



香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

String Algorithms

Darren Lu {firewater}

2026-06-27

Table of Contents

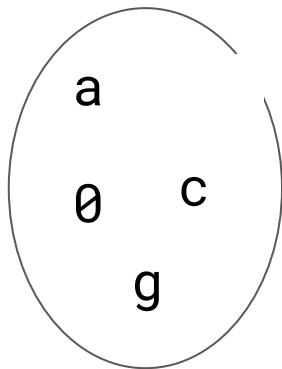
- Introduction
- Prefix Function
- Knuth–Morris–Pratt (KMP)
- From Prefix Function to Automata
- Aho-Corasick algorithm
- Suffix Array
- Others

Introduction

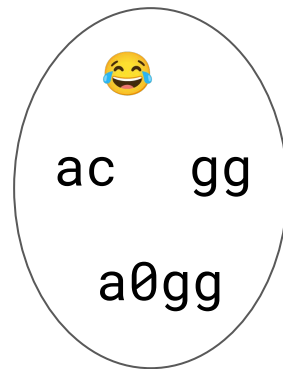
An **alphabet** Σ is a **finite** nonempty set of **symbols/characters**.

A **string** (over Σ) is a **finite** sequence of **symbols/characters** from Σ .

- `char s[SIZE];`
- `string s;`
- `vector<char>`
- "Hello world!"



Σ

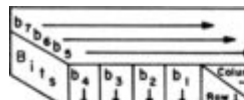


some strings over Σ

ASCII Code

American Standard Code for Information Interchange

- Character encoding
 - 7 bits / 8 bits (1 byte)
 - 0-127 / 0-255
-
- $'\backslash 0' = 0$, $'\backslash n' = 10$ $'\backslash r' = 13$
 - Digits: $'0' = 48+0$ $'9' = 48+9$
 - UpperCase: $'A' = 64+1$ $'Z' = 64+26$
 - LowerCase: $'a' = 96+1$ $'z' = 96+26$



	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Column	0	1	2	3	4	5	6	7
Row 0	0 0 0 0 0	0 0 0 0 1	0 0 0 1 0	0 0 0 1 1	0 0 1 0 0	0 0 1 0 1	0 0 1 1 0	0 0 1 1 1
Row 1	NUL	DLE	SP	@	P	\	p	
Row 2	SOH	DC1	!	A	Q	a	q	
Row 3	STX	DC2	"	B	R	b	r	
Row 4	ETX	DC3	#	C	S	c	s	
Row 5	EOT	DC4	\$	D	T	d	t	
Row 6	ENQ	NAK	%	E	U	e	u	
Row 7	ACK	SYN	&	F	V	f	v	
Row 8	BEL	ETB	'	G	W	g	w	
Row 9	BS	CAN	(	H	X	h	x	
Row 10	HT	EM	)	I	Y	i	y	
Row 11	LF	SUB	*	J	Z	j	z	
Row 12	VT	ESC	+	K	[	k	{	
Row 13	FF	FS	<	L	\	l		
Row 14	CR	GS	=	M	]	m	}	
Row 15	SO	RS	.	N	^	n	~	
Row 16	SI	US	/	?	O	_	o	DEL

String I/O and Functions

- scanf, printf
- fgets, puts
- getline
- [<cstring>](#)
- [<string>](#)
- See [Programming using C++ \(2026\)](#) for more details

Common String Terminology

Concatenation

- Addition in string

For strings s and u ,

- $t = \text{Concat}(s, u) = su$
- $1 + 2 = 3$
- $"1" + "2" = "12"$
- $"ab" + "bcd" = "abbcd"$

Lexicographic order

- Order/Comparison in string
- Dictionary order
 - Same length: numerical order of ASCII code
 - Different length: append null characters
- $"123" < "132" < "2" < "23" < "3"$
- $"a" < "bc" < "bcf" < "bd" < "z"$

Common String Terminology

Substring:

- A contiguous sequence of characters within a string
- e.g. BCD is a substring of ABCDE
- s is substring of t if there exist strings u and v such that $t = usv$.
- $s = t[i..j]$ (inclusive), $0 \leq i \leq j \leq |t| - 1$

Subsequence:

- A sequence obtained by deleting some or no characters of a string
- e.g. BDE is a subsequence of ABCDE
- Order of characters is kept

Common String Terminology

Prefix:

- A substring that starts from the beginning of a string
- e.g. ABCDE, ABCDE, ABCDE
- s is prefix of t if there exist string v such that $t = sv$.
- $s = \text{Prefix}(t, k) = t[0..k-1], 0 \leq k \leq |t|$

Suffix:

- A substring that ends with the end of a string
- e.g. ABCDEE, ABCDED, ABCDEC
- s is suffix of t if there exist string u such that $t = us$.
- $s = \text{Suffix}(t, k) = t[|t|-k..|t|-1], 0 \leq k \leq |t|$

Common String Terminology

- A proper prefix/suffix of a string is one is not equal to the string itself
- All prefixes and suffixes are substrings
- Every substring is a prefix of suffix: $t = u\underline{sv}$
- Every substring is a suffix of prefix: $t = \underline{u}sv$
- All substrings are subsequences, but not vice versa

Palindrome:

- A string that is the same when reversed
- e.g. A, ABBA, RACECAR, GAG

Why Strings?

Some interesting theoretical & practical problems

- Exact (and approximate) String Matching
- Counting distinct substrings?
- Finding palindromic substrings?
- Finding the shortest program that can output a particular piece of text?
(Kolmogorov complexity)
- Document retrieval?

Prefix Function

Prefix Function – Definition

Given a string S . The **prefix function** for this string is defined as an array π of length $|S|$, where $\pi[i]$ is the length of the longest proper prefix of the substring $S[0..i]$ which is also a suffix of this substring.

- A *proper* prefix of a string is a prefix that is not equal to the string itself
- By definition, $\pi[0] = 0$

$$\begin{aligned}\pi[i] &= \max_{0 \leq k \leq i} \{k: \text{Prefix}(S[0..i], k) = \text{Suffix}(S[0..i], k)\} \\ &= \max_{0 \leq k \leq i} \{k: S[0..k-1] = S[i-(k-1)..i]\}\end{aligned}$$

Prefix Function – Example

$$\pi[i] = \max_{0 \leq k \leq i} \{k: \text{Prefix}(S[0..i], k) = \text{Suffix}(S[0..i], k)\}$$

$S = \text{"ABABACB"}$

	A	B	A	B	A	C	B
$\pi[i]$	0						

Prefix Function – Example

$$\pi[i] = \max_{0 \leq k \leq i} \{k: \text{Prefix}(S[0..i], k) = \text{Suffix}(S[0..i], k)\}$$

$i = 1$
 “AB”
 AB $\neq$ AB

$S = \text{“ABABACB”}$

	A	B	A	B	A	C	B
$\pi[i]$	0	0					

Prefix Function – Example

$$\pi[i] = \max_{0 \leq k \leq i} \{k: \text{Prefix}(S[0..i], k) = \text{Suffix}(S[0..i], k)\}$$

$i = 2$

“ABA”

ABA $\neq$ ABA

ABA = ABA

$S = \text{“ABABACB”}$

	A	B	A	B	A	C	B
$\pi[i]$	0	0	1				

Prefix Function – Example

$$\pi[i] = \max_{0 \leq k \leq i} \{k: \text{Prefix}(S[0..i], k) = \text{Suffix}(S[0..i], k)\}$$

$i = 3$

“ABAB”

ABAB $\neq$ ABAB

ABAB = ABAB

$S = \text{“ABABACB”}$

	A	B	A	B	A	C	B
$\pi[i]$	0	0	1	2			

Prefix Function – Example

$$\pi[i] = \max_{0 \leq k \leq i} \{k: \text{Prefix}(S[0..i], k) = \text{Suffix}(S[0..i], k)\}$$

$S = \text{"ABABACB"}$

$i = 4$

"ABABA"

ABABA $\neq$ ABABA

ABABA = ABABA

	A	B	A	B	A	C	B
$\pi[i]$	0	0	1	2	3		

Prefix Function – Example

$$\pi[i] = \max_{0 \leq k \leq i} \{k: \text{Prefix}(S[0..i], k) = \text{Suffix}(S[0..i], k)\}$$

$S = \text{"ABABACB"}$

$i = 5$

"ABABAC"

ABABAC $\neq$ ABABAC

ABABAC $\neq$ ABABAC

ABABAC $\neq$ ABABAC

ABABAC $\neq$ ABABAC

ABABAC $\neq$ ABABAC

	A	B	A	B	A	C	B
$\pi[i]$	0	0	1	2	3	0	

Prefix Function – Example

$$\pi[i] = \max_{0 \leq k \leq i} \{k: \text{Prefix}(S[0..i], k) = \text{Suffix}(S[0..i], k)\}$$

$S = \text{"ABABACB"}$

$i = 6$

"ABABACB"

ABABACB $\neq$ ABABACB

ABABACB $\neq$ ABABACB

ABABACB $\neq$ ABABACB

ABABACB $\neq$ ABABACB

ABABACB $\neq$ ABABACB

ABABACB $\neq$ ABABACB

	A	B	A	B	A	C	B
$\pi[i]$	0	0	1	2	3	0	0

Trivial Algorithm

```
vector<int> prefix_function(string S) {  
    vector<int> pi(|S|);  
    for(int i = 1; i < |S|, i++)  
        for(int j = i; j > 0; j--)  
            if(S[0..j-1] == S[i-j+1, i])  
                pi[i] = j, break;  
    return pi;  
}
```

Time Complexity: $O(|S|^2)$ comparisons $\times O(|S|)$ per comparison = $O(|S|^3)$

Observation 1

$$\pi[i] \leq \pi[i-1] + 1$$

	A	B	A	B	A	C	B
$\pi[i]$	0	0	1	2	3	0	0

Proof: When $\pi[i] = 0$ then the statement is obviously true. When $\pi[i] \geq 1$,

$$\begin{array}{c}
 0 \qquad \qquad \qquad i \\
 \text{XXXXXXXXXXXXXXXXXXXXXXXXX} \\
 \hline
 \pi[i] \qquad \qquad \qquad \pi[i]
 \end{array}$$

Observation 1

$$\pi[i] \leq \pi[i-1] + 1$$

	A	B	A	B	A	C	B
$\pi[i]$	0	0	1	2	3	0	0

Proof: When $\pi[i] = 0$ then the statement is obviously true. When $\pi[i] \geq 1$,

0 i
 XXXXXXXXXXXXXXXXXXXXXXXX

Observation 1

$$\pi[i] \leq \pi[i-1] + 1$$

	A	B	A	B	A	C	B
$\pi[i]$	0	0	1	2	3	0	0

Proof: When $\pi[i] = 0$ then the statement is obviously true. When $\pi[i] \geq 1$,



$$\text{Prefix}(S[0..i-1], \pi[i]-1) = \text{Suffix}(S[0..i-1], \pi[i]-1)$$

Observation 1

$$\pi[i] \leq \pi[i-1] + 1$$

	A	B	A	B	A	C	B
$\pi[i]$	0	0	1	2	3	0	0

Proof: When $\pi[i] = 0$ then the statement is obviously true. When $\pi[i] \geq 1$,



$$\text{Prefix}(S[0..i-1], \pi[i]-1) = \text{Suffix}(S[0..i-1], \pi[i]-1)$$

$$\Rightarrow \pi[i-1] \geq \pi[i] - 1$$

Recall: Trivial Algorithm

```
vector<int> prefix_function(string S) {  
    vector<int> pi(|S|);  
    for(int i = 1; i < |S|, i++)  
        for(int j = i; j > 0; j--)  
            if(S[0..j-1] == S[i-j+1, i])  
                pi[i] = j, break;  
    return pi;  
}
```

Time Complexity: $O(|S|^2)$ comparisons $\times O(|S|)$ per comparison = $O(|S|^3)$

Algorithm – 1st Optimisation

```
vector<int> prefix_function(string S) {  
    vector<int> pi(|S|);  
    for(int i = 1; i < |S|, i++)  
        for(int j = pi[i-1]+1; j > 0; j--)  
            if(S[0..j-1] == S[i-j+1, i])  
                pi[i] = j, break;  
    return pi;  
}
```

Algorithm – 1st Optimisation

$$\pi[i] \leq \pi[i-1] + 1$$

- When moving to the next position, the value of the prefix function can only increase by at most one
- In total the function can grow at most $(|S|-1)$ steps
- Therefore it also can only decrease at most $(|S|-1)$ steps.
- $\Rightarrow$ We only need to perform at most $2|S|$ string comparisons.

Time Complexity: $O(|S|)$ comparisons $\times O(|S|)$ per comparison = $O(|S|^2)$

Observation 2

We want to get rid of the string comparisons.

In Observation 1, we have $\pi[i] \leq \pi[i-1] + 1$.

If $S[\pi[i-1]] == S[i]$, then we can say with certainty that $\pi[i] = \pi[i-1] + 1$.



Observation 2

We want to get rid of the string comparisons.

In Observation 1, we have $\pi[i] \leq \pi[i-1] + 1$.

If $S[\pi[i-1]] == S[i]$, then we can say with certainty that $\pi[i] = \pi[i-1] + 1$.

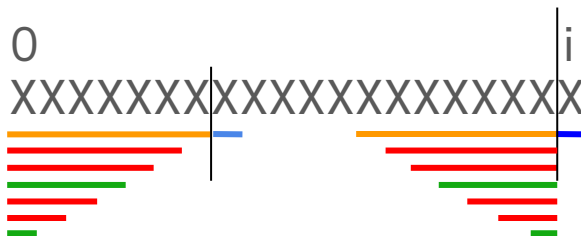


What if $S[\pi[i-1]] \neq S[i]$? What should we try next?

Observation 2

Case $S[\pi[i-1]] \neq S[i]$. What should we try next?

- The **red** prefix/suffix pairs do not match, anything appended to their ends will not make the substrings match.
- The **green** prefix/suffix pairs match. We can try extending these pairs.



Observation 2

Case $S[\pi[i-1]] \neq S[i]$. What should we try next?

- ⇒ We only need to consider all those (proper) **prefixes** which are also **suffixes** of $S[0..i-1]$.
- Once we have these pairs, string comparisons are no longer necessary.
- ⇒ We just need to compare $S[i]$ with the **character** behind the **prefix**.



Observation 2

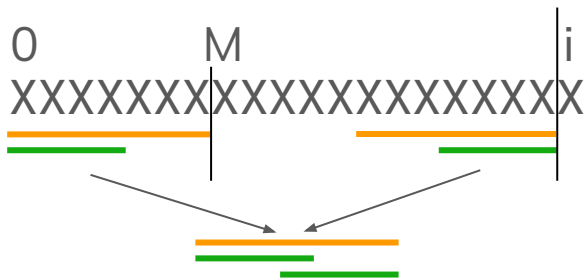
We consider all those proper **prefixes** which are also **suffixes** of $S[0..i-1]$, in decreasing order of length.

- The longest such prefix/suffix has length $M = \pi[i-1]$.
- What is the length of the 2nd longest such prefix/suffix?
- More generally, after considering current **prefix/suffix of length M** , what is the **next longest prefix/suffix**?



Observation 2

- The **orange** parts are the same.
 - The **next longest prefix/suffix of interest** is contained in the **orange** part.
- ⇒ The **green** substring is both prefix and suffix of the **orange** substring.
- ⇒ Length of **green** part = $\pi[M-1]$



Algorithm – 2nd Optimisation

```
vector<int> prefix_function(string S) {  
    vector<int> pi(|S|);  
    for(int i = 1; i < |S|, i++) {  
        int M = pi[i-1];  
        while(M > 0 && S[i] != S[M]) M = pi[M-1];  
        pi[i] = M + (S[i] == S[M]);  
    }  
    return pi;  
}
```

Algorithm – 2nd Optimisation

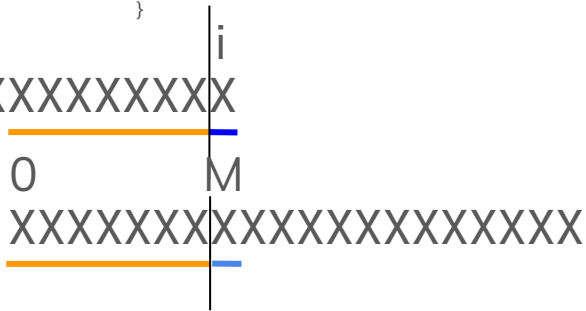
- Time Complexity: $O(|S|)$ comparisons $\times O(1)$ per comparison = $O(|S|)$
- Memory Complexity = $O(|S|)$
- This is an **online** algorithm, i.e. it processes the data as it arrives
- Can read the string characters one by one and compute the value of prefix function immediately
- Still requires storing the string itself

Let's look at the demonstration

$S = \text{"ABABACB"}$



XXXXXXXXXXXXXXXXXXXXXXXXX



```

vector<int> prefix_function(string S) {
    vector<int> pi(|S|);
    for(int i = 1; i < |S|, i++) {
        int M = pi[i-1];
        while(M > 0 && S[i] != S[M]) M = pi[M-1];
        pi[i] = M + (S[i] == S[M]);
    }
    return pi;
}

```

	A	B	A	B	A	C	B
pi[i]	0						

Let's look at the demonstration

$S = \text{"ABABACB"}$

```

vector<int> prefix_function(string S) {
    vector<int> pi(|S|);
    for(int i = 1; i < |S|, i++) {
        int M = pi[i-1];
        while(M > 0 && S[i] != S[M]) M = pi[M-1];
        pi[i] = M + (S[i] == S[M]);
    }
    return pi;
}
  
```

ABABACB
ABABACB

	A	B	A	B	A	C	B
pi[i]	0						

Let's look at the demonstration

$S = \text{"ABABACB"}$

$i=1$ **A**BABACB
 $M=0$ **A**BABACB

```
vector<int> prefix_function(string S) {
    vector<int> pi(|S|);
    for(int i = 1; i < |S|, i++) {
        int M = pi[i-1];
        while(M > 0 && S[i] != S[M]) M = pi[M-1];
        pi[i] = M + (S[i] == S[M]);
    }
    return pi;
}
```

	A	B	A	B	A	C	B
pi[i]	0	0					

Let's look at the demonstration

$S = \text{"ABABACB"}$

$i=2$

ABABACB

$M=0$

ABABACB

```
vector<int> prefix_function(string S) {
    vector<int> pi(|S|);
    for(int i = 1; i < |S|, i++) {
        int M = pi[i-1];
        while(M > 0 && S[i] != S[M]) M = pi[M-1];
        pi[i] = M + (S[i] == S[M]);
    }
    return pi;
}
```

	A	B	A	B	A	C	B
$pi[i]$	0	0	1				

Let's look at the demonstration

S = "ABABACB"

i=3

ABABACB

M=1

ABABACB

```
vector<int> prefix_function(string S) {
    vector<int> pi(|S|);
    for(int i = 1; i < |S|, i++) {
        int M = pi[i-1];
        while(M > 0 && S[i] != S[M]) M = pi[M-1];
        pi[i] = M + (S[i] == S[M]);
    }
    return pi;
}
```

	A	B	A	B	A	C	B
pi[i]	0	0	1	2			

Let's look at the demonstration

S = "ABABACB"

i=4

ABABACB

M=2

ABABACB

```
vector<int> prefix_function(string S) {
    vector<int> pi(|S|);
    for(int i = 1; i < |S|, i++) {
        int M = pi[i-1];
        while(M > 0 && S[i] != S[M]) M = pi[M-1];
        pi[i] = M + (S[i] == S[M]);
    }
    return pi;
}
```

	A	B	A	B	A	C	B
pi[i]	0	0	1	2	3		

Let's look at the demonstration

$S = \text{"ABABACB"}$

$i=5$

ABAB**C**B

$M=3$

ABAB**B**ACB

```
vector<int> prefix_function(string S) {
    vector<int> pi(|S|);
    for(int i = 1; i < |S|, i++) {
        int M = pi[i-1];
        while(M > 0 && S[i] != S[M]) M = pi[M-1];
        pi[i] = M + (S[i] == S[M]);
    }
    return pi;
}
```

	A	B	A	B	A	C	B
$pi[i]$	0	0	1	2	3		

Let's look at the demonstration

$S = \text{"ABABACB"}$

```
vector<int> prefix_function(string S) {
    vector<int> pi(|S|);
    for(int i = 1; i < |S|, i++) {
        int M = pi[i-1];
        while(M > 0 && S[i] != S[M]) M = pi[M-1];
        pi[i] = M + (S[i] == S[M]);
    }
    return pi;
}
```

$i=5$

ABAB**CB**

$M=1$

ABABACB

	A	B	A	B	A	C	B
$pi[i]$	0	0	1	2	3		

Let's look at the demonstration

$S = \text{"ABABACB"}$

```
vector<int> prefix_function(string S) {
    vector<int> pi(|S|);
    for(int i = 1; i < |S|, i++) {
        int M = pi[i-1];
        while(M > 0 && S[i] != S[M]) M = pi[M-1];
        pi[i] = M + (S[i] == S[M]);
    }
    return pi;
}
```

$i=5$

ABABACB

$M=0$

ABABACB

	A	B	A	B	A	C	B
$pi[i]$	0	0	1	2	3	0	

Let's look at the demonstration

$S = \text{"ABABACB"}$

```
vector<int> prefix_function(string S) {
    vector<int> pi(|S|);
    for(int i = 1; i < |S|, i++) {
        int M = pi[i-1];
        while(M > 0 && S[i] != S[M]) M = pi[M-1];
        pi[i] = M + (S[i] == S[M]);
    }
    return pi;
}
```

$i=6$

ABABACB

$M=0$

ABABACB

	A	B	A	B	A	C	B
$pi[i]$	0	0	1	2	3	0	0

Example

HKOJ R002 Power Strings

Define uv to be the concatenation of strings u and v .

For each string s , find the largest n such that $s = a^n$ for some string a .

$$\text{"aaaa"} = \text{"a"}^4$$

$$\text{"ababab"} = \text{"ab"}^3$$

Power Strings

HKOJ R002 Power Strings

Idea:

shortest repeating unit = $|S| - \pi[|S| - 1]$

ABCABCABCABCABC



Power Strings

HKOJ R002 Power Strings

Idea:

$$\text{shortest repeating unit} = |S| - \pi[|S| - 1]$$

ABCABCABCABCABC

ABCABCABCABCAB

- If a string cannot be perfectly divided by its smallest repeating unit, it cannot be perfectly divided by any other repeating unit.
- Proof is left as exercise.

Knuth-Morris-Pratt (KMP)

String Matching

Exact String Matching:

- Given a string **S** and a string/pattern **T**,
- Find the positions of all occurrences of the pattern **T** in **S**.

Problems with previous approaches

Comparing whole string is too slow

- $O(|T|)$

Comparing hash value of string instead

- $O(1)$
- Maintain hash value with sliding window
- Time Complexity = $O(|S| + |T|)$
- **Collisions** may result in WA

KMP Algorithm

From the naive approach:

Need to recalculate everything after a mismatch

Can we do better?

Idea: make use of the **information from the previous (mis)match(es)**
 skip over positions that are guaranteed not to match the pattern
 avoid rematching

Example

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABABABACB

ABABACB

Example

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABA**B**ABACB

ABABA**C**B

Pattern not found at index 0

Where should we look at next?

Example

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABABABACB

ABABACB

“Sliding window” in hash...

Example

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABABABACB

ABABACB

“Sliding window” in hash...

KMP Algorithm

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABABABACB

ABABACB

KMP Algorithm

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABACB#ABABABABACB

ABABACB#ABABABABACB

	A	B	A	B	A	C	B	#	...
$\pi[i]$	0	0	1	2	3	0	0	0	...

KMP Algorithm

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABACB#ABABABABACB

ABABACB#ABABABABACB

	A	B	A	B	A	C	B	#	...
$\pi[i]$	0	0	1	2	3	0	0	0	...

KMP Algorithm

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABACB#ABABABABACB

ABABACB#ABABABABACB

	A	B	A	B	A	C	B	#	...
$\pi[i]$	0	0	1	2	3	0	0	0	...

KMP Algorithm

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABACB#ABABABACB

ABABACB#ABABABABACB

	A	B	A	B	A	C	B	#	...
$\pi[i]$	0	0	1	2	3	0	0	0	...

KMP Algorithm

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABACB#ABABABACB

ABABACB#ABABABABACB

	A	B	A	B	A	C	B	#	...
$\pi[i]$	0	0	1	2	3	0	0	0	...

KMP Algorithm

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABACB#ABABABABACB

ABABACB#ABABABABACB

	A	B	A	B	A	C	B	#	...
$\pi[i]$	0	0	1	2	3	0	0	0	...

KMP Algorithm

$S = \text{"ABABABABACB"}$

$T = \text{"ABABACB"}$

ABABACB#ABAB**ABABACB**

ABABACB#ABABABABACB

$$\Rightarrow \pi[18] = 7$$

$$\Rightarrow \text{Position of occurrence of } T \text{ in } S = 18 - (|T|+1) - (|T|-1) = 4$$

KMP Algorithm – Combined Version

```
vector<int> KMP(string S, string T) {  
    vector<int> pi = prefix_function(T + '#' + S);  
    vector<int> res;  
    for(int i = |T|+1; i <= |T|+|S|, i++)  
        if(pi[i] == |T|) res.push_back(i - 2*|T|);  
    return res;  
}
```

Time Complexity: $O(|S| + |T|)$

KMP Algorithm

Compute the prefix function for the string $(T + \text{"\#"} + S)$

- ‘#’ is a separator that appears neither in S nor in T (or even more so, not in the alphabet Σ)

KMP Algorithm

By the definition and construction of the prefix function, for positions after ‘#’, $\pi[|T|+1+i]$ represents the length of the longest proper:

- Prefix (which belongs to the T part) which is also a
- Suffix (which belongs to the S part)

of the substring (T + “#” + S[0..i])

- Due to the separator, both prefix and suffix do not cross over ‘#’
- Therefore $\pi[i] \leq |T|$, and when $\pi[i] = |T|$, we have a match!

KMP Algorithm

Solves single pattern matching problem

- Search pattern T in string S

Question: What if we want to search T in multiple input strings?

- The prefix function for T is the same

Recall: Prefix Function

```
vector<int> prefix_function(string S) {  
    vector<int> pi(|S|);  
    for(int i = 1; i < |S|, i++) {  
        int M = pi[i-1];  
        while(M > 0 && S[i] != S[M]) M = pi[M-1];  
        pi[i] = M + (S[i] == S[M]);  
    }  
    return pi;  
}
```

Recall: Prefix Function (tiny rearrangements)

```
vector<int> prefix_function(string S) {  
    vector<int> pi(|S|);  
    for(int i = 1, M = 0; i < |S|, i++) {  
        while(M > 0 && S[i] != S[M]) M = pi[M-1];  
        if(S[i] == S[M]) M++;  
        pi[i] = M;  
    }  
    return pi;  
}
```

KMP Algorithm – Split Version

```
vector<int> KMP(string S, string T) {  
    vector<int> res, pi = prefix_function(T);  
    for(int i = 0, M = 0; i < |S|; i++) {  
        while(M > 0 && S[i] != T[M]) M = pi[M-1];  
        if(S[i] == T[M]) M++;  
        if(M == |T|) res.push_back(i-|T|+1), M = pi[M-1];  
    }  
    return res;  
}
```

KMP Algorithm

Further constant optimisation made by Knuth by defining the array π beyond the prefix function and serve specifically for the KMP algorithm

No improvement on the time complexity though, thus sometimes ignored

Example 1

Codeforces 808G Anthem of Berland

- You are given two strings S and T , some letters in S is $?$
- Replace $?$ with letters
- Maximize the number of occurrences of T in S
- $|T| \times |S| \leq 10^7$

Anthem of Berland

- $DP[i][j]$: we filled first i letter in S , and the suffix of $S[0 \dots i]$ is matching $T[0 \dots j]$
- If $S[i+1]$ is not '?', find the matching state that match $S[i+1]$ by KMP
- If $S[i+1]$ is '?', do one of the following
 - let $S[i+1] = T[j+1]$ (match with $T[j+1]$)
 - Let $j = pi[j]$ (find another matching state)
- Time complexity $O(|S||T|)$

Example 2

Codeforces 1017E The World Is a Theatre

- You are given two sets of points representing two convex polygons.
- Determine if the first polygon can be translated and rotated so that it completely coincides with the second polygon.
- $N \leq 10^5$

The World Is a Theatre

- Find the convex polygon and save the nodes in clockwise direction
 - Let array $A = [\text{dis } 0-1, \text{angle } 0-1-2, \text{dis } 1-2, \text{angle } 1-2-3, \dots, \text{dis } (N-2)-(N-1), \text{angle } (N-2)-(N-1)-0]$ of first polygon
 - B be the array of second polygon
 - Check whether we can make A equal to B by rotating
-
- Precompute the prefix function of A
 - Insert $B[0 \dots 2M]$ into KMP
 - Insert $B[0 \dots 2M]$ into KMP again, check whether $\text{pi}[i] = 2N$
 - Time complexity $O(N+M)$

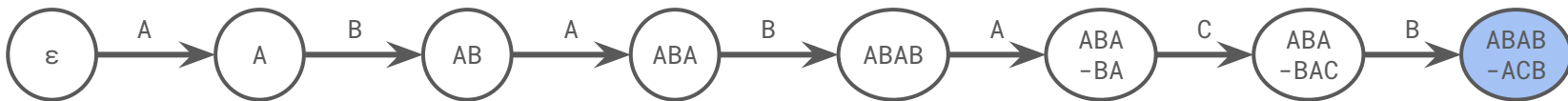
From Prefix Function to Automata

Recall: Trie: Prefix tree

In a **trie**, every node represents a word, or a **prefix** of a word

If the **dictionary** consists of only one string, then the constructed trie is actually a “**prefix chain**”

e.g. $S = \text{“ABABACB”}$

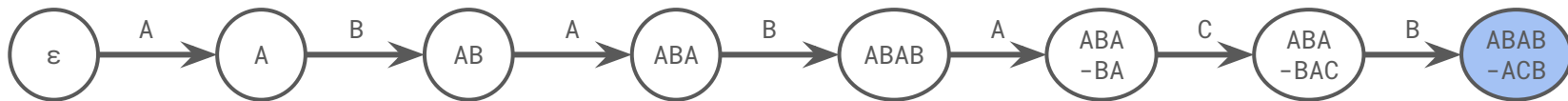


Recall: Prefix Function

The **prefix function** for string S is defined as an array π of length $|S|$, where $\pi[i]$ is the length of the longest proper prefix of the substring $S[0..i]$ which is also a suffix of this substring.

e.g. $S = \text{"ABABACB"}$

	A	B	A	B	A	C	B
$\pi[i]$	0	0	1	2	3	0	0

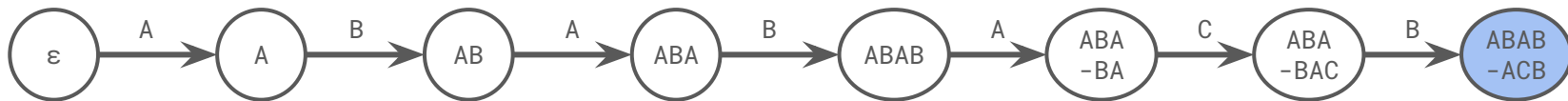


Suffix Link

A **suffix link** for node p is an edge that points to the longest proper suffix of p that is in the trie.

- If that is in the trie, that must be a prefix of some word (in our case S) by the definition of trie.

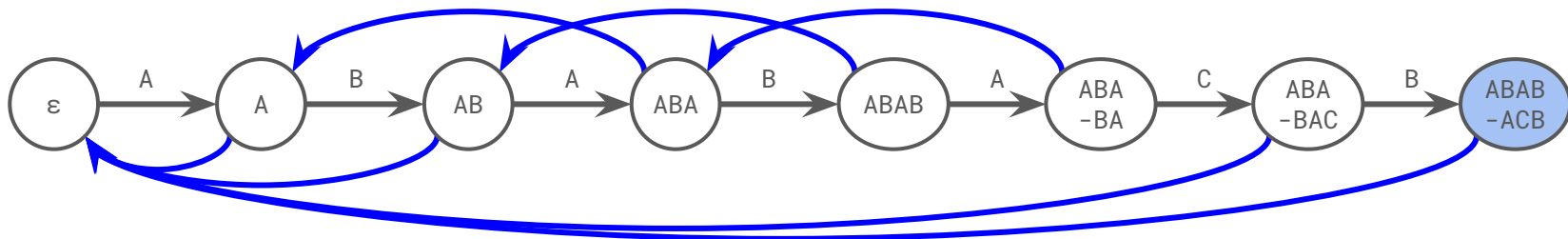
⇒ Prefix function applies, we create **suffix links** according to $\pi[i]$



Suffix Link

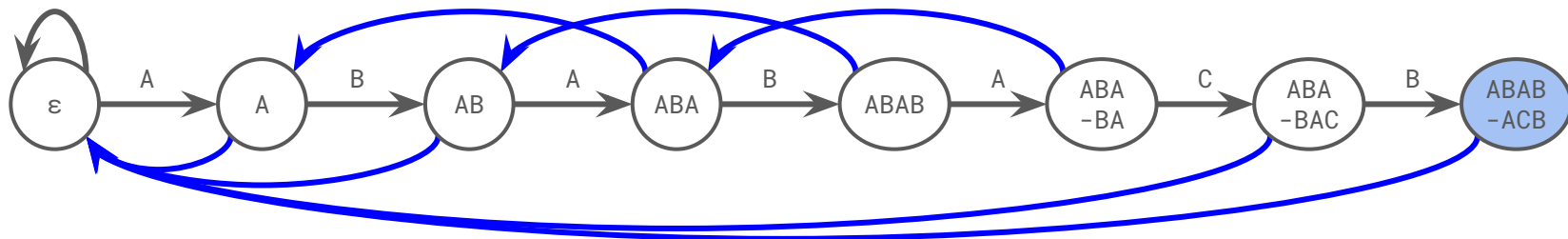
$S = \text{"ABABACB"}$

	A	B	A	B	A	C	B
$pi[i]$	0	0	1	2	3	0	0



Let's add one more thing...

otherwise

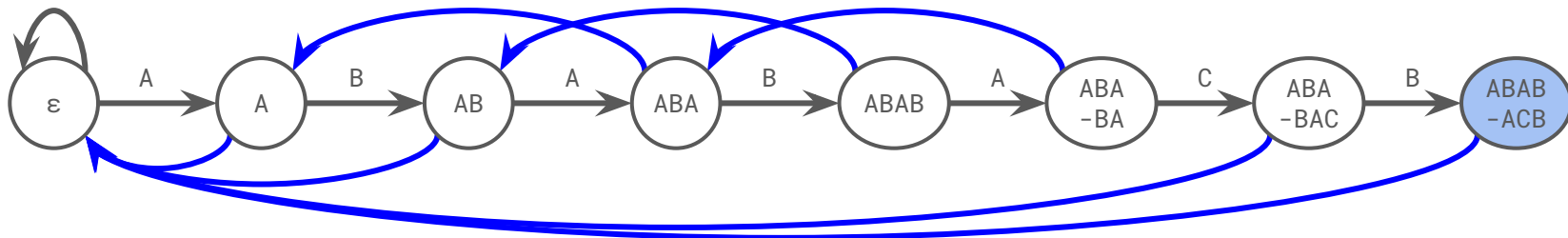


KMP Algorithm: Automaton Version

Done!

What you are seeing now is (almost) a **Deterministic Finite Automaton (DFA)**

otherwise



Deterministic Finite Automaton (DFA)

A **deterministic finite automaton (DFA)** is a tuple $M = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states
- Σ is an alphabet
- $\delta: Q \times \Sigma \rightarrow Q$ is a transition function
 - $s' = \delta(s, a)$
- $q_0 \in Q$ is the initial state
- $F \subset Q$ is the set of accepting states

Deterministic Finite Automaton (DFA)

In OI terms,

- DFA is a directed graph that takes a string as input
- Return yes/no according to the ending state, check whether they belong to the accepting states
 - In our string matching case, we declare a match whenever we visit an accepting state
- A string with its every element in Σ is a valid input to the DFA

Deterministic Finite Automaton (DFA)

Finite: The number of states is finite

Deterministic: The transition function is deterministic

- Every state has exactly one transition for each possible input symbol

Automaton:

- “*designed to automatically follow a sequence of operations, or respond to predetermined instructions*”
- It takes a string as input, processes the symbols in order one by one
- For each symbol, it moves from its current state to the next state following the predetermined transition function

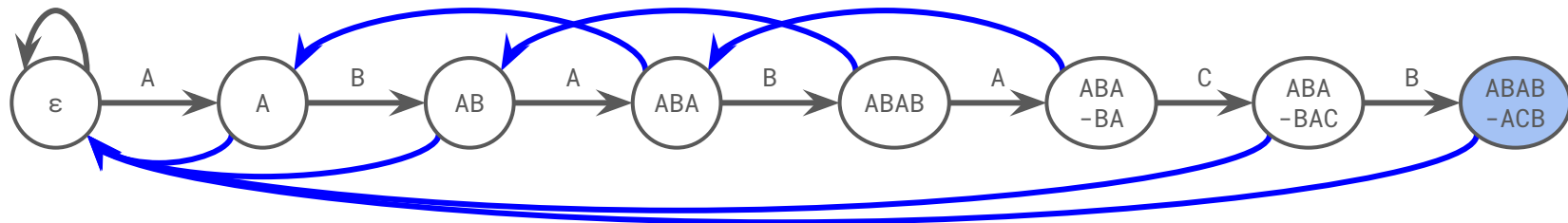
KMP Algorithm: Automaton Version

Exact String Matching Problem: Find all occurrences of the pattern **T** in **S**.

From all our previous insights, our goal is to:

- As we process the input symbols of **S** one by one,
- Indicate the maximum match we have between
 - The end (suffix) of the input processed so far, and
 - The beginning (prefix) of the pattern **T**

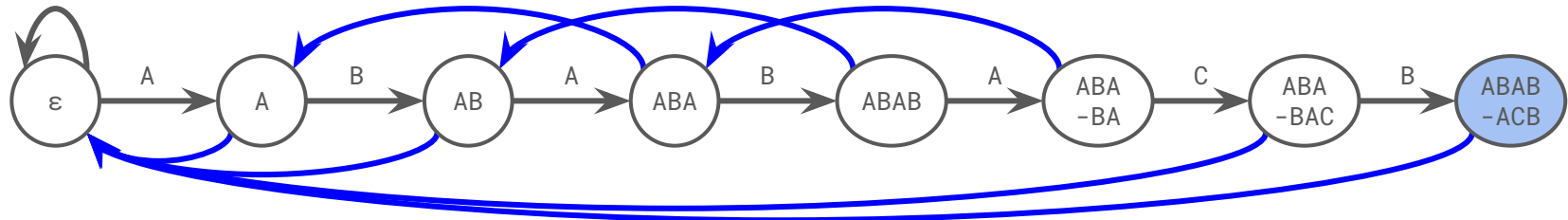
otherwise



KMP Algorithm: Automaton Version

- States: After matching the first i -th characters of T
- Initial states: Matching from empty
- Accepting states: After matching all characters of T
- Transition function: ???

otherwise



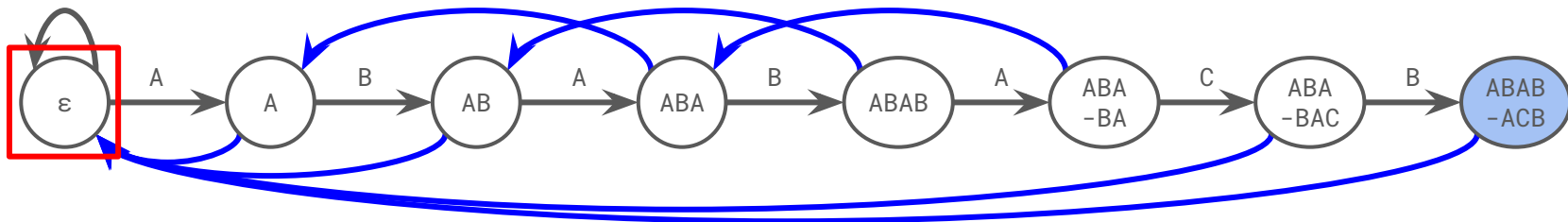
KMP Algorithm: Automaton Version

Transition function: Let input be c and the current state be i (i.e. $T[0..i-1]$)

- If $c == T[i]$, matches with the next character, proceed to next state:
 $\Rightarrow \delta(i, c) = i+1$

$S = \text{"ABAAABABACB"} , T = \text{"ABABACB"}$

otherwise



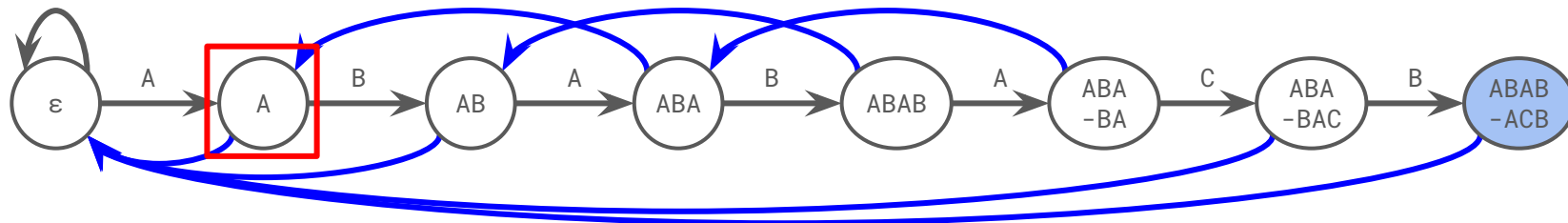
KMP Algorithm: Automaton Version

Transition function: Let input be c and the current state be i (i.e. $T[0..i-1]$)

- If $c == T[i]$, matches with the next character, proceed to next state:
 $\Rightarrow \delta(i, c) = i+1$

$S = \text{"ABAAABABACB"} \text{, } T = \text{"ABABACB"}$

otherwise



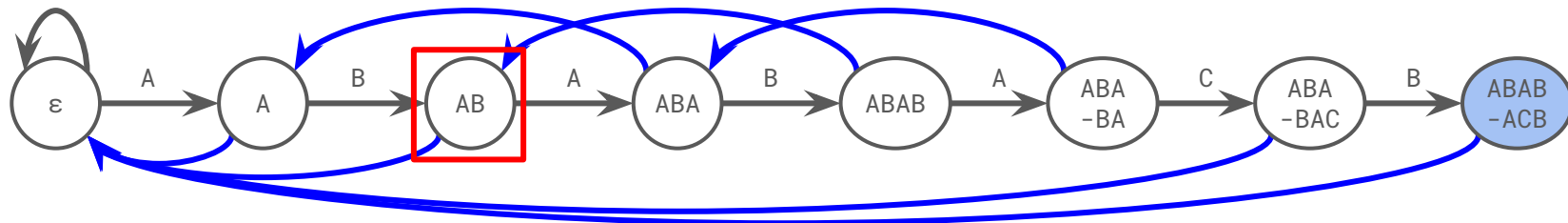
KMP Algorithm: Automaton Version

Transition function: Let input be c and the current state be i (i.e. $T[0..i-1]$)

- If $c == T[i]$, matches with the next character, proceed to next state:
 $\Rightarrow \delta(i, c) = i+1$

$S = \text{"ABAAABABACB"}\text{"}$, $T = \text{"ABABACB"}\text{"}$

otherwise



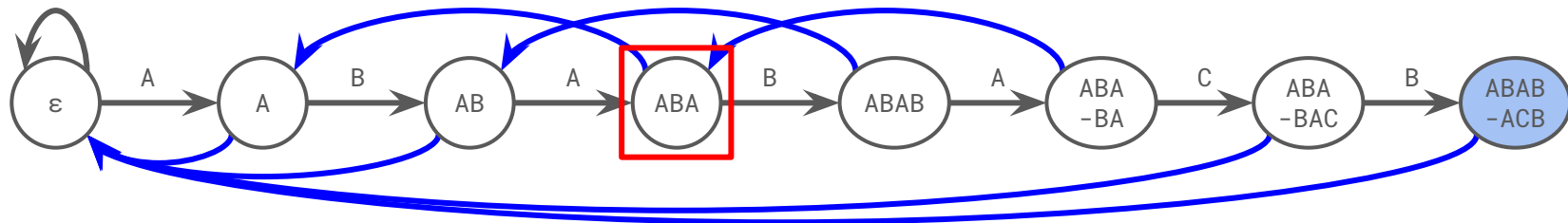
KMP Algorithm: Automaton Version

Transition function: Let input be c and the current state be i (i.e. $T[0..i-1]$)

- If $c == T[i]$, matches with the next character, proceed to next state:
 $\Rightarrow \delta(i, c) = i+1$

$S = \text{"ABA AABABACB"} \text{, } T = \text{"ABABACB"}$

otherwise



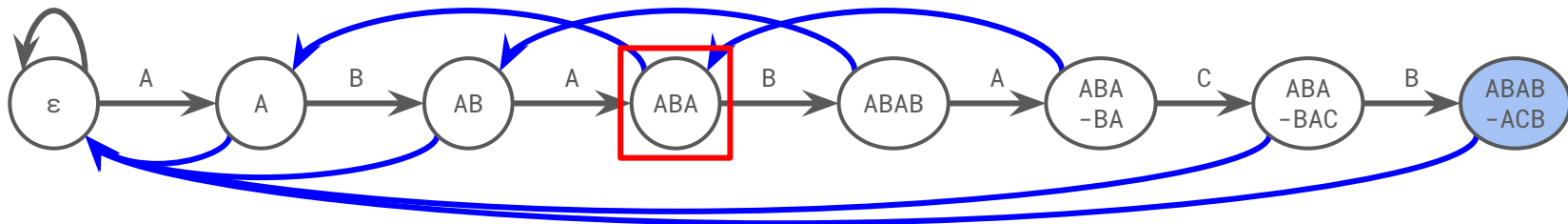
KMP Algorithm: Automaton Version

Transition function: Let input be c and the current state be i (i.e. $T[0..i-1]$)

- If $c \neq T[i]$, we cannot follow the grey edge originally in the trie
- We **fall back** to the longest proper suffix of $T[0..i-1]$ that is in the trie

$S = \text{"ABAABABACB"}\text{"}$, $T = \text{"ABABACB"}\text{"}$

otherwise



KMP Algorithm: Automaton Version

Transition function: Let input be c and the current state be i (i.e. $T[0..i-1]$)

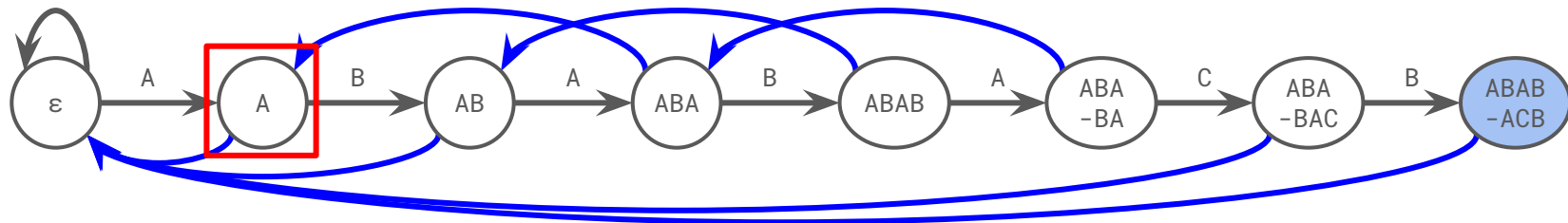
- If $c \neq T[i]$, we cannot follow the grey edge originally in the trie
- We **fall back** to the longest proper suffix of $T[0..i-1]$ that is in the trie

⇒ This is where **suffix links** come into play: they are for **mismatches**

⇒ $\delta(i, c) = \delta(\pi[i-1], c)$

$S = \text{"AB}^{\text{green}}\text{A}^{\text{red}}\text{ABABACB"}\text{"}, T = \text{"ABABACB"}$

otherwise



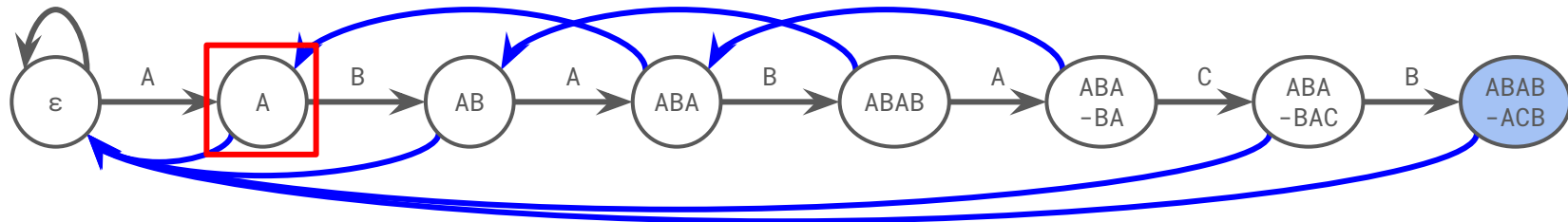
KMP Algorithm: Automaton Version

Transition function: Let input be **c** and the current state be **i** (i.e. $T[0..i-1]$)

- If $c \neq T[i]$, we follow the **suffix link** instead, then try to perform from there
 - Note that c still has not been consumed yet

S = "ABABABACB", T = "ABABACB"

otherwise



KMP Algorithm: Automaton Version

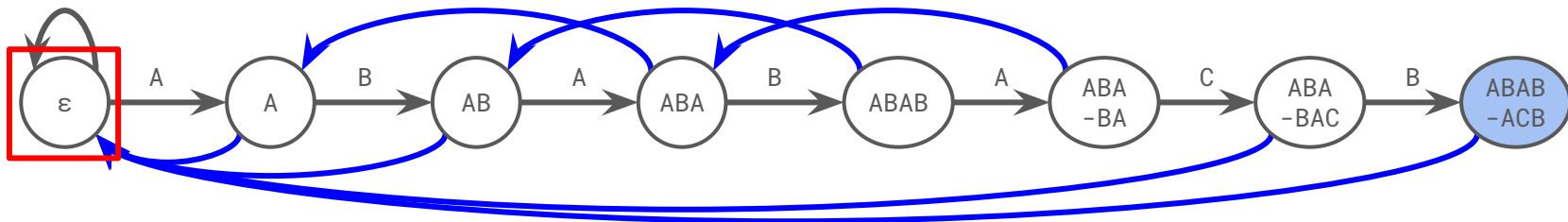
Transition function: Let input be c and the current state be i (i.e. $T[0..i-1]$)

- If $c \neq T[i]$, we follow the **suffix link** instead, then try to perform from there
 - Note that c still has not been consumed yet
 - If still doesn't match, follow more **suffix links**

$\Rightarrow$ It is possible that $\delta(i, c) = \delta(\pi[i-1], c) = \delta(\pi[\pi[i-1]-1], c) = \dots$

$S = \text{"ABAABABACB"}^*$, $T = \text{"ABABACB"}$

otherwise



KMP Algorithm: Automaton Version

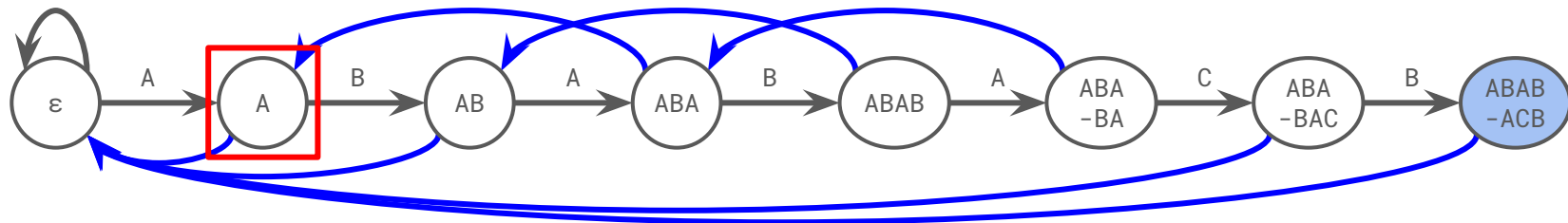
Transition function: Let input be c and the current state be i (i.e. $T[0..i-1]$)

- If $c \neq T[i]$, we follow the **suffix link** instead, then try to perform from there
 - Note that c still has not been consumed yet
 - If still doesn't match, follow more **suffix links**

⇒ It is possible that $\delta(i, c) = \delta(\pi[i-1], c) = \delta(\pi[\pi[i-1]-1], c) = \dots$

$S = \text{"ABAABABACB"}^*$, $T = \text{"ABABACB"}$

otherwise

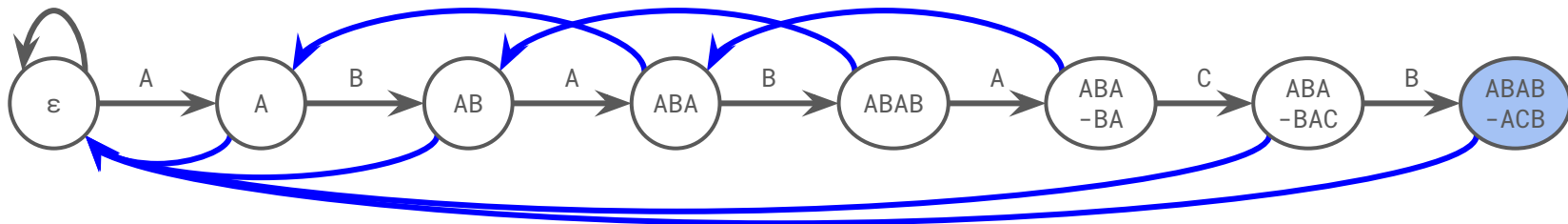


KMP Algorithm: Automaton Version

The graph you are seeing now is *almost* a DFA, because **suffix link** is not real transition function; they are “fallback” mechanisms.

- The real transition function δ is formed by recursive relations thanks to suffix links / the prefix function
- From the context of the graph, a real transition is formed by **suffix link(s)** until the soonest matching grey edge using the current letter

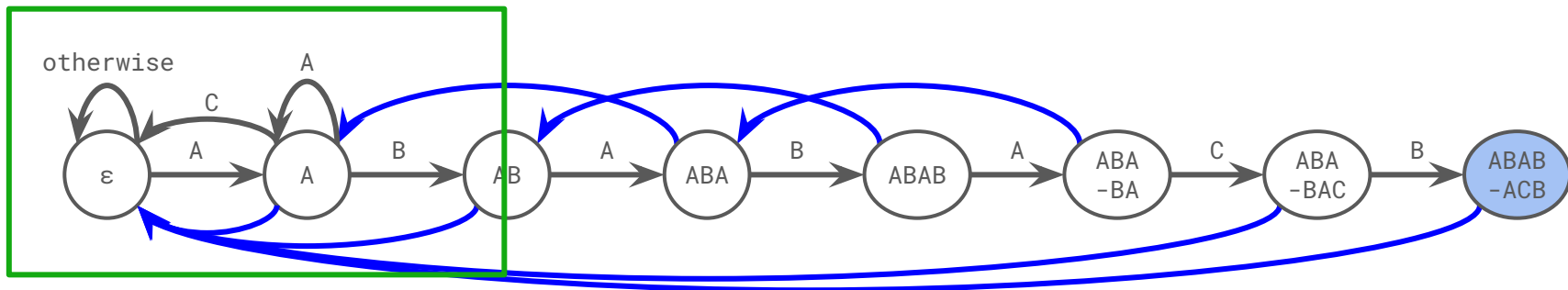
otherwise



KMP Algorithm: Automaton Version

The graph you are seeing now is *almost* a DFA, because **suffix link** is not real transition function; they are “fallback” mechanisms.

- The real transition function δ is formed by recursive relations thanks to suffix links / the prefix function
- From the context of the graph, a real transition is formed by **suffix link(s)** until the soonest matching grey edge using the current letter

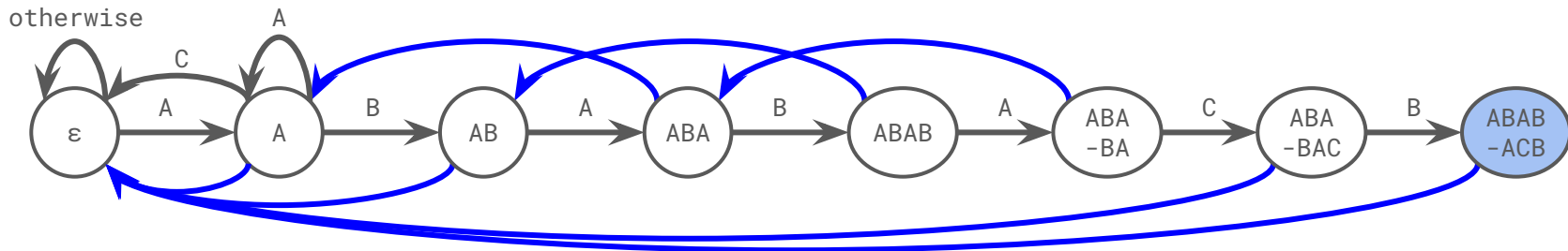


KMP Algorithm: Automaton Version

- $\delta(i, c) = (c == T[i]) ? (i+1) : \delta(\pi[i-1], c)$
- Computing the transition function in increasing order of i + Memoization
 $\Rightarrow$ Dynamic Programming

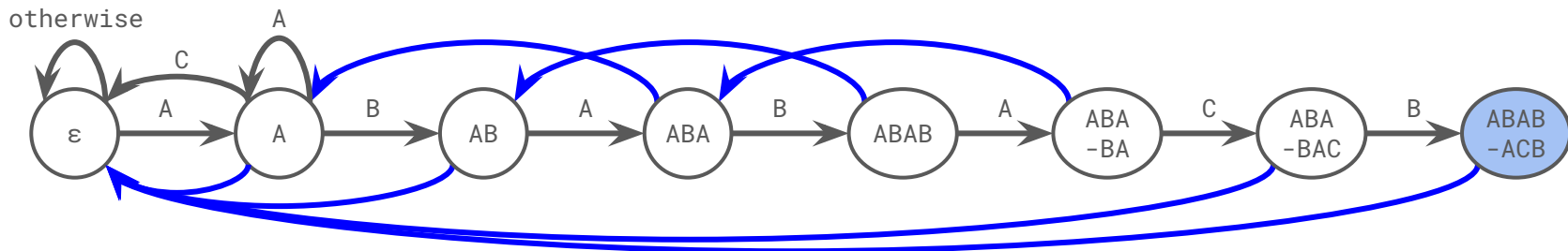
Time Complexity:

- Build automaton: $O(|T|)$ prefix function + $O(|\Sigma| \times |T|)$
- String matching (run automaton): $O(|S|)$



KMP Algorithm: Automaton Version

i	$\emptyset(\epsilon)$	1(A)	2(B)	3(A)	4(B)	5(A)	6(C)	7(B)	8(#)
A	1	1	3	1	5	1	1	1	/
B	$\emptyset$	2	$\emptyset$	4	$\emptyset$	4	7	$\emptyset$	/
C	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	6	$\emptyset$	$\emptyset$	/



KMP Algorithm: Automaton Version

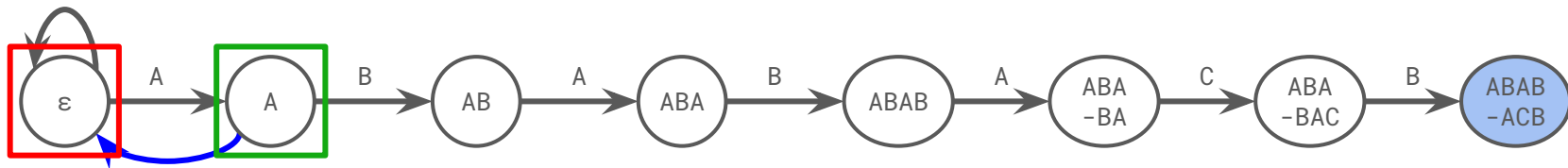
Final question: How to build **without precomputing prefix function**?

Method 1:

- Since prefix function stores length of the longest **proper** prefix/suffix,
 $\Rightarrow$ Run $T[1..|T|-1]$, the longest possible proper suffix, on the partially constructed automaton as we construct

$T = \text{"ABABACB"} \text{, } T[1..] = \text{"ABABACB"}$

otherwise



KMP Algorithm: Automaton Version

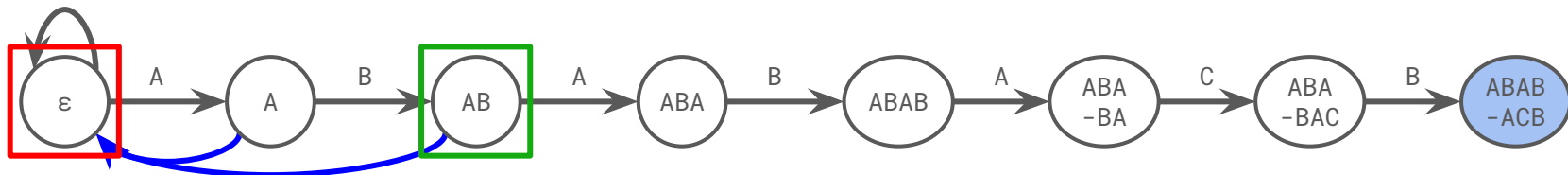
Final question: How to build **without precomputing prefix function**?

Method 1:

- Since prefix function stores length of the longest **proper** prefix/suffix,
 $\Rightarrow$ Run $T[1..|T|-1]$, the longest possible proper suffix, on the partially constructed automaton as we construct

$T = \text{"ABABACB"} \text{, } T[1..] = \text{"ABABACB"}$

otherwise



KMP Algorithm: Automaton Version

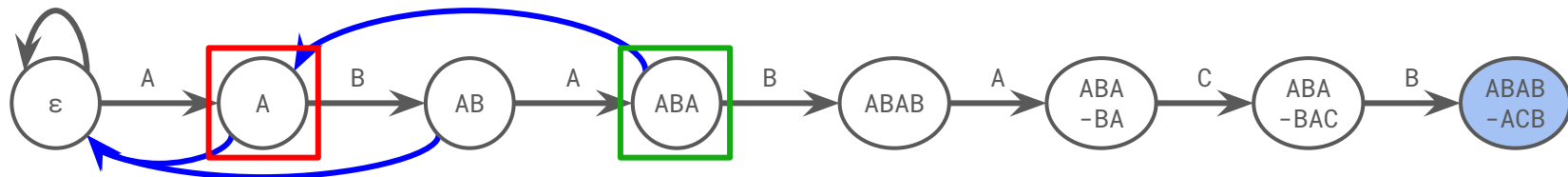
Final question: How to build **without precomputing prefix function**?

Method 1:

- Since prefix function stores length of the longest **proper** prefix/suffix,
 $\Rightarrow$ Run $T[1..|T|-1]$, the longest possible proper suffix, on the partially constructed automaton as we construct

$T = \text{"ABABACB"} \text{, } T[1..] = \text{"ABABACB"}$

otherwise



KMP Algorithm: Automaton Version

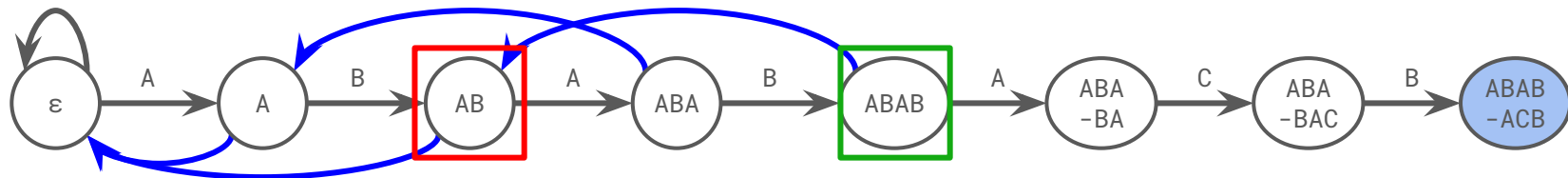
Final question: How to build **without precomputing prefix function**?

Method 1:

- Since prefix function stores length of the longest **proper** prefix/suffix,
⇒ Run $T[1..|T|-1]$, the longest possible proper suffix, on the partially constructed automaton as we construct

$T = \text{"ABABACB"} \text{, } T[1..] = \text{"ABABACB"}$

otherwise



KMP Algorithm: Automaton Version

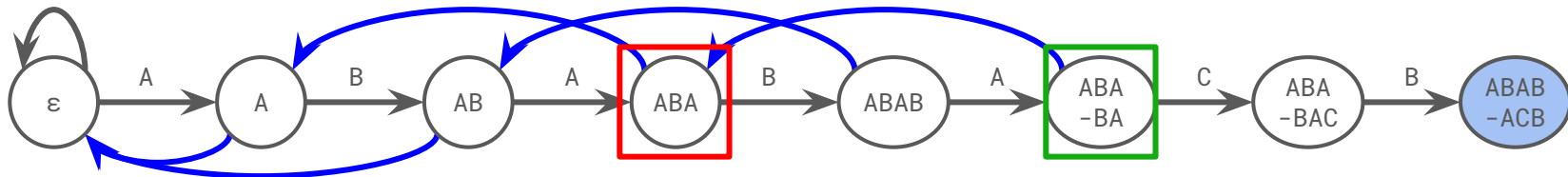
Final question: How to build **without precomputing prefix function**?

Method 1:

- Since prefix function stores length of the longest **proper** prefix/suffix,
 $\Rightarrow$ Run $T[1..|T|-1]$, the longest possible proper suffix, on the partially constructed automaton as we construct

$T = \text{"ABABACB"} \text{, } T[1..] = \text{"ABABACB"}$

otherwise



KMP Algorithm: Automaton Version

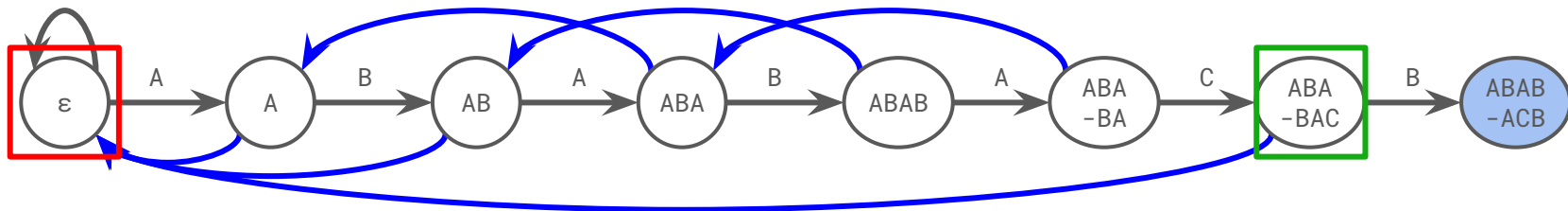
Final question: How to build **without** precomputing prefix function?

Method 1:

- Since prefix function stores length of the longest **proper** prefix/suffix,
 $\Rightarrow$ Run $T[1..|T|-1]$, the longest possible proper suffix, on the partially constructed automaton as we construct

$T = \text{"ABABACB"} \text{, } T[1..] = \text{"ABABACB"}$

otherwise



KMP Algorithm: Automaton Version

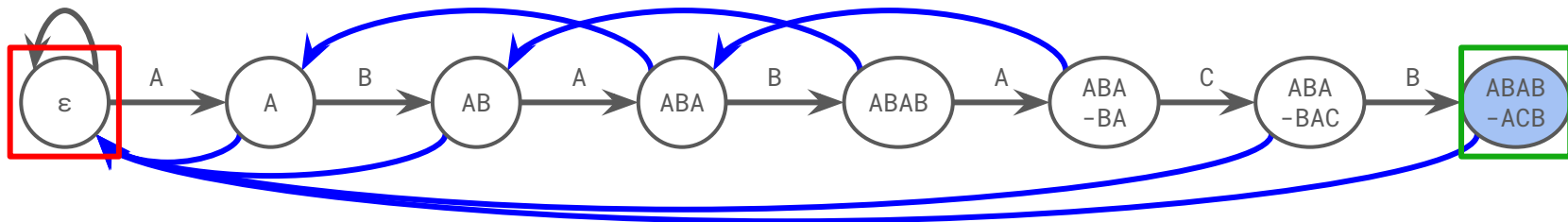
Final question: How to build **without precomputing prefix function**?

Method 1:

- Since prefix function stores length of the longest **proper** prefix/suffix,
 $\Rightarrow$ Run $T[1..|T|-1]$, the longest possible proper suffix, on the partially constructed automaton as we construct

$T = \text{"ABABACB"} \text{, } T[1..] = \text{"ABABACB"}$

otherwise



KMP Algorithm: Automaton Version

Final question: How to build **without precomputing prefix function**?

Method 2:

- Referring back to Prefix Function Observation 1
- Assume we are currently at node p
- Let $u+c'$ be the longest proper suffix of $p+c'$
 - $u+c'$ = longest proper suffix of $p+c'$
 - $u+c'$ = proper suffix of $p+c'$
 - u = proper suffix of p

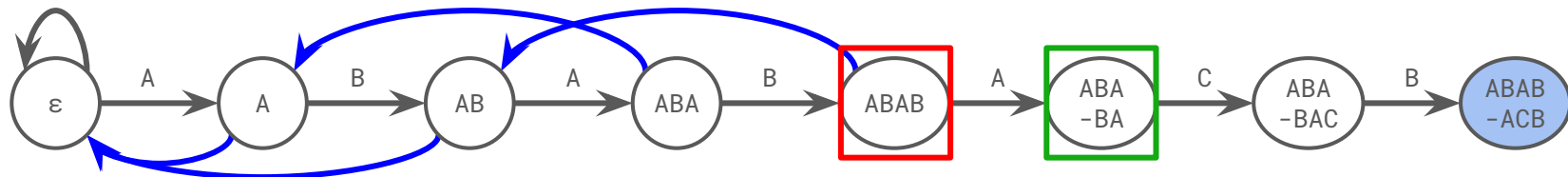
KMP Algorithm: Automaton Version

Final question: How to build **without precomputing prefix function**?

Method 2:

- Therefore, to construct the **suffix link** for a node **v**,
- We find its parent **p** in the trie and the label **c** of the **p-v** edge
- Then follow **p**'s suffix link to **u**, and perform the **c**-transition from there

otherwise



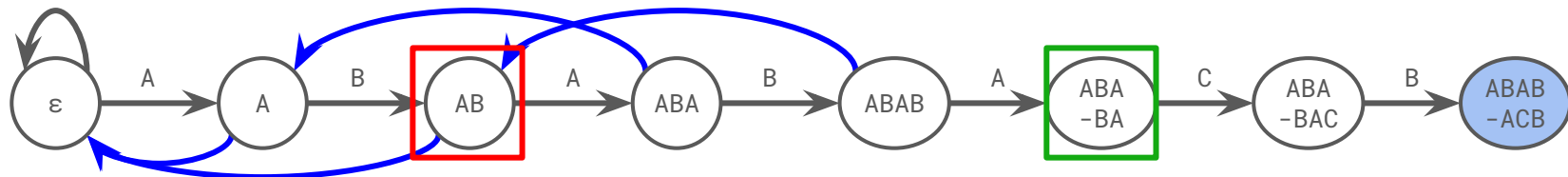
KMP Algorithm: Automaton Version

Final question: How to build **without precomputing prefix function**?

Method 2:

- Therefore, to construct the **suffix link** for a node **v**,
- We find its parent **p** in the trie and the label **c** of the **p-v** edge
- Then follow **p**'s suffix link to **u**, and perform the **c**-transition from there

otherwise



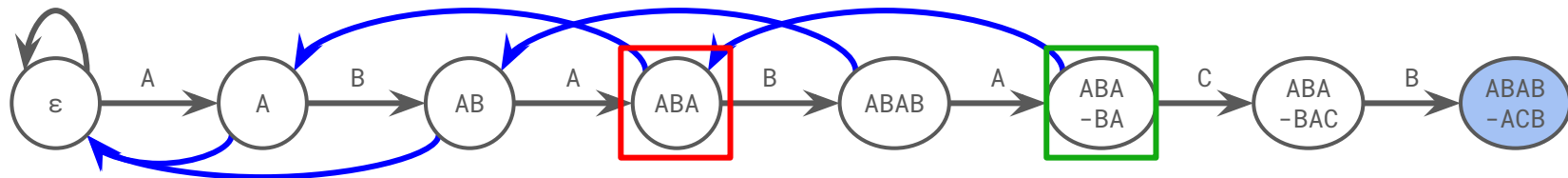
KMP Algorithm: Automaton Version

Final question: How to build **without precomputing prefix function**?

Method 2:

- Therefore, to construct the **suffix link** for a node **v**,
- We find its parent **p** in the trie and the label **c** of the **p-v** edge
- Then follow **p**'s suffix link to **u**, and perform the **c**-transition from there

otherwise



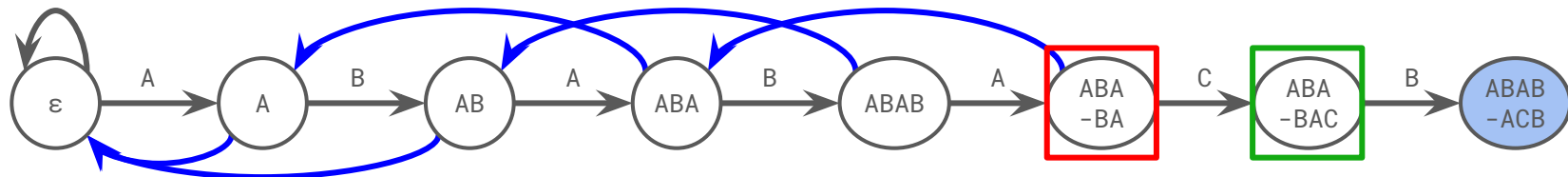
KMP Algorithm: Automaton Version

Final question: How to build **without precomputing prefix function**?

Method 2:

- Therefore, to construct the **suffix link** for a node **v**,
- We find its parent **p** in the trie and the label **c** of the **p-v** edge
- Then follow **p**'s suffix link to **u**, and perform the **c**-transition from there

otherwise



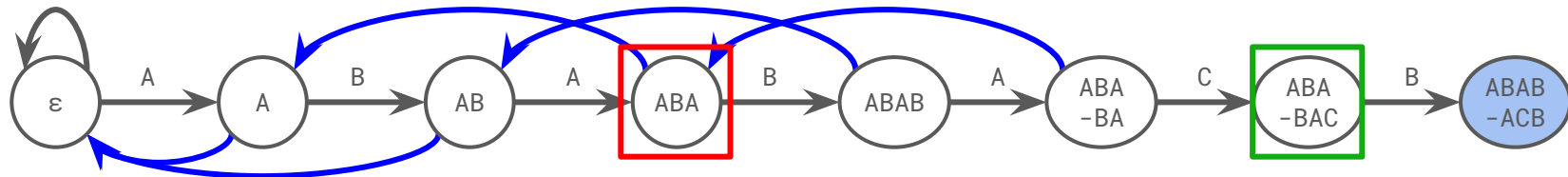
KMP Algorithm: Automaton Version

Final question: How to build **without precomputing prefix function**?

Method 2:

- Therefore, to construct the **suffix link** for a node **v**,
- We find its parent **p** in the trie and the label **c** of the **p-v** edge
- Then follow **p**'s suffix link to **u**, and perform the **c**-transition from there

otherwise



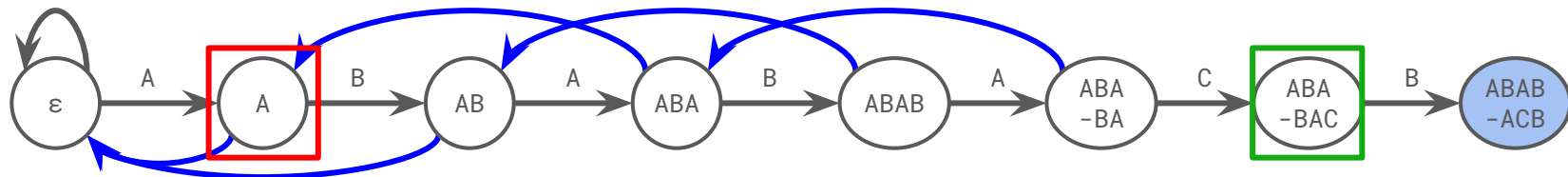
KMP Algorithm: Automaton Version

Final question: How to build **without precomputing prefix function**?

Method 2:

- Therefore, to construct the **suffix link** for a node **v**,
- We find its parent **p** in the trie and the label **c** of the **p-v** edge
- Then follow **p**'s suffix link to **u**, and perform the **c**-transition from there

otherwise



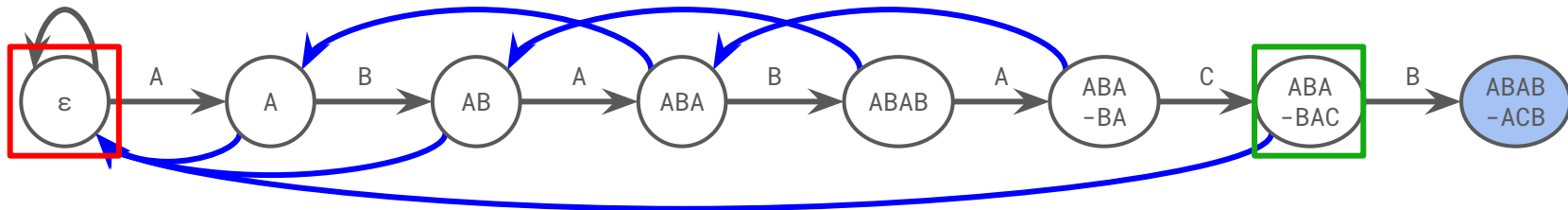
KMP Algorithm: Automaton Version

Final question: How to build **without precomputing prefix function**?

Method 2:

- Therefore, to construct the **suffix link** for a node **v**,
- We find its parent **p** in the trie and the label **c** of the **p-v** edge
- Then follow **p**'s suffix link to **u**, and perform the **c**-transition from there

otherwise



Aho-Corasick algorithm

Recall: KMP Algorithm

Solves single pattern matching problem

- Preprocess pattern T : $O(|T|)$ / $O(|\Sigma| \times |T|)$
- String matching in S : $O(|S|)$

What if we have multiple patterns to match?

- k patterns, total length = m
- Preprocess patterns individually: $O(m)$ / $O(|\Sigma| \times m)$
- String matching in S : $O(|S| \times k)$

Aho-Corasick algorithm

= Trie + KMP

KMP Automaton: built on a chain

AC Automaton: built on a trie

Aho-Corasick algorithm

KMP Automaton:

- Build a chain of prefixes
- Compute suffix links and transition functions in increasing order of i

AC Automaton:

- Build a trie with the patterns
 - Compute suffix links and transition functions in increasing order of length
 - Use Method 2; Method 1 is not applicable here
- ⇒ BFS on the trie

Aho-Corasick algorithm

Time Complexity:

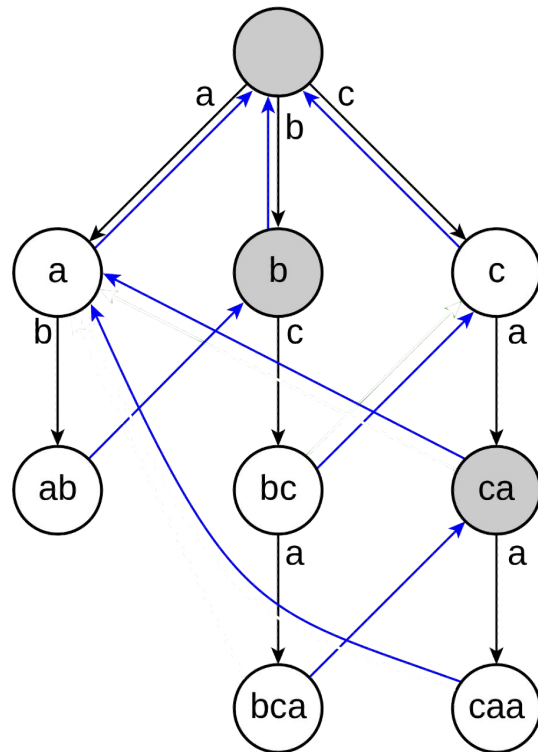
- Build AC Automaton: $O(|\Sigma| \times m)$
- Search patterns in S = Run AC Automaton: $O(|S|)$?

Example

Patterns = {"a", "ab", "bc", "bca", "c", "caa"}

S = "cabca"

- c found c
- c → ca
- ca → a → ab found ab
- ab → b → bc found bc
- bc → bca found bca



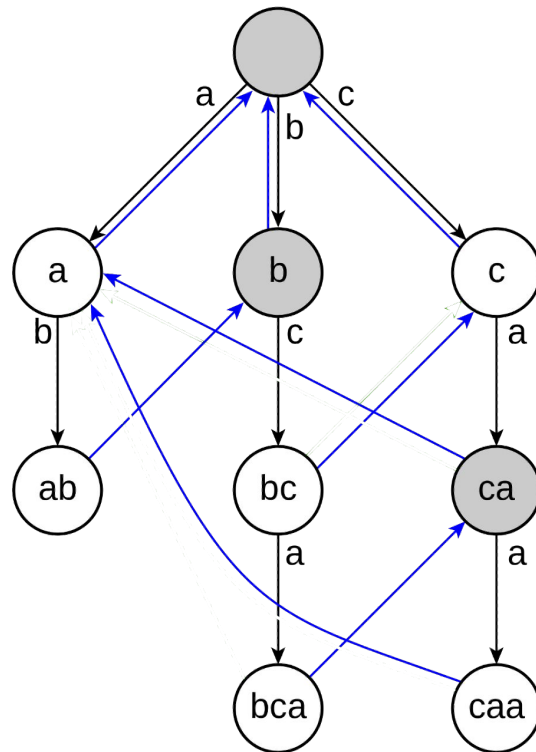
Example

Patterns = {"a", "ab", "bc", "bca", "c", "caa"}

S = "cabca"

- c found c
- c → ca
- ca → a → ab found ab
- ab → b → bc found bc
- bc → bca found bca

Answer = 4? Wrong!



Example

Patterns = {"a", "ab", "bc", "bca", "c", "caa"}

S = "cabca"

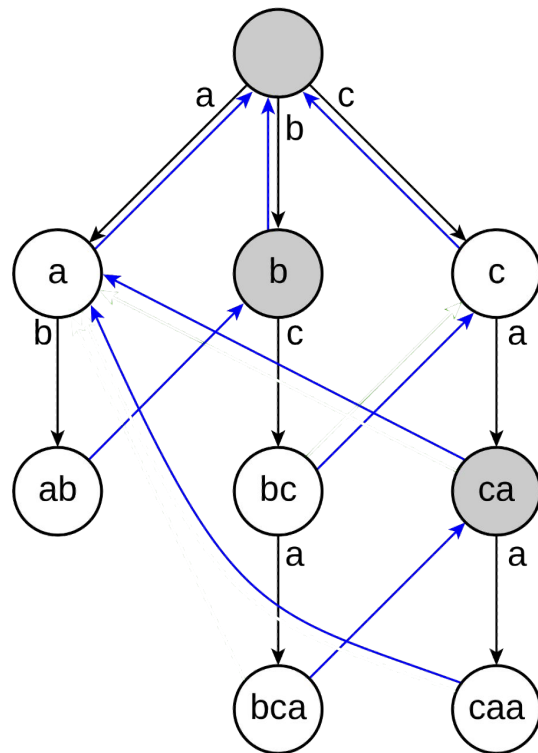
- c found c
- c → ca
- ca → a → ab found ab
- ab → b → bc found bc
- bc → bca found bca

Correct answer = 7

suffix(es) of ca: a

suffix(es) of bc: c

suffix(es) of bca: a

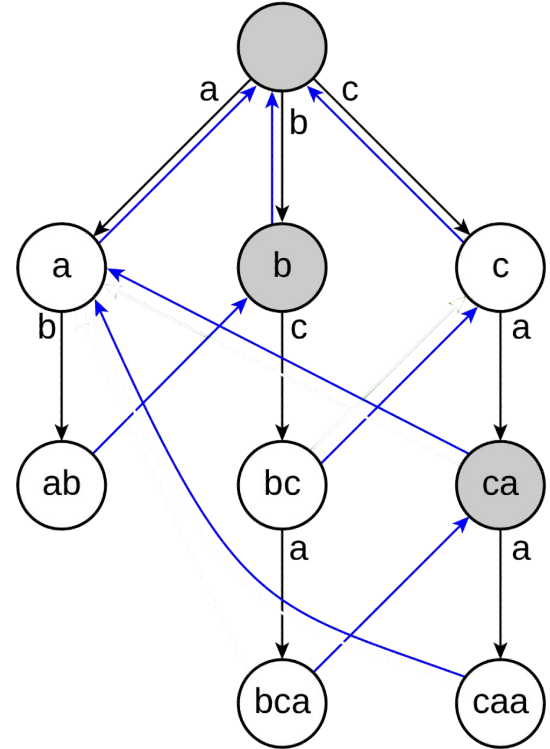


Aho-Corasick algorithm

Problem: whenever we arrive at a state, that state is not the only possible match, but its suffixes (that also end at the same position) are also potential matches

If we can reach one or more output vertices by moving along the suffix links, then there will be also a match corresponding to each found output vertex

⇒ Time Complexity: $O(|S| \times \max(|T_i|))$

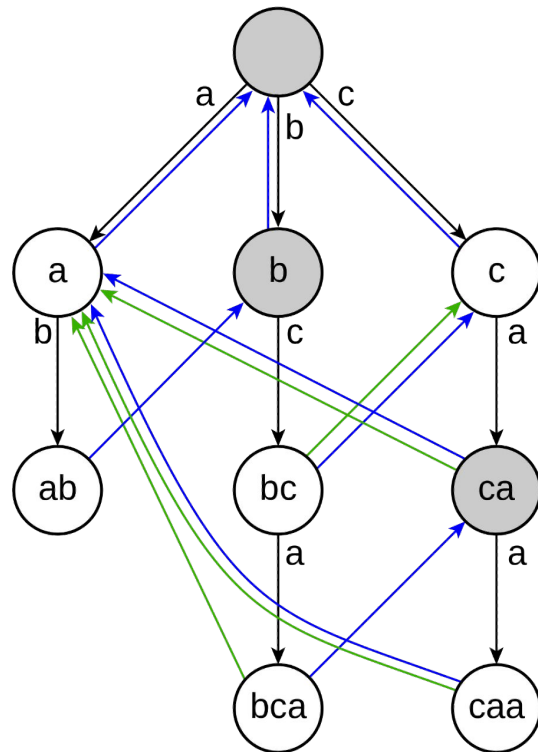


Aho-Corasick algorithm

Solution: Create **output links** that point to the nodes' longest proper suffix that is a pattern.

- They can be created along with **suffix links** lazily in $O(1)$ time during automaton building stage
- Now whenever we arrive at a state, to find all potential matches, instead of following **suffix links** we follow **output links**.

⇒ Time Complexity: $O(|S| + z)$, where z is the size of the answer, i.e. total number of occurrences
(Just count occurrence of each pattern: $O(|S| + m)$)



Application

Besides the multiple string pattern matching problem (which means you then may or may not need those **output links**), it is also often combined with **DP**

- Node on AC automaton = State in DP
- Edge on AC automaton = Transition in DP
- e.g. `for(int k = 0; k < 26; k++) dp[i + 1][δ[j][k]] += dp[i][j]`

Example 1

POJ 2778 DNA Sequence

- Given m patterns and $\Sigma = \{A, C, T, G\}$
- Find the number of length- n string such that it doesn't contain any of those patterns

$n \leq 2 \times 10^9$, $m \leq 10$, length of patterns ≤ 10

DNA Sequence

- Dangerous nodes, nodes that when you visit will become invalid, either belong to one of the **patterns**, or contains an **output link**
- Ans = Number of length- n path start at root s.t. it doesn't pass any dangerous nodes
- Use matrix to speed up

Example 2

Codeforces 1437G Death DBMS

- You are given a set of N names, where each name initially has a value of 0.
- You need to process Q queries:
 - Update the value of a specific name
 - Find the maximum value of any name that appears as a substring inside a given text S

Sum of $|Name| \leq 300,000$, $Q \leq 300,000$, Sum of $|S| \leq 300,000$

Death DBMS

- Build the AC Automaton, save the ending point of each name
- Save the update and query in corresponding ending point
- Build the tree about the fail link, dfs loop through the tree
- A query is asking the sum of the path from the node to root
- Use a segment tree on time to save the sum from root to current node
- Time complexity $O(N \log N)$

Suffix Array

Suffix Array

Sorted list of suffixes of string

- Store the indices instead of the actual suffix

Can do string matching in $O(|T| \log |S|)$

- Binary search** T on suffix array of S

Perform many different query on the string

1	attcatg\$
2	ttcatg\$
3	tcatg\$
4	catg\$
5	atg\$
6	tg\$
7	g\$
8	\$

sort the suffixes
alphabetically



the indices just
"come along for
the ride"

8	\$
5	atg\$
1	attcatg\$
4	catg\$
7	g\$
3	tcatg\$
6	tg\$
2	ttcatg\$

Suffix Array

Suffix array of “ABRACADABRA”

i	sa[i]	suffix(sa[i])
0	10	A
1	7	ABRA
2	0	ABRACADABRA
3	3	ACADABRA
4	5	ADABRA
5	8	BRA
6	1	BRACADABRA
7	4	CADABRA
8	6	DABRA
9	9	RA
10	2	RACADABRA

Suffix Array

Naive construction

- Sort every suffix
- $O(N \log N)$ comparison
- $O(N)$ per comparison
- Time complexity = $O(N^2 \log N)$

```
sort vector<pair<string, int>>
```

Suffix Array

$O(N)$ per comparison

- Too slow

We can use the technique of **doubling** to speed up the construction

Suffix Array

- Instead of comparing the whole suffix
- Comparing the **length of power of two**
- Sort $O(\log N)$ times
- Compare strings by the **rank of previous sorting**

$\text{rank}[i]$ = the rank of the first len characters of the i^{th} suffix

$\text{len} = 1, 2, 4, 8, \dots$

Suffix Array

1. Compute $\text{rank}[]$ by the first 1 character of the i^{th} suffix (ASCII)
2. Sort $\text{SA}[]$ by the first 2 character
How? By comparing elements by $\text{rank}[]$
3. Update the $\text{rank}[]$
4. Sort $\text{SA}[]$ by the first 4 character
5. So on.....

Finally, we sorted $\text{SA}[]$ by the whole suffix ($\text{len} \geq N$)

Suffix Array

Compare elements by $\text{rank}[]$

- Assume we computed $\text{rank}[]$ when $\text{len} = k$

$\text{len} = 2k$, compare i and j (suffixes start at i and j)

Compare $(\text{rank}[i], \text{rank}[i+k])$ and $(\text{rank}[j], \text{rank}[j+k])$

Suffix Array

$\text{len} = 2k$, compare i and j (suffixes start at i and j)

Compare $(\text{rank}[i], \text{rank}[i+k])$ and $(\text{rank}[j], \text{rank}[j+k])$

compare $\text{rank}[i]$ and $\text{rank}[j]$ = compare $S[i..i+k-1]$ and $S[j..j+k-1]$

compare $\text{rank}[i+k]$ and $\text{rank}[j+k]$ = compare $S[i+k..i+2k-1]$ and $S[j+k..j+2k-1]$

Suffix Array

```
bool cmpSA(int u, int v) { //comparing  $u^{\text{th}}$  and  $v^{\text{th}}$  suffix when  $\text{len} = 2k$ 
    if(RANK[u] != RANK[v]) return RANK[u] < RANK[v];
    else {
        int x, y;
        x = u + k < N ? RANK[u + k] : -1; //special handle with  $u + k \geq N$ 
        y = v + k < N ? RANK[v + k] : -1;
        return x < y;
    }
}
```

Suffix Array

SA of “ABRACADABRA”

len = 1

sa[i]	S[sa[i]]	rank[sa[i],1]
0	A	0
3	A	0
5	A	0
7	A	0
10	A	0
1	B	1
8	B	1
4	C	2
6	D	3
2	R	4
9	R	4

Suffix Array

SA of “ABRACADABRA”

len = 2

sa[i]	S[sa[i]..sa[i]+1]	rank[sa[i],1]	rank[sa[i]+1,1]
10	A	0	-1
0	AB	0	1
7	AB	0	1
3	AC	0	2
5	AD	0	3
1	BR	1	4
8	BR	1	4
4	CA	2	0
6	DA	3	0
2	RA	4	0
9	RA	4	0

Suffix Array

SA of “ABRACADABRA”

len = 2

sa[i]	S[sa[i]..sa[i]+1]	rank[sa[i],2]
10	A	0
0	AB	1
7	AB	1
3	AC	2
5	AD	3
1	BR	4
8	BR	4
4	CA	5
6	DA	6
2	RA	7
9	RA	7

Suffix Array

SA of "ABRACADABRA"

len = 4

sa[i]	S[sa[i]..sa[i]+3]	rank[sa[i],2]	rank[sa[i]+2,2]
10	A	0	-1
0	ABRA	1	7
7	ABRA	1	7
3	ACAD	2	3
5	ADAB	3	1
8	BRA	4	0
1	BRAC	4	2
4	CADA	5	6
6	DABR	6	4
9	RA	7	-1
2	RACA	7	5

Suffix Array

SA of “ABRACADABRA”

len = 4

sa[i]	S[sa[i]..sa[i]+3]	rank[sa[i],4]
10	A	0
0	ABRA	1
7	ABRA	1
3	ACAD	2
5	ADAB	3
8	BRA	4
1	BRAC	5
4	CADA	6
6	DABR	7
9	RA	8
2	RACA	9

Suffix Array

SA of “ABRACADABRA”

len = 8

sa[i]	S[sa[i]..sa[i]+7]	rank[sa[i],4]	rank[sa[i]+4,4]
10	A	0	-1
7	ABRA	1	-1
0	ABRACADA	1	6
3	ACADABRA	2	1
5	ADABRA	3	8
8	BRA	4	-1
1	BRACADAB	5	3
4	CADABRA	6	4
6	DABRA	7	0
9	RA	8	-1
2	RACADABR	9	7

Suffix Array

SA of “ABRACADABRA”

len = 8

sa[i]	S[sa[i]..sa[i]+7]	rank[sa[i],4]
10	A	0
7	ABRA	1
0	ABRACADA	2
3	ACADABRA	3
5	ADABRA	4
8	BRA	5
1	BRACADAB	6
4	CADABRA	7
6	DABRA	8
9	RA	9
2	RACADABR	10

Suffix Array

SA of "ABRACADABRA"

len = 16

i	sa[i]	suffix(sa[i])
0	10	A
1	7	ABRA
2	0	ABRACADABRA
3	3	ACADABRA
4	5	ADABRA
5	8	BRA
6	1	BRACADABRA
7	4	CADABRA
8	6	DABRA
9	9	RA
10	2	RACADABRA

Suffix Array

Observation:

- If $sa[i] = k$
 - Then $i = rank[k]$
- $\Rightarrow sa[rank[k]] = k$

i	sa[i]	suffix(sa[i])
0	10	A
1	7	ABRA
2	0	ABRACADABRA
3	3	ACADABRA
4	5	ADABRA
5	8	BRA
6	1	BRACADABRA
7	4	CADABRA
8	6	DABRA
9	9	RA
10	2	RACADABRA

Suffix Array

```
void build_SA(string s) {  
    N = s.size();  
    for (i = 0; i < N; i++) {  
        SA[i] = i;  
        RANK[i] = s[i]; // len=1, rank=ASCII  
    }  
    for (i = 1; i < N; i *= 2) {  
        sort(SA, SA + N, cmpSA);  
  
        TMP[SA[0]] = 0;  
        for (j = 1; j < N; j++)  
            TMP[SA[j]] = TMP[SA[j - 1]] + cmpSA(SA[j - 1], SA[j]);  
        for (j = 0; j < N; j++)  
            RANK[j] = TMP[j]; // updating the rank[]  
    }  
}
```

Suffix Array

No. of sorting = $O(\log N)$

No. of comparison per sorting = $O(N \log N)$

Time complexity per comparison = $O(1)$

Overall time complexity = $O(N \log^2 N)$

Space complexity = $O(N)$

Suffix Array

Observe that range of $\text{rank}[] < N$

⇒ replace `std::sort` with radix sort

$O(N \log N) \rightarrow O(N)$

Overall time complexity: $O(N \log^2 N) \rightarrow O(N \log N)$

$O(N)$ build Suffix array

- <http://gaggy.blogspot.com/2012/08/linear-time-suffix-array-dc3.html>

Suffix Array – LCP

Only having the SA is not really helpful

Calculate another array `lcp[ ]`

- `lcp[i] = longest common prefix of suffix(sa[i]) and suffix(sa[i-1])`

Longest Common Prefix

e.g. **AB**CDE **AB**EDC, `lcp = 2`

Suffix Array – LCP

$lcp[i] = \text{longest common prefix of}$
 $\text{suffix}(sa[i]) \text{ and}$
 $\text{suffix}(sa[i-1])$

	i	sa[i]	suffix(sa[i])
	0	10	A
1	1	7	ABRA
4	2	0	ABRACADABRA
1	3	3	ACADABRA
1	4	5	ADABRA
0	5	8	BRA
3	6	1	BRACADABRA
0	7	4	CADABRA
0	8	6	DABRA
0	9	9	RA
2	10	2	RACADABRA

Suffix Array – LCP

With `lcp[ ]`, we can calculate `lcp(suffix(i), suffix(j))` with `lcp[ ]`

- or LCP of any two substring

Suffix Array – LCP

Example:

$\text{lcp}(\text{"ABRA"}, \text{"ADABRA"}) = 1$

i	sa[i]	suffix(sa[i])	LCP[i]
0	10	A	0
1	7	ABRA	1
2	0	ABRACADABRA	4
3	3	ACADABRA	1
4	5	ADABRA	1
5	8	BRA	0
6	1	BRACADABRA	3
7	4	CADABRA	0
8	6	DABRA	0
9	9	RA	0
10	2	RACADABRA	2

Suffix Array – LCP

Example:

$\text{lcp}(\text{"ABRA"}, \text{"ADABRA"}) = 1$

Since strings in Suffix Array are **sorted** (in lexicographic/dictionary order), it is guaranteed that

$\text{lcp}(\text{"ABRA"}, \text{"ADABRA"})$
 $= \min(\{\text{lcp}[2], \text{lcp}[3], \text{lcp}[4]\})$

i	sa[i]	suffix(sa[i])	LCP[i]
0	10	A	0
1	7	ABRA	1
2	0	ABRACADABRA	4
3	3	ACADABRA	1
4	5	ADABRA	1
5	8	BRA	0
6	1	BRACADABRA	3
7	4	CADABRA	0
8	6	DABRA	0
9	9	RA	0
10	2	RACADABRA	2

Suffix Array – LCP

With `lcp[ ]`, we can calculate `lcp(suffix(i), suffix(j))` with `lcp[ ]`

- or LCP of any two substring

$\text{lcp}(\text{suffix}(i), \text{suffix}(j)) = \min\{\text{lcp}[\text{rank}[i]+1], \text{lcp}[\text{rank}[i]+2], \dots, \text{lcp}[\text{rank}[j]]\}$

- Assume $\text{rank}[i] \leq \text{rank}[j]$
- = minimum value in `lcp[rank[i] + 1..rank[j]]`

Use data structure for RMQ

- Segment tree, Sparse Table

Suffix Array – LCP

Observation: $sa[rank[i]] = i$

$$\begin{aligned} & lcp[rank[i]] \\ &= lcp(suffix(sa[rank[i]]), suffix(sa[rank[i]-1])) \\ &= lcp(suffix(i), suffix(sa[rank[i]-1])) \end{aligned}$$

$$\begin{aligned} & lcp[rank[i+1]] \\ &= lcp(suffix(sa[rank[i+1]]), suffix(sa[rank[i+1]-1])) \\ &= lcp(suffix(i+1), suffix(sa[rank[i+1]-1])) \end{aligned}$$

Suffix Array – LCP

We don't know relationship between adjacent `suffix(sa[i])`, but we do know the relationship between adjacent `suffix(i)`

$$\begin{aligned} & \text{lcp}[\text{rank}[i]] \\ &= \text{lcp}(\text{suffix}(\text{sa}[\text{rank}[i]]), \text{suffix}(\text{sa}[\text{rank}[i]-1])) \\ &= \text{lcp}(\boxed{\text{suffix}(i)}, \text{suffix}(\text{sa}[\text{rank}[i]-1])) \end{aligned}$$

$$\begin{aligned} & \text{lcp}[\text{rank}[i+1]] \\ &= \text{lcp}(\text{suffix}(\text{sa}[\text{rank}[i+1]]), \text{suffix}(\text{sa}[\text{rank}[i+1]-1])) \\ &= \text{lcp}(\boxed{\text{suffix}(i+1)}, \text{suffix}(\text{sa}[\text{rank}[i+1]-1])) \end{aligned}$$

Suffix Array – LCP

Let $\text{lcp}[\text{rank}[i]] = \text{lcp}(\text{suffix}(i), \text{suffix}(\text{sa}[\text{rank}[i]-1])) = h$.

$\Rightarrow$ $\text{suffix}(i)$ is right after $\text{suffix}(\text{sa}[\text{rank}[i]-1])$ in the SA, their $\text{lcp}[\text{rank}[i]] = h$

$\Rightarrow$ $\text{suffix}(i)$ is after $\text{suffix}(\text{sa}[\text{rank}[i]-1])$ in the SA, their $\text{lcp}[\text{rank}[i]] \geq h$

$\Rightarrow$ $\text{suffix}(i+1)$ is after $\text{suffix}(\text{sa}[\text{rank}[i]-1]+1)$ in the SA, their $\text{lcp}[\text{rank}[i+1]] \geq h-1$

Consider $\text{lcp}[\text{rank}[i+1]] = \text{lcp}(\text{suffix}(i+1), \text{suffix}(\text{sa}[\text{rank}[i+1]-1]))$

$\Rightarrow$ $\text{suffix}(i+1)$ is right after $\text{suffix}(\text{sa}[\text{rank}[i+1]-1])$ in the SA

$\Rightarrow$ $\text{suffix}(\text{sa}[\text{rank}[i+1]-1])$ is between $\text{suffix}(i+1)$ and $\text{suffix}(\text{sa}[\text{rank}[i]-1]+1)$ in SA

$\Rightarrow \text{lcp}[\text{rank}[i+1]] \geq h-1$

Suffix Array – LCP

We have proved:

- If $\text{lcp}[\text{rank}[i]] = h$
- Then $\text{lcp}[\text{rank}[i + 1]] \geq h - 1$

Calculate $\text{lcp}[\]$ in the order of $\text{rank}[0], \text{rank}[1], \text{rank}[2], \dots$

- Position that contains 0, 1, 2, in the SA[] array

Suffix Array – LCP

e.g. $S = \text{"BCEABCDABCEB"}$

$SA[\text{rank}[7]] = \text{ABCEB}$

$\text{lcp}[\text{rank}[7]] = 3$, i.e. there exist suffix = **ABCD**.....

$SA[\text{rank}[7 + 1]] = \text{BCEB}$

$\text{lcp}[\text{rank}[7 + 1]] \geq 2$, i.e. there exist suffix = **BCD**.....

p.s. $\text{lcp}[\text{rank}[7+1]]$ may not be 2, in this case $\text{lcp}[\text{rank}[7+1]] = 3$

Suffix Array – LCP

```
for (i = 0; i < N; i++) {  
    if (RANK[i] == 0) continue;  
    int t = SA[RANK[i] - 1];  
    h = max(0, h - 1);  
    while (i + h < N && t + h < N) {  
        if (s[i + h] != s[t + h]) break;  
        h++;  
    }  
    LCP[RANK[i]] = h;  
}
```

Suffix Array – LCP

For each loop

- h will decrease 1
- h won't exceed N

Time complexity = $O(N)$

```
for (i = 0; i < N; i++) {  
    if (RANK[i] == 0) continue;  
    int t = SA[RANK[i] - 1];  
    h = max(0, h - 1);  
    while (i + h < N && t + h < N) {  
        if (s[i + h] != s[t + h]) break;  
        h++;  
    }  
    LCP[RANK[i]] = h;  
}
```

Application Example

Given a string S , find the number of distinct substrings

ababa

Ans = 9

$\{ "a", "b", "ab", "ba", "aba", "bab", "abab", "baba", "ababa" \}$

Exhaust all substrings, count the number of distinct substrings

Time complexity: $O(|S|^2)$

Application Example

Consider prefix of each suffix

ababa

{ 'a' }

ans = 1

i	sa[i]	suffix(sa[i])	LCP[i]
0	4	a	0
1	2	aba	1
2	0	ababa	3
3	3	ba	0
4	1	baba	2

Application Example

Consider prefix of each suffix

ababa

{ 'a', 'ab', 'aba' }

$LCP[i] = 1$

- don't add 'a'

ans = 3

i	sa[i]	suffix(sa[i])	LCP[i]
0	4	a	0
1	2	aba	1
2	0	ababa	3
3	3	ba	0
4	1	baba	2

Application Example

Consider prefix of each suffix

ababa

{ 'a', 'ab', 'aba', 'abab', 'ababa' }

$LCP[i] = 3$

- don't add 'a', 'ab', 'aba'

ans = 5

i	sa[i]	suffix(sa[i])	LCP[i]
0	4	a	0
1	2	aba	1
2	0	ababa	3
3	3	ba	0
4	1	baba	2

Application Example

Consider prefix of each suffix

ababa

{ 'a', 'ab', 'aba', 'abab', 'ababa',
'b', 'ba' }

$LCP[i] = 0$

ans = 7

i	sa[i]	suffix(sa[i])	LCP[i]
0	4	a	0
1	2	aba	1
2	0	ababa	3
3	3	ba	0
4	1	baba	2

Application Example

Consider prefix of each suffix

ababa

{ 'a', 'ab', 'aba', 'abab', 'ababa',
'b', 'ba', 'bab', 'baba' }

$LCP[i] = 2$

- don't add 'b', 'ba'

ans = 9

i	sa[i]	suffix(sa[i])	LCP[i]
0	4	a	0
1	2	aba	1
2	0	ababa	3
3	3	ba	0
4	1	baba	2

Application Example

```
int res = n - sa[0];  
for (int i = 1; i < n; i++) res += (n - sa[i]) - LCP[i];  
return res;
```

Time complexity: $O(N)$

Problem: <https://www.spoj.com/problems/DISUBSTR/>

Example

Codeforces 1202E You Are Given Some Strings...

- You are given a string T and N strings S_1 to S_N
- Let $f(x, y)$ be number of occurrences of y in x
- Find $f(T, S_x + S_y)$ for all $1 \leq x, y \leq N$

$|T| \leq 200,000$, sum of $|S_i| \leq 200,000$

You Are Given Some Strings...

Exhaust the “splitting position” of $S_x + S_y$ in t

For each “splitting position” i , increase answer by number of y such that $S_y = T[i + 1 \dots i + |S_y|] \times$ number of x such that $S_x = T[i - |S_x| + 1 \dots i]$

To calculate the first part, put all T as well as all S into a suffix array

Iterate through y , string S_y can contribute to splitting position i iff $\text{LCP}(T[i \dots |T|], S_y) = |S_y|$

The valid i forms a continuous range on suffix array

Reverse all S and T for the second part

Suffix Array – Application

Other application

- longest repeated substring
 - overlap / non-overlap
- longest common substring
- longest palindromic substring

Useful application: binary search on answer

- Group indices by lcp

Exercise

KMP

- HKOJ 01002
- HKOJ M0932
- HKOJ P002
- NOI14 動物園

Aho-Corasick algorithm

- HKOJ M2441
- HKOJ M2641
- HDU 2222
- HDU 3065
- POJ 2778

Suffix array

- HKOJ M1762
- POJ 2774
- NOI15 品酒大會

Others

Suffix Automaton

- Minimal DFA that accepts all suffixes of a string
- Linear construction
- Popular in Competitive Programming
- [Suffix Automaton - Algorithms for Competitive Programming \(cp-algorithms.com\)](http://cp-algorithms.com)

Others

Manacher algorithm

- Finding all palindromic substrings in linear time
- [Manacher's Algorithm - Finding all sub-palindromes in \$O\(N\)\$](#)

Palindrome Tree / EERTREE

- [Palindromic tree | Blog | Adilet.org](#)

Z-algorithm / Boyer-Moore algorithm

- Linear time string matching
- For most of the time, just use KMP

Others

You probably won't see these algorithms in competitive programming

Suffix Tree

- A compressed suffix trie
- Constructed in linear time by Ukkonen's algorithm
- Very powerful, capable of doing everything we discussed and more
- Suffix array can be obtained by a DFS on the suffix tree
- Can also be constructed by building a Cartesian tree on suffix array

Suffix Tray

- Combination of suffix array and suffix tree

More on these data structures: [MIT OCW - Advanced Data Structures](#)

Q & A