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

String Algorithms

Vincent Chiu {VCLH}

2024-06-28

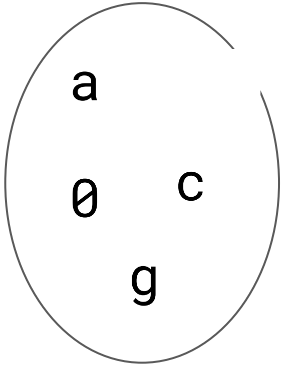
Table of Contents

Introduction

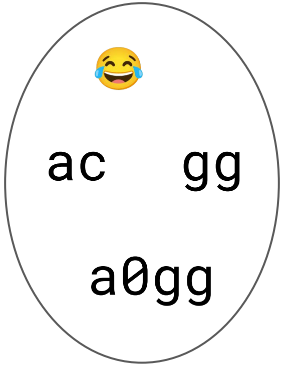
An **alphabet** ∑ is a **finite** nonempty set of **symbols/characters**.

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

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



∑



some strings over ∑

ASCII Code

American **S**tandard **C**ode for **I**nformation **I**nterchange

- Character encoding
- 7 bits / 8 bits (1 byte)
- 0-127 / 0-255

- `'\0'` = 0, `'\n'` = 10 `' '` = 32
- Digits: `'0'` = 48+0 `'9'` = 48+9
- UpperCase: `'A'` = 64+1 `'Z'` = 64+26
- LowerCase: `'a'` = 96+1 `'z'` = 96+26

$b_7 b_6 b_5$ →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits b_4	b_3	b_2	b_1	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	—	o	DEL

String I/O and Functions

- scanf, printf
- fgets, puts
- getline
- <cstring>
- <string>
- See String Algorithms (2018) and Programming using C++ (2022) for more details

Common String Terminology

Concatenation

- Addition in string

For strings `s` and `u`,

- `t` = 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 A**BCD**E
- `s` is substring of `t` if there exist strings `u` and `v` such that `t = usv`.
- `s = t[i..j]` (inclusive), `0≤i≤j≤|t|-1`

Subsequence:

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

Common String Terminology

Prefix:

- A **substring** that starts from the **beginning** of a string
- e.g. **A**BCDE, **ABC**DE, **ABCDE**
- `s` is prefix of `t` if there exist string `v` such that `t = sv`.
- `s = Prefix(t, k) = t[0..k-1], 0≤k≤|t|`

Suffix:

- A **substring** that ends with the **end** of a string
- e.g. ABCD**E**, AB**CDE**, **ABCDE**
- `s` is suffix of `t` if there exist string `u` such that `t = us`.
- `s = Suffix(t, k) = t[|t|-k..|t|-1], 0≤k≤|t|`

Common String Terminology

- A **<u>proper</u>** 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: $t = u\underline{\mathbf{s}v}$
- Every substring is a suffix of prefix: $t = \underline{u\mathbf{s}}v$
- 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?

String Matching

String Matching

Exact String Matching:

- Given a string **S** and a string/**pattern** T,
- Is **T** a substring of **S**?
- If so, how many times does **T** appear?
- ⇒ Find the positions of all occurrences of the pattern **T** in **S**.

Approximate String Matching:

- Same problem but with error tolerance
- (Out of scope)

Naive String Matching

Brute force

```
vector<int> match(string S, string T) {
  vector<int> ans;
  for(int i = 0; i <= |S|-|T|, i++)
    if(S[i..i+|T|-1] == T)
      ans.push_back(i);
  return ans;
}
```

Naive String Matching

Brute force

```
vector<int> match(string S, string T) {
  vector<int> ans;
  for(int i = 0; i <= |S|-|T|, i++)
    if(S[i..i+|T|-1] == T)
      ans.push_back(i);
  return ans;
}
```

```
for(int j = 0; j < |T|; j++)
  if(S[i+j] != T[j]) break;
```

Naive String Matching

Brute force: O(|S||T|). Too slow! How to optimize?

```
vector<int> match(string S, string T) {
  vector<int> ans;
  for(int i = 0; i <= |S|-|T|, i++)
    if(S[i..i+|T|-1] == T)
      ans.push_back(i);
  return ans;
}
```

```
for(int j = 0; j < |T|; j++)
  if(S[i+j] != T[j]) break;
```

Hashing (Rabin-Karp)

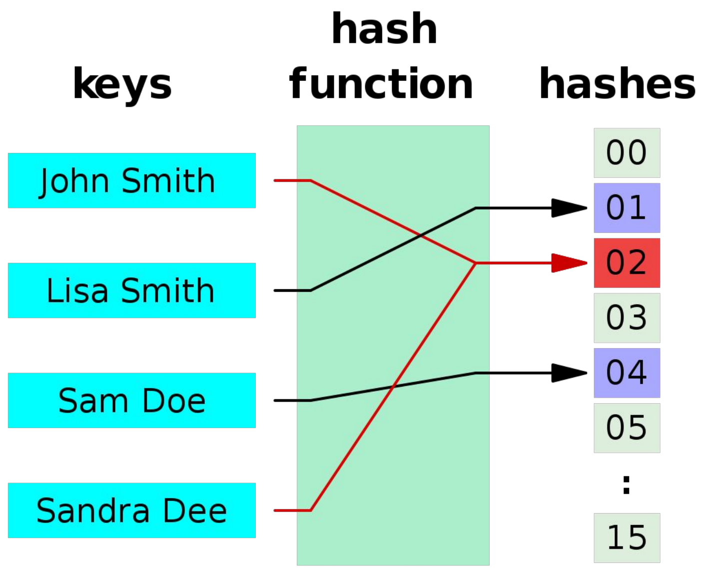
Recall: Hashing

Hashing in number

- number → number
- Hash table
- Taught in Data Structures (II)

Hashing in string

- string → number
- compare number instead of string



Hashing

What's wrong with the naive approach?

```
vector<int> match(string S, string T) {
  vector<int> ans;
  for(int i = 0; i <= |S|-|T|, i++)
    if(S[i..i+|T|-1] == T)
      ans.push_back(i);
  return ans;
}
```

```
for(int j = 0; j < |T|; j++)
  if(S[i+j] != T[j]) break;
```

Hashing

What’s wrong with the naive approach?

```
vector<int> match(string S, string T) {
  vector<int> ans;
  for(int i = 0; i <= |S|-|T|, i++)
    if(S[i..i+|T|-1] == T)
      ans.push_back(i);
  return ans;
}
```

Comparing S[i..i+|T|-1] and T is linear!

```
for(int j = 0; j < |T|; j++)
  if(S[i+j] != T[j]) break;
```

Hashing

Solution: Compare the **hashed** values of the string instead

Comparison is O(1) if the hash function returns an integer.

A common hash function is **polynomial rolling hash**:

$$\text{hash}(S) = (c_0 a^{k-1} + c_1 a^{k-2} + \ldots + c_{k-2} a + c_{k-1}) \bmod p$$

- Only uses multiplications and additions
- **p**: a large constant, usually prime, e.g. 1e9+7 or 1e9+9
- **a**: arbitrary small constant, desirably larger than alphabet size, e.g. 256 / 26 / 37 / 53
- **c_i**: ASCII value of i^{th} character / A=0, B=1, ..., Z=25

Hashing

$$hash(S) = (c_0a^{k-1} + c_1a^{k-2} + \ldots + c_{k-2}a + c_{k-1}) \bmod p$$

"HKOI"

a = 26, p = 64997

c_0 = 7, c_1 = 10, c_2 = 14, c_3 = 8

- Subtract 'A'

$$\begin{aligned} H &= (7\times26^3 + 10\times26^2 + 14\times26 + 8) \bmod 64997 \\ &= 130164 \bmod 64997 \\ &= 170 \end{aligned}$$

Hashing

```
vector<int> match(string S, string T) {
  long long T_hash = hash(T);
  vector<int> ans;
  for(int i = 0; i <= |S|-|T|, i++)
    if(hash(S[i..i+|T|-1]) == T_hash)
      ans.push_back(i);
  return ans;
}
```

Hashing

```
vector<int> match(string S, string T) {
  long long T_hash = hash(T);
  vector<int> ans;
  for(int i = 0; i <= |S|-|T|, i++)
    if(hash(S[i..i+|T|-1]) == T_hash)
      ans.push_back(i);
  return ans;
}
```

Sounds fishy? Isn't the hash function O(|T|) as well?

Hashing

Polynomial rolling hash: **Sliding window**

- $hash(S[i..i+|T|-1]) = (c_i a^{|T|-1} + c_{i+1} a^{|T|-2} + \ldots + c_{i+|T|-2} a + c_{i+|T|-1}) \bmod p$
- $hash(S[i+1..i+|T|]) = (c_{i+1} a^{|T|-1} + c_{i+2} a^{|T|-2} + \ldots + c_{i+|T|-1} a + c_{i+|T|}) \bmod p$

$$hash(S[i+1..i+|T|]) = hash(S[i..i+|T|-1]) \times a - S[i] \times a^{|T|} + S[i+|T|] \quad (\bmod p)$$

- Handle negative number → (x + p - y % p) mod p

Hashing

Polynomial rolling hash: **Sliding window**

- The “window” moves through the input, and we evaluate the hash value of the substring within the window.
- When the window “slides”, the old value is removed from the window, and the new value added to the window; hash value updated accordingly

- ⇒ The hash value of adjacent substrings can be updated in O(1)

- There you have it: Rabin-Karp algorithm for string matching

Collision

$$hash(S) = (c_0a^{k-1} + c_1a^{k-2} + \dots + c_{k-2}a + c_{k-1}) \bmod p$$

"GO"

a = 26, p = 64997

c_0 = 6, c_1 = 14

- Subtract 'A'

H = (6×26+14) mod 64997
 = 170 mod 64997
 = 170

Collision

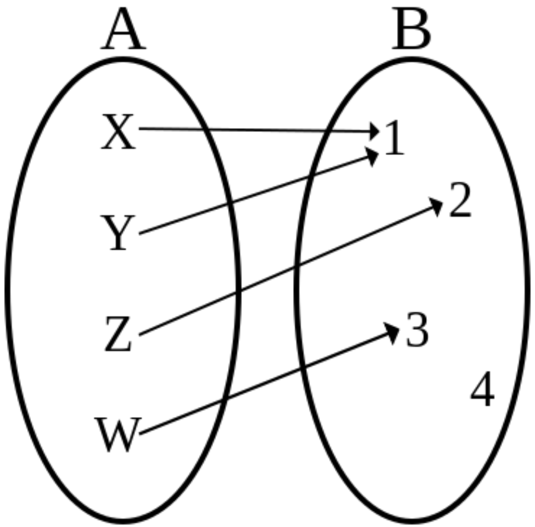
`H("HKOI") = 170 = H("GO")`

`"HKOI" ≠ "GO"`

collision occurs

Hash function is <u>not an injective function</u>

One hash value may represents multiple strings



Collision

Solution 1: pick better constants a and p to reduce the probability of collision

- p → prime number
- $a > \max(c_i)$

Solution 2 : Double hashing

- Use two pair of a and p, compare two hash values

Solution 3 : Give up hashing

- Use exact algorithm, e.g. KMP algorithm (to be covered later)

Example 1

HKOJ 01002 A Counting Problem

Given string S and T, find number of occurrences of string T in string S

- Allow overlap

```
abcdefabcghiabcabcjklmnlabcw
abc
```

```
Ans = 5
```

A Counting Problem

HKOJ 01002 A Counting Problem

Idea: compare T to each substring of S with length |T|

Compare character by character: $O(|S| \times |T|)$

Compare by hash values: $O(|S| + |T|)$

Example 2

HKOJ M0932 String Rotation

Given string S and T, find the number of rotation required so that string S' = T
|S| = |T|

ABCDE → BCDEA → CDEAB → DEABC → EABCD
EABCD

Ans = 4

String Rotation

HKOJ M0932 String Rotation

Idea: compare T to each rotation of S

Compare character by character: $O(|S|^2)$
Compare by hash values using sliding windows: $O(|S|)$

Hashing

In contest, problem setter may prepare some anti hash test

- https://codeforces.com/blog/entry/4898
- receive WA
- depends on the value of p (modules)

Use different choices of p

Double hashing

Hashing with partial sum

What if we need to find the hash value of an arbitrary substring?

- This appears more common than sliding window

Similar idea to get hash values of substring in O(1): partial hash sum array

$$hash(S[i..j]) = hash(S[0..j]) - hash(S[0..i-1]) \times a^{j-i+1} \pmod{p}$$

Hash table

When you have the polynomial hash value computed, it may be tempting to use them with std::unorderd_map directly

- Don’t do this unless necessary!!!
- E.g. Counting distinct substring of fixed length

People have made better hash function and hash tables, e.g.

1. std::unordered_map / std::unordered_set
2. Policy-based Data Structures: Codeforces blog
3. Swiss table by Google
4. F14 hash table by Facebook

Hash table – C++ Library

- std::hash
- std::unordered_map / std::unordered_set in C++ implements hash table
- Support insert, delete, query exact operations in O(1)
- Provides a default hash function for basic data types and string
    - Can ignore hash collision
    - Rehashing when needed
- Supported from C++11 and onwards

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 π[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, π[0] = 0

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

Prefix Function – Example

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

S = “ABABACB”

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

Prefix Function – Example

i = 1
"AB"
AB ≠ AB

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

S = "ABABACB"

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

Prefix Function – Example

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

S = "ABABACB"

i = 2
"ABA"
ABA ≠ ABA
ABA = ABA

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

Prefix Function – Example

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

S = “ABABACB”

i = 3

“ABAB”

ABAB ≠ ABAB

ABAB = ABAB

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

Prefix Function – Example

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

S = “ABABACB”

i = 4
“ABABA”
ABABA ≠ ABABA
ABABA = ABABA

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

Prefix Function – Example

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

S = "ABABACB"

i = 5
"ABABAC"
ABABAC ≠ ABABAC
ABABAC ≠ ABABAC
ABABAC ≠ ABABAC
ABABAC ≠ ABABAC
ABABAC ≠ ABABAC

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

Prefix Function – Example

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

S = "ABABACB"

i = 6
"ABABACB"

ABABACB ≠ ABABACB
ABABACB ≠ ABABACB
ABABACB ≠ ABABACB
ABABACB ≠ ABABACB
ABABACB ≠ ABABACB
ABABACB ≠ 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 × $O(|S|)$ 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 $\pi[i] \geq 1$,

$$\begin{array}{l} 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad i \\ \underbrace{\text{XXXXXXX}}_{\pi[i]}\text{XXXXXX}\underbrace{\text{XXXXXXX}}_{\pi[i]} \end{array}$$

Observation 1

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

	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 $\pi[i] \ge 1$,

```
0                   i
XXXXXXXXXXXXXXXXXXXX
```

Observation 1

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

	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 $\pi[i] \ge 1$,

0 i
XXXXXXXXXXXXXXXXXXXX

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

Observation 1

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

	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 $\pi[i] \ge 1$,

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

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

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 × $O(|S|)$ 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.
- ⇒ We only need to perform at most 2|S| string comparisons.

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

Observation 2

We want to get rid of the string comparisons.

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

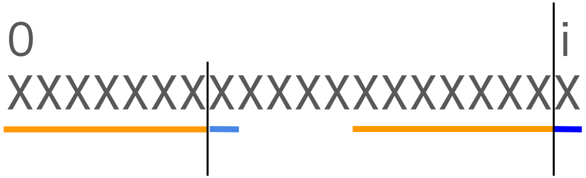
If $S[\pi[i-1]] == S[i]$, then we can say with certainty that $\pi[i] = \pi[i-1] + 1$.

Observation 2

We want to get rid of the string comparisons.

In Observation 1, we have π[i] ≤ π[i-1] + 1.

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

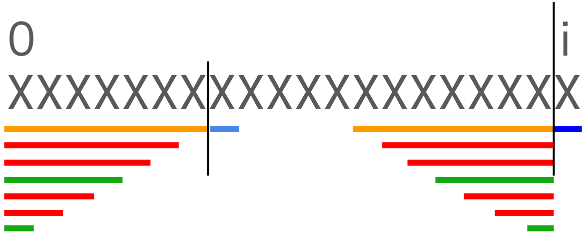


What if S[π[i-1]] ≠ S[i]? What should we try next?

Observation 2

Case S[π[i-1]] ≠ 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.



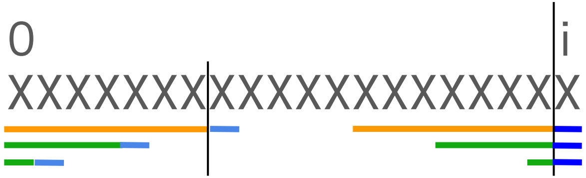
Observation 2

Case S[π[i-1]] ≠ 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.

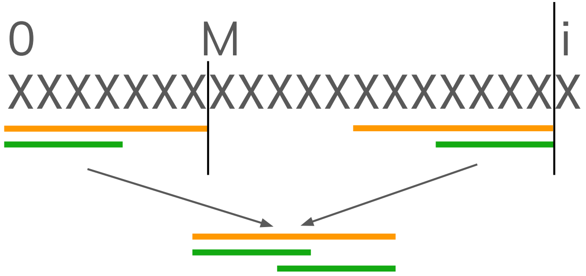
- The longest such prefix/suffix has length π[i-1].
- What is the length of the 2nd longest such prefix/suffix?
- More generally, after considering current prefix/suffix of length M, what is the **next longest** prefix/suffix?

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 = π[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 × 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

Example 3

HKOJ P002 Power Strings

Define u*v 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.

“aaaa” = $“a”^4$
“ababab” = $“ab”^3$

Power Strings

HKOJ P002 Power Strings

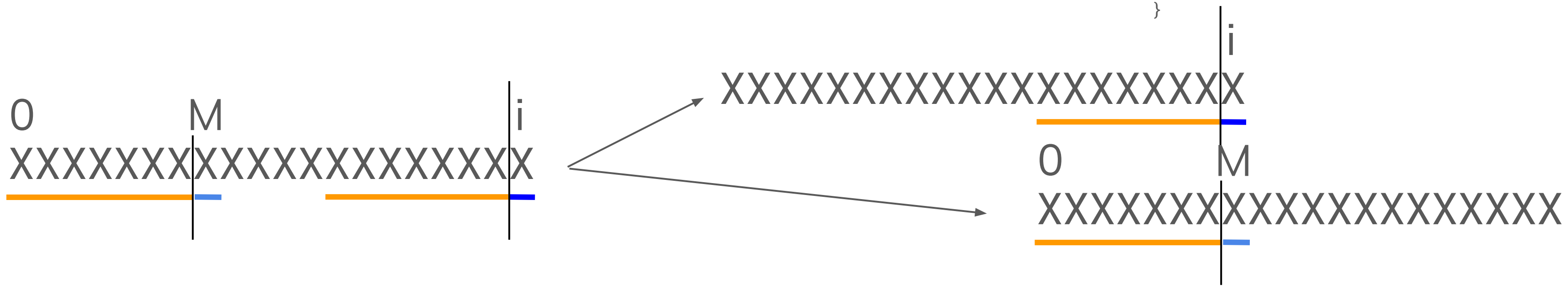
Idea:

ABCABCABCABCABC   ABCABCABCABCAB

Let’s look at the demonstration one more time

```
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;
}
```

S = “ABABACB”



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

Let’s look at the demonstration one more time

S = “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 one more time

```
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;
}
```

S = “ABABACB”

```
i=1     ABABACB
M=0      ABABACB
```

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

Let's look at the demonstration one more time

```
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;
}
```

S = "ABABACB"

i=2 ABABACB

M=0 ABABACB

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

Let’s look at the demonstration one more time

S = “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=3    ABABACB
M=1      ABABACB
```

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

Let’s look at the demonstration one more time

```
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;
}
```

S = “ABABACB”

```
i=4    ABABACB
M=2      ABABACB
```

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

Let’s look at the demonstration one more time

```
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;
}
```

S = “ABABACB”

```
i=5      ABABACB
M=3        ABABACB
```

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

Let's look at the demonstration one more time

S = "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=1         ABABACB
```

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

Let’s look at the demonstration one more time

S = “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 one more time

```
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;
}
```

S = “ABABACB”

```
i=6    ABABACB
M=0          ABABACB
```

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

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 = “ABABABABACB”

T = “ABABACB”

```
ABABABABACB
ABABACB
```

Example

S = “ABABABABACB”

T = “ABABACB”

```
ABABABABACB
ABABACB
```

Pattern not found at index 0

Where should we look at next?

Example

S = “ABABABABACB”

T = “ABABACB”

```
ABABABABACB
  ABABACB
```

“Sliding window” in hash…

Example

S = "ABABABABACB"

T = "ABABACB"

ABABABABACB

ABABACB

"Sliding window" in hash...

KMP Algorithm

S = “ABABABABACB”

T = “ABABACB”

ABABABABACB

ABABACB

KMP Algorithm

S = “ABABABABACB”

T = “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 = “ABABABABACB”

T = “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 = “ABABABABACB”

T = “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 = “ABABABABACB”

T = “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 = “ABABABABACB”

T = “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 = “ABABABABACB”

T = “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 = “ABABABABACB”

T = “ABABACB”

ABABACB#ABABABABACB

ABABACB#ABABABABACB

⇒ π[18] = 7

⇒ Position of occurrence of T 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 + “#” + S)

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

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 $\pi[i] \le |T|$, 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

Take a break

Trie (Prefix Tree)

Trie

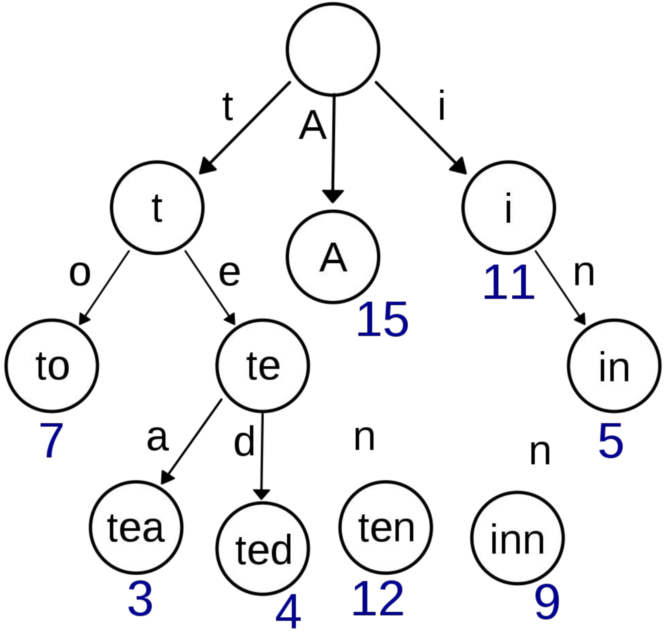
Not a formal word

- come from the word re**trie**val
- Pronounce as “try”

Tree data structure

Dictionary of a set of strings

- support quick insert and lookup
- searching/counting strings/prefixes

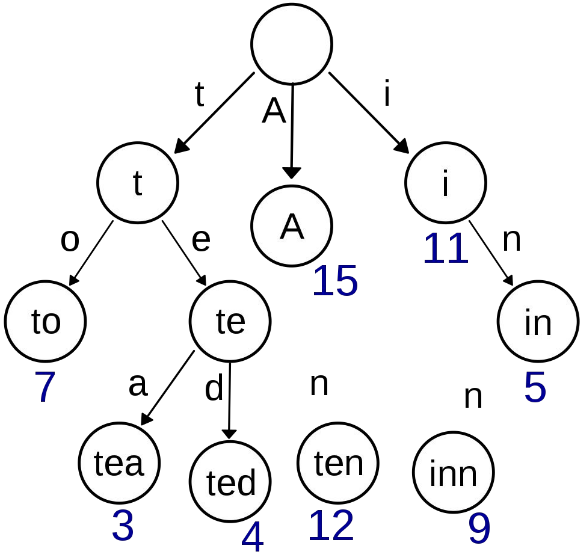


Trie

Edge: addition of a character

Node: a word / prefix of a string

- obtained by concatenating characters from root to this node



Usually, we

- Store a boolean, indicating whether this node represent **the end of a word**
- Store a integer to represent **frequency** is duplicated string is allowed

Trie

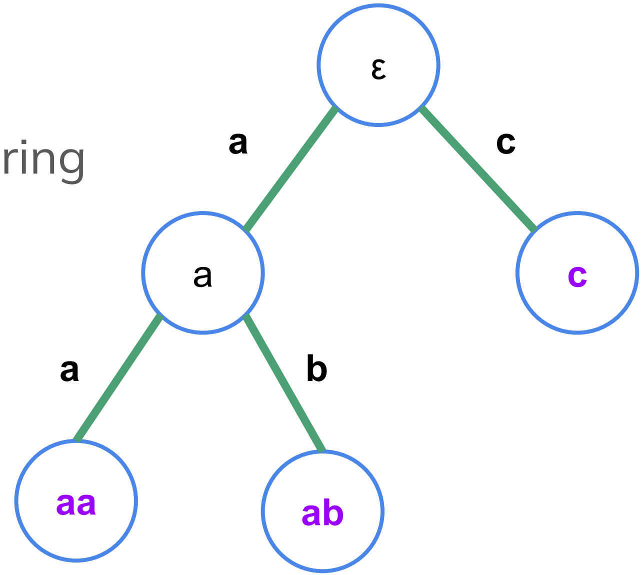
```
struct TrieNode {
  vector<int> child;    // point to each alphabet
  bool isWord = false;
  TrieNode() : child(ALPHABET_SIZE, -1) {}
};
vector<TrieNode> trie(1);
```

- ALPHABET_SIZE depends of your input
  - lowercase letters -> 26   ascii -> 128
- We assume the input is all lowercase letters in the following slides

Trie

Trie of {“aa”, “ab”, “c”}

- Nodes in **purple** indicate the end of a string
- The empty string (root) is denoted by ε
- Pointers to null are omitted
- (e.g. there is no string ”ca”)



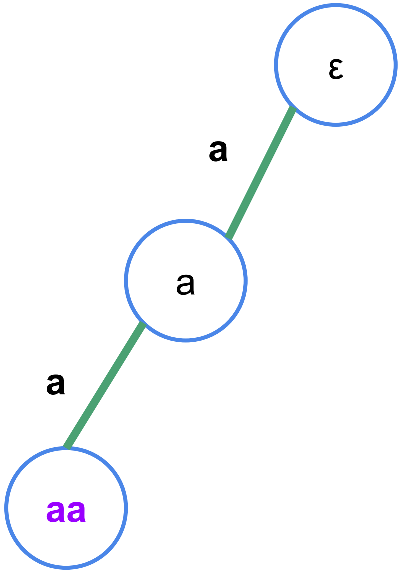
Trie – Insertion

```
void insert(string S) {
  int cur = 0;
  for(int i = 0; i < |S|; i++) {
    if(trie[cur].child[S[i]-'a'] == -1) {
      trie[cur].child[S[i]-'a'] = trie.size();
      trie.emplace_back(args…);
    }
    cur = trie[cur].child[S[i]-'a'];
  }
  trie[cur].isWord = true;
}
```

Similar implementation
for delete operation

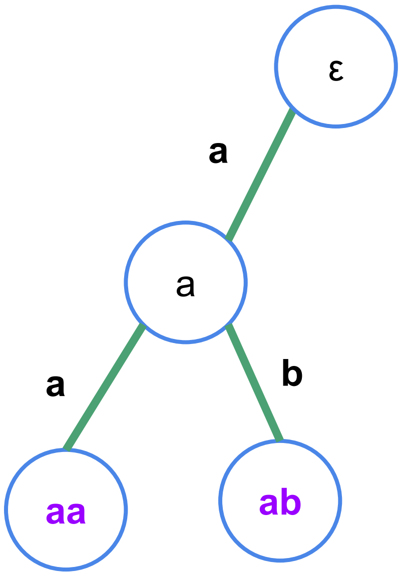
Trie – Insertion

`insert("aa")`



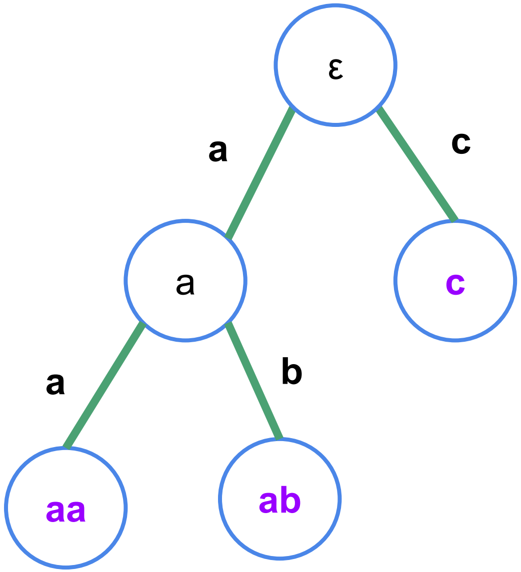
Trie – Insertion

insert("ab")



Trie – Insertion

```
insert(“c”)
```



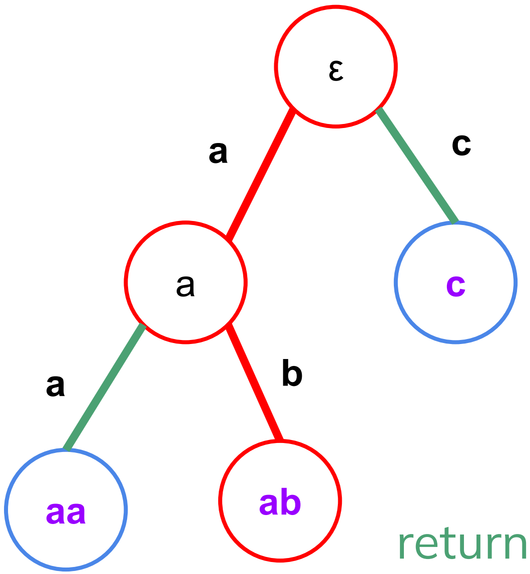
Trie – Lookup

```
bool lookup(string S) {
  int cur = 0;
  for(int i = 0; i < |S|; i++) {
    if(trie[cur].child[S[i]-'a'] == -1)
      return false;
    cur = trie[cur].child[S[i]-'a'];
  }
  return trie[cur].isWord;
}
```

Can be easily modified to search a prefix instead of the entire string

Trie – Lookup

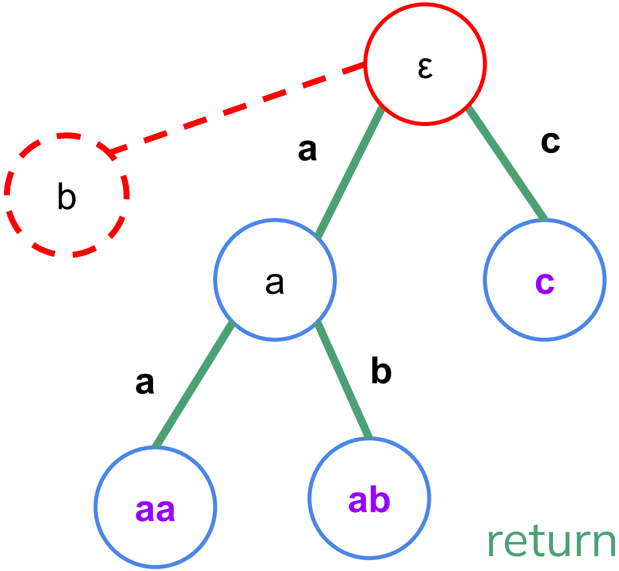
`lookup(“ab”)`



return true

Trie – Lookup

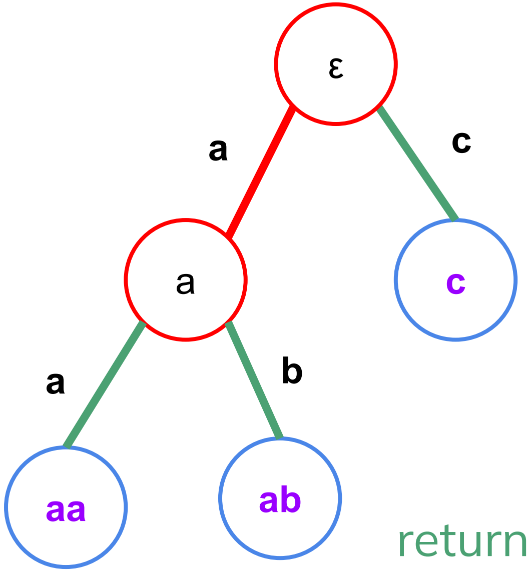
`lookup(“b”)`



return false

Trie – Lookup

lookup(“a”)



return false

Trie – Analysis

N: total length M: length of the target string

Time Complexity:

- Insert: O(M)
- Delete: O(M)
- Searching/Counting: O(M)
- Sorting: O(N)
- All of them are DFS

Memory Complexity: O(ALPHABET_SIZE × N)

Application

Commonly used with greedy in string problems

e.g. MAX-XOR problem

Practice Tasks:

1. CF 706D - Vasiliy's Multiset (MAX-XOR)
2. CF 455B - A lot of games
3. HKOI Judge I0811 - Printer

Example 4: MAX-XOR problem

Statement:

- Given **n** integers {**a**[1], **a**[2], ... **a**[n]} where 0 <= **a**[i] <= 2^m
- For a given **x**, find max(**x** xor **y** for **y** in **a**)

Sample:

a = {0101, 1100, 0011, 0110}

x = 1011

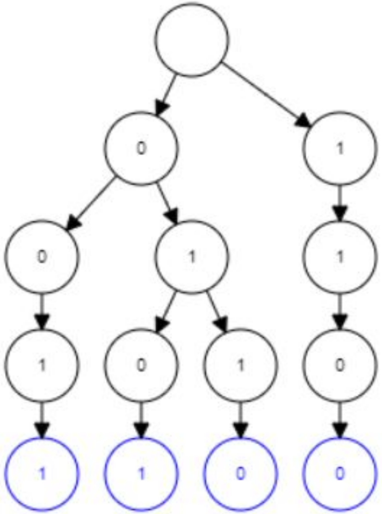
* All numbers in binary

MAX-XOR solution

Insert all integers in **a** as a binary string from the most significant bit

- Right figure:
  Trie for **a** = {0101, 1100, 0011, 0110}

- Query:
  Idea is to greedily traverse in the direction of the other bit

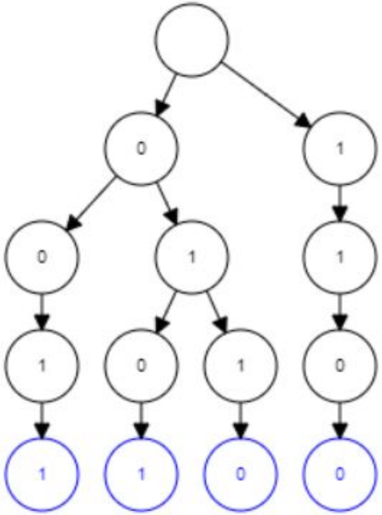


MAX-XOR solution

Query(1011)

Current node (in **red**) : ε

Traverse to 0

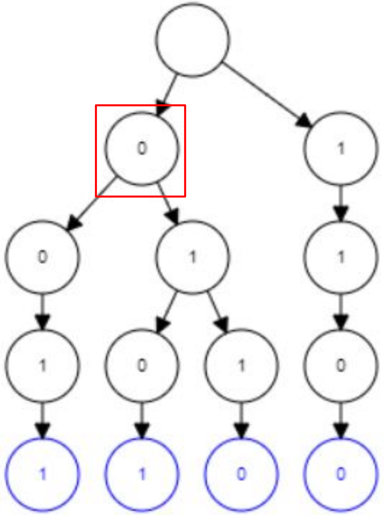


MAX-XOR solution

Query(1011)

Current node (in **red**) : 0

Traverse to 01

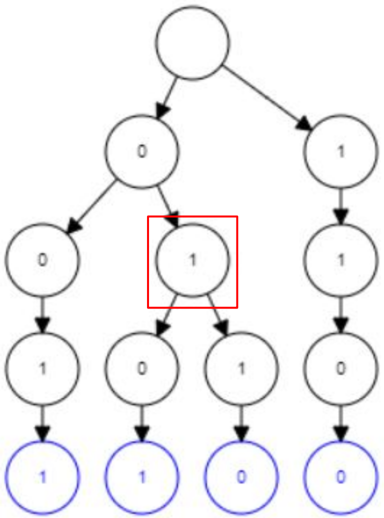


MAX-XOR solution

Query(1011)

Current node (in **red**) : 01

Traverse to 010

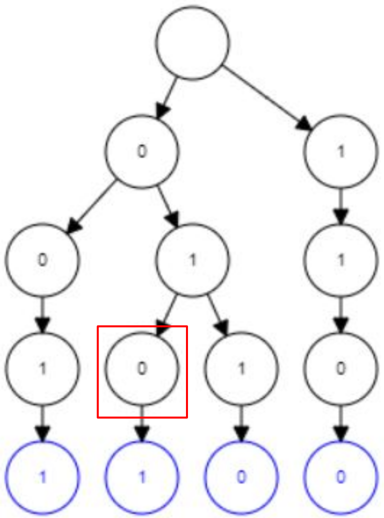


MAX-XOR solution

Query(1011)

Current node (in **red**) : 010

Edge 0 does not exist, forced to traverse to 0101



MAX-XOR solution

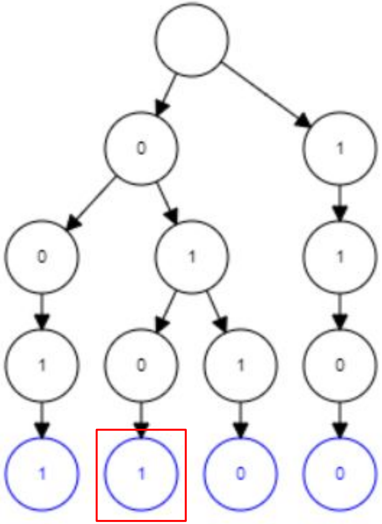
Query(1011)

Current node (in red) : 0101

Reached end, max-xor = 1011 xor 0101 = 1110

Many bitwise problems can be done similarly

⇒ **01-trie**: trie where the alphabet ∑ = {0, 1}



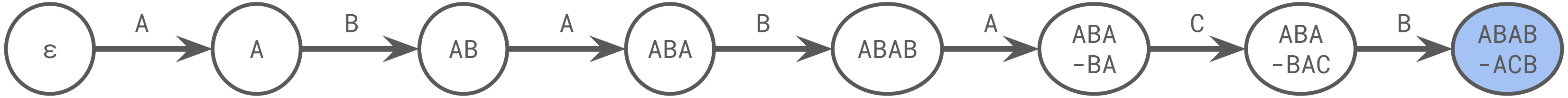
From Prefix Function to Automata

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 = “ABABACB”

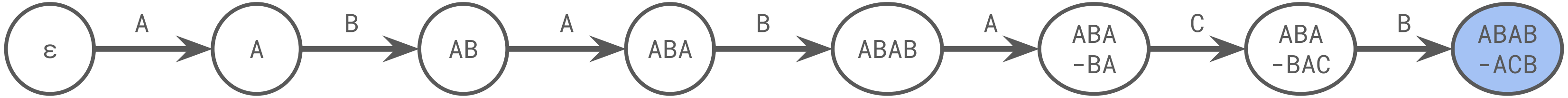


Recall: Prefix Function

The **prefix function** for string S is defined as an array π of length |S|, where π[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 = “ABABACB”

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

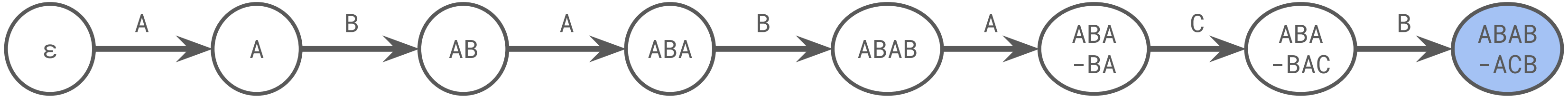


Suffix Link

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.

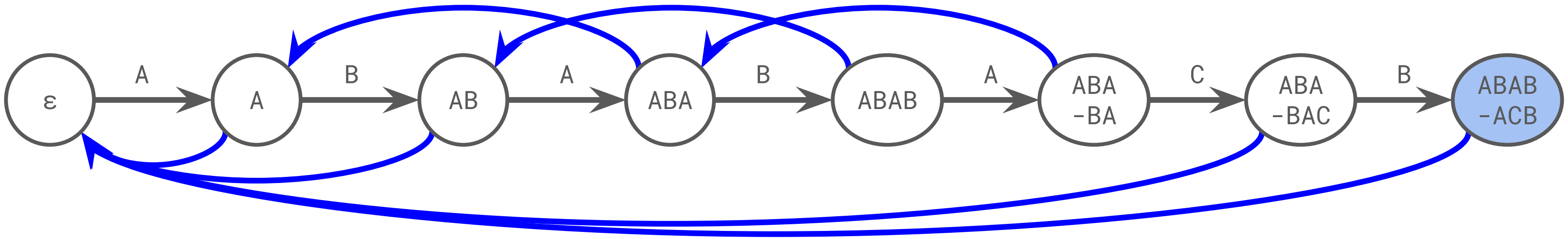
⇒ Prefix function applies, we create **suffix links** according to π[i]



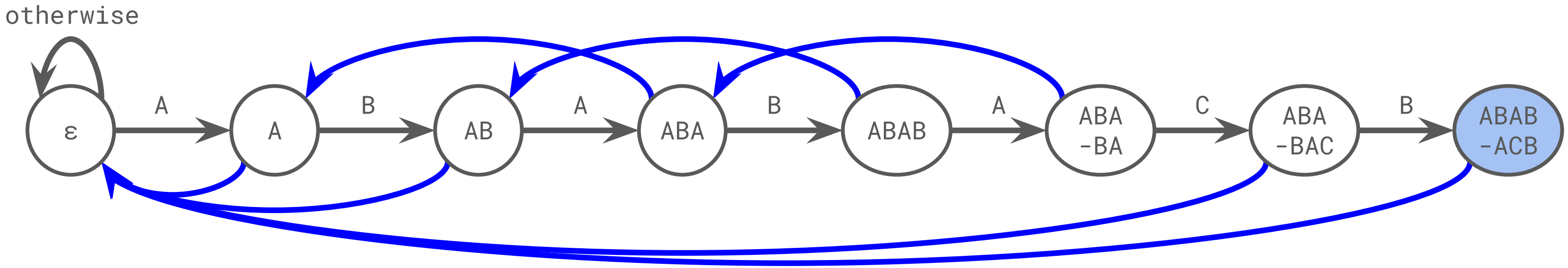
Suffix Link

S = “ABABACB”

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



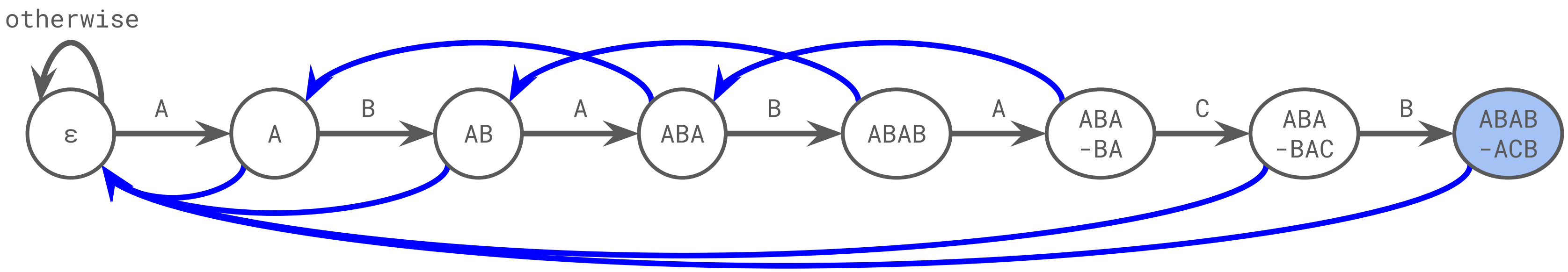
Let’s add one more thing…



KMP Algorithm: Automaton Version

Done!

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



Deterministic Finite Automaton (DFA)

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

- Q is a finite set of states
- $\sum$ is an alphabet
- $\delta: Q \times \sum \rightarrow Q$ 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 <u>directed graph</u> that <u>takes a string as input</u>
- 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 $\sum$ 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:

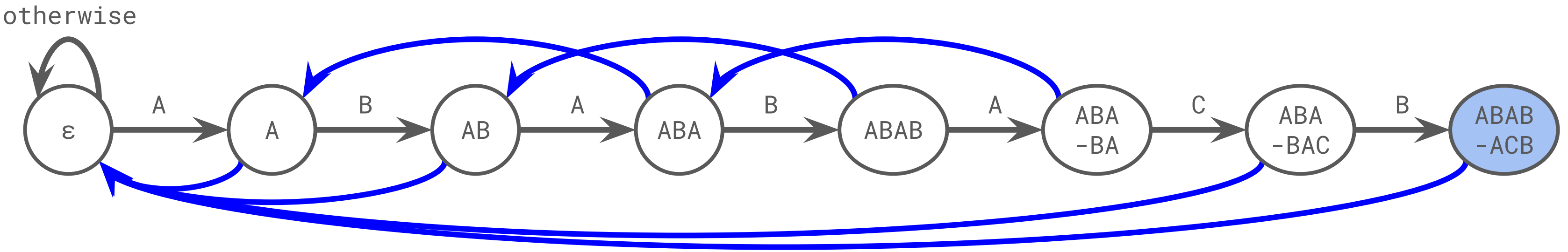
- “*designed to automatically follow a sequence of operations, or respond to predetermined instructions*”
- It takes a string as input, processes the symbols in order one by one
- For each symbol, it moves from its current state to the next state following the predetermined transition function

KMP Algorithm: Automaton Version

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

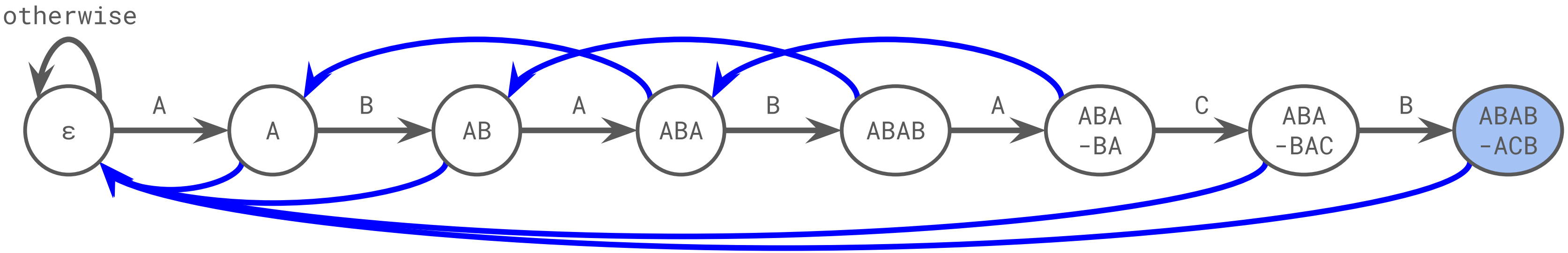
From all our previous insights, our goal is to:

- 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**



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: ???

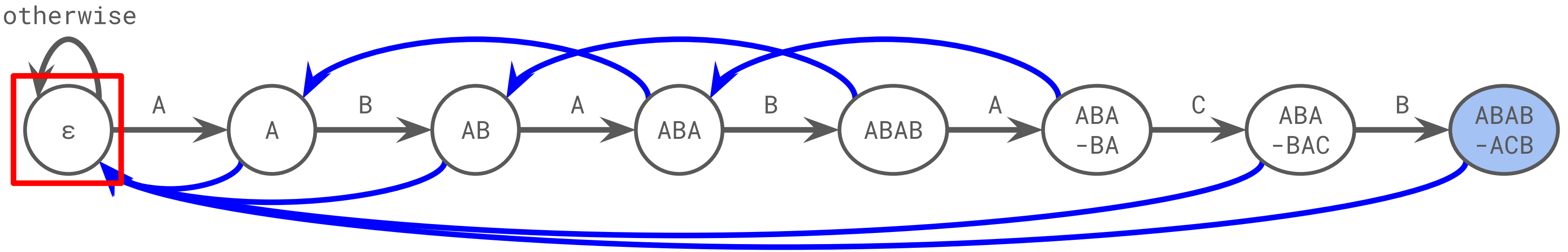


KMP Algorithm: Automaton Version

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 = “ABAAABABACB”, T = “ABABACB”

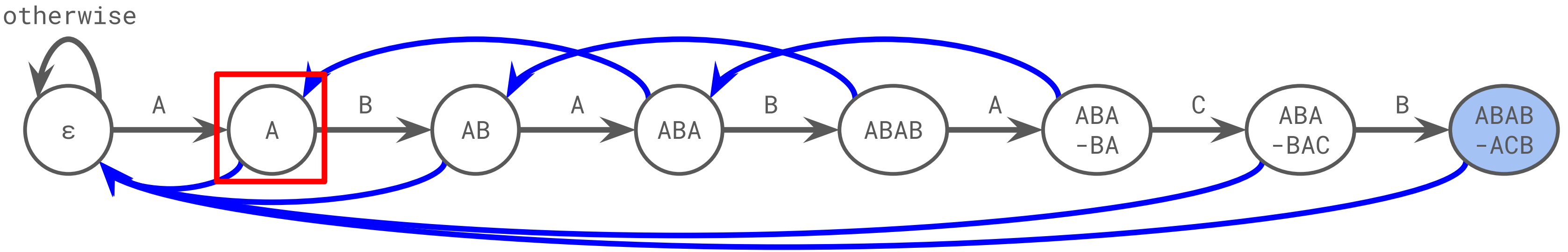


KMP Algorithm: Automaton Version

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:
  ⇒ $\delta(i, c) = i+1$

S = “ABAAABABACB”, T = “ABABACB”

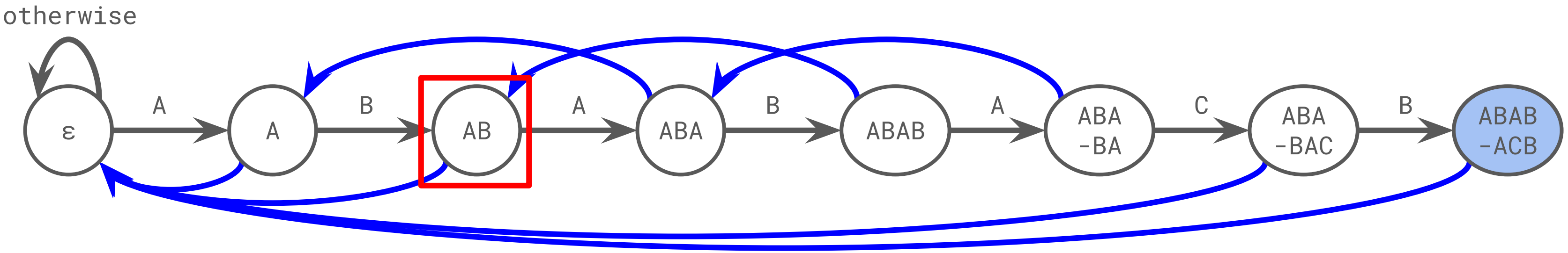


KMP Algorithm: Automaton Version

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:
  ⇒ δ(i, c) = i+1

S = "ABAAABABACB", T = "ABABACB"

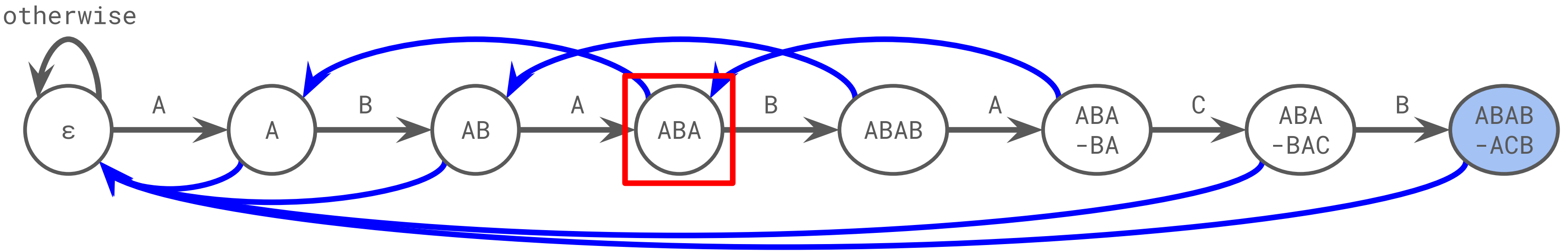


KMP Algorithm: Automaton Version

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:
  ⇒ δ(i, c) = i+1

S = "**ABA**AABABACB", T = "ABABACB"

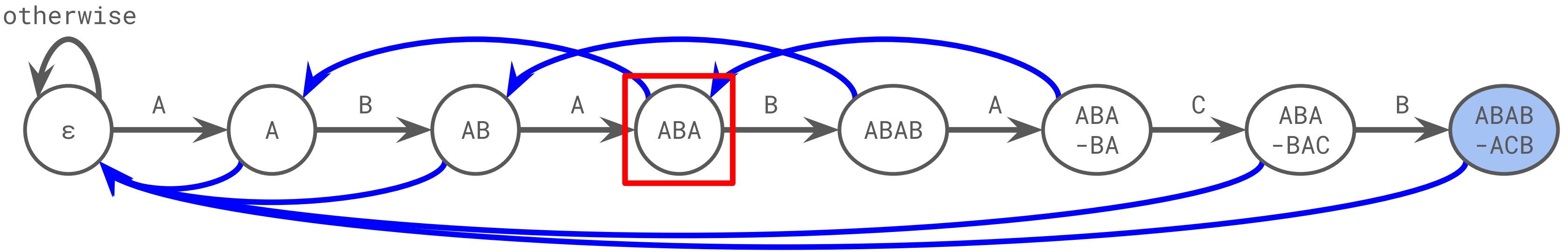


KMP Algorithm: Automaton Version

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

- If c != T[i], 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 = "ABAAABABACB", T = "ABABACB"



KMP Algorithm: Automaton Version

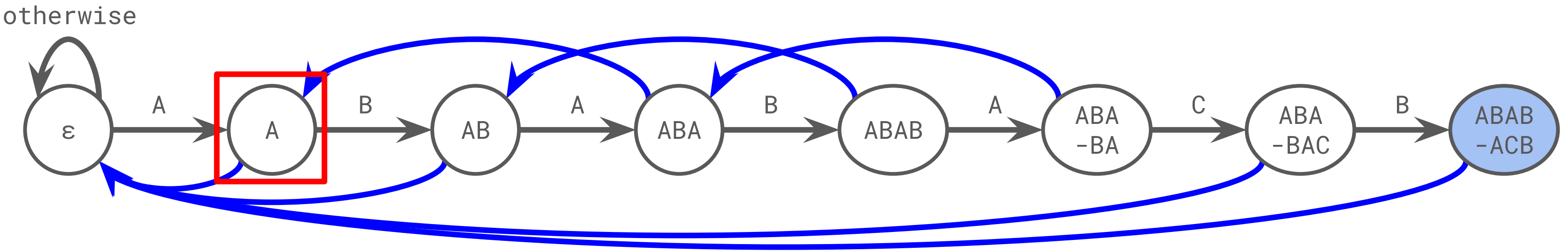
Transition function: Let input be **c** and the current state be **i** (i.e. T[0..i-1])

- If c != T[i], 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 = "**ABAA**ABABACB", T = "ABABACB"

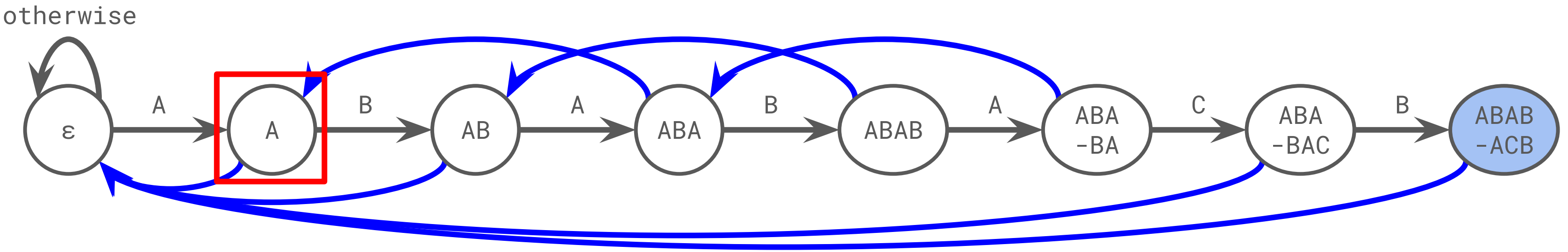


KMP Algorithm: Automaton Version

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

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

S = "**ABAA**ABABACB", T = "ABABACB"



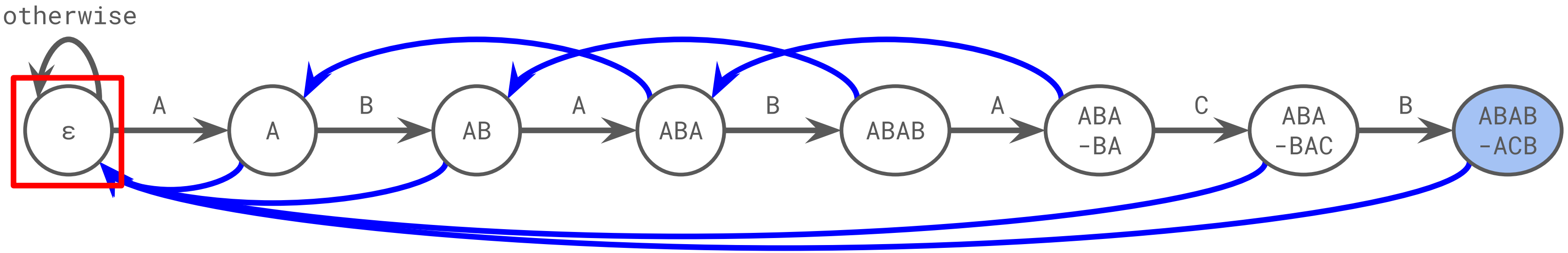
KMP Algorithm: Automaton Version

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

- If c != T[i], 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 δ(i, c) = δ(π[i-1], c) = δ(π[π[i-1]-1], c) = ...

S = "**ABAA**ABABACB", T = "ABABACB"



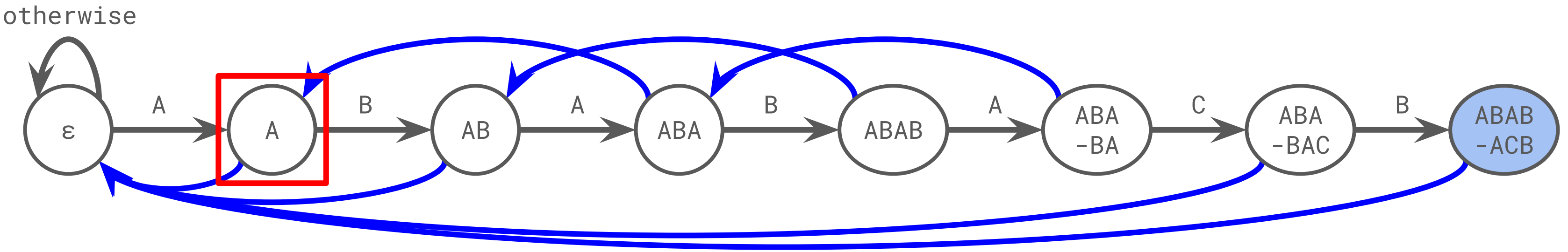
KMP Algorithm: Automaton Version

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

- If c != T[i], 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 $\delta(i, c) = \delta(\pi[i-1], c) = \delta(\pi[\pi[i-1]-1], c) = \ldots$

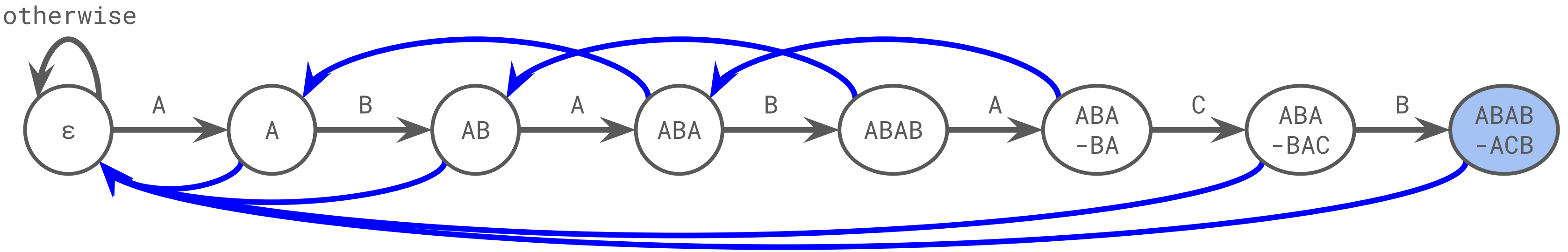
S = "**ABAA**ABABACB", T = "ABABACB"



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.

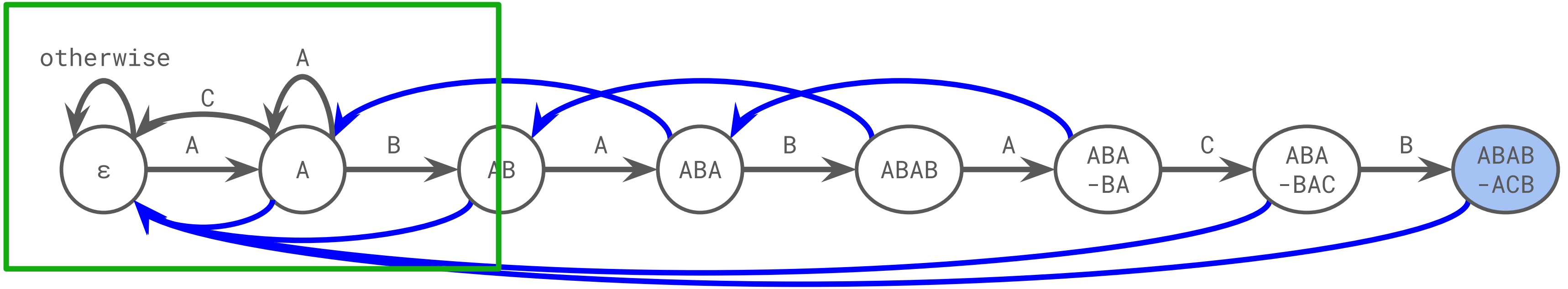
- The real transition function δ is formed by recursive relations thanks to suffix links / the prefix function
- From the context of the graph, a real transition is formed by **suffix link(s)** until the **soonest** matching grey edge using the current letter



KMP Algorithm: Automaton Version

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



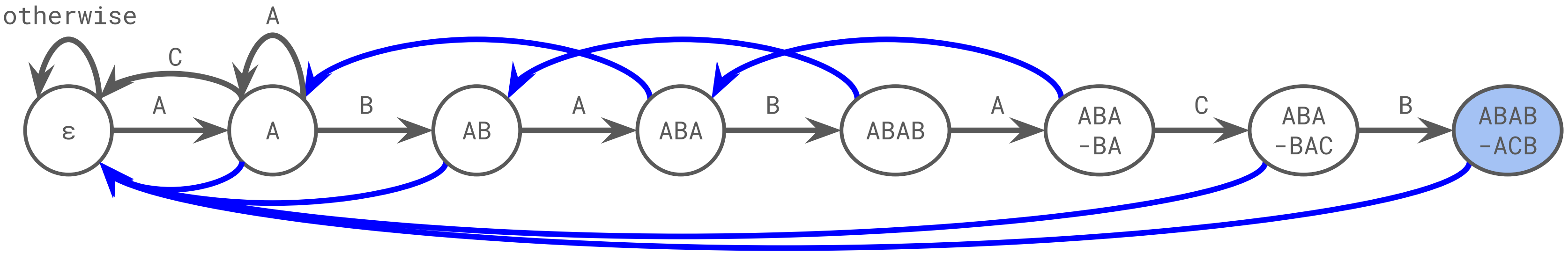
KMP Algorithm: Automaton Version

- $\delta(i, c) = (c == T[i])\ ?\ (i+1) : \delta(\pi[i-1], c)$
- Computing the transition function <u>in increasing order of **i**</u> + Memoization

  ⇒ Dynamic Programming

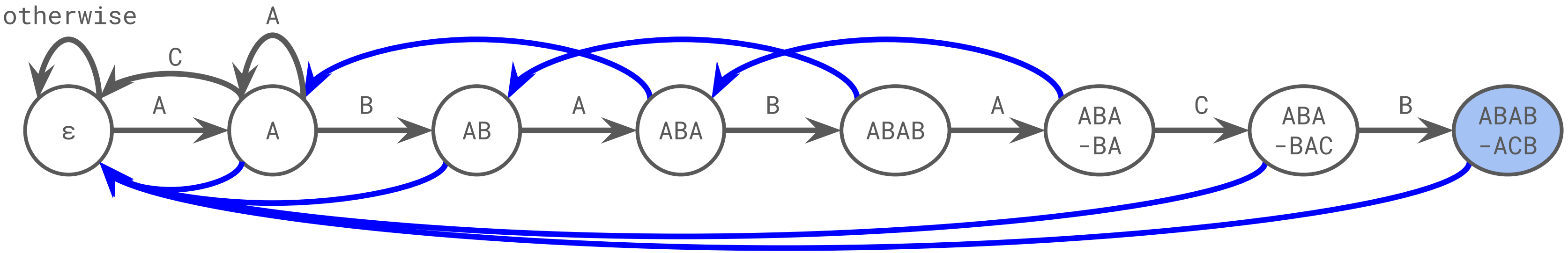
Time Complexity:

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



KMP Algorithm: Automaton Version

i	0(ε)	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	0	2	0	4	0	4	7	0	/
C	0	0	0	0	0	6	0	0	/



KMP Algorithm: Automaton Version

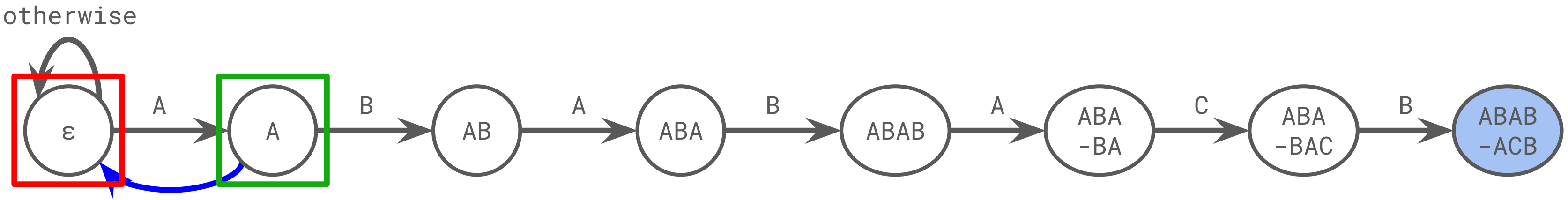
Final question: How to build **without precomputing prefix function**?

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 = "**A**BABACB", T[1..] = "ABABACB"



KMP Algorithm: Automaton Version

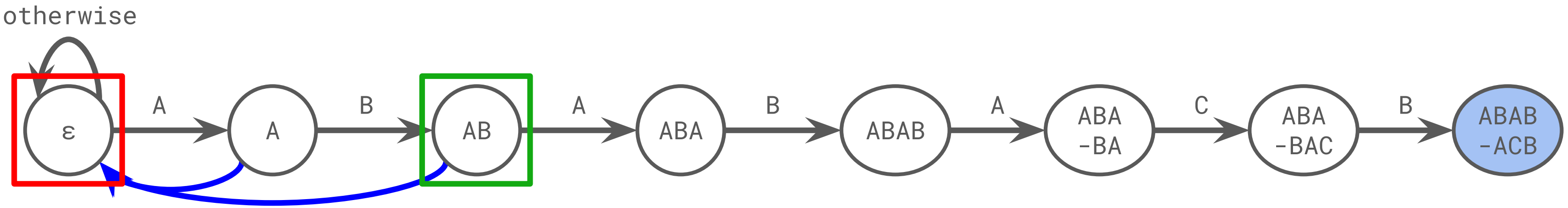
Final question: How to build **without precomputing prefix function**?

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 = “**AB**ABACB”, T[1..] = “A**B**ABACB”



KMP Algorithm: Automaton Version

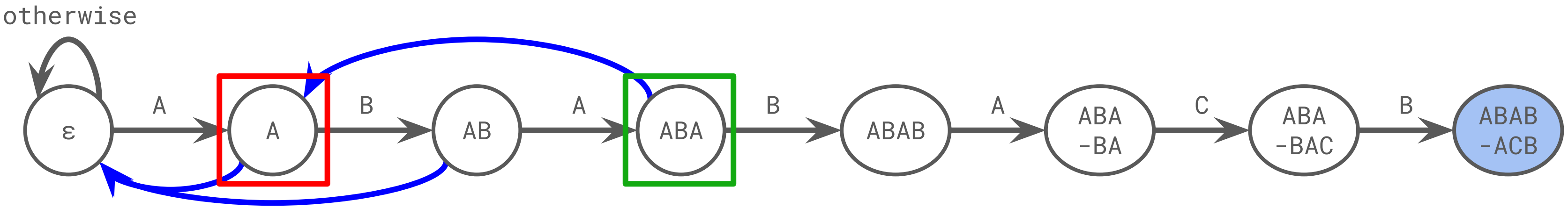
Final question: How to build **without precomputing prefix function**?

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 = "**ABA**BACB", T[1..] = "A**BA**BACB"



KMP Algorithm: Automaton Version

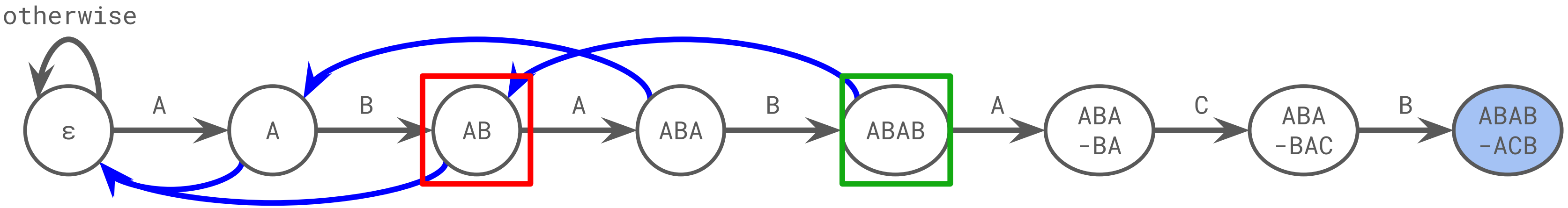
Final question: How to build **without precomputing prefix function**?

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 = "**ABAB**ACB", T[1..] = "A**BAB**ACB"



KMP Algorithm: Automaton Version

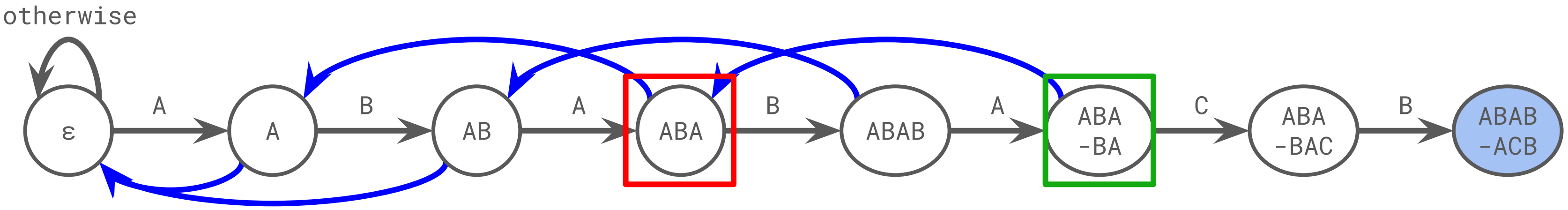
Final question: How to build **without precomputing prefix function**?

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 = “**ABABA**CB”, T[1..] = “A**BABA**CB”



KMP Algorithm: Automaton Version

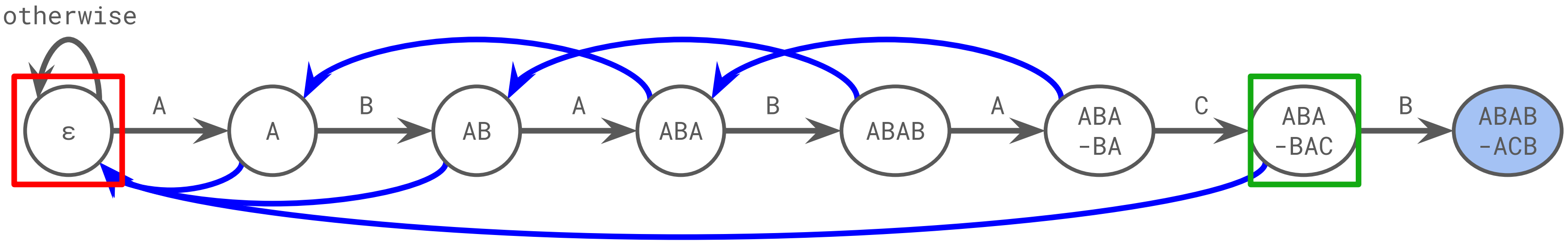
Final question: How to build **without precomputing prefix function**?

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 = “**ABABAC**B”, T[1..] = “A**BABAC**B”



KMP Algorithm: Automaton Version

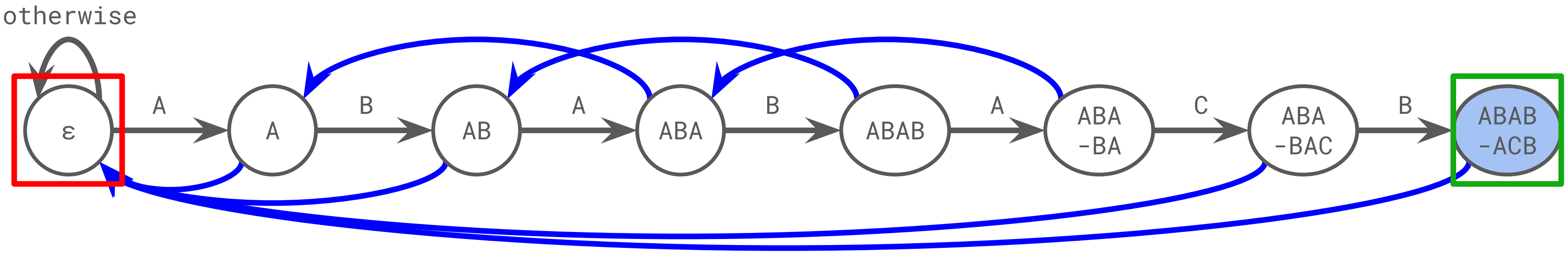
Final question: How to build **without precomputing prefix function**?

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 = “**ABABACB**”, T[1..] = “A**BABACB**”



KMP Algorithm: Automaton Version

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

Method 2:

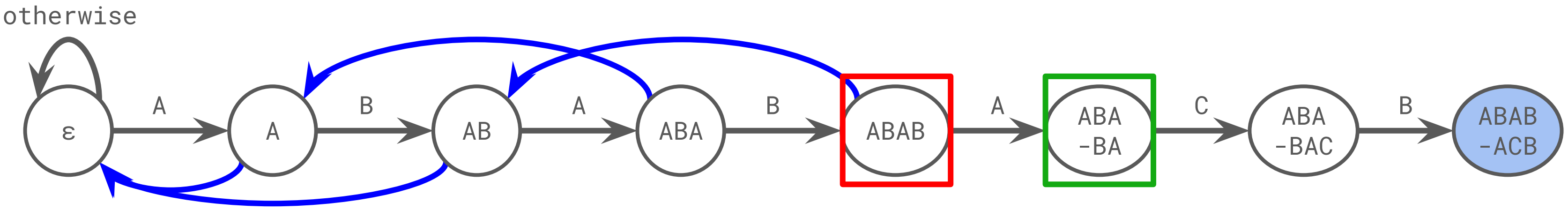
- Referring back to Prefix Function Observation 1
- Assume we are currently at node **p**
- Let **u**+’c’ be the longest proper suffix of **p**+’c’
  - **u**+’c’ = longest proper suffix of **p**+’c’
  - **u**+’c’ = proper suffix of **p**+’c’
  - **u** = proper suffix of **p**

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

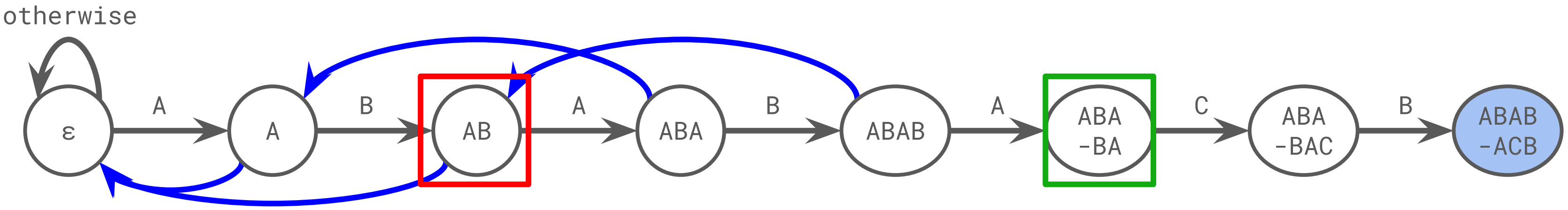


KMP Algorithm: Automaton Version

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

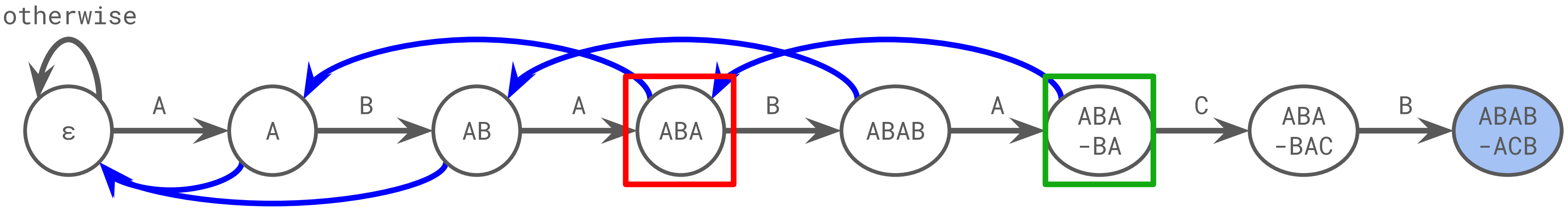


KMP Algorithm: Automaton Version

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

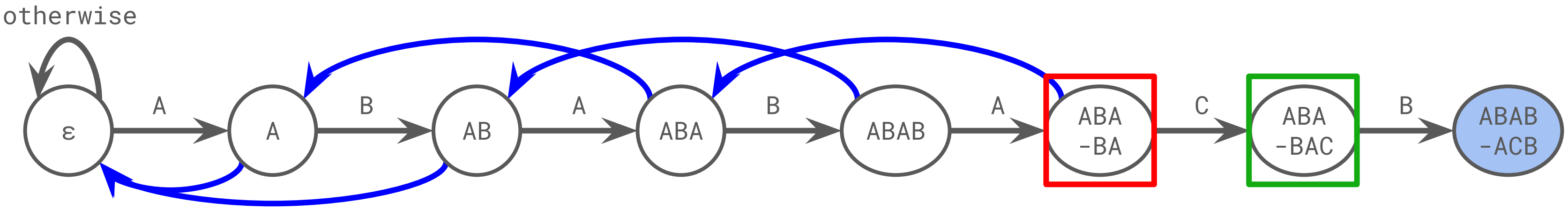


KMP Algorithm: Automaton Version

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

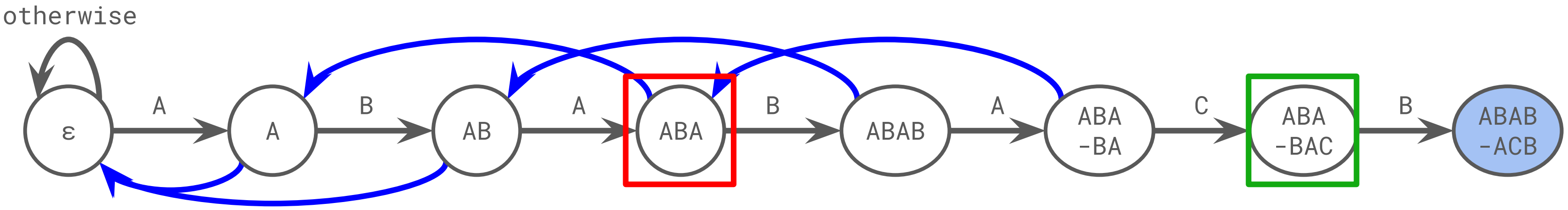


KMP Algorithm: Automaton Version

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

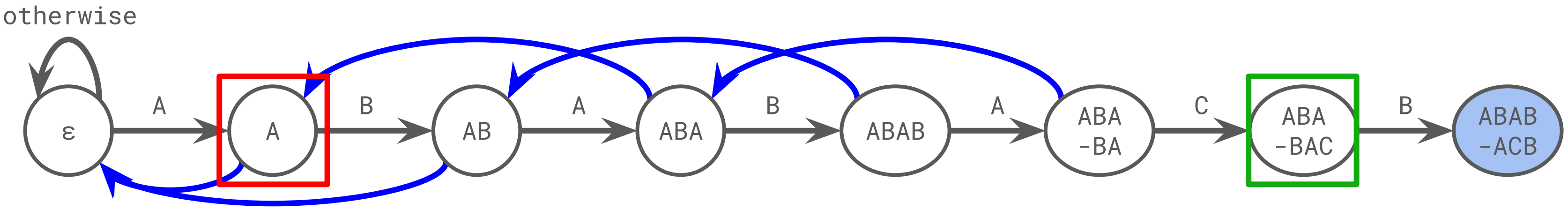


KMP Algorithm: Automaton Version

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

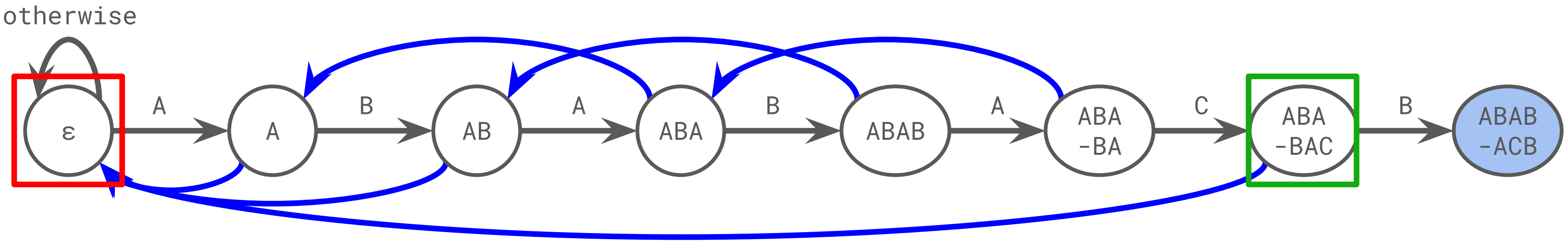


KMP Algorithm: Automaton Version

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



Aho-Corasick algorithm

Recall: KMP Algorithm

Solves single pattern matching problem

- Preprocess pattern T: O(|T|) / O(|∑| × |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(|∑| × m)
- String matching in S: O(|S| × 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:

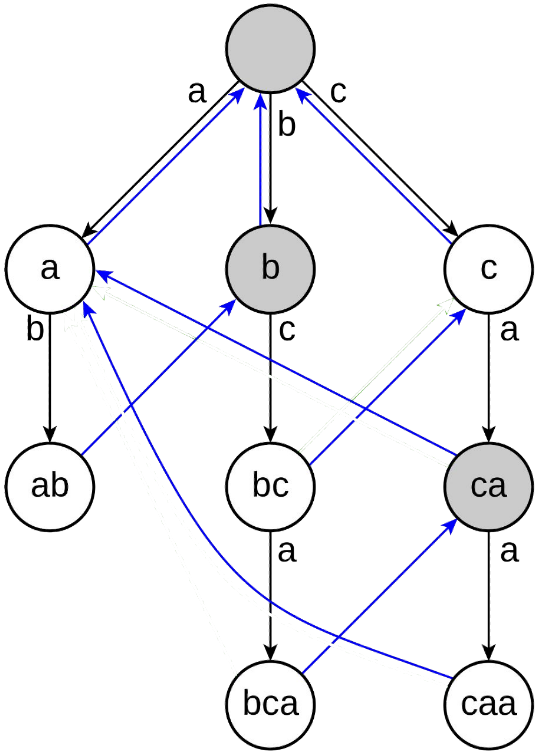
- Build AC Automaton: $O(|\Sigma| \times m)$
- Search patterns in S = Run AC Automaton: $O(|S|)$?

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



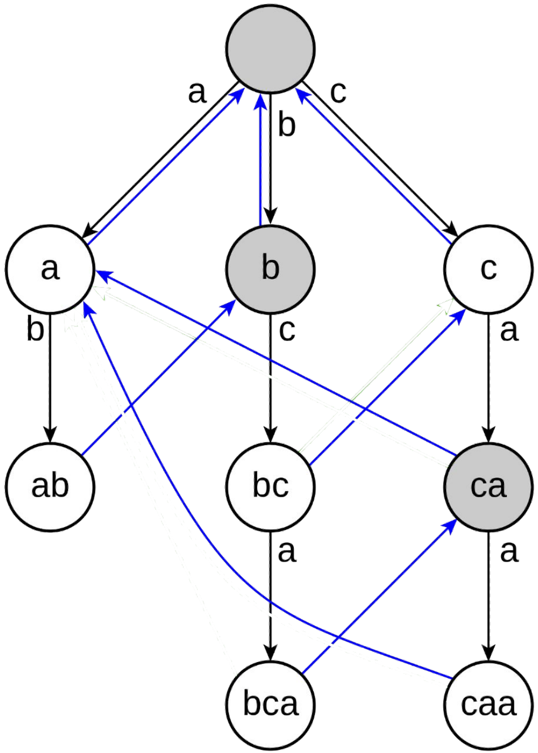
Example

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!



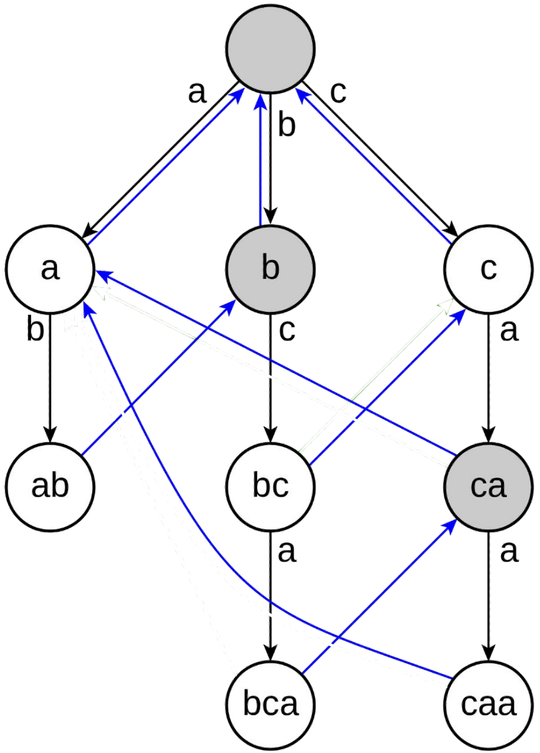
Example

Patterns = {“a”, “ab”, “bc”, “bca”, “c”, “caa”}

S = “cabca”

- c found c
- c → ca suffix(es) of ca: a
- ca → a → ab found ab
- ab → b → bc found bc suffix(es) of bc: c
- bc → bca found bca suffix(es) of bca: a

Correct answer = 7

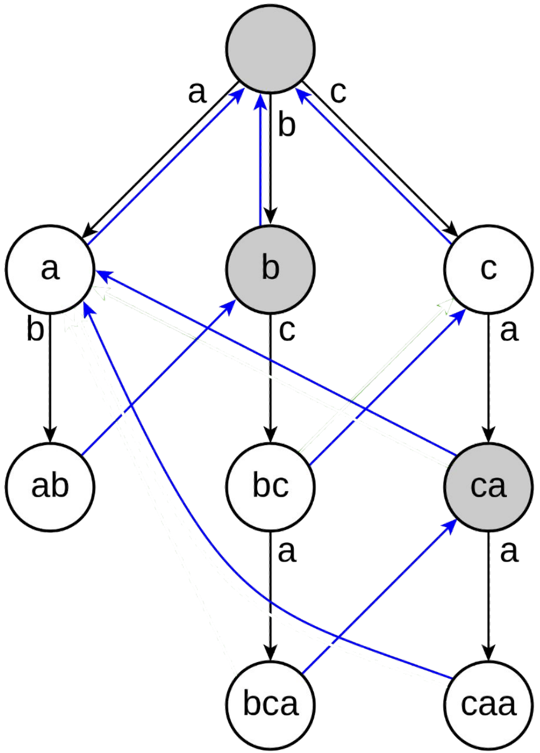


Aho-Corasick algorithm

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: $O(|S| \times \max(|T_i|))$



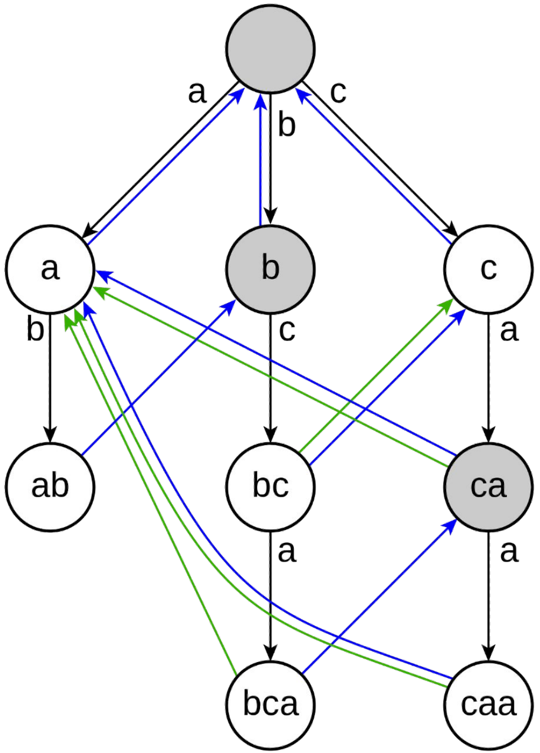
Aho-Corasick algorithm

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 5

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

4 3
AT, AC, AG, AA
Ans = 36

DNA Sequence

- Dangerous nodes, nodes that when you visit will become invalid, either belong to one of the **pattern**s, 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

Take a break

Suffix Array

Suffix Array

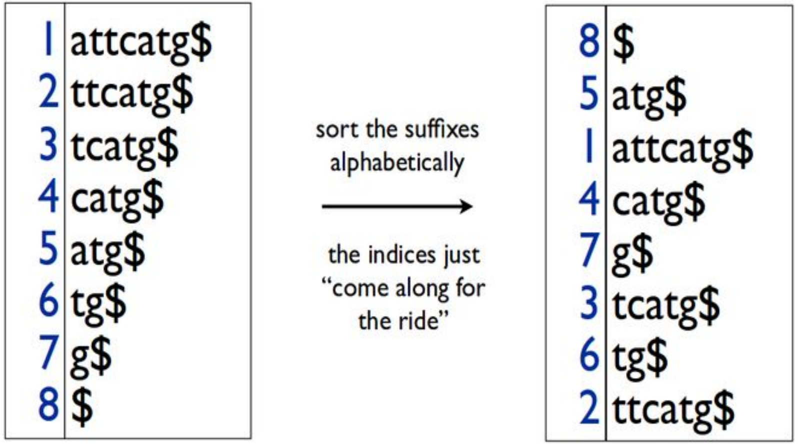
Sorted list of suffixes of string

- 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



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**

`rank[i]` = the rank of the first `len` characters of the `i`th suffix

`len = 1, 2, 4, 8, ….`

Suffix Array

1. Compute `rank[]` by the first 1 character of the i^{th} suffix (ASCII)
2. Sort `SA[]` by the first 2 character

   How? By comparing elements by `rank[]`
3. Update the `rank[]`
4. Sort `SA[]` by the first 4 character
5. So on.......

Finally, we sorted `SA[]` by the whole suffix (`len ≥ N`)

Suffix Array

Compare elements by `rank[]`

- Assume we computed `rank[]` when `len = k`

`len = 2k`, compare `i` and `j` (suffixes start at `i` and `j`)

Compare `(rank[i],rank[i+k])` and `(rank[j],rank[j+k])`

Suffix Array

`len = 2k`, compare `i` and `j` (suffixes start at `i` and `j`)

Compare (`rank[i]`, `rank[i+k]`) and (`rank[j]`, `rank[j+k]`)

compare `rank[i]` and `rank[j]` = compare `S[i..i+k-1]` and `S[j..j+k-1]`

compare `rank[i+k]` and `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^th and v^th suffix when 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 >= 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]`

⇒ `sa[rank[k]] = k`

i	sa[i]	suffix(sa[i])
0	10	A
1	7	ABRA
2	0	$\boxed{\text{ABRACADABRA}}$
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 `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://gagguy.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. ABCDE ABEDC, lcp = 2

Suffix Array – LCP

`lcp[i]` = <u>longest common prefix</u> of
`suffix(sa[i])` and
`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:

lcp(“ABRA”, “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:

`lcp(“ABRA”, “ADABRA”) = 1`

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

`lcp(“ABRA”, “ADABRA”)`
`= min({lcp[2], lcp[3], 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

`lcp(suffix(i),suffix(j)) = min{lcp[rank[i]+1],lcp[rank[i]+2],…,lcp[rank[j]]}`

- Assume `rank[i] ≤ 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`

```
  lcp[rank[i]]
= lcp(suffix(sa[rank[i]]), suffix(sa[rank[i]-1]))
= lcp(suffix(i), suffix(sa[rank[i]-1]))

  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]))
```

Suffix Array – LCP

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

```
  lcp[rank[i]]
= lcp(suffix(sa[rank[i]]), suffix(sa[rank[i]-1]))
= lcp(suffix(i), suffix(sa[rank[i]-1]))

  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]))
```

Suffix Array – LCP

Let lcp[rank[i]] = lcp(suffix(i), suffix(sa[rank[i]-1])) = h.

⇒	suffix(i) is ***right* after**	suffix(sa[rank[i]-1]) in the SA,	their LCP = h
⇒	suffix(i) is after	suffix(sa[rank[i]-1]) in the SA,	their LCP ≥ h
⇒	suffix(i+1) is after	suffix(sa[rank[i]-1]+1) in the SA,	their LCP ≥ h-1

Consider lcp[rank[i+1]] = lcp(suffix(i+1), suffix(sa[rank[i+1]-1]))

⇒ suffix(i+1) is ***right* after** suffix(sa[rank[i+1]-1]) in the SA

\+ suffix(sa[rank[i]-1]+1) is in between these two suffixes in the SA

⇒ lcp[rank[i+1]] ≥ h-1

Suffix Array – LCP

We have proved:

- If `lcp[rank[i]] = h`
- Then `lcp[rank[i + 1]] ≥ h-1`

Calculate `lcp[]` in the order of `rank[0]`, `rank[1]`, `rank[2]`, ...

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

Suffix Array – LCP

e.g. S = “BCEABCDABCEB”

`SA[rank[7]] = ABCEB`

`lcp[rank[7]] = 3`, i.e. there exist suffix = ABCD……

`SA[rank[7 + 1]] = BCEB`

`lcp[rank[7 + 1]] ≥ 2`, i.e. there exist suffix = BCD……

p.s. `lcp[rank[7+1]]` may not be 2, in this case `lcp[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 substring

```
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/

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

Trie

- IOI08 Type Printer

KMP

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

Suffix array

- HKOI M1762
- POJ 2774
- NOI15 品酒大會

Aho-Corasick algorithm

- HDU 2222
- HDU 3065
- POJ 2778

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)

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