字符串匹配算法之——

Sunday算法

目 录
Contents

算法原理

1. 思路
2. 步骤
3. 实例

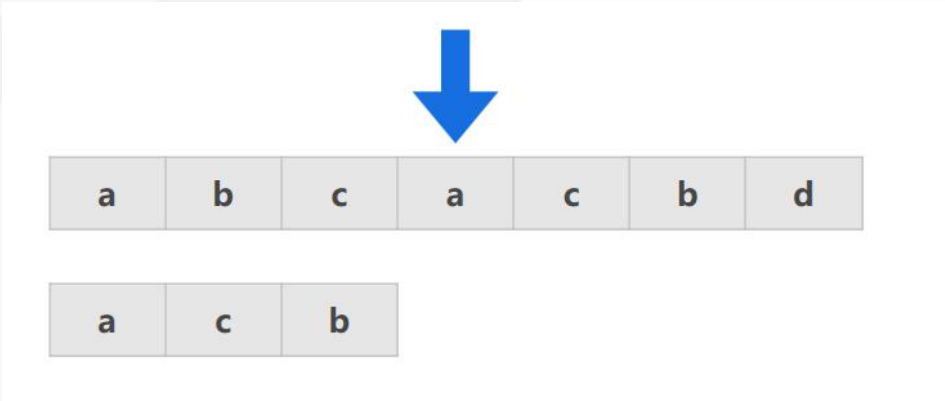
算法原理—思路简述 I

先来看看DS的简介：
Sunday算法的核心思想是利用匹配失败时，文本串中下一个未参与匹配的字符信息，动态调整子串的跳跃步长，从而减少无效比较次数。

概括一下，就是：

在匹配失败时，
重点关注母串中
参加匹配的最末位字符的下一位字符

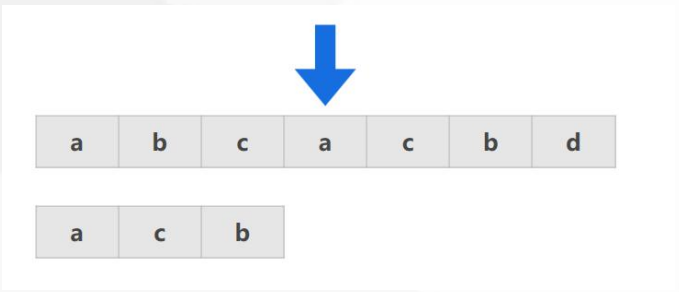
图解一下：



此时匹配失败，
关注母串 abcacbd 中
参加匹配的最末位字符c
的下一位字符a

不妨把这个字符记为ch.

算法原理—思路概述II



关注这个的原因是它能指导我们下一步的匹配。

具体而言：

①如果该字符没有出现在子串中，则直接移动到ch的下一位开始匹配。
即：**子串的移动位数 = 子串长度 + 1**

②否则，子串与母串按ch对齐。
即：**子串的移动位数**
= 子串长度 - 该字符最右出现的位置(以0开始)
= 子串中该字符最右出现的位置到尾部的距离 + 1

算法原理—实现步骤

根据刚刚的思路，可以概括出具体实现的步骤：

首先进行**预处理**：

记录子串中每个字符最后一次出现的位置

（e.g. 子串是"abc"，则a的位置是0，b是1，c是2）

这一步的作用是为匹配打基础

然后**开始匹配**：

将子串和文本对齐，从左到右逐个字符比较。

如果完全匹配：

匹配上了！直接返回当前位置。

如果不匹配：

①先看文本中子串末尾后一个字符

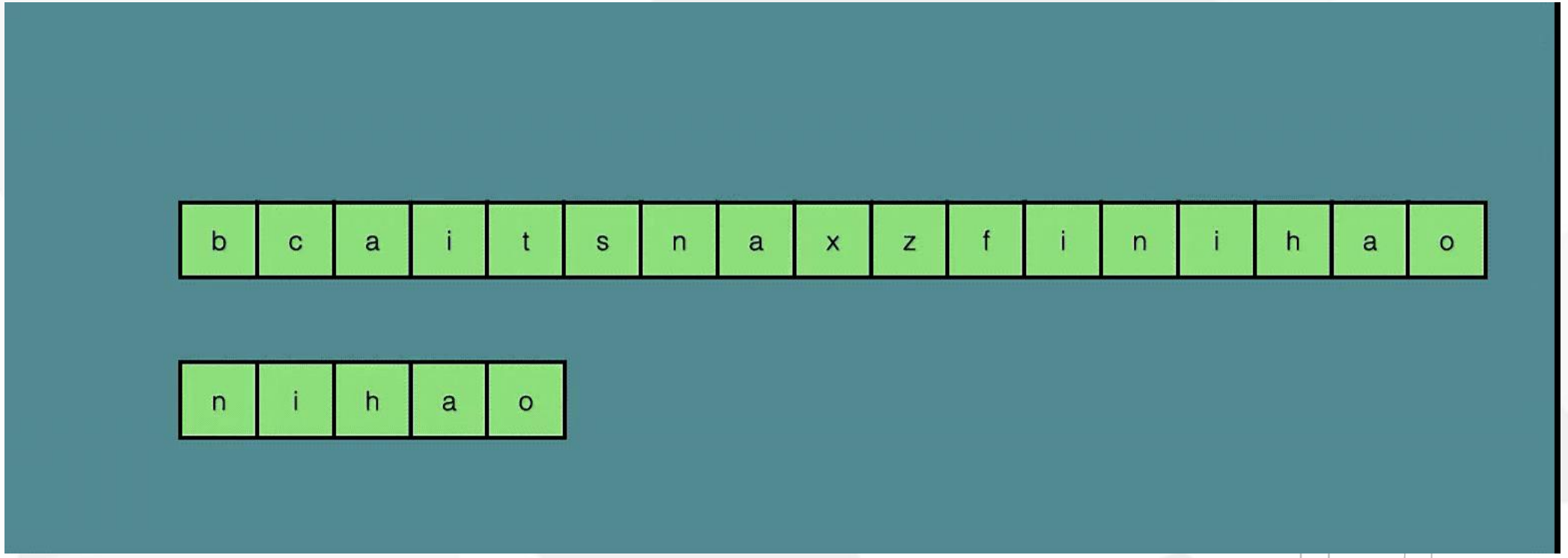
（e.g. 子串长度是3，当前比较到文本第5位，就看第6位的字符）

②根据预处理的位置表，决定移动多少步

（e.g. 若这个字符在子串最后出现的位置是1，就移动3-1=2步；现在字符不在子串中，直接移动3+1=4步）

重复匹配过程，直到找到匹配或超出文本范围。

算法原理—实例说明



注：由于PDF中不可查看视频，请访问网址：

https://zhuanlan.zhihu.com/p/142895983

其中有此动图。

算法实现

注：代码使用C语言实现

1. 预处理阶段
2. 匹配阶段

算法实现—预处理阶段

事实上，前文“记录子串中每个字符最后一次出现的位置”就是“建立偏移表”的过程。

建立偏移表的C语言代码实现如下，必要的解释以注释形式呈现：

```c
// 预处理阶段：生成字符偏移表
void sunday_preprocess(const char *pattern, int m, int shift[]) {
    // 初始化所有字符的默认偏移值（子串长度+1）
    // 默认偏移即是字符不在子串中时，直接移动到文本中子串末尾的后一个字符
    for(int i=0; i<ASCII_SIZE; i++)
        shift[i] = m + 1;

    // 更新子串中存在的字符偏移值
    // 对子串存在的字符，计算最后一次出现位置到末尾的距离，对偏移值进行特殊处理
    for(int j=0; j<m; j++) {
        // 字符最后一次出现的位置到末尾的距离
        shift[(unsigned char)pattern[j]] = m - j;
    }
}
```

注：代码部分主要由DS生成，核心注释为人工添加，之后不特殊说明均如此

算法实现—匹配阶段

右侧的代码直接看起来挺头疼的。

所以我把先前的步骤与右侧代码进行一一对应，
结合图中注释和之前动图应该可以理解：

1. 获取文本和模式串的长度
2. 预处理生成偏移表
3. 匹配循环中先逐字符比对：
4. 如果完全匹配：返回匹配位置
5. 如果没匹配上：

   计算下一个字符的偏移，并根据偏移表移动子串
6. 未找到则返回-1

```c
// 匹配阶段
int sunday_search(const char *text, const char *pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int shift[ASCII_SIZE];
    sunday_preprocess(pattern, m, shift);   // 生成偏移表

    // 匹配循环
    int i = 0;  // i表示文本当前匹配的位置
    while(i <= n - m) {
        int k = i;      // 记录当前比对起点
        int j = 0;      // 记录子串正在匹配的字符位置

        // 逐字符比对
        while(j < m && text[k] == pattern[j]) {
            k++;
            j++;
        }

        // 如果完全匹配，返回匹配位置
        if(j == m) return i;

        // 如果没匹配上——

        if(i + m >= n) break;
        // 注意此处的特判！如果已到文本末尾则意味着没找到！

        // 计算下一个字符的偏移，并根据偏移表移动子串
        unsigned char next_char = text[i + m];

        // 据偏移表移动子串
        i += shift[next_char];
    }
    return -1;  // 未找到
}
```

算法分析

1. 时间复杂度
2. 空间复杂度
3. 与其他字符串匹配算法的对比

算法分析—时空复杂度

时间复杂度

平均时间复杂度

Sunday算法的时间复杂度在平均情况下很不错，尤其是处理普通文本的时候。
这是因为他基本上能跳过大段不需要比较的地方。

平均时间复杂度为

$O(n/m)$

这里n是文本长度，m是模式串长度

最坏时间复杂度

Sunday算法在极端情况下效率比较低。
比如子串全是重复的字符（比如"aaaaa"），而文本里又有一堆类似的重复（比如"aaaabaaaab..."）。
由于这时候子串每次只能挪一两位，且每次匹配还几乎要匹配到底。所以：

最坏时间复杂度为$O(n*m)$

这跟暴力匹配差不多。
不过这种情况现实中不常见，所以Sunday算法还是很实用的。

空间复杂度

Sunday算法需维护一个大小为字符集大小的偏移表，例如ASCII字符集对应256长度的数组，
所以空间复杂度不大。

算法分析—与其他字符串匹配算法对比

列表对比分析如下

对比维度	朴素算法	KMP算法	BM算法	Sunday算法
核心思想	暴力逐字符匹配	最长公共前后缀跳跃	坏字符+好后缀双规则跳跃	利用下一个字符快速跳跃
平均时间复杂度	O(nm)	O(n + m)	O(n)	O(n/m)
最坏时间复杂度	O(nm)	O(n + m)	O(nm)	O(nm)
空间复杂度	O(1)	O(m)	O(m + Σ)	O(Σ)
适用场景	短文本且代码简单	重复子串多、稳定性高	大字符集（如中文）	随机文本、小字符集（如英文）

THANKS!

下载PPT及源码请访问：https://github.com/heathera-Jiang/HNU_2025DataStructure

请star请star请star谢谢啦

To_Teacher： 老师，我检查过了，这里面没有泄露我个人信息（包括头像、性别和其他仓库等）