多维数组和字符串

矣振扬

September 22, 2021

本章主要内容

- 多维数组
- 特殊矩阵
- 字串
- 模式匹配

线性表的特例

- 固定大小的二维数组就是数学中常见的矩阵。比如 C++ 里的声明 `int A[5][5];` 就代表一个 5×5 整数系数矩阵。
- 上述的可以看为一个线性表，其中每个元素就是矩阵的一行，可把每行的元素用一个线性表装起来。
- 这样我们可以看到二维数组就是每个数据元素为线性表的线性表。
- 同理，我们可以推广至多维数组。
- 但线性表视点实际还是有那么一点不同，因为每个线性表的长度可以不同，而做成比较奇怪的结构。

n 维数组

Definition

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。

与其他结构不同，数组具有固定格式和数量的数据集合；而且插入及删除毫无意义。

数组的基本操作

- 存取：给定下标，读出对应元素的数值。
- 修改：给定下标，修改对应元素的数值。
- 上述统称寻址类操作。
- 基于以上部分：数组极度适合使用顺序存储结构。（使用空间固定……）

数组的存储结构：二维数组

- 实际上 n 维数组内存还是一位数组。
- 二维数组可按：先行后列或先列后行来存储。
- 多维数组也是一样。

一般矩阵与特殊矩阵

- 一般来说一个 $m \times n$ 的矩阵，我们就是要 mn 个空间。
- 但有些矩阵里元素分布有一定规律，可能可以省一点空间。
- 基本思路是确定相同的元素只记一个，零的话更可能完全不记。
- 例子：对称矩阵、上（下）三角矩阵、三对角矩阵……
- 稀疏矩阵：很多零元素的矩阵。

稀疏矩阵：三元组表

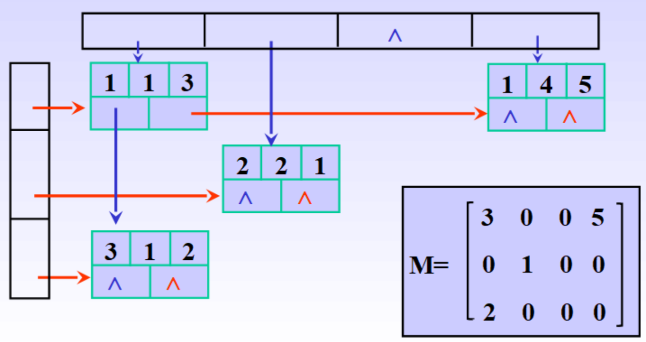
- 对每个非零元素，使用一个三元组存储该元素。
- 三元组具体为：行、列、数据。

$$A=\begin{bmatrix} 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

三元组表 = ((0,0,15), (1,1,11), (2,3,6), (4,0,9))

稀疏矩阵：十字链表

- 对每个非零元素，给一个结点存数据及其行列后继的指针。
- 另外每行、每列首位为空结点。



字串的逻辑结构

Definition

字串为零或多个字符组成的有限序列。

字符个数为字串长度。空串写为“”。

非空串记为 $S = \text{“}s_0 s_1 \ldots s_{n-1}\text{”}$，其中双引号为定界符。

对上述的字串，其中连续的子序列均称为子串。我们定义子串的位置为其首字符于原字串的位置。

常用字符集：ASCII、扩展的 ASCII、Unicode 等……

字串比较

- 给定两个串 $X =$ “$x_0x_1 \ldots x_{n-1}$” 和 $Y =$ “$y_0y_1 \ldots y_{m-1}$”，比较为：
- 1. 当 $n = m$，而且对所有 i，均有 $x_i = y_i$，则 $X = Y$。
- 2. 当 $X < Y$：若 $n < m$，且对 $0 \leq i < n$，$x_i = y_i$；
- 存在 $k \leq \min(m, n)$，使得对 $0 \leq i < k$，有 $x_i = y_i$ 且 $x_k < y_k$。
- 简单一点说就基本是字典的排序。

字串的存储结构

- 基本存储都是字符的数组。问题一般为如何标记字串的结束（长度、或特殊标记）
- C 的话使用特殊字符 \0。
- C++ 的话则是字串类 `string`。实际上就不用我们特殊处理。

模式匹配问题

- 给定主串 S，模式串 T，在 S 中寻找 T 的位置称为模式匹配。
- 若匹配成功，返回 T 在 S 中的位置；若失败，返回 -1。
- 算法一次执行时间不容忽视。
- 因此有一个好的算法的长期效果不容忽视。

BF 算法（暴力算法）

- 就是对每个 S 的位置 i，与 T 进行一一对比；当不对应时 T 串回溯到首位，与 S 的位置 $i+1$ 开始重新比较。
- 正确率有绝对的保证。
- 时间复杂度大概为 $O(|S| \cdot |T|)$。
- 自然的问题：这个是否能改进？

BF 算法：例子及其启示

- $S =$ "$ababcabcacbab$"
- $T =$ "$abcac$"

$$ababcabcacbab$$

$$abcac$$

BF 算法：例子及其启示

- $S =$ "$ababcabcacbab$"
- $T =$ "$abcac$"

$$\color{red}{a}babcabcacbab$$

$$\color{red}{a}bcac$$

BF 算法：例子及其启示

- $S=$ “$ababcabcacbab$”
- $T=$ “$abcac$”

$$a\textcolor{red}{b}abcabcacbab$$

$$a\textcolor{red}{b}cac$$

BF 算法：例子及其启示

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

ababcabcacbab

abcac

BF 算法：例子及其启示

- $S=$ "$ababcabcacbab$"
- $T=$ "$abcac$"

$$a\textcolor{red}{b}abcabcacbab$$

$$\textcolor{red}{a}bcac$$

BF 算法：例子及其启示

- $S =$ "$ababcabcacbab$"
- $T =$ "$abcac$"

$$ab\color{red}{a}\color{black}{bcabcacbab}$$

$$\color{red}{a}\color{black}{bcac}$$

BF 算法：例子及其启示

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

ababcabcacbab

abcac

BF 算法：例子及其启示

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

ababcabcacbab

abcac

BF 算法：例子及其启示

- $S =$ "$ababcabcacbab$"
- $T =$ "$abcac$"

$$ababcabcacbab$$

$$abcac$$

BF 算法：例子及其启示

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

$$ababca\textcolor{red}{b}cacbab$$

$$abca\textcolor{red}{c}$$

BF 算法：例子及其启示

- $S =$ “$ababcabcacbab$”
- $T =$ “$abcac$”

$$ababcabcacbab$$

$$abcac$$

BF 算法：例子及其启示

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

ababcabcacbab

abcac

BF 算法：例子及其启示

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

$$ababcabcacbab$$

$$abcac$$

BF 算法：例子及其启示

- $S =$ "$ababcabcacbab$"
- $T =$ "$abcac$"

$$ababca\textcolor{red}{b}cacbab$$

$$a\textcolor{red}{b}cac$$

BF 算法：例子及其启示

- $S =$ "$ababcabcacbab$"
- $T =$ "$abcac$"

$$ababcabcacbab$$

$$abcac$$

BF 算法：例子及其启示

- $S =$ “$ababcabcacbab$”
- $T =$ “$abcac$”

$$ababcabc\textcolor{red}{a}cbab$$

$$abc\textcolor{red}{a}c$$

BF 算法：例子及其启示

- $S =$ “$ababcabcacbab$”
- $T =$ “$abcac$”

$$ababcabcacbab$$

$$abcac$$

至此我们成功找到 T 的位置是 5。

KMP 算法的起源

- BF 算法不好的地方是大量的回溯，没有利用好已知部分匹配的结果。
- 不过如果我们不对 T 每次完全回溯，那就需要直接把 T 向右滑动。
- 具体我们再回到前面的例子……

KMP 算法：概念

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

ababcabcacbab

abcac

KMP 算法：概念

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

ababcabcacbab

abcac

KMP 算法：概念

- $S =$ "ababcabcacbab"
- $T =$ "abcac"

ababcabcacbab

abcac

KMP 算法：概念

- $S =$ “$ababcabcacbab$”
- $T =$ “$abcac$”

$$ababcabcacbab$$

$$abcac$$

KMP 算法：概念

- $S =$ “$ababcabcacbab$”
- $T =$ “$abcac$”

$$ababcabcacbab$$

$$\ \ abcac$$

这个比较其实是不需要的，因为根据我们目前已知：前两位对上了，而且不一样，因此上部失配时我们可以直接跳到下一步

KMP 算法：概念

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

$$ababcabcacbab$$

$$abcac$$

KMP 算法：概念

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

$$aba\textcolor{red}{b}cabcacbab$$

$$a\textcolor{red}{b}cac$$

KMP 算法：概念

- $S =$ "$ababcabcacbab$"
- $T =$ "$abcac$"

$$ababcabcacbab$$

$$abcac$$

KMP 算法：概念

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

ababcabcacbab

abcac

KMP 算法：概念

- $S =$ “*ababcabcacbab*”
- $T =$ “*abcac*”

ababcabcacbab

abcac

这个位置失配了，而根据之前的配对，我们知道我们应该可以往前移三个位置

KMP 算法：概念

- $S =$ "ababcabcacbab"
- $T =$ "abcac"

$$ababcabcacbab$$

$$abcac$$

这个位置失配了，而根据之前的配对，我们知道我们应该可以往前移三个位置

KMP 算法：概念

- $S =$ “*ababcabcacbab*”
- $T =$ “*abcac*”

$$ababcabcacbab$$

$$abcac$$

这个位置失配了，而根据之前的配对，我们知道我们应该可以往前移三个位置

KMP 算法：概念

- $S =$ "$ababcabcacbab$"
- $T =$ "$abcac$"

$$ababc\textcolor{red}{a}bcacbab$$

$$\textcolor{red}{a}bcac$$

KMP 算法：概念

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

$$ababca\color{red}{b}\color{black}{cacbab}$$

$$a\color{red}{b}\color{black}{cac}$$

KMP 算法：概念

- $S =$ "*ababcabcacbab*"
- $T =$ "*abcac*"

$$ababcab\textcolor{red}{c}acbab$$

$$ab\textcolor{red}{c}ac$$

KMP 算法：概念

- $S =$ "$ababcabcacbab$"
- $T =$ "$abcac$"

$$ababcabc\textcolor{red}{a}cbab$$

$$abc\textcolor{red}{a}c$$

KMP 算法：概念

- $S =$ "$ababcabcacbab$"
- $T =$ "$abcac$"

$$ababcabcac\color{red}{c}bab$$

$$abca\color{red}{c}$$

至此我们成功找到 T 的位置是 5。

KMP 算法：滑动距离

$$\begin{array}{lcl} abcabcacabc & \Longrightarrow & abcabcacabc \\ abcabcb & \Longrightarrow & \quad abcabcb \end{array}$$

- 假设我们目前在配对 S 的位置 i 和 T 的第 j 个位置。错配后，如果应该把模式滑动到 T 的第 k 个位置，那其实代表了：
- $T_0T_1\dots T_{j-1} = S_{i-j}S_{i-j+1}\dots S_{i-1}$，
- $T_0T_1\dots T_{k-1} = S_{i-k}S_{i-k+1}\dots S_{i-1}$。
- 那其实我们可以推断到：$T_{j-k}T_{j-k+1}\dots T_{j-1} = T_0T_1\dots T_{k-1}$。

KMP 算法：滑动距离（续）

- 如果对于 j 有唯一（或没有）这样的 k，那么滑动距离十分明显。
- 如果有多个这样的 k 又怎样？为了没有漏配，当然只能选最大的。

KMP 算法：滑动距离（续）

- 如果对于 j 有唯一（或没有）这样的 k，那么滑动距离十分明显。
- 如果有多个这样的 k 又怎样？为了没有漏配，当然只能选最大的。

$$ABABAB\textcolor{red}{B}ABABABA$$

$$ABABAB\textcolor{red}{A}$$

KMP 算法：滑动距离（续）

- 如果对于 j 有唯一（或没有）这样的 k，那么滑动距离十分明显。
- 如果有多个这样的 k 又怎样？为了没有漏配，当然只能选最大的。

$$\mathit{ABABA}{\color{blue}\mathit{AB}}{\color{red}\mathit{B}}\mathit{ABABABA}$$

$${\color{blue}\mathit{AB}}\mathit{ABAB}{\color{red}\mathit{A}}$$

KMP 算法：滑动距离（续）

- 如果对于 j 有唯一（或没有）这样的 k，那么滑动距离十分明显。
- 如果有多个这样的 k 又怎样？为了没有漏配，当然只能选最大的。

$ABABABBABABABA$

$ABABABA$

KMP 算法：滑动距离（续）

- 如果对于 j 有唯一（或没有）这样的 k，那么滑动距离十分明显。
- 如果有多个这样的 k 又怎样？为了没有漏配，当然只能选最大的。

$$\mathit{AB}\color{blue}{\mathit{ABAB}}\color{red}{\mathit{B}}\color{black}{\mathit{ABABABA}}$$

$$\color{blue}{\mathit{ABAB}}\color{black}{\mathit{ABA}}$$

KMP 算法：滑动距离（续）

- 如果对于 j 有唯一（或没有）这样的 k，那么滑动距离十分明显。
- 如果有多个这样的 k 又怎样？为了没有漏配，当然只能选最大的。

$$ABAB\color{blue}{AB}\color{red}{B}\color{black}{ABABABA}$$

$$\color{blue}{AB}\color{black}{ABABA}$$

KMP 算法：next 数组

- 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（也就是到底应该会苏多少）。
- 根据之前的分析，我们有：

$$\text{next}[j] := \begin{cases} -1 & \text{当 } j = 0 \\ \max\left(\max\{k \middle| 1 \le k < j, T_0 T_1 \ldots T_{k-1} = T_{j-k} \ldots T_{j-1}\}, 0\right) & \text{当} j \neq 0 \end{cases}.$$

- 这个算法里，i 从来没有回溯。如果你检查一下 T 的动态，T 关于 S 的相关位置也是没有向后退，因此除掉计算 next 数组的时间，算法时间复杂度为 $O(|S| + |T|)$。

KMP 算法：next 数组例子

- T = A B A B C

KMP 算法：next 数组例子

- `T =   A B A B C`
- `next:-1 0 0 1 2`

KMP 算法：next 数组例子

- T = A B A B C
- next:-1 0 0 1 2

$$ABABAABABCB$$
$$ABABC$$

KMP 算法：next 数组例子

- `T =    A B A B C`
- `next:-1 0 0 1 2`

$$ABABAABABCB$$

$$ABABC$$

这个时候失配的是 $j = 4$，查看 next 数组有新的 $j = 2$。

KMP 算法：next 数组例子

- T =   A B A B C
- next:-1 0 0 1 2

ABABAABABCB

ABABC

这个时候失配的是 $j = 4$，查看 next 数组有新的 $j = 2$。

KMP 算法：next 数组例子

- `T =    A B A B C`
- `next:-1 0 0 1 2`

$$ABABA\textcolor{red}{A}BABCB$$

$$ABA\textcolor{red}{B}C$$

这个时候失配的是 $j = 3$，查看 next 数组有新的 $j = 1$。

KMP 算法：next 数组例子

- `T =    A B A B C`
- `next:-1 0 0 1 2`

$$ABABAABABCB$$

$$ABABC$$

这个时候失配的是 $j = 3$，查看 next 数组有新的 $j = 1$。

KMP 算法：next 数组例子

- `T =    A B A B C`
- `next:-1 0 0 1 2`

$$ABABAABABCB$$

$$ABABC$$

这个时候失配的是 $j=1$，查看 next 数组有新的 $j=0$。

KMP 算法：next 数组例子

- `T =    A B A B C`
- `next:-1 0 0 1 2`

$$ABABAABABCB$$

$$ABABC$$

这个时候失配的是 $j = 1$，查看 next 数组有新的 $j = 0$。

KMP 算法：next 数组例子

- T =   A B A B C
- next:-1 0 0 1 2

$$\mathit{ABABA}\textcolor{red}{\mathit{ABABC}}\mathit{B}$$

$$\textcolor{red}{\mathit{ABABC}}$$

完全配上了，配对好的位置为 $i = 5$。

KMP 算法：next 数组的有效算法

我们先从例子看起吧……

A	B	A	A	B	B	A	B	A

KMP 算法：next 数组的有效算法

我们先从例子看起吧……

A	B	A	A	B	B	A	B	A
-1	0							

KMP 算法：next 数组的有效算法

我们先从例子看起吧……

A	B	A	A	B	B	A	B	A
-1	0	0						

KMP 算法：next 数组的有效算法

我们先从例子看起吧……

A	B	A	A	B	B	A	B	A
-1	0	0						

KMP 算法：next 数组的有效算法

我们先从例子看起吧……

A	B	A	A	B	B	A	B	A
-1	0	0	1					

KMP 算法：next 数组的有效算法

我们先从例子看起吧……

A	B	A	A	B	B	A	B	A
-1	0	0	1					

KMP 算法：next 数组的有效算法

我们先从例子看起吧……

A	B	A	A	B	B	A	B	A
-1	0	0	1					

KMP 算法：next 数组的有效算法

我们先从例子看起吧……

A	B	A	A	B	B	A	B	A
-1	0	0	1	1				

KMP 算法：next 数组的有效算法

我们先从例子看起吧……

A	B	A	A	B	B	A	B	A
-1	0	0	1	1				

KMP 算法：next 数组的有效算法

我们先从例子看起吧……

A	B	A	A	B	B	A	B	A
-1	0	0	1	1	2			

KMP 算法：next 数组的有效算法（续）

假定我们要算 next[j]：

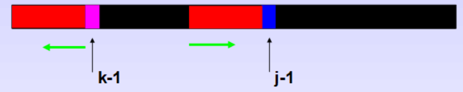
图中红色的部分为 next[j-1] 表示前面部分相配的最长前缀、后缀，为了方便解说，记为 k。

若 $T_k = T_{j-1}$，那么则有 $T_0 \ldots T_k = T_{j-k-1} \ldots T_{j-1}$，也就是说这时有 next[j] = next[j-1]+1。

$T_k \neq T_{j-1}$ 呢？

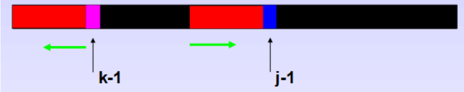
next 数组计算：当 $T_k \neq T_{j-1}$ 时

- 简单来说，这个时候答案肯定比 $k+1$ 小，也就要往小一点的 k 检查。



next 数组计算：当 $T_k \neq T_{j-1}$ 时

- 简单来说，这个时候答案肯定比 $k+1$ 小，也就要往小一点的 k 检查。



- 但是每个 k 都检查的话好像太浪费了……其实我们在算 next 的本质就是拿 T 与自己匹配。所以我们还可以用类似回溯手段的。（使用红色部分的最大匹配的前缀与后缀）