

字符串1

以下若无特殊说明，字符串下标都是从1开始

字符串表面上就是只存字符的动态数组，即 `vector<char>`，但它拥有一些普通数组不关心的问题，例如字符串匹配问题，回文问题，等等。同时，字符串之间还可以比较大小（我们一般不会关系普通数组之间的“大小”），该比较被称为“字典序”。两个字符串 s 和 t 的大小关系的比较方法如下：从左到右按次比较 $s[i]$ 和 $t[i]$ ，如果首次发现了 $s[i] \neq t[i]$ ，那么谁小，哪个字符串就小；如果一直没发现 $s[i] \neq t[i]$ ，那么哪个字符串短，哪个就小；否则就相等。容易验证，这个字典序是个偏序关系（这就导致了二叉搜索树 Binary Search Tree 里也可以直接存字符串。以后会介绍 BST）。

字符串的题目可以很简单，也可以非常的难。这里不讨论非常进阶的字符串问题算法，只考虑比较简单的问题。

自动机模型

自动机有很多种。这里只介绍最简单的确定性有限自动机。

确定性有限自动机（Deterministic Finite Automaton，简称 DFA）是一种数学模型，用于描述一个具有有限个状态的系统，其中系统的状态转移完全由当前状态和输入符号唯一决定。DFA 是有限状态自动机的一种特殊形式，具有确定性的特点。

DFA 的组成要素：

1. **状态集合（States，记作 Q ）**：系统可能处于的有限个状态的集合。
2. **输入字母表（Alphabet，记作 Σ ）**：系统能够接收的输入符号的有限集合。
3. **转移函数（Transition Function，记作 δ ）**：定义了状态和输入符号之间的关系，形式为 $\delta : Q \times \Sigma \rightarrow Q$ 。
4. **初始状态（Start State，记作 q_0 ）**：系统开始时所处的状态，属于状态集合 Q 。
5. **接受状态集合（Accept States，记作 F ）**：系统在这些状态下被认为接受输入串，是状态集合 Q 的子集。

DFA 的工作原理：

- **输入处理**：DFA 从初始状态开始，依次读取输入串中的每一个符号。
- **状态转移**：在每一步，根据当前状态和当前输入符号，通过转移函数 δ 唯一确定下一个状态。
- **接受条件**：当输入串全部读取完毕后，如果最终状态属于接受状态集合 F ，则 DFA 接受该输入串；否则，拒绝。

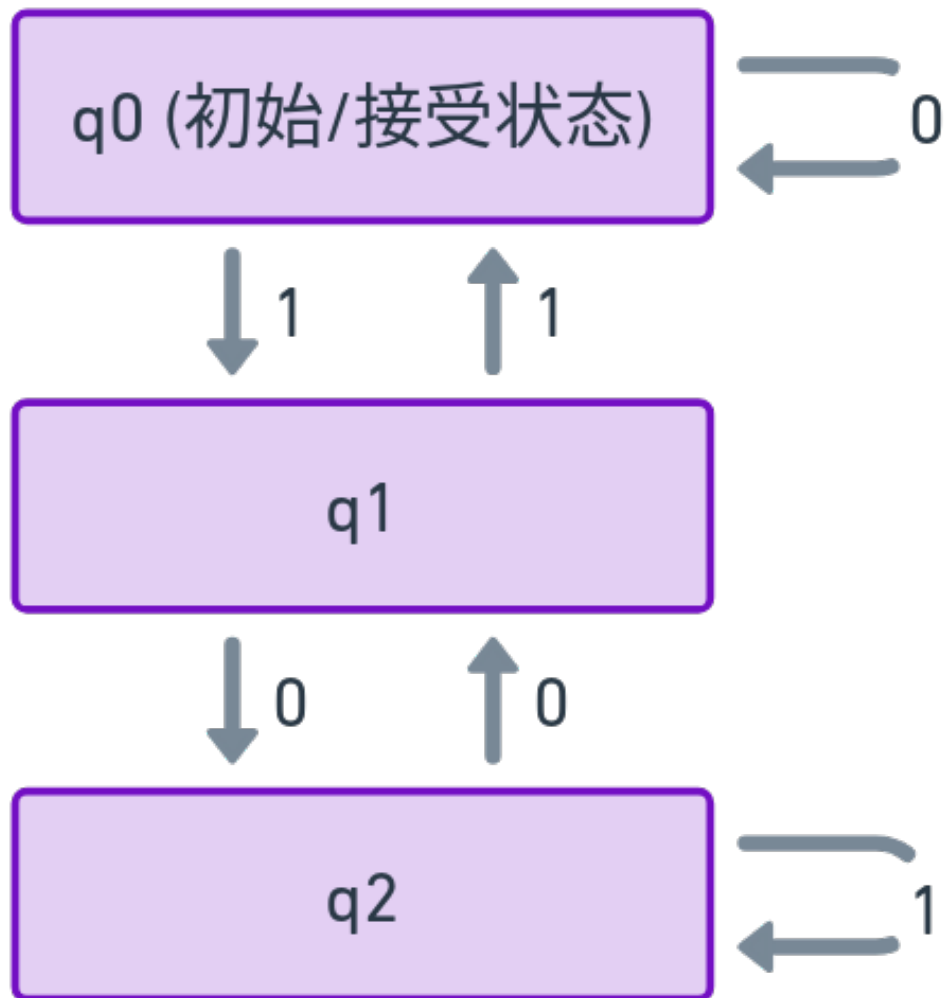
自动机的例子

假设需要构建一个 DFA，用于识别所有能被 3 整除的二进制数。

- 状态集合: $Q = \{q_0, q_1, q_2\}$, 分别表示当前余数为0、1、2。
- 输入字母表: $\Sigma = \{0, 1\}$
- 转移函数:
 - $\delta(q_0, 0) = q_0$; $\delta(q_0, 1) = q_1$
 - $\delta(q_1, 0) = q_2$; $\delta(q_1, 1) = q_0$
 - $\delta(q_2, 0) = q_1$; $\delta(q_2, 1) = q_2$
- 初始状态: q_0
- 接受状态集合: $F = \{q_0\}$

在这个 DFA 中，每个状态表示当前读取的二进制数除以3的余数。通过状态转移，可以判断任意一个二进制串是否能被3整除。

示意图如下：



根据这个状态机，我们可以给几个例子：

例子1

输入：110（对应十进制的 6）

- 初始状态：q0（余数为 0，接受状态）
- 读取第一位 1
 - 当前状态：q0
 - 转移条件：1
 - 新状态：q1 $((0 \times 2 + 1) \% 3 = 1)$
- 读取第二位 1
 - 当前状态：q1
 - 转移条件：1
 - 新状态：q0 $((1 \times 2 + 1) \% 3 = 0)$
- 读取第三位 0
 - 当前状态：q0
 - 转移条件：0
 - 新状态：q0 $((0 \times 2 + 0) \% 3 = 0)$

最终状态：q0（接受状态）

结论：输入的二进制数 110 能被 3 整除。

例子2

输入：101（对应十进制的 5）

- 初始状态：q0（余数为 0，接受状态）
- 读取第一位 1
 - 当前状态：q0
 - 转移条件：1
 - 新状态：q1 $((0 \times 2 + 1) \% 3 = 1)$
- 读取第二位 0
 - 当前状态：q1
 - 转移条件：0
 - 新状态：q2 $((1 \times 2 + 0) \% 3 = 2)$
- 读取第三位 1
 - 当前状态：q2
 - 转移条件：1

- 新状态: **q2** ($(2 \times 2 + 1) \% 3 = 2$)

最终状态: **q2** (非接受状态)

结论: 输入的二进制数 101 不能被 3 整除。

简单来说, DFA就是判断某个串是否满足某种条件的一个数学模型。

现在我们介绍一个关于字符串的最简单的DFA, 序列自动机。

序列自动机

考虑一个非常简单的问题。设字符串 s 长 n , 给出若干询问, 每次询问问某个串 t 是否是 s 的子序列。不妨假设 t 的长度是 m 。

如果只询问一次, 算法是很简单的, 只需要在 s 里依次寻找 $t[1], t[2], \dots$, 如果能一直寻找到 $t[m]$, 则说明 t 是 s 的子序列。单次询问的时间复杂度是 $O(n + m)$, 不可能更低了。

如果询问多次, 我们考虑能否构造一个数据结构, 使得每次询问的时间复杂度降下来。我们定义 $f[i][c]$ 为, 字符 c 在 $s[i + 1 :]$ 首次出现的位置, 如果不出现则定义其为0。假设 f 已经全部求出, 那么对于 t , 查询的过程如下:

1. 首先我们要看 $t[1]$ 是否出现, 于是看 $f[0][t[1]]$, 如果 $f[0][t[1]] = -1$, 那 t 不可能是 s 的子序列, 返回false。否则令 $x_1 = f[0][t[1]]$, 说明 $s[x_1] = t[1]$;
2. 接下来要看 $s[x_1 + 1 :]$ 里 $t[2]$ 首次出现的位置, 于是看 $f[x_1][t[2]]$, 如果其等于 -1 , 那也返回false。否则令 $x_2 = f[x_1][t[2]]$;
3. 以此类推, 继续看 $t[3], t[4], \dots$, 一直查询到 $f[x_{m-1}][t[m]]$, 如果依然不等于 -1 , 那整个 t 查询完毕了, 说明 t 确实是 s 的子序列。

我们可以看到, 如果有 $n + 2$ 个状态分别记为 $-1, 0, 1, 2, \dots, n$, 初始状态为0, 那么 f 其实是定义了一个状态转移函数, 每次读到一个 t 的字符, f 就规定了下一个跳到的状态是什么。只要读完 t 之后, 停留在了非 -1 的状态, 那么都说 t 是 s 的子序列, 即非 -1 状态就是接受状态集; 而 -1 其实是一个“死亡状态”, 即一旦进入了这个状态, 那之后再出不来了。算法里我们可以直接加入if判断, 如果走到了 -1 即返回false。注意到按照这种定义, 空串也算是 s 的子序列。

接下来考虑该DFA的转移函数 f 怎么构建。首先 $\forall c, f[n][c] = -1$ 。接着对于下标 k , 如果 $s[k + 1] = c$, 那么显然 $f[k][c] = k + 1$, 而 $\forall d \neq c, f[k][d] = f[k + 1][d]$ 。其含义是非常显然的。假设所有字符串都只含英文小写字母, 那么程序可以这么写:

```

1  vector<vector<int>> buildDFA(string& s) {
2      int n = s.size();
3      // 为了让s下标从1开始，这里前面加一个空格
4      s = " " + s;
5      vector<vector<int>> dfa(n + 1, vector<int>(26, -1));
6      for (int i = n - 1; i >= 0; i--)
7          for (char ch = 'a'; ch <= 'z'; ch++)
8              dfa[i][ch] = s[i + 1] == ch ? i + 1 : dfa[i + 1][ch];
9      return dfa;
10 }

```

而回答询问的函数可以这么写：

```

1  bool query(string& t) {
2      int x = 0;
3      for (char ch : t)
4          if (!~(x = dfa[x][ch])) return false;
5      return true;
6  }

```

预处理时间复杂度 $O(n|\Sigma|)$ ， Σ 是字母表集合（上面例子里 Σ 就是全体英文小写字母）。每次询问时间 $O(m)$ ，比单次询问时间快很多。

作业

实现序列自动机。上面是用vector实现的，当然还可以用unordered_map来实现以达到空间的节省。

Leetcode 1055