Hash & Trie & KMP

唐懿宸

清华大学致理书院

Hash 哈希

字符串是复杂的，常规的字符串比较方法都是从两个串的开头一个一个字符地比较，这样的时间复杂度是$O(n)$的。

我们希望有一个高效的方法来表示这个字符串，满足可以快速比较两个字符串是否相等。

Hash

我们知道数字的比较是简单直接的（$O(1)$），如果每一个字符串都能映射到一个数字上，并且不同的字符串映射到不同的数字上，我们就可以用这个数字来比较字符串。

因此，直观上看，hash 是一个映射函数 f，$f : \text{String} \to \text{Integer}$。

Hash

但是 `long long` 的范围只有 2^{64}，而长度为 n 的小写字母字符串就有 26^n 种，而为了这个映射写一个高精度反而丢弃了我们快速比较的要求。

数字比字符串少，根据鸽巢原理，必然会有两个字符串映射到同一个数字上。

我们将 hash 函数值一样但原字符串不一样的现象称为哈希碰撞。

目标是设计一个足够好的 hash 函数来让碰撞的概率尽可能小。

Hash 函数设计

通常采用多项式 hash，对于一个长度为 n 的字符串 S，定义它的 hash 函数为：

$$\text{hash}(S) = \sum_{i=1}^{n} S[i] \times b^{n-i} \bmod M$$

其中 b 是一个大于字符集大小的数，M 是一个大数，用来控制 hash 值大小。直观上看就是把字符串视为了一个 b 进制数。

显然你也可以用 $\sum_{i=1}^{n} S[i] \times b^{i-1} \bmod M$。用的时候不要记混了。

Hash 冲突的概率

在字符串足够长 **并且随机** 的情况下，我们可以认为 hash 值是随机分布的。假设需要被 hash 的字符串数量是 n，hash 函数的值域大小为 d，则不出现 hash 冲突的概率为
$1 \times \left(1 - \frac{1}{d}\right) \times \left(1 - \frac{2}{d}\right) \times \cdots \times \left(1 - \frac{n-1}{d}\right) = \frac{d!}{d^n (d-n)!}$

因为 $\exp(x)$ 在 $x \to 0$ 的时候趋近于 $1 + x$，我们用 $exp(-\frac{x}{d})$ 替换上面的 $1 - \frac{x}{d}$，上式子可以转换成 $1 \times \prod_{i=1}^{n-1} \exp(-\frac{i}{d}) = \exp(-\frac{n(n-1)}{2d})$

那么出现 hash 冲突的概率就是 $1 - \exp(-\frac{n(n-1)}{2d})$

Hash 函数设计

对于 M，往往有以下几种策略：

1. 令 M 是一个大质数，例如 10^9+7：

   根据上面的计算结果，取 $M=10^9+7$，随机生成 10^6 个长度为 6 的字符串，出现 hash 值相同的概率高达 90%

2. 用 `unsigned long long` 来作为 hash 结果，等价于 $M=2^{64}$（自然溢出）：

   虽然此时 M 很大导致 hash 值域很大，但是这种固定的方法可以通过精心设计的字符串卡掉

Hash 函数设计

3. 一个 M 不够用多个：

用两个及以上的大质数作为模数，只有所有取模的 hash 值相同才认为两个字符串相同

简单粗暴提升 hash 值域的方法，一般来说两个大质数就足够了。

显然你也可以用两个 b，分别对应两个 M。

Hash 实现

```
typedef unsigned long long ull;
ull base = 131;
ull mod1 = 212370440130137957, mod2 = 1e9 + 7;

pair<ull, ull> get_hash(std::string s) {
    int len = s.size();
    ull ans1 = 0, ans2 = 0;
    for (int i = 0; i < len; i++)
        ans1 = (ans1 * base + (ull)s[i]) % mod1;
    for (int i = 0; i < len; i++)
        ans2 = (ans2 * base + (ull)s[i]) % mod2;
    return make_pair(ans1, ans2);
}
```

Hash 函数使用

如果我们现在令 $\mathrm{hash}(S) = \sum_{i=1}^{n} S[i] \times b^{n-i}$

怎么快速得到 S 的子串的 hash 值呢?

Hash 函数使用

$$\text{hash}(S[1, l-1]) = \sum_{k=1}^{l-1} S[k] \times b^{l-k-1}$$

$$\text{hash}(S[1, r]) = \sum_{k=1}^{r} S[k] \times b^{r-k}$$

$$\begin{aligned}\text{hash}(S[l, r]) &= \sum_{k=1}^{r-l+1} S[l+k-1] \times b^{r-l-k+1} = \sum_{k=l}^{r} S[k] \times b^{k-l} \\ &= \text{hash}(S[1, r]) - \text{hash}(S[1, l-1]) \times b^{r-l+1}\end{aligned}$$

Hash 应用

直观上看就是做字符串匹配

按照 hash 的本质是一个映射 $f : \mathrm{A} \to \mathrm{Integer}$，这个 A 显然不要求一定是一个字符串，甚至可能是树、网格之类的
~~但是这是字符串专题所以不会讲~~

hash 的维护也有很多种方法，例如用线断树、平衡树维护 hash

Trie 树

先看一个基本问题：

给定 n 个字符串 $s_1, \cdots, s_n$ 和 q 个询问，每次询问给定一个字符串 t，求 $s_1 \cdots s_n$ 中有多少个字符串 s_i 满足 t 是 s_i 的前缀。

前缀：字符串 t 是字符串 s 的前缀当且仅当从 s 的末尾删去若干个连续字符（可以不删）后和 t 相同。

$1 \leqslant n, q \leqslant 10^5$，$\sum |s_i| \leqslant 5 \times 10^6$

Trie 树

Trie 树一种像字典一样的树，目的是为了匹配字符串，也叫字典树。

形式上说就是把一大堆字符串结构化成一棵树。

Trie 树

这棵字典树**用边来代表字母**，而从根结点到树上某一结点的路径就代表了一个字符串。

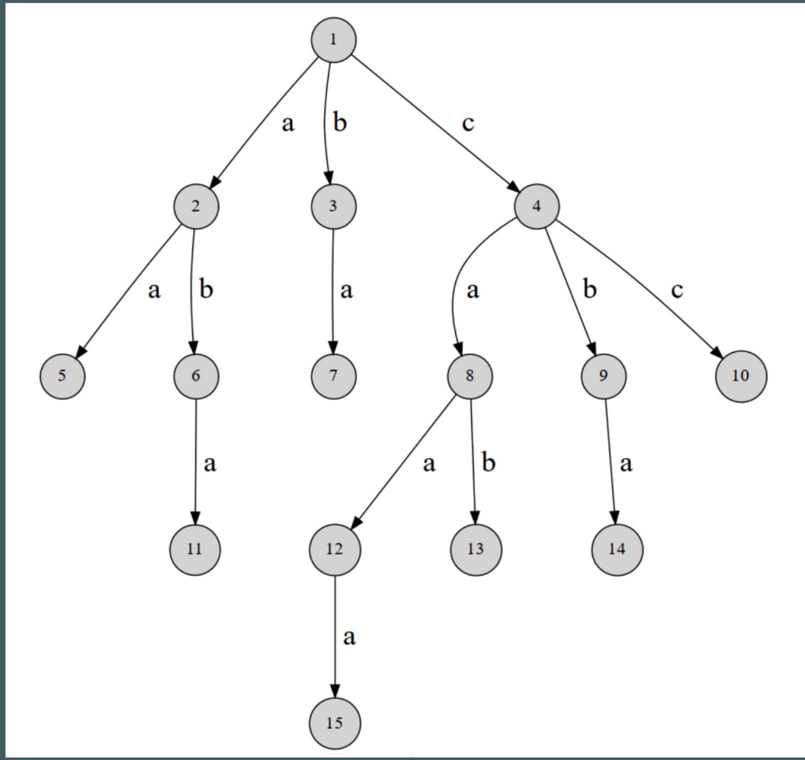
用 $\delta(u, c)$ 表示结点 u 的 c 字符边指向的下一个结点是谁，c 的取值范围和字符集大小有关，不一定是 $0 \sim 25$。

Trie 上的每个结点表示根节点到它的路径所形成的字符串。

根代表空串。

举个例子，对于左边的字符串，我们可以画出来右边的 Trie 树。

aa
aba
ba
caaa
caab
cba
cc



但是，对于这张图，我们只知道一定有字符串 `aba`，但是不能确定有没有字符串 `ab`。

所以，我们需要在每个结点上记录一个标记，表示这个结点是不是一个字符串的结尾。

也可能是标记有多少个字符串以这个结点为结尾。

在 Trie 上插入一个字符串就相当于建立一条从根节点出发的路径，并在末尾打上标记。

Trie 最适合用来处理匹配问题，尤其是**前缀匹配**。

Trie 树实现

树节点定义

```
struct node
{
    int son[62];
    int size; // 有多少个字符串包含这个结点对应的前缀
    void clr() // 如果有多组数据需要清空节点
    {
        size = 0;
        memset(son, 0, sizeof(son));
    }
} tr[N];
```

插入

```
void insert(char *str)
{
    int len = strlen(str), o = 0;
    for (int i = 0; i < len; i++)
    {
        tr[o].size++;
        int d = c2i(str[i]); // c2i 是将字符转换为数字
        if (!tr[o].son[d])
        {
            tr[o].son[d] = ++ncnt;
            tr[ncnt].clr();
        }
        o = tr[o].son[d];
    }
    tr[o].size++;
}
```

查询

```
int query(char *str)
{
    int len = strlen(str), o = 0;
    for (int i = 0; i < len; i++)
    {
        int d = c2i(str[i]);
        if (!tr[o].son[d])
            return 0;
        o = tr[o].son[d];
    }
    return tr[o].size;
}
```

Trie 树应用

直观上看就是做字符串匹配 ~~这句话在 Hash 应用上也出现过~~

Trie 树是 AC 自动机的一部分，会在明天的课程中继续展开

01-Trie 很适合用来维护异或和

可持久化 Trie

维护一棵 Trie，支持：查询历史版本，从第 k 个版本插入一个字符串作为新版本，在第 k 个版本查询一个字符串是否存在。

能维护这种东西的 Trie 叫做可持久化 Trie

本质上和主席树（可持久化线段树）没有任何区别。

KMP

超级匹配算法。

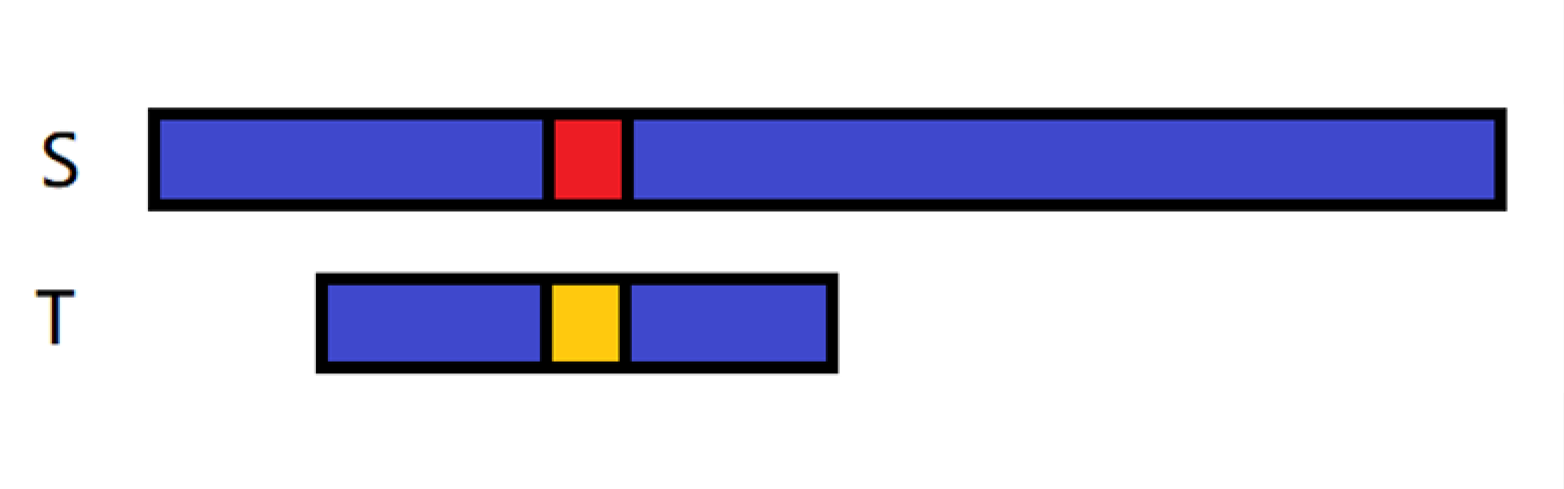
在 $O(n)$ 时间内实现匹配两个字符串的算法。

匹配：给定你两个字符串 S 和 T，需要 T 在 S 中所有出现的位置。

基础匹配算法

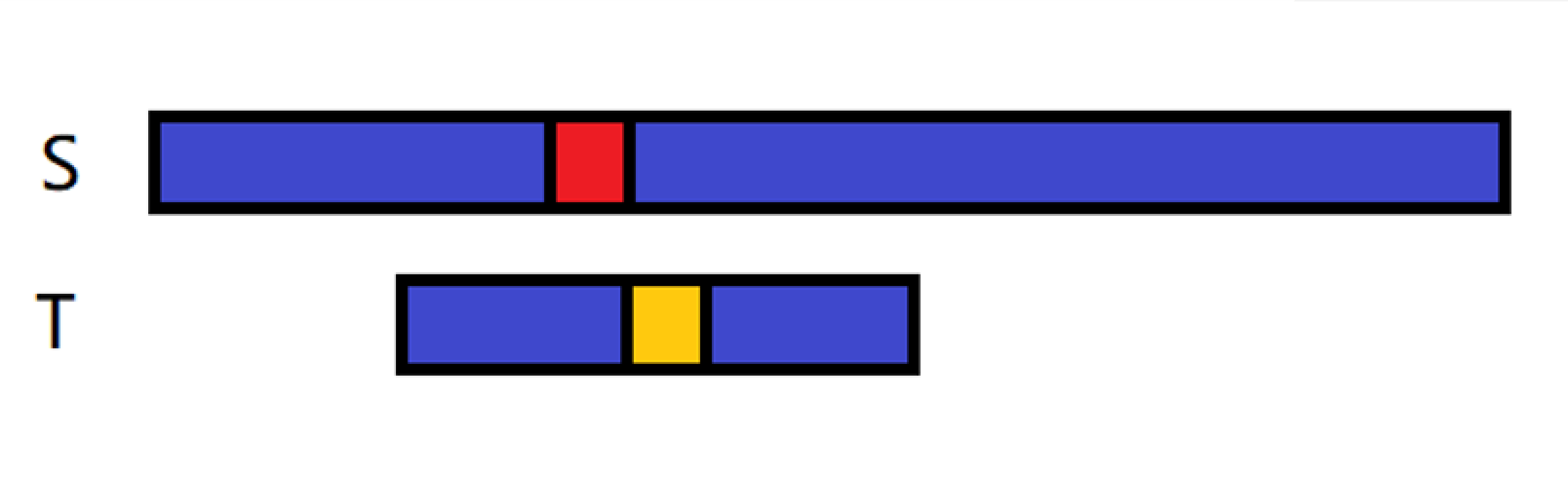
假如说我们现在要在字符串 S 中匹配字符串 T。

现在它们失配了



基础匹配算法

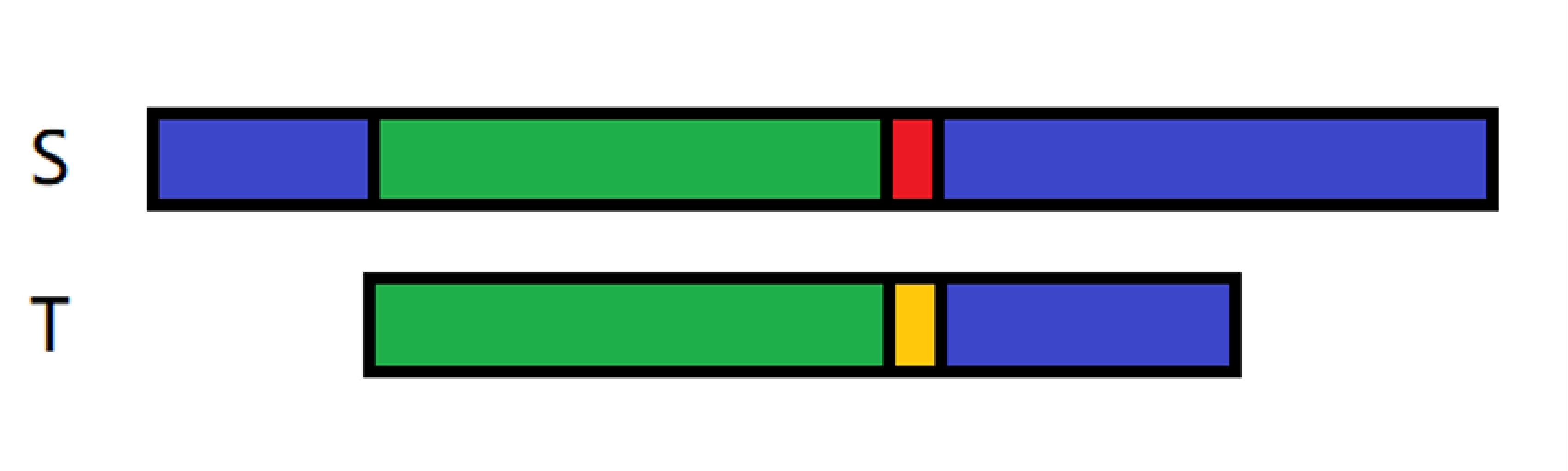
传统的暴力做法是：将 T 向右移动一格，然后继续匹配，如下图：



然后从 T 的开头开始重新匹配。这个算法的最大问题就是，每次移动后都需要从头开始匹配，就浪费了之前的匹配信息。

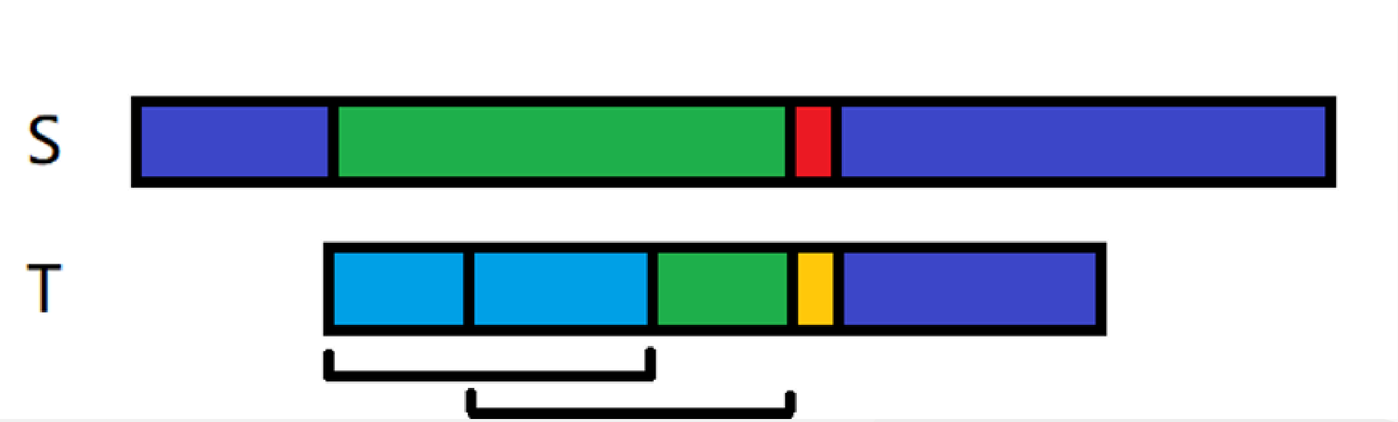
KMP

如果现在我们知道，至少这两个串的绿色部分是匹配的：



KMP

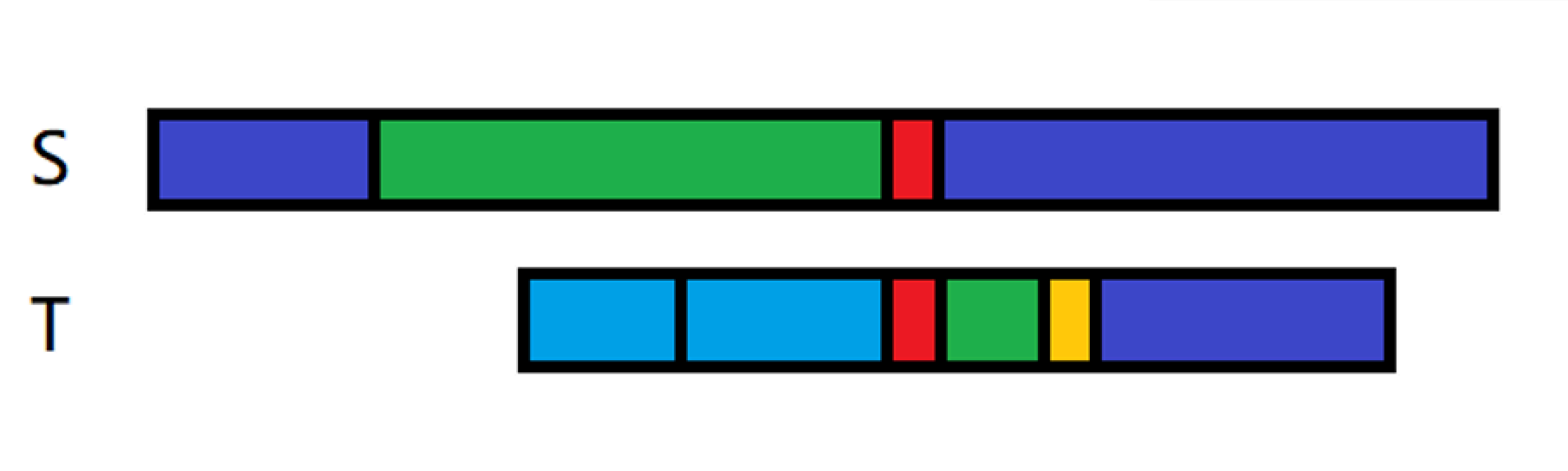
我们考虑在绿色部分中找到一个最长的前缀，也就是蓝色部分，满足蓝色部分和后半段部分一致。



换句话说蓝色部分是 绿色部分 的所有前缀中满足 前缀 = 后缀 的最长前缀。

KMP

接着我们直接挪动 T 串向前匹配：



容易发现，由于青色部分与后半段是一致的，因此到失配位置（红色）之前是都能匹配上的，只需要从红色向后匹配即可。

KMP 匹配证明

证明 KMP 的跳法不会漏掉任何匹配：

我们假设我们将 T 向前移动了 k 步，失配位置为 T 串的第 l 位，当前匹配到 S 串的第 i 位，那么有 $S[i, i+l-2] = T[1, l-1]$。

KMP 匹配证明

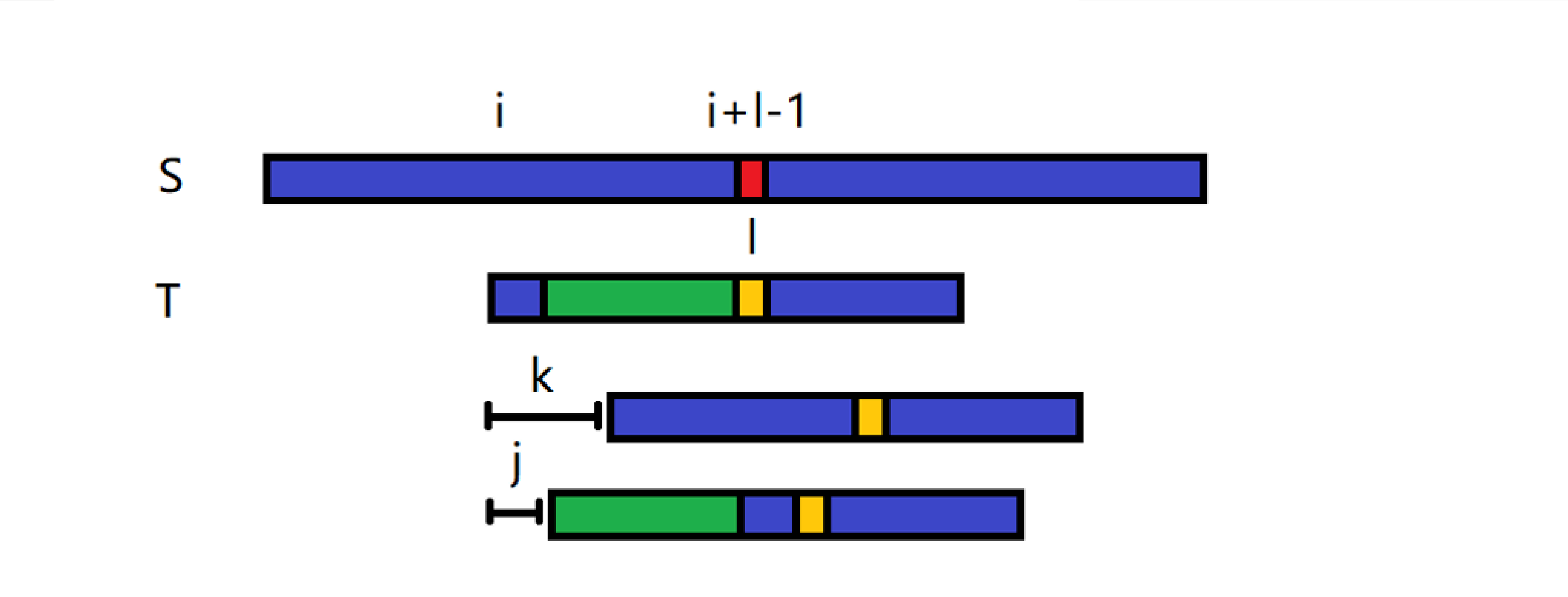
我们假设存在一个更小的 j，使得移动 j 步也能找到一个匹配，则有：$S[i+j, i+l-2] = T[1, l-1-j]$。

那么有 $T[1, l-1-j] = S[i+j, i+l-2] = T[j+1, l-1]$，有移动 k 对应的不是最长前缀，这与我们的假设矛盾。

同时，因为对于其他小于 k 步的移动方式，可以发现匹配位置不会超过 $i+l-2$，故证得 k 是最小的移动步数。

KMP 匹配证明

用图来表示大概是这样：由于绿色部分相同，那么移动 k 那一次对应的就不是最长前缀，这与假设矛盾



KMP 时间复杂度

分析这样做的时间复杂度：

每次操作，要么将匹配位置指针向前移动一位（在 S 串上前进一位），要么将 T 在 S 上匹配的起点向后移动了至少一位。

在 S 上最多前进 $|S|$ 位，T 在 S 上匹配的起点也只会向后移动 $|S|$ 位，所以匹配的时间复杂度为 $O(|S|)$。

KMP next/border 数组

现在，问题变成了如何高效地求得 T 串每个位置对应的 “最长前缀”，我们定义 nxt 表示对于 i 位置而言，“最长前缀”能匹配到 $\mathrm{nxt}[i]$，即 $T[1,\mathrm{nxt}[i]] = T[i-\mathrm{nxt}[i]+1,i]$，并且不存在 j 使得 $j > \mathrm{nxt}[i]$ 且有 $T[1,j] = T[i-j+1,i]$。此时有 $\mathrm{nxt}[i] < i$。

约定 $\mathrm{nxt}[0] = 1$。

直接求 “最长前缀” 效率是 $O(n^3)$ 的（枚举终点，枚举起点，暴力匹配），还不如直接暴力匹配两个字符串

KMP next/border 数组

那么假设我们已经有了 $\text{nxt}[1], \text{nxt}[2], \cdots, , \text{nxt}[k-1]$，需要求出 $\text{nxt}[k]$。

因为我们已经有了 $\text{nxt}[k-1]$，所以 $T[1, \text{nxt}[k-1]] = T[\text{nxt}[k-1], k-1]$

如果 $T[\text{nxt}[k-1]+1] = T[k]$，那么 $\text{nxt}[k] = \text{nxt}[k-1]+1$，这样 $\text{nxt}[k]$ 一定是最大的（由反证法可以证明）

如果不满足，有下面这样一张图：

容易发现浅蓝色部分是相同的，淡绿色部分也是相同的，并且，淡绿色部分也是满足“前缀与后缀相同”条件的，除去淡蓝色中最长的一个（同样用反证法可以证明）

KMP next/border 数组

于是我们有了这样一个思路：

对于位置 k，我们有一个指针 p，p 一开始是 $\mathrm{nxt}[k-1]$

若 $T[p+1]=T[k]$，则令 $\mathrm{nxt}[k]=p+1$

否则，令 $p=\mathrm{nxt}[p]$，再次执行上述操作，直到 $p=0$。

若直到 $p=0$ 仍未找到一个合法位置，那么只需要判断是否 $T[1]=T[k]$，若是则 $\mathrm{nxt}[k]=1$，否则 $\mathrm{nxt}[k]=0$。

KMP next/border 数组

分析求 nxt 的时间复杂度：

看起来这样做是 $O(n^2)$ 的

不妨分析一下求 nxt 的过程，是不是长得有点像 T 在匹配自己？ $\mathrm{nxt}[k] = \mathrm{nxt}[k-1]+1$ 相当于是指针向后移动一位；往回跳 p 相当于是向右移动 T。

上文分析过，匹配字符串的时间复杂度是 $O(\mathrm{len})$ 的，因此求 nxt 也是 $O(|T|)$ 的。

KMP 匹配

然后是匹配两个字符串的匹配算法，有了刚才的 nxt 数组，做起来很简单：

设置一个指针 p 表示当前匹配到了 T 串的 p 位置，设置指针 i 表示匹配到了 S 串的 i 位置

首先将 i 的初值设为 1（字符串下标从 1 开始），p 的初值设为 0，接着循环执行如下操作：

KMP 匹配

当 p 不等于 0 时，循环判断 $T[p+1]$ 是否等于 $S[i]$，若是则退出循环，否则将 p 设置为 $\mathrm{nxt}[p]$，再次判断；

之后，判断 $T[p+1]$ 是否等于 $S[i]$，若等于则令 p 自增 1；

若此时 $p = \mathrm{strlen}(T)$，记录匹配位置

KMP next 数组

```
void get_next(char *str,const int len)
{
    // nxt[i] 表示 [0, i-1] 的最长前缀
    nxt[0] = nxt[1] = 0;
    for (int i = 1; i < len; i++)
    {
        int p = nxt[i];
        while (p && str[i] != str[p]) // 失配
            p = nxt[p];
        nxt[i + 1] = (str[i] == str[p]) ? p + 1 : 0;
    }
}
```

KMP 匹配

```
void match(const int n, const int m) // n 为 S 的长度, m 为 T 的长度
{
    int p = 0;
    for (int i = 0; i < n; i++)
    {
        while (p && S[i] != T[p]) // 失配
            p = nxt[p];
        if (S[i] == T[p])
            p++;
        if (p == m)
            printf("%d\n", i - m + 2);
    }
}
```

谢谢！