后缀数组

芜湖一中　许智磊

后缀数组——字符串处理中的有力武器

后缀树的一个简单而高效的替代品

当今字符串处理研究中的热门

让我们一同揭开她神秘的面纱

后缀数组——定义和符号

字符集、字符、字符串都按照惯常的定义

字符串 S 的长度表示为 len(S)

字符串的下标从 1 开始到 len(S) 结束

字符串 S 的第 i 个字符表示为 S[i]

从 i 到 j 这一段的子串表示为 S[i..j]

后缀是一种特殊的子串

从某个位置 i 开始到整个串的末尾结束

S 的从 i 开头的后缀等价于 S[i..len(S)]

后缀数组——定义和符号

约定一个字符集 Σ

待处理的字符串约定为 S，约定 len(S)=n

规定 S 以字符 “ $” 结尾，即 S[n]=“$”

“$” 小于 Σ 中所有的字符

除了 S[n]=“$” 之外， S 的其他字符都属于 Σ

对于约定的字符串 S，其 i 开头的后缀表示为

Suffix(i)

后缀数组——定义和符号

字符串的大小关系 按照通常所说的"**字典顺序**"进行比较

我们对 S 的 n 个后缀按照字典顺序从小到大排序

将排序后的后缀的开头位置顺次放入数组 SA 中，称为

后缀数组

令 Rank[i] 保存 Suffix(i) 在排序中的名次，称数组 Rank 为

名次数组

后缀数组——构造方法

如何构造后缀数组？

把 n 个后缀当作 n 个字符串，按照普通的方法进行排序 —— $O(n^2)$

低效的原因 —— **把后缀仅仅当作普通的、独立的字符串，忽略了后缀之间存在的有机联系。**

后缀数组——构造方法

对字符串 u，定义 $u^k = \begin{cases} u[1..k] & ,len(u) \ge k \\ u & ,len(u) < k \end{cases}$

倍增算法(Doubling Algorithm)

定义 k- 前缀比较关系 $<_k$，$=_k$ 和 $\le_k$

对两个字符串 u,v，

$u <_k v$ 当且仅当 $u^k < v^k$

$u =_k v$ 当且仅当 $u^k = v^k$

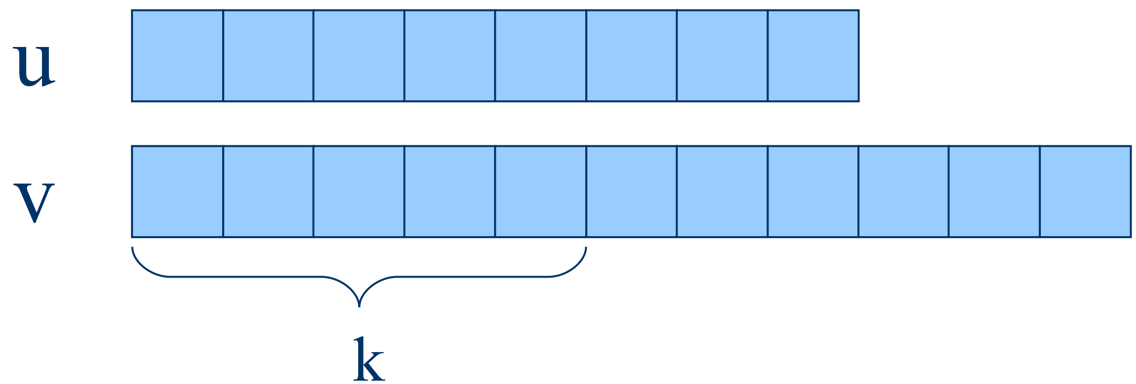
$u \le_k v$ 当且仅当 $u^k \le v^k$

后缀数组——构造方法

$u <_k v$?

$u =_k v$?

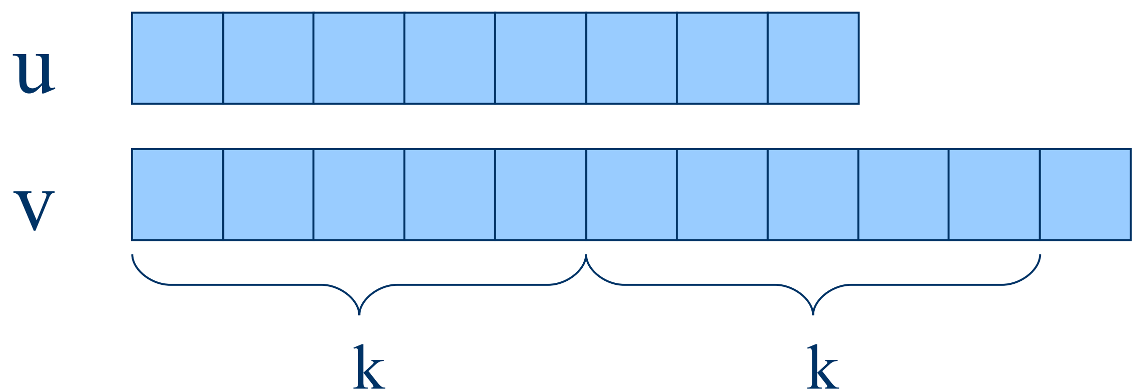
$u \le_k v$?



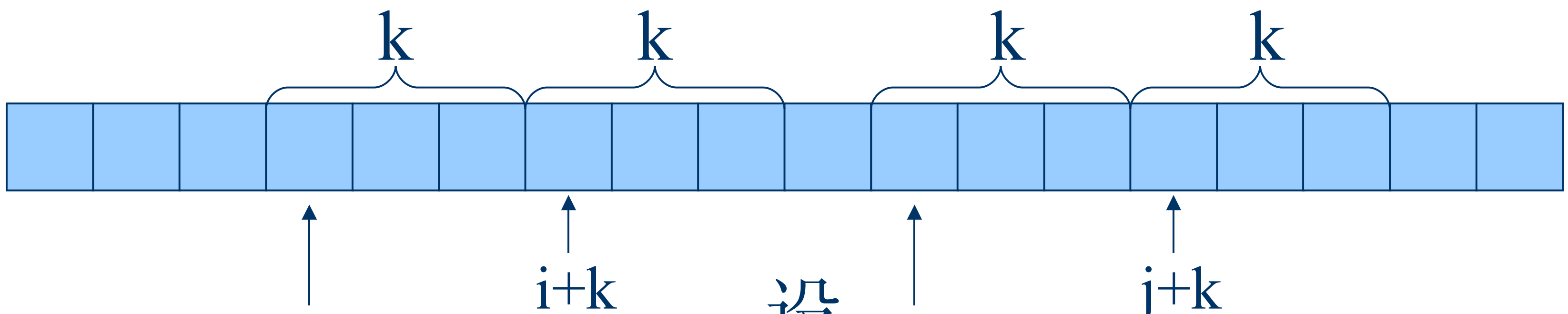
$u <_{2k} v$?

$u =_{2k} v$?

$u \le_{2k} v$?



后缀数组——构造方法



后缀 u，以 i 开头

后缀 v，以 j 开头

设 u=Suffix(i)， v=Suffix(j)

比较红色字符相当于在 k- 前缀意义下比较

在 2k- 前缀意义下比较两个后缀可以转化成

对 u、v 在 2(k)- 前缀意义下比较

Suffix(i) 和 Suffix(j) 进行比较

在 k- 前缀意义下比较两个后缀

比较绿色字符相当于在 k- 前缀意义下比较

Suffix(i+k) 和 Suffix(j+k)

后缀数组——构造方法

把 n 个后缀按照 k- 前缀意义下的大小关系从小到大排序

将排序后的后缀的开头位置顺次放入数组 SA_k 中，称为

k- 后缀数组

用 $Rank_k[i]$ 保存 Suffix(i) 在排序中的名次，称数组 $Rank_k$ 为

k- 名次数组

后缀数组——构造方法

利用 SA_k 可以在 O(n) 时间内求出 $Rank_k$

利用 $Rank_k$ 可以在常数时间内对两个后缀进行 k- 前缀意义下的大小比较

后缀数组——构造方法

如果已经求出 $Rank_k$

⇨ 可以在常数时间内对两个后缀进行 k- 前缀意义下的比较

⇨ 可以在常数时间内对两个后缀进行 2k- 前缀意义下的比较

⇨ 可以很方便地对所有的后缀在 2k- 前缀意义下排序

- 采用快速排序 O(nlogn)
- 采用基数排序 O(n)

➡ 可以在 O(n) 时间内由 $Rank_k$ 求出 SA_{2k}

➡ 也就可以在 O(n) 时间内求出 $Rank_{2k}$

后缀数组——构造方法

1- 前缀比较关系实际上是对字符串的第一个字符进行比较

可以直接根据开头字符对所有后缀进行排序求出 SA_1

- 采用快速排序，复杂度为 O(nlogn)

然后根据 SA_1 在 O(n) 时间内求出 $Rank_1$

可以在 O(nlogn) 时间内求出 SA_1 和 $Rank_1$

后缀数组——构造方法

直接根据首字符排序 $\xrightarrow{O(n\log n)}$ SA_1 $Rank_1$ $\xrightarrow{O(n)}$ SA_2 $Rank_2$ $\xrightarrow{O(n)}$ SA_4 $Rank_4$ $\xrightarrow{O(n)}$ SA_8 $Rank_8$ $\xrightarrow{O(n)}$ …… $\xrightarrow{O(n)}$ SA_m $Rank_m$

$m=2^t$ 且 $m \geq n$

O(nlogn) 的操作一次

O(n) 的操作 $\lceil \log n \rceil$ 次

总复杂度为

O(nlogn)

后缀数组——构造方法

当 m≥n 时，对任意 u=Suffix(x)， $u^m=u$

$\text{Suffix}(i)\le_m \text{Suffix}(j) \Leftrightarrow \text{Suffix}(i)\le \text{Suffix}(j)$

$\text{Suffix}(i)<\text{Suffix}(j)$

O(nlogn) 求出

SA_m 和 $Rank_m$

$$\mathbf{SA_m=SA}$$

$$\mathbf{Rank_m=Rank}$$

可以在 O(nlogn) 时间内求出后缀数组 SA 和名次数组 Rank

后缀数组——构造方法

$m \geq n$，

$SA_m = SA$

$Rank_m = Rank$

我们已经在 O(nlogn) 的时间内构造出了

后缀数组 SA 和 名次数组 Rank

后缀数组——方法总结

利用到后缀之间的联系

用 k- 前缀比较关系来表达 2k- 前缀比较关系

每次可以将参与比较的前缀长度加倍

根据 SA_k 、 $Rank_k$ 求出 SA_{2k} 、 $Rank_{2k}$

参与比较的前缀长度达到 n 以上时结束

倍增思想

后缀数组——辅助工具

仅仅靠后缀数组和名次数组

有时候还不能很好地处理问题

后缀数组的最佳搭档—— LCP

定义两个字符串的最长公共前缀 Longest Common Prefix

$$lcp(u,v)=\max\{i|u=_i v\}$$

也就是从头开始比较 u 和 v 的对应字符持续相等的最远值

后缀数组——辅助工具

定义 LCP(i,j)=lcp(Suffix(SA[i]),Suffix(SA[j]))

也就是 SA 数组中第 i 个和第 j 个后缀的最长公共前缀

可以用跨度为 1 的 LCP 值来表示任何一个 LCP 值

LCP Theorem

若 i>j

LCP(i,j)=LCP(j,i)

对任何 $1 \le i < j \le n$

$$LCP(i,j)=\min\{LCP(k-1,k) \mid i+1 \le k \le j\}$$

称 j-i 为 LCP(i,j) 的“**跨度**”， LCP Theorem 意义为：

跨度大于 1 的 LCP 值可以表示成一段跨度等于 1 的 LCP 值的**最小值**

后缀数组——辅助工具

定义 $LCP(i,j)=lcp(Suffix(SA[i]),Suffix(SA[j]))$

也就是 SA 数组中第 i 个和第 j 个后缀的最长公共前缀

LCP Theorem

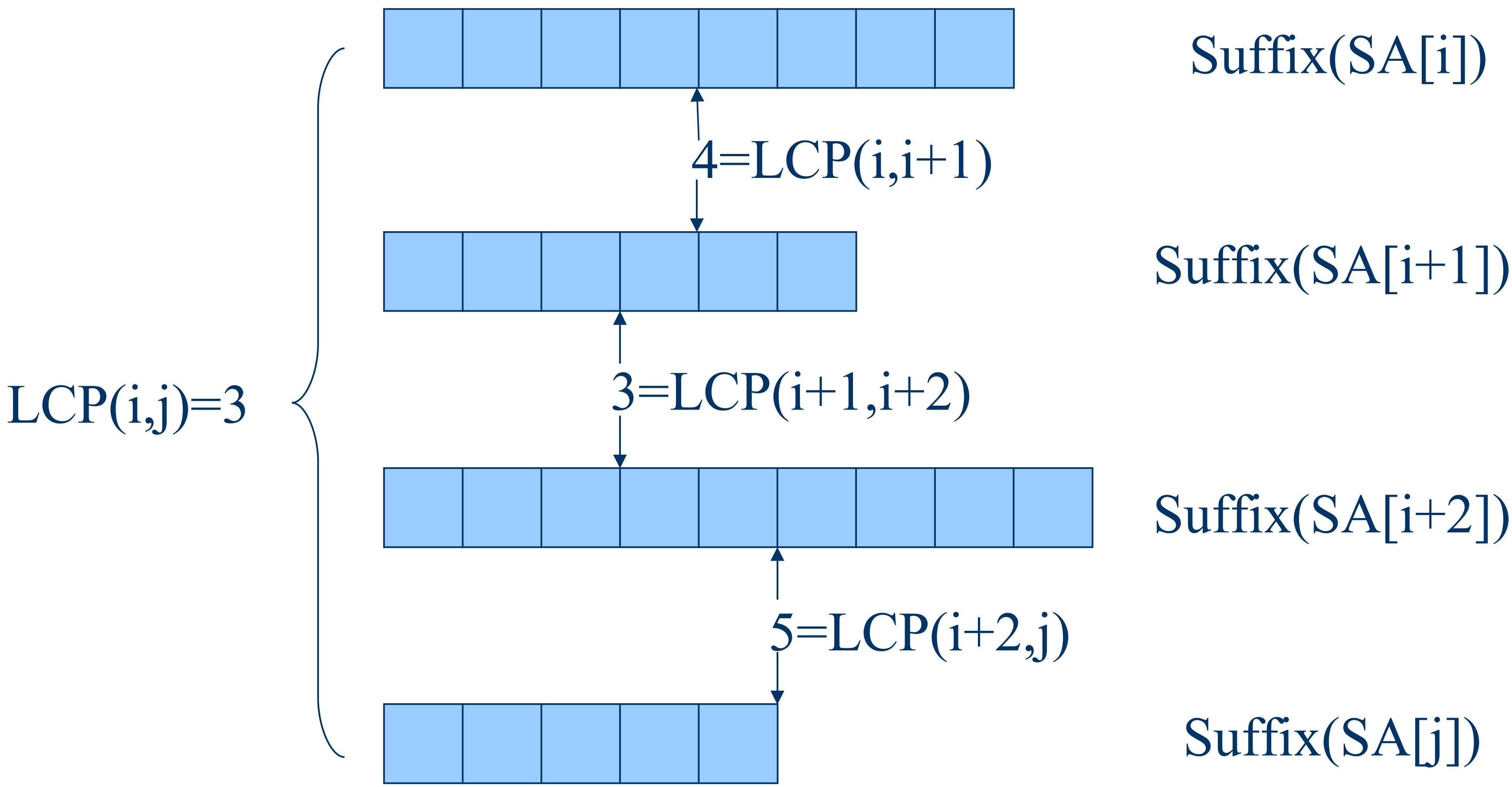
对任何 $1\le i<j\le n$

$$LCP(i,j)=\min\{LCP(k-1,k) \mid i+1\le k\le j\}$$

称 j-i 为 LCP(i,j) 的“**跨度**”， LCP Theorem 意义为：

跨度大于 1 的 LCP 值可以表示成一段跨度等于 1 的 LCP 值的**最小值**

后缀数组——辅助工具


Suffix(SA[i])
4=LCP(i,i+1)
Suffix(SA[i+1])
LCP(i,j)=3
3=LCP(i+1,i+2)
Suffix(SA[i+2])
5=LCP(i+2,j)
Suffix(SA[j])

后缀数组——辅助工具

设 height[i]=LCP(i-1,i)

根据 LCP Theorem

LCP(i,j)=min{height[k] | i+1≤k≤j}

计算 LCP(i,j) 等价于

询问数组 height 中下标从 i+1 到 j 范围内所有元素的**最小值**

经典的 RMQ (Range Minimum Query) 问题 !!!

- 线段树、排序树 —— O(nlogn) 预处理 , O(logn) 每次询问

后缀数组——辅助工具

采用一种“神奇的”方法，可以在 O(n) 时间内计算出 height 数组

采用标准 RMQ 方法在 O(n) 时间内进行预处理

之后就可以在常数时间内算出任何的 LCP(i,j)

根据 lcp(Suffix(i),Suffix(j))=LCP(Rank[i],Rank[j])

可以在**常数时间**内计算出

任何两个后缀的**最长公共前缀**

后缀数组——辅助工具

采用一种“神奇的”方法，可以在 O(n) 时间内计算出 height 数组

采用标准 RMQ 方法在 O(n) 时间内进行预处理

之后就可以在常数时间内算出任何的 LCP(i,j)

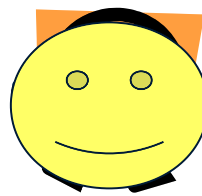
这是后缀数组

可以在**常数时间**内计算出

最常用以及最强大的功能之一

任何两个后缀的**最长公共前缀**

后缀数组——应用举例

几个**的问题**

$O(n^2)$ $O(n\log n)$ 问题一　给定一个字符串 S，对它的所有后缀进行排序。

$O(m+n)$ $O(m\log n)$ 问题二　给定一个待匹配串 S，每次输入一个模式串 P，要求返回 P 在 S 中的一个匹配的开头位置，或者返回无匹配。

$O(n^2)$ $O(n\log n)$ 问题三　给定一个字符串 S，求出 S 中的最长回文子串。

后缀数组——应用举例

怎样使用后缀数组?

后缀数组——应用举例

回文串——顺读和倒读完全一样的字符串

回文串

奇回文串　字符串 u 满足：

- $len(u)=p$ 为奇数
- 对任何 $1 \le i \le (p-1)/2$， $u[i]=u[p-i+1]$

偶回文串　字符串 v 满足：

- $len(v)=q$ 为奇数
- 对任何 $1 \le i \le q/2$， $v[i]=v[q-i+1]$

后缀数组——应用举例

字符串 T 的**回文子串**——T 的**子串**，并且是**回文串**

字符串 T 的**最长回文子串**——T 的**回文子串**中长度**最大**的

给出一个字符串 T，求它的最长回文子串

给出最大长度即可

后缀数组——应用举例

分析求最长奇回文子串的算法

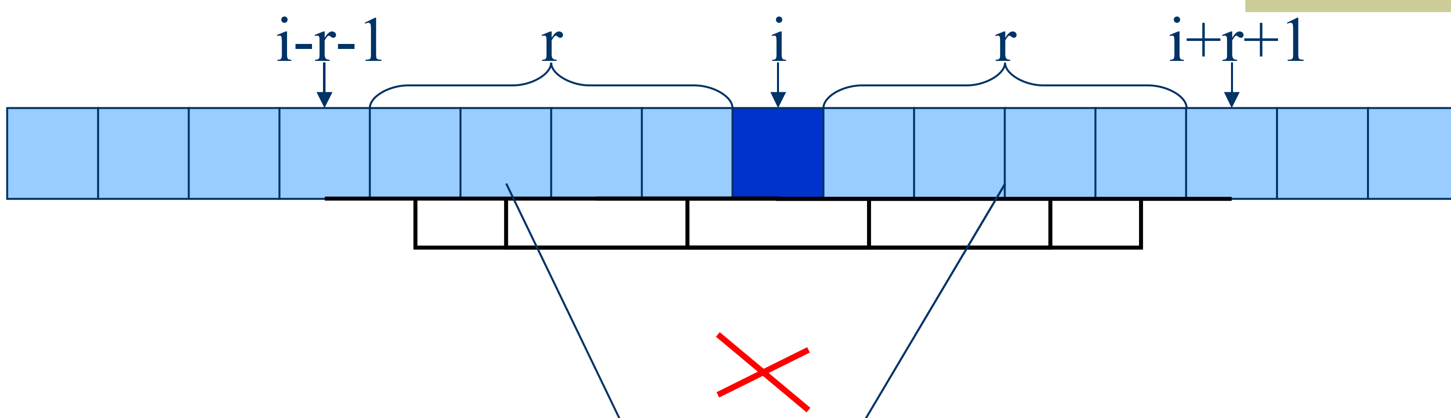
最长偶回文子串可以类似求出

后缀数组——应用举例

枚举奇回文串中间一个字符的位置

尽量向两边扩展

后缀数组——应用举例



以某个位置为中心向两边扩展的复杂度为

$O(len(T))$

反射相等

整个算法的复杂度为

$O(len(T)^2)$

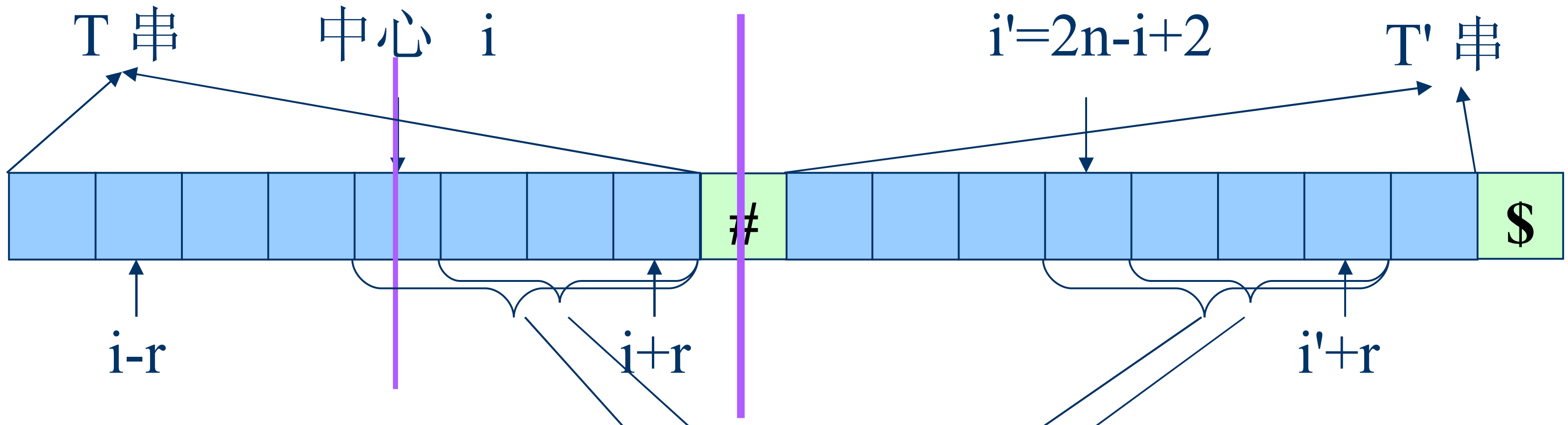
后缀数组——应用举例

求以一个位置 i 为中心向两边扩展的最远值

是算法的核心部分

需要降低这一步的复杂度

后缀数组——应用举例



求以 i 为中心向两边扩展的最远值，等价于

求 Suffix(i) 和 Suffix(i') 的最长公共前缀

Suffix(i) 和 Suffix(i') 的公共前缀

后缀数组！！！

后缀数组——应用举例

解法：

1. 初始化答案为 0 。按照前述方法修改串 T ，得到串 S
1. 求出后缀数组 SA 、名次数组 Rank
1. 计算 height 数组并进行标准 RMQ 方法预处理
1. 枚举 i ，计算以 i 为对称中心的极长回文串并更新答案

复杂度：

$$O(n) + O(n\log n) + 2*O(n) + n*O(1) = \mathbf{O(n\log n)}$$

后缀数组 VS 后缀树

后缀 事情

后缀 呢?

后缀数组 VS 后缀树

后缀数组在信息学竞赛中最大的优势：

易于理解，易于编程，易于调试

后缀数组比后缀树占用的空间少

—— 处理长字符串，如 DNA 分析

后缀数组 VS 后缀树

时间复杂度的比较

按照字符总数 |Σ| 把字符集 Σ 分为三种类型：

Constant Alphabet —— |Σ| 是一个常数

Integer Alphabet —— |Σ| 和字符串长度 **n** 规模相当

General Alphabet —— 对 |Σ| 没有任何限制

Constant Alphabet ⊂ **Integer** Alphabet ⊂ **General** Alphabet

后缀数组 VS 后缀树

结论

对于 **Integer** 和 **General** 以及 **|Σ|** 较大的 **Constant** Alphabet， 后缀树甚至在时间复杂度上都无法胜过后缀数组。

但是对于 **|Σ|** 较小的 **Constant** Alphabet，

后缀树还是有着速度上的优势的。

——我们要根据实际情况，因“题”制宜选择合适的数据结构

后缀数组——最后的话

研究后缀数组，不是因为害怕后缀树的繁琐

也没有贬低后缀树，抬高后缀数组的意思

对于功能相似的两个数据结构，我们应该灵活地掌握，有比较有选择地使用

构造后缀数组用到的倍增思想对我们的思考也是有帮助的

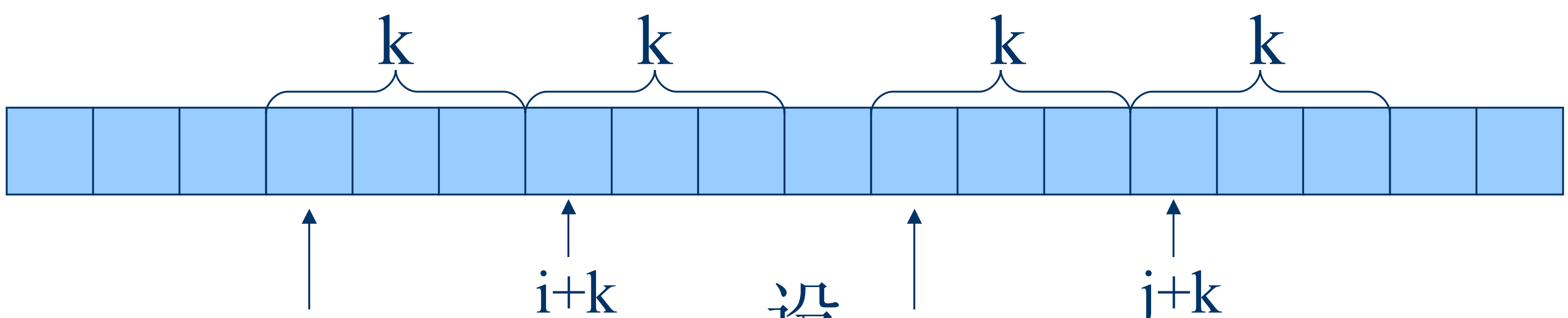
后缀数组

谢谢大家！

后缀数组——关于“$”

为什么规定S以“$”结尾?

后缀数组——关于“$”



后缀u，以i开头

后缀v，以j开头

设 u=Suffix(i)， v=Suffix(j)

比较红色字符相当于在k-前缀意义下比较

在2k-前缀意义下比较两个后缀可以转化成

Suffix(i) 和 Suffix(j)

对u、v在2k-前缀意义下进行比较

在k-前缀意义下比较两个后缀

比较绿色字符相当于在k-前缀意义下比较

Suffix(i+k) 和 Suffix(j+k)

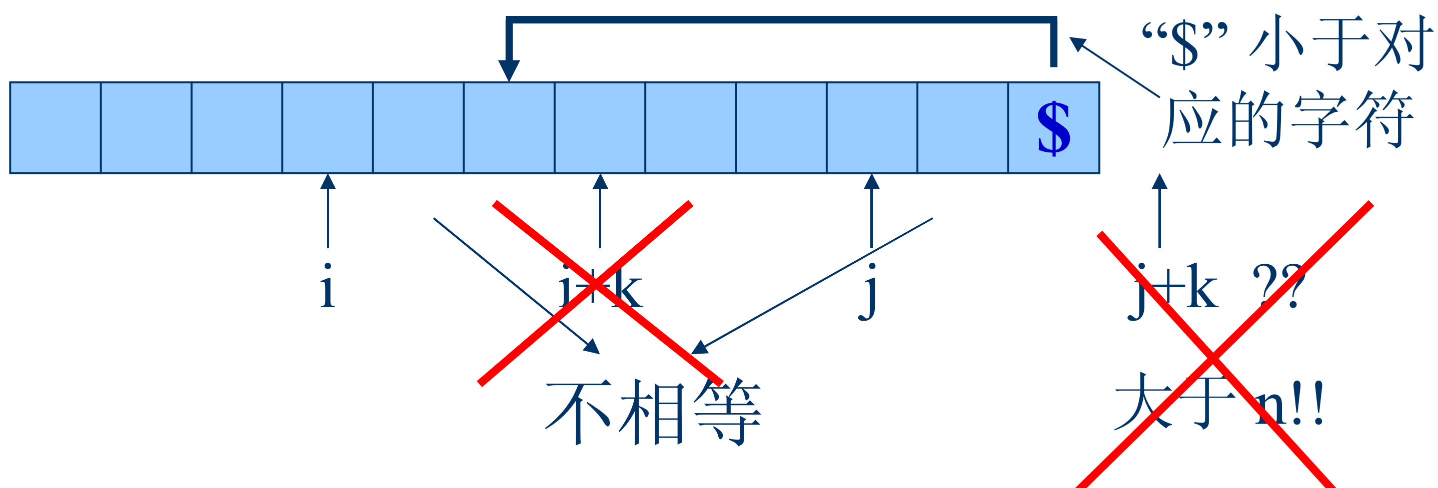
后缀数组——关于“$”

2k- 前缀比较关系可以用两个 k- 前缀比较关系来表达

$$u<_{2k}v \quad \Leftrightarrow \quad u<_{k}v \text{ OR } (u=_{k}v \text{ AND } \mathrm{Suffix}(i+k)<_{k}\mathrm{Suffix}(v,j+k))$$

$$u=_{2k}v \quad \Leftrightarrow \quad u=_{k}v \text{ AND } \mathrm{Suffix}(i+k)=_{k}\mathrm{Suffix}(j+k)$$

$$u\leq_{2k}v \quad \Leftrightarrow \quad u<_{k}v \text{ OR } (u=_{k}v \text{ AND } \mathrm{Suffix}(i+k)\leq_{k}\mathrm{Suffix}(j+k))$$



后缀数组——关于“ $”

结尾的“ $”避免了下标越界造成无意义表达式的麻烦

为什么规定“ $”小于Σ中的任何字符?

规定“ $”不等于Σ中的任何字符可以达到同样的目的?

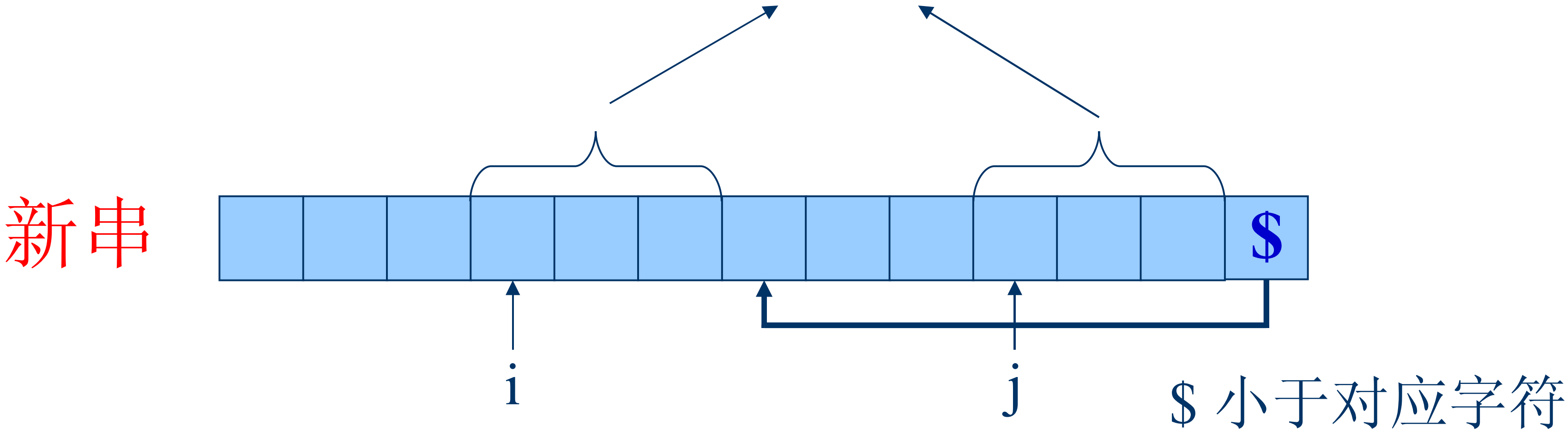
后缀数组——关于“$”

仍然能得到

i 开头的后缀 < j 开头的后缀

i 开头的后缀 < j 开头的后缀

相等



后缀数组——关于“$”

规定“$”小于 Σ 中的任何字符是为了保证

在串 S 结尾添加“$”改造为 S' 之后，

S 中的后缀之间的大小关系在 S' 中依然成立。

于是 S' 的后缀数组、名次数组都和 S 的一样。

另外不难看出

S' 的 **height** 数组和 S 的也是一样的。

后缀数组——关于“$”

在待处理的串后添加“$”

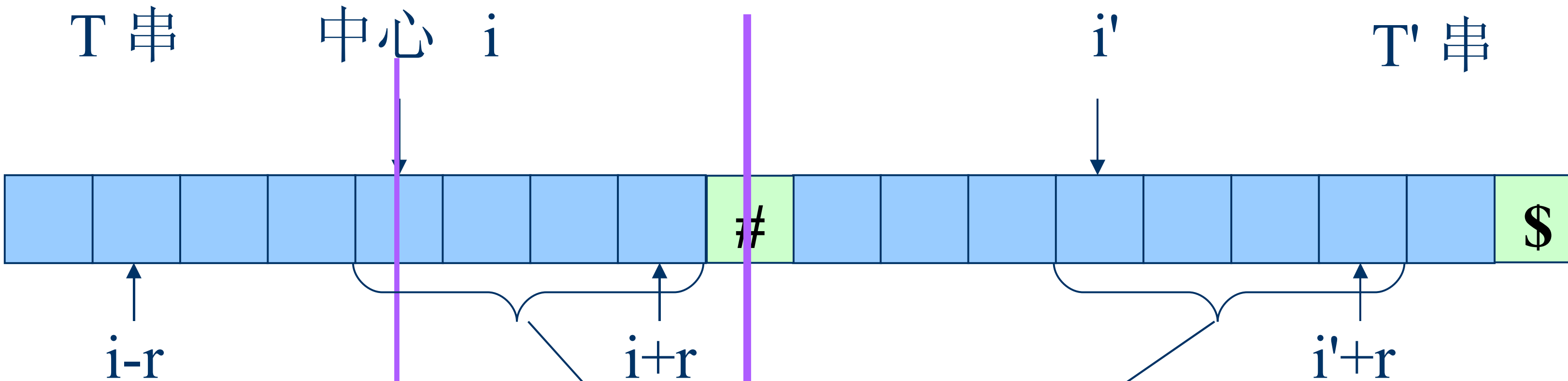
不会影响结果的正确性，

只是令操作变得方便。

后缀数组——关于“#”

为什么要先在 T 串后加“#”然后再反射 T 串？

后缀数组——关于“#”



Suffix(i) 和 Suffix(i') 的最长公共前缀

不再能准确地反映向两边扩展的最远值

Suffix(i) 和 Suffix(i') 的公共前缀

因为左边的浅绿色串用到了 T' 中的字符

这是不合实际的

后缀数组——关于“#”

在 T 的结尾加上“#”保证了

Suffix(i) 和 Suffix(i') 的最长公共前缀能正确反映

以 i 为中心向两边扩展的最远值

特殊判断也可以做到这一点，

但是加一个“#”稍微方便一些。

后缀数组——关于线性算法

采用 Farach 的构造方法，对于 **Integer** Alphbet，

可以在 O(n) 时间内构造出后缀树

我们不打算把 Farach 方法列入考察范围

后缀数组——关于线性算法

这个算法实现极为繁琐

更像是挖空心思将几个算法凑在一起的“大杂烩”

竞赛的有限时间内几乎无法完成

后缀数组——关于线性算法

即使构造完成了，如果想使用后缀树，

还是得想办法处理每个节点指向儿子的指针。

对字符总数太大的情况只能排序存储

时间复杂度立刻增加到 $O(n\log|\Sigma|)$

并没有得到改善，也未必比后缀数组快

后缀数组——关于线性算法

访问的时候要二分查找指向儿子的指针

几乎所有的基本操作复杂度都要乘上系数 $\log|\Sigma|$

某些情况下甚至比后缀数组差，如多模式串匹配

后缀数组——关于线性算法

只有极少数情况下后缀树才能真正做到线性时间构造，常数时间基本操作

Farach 构造方法的理论价值大于它在竞赛中的实际价值

不适合拿来和本文所讲的后缀数组相比

后缀数组——关于线性算法

更令人吃惊的是

后缀数组也有对 **Integer Alphbet** 情况下线性构造的算法

这是一种三分法，比 Farach 方法优美得多

但是我们也无意在本文中探讨

后缀数组——关于线性算法

虽然说它比 Farach 方法优美，

但是实现起来仍然比倍增算法复杂得多

三分法的技巧性显得较强

与技巧相比

不适合在竞赛中使用

我更加欣赏倍增算法所体现出的深刻思想

有兴趣的同学可以自行研究

后缀数组——关于空间

从 $Rank_k$ 推出 SA_k 和 $Rank_{2k}$

需要两个数组（共 2n 个整数）以实现基数排序

同一时刻 $Rank_k$ 和 $Rank_{2k}$ 只需要保存一个

SA_{2k} 可以直接覆盖 SA_k

2n 个整数保存结果 ＋2n 个整数辅助计算

技巧性地操作可以将辅助计算的空间减少至 n 个整数

后缀数组——关于空间

后缀树通常有 2n 个以上节点

通常每个节点要两个整数，至少要保存一个整数

每个节点两个指针

4n 个指针 + 2n 个整数

至少是 4n 个指针 + n 个整数

后缀数组——关于空间

为什么不算上 height 数组和 RMQ 预处理的空间
？

—— 为了处理问题，后缀树需要预处理以便计算节点的最近公共祖先（ Least Common Ancestor ）

LCA 问题和 RMQ 问题是等价的

后缀数组——关于 RMQ

RMQ 问题的标准解法是怎么做的？

—— 同样用到了倍增思想

对每个位置 i 记录从开始向后 1,2,4,8...

长度的一段中的最小值

总共有 nlogn 个值，通过动态规划计算

后缀数组——关于 RMQ

采用对待询问数组建立 Treap 的方法转化为 0-1 RMQ

采用模板方法将复杂度降为线性

竞赛中用 O(nlogn) 预处理的方法已经足够

后缀数组——关于倍增思想

倍增思想，本质上是一种特殊的

动态规划思想

与一般的动态规划不同的是，

它划分阶段是按照规模的对数来分

也就是先处理规模为 2^0 的问题

然后顺次推出规模为 2^1， 2^2， 2^3， … 的问题

后缀数组——关于倍增思想

关键在于找到 2^k 到 2^{k+1} 转换的**桥梁**

本文中的桥梁就是

2k- 前缀比较关系可以转化为 k- 前缀比较关系

后缀数组——关于倍增思想

知易行难

要用好用活倍增思想不是那么简单的事情

难点也就在于寻找转化的桥梁

后缀数组——关于倍增思想

《道德经》云：

道生一，一生二，二生三，三生万物

是否能用好倍增思想，做到：

一生二，二生四，四生八，……

要看各人的道行如何了

后缀数组——关于倍增思想

如果能够用好倍增思想

虽然不见得能化生万物

但是相信能够在很多情况下帮助你

独辟蹊径，解决规模巨大的题目

举重若轻

挥洒自如

后缀数组

谢谢大家！