# 算法作业

丁元杰 17231164

2019 年 11 月 10 日

## 1 最长回文子序列问题

### 算法思路

对原序列$S$和原序列的逆序列$S^R$求出最长公共子序列（LCS），可以求得一个长度为$m$的串$T[1..m]$。现在断言$m$即为原问题（LPS）所求序列的长度，且由串$T[1..m/2]+(T[m/2+1..m])^R$拼接而成的新串是原问题的一个解。由此即可将LPS转化为一个等价的LCS问题，由现有的$O(n^2)$的LCS算法求解得到。

其中，序列$S[1..n]$的逆序列$S^R[1..n]$定义为

$$S^R[i] = S[n+1-i],\ \forall 1 \le i \le n$$

下面给出上面所给断言的形式化描述和严格证明。

**命题 1** $S[1..n]$与$S^R[1..n]$可求得一个最长公共子序列，即存在一个指标序列$a[1..m]$，$b[1..m]$，使得$S[a[i]] = S^R[b[i]],\ \forall 1 \le i \le m$。并且存在另一个$S$的指标序列$c[1..m]$，使得$S[c[i]] = S[c[m+1-i]],\ \forall 1 \le i \le m$，并且有

$$c[i] = \begin{cases} a[i] & ,\text{if } i \le m/2 \\ n+1-b[m+1-i] & ,\text{if } i \ge m/2+1 \end{cases}$$

并且由此指标集限定的$S$的子序列是它的一个最长回文子序列。

**证** 先证明这个串是一个回文串。可知

$$c[i] = \begin{cases} a[i] & ,\text{if } i \le m/2 \\ n+1-b[m+1-i] & ,\text{if } i \ge m/2+1 \end{cases}$$

那么，$\forall 1 \le i \le m/2$

$$\begin{aligned} S[c[m+1-i]] &= S[n+1-b[m+1-(m+1-i)]] \\ &= S[n+1-b[i]] \\ &= S^R[b[i]] \\ &= S[a[i]] \\ &= S[c[i]] \end{aligned}$$

因此，$S[c[i]]$必然是一个回文子序列。

再证明这个回文串是最长的。如果存在一个更长的子序列，则表示存在一个指标序列$c'[1..m']$，使得$m' > m$且$S[c'[i]] = S[c'[m'+1-i]], \forall 1 \le i \le m'$。那么可以构造$a'[i] = c'[i]$，以及$b'[i] = n+1-c'[m'+1-i]$，那么

$$\begin{aligned} S^R[b'[i]] &= S^R[n+1-c'[m'+1-i]] \\ &= S[c'[m'+1-i]] \\ &= S[c'[i]] \\ &= S[a[i]] \end{aligned}$$

使得$S[a'[i]] = S^R[b'[i]], \ \forall 1 \le i \le m'$。我们成功构造了一个长$m'$的序列，使得其是$S$和$S^R$的公共子序列，此与原假设中关于公共子序列的最长性质矛盾。

由于这是一个构造性的证明，在证明的同时给出了构造的方案，因此可以方便地由求LCS（求解方案）的算法直接得到LPS（求解方案）的算法。

### 伪代码

参见算法1

## 2 餐厅选址问题

### 算法思路及复杂度分析

**思路** 考虑动态规划的思想，设$f[i]$表示所选的所有地址中，位置最大为$m_i$的最大收益。设这个局部的最大收益的方案为一个集合$S_i \subseteq [1,n]$，是候选位置下标集的子集。则由假设，

$$i = \max S_i$$

并且根据题目中对于任意两个餐厅的距离限制，可以得出：

$$m_i > \max_{i \neq j \in S_i} m_j + k$$

那么我们可以给出转移方程：

$$f[i] = \max_{j<i \& m_j + k \le m_i} f[j] + p_i$$

其中如果max所应用的集合为空，那么max得到0的结果。

最终所求的答案，就是

$$ans = \max_i f[i]$$

这个递推的思路是，如果存在某一个局部的最优解，那么这个局部的最优解必定存在位置最大的地址。去掉这一个位置最大的地址后，剩下的子集必然是满足基本限制的最优解（否则，就可以通过调整获得更优的解法）。

**算法 1** 求序列$S$的最长回文子序列

**输入:** $S[1..n]$序列

**输出:** 原序列的最长回文子序列$T$

```
1: function LPS(S[1..n])
2:     S' ← REVERSE(S)
3:     f[1..n][1..n], g[1..n][1..n] 是两个数组
4:     f[0][0] ← 0, g[0][0] ← −1
5:     ans = 0
6:     anp = (−1, −1)
7:     for i = 1 → n do
8:         for j = 1 → n do
9:             if S[i] = S'[j] then
10:                f[i][j] ← f[i − 1][j − 1] + 1
11:                g[i][j] ← (i − 1, j − 1)
12:            else if f[i − 1][j] > max{f[i − 1][j − 1], f[i][j − 1]} then
13:                f[i][j] ← f[i − 1][j]
14:                g[i][j] ← (i − 1, j)
15:            else if f[i − 1][j − 1] > max{f[i − 1][j], f[i][j − 1]} then
16:                f[i][j] ← f[i − 1][j − 1]
17:                g[i][j] ← (i − 1, j − 1)
18:            else if f[i][j − 1] > max{f[i − 1][j], f[i − 1][j − 1]} then
19:                f[i][j] ← f[i][j − 1]
20:                g[i][j] ← (i, j − 1)
21:            end if
22:            if f[i][j] > ans then
23:                ans ← f[i][j]
24:                anp ← (i, j)
25:            end if
26:        end for
27:    end for
28:    T ← GET_ANSWER(S, g, ansp)
29:    return T
30: end function
31: function GET_ANSWER(S[1..n], g[1..n][1..n], ansp)
32:    T ← ''
33:    nowp ← ansp
34:    while nowp ≠ (−1, −1) do
35:        T+ = S[nowp.first]
36:        nowp ← g[nowp.first][nowp.second]
37:    end while
38:    return REVERSE(T)
39: end function
```

**朴素实现的复杂度** 接下来分析一下这个动态规划的朴素实现所需的时间复杂度。可以看出，总计有$f[1..n]$，共$n$个状态。我们的算法顺序求解$1..n$的状态。其中为了求得$f[i]$的答案，需要优先遍历所有$j : j < i$，检查其是否满足基本限制$m_j + k \leq m_i$，以及最优性。所以$f[i]$所花的时间是$i$。

那么最后，我们的算法时间复杂度是：

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} i \\ &= \frac{n(n+1)}{2} \\ &= O(n^2) \end{aligned}$$

所以朴素算法的时间复杂度是$O(n^2)$的。

**第一次优化** $O(n^2)$的复杂度不尽如人意，原因是算法虽然是单调无后效性地计算$f[i]$，较小的$f[j]$却会被反复查询，用于求解最优性。为了解决这个问题，可以引入另一个数组$g[i]$，表示$f[i]$的前缀最大值，即：

$$g[i] = \max_{j \leq i} g[i]$$

显然，$g[i]$数组随着$i$的增大非严格单调递增，那么只要我们能快速确定$i$状态可以转移的最大前驱$j^*$，就可以直接从$g[j^*]$得到我们需要的前驱状态。即：

$$j^* = \max_{j \leq i \& m_j + k \leq m_i} j$$

以及

$$f[i] = g[j^*] + p_i$$

其中，由于$m_j$的单调性，$j^*$可以通过二分查找，使用$O(\log i)$的时间求得。因此，此优化的最终复杂度变为：

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} \log i \\ &= \log n! \\ &\approx n \log n \\ &= O(n \log n) \end{aligned}$$

**最终优化** 使用双指针法优化（two pointers）。

记对状态$i$求出的$j^*$为$j_i$，那么由定义式：

$$j_i = \max_{j \leq i \& m_j + k \leq m_i} j$$

可以看出$j_i \leq j_{i+1}$，即$j_i$随$i$满足单调性（非严格的）。既然此单调性存在，就可以由$j_i$开始，向后依次枚举，直到$j_{i+l+1}$不再满足性质$m_{j_i+l+1} \leq m_i$时，取$j_{i+1} \leftarrow j_i + l$，即可得到$j_{i+1}$。

虽然此方法把二分查找更换为了线性扫描，但是由于$j_i$单调递增，总计的尝试次数不会超过$n$，因此无论左指针$j_i$还是右指针$i$，都只会为求解贡献$n$次计算。此方案的时间复杂度为$O(n)$

## 伪代码

参见算法3。

**算法 2** 求解此问题

**输入:** $m[1..n], p[1..n]$数组，$k$参数

**输出:** 最优方案的值$p_m$

```
1: function BestPlaces(m[1..n], p[1..n], k)
2:     f[1..n] ← 0
3:     g[0..n] ← 0
4:     l ← 0
5:     r ← 1
6:     while r ≤ n do
7:         while l + 1 < r&&m[l + 1] + k ≤ m[r] do
8:             l ← l + 1
9:         end while
10:        f[r] ← g[l] + p[r]
11:        g[r] ← max{f[r], g[r − 1]}
12:        r ← r + 1
13:    end while
14:    return g[n]
15: end function
```

表 1: 状态转移表格

| | 00 | 01 | 10 |
|---|---|---|---|
| 00 | 可 | 可 | 可 |
| 01 | 可 | 不可 | 可 |
| 10 | 可 | 可 | 不可 |

# 3 球队组建问题

## 算法思路

设计动态规划状态$f[i][j]$，其中$1 \le i \le n$，且$j \in \{0,1,2\}$，表示在前$i$列的所有方案中，最后一列的两人选择方案为$j$时的最优解。在这里，$j$的三个取值分别表示其对应二进制数的状态，对应二进制数位为0，表示不选择该人；反之为选择该人。

转移方向为$f[i][j]$到$f[i+1][j']$，合法的转移参见表

在合理的转移方向之间找到最优的转移，答案就在

$$ans = \max_{j \in \{0,1,2\}} f[n][j]$$

## 伪代码

## 复杂度分析

后面的DP部分，由于循环只对数组进行了$O(n)$次访问，因此总共复杂度为$O(n)$

**算法 3** 求解此问题

**输入:** $h[2][1..n]$

**输出:** 最优方案的值$ans$

```
1: function BestTeam(h[2][1..n])
2:     f[0..n][3]
3:     f[0][0] ← 0
4:     f[0][1] ← 0
5:     f[0][2] ← 0
6:     for i = 1 → n do
7:         f[i][0] ← max{f[i − 1][0], f[i − 1][1], f[i − 1][2]}
8:         f[i][1] ← max{f[i − 1][0], f[i − 1][2]} + h[1][i]
9:         f[i][2] ← max{f[i − 1][0], f[i − 1][1]} + h[2][i]
10:    end for
11:    return max{f[n][0], f[n][1], f[n][2]}
12: end function
```

# 4 数组排序问题

## 性质观察

在正式解决这个问题之前，需要先观察清楚几个问题的固有性质。

**每种数最多只被操作一次** 如果一个数被操作了两次，那么第一次的操作不会对之后的结果产生任何效果。因为操作的聚集效果和移动效果都具有覆盖的性质，无论第一次操作与否，第二次都可以产生相同的效果。

**操作用于头插和尾插的数，在时间上具有单调性** 既然每种数只会被操作一次，操作可以分为头插和尾插，那么可以把所有数分为头插和尾插两类。以头插举例，在所有对数进行头插的操作中，操作的数必然随着时间单调减小，否则不可能造成最后的有序场面；尾插同理。

**操作用于头插和尾插的数，在值上具有单调性** 这是在说，如果$x$被用于头插，那么所有满足条件$y < x$的$y$都会被应用头插；如果$x$被用于尾插，那么所有大于$x$的$y$都会被用于尾插。由此，如果将所有的整数预先排序去重，那么它们在整个排序的过程中被应用的操作将呈现

$$head, head, \ldots, non, \ldots, tail, \ldots$$

的模样

**要想获得最少的操作次数，就要找出最多的不用操作的数** 由上面的几条性质，可以综合得到最后的一条。

## 算法思路

有了基本的性质，就可以设计找出最多不用操作的数的算法了。设两个关联数组$l, r : X \to I$其中$X$是输入序列的元素集合，$I$是原序列的指标集。其中$l[x]$表示在所有的$x$中，下标最小的那一个下标；$r[x]$则表示所有序列中的$x$中，下标最大的那一个下标。

设$A$是最终求得的，不用操作的数的集合，那么可以看出$A$满足几条性质：

1. $\forall x \in X : \min A \leq x \leq \max A \Rightarrow x \in A$
2. $\forall x, y \in A : x < y \Rightarrow r[x] < l[y]$

其中，第1条性质说明了不操作的数必然是“连续”的一段，这里的连续并不是自然数的连续的含义，而是其中不含有任何其他的$X$的元素的含义，也即在$X$的范围中连续。

我们再引入一个映射$\sigma : X \to [0..|X|-1]$和它的逆映射$\tau : [0..|X|-1] \to X$，其中$\sigma(x)$表示在$X$中小于$x$的元素数量。有了这个映射，上段所描述的$x, y$在$X$的范围中连续，就可以转化为$\sigma(x), \sigma(y)$在$\mathbb{N}$的范围中连续。

有了这些记号，我们就能够方便地描述我们的工作。最终所期望的结果，是找出$[0..|X|-1] =: M$的一个最大的连续子集$A^*$，使得下面的性质成立：

$$\forall x, y \in A^* : r_{\tau(x)} < l_{\tau(y)}$$

现在使用动态规划的思想解决问题，设$f[i]$表示以$i$结尾的$M$的子区间中，满足上述条件最长子区间长度。由此可以得到一个简略的状态转移函数：

$$f[i] = \begin{cases} 1 & , i = 0 \\ 1 & , r_{\tau(i-1)} > l_{\tau(i)} \\ f[i-1]+1 & , r_{\tau(i-1)} < l_{\tau(i)} \end{cases}$$

### 伪代码

参见算法4

### 复杂度分析

可以从伪代码中看出，在使用 Map 作为有序关联数组时，可以使用基于平衡二叉树实现的关联数组，从而做到单次$O(\log n)$的访问和修改复杂度。上面有$n$次访问，因此因为关联数组带来的复杂度有$O(n \log n)$。

综上所述，算法的总体复杂度为$O(n \log n)$

## 5 能耗降低问题

### 算法思路和复杂度分析

在$k$小于所有字符串中1的个数时，我们无需将任何0变成1，因此期望每一次翻转都对答案有正向的贡献；反之，如果$k$太大，那么答案可以直接记为$k - \#1$。所以我们现在只考虑消除1的情形。

考虑到每一个串在消除掉一定量$k$时，其产生的能耗是一定的。我们可以预处理出每一个串在消除$j$个1之后节省的能耗。如果第$i$个串有$s_i$个1，那么可以把这个串拆成$s_i$件物品，其中第$j$个物品表示第$i$个串去掉$j$个1之后，节省的能耗。这样，以去掉的1的个数为体积，$k$为容量，原问题被转化为了若干个类型的分组背包。

由于分组背包有着完整的、已解决的实现，我们这里着重关注如何将每一个字符串拆成若干个类型相同的物品。

**算法 4** 求解此问题

**输入:** $A[1..n]$,

**输出:** 最优方案的值$ans$

```
1: function MinOperation(A[1..n])
2:     f[0..n − 1]
3:     l ← Map⟨int, int⟩
4:     r ← Map⟨int, int⟩
5:     for i = 1 → n do
6:         l[A[i]] ← min {l[A[i]], i}
7:         r[A[i]] ← max {l[A[i]], i}
8:     end for
9:     m ← l.size()
10:    ll[0..m]
11:    rr[0..m]
12:    i ← 0
13:    for x ∈ l do
14:        ll[i] ← x
15:        i ← i + 1
16:    end for
17:    i ← 0
18:    for x ∈ r do
19:        rr[i] ← x
20:        i ← i + 1
21:    end for
22:    f[0] ← 1
23:    ans ← 0
24:    for i = 1 → m − 1 do
25:        if rr[i − 1] < ll[i] then
26:            f[i] ← f[i − 1] + 1
27:        else
28:            f[i] = 1
29:        end if
30:        ans ← max {ans, f[i]}
31:    end for
32:    return ans
33: end function
```

对于第$i$个字符串，可以抽取一个单调的指标序列$A_i$，并且有$s_i = |A_i|$。其中，$A_i$储存了$S_i$字符串的所有1出现位置的下标。现在用$w_{ij}$表示第$i$类的第$j$个物品的价值，含义是字符串$S_i$去掉$j$个1之后节省的能耗，其体积为$j$。根据定义

$$w_{ij} = w_{i0} - \max_{0 \le k \le j} A_i[k + s_i - j] - A_i[k + 1] + 1$$

不难看出，为了计算$w_{ij}$需要花费$O(m^2)$的时间。因此，为了初始化得到所有的物品，需要花费$O(nm^2)$的时间。

最后，我们得到了$n$组物品，其中每组物品最多$m$件，总容量为$k$，由分组背包的时间复杂度可以知道，用于动态规划的时间是$O(nmk)$。

综上，总体的时间复杂度

$$T(n, m, k) = O(nmk + nm^2)$$

## 伪代码

参见算法5

**算法 5** 求解此问题

**输入:** $S[1..n][1..m], k$,

**输出:** 最优方案的值$ans$

```
1:  function MinCost(S[1..n][1..m], k)
2:      w[1..n][1..m]
3:      p[1..n]
4:      for i = 1 → n do
5:          A ← []
6:          for j = 1 → m do
7:              if S[i][j] = '1' then
8:                  A.append(j)
9:              end if
10:         end for
11:         s ← A.size()
12:         p[i] ← s
13:         w[i][0] ← max A − min A
14:         for j = 1 → s do
15:             w[i][j] ← 0
16:             for l = 0 → j do
17:                 w[i][j] ← max{w[i][j], w[i][0] − (A[k + s − j] − A[k + 1] + 1)}
18:             end for
19:         end for
20:     end for
21:     return MultiPack(w, p)
22: end function
```