数据结构与算法期末复习
Homework

尹超

中国科学院大学，北京 100049

Carter Yin

University of Chinese Academy of Sciences, Beijing 100049, China

2024.8 – 2025.1

序言

本文为笔者数据结构与算法的期末复习笔记。

望老师批评指正。

目录

题目 第十二章串

200

> 文本串的长度为 n，模式串的长度为 m，蛮力匹配的时间复杂度为 A. O(m) B. O(n) C. O(mn) D. O(mlogn)

Solution 1. 正确答案是 C。

详细分析:

蛮力匹配算法（*Brute-Force Algorithm*）是最直观、最简单的字符串匹配算法。其工作原理如下:

(1) 将模式串 *'P'* *(*长度为 *'m')* 与文本串 *'T'* *(*长度为 *'n')* 的开头对齐。

(2) 从左到右逐个比较 *'P'* 和 *'T'* 中对应位置的字符。

(3) 如果所有 *'m'* 个字符都匹配成功，则找到了一个匹配。

(4) 如果在比较过程中遇到不匹配的字符，则将模式串 *'P'* 向右移动一个位置，然后回到第 *2* 步，从 *'P'* 的开头重新开始与 *'T'* 的新位置进行比较。

(5) 重复此过程，直到 *'P'* 的末尾超出了 *'T'* 的末尾。

时间复杂度分析:

- **外层循环（对齐次数）**：模式串 *'P'* 需要在文本串 *'T'* 中尝试所有可能的起始对齐位置。第一个对齐位置是 *'T[0]'*，最后一个可能的对齐位置是 *'T[n-m]'*。因此，总共有 *'n - m + 1'* 个可能的对齐位置。这个循环的次数是 $O(n)$。
- **内层循环（比较次数）**：对于每一个对齐位置，在最坏的情况下，都需要将模式串 *'P'* 的所有 *'m'* 个字符与文本串 *'T'* 的对应部分进行比较。例如，当文本串是*"aaaaaaaaab"* 而模式串是*"aaab"* 时，每次对齐都需要比较多次才能发现不匹配。这个循环的次数是 $O(m)$。

总时间复杂度: 由于是嵌套循环，总的时间复杂度是外层循环次数乘以内层循环次数。$T(n,m) = O((n-m+1)\times m) = O(n\times m) = O(mn)$。

201

> 下面哪位不是 KMP 算法的发明者: A. Knuth B. Morris C. Pratt D. Prim

Solution 2. 正确答案是 D。

详细分析:

- ***KMP* 算法**：这是一种高效的字符串匹配算法，其名称*"KMP"* 是三位发明者姓氏的首字母缩写。
- **发明者:**
  - ***K**nuth (Donald Knuth)*
  - ***M**orris (James H. Morris)*
  - ***P**ratt (Vaughan Pratt)*

  因此，选项 *A*、*B*、*C* 都是 *KMP* 算法的发明者。
- ***Prim (Robert C. Prim)***: *Prim* 是一位著名的计算机科学家，但他以发明用于寻找图的**最小生成树**（***Minimum Spanning Tree***）的 ***Prim* 算法**而闻名，与字符串匹配无关。

202

> KMP 算法的过程中，若某次比对在模式串 P 的第 j 个位置 P[j] 处失败，则将对齐位置换为：A. prev[j] B. next[j] C. prev[j] + 1 D. next[j] + 1

Solution 3. 正确答案是 *B*。

详细分析：

KMP 算法的核心在于利用一个预先计算好的 ‘*next*‘数组（也称为“失效函数”或“部分匹配表”）来避免不必要的回溯。

(1) ‘*next*‘数组的含义： ‘*next[j]*‘ 的值代表了模式串 ‘*P*‘ 的子串 ‘*P[0...j-1]*‘ 中，最长的相等的前缀和后缀的长度。例如，如果 ‘*P = ”ababa”*‘，那么 ‘*next[5]*‘ 的值为 *3*，因为 ‘*P[0...4]*‘（即”*ababa*”）的最长相等前后缀是”*aba*”，长度为 *3*。

(2) 匹配失败时的操作：

- 假设在匹配过程中，文本串指针为 ‘*i*‘，模式串指针为 ‘*j*‘。
- 当 ‘*P[j]*‘ 与文本串中的对应字符不匹配时，说明 ‘*P*‘ 的前 ‘*j*‘ 个字符 ‘*P[0...j-1]*‘ 已经与文本串中的某一部分成功匹配。
- 此时，我们不需要像蛮力算法那样将模式串仅仅右移一位。相反，我们查询 ‘*next[j]*‘。
- ‘*next[j]*‘ 的值告诉我们，在已经匹配的 ‘*P[0...j-1]*‘ 中，其长度为 ‘*next[j]*‘ 的前缀 ‘*P[0...next[j]-1]*‘ 与其后缀是相同的。
- 因为这个后缀已经与文本串的对应部分匹配了，所以我们无需再次比较。我们可以直接将模式串滑动到这个前缀之后的位置，继续进行比较。
- 这个新的比较位置就是 ‘*next[j]*‘。因此，我们将模式串的指针 ‘*j*‘ 更新为 ‘*next[j]*‘，而文本串的指针 ‘*i*‘ 保持不变，然后继续比较 ‘*P[next[j]]*‘ 和文本串的当前字符。

结论： 当在 ‘*P[j]*‘ 处发生不匹配时，*KMP* 算法通过 ‘*j = next[j]*‘ 来更新模式串的对齐位置，实现高效的“滑动”，从而避免了文本串指针的回溯。

203

> 对于长度为 n 的文本串和长度为 m 的模式串, KMP 算法的时间复杂度为: A. O(n^2) B. O(mn) C. O(mlogn) D. O(m+n)

Solution 4. 正确答案是 *D*。

详细分析：

KMP 算法的执行过程可以分为两个独立的部分，其总时间复杂度是这两部分之和。

(1) 预处理阶段：构建 ‘*next*‘数组

- 这个阶段只对长度为 ‘*m*‘ 的模式串进行操作，以计算出 ‘*next*‘数组（或称部分匹配表）。
- 构建 ‘*next*‘数组的过程只需要对模式串进行一次扫描。
- 因此，这个阶段的时间复杂度与模式串的长度 ‘*m*‘ 成正比，即 $O(m)$。

(2) 匹配阶段：在文本串中搜索

- 在这个阶段，算法使用构建好的 ‘*next*‘数组在长度为 ‘*n*‘ 的文本串中进行匹配。
- *KMP* 算法最关键的特性是，文本串的指针 ‘*i*‘ **永远不会回溯**（即 ‘*i*‘ 只会增加，不会减少）。

- 虽然模式串的指针 'j' 可能会因为不匹配而通过 'j = next[j]' 回溯，但 'j' 的回溯次数总和不会超过 'i' 的前进次数。
- 因此，在整个匹配过程中，对文本串的扫描也是线性的。
- 这个阶段的时间复杂度与文本串的长度 'n' 成正比，即 $O(n)$。

总时间复杂度: 将两个阶段的复杂度相加: $T(n,m) = T(\text{预处理}) + T(\text{匹配}) = O(m) + O(n) = O(m+n)$。这使得 *KMP* 算法比蛮力匹配的 $O(mn)$ 复杂度要高效得多。

204

> 给定一个进行串匹配的算法，如何衡量它的效率?
> A. 随机生成大量的文本串 T 和模式串 P 作为输入，通过实验的方法进行测量
> B. 认为所有不同文本串 T 和模式串 P 出现的概率是相等的，依此计算时间复杂度的期望
> C. 对于成功匹配和失败匹配两种情况分别讨论其时间复杂度
> D. 选取固定的文本串 T，随机选取模式串 P，计算时间复杂度的期望

Solution 5. 正确答案是 *C*。

详细分析:

衡量一个算法的效率，尤其是像字符串匹配这样的算法，通常需要进行全面的理论分析，而不仅仅是实验测试或基于特定假设的期望计算。

- ***A.* 实验测量:** 实验方法可以提供算法在特定数据集和机器上的实际性能数据，但结果可能不具有普适性，并且很难覆盖所有情况，特别是难以触发"最坏情况"。它是一种验证手段，但不是主要的理论衡量方法。
- ***B.* 概率均等假设:** 假设所有串出现的概率相等，这在现实世界中通常是不成立的（例如，在英文文本中，字母*'e'* 的出现频率远高于*'z'*）。这种平均情况分析有其价值，但它基于一个很强的、可能不切实际的假设。
- ***C.* 分情况讨论:** 这是进行算法分析最严谨和标准的方法。通过分别讨论成功匹配和失败匹配（以及匹配/失败发生的位置），我们可以分析出算法的:
  - **最坏情况时间复杂度（*Worst-case*）:** 算法在任何输入下运行时间的一个上界。这是衡量算法效率最重要的指标。
  - **最好情况时间复杂度（*Best-case*）:** 算法运行时间的一个下界。
  - **平均情况时间复杂度（*Average-case*）:** 在所有可能输入下的期望运行时间。

  例如，蛮力算法在文本串为*"aaaa...a"*、模式串为*"a...ab"* 时达到最坏情况（一种失败匹配）。而在文本串和模式串第一个字符就不匹配时达到最好情况。分别讨论这些情况是得出完整效率评价的基础。
- ***D.* 固定文本串:** 这种方法过于片面，得到的结果严重依赖于所选的固定文本串，无法推广到一般情况。

结论: 最全面和理论上最可靠的衡量方法是分别讨论各种情况（特别是成功和失败），从而确定算法的最坏、最好和平均时间复杂度。

205

以下是蛮力串匹配的代码:

```
int match(const char * P, const char * T)
{
    int n =strlen(T);
    int m = strlen(P);
    int i = 0;
    int j = 0;
    while (j < m && i < n) // Note: Corrected from the likely typo 'j < n'
    {
        if (T[i] == P[j])
        {
            i++;
            j++;
        }else
        {
            i = i - j + 1; // Note: Corrected from the typo 'i -= |j - 1|'
            j = 0;
        }
    }
    return i-j;
}
```

当匹配成功/失败时的返回值分别为:

A. P 在 T 中首次出现的位置 / -1

B. P 在 T 中首次出现的位置 / 一个大于 n-m 的数

C. P 在 T 中最后一次出现的位置 / 一个大于 n-m 的数

D. P 在 T 中最后一次出现的位置 / -1

Solution 6. 正确答案是 B。

详细分析:

该代码是一个有明显错误的蛮力串匹配算法实现。为了分析其意图，我们必须将其修正为标准的蛮力算法逻辑来分析其返回值。

***(1)* 匹配成功时:**

- 算法从左到右扫描文本串 T，因此它找到的任何匹配都是**首次出现**的匹配。
- 当匹配成功时，循环会因为 'j' 达到 'm' 而终止。
- 此时，'i' 的值是 '$s+m$'，其中 's' 是匹配开始的位置。'j' 的值是 'm'。
- 返回值 'i - j' 就等于 '$(s+m)$ - $m = s$'。
- 因此，成功时返回的是 P 在 T 中首次出现的位置（起始索引）。

***(2)* 匹配失败时:**

- 当匹配失败时，循环会因为 'i' 遍历完所有可能的起始位置而终止。

- 最后一个可能的起始位置是 ‘$n-m$‘。当从这个位置开始的匹配也失败后，‘i‘ 会继续增加，直到 ‘i‘ 的值达到 ‘n‘，导致循环条件 ‘$i < n$‘ 不满足而终止。
- 此时，‘i‘ 的值近似为 ‘n‘，而 ‘j‘ 的值是某个小于 ‘m‘ 的数（因为匹配未完成）。
- 返回值 ‘$i-j$‘ 将会是 ‘$n-j$‘。
- 因为 ‘$j < m$‘，所以 ‘$-j > -m$‘，因此 ‘$n-j > n-m$‘。
- 所以，在大多数失败情况下，返回值是一个大于 ‘$n-m$‘ 的数。这个值超出了所有可能的合法起始索引（‘0‘ 到 ‘n-m‘），因此可以作为匹配失败的标志。

结论：

- 成功时返回首次出现的位置。
- 失败时返回一个大于 ‘n-m‘ 的数。

选项 *B* 最符合这个分析。

206

> 在文本串 YHNMQWERTYFLNYCQWERTYFGIOERNSJTYAFFA 中用 KMP 算法查找模式串 QWERTYFLNYCQWERTYO 已经对齐了第一个 QWERTYFLNYC 下一步的对齐位置是

Solution 7. 下一步的对齐位置是将模式串的开头 ‘Q‘ 与文本串中第 *15* 个位置的 ‘Q‘ 对齐。

详细分析：

KMP 算法的核心是利用 ‘next‘ 数组在发生不匹配时，计算出模式串应该“滑动”多远，从而避免文本串指针的回溯。

***(1)* 确定不匹配的位置：**

- 文本串 *T*: ‘...*Q W E R T Y F L N Y C Q W E R T Y F G*...‘
- 模式串 *P*: ‘*Q W E R T Y F L N Y C Q W E R T Y O*‘
- 初始对齐在 *T* 的第 *4* 个位置（索引从 *0* 开始）。
- 我们逐一比较，发现 ‘*T[4...20]*‘ 与 ‘*P[0...16]*‘ 完全匹配，均为 ‘*QWERTYFLNYCQWERTY*‘。
- 不匹配发生在下一个位置：文本串 ‘*T[21]*‘ 是 ‘*F*‘，而模式串 ‘*P[17]*‘ 是 ‘*O*‘。
- 因此，不匹配发生在模式串的第 ‘*j=17*‘ 个位置。

***(2)* 计算 ‘*next*‘ 数组的值：**

- 当在 ‘*P[j]*‘ 处不匹配时，我们需要查找 ‘*next[j]*‘ 的值。这里 ‘*j=17*‘。
- ‘*next[j]*‘ 的值是模式串子串 ‘*P[0...j-1]*‘ 中，最长的相等的“前缀”和“后缀”的长度。
- 我们需要分析的子串是 ‘*P[0...16]*‘，即 ‘*QWERTYFLNYCQWERTY*‘。
- 观察这个子串：
  - 它的前缀包括 ‘*Q*‘, ‘*QW*‘, ‘*QWE*‘, ‘*QWER*‘, ‘*QWERT*‘, ***‘QWERTY‘***, ...
  - 它的后缀包括 ‘*Y*‘, ‘*TY*‘, ‘*RTY*‘, ‘*ERTY*‘, ‘*WERTY*‘, ***‘QWERTY‘***, ...
- 最长的相等的“前缀”和“后缀”是 ‘*QWERTY*‘，其长度为 *6*。
- 因此，‘*next[17] = 6*‘。

***(3)* 确定下一步对齐位置：**

- *KMP* 算法的规则是，当在 ‘*j*‘ 处不匹配时，将模式串的指针移动到 ‘*next[j]*‘。
- 所以，模式串的指针 ‘*j*‘ 从 *17* 变为 *6*。
- 这相当于将模式串向右滑动，使得模式串中长度为 ‘*next[j]*‘ 的前缀 ‘*P[0...5]*‘（即 ‘*QWERTY*‘）与刚刚匹配上的文本串部分的后缀对齐。

- 刚刚匹配上的文本串部分是 *'T[4...20]'*，其长度为 6 的后缀是 *'T[15...20]'*（即 *'QWERTY'*）。
- 因此，新的对齐位置是将模式串的 *'P[0]'* 对齐到文本串的 *'T[15]'*。

结论: 下一步是将模式串的开头 *'Q'* 与文本串中的第二个 *'Q'*（位于索引 15）对齐，然后从 *'P[6]'* 和 *'T[21]'* 开始继续比较。

207

> KMP 算法的查询表为 next[]，模式串为 P，若 P[0,j) 与文本串匹配，而在 P[j] 处失配，则:
> A. P[0, next[j]) = P[j - next[j], j)
> B. P[0, next[j]+1) = P[j - next[j],j+1)
> C. P[0, next[j] - 1) = P[j - next[j],j)
> D. P[0, next[j]) = P[j - next[j] - 1,j)

Solution 8. 正确答案是 *A*。

详细分析:

这道题考查的是 *KMP* 算法中 *'next'* 数组（或称"失效函数"）的精确定义。

(1) **背景:** 当匹配进行到模式串 *'P'* 的第 *'j'* 个位置（索引为 *'j'*）时发生不匹配，这意味着 *'P'* 的前 *'j'* 个字符，即子串 *'P[0...j-1]'*（在题目表示法中为 *'P[0, j)'*），已经与文本串的某一部分成功匹配。

(2) ***'next[j]'*** **的定义:** *'next[j]'* 的值被定义为：在已经匹配的子串 *'P[0...j-1]'* 中，其**最长的相等的"真前缀"和"真后缀"的长度**。

- "真前缀"是指不包括整个字符串本身的前缀。
- "真后缀"是指不包括整个字符串本身的后缀。

(3) **将定义转化为数组索引:**

- 设 *'k = next[j]'*。
- 根据定义，*'P[0...j-1]'* 的一个长度为 *'k'* 的前缀，与它的一个长度为 *'k'* 的后缀是相等的。
- 长度为 *'k'* 的前缀是 *'P[0...k-1]'*。用区间表示法就是 *'P[0, k)'*。
- 长度为 *'k'* 的后缀是 *'P[(j-1) - k + 1 ... j-1]'*，即 *'P[j-k ... j-1]'*。用区间表示法就是 *'P[j-k, j)'*。

(4) **结论:** 因此，*'next[j]'* 的定义直接导出了等式：*'P[0, k) = P[j-k, j)'* 将 *'k'* 替换回 *'next[j]'*，我们得到：*'P[0, next[j]) = P[j - next[j], j)'*

这个等式是 *KMP* 算法能够实现高效"滑动"的数学基础。当在 *'P[j]'* 处失配时，算法知道文本串中刚刚匹配过的部分，其后缀等于模式串 *'P'* 的一个前缀（*'P[0, next[j])'*），因此可以直接将模式串滑动到这个位置继续比较，而无需从头开始。

208

> 令 A = {t | P[0, t) = P[j - t, j)} ，即 A 是所有使得 P[0,j) 的前缀与后缀相等的长度 t，如何计算 next[j]?
> A. next[j] = min A
> B. next[j] = max A
> C. next[j] = |A| (A 中的元素个数)
> D. next[j] = max A - |A|

Solution 9. 正确答案是 *B*。

详细分析:

(1) **理解集合 *A* 的定义:** 集合 '*A*' 被定义为 '*t* | *P[0, t) = P[j - t, j)*'。

- '*P[0, j)*' 表示模式串 '*P*' 从索引 *0* 开始，长度为 '*j*' 的子串，即 '*P[0...j-1]*'。
- '*P[0, t)*' 是 '*P[0...j-1]*' 的一个长度为 '*t*' 的前缀。
- '*P[j - t, j)*' 是 '*P[0...j-1]*' 的一个长度为 '*t*' 的后缀。

因此，集合 '*A*' 包含了所有使得 '*P[0...j-1]*' 的前缀与后缀相等的长度 '*t*'。

(2) **回顾 '*next[j]*' 的定义:** *KMP* 算法中的 '*next[j]*' 被精确地定义为：在子串 '*P[0...j-1]*' 中，其**最长的**相等的"真前缀"和"真后缀"的长度。("真"表示不等于字符串本身，这个条件在集合 *A* 的定义中是隐含的，因为 '*t*' 必须小于 '*j*')。

(3) **建立联系:**

- 集合 '*A*' 包含了所有满足条件的长度。
- '*next[j]*' 需要的是这些长度中的**最大值**。

因此，'*next[j]*' 就是集合 '*A*' 中的最大元素。

(4) **结论:** '*next[j] = max A*'。

示例: 假设 '*P = "ababa"*'，我们要计算 '*next[5]*'。

- 此时 '*j=5*'，子串是 '*P[0, 5)*' 即 '*"ababa"*'。
- 我们寻找其相等的前后缀:
  - 长度 '*t=1*': 前缀*"a"* == 后缀*"a"*。所以 '*1*' 在集合 '*A*' 中。
  - 长度 '*t=2*': 前缀*"ab"* != 后缀*"ba"*。
  - 长度 '*t=3*': 前缀*"aba"* == 后缀*"aba"*。所以 '*3*' 在集合 '*A*' 中。
  - 长度 '*t=4*': 前缀*"abab"* != 后缀*"baba"*。
- 集合 '*A = 1, 3*'。
- '*next[5] = max A = max(1, 3) = 3*'。

209

> 在通过 next[j] 计算 next[j+1] 的递推过程中 next[j+1] == next[j] + 1 当且仅当:
> A. j = 0
> B. P[j] = P[next[j] - 1]
> C. T[j] = P[j]
> D. P[j] = P[next[j]]

Solution 10. 正确答案是 *D*。

详细分析:

这道题考查的是 *KMP* 算法中 '*next*'数组的递推构造过程。

(1) **递推基础:** 假设我们已经计算出了 '*next[j]*'，现在要计算 '*next[j+1]*'。

- '*next[j]*' 的值，我们记为 '*k*'，代表了子串 '*P[0...j-1]*' 的最长相等前后缀的长度。
- 这意味着 '*P[0...k-1] == P[j-k...j-1]*'。

(2) **寻找 '*next[j+1]*':** '*next[j+1]*' 是子串 '*P[0...j]*' 的最长相等前后缀的长度。我们希望在 '*next[j]*' 的基础上进行扩展。

- 我们已经有了一个长度为 '*k*' 的匹配：前缀 '*P[0...k-1]*' 和后缀 '*P[j-k...j-1]*'。

- 为了将这个匹配的长度扩展到 'k+1'，我们需要比较这两个匹配部分的下一个字符。
- 前缀 'P[0...k-1]' 的下一个字符是 'P[k]'。
- 后缀 'P[j-k...j-1]' 的下一个字符是 'P[j]'。

(3) **得出条件:**

- 如果 'P[k] == P[j]'，那么我们就可以将这个匹配长度加一。
- 这意味着 'P[0...k]' 等于 'P[j-k...j]'。
- 此时，'P[0...j]' 的最长相等前后缀长度就是 'k+1'。
- 将 'k' 替换回 'next[j]'，我们得到：如果 'P[next[j]] == P[j]'，那么 'next[j+1] = next[j] + 1'。

结论: 'next[j+1]' 能在 'next[j]' 的基础上加 1，当且仅当 'P[j]' 这个新加入的字符，恰好等于 'next[j]' 所指向的前缀的下一个字符，即 'P[next[j]]'。

其他选项分析:

- *A. j = 0:* 这是初始条件，不是递推关系。
- *B. P[j] = P[next[j] - 1]:* 比较的是前缀的最后一个字符，而不是下一个字符，逻辑错误。
- *C. T[j] = P[j]:* 'next'数组的计算是预处理阶段，只与模式串 P 有关，与文本串 T 无关。

210

> 对于模式串 CHINCHILLA，计算其 next[]
> A. -1 0 0 0 0 1 2 3 0 0
> B. -1 2 5 4 5 6 2 3 9 1
> C. 1 0 0 2 6 7 2 5 7 1
> D. 0 0 1 -1 0 0 0 3 2 0

Solution 11. 正确答案是 *A*。

详细分析:

'next[j]' 的值是模式串 'P' 的子串 'P[0...j-1]' 中，最长的相等的"真前缀"和"真后缀"的长度。我们按照这个定义逐个计算。（约定 'next[0] = -1'）

- ***j=0:*** 'next[0] = -1' (按约定)
- ***j=1:*** P[0] = "C"。没有真前缀/后缀。'next[1] = 0'。
- ***j=2:*** P[0...1] = "CH"。前缀"C"，后缀"H"。不相等。'next[2] = 0'。
- ***j=3:*** P[0...2] = "CHI"。前缀"C", "CH"，后缀"I", "HI"。不相等。'next[3] = 0'。
- ***j=4:*** P[0...3] = "CHIN"。前缀"C", "CH", "CHI"，后缀"N", "IN", "HIN"。不相等。'next[4] = 0'。
- ***j=5:*** P[0...4] = "CHINC"。前缀"C", ...，后缀"C", ...。最长相等的是"C"，长度为 1。'next[5] = 1'。
- ***j=6:*** P[0...5] = "CHINCH"。前缀"C", "CH", ...，后缀"H", "CH", ...。最长相等的是"CH"，长度为 2。'next[6] = 2'。
- ***j=7:*** P[0...6] = "CHINCHI"。前缀"C", "CH", "CHI", ...，后缀"I", "HI", "CHI", ...。最长相等的是"CHI"，长度为 3。'next[7] = 3'。
- ***j=8:*** P[0...7] = "CHINCHIL"。没有相等的前后缀。'next[8] = 0'。
- ***j=9:*** P[0...8] = "CHINCHILL"。没有相等的前后缀。'next[9] = 0'。

将结果组合起来，得到 'next' 数组为：'-1, 0, 0, 0, 0, 1, 2, 3, 0, 0'。