数组分别为 $a[n], b[n]$

状态 $dp[i][j]$ 表示所有由第一个序列的前 i 个字母，和第二个序列的前 j 个字母构成的以 $b[j]$ 结尾的所有公共上升子序列中长度的最大值

3.2 状态转移

对于本题所给状态，状态转移的依据主要有两点，**一是公共，二是上升**，针对这两点，我们容易推导出状态转移方程：

$$dp[i][j] = dp[i-1][j], (a_i \neq b_j)$$

$$dp[i][j] = \max_{0 \le k < j, b_k < a_i} dp[i-1][k] + 1, (a_i = b_j)$$

说明:

如果 $A_i \neq B_j$，那么以 B_j 结尾的最长公共上升子序列还是之前那个（因为 A_i 和 B_j 不相等，所以不满足“公共”这个条件，因此长度不变）。所以：

$$dp[i][j] = dp[i-1][j], (a_i \neq b_j)$$

当 $A_i == B_j$ 时：如果 $A_i == B_j$，那么出现了新的公共元素。这时，我们需要考虑是否满足“上升”这个条件。

为此，循环 k 次，从满足上升条件的之前的最优状态转移过来。所以：

$$dp[i][j] = \max_{0 \leq k < j, b_k < a_i} dp[i-1][k] + 1, (a_i = b_j)$$

3.3 边界条件与目标状态

起始条件显然有 $dp[0] = 0$ 最终结果为 $Max = \max_{1 \leq i \leq n} dp[n][i]$，$Max$ 即为所求最长公共上升子序列的长度。

3.4 DP 优化

通过状态转移方程我们知道，进行状态转移时，时间复杂度为 $O(k)$，考虑到 k 的数量级，我们知道，该方法的总体时间复杂度为 $O(n^3)$，这显然太过复杂。

在转移过程中，我们把满足 $0 \leq k < j, b_k < a_i$ 的 k 的集合称为 $dp[i][j]$ 的决策集合，记作 $S(i, j)$。注意到在状态转移的循环中，A 的下标 i 是不变的，这意味着 $B_k < A_i$ 是固定的。因此，当变量 j 增加 1 时，k 的取值范围从 $0 \leq k < j$ 变为 $0 \leq k < j + 1$，即 j 有可能进入新的决策集合。所以，我们只需要花费 $O(1)$ 来检查 $B_j < A_i$ 是否满足，因为已经在决策集合中的元素不会删除。

这意味着，我们可以通过记录并实时更新 $\max_{0 \leq k < j, b_k < a_i} dp[i-1][k]$ 的值，来将状态转移的时间复杂度降为 $O(1)$，从而将整体的时间复杂度降为 $O(n^2)$。

3.5 时间复杂度分析

由上述 DP 优化部分可知，状态数为 n^2，每次状态转移时间复杂度为 $O(1)$，因此复杂度为 $O(n^2)$。

3.6 最长公共上升子序列伪代码

Algorithm 3: LCIS($A[1..n], B[1..n]$)

Input: 两个序列 $A[1..n], B[1..n]$

Output: 一个正整数 res，表示最长公共上升子序列的长度

```
1  global A[1..n], B[1..n]
2  初始化数组dp[1..n][1..n]
3  for i ← 1 to n do
4      tmp ← 0;
5      if B[0] < A[i] then
6          tmp ← dp[i − 1][0];
7      end
8      for j ← 1 to n do
9          if A[i] = B[j] then
10             dp[i][j] ← tmp + 1;
11             if res < dp[i][j] then
12                 res ← dp[i][j];
13             end
14         else
15             dp[i][j] ← dp[i − 1][j];
16         end
17         if B[j] < A[i] then 更新该决策集合 tmp 的值
18             tmp ← max(tmp, dp[i − 1][j]);
19         end
20     end
21 end
22 return res
```

4 叠塔问题

给定 n 块积木，编号为 1 到 n。第 i 块积木的重量为 w_i（w_i 为整数），硬度为 s_i，价值为 v_i。

现要从中选择部分积木垂直摞成一座塔，要求每块积木满足如下条件:

若第 i 块积木在积木塔中，那么在其之上摆放的所有积木的重量之和不能超过第 i 块积木的硬度。

试设计算法求出满足上述条件的价值和最大的积木塔，输出摆放方案和最大价值和。请描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。

4.1 决策逻辑分析

本题条件较多，直接设计 DP 状态不易，我们先分析叠塔的相关条件:

若第 i 块积木在积木塔中，那么在其之上摆放的所有积木的重量之和不能超过第 i 块积木的硬度

通过这个条件我们知道，如果现在已经放置了 W 重量的积木，现要第将 i 块积木插入塔底，则有 $W-w_i \leq s_i$

先假设有 i, j 两块积木要插入塔底，则假定最优策略是 j 在 i 的底部，则必有:

$$s_i < W + w_j$$

$$s_j \geq W + w_i$$

由此推知 $s_i + w_i < s_j + w_j$，因此，通过将积木按 $s_i + w_i$ 从小到大的顺序排序，便可自顶向下得到最优方案

4.2 状态设计

我们将积木按照 $s_i + w_i$ 从小到大排序，令 $dp[i][W]$ 表示，当前已经考虑了前 i 个积木，搭建了重量为 W 的积木塔的最大价值。

4.3 状态转移

由前文分析可知，只有满足 $W-w_i \le s_i$，才能进行状态转移，此时可以将 (s_i, w_i, v_i) 这块积木插入到状态 $dp[i-1][W-w_i]$ 对应的塔的最底下。

否则不满足放置条件，只能考虑舍弃这块积木无法放置，即继承状态 $dp[i-1][W]$。

故状态转移方程为：

$$dp[i][W] = \begin{cases} dp[i-1][W] & W - w_i > s_i \\ \max(dp[i-1][W], dp[i-1][W-w_i] + v_i) & W - w_i \le s_i \end{cases}$$

4.4 边界条件与目标状态

我们令 $select[n][w]$ 表示 $dp[i][w]$ 状态时有哪些积木块被选择了。

对于没有考虑任何积木时，即 $dp[0][0 \sim \sum_i w_i]$，获得积木塔的价值都是 0，且 $select[0][0 \sim \sum_i w_i]$ 均为 $\emptyset$。

最优方案为 $select[n][w_{res}]$，对应的最大价值为 $dp[n][w_{res}]$，其中 $w_{res} = arg\max_w dp[n][w]$。

4.5 摆放方案记录

由前文分析知只有在 $W - w_i \le s_i$ 的条件下，且选择了决策方案 $dp[i-1][W-w_i] + v_i$ 时才会选择积木块 (s_i, w_i, v_i)。则在转移选了决策 $dp[i-1][W-w_i] + v_i$ 时有

$$select[i][W] = select[i-1][W-w_i] \cup \{(s_i, w_i, v_i)\},$$

否则 $select[i][W]$ 继承 $select[i-1][W]$ 。

4.6 时间复杂度分析

算法的时间复杂度分析如下：

状态数为 $O(n \times \sum_i w_i)$，其中 n 是积木的数量，$\sum_i w_i$ 是所有积木重量的总和。每个状态需要 $O(1)$ 的时间进行转移。排序的时间复杂度为 $O(n \log n)$。

由于 $\sum_i w_i > n$，因此总的时间复杂度主要由状态转移部分决定，即总的复杂度为 $O(n \times \sum_i w_i)$。

4.7 作业题目 5 伪代码

Algorithm 4: tower(n, w_n, s_n, v_n)

Input: n 块积木，其中第 i 块积木的重量为 w_i，硬度为 s_i，价值为 v_i

Output: 摆放方案和最大价值和

初始化$dp[0..n][0..\sum_i w_i] \leftarrow 0 \quad select[0..n][0..\sum_i w_i] \leftarrow \emptyset$
$sort(w_n, s_n, v_n)$ *in ascending order by* $(w_i + s_i)$
$sum \leftarrow 0$; 利用 sum 记录当前最大塔顶积木的值
for $i \leftarrow 1$ *to* n **do**
  for $w \leftarrow w_i$ *to* sum **do**
    if $w - w_i \leq s_i$ **then**
      if $dp[i-1][w-w_i] + v_i > dp[i-1][w]$ **then**
        $dp[i][w] \leftarrow dp[i-1][w-w_i] + v_i$;
        $select[i][w] = select[i-1][w-w_i] \cup \{(s_i, w_i, v_i)\}$;
      end
    else
      $dp[i][w] \leftarrow dp[i-1][w]$;
      $select[i][w] = select[i-1][w]$;
    end
  end
  $sum \leftarrow sum + w_i$;
end
$w_{res} = arg\max_w dp[n][w]$;
return $dp[n][w_{res}], select[n][w_{res}]$

5 最小划分问题

对一个序列 a 上的某两个数 a_i 和 a_j，若 $i < j$ 且 $a_i \neq a_j$，则称 (i, j) 为一个不同数对。一个序列 a 的不同数对数为序列中所有不同数对的个数之和。例如，序列 $\langle 1, 0, 1, 0\rangle$ 的不同数对有 $(1, 2), (1, 4), (2, 3), (3, 4)$ 四个。

给你一个长度为 n 的正整数序列 a 和一个正整数 k，满足序列中的任意一个数 $a_i \in [1, n]$。请你将其划分为 k 个子段，最小化每个子段最小不同数对数相加的和。你只需要回答这个最小的和。

例如，给定序列为 $\langle 1, 1, 3, 3, 3, 2, 1\rangle$ 和参数 $k = 3$，则答案为 1，对应的一个划分方法为 $\langle 1, 1\rangle, \langle 3, 3, 3\rangle, \langle 2, 1\rangle$。

请使用动态规划求解该问题，描述算法的核心思想，给出算法伪代码并分析其对应的时间复杂度。

5.1 决策逻辑分析

本题不同数对计数条件较为复杂，前后数字相互影响，似乎无法通过动态规划解决问题。

但是，考虑到数对内部 (i, j) 的有序性，我们容易得到以下结论，令序列为 $A[n]$，则 $A[i]$ 所带来的不同数对个数为 $num[i] = (i-1) + rec[A[i]]$，其中 $rec[n]$ 为 $0 \to (i-1)$ 中 n 的个数。

由此我们得到了一个时间复杂度为 $O(n)$ 的求从 $A[i]$ 到 $A[j]$ 中不同数对个数的方法，设该方法为 $pairs(l, r)$。

由此，我们可以利用**区间 DP** 的思想进行动态规划。

5.2 状态设计

我们令 $dp[i][j]$ 表示将前 i 个元素划分为 j 个子段的最小各段不同数对数之和，这样，我们可以通过枚举不同划分的分界点来实现动态规划。

5.3 状态转移

由于动态规划的各过程最优性，事实上，我们只需要考虑所求状态的前一个状态即可，由此我们故状态转移方程为：

$$dp[i][j] = \min_{j-2 \le m < i} \{dp[i][j], dp[m][j-1] + pairs(m+1, i)\}$$

此时总状态数为 $O(n \times k)$，计算状态数时内层需遍历 i，时间复杂度为 $O(n)$，状态转移时间复杂度为 $O(n)$，故总体时间复杂度为 $O(n^3 \times k)$。

5.4 边界条件与目标状态

我们令所有 $dp[i][j]$ 初始化为 ∞，我们所求最小值即为 $dp[n][k]$。

5.5 DP 优化

通过前文分析我们可以得到基础时间复杂度为 $O(n^2 \times k)$，由于状态转移时间复杂度为 $O(n)$ 级别，造成总体时间复杂度过高，这显然是我们无法接受的。

观察 $pairs$ 函数的表达式，我们发现，由于 $pairs$ 计数只与前缀和有关，与后缀无关，我们考虑采用前缀和化简 $pairs$ 函数，减少状态转移这一步的时间复杂度。

我们设计两个辅助数组 $rec[n][n], pre[n][n]$，$rec[i][j]$ 表示到 $A[j]$ 为止，j 的个数，$pre[i][j]$ 表示从 $A[i]$ 到 $A[j]$ 中不同数对个数。且对于 rec 和 pre 有：

$$pre[i][j] = pre[i][j-1] + (j - i - rec[j-1][a[j]] + rec[i-1][a[j]]);$$

对于 rec，每次更新需要 2 步：

1. 继承 $i-1$ 的 rec，时间复杂度为 $O(n)$
2. 更新：$rec[i][a[i]] \leftarrow rec[i][a[i]] + 1$，时间复杂度为 $O(1)$

总体时间复杂度为 $O(n^2)$

对于 pre，完成更新需要一个**二重循环**，循环中每次更新时间复杂度为 $O(1)$，故总体时间复杂度为 $O(n^2)$

通过进行这样的预处理，我们成功的将 $pairs$ 函数的时间复杂度降为 $O(1)$，此时状态转移方程为

$$dp[i][j] = \min_{j-2 \le m < i} \{dp[i][j], dp[m][j-1] + pre[m+1][i]\}$$

5.6 时间复杂度分析

由前文分析可知：

预处理时间复杂度为 $O(n^2)$，总状态数为 $O(n \times k)$，计算状态数时内层需遍历 i，时间复杂度为 $O(n)$，状态转移时间复杂度为 $O(1)$，故 **DP 过程**时间复杂度为 $O(n^2 \times m)$，总体时间复杂度为 $O(n^2 \times m)$。

5.7 作业题目 5 伪代码

Algorithm 5: preProcess($A[1..n]$)

Input: 序列 $A[1..n]$

Output: 前缀和数组 $pre[1..n][1..n]$

初始化$rec[1..n][1..n] \leftarrow 0$, $sum[1..n] \leftarrow 0$
$tot \leftarrow 0$
for $i \leftarrow 1$ *to* n **do**
  $rec[i] \leftarrow rec[i-1]$; //继承 $i-1$ 的 rec
  $tot \leftarrow tot + (i-1) - rec[i][A[i]]$;
  $sum[i] \leftarrow sum[i] + tot$;
  $rec[i][a[i]] \leftarrow rec[i][A[i]] + 1$;
end
for $i \leftarrow 1$ *to* n **do** 预处理 $pre[1][1..n]$
  $pre[1][i] \leftarrow sum[i]$;
end
for $i \leftarrow 2$ *to* n **do**
  for $j \leftarrow (i+1)$ *to* n **do**
    $pre[i][j] = pre[i][j-1] + (j - i - rec[j-1][A[j]] + rec[i-1][A[j]])$;
  end
end
return $pre[1..n][1..n]$

Algorithm 6: minDifferentPairs($A[1..n], k$)

Input: 序列 $A[1..n]$，划分数 k

Output: 最小不同数对数

初始化$dp[1..n][1..k] \leftarrow \infty$
$dp[0][0] \leftarrow 0$;
$pre[1..n][1..n] \leftarrow preProcess(A[1..n])$; //完成前缀和初始化
for $i \leftarrow 1$ *to* n **do**
  $dp[i][1] \leftarrow pre[1][i]$;
end
for $i \leftarrow 2$ *to* n **do** 开始 DP
  for $j \leftarrow 2$ *to* $\min(k, i+1)$ **do**
    for $m \leftarrow (j-2)$ *to* $(i-1)$ **do**
      $dp[i][j] \leftarrow \min(dp[i][j], dp[m][j-1] + pre[m+1][i])$;
    end
  end
end
return $dp[n][k]$

5.8 进一步的 DP 优化

在完成上述算法的设计之后，我们不禁要问，遍历一个状态需要 $O(n)$ 的复杂度，似乎太高了，我们是否有方法能够更快的完成**最优分割点的查找?**

事实上，这是完全可以的，观察算法，我们在更新 $dp[i][k]$ 时，当 k 固定，我们要通过遍历 $0 \to i$ 来完成最优分割点的查找，通过分治思想，我们能够想到，可以通过二分查找的方式提高查找效率，假设我

们将数组分为两半，得到左侧数组最优分割点为 $optL$，右侧最优分割点为 $optR$，那么在处理最优分割点可能位于 $optL \to optR$ 时，再通过遍历 $optL \to optR$ 来完成。

显然，由前文的时间复杂度较高的算法的分析部分我们知道，通过维护类似前文 rec 数组这样记录到当前位置各数字出现次数，这样时间复杂度为 $O(n)$ 的合并过程是可以完成的。

5.9 优化后的 DP 转移

我们假设当前分治区间两端分别为 l, r，当前所得前一步划分左部最优为 $optL$，右部为 $optR$，则有状态转移方程：

$$dp[mid][k] = \min_{optL \le i \le \min(mid-1, optR)} dp[i][k-1] + (allPairs - samePairs)$$

其中 $allPairs = \frac{(mid-i)\times(mid-i-1)}{2}$，$samePairs$ 可以通过不断排除 $a[i]$ 前数组带来的影响来动态更新。

最终我们可以得到 $dp[mid][k]$。

5.10 优化后时间复杂度分析

我们知道，之前经简单优化的算法时间复杂度为 $O(n^2 \times k)$，对于求解 $dp[i][k]_{0\le i\le n}$ 时间复杂度为 $O(n^2)$。

我们采用分治算法，假设求解 $dp[i][k]_{0\le i\le n}$ 时间复杂度为 $T(n)$，有

$$T(n) = 2 \times T(\frac{n}{2}) + O(n)$$

由主定理知 $T(n) = O(n \times \log n)$，此时算法总时间复杂度为 $O(n \times \log n \times k)$。

5.11 优化后 DP 伪代码

Algorithm 7: optMinDifferentPairs($A[1..n], k$)

Input: 序列 $A[1..n]$，划分数 k

Output: 最小不同数对数

```
1 global A[1..n]
2 初始化dp[0..n][0..k] ← 0   其余元素为∞
3 for i ← 1 to k do 分别求解 dp[1..n][k]
4 |   SOLVE(t, 1, n, 0, n − 1);
5 end
6 return dp[n][k]
```

Algorithm 8: SOLVE($k, l, r, optL, optR$)

Input: 划分数 k，当前分治区间端点 l, r，当前最优分割点 $optL, optR$

Output: 无

```
if l > r then
    return
end
mid ← ⌊(l+r)/2⌋;
minPairs ← ∞;
pos ← −1;
初始化freq[1..n] ← 0
samePairs ← 0;
maxL ← optL + 1;
for maxR ← mid to maxL do 初步求解区间全局 samePairs
    samePairsNum ← samePairsNum + freq[A[maxR]];
    freq[A[maxR]]←freq[A[maxR]] + 1;
end
for i ← optL to min(mid − 1, optR) do
    freq[A[i + 1]] ← freq[A[i + 1]] − 1;
    samePairs ← samePairs − freq[A[i + 1]];
    allPairs ← ((mid−i)×(mid−i−1))/2;
    nowPairst ← dp[i][k − 1] + (allPairs − samePairs);
    if nowPairs < minPairs then
        minPairs ← nowPairs;
        pos ← i;
    end
end
dp[mid][k] ← minPairs;
SOLVE(k, l, mid − 1, optL, pos);
SOLVE(k, mid + 1, r, pos, optR);
```