
PSO 13
(Suffix) Tries, Forward Pattern Matching

(Trie & lexicographic sort)

Given two bit strings $a = a_0 a_1 \ldots a_p$ and $b = b_0 b_1 \ldots b_q$, we assume WLOG that $p \leq q$. Recall that a is said to be **lexicographically less** than b if one of the following happens:

- there exists an integer $j \leq p$ such that $a_i = b_i$ for all $0 \leq i < j$ and $a_j < b_j$.
- $p < q$ and $a_i = b_i$ for all $0 \leq i \leq p$.

Given a set S of distinct bit strings whose lengths sum to n, show how to use a radix tree (a.k.a. trie for bit strings) to sort S lexicographically in $O(n)$ time. For example, if $S = \{1011, 10, 011, 100, 0\}$, then the output should be the sequence 0, 011, 10, 100, 1011.

Question 2

(Suffix Tries)

(1) Draw the suffix tree for the string $s = \texttt{abracadabra}$.

(2) What is the longest substring that appears more than once in the string $s = \texttt{abracadabra}$?

(3) How can we generally find the longest substring appearing more than once given a suffix trie?

Question 3

(Forward pattern matching)

Another efficient pattern matching algorithm, named the Knuth-Morris-Pratt (KMP) algorithm, is based upon forward pattern matching, in which a failure function (also named as "suffix function") is calculated to determine the most distance we can shift the pattern to avoid redundant comparisons. Specifically, for a pattern P, its corresponding failure function $F_P(j)$, or $F(j)$ for short, is defined as

$$F(j) := \max_k \{k \leq j-1 : P[0:k] = P[j-k:j]\}.$$

In other words, $F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

Answer the following questions:

(1) Apply the KMP algorithm to the Boyer-Moore pattern matching problem in previous PSO, where

$$T := \underbrace{aaaaaaaaa}_{9} \quad \text{and} \quad P := baaaaa.$$

1. Does it perform much better than Boyer-Moore?

(2) What is the failure function for the pattern $P :=$ "mamagama"?

(3) Let $T :=$ "rahrahahahahromaromamagagaoohlala", run the KMP pattern matching algorithm for the pattern P in (2).

(Trie & lexicographic sort)

Given two bit strings $a = a_0 a_1 \dots a_p$ and $b = b_0 b_1 \dots b_q$, we assume WLOG that $p \le q$. Recall that a is said to be **lexicographically less** than b if one of the following happens:

- there exists an integer $j \le p$ such that $a_i = b_i$ for all $0 \le i < j$ and $a_j < b_j$.
- $p < q$ and $a_i = b_i$ for all $0 \le i \le p$.

Given a set S of distinct bit strings whose lengths sum to n, show how to use a radix tree (a.k.a. trie for bit strings) to sort S lexicographically in $O(n)$ time. For example, if $S = \{1011, 10, 011, 100, 0\}$, then the output should be the sequence 0, 011, 10, 100, 1011.

n=13

Lexigraphic Ordering Practice:

- 'c' vs 'ab' >
- 'abc' vs 'abca' <
- 'abbbbb' vs 'baaaaa' <

trie.insert

BST insert but with letters = {0,1}

(Trie & lexicographic sort)

Given two bit strings $a = a_0 a_1 \ldots a_p$ and $b = b_0 b_1 \ldots b_q$, we assume WLOG that $p \leq q$. Recall that a is said to be **lexicographically less** than b if one of the following happens:

- there exists an integer $j \leq p$ such that $a_i = b_i$ for all $0 \leq i < j$ and $a_j < b_j$.
- $p < q$ and $a_i = b_i$ for all $0 \leq i \leq p$.

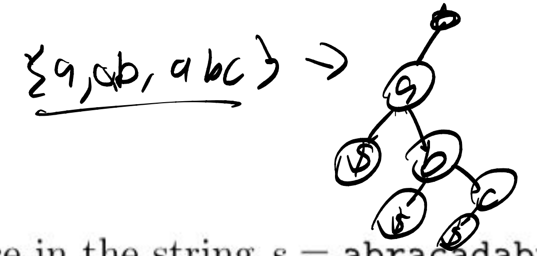
Given a set S of distinct bit strings whose lengths sum to n, show how to use a radix tree (a.k.a. trie for bit strings) to sort S lexicographically in $O(n)$ time. For example, if $S = \{1011, 10, 011, 100, 0\}$, then the output should be the sequence 0, 011, 10, 100, 1011.

S sorted = {0, 011, 10, 100, 1011}

Form the trie for S

first trie starts out as a null root

Question 2



(Suffix Tries)

(1) Draw the suffix tree for the string $s =$ `abracadabra`.

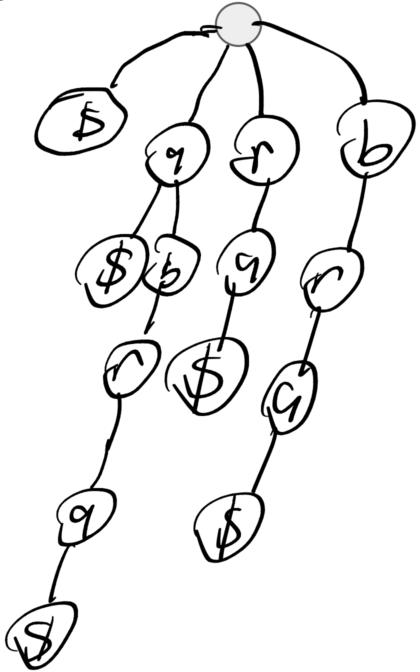
(2) What is the longest substring that appears more than once in the string $s =$ `abracadabra`?

(3) How can we generally find the longest substring appearing more than once given a suffix trie?

Drawing a suffix tree:

1. List out the suffixes, S = {a, ra, bra, abra, ...}
2. Use S to populate the trie

S = {$,a,ra,bra,abra,dabra,adabra,adabra,cadabra,acadabra,racadabra,bracadabra,abracadabra}



abracadabra

(2) What is the longest substring that appears more than once in the string s = abracadabra?

(3) How can we generally find the longest substring appearing more than once given a suffix trie?

Question 3

(Forward pattern matching)

Another efficient pattern matching algorithm, named the Knuth-Morris-Pratt (KMP) algorithm, is based upon forward pattern matching, in which a failure function (also named as "suffix function") is calculated to determine the most distance we can shift the pattern to avoid redundant comparisons. Specifically, for a pattern P, its corresponding failure function $F_P(j)$, or $F(j)$ for short, is defined as

$$F(j) := \max_k \{k \leq j-1 : P[0:k] = P[j-k:j]\}.$$

In other words, $F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

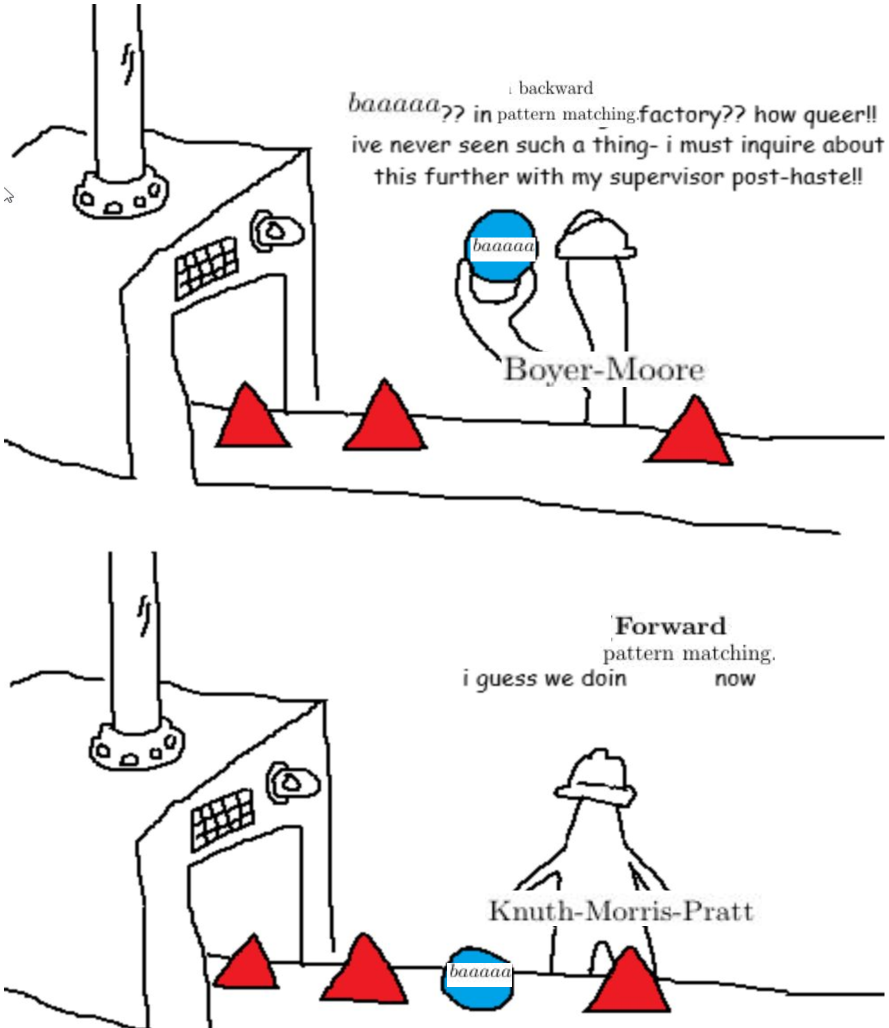
Answer the following questions:

(1) Apply the KMP algorithm to the Boyer-Moore pattern matching problem in previous PSO, where

T	a	a	a	a	a	a	a	a	a
P	b	a	a	a	a	a			
	a	a	a	a	a	b			

Boyer Moore did 24 comparisons (worse than forward brute force)!

forward: 6 comparisons


backward
baaaaa?? in pattern matching factory?? how queer!! ive never seen such a thing- i must inquire about this further with my supervisor post-haste!!
baaaaa
Boyer-Moore
Forward
i guess we doin pattern matching. now
Knuth-Morris-Pratt
baaaaa

P	b	a	a	a	a	a			

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

$$F(j) := \max_k \{k \leq j-1 : P[0:k] = P[j-k:j]\}.$$

Failure function

j	1	2	3	4	5	6	7
f(j)							

P	b	a	a	a	a	a			

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

$$F(j) := \max_{k} \{k \leq j-1 : P[0:k] = P[j-k:j]\}.$$

Failure function

j	1	2	3	4	5	6	7
f(j)							

P								
b	a	a	a	a	a			

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

$$F(j) := \max_k \{k \leq j-1 : P[0:k] = P[j-k:j]\}.$$

Failure function

j	1	**2**	3	4	5	6	7
f(j)	0						

P	b	a	a	a	a	a			

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

$$F(j) := \max_k \{k \leq j-1 : P[0:k] = P[j-k:j]\}.$$

Failure function

j	1	**2**	3	4	5	6	7
f(j)	0	0					

P								
b	a	a	a	a	a			

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

$$F(j) := \max_k \{k \leq j-1 : P[0:k] = P[j-k:j]\}.$$

Failure function

j	1	**2**	3	4	5	6	7
f(j)	0	0	0	0	0	0	0

Why all 0s?

(2) What is the failure function for the pattern $P :=$ "mamagama"?

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

P[0:1] P[1:1]

m a m a g a m a

(j=1)

j	1	2	3	4	5	6	7
f(j)	0						

(2) What is the failure function for the pattern $P :=$ "mamagama"?

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

P:[0:2] P[1:2]

m a m a g a m a

j	1	2	3	4	5	6	7
f(j)	0	1					

(2) What is the failure function for the pattern $P :=$ "mamagama"?

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

m a m a g a m a

j	1	2	3	4	5	6	7
f(j)	0	1	2				

(2) What is the failure function for the pattern $P :=$ "mamagama"?

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

m a m a g a m a

j	1	2	3	**4**	5	6	7
f(j)	0	1	2	0			

(2) What is the failure function for the pattern $P :=$ "mamagama"?

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

m a m a g a m a

j	1	2	3	4	**5**	6	7
f(j)	0	1	2	0			

(2) What is the failure function for the pattern $P :=$ "mamagama"?

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

m a m a g a m a

j	1	2	3	4	5	**6**	7
f(j)	0	1	2	0	0	1	

(2) What is the failure function for the pattern $P :=$ "mamagama"?

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

m a m a g a m a

j	1	2	3	4	5	6	**7**
f(j)	0	1	2	0	0	1	2

(2) What is the failure function for the pattern $P :=$ "mamagama"?

$F(j)$ represents the size of the largest prefix of $P[0:j]$ that is also a suffix of $P[1:j]$.

m a m a g a m a

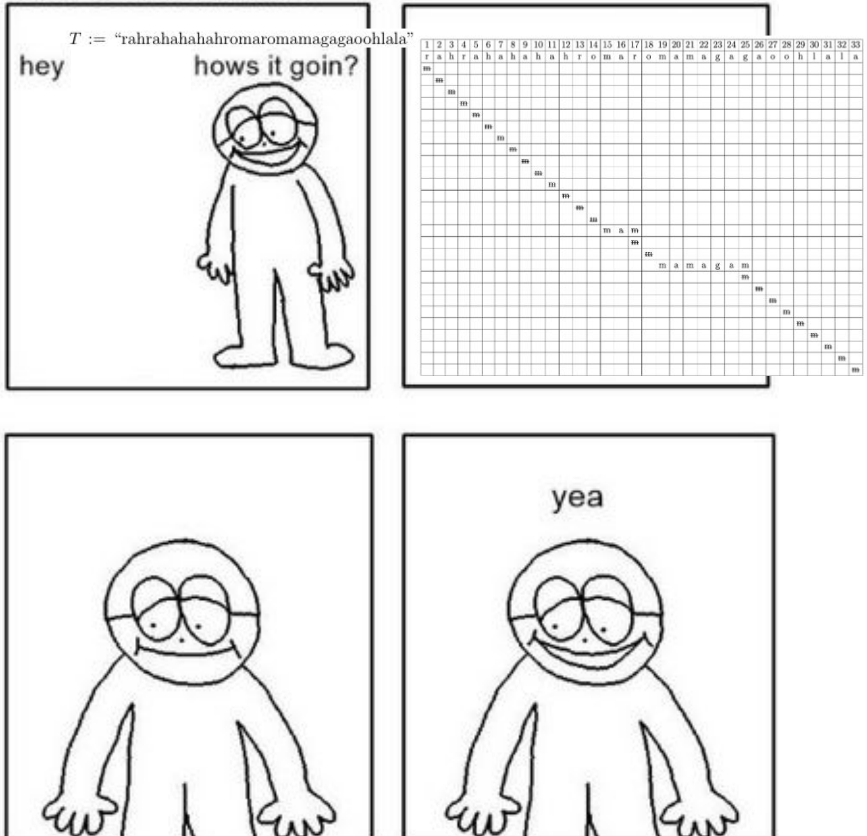
j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

(3) Let $T :=$ "rahrahahahahromaromamagagaoohlala", run the KMP pattern matching algorithm for the pattern P in (2).

This example is a bit long..



In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

T[i]

Let T = “rahmamamamamagama”

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

(i = 0)

T	**r**	a	h	m	a	m	a	m	a	m	a	m	a
P	**m**	a	m	a	g	a	m	a					

(j = 0)

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

Let T = “rahmamamamamagama”

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

(i = 1)

T	r	**a**	h	m	a	m	a	m	a	m	a	m	a
P		**m**	a	m	a	g	a	m	a				

(j = 0)

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

Let T = "rahmamamamagama"

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

(i = 2)

T	r	a	**h**	m	a	m	a	m	a	m	a	m	a
P			**m**	a	m	a	g	a	m	a			

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

$T[7]$

$P[f(3)] = P[2]$

Let T = “rahmamamamagama”

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

(i = 3)

(i=7)

T	r	a	h	m	a	m	a	m	a	m	a	m	a
P				m	a	m	a	g	a	m	a		

(j=4)

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

Let T = “rahmamamamagama”

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

(i = 7)

T	r	a	h	m	a	m	a	**m**	a	m	a	m	a
P				m	a	m	a	**g**	a	m	a		

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

Let T = “rahmamamamamagama”

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

(i = 7)

T	r	a	h	m	a	m	a	**m**	a	m	a	m	a
P				m	a	m	a	**g**	a	m	a		

(j = 4)

Mismatch at P[4]

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

Let T = “rahmamamamagama”

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

T	r	a	h	m	a	m	a	**m**	a	m	a	m	a
P				m	a	m	a	**g**	a	m	a		

Mismatch at P[4], align P[2] with T[7]

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

Let T = “rahmamamamagama”

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

T	r	a	h	m	a	m	a	**m**	a	m	a	m	a
P						m	a	m	a	**g**	a	m	a
				m	a	m	a	**g**	a	m	a		

Mismatch at P[4], align P[2] with T[7] **Why?**

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

Let T = “rahmamamamagama”

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

T	r	a	h	m	a	m	a	**m**	a	m	a	m	a
P				m	a	m	a	**g**	a	m	a		

Mismatch at P[4], align P[2] with T[7] **Why?** **f(3) says these are equal**

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

Let T = “rahmamamamagama”

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

T	r	a	h	m	a	m	a	**m**	a	m	a	m	a
P				m	a	m	a	**g**	a	m	a		

Mismatch at P[4], align P[2] with T[7] **Why?**

Mismatch at P[4] → No mismatch before P[4]

In brief, the KMP algorithm can be described as: When a mismatch occurs at $T[i]$, if you are

- currently at $P[j]$ with some $j > 0$, then shift P to align $P[F(j-1)]$ with $T[i]$.
- currently at $P[0]$, then shift $P[0]$ to align with $T[i+1]$.

Let T = "rahmamamamamagama"

j	1	2	3	4	5	6	7
f(j)	0	1	2	0	0	1	2

T	r	a	h	m	a	m	a	**m**	a	m	a	m	a
P						m	a	m	a	**g**	a	m	a
				m	a	m	a	**g**	a	m	a		

Mismatch at P[4], align P[2] with T[7] **Why?**

No mismatch before P[4] → I can move pattern two spaces