多维数组和字符串

Tan Yiqing

2025 年 10 月 31 日

1 多维数组

多维数组是由多个一维数组组成的数组。最常见的就是二维数组，可以看作是一个表格，具有行和列。上述的矩阵可以看做一个线性表，每个元素是矩阵的一行，把每行的元素用一个线性表装起来。同理可以推广到多维数组。

1.1 数组的定义

数组是由一组类型相同的数据元素构成的有序集合，每个数据元素称为一个数组元素，受 n 个线性关系的约束。我们使用 n 个序号来描述每个元素的位置。

1.2 数组的基本操作

数组的基本操作包括：

- 存取元素：给定下标，读出数组中的元素。
- 修改元素：给定下标，修改数组中的元素。
- 上述称寻址类操作，数组使用顺序结构储存。

1.3 数组的存储结构

n 维数组内存还是一维数组，二维数组可按先行后列来储存。

2 特殊矩阵

一般来说一个 $m*n$ 的矩阵需要 $m*n$ 的空间来存储，但是有些矩阵有特殊的性质，可以节省空间。基本思路是相同元素只记一个，0 可以不记。

2.1 对称矩阵

2.2 上三角矩阵和下三角矩阵

2.3 三对角矩阵：川形矩阵

2.4 稀疏矩阵：很多 0 的矩阵

稀疏矩阵是指矩阵中大部分元素为 0 的矩阵。稀疏矩阵可以用三元组顺序表来存储。

- 三元组顺序表：用一个三元组（行号，列号，值）来表示矩阵中的非零元素。
- 例如：

$$\begin{bmatrix} 0 & 0 & 3 & 0 \\ 0 & 0 & 5 & 7 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 \end{bmatrix}$$

可以表示为：

$$\Big[(1,3,3)(2,3,5)(2,4,7)(4,2,2)(4,3,6)\Big]$$

也可以用十字链表来实现。

- 十字链表：每个非零元素有两个指针，一个指向同一行的下一个非零元素，一个指向同一列的下一个非零元素。
- 例如上面的矩阵，可以表示为：

$$\begin{bmatrix} (1,3,3) \to (2,3,5) \to (4,3,6) \\ (2,4,7) \\ (4,2,2) \end{bmatrix}$$

三元组表使用的多，而且简单；十字链表写起来复杂，但是矩阵乘法操作方便。

3 字符串

字符串是由零个或多个字符组成的有限序列。字符个数成为字串长度，空串记为""，长度为 0。非空字串记为"$s_0 s_1 ... s_{n-1}$"，长度为 n，其中双引号为字符串的界定符。

3.1 字符串的操作

3.1.1 字符串比较

字符串比较是指比较两个字符串的大小关系。一般来说，字符串的比较是按字典序进行的。　　连续的子序列均称为子串。子串的位置为其首字符于原字串的位置。

- 字典序：按字符的 ASCII 码值逐个比较，直到遇到不同的字符为止。若一个字符串是另一个字符串的前缀，则较短的字符串较小。
- 例如：”apple” < ”banana”，因为’a’ < ’b’；”apple” < ”apples”，因为前者是后者的前缀。
- 算法：从第一个字符开始，逐个比较两个字符串的字符，直到遇到不同的字符或其中一个字符串结束为止。

3.1.2 字符串连接

字符串连接是指将两个字符串合并成一个新的字符串。连接符通常用”+” 表示。

- 例如：”hello” + ”world” = ”helloworld”
- 算法：创建一个新的字符串，长度为两个字符串长度之和，然后将第一个字符串的字符复制到新字符串的前半部分，再将第二个字符串的字符复制到新字符串的后半部分。

3.1.3 子串

子串是指字符串中的一个连续的字符序列。子串可以通过指定起始位置和长度来获取。

- 例如：字符串”hello” 的子串有”he”、”ell”、”lo” 等。
- 算法：给定起始位置和长度，从原字符串中提取相应的字符，形成新的子串。

3.2 字符串的存储结构

c++ 用字符串类 string 来存储字符串，无需特殊处理。

4 模式匹配:pattern matching

模式匹配是指在一个字符串中查找另一个字符串的位置。模式匹配广泛应用于文本编辑器、搜索引擎等领域。这里只介绍最简单的模式匹配算法。

给定主串 S 和模式串 T，在 S 中寻找 T 的位置成为模式匹配问题。若匹配成功，返回 T 在 S 中的位置；若失败，返回-1。

注意: 算法执行一次的时间不容忽视，因此一个好的算法的长期效果不容忽视。

4.1 BF 算法：Brute Force

BF 算法是最简单的模式匹配算法。对 S 的每个位置 i，与 T 进行比较。当不对应时 T 回溯到第一个字符，继续与 S 的位置 i+1 重新开始比较。

它的正确率有绝对保证，但是时间复杂度为 O(m*n) 很高，其中 m 为 S 的长度，n 为 T 的长度。

- 算法步骤：
  1. 初始化两个指针 i 和 j，分别指向 S 和 T 的起始位置。
  2. 比较 S[i] 和 T[j]，如果相等，则 i++，j++；如果不相等，则 i = i - j + 1，j = 0。
  3. 重复步骤 2，直到 j 等于 T 的长度（匹配成功）或 i 等于 S 的长度（匹配失败）。
  4. 如果匹配成功，返回 i - j；否则返回-1。

4.2 KMP 算法：Knuth-Morris-Pratt

KMP 算法是改进的模式匹配算法。它利用已经匹配过的信息，避免了不必要的比较，从而提高了效率。

让我们先思考一个情况，设模式串 abcac，假设前四位已经匹配成功了，如果 kmp 算法有改进，那么主串的第五位一定是 a，这就说明了模式串的前缀和后缀有相同的部分。

假设我们目前在配对 S 的第 i 个位置和 T 的第 j 个位置。错配后，如果应该把模式滑到 T 的第 k 个位置，那其实代表了：

$$T_0T_1\dots T_{j-1} = S_{i-j}S_{i-j+1}\dots S_{i-1}$$

$$T_0T_1\dots T_{k-1} = S_{i-k}S_{i-k+1}\dots S_{i-1}$$

由此我们可以推断得到：

$$T_0T_1\dots T_{k-1} = T_{j-k}T_{j-k+1}\dots T_{j-1}$$

即 T 中有一部分是重复的，具体来说，是 0 到第 k 个字符和第 j-k 个到第 j-1 个字符相同。

对于每一个 j，（如果有多个 k）我们都可以找到一个最大的 k，使得上述等式成立。

KMP 算法的关键就是要知道如果在模式的第 j 个位置失配，我们应该改变成与模式的第几个位置再进行匹配（也就是到底应该回溯多少），这个信息存储在一个叫做 next 数组的数组中。

$$\text{next}[j] := \begin{cases} -1 & \text{当 } j = 0 \\ \max\left(\max\{k \mid 1 \le k < j, T_0T_1\dots T_{k-1} = T_{j-k}\dots T_{j-1}\}, 0\right) & \text{当 } j \ne 0 \end{cases}$$

4.3 next 数组的计算

next 数组的计算是 KMP 算法的关键步骤。next 数组用于记录模式串中每个位置的最长相等前后缀的长度。

1. 初始化：$next[0] = -1$，$j = -1$。
2. 对 $i = 1, 2, \dots, n-1$ 依次执行：
   (a) 当 $j \ge 0$ 且 $T[i] \ne T[j+1]$ 时，令 $j \leftarrow next[j]$（沿前缀函数回退）。
   (b) 若此时 $T[i] = T[j+1]$，则 $j \leftarrow j+1$（成功扩展一位）。
   (c) 置 $next[i] = j$。

伪代码

```
Input: pattern T[0..n-1]
Output: next[0..n-1]    // 长度版

next[0] <- 0
j <- 0                          // 已匹配前缀长度
for i from 1 to n-1:
    while j > 0 and T[i] != T[j]:
        j <- next[j-1]
    if T[i] == T[j]:
        j <- j + 1
    next[i] <- j
```

示例：模式串 "ABCABD" 的跳转表构造

1. 位置 0：子串 "A"
   前缀：[ ]，后缀：[ ]；最长公共前后缀：0 ⇒ 跳转值 = 0
2. 位置 1：子串 "AB"
   前缀：["A"]，后缀：["B"]；无相同 ⇒ 跳转值 = 0
3. 位置 2：子串 "ABC"
   前缀：["A","AB"]，后缀：["BC","C"]；无相同 ⇒ 跳转值 = 0
4. 位置 3：子串 "ABCA"
   前缀：["A","AB","ABC"]，后缀：["BCA","CA","A"]；公共部分："A" ⇒ 跳转值 = 1
5. 位置 4：子串 "ABCAB"
   前缀：["A","AB","ABC","ABCA"]，后缀：["BCAB","CAB","AB","B"]；公共部分："AB" ⇒ 跳转值 = 2
6. 位置 5：子串 "ABCABD"
   前缀：["A","AB","ABC","ABCA","ABCAB"]，后缀：["BCABD","CABD","ABD","BD","D"]；无相同 ⇒ 跳转值 = 0

得到跳转表：[−1, 0, 0, 1, 2, 0]。

// ...existing code...

4.4 kmp 算法的执行

本节按“长度版”next（lps）：$next[i]$ 为 $T[0..i]$ 的最长相等真前后缀长度，$next[0] = 0$。

执行流程

1. 设主串 S 长度为 m，模式串 T 长度为 n，已知 $next[0..n-1]$。
2. 置 $i \leftarrow 0$, $j \leftarrow 0$。
3. 当 $i < m$ 时循环：
   (a) 若 $j > 0$ 且 $S[i] \neq T[j]$，则 $j \leftarrow next[j-1]$（回退到可重叠的最长前缀）。
   (b) 若 $S[i] = T[j]$，则 $i \leftarrow i+1$, $j \leftarrow j+1$。
   (c) 若 $j = n$，则在 $i-n$ 处匹配成功，记录 $i-n$；随后 $j \leftarrow next[n-1]$ 继续。
   (d) 否则（无法回退且不相等时），仅 $i \leftarrow i+1$。

伪代码（长度版）

```
Input: text S[0..m-1], pattern T[0..n-1], next[0..n-1]
Output: 所有匹配起始下标

i <- 0; j <- 0
while i < m:
    while j > 0 and S[i] != T[j]:
        j <- next[j-1]
    if S[i] == T[j]:
        i <- i + 1
        j <- j + 1
    else:
```

```
        i <- i + 1
    if j == n:
        report (i - n)
        j <- next[n-1]
```

示例：在 "ABCABCABD" 中找 "ABCABD" 模式的跳转表（长度版）：$next = [-1, 0, 0, 1, 2, 0]$。

S: A B C A B C A B D　　T: A B C A B D

- 从 $i = 0, j = 0$ 开始，匹配到 $i = 5, j = 5$ 时失配：$S[5] = \texttt{C} \neq T[5] = \texttt{D}$。
- 回退：$j \leftarrow next[4] = 2$（i 不变，仍为 5）。
- 继续比较：$S[5] = \texttt{C} = T[2]$，于是 $i = 6, j = 3$；接着 $S[6] = \texttt{A} = T[3]$（$i = 7, j = 4$），$S[7] = \texttt{B} = T[4]$（$i = 8, j = 5$），$S[8] = \texttt{D} = T[5]$（$i = 9, j = 6$）。
- 此时 $j = n = 6$，匹配成功，起始下标为 $i - n = 9 - 6 = 3$（0 基）。

说明：若以“当前比较位置”为 $i' = i - 1$ 的记法，则返回 $i' - n + 1 = 8 - 6 + 1 = 3$，一致。