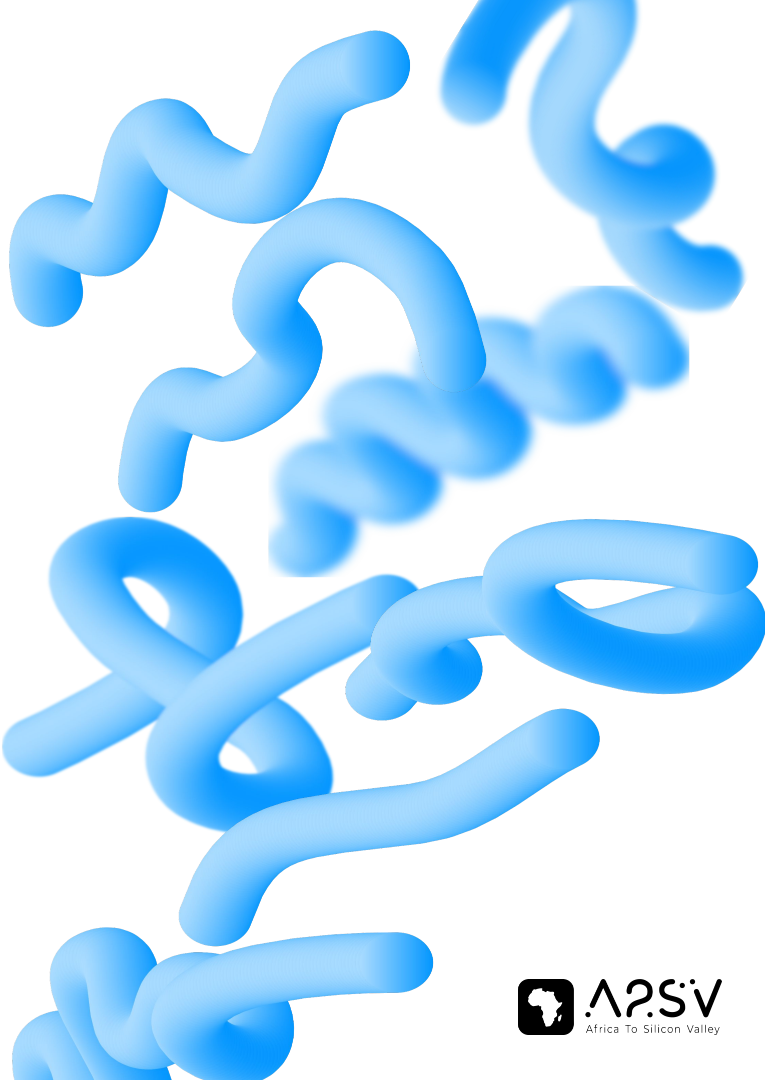
# Advanced String Algorithms

## Substring search



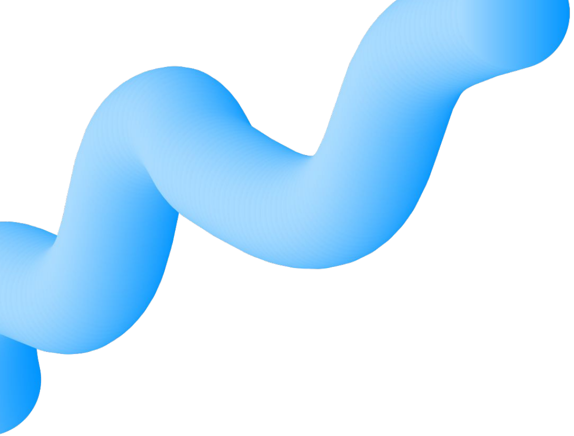
A2SV
Africa To Silicon Valley

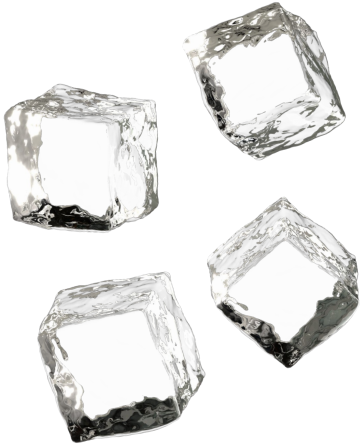
# Lecture Outline

- Prerequisites
- Substring Search (The Naive Way)
- Rabin-Karp Algorithm
- Knuth-Morris-Pratt Algorithm
- Applications of Rabin-Karp and Knuth-Morris-Pratt Algorithm
- Additional String Algorithms
- Quote of the Day

# Pre-requisites

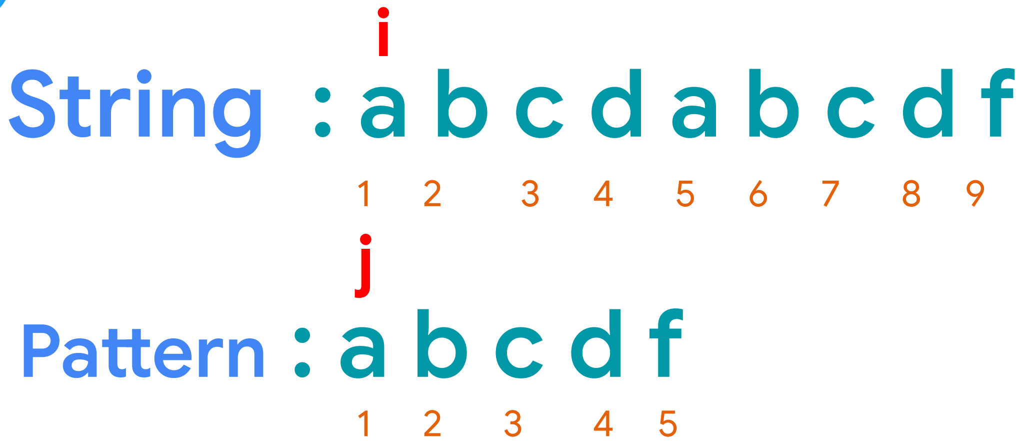
- Math II
- String manipulation in Python
- Time and Space complexity analysis





# What is a substring search?

# Naive Method



i

String : a b c d a b c d f

1 2 3 4 5 6 7 8 9

j

Pattern : a b c d f

1 2 3 4 5

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  | ✓ | i |  |  |  |  |  |  |  |
| String : | a | b | c | d | a | b | c | d | f |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

|  |  |  |  |  |  |
|---|---|---|---|---|---|
|  | ✓ | j |  |  |  |
| Pattern : | a | b | c | d | f |
|  | 1 | 2 | 3 | 4 | 5 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | ✓ | ✓ | i | | | | | | |
| String : | a | b | c | d | a | b | c | d | f |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | ✓ | ✓ | j | | | | | | |
| Pattern : | a | b | c | d | f | | | | |
| | 1 | 2 | 3 | 4 | 5 | | | | |

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  | ✓ | ✓ | ✓ | i |  |  |  |  |  |
| String : | a | b | c | d | a | b | c | d | f |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

|  |  |  |  |  |  |
|---|---|---|---|---|---|
|  | ✓ | ✓ | ✓ | j |  |
| Pattern : | a | b | c | d | f |
|  | 1 | 2 | 3 | 4 | 5 |

A2SV
Africa To Silicon Valley

✓ ✓ ✓ ✓ i

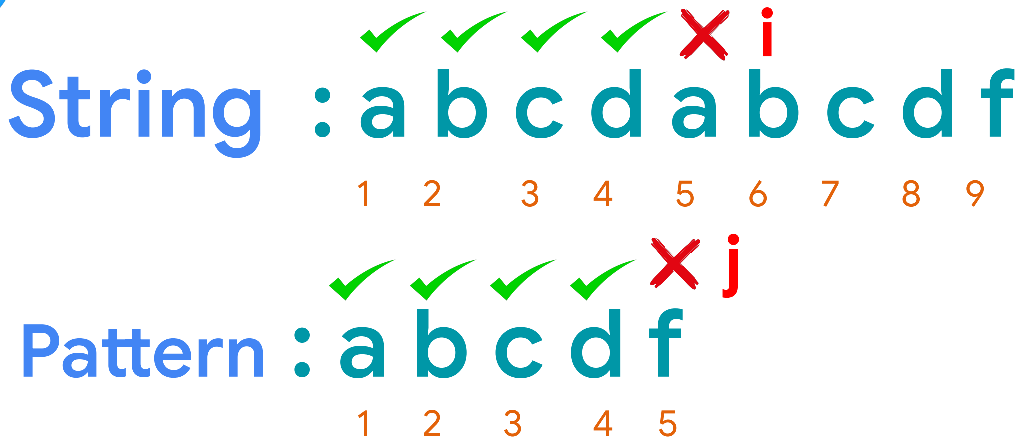
String : a b c d a b c d f

1 2 3 4 5 6 7 8 9

✓ ✓ ✓ ✓ j

Pattern : a b c d f

1 2 3 4 5



✓ ✓ ✓ ✓ ✗ i

String : a b c d a b c d f

1 2 3 4 5 6 7 8 9

✓ ✓ ✓ ✓ ✗ j

Pattern : a b c d f

1 2 3 4 5

# Okay, let’s try again

i

String : a b c d a b c d f

1 2 3 4 5 6 7 8 9

j

Pattern : a b c d f

1 2 3 4 5

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | ✗ | i | | | | | | |
| String : | a | b | c | d | a | b | c | d | f |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | ✗ | j | | | | | | | |
| Pattern : | a | b | c | d | f | | | | |
| | 1 | 2 | 3 | 4 | 5 | | | | |

# Failed yet again.

## AGAIN !

```
              i
String  : a b c d a b c d f
          1 2 3 4 5 6 7 8 9
          j
Pattern : a b c d f
          1 2 3 4 5
```

✕ i

String : a b c d a b c d f

1 2 3 4 5 6 7 8 9

✕ j

Pattern : a b c d f

1 2 3 4 5

# AGAINNN !!!

i

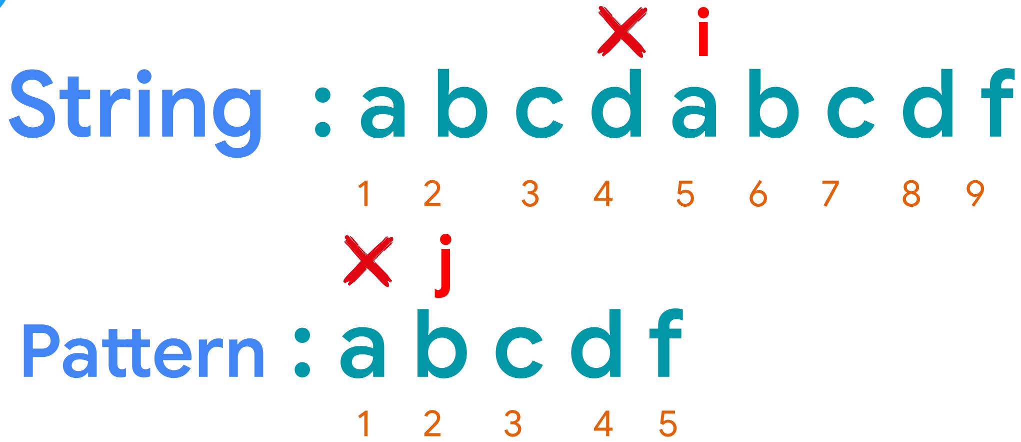
String : a b c d a b c d f

1 2 3 4 5 6 7 8 9

j

Pattern : a b c d f

1 2 3 4 5



✕ i

String : a b c d a b c d f

1 2 3 4 5 6 7 8 9

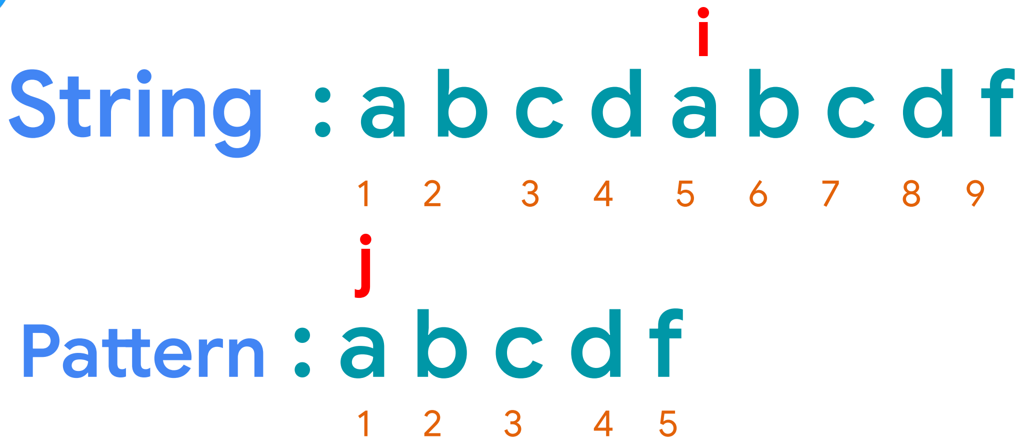
✕ j

Pattern : a b c d f

1 2 3 4 5

# Hmm :/

# Again ?



i

String : a b c d a b c d f

1 2 3 4 5 6 7 8 9

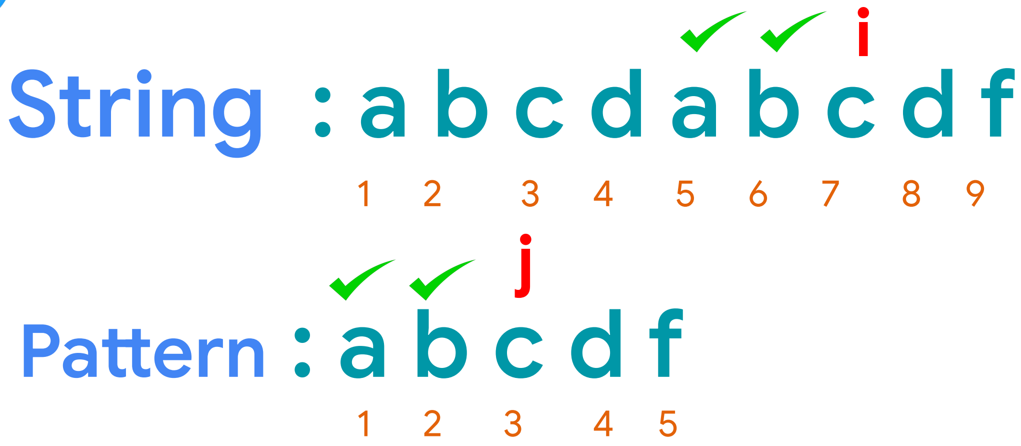
j

Pattern : a b c d f

1 2 3 4 5

```
                    ✓ i
String  : a b c d a b c d f
          1 2 3 4 5 6 7 8 9

          ✓ j
Pattern : a b c d f
          1 2 3 4 5
```



✓ ✓ i

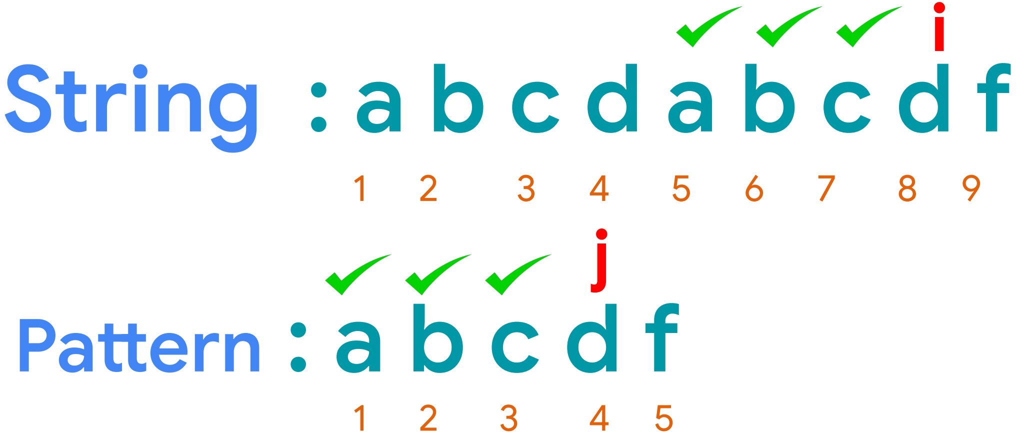
String : a b c d a b c d f

1 2 3 4 5 6 7 8 9

✓ ✓ j

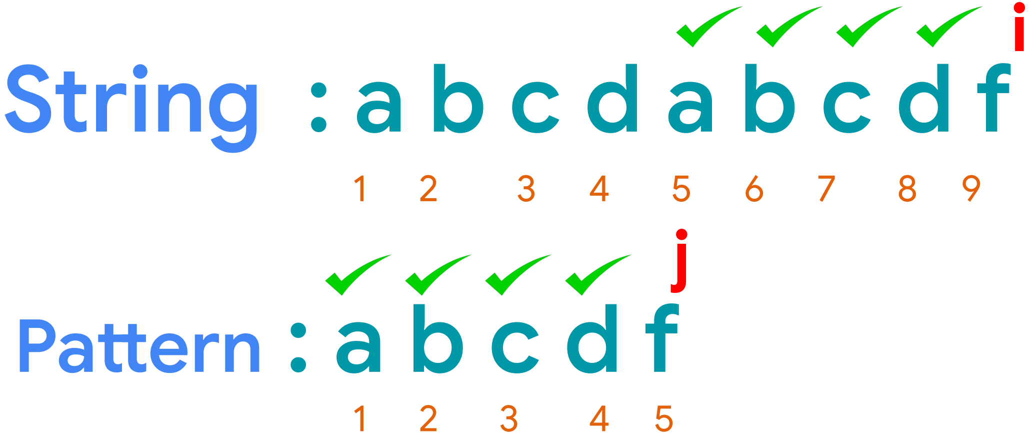
Pattern : a b c d f

1 2 3 4 5



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | ✓ | ✓ | ✓ | i | |
| String : | a | b | c | d | a | b | c | d | f |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

| | | | | | |
|---|---|---|---|---|---|
| | ✓ | ✓ | ✓ | j | |
| Pattern : | a | b | c | d | f |
| | 1 | 2 | 3 | 4 | 5 |


✓ ✓ ✓ ✓ i
String : a b c d a b c d f
1 2 3 4 5 6 7 8 9
✓ ✓ ✓ ✓ j
Pattern : a b c d f
1 2 3 4 5

✓ ✓ ✓ ✓ ✓ i

String : a b c d a b c d f

1 2 3 4 5 6 7 8 9

✓ ✓ ✓ ✓ ✓ j

Pattern : a b c d f

1 2 3 4 5

# Okay we got somewhere, but how long did it take us ?

$O(n*m)$

# Practice Problem

[Find the index of the first occurrence in a string]

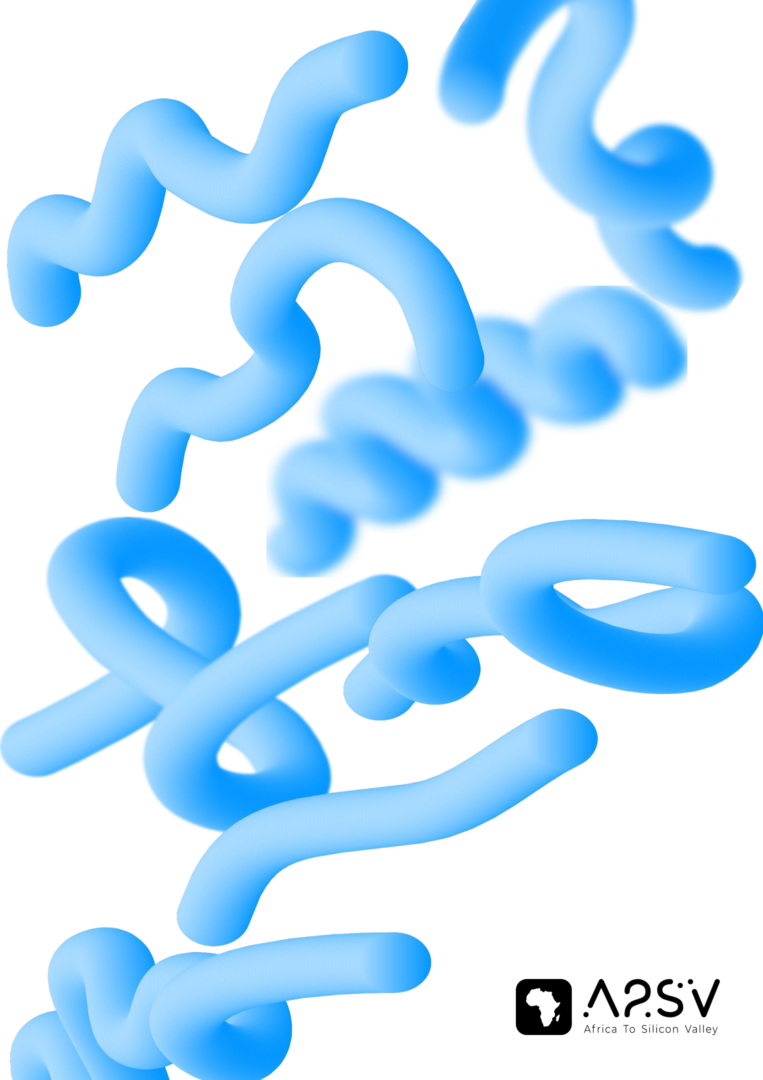
```python
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        if len(needle) > len(haystack):
            return -1

        i = j = 0
        while i < len(haystack):
            ans = i
            while i < len(haystack) and haystack[i] == needle[j]:
                i += 1
                j += 1

                if j == len(needle):
                    return ans
            i = ans + 1
            j = 0
        return -1
```

# Rabin-Karp Algorithm

Average O(n + m) Time



A2SV Africa To Silicon Valley

# What is Hashing ?

A2SV
Africa To Silicon Valley

# Why do we need to encode strings ?

# Encoding Strings

For s = “abcad”,

let’s start thinking in base alphabets.

# Encoding Strings

For s = “abcad”,

`a` * $26^4$ + `b` * $26^3$ + `c` * $26^2$ + `a` * $26^1$ + `d` * $26^0$

We need to find some values to represent each of the above letters. Any ideas ?

# Encoding Strings

`a` = 0
`b` = 1
`c` = 2
`d` = 3
.
.
`z` = 25

# This will result in an edge case if we represent strings this way

“aaa” => $0 * 26^2 + 0 * 26^1 + 0 * 26^0 = 0$

“aa” => $0 * 26^1 + 0 * 26^0 = 0$

# There are two ways to fix this problem

1. Encode the length in the hash (messy)
2. Don’t use 0, encode (alphabet + 1) size

# Operations on Hashes

# Operation: addLast

let $a = 26 + 1$

“abc” + “x” = ?

“abc” => $(1 * a^2 + 2 * a^1 + 3 * a^0)$

“x” => $(24 * a^0)$

“abc” + “x” => $(1 * a^2 + 2 * a^1 + 3 * a^0) * a + (24 * a^0)$

“abcx” => $1 * a^3 + 2 * a^2 + 3 * a^1 + 24 * a^0$ ✓

# Operation: pollFirst

let $a$ = 26 + 1

“abcx” = let’s try to remove the `a` ?

“abcx” => $1 * a^3 + 2 * a^2 + 3 * a^1 + 24 * a^0$

“bcx” => $(1 * a^3 + 2 * a^2 + 3 * a^1 + 24 * a^0) - (1 * a^3)$

“bcx” => $2 * a^2 + 3 * a^1 + 24 * a^0$ ✓

# For Rabin-Karp, the above two operations suffice for most cases

# Operation: addFirst

let $a = 26 + 1$

“x” + “abc” = ?

“x” => $(24 * a^0)$

“abc” => $(1 * a^2 + 2 * a^1 + 3 * a^0)$

“xabc” => $(24 * a^0) * a^3 + (1 * a^2 + 2 * a^1 + 3 * a^0)$ ✓

A2SV
Africa To Silicon Valley

# Operation: pollLast

let $a = 26 + 1$

“abcx” = let’s try to remove the `x` ?

“abcx” => $1 * a^3 + 2 * a^2 + 3 * a^1 + 24 * a^0$

“abc” => $((1 * a^3 + 2 * a^2 + 3 * a^1 + 24 * a^0) - (24 * a^0)) / a$

“abc” => $1 * a^2 + 2 * a^1 + 3 * a^0$ ✓

Most of the time, the hash values are very large numbers hence we need to use them under mod.

Therefore, the last operation is trickier than we made it look like; since it involves knowing division under mod

# TIP: Precompute all $a^k$

A2SV

Africa To Silicon Valley

TIP: Pick a Prime number for modulus.

Typically, 10 ** 9 + 7

(Fermat’s Little theorem)

# Why Choose a Prime Modulus in Rabin-Karp?

- Reduces Hash Collisions: Primes ensure a uniform distribution of hash values.
- Prevents Overflow: Large prime modulus like 10 ** 9 + 7 keeps hash values within limits.
- Fermat's Little Theorem: Enables efficient calculation of modular inverses for rolling hashes.

TIP: Use multiple primes to decrease the chance of collisions

# Rabin-Karp: Demonstration

String: abacdabazxywp

pattern: abaz

# Rabin-Karp: Demonstration

pattern: abaz

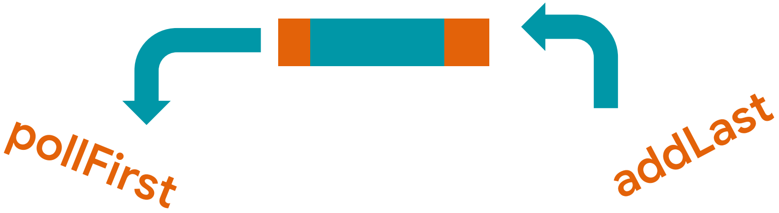
String: abacdabazxywp

$$(1 * a^3 + 2 * a^2 + 1 * a^1 + 3 * a^0)$$

# Rabin-Karp: Demonstration

pattern: abaz

String: abacdabazxywp



pollFirst

addLast

# Practice Problem

[Find the index of the first occurrence in a string]

A2SV
Africa To Silicon Valley

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        MOD = 10**9 + 7
        base = 27

        def convert(char):
            return ord(char) - 96

        def add_last(Hash, char):
            return (Hash * base + convert(char)) % MOD

        def poll_first(Hash, char, base_power):
            return (Hash - convert(char) * base_power) % MOD
```

```
N1, N2 = len(haystack), len(needle)
if N1 < N2:
    return -1

# Precompute base powers mod MOD
base_powers = [1] * (N2 + 1)
for i in range(1, N2 + 1):
    base_powers[i] = (base_powers[i - 1] * base) % MOD

target = window_hash = 0
# Calculate the hash of the needle and the initial window in haystack
for char in needle:
    target = add_last(target, char)

for i in range(N2):
    window_hash = add_last(window_hash, haystack[i])
```

```
if window_hash == target:
    # Verify the actual substring matches
    if haystack[:N2] == needle:
        return 0

# Slide the window over the haystack
for right in range(N2, N1):
    left = right - N2
    window_hash = add_last(window_hash, haystack[right])
    window_hash = poll_first(window_hash, haystack[left], base_powers[N2])
    if window_hash == target:
        # Verify the actual substring matches
        if haystack[left+1:right+1] == needle:
            return right - N2 + 1

return -1
```

**Note:** If you have to do things under mod given your constraints, a hash match doesn't necessarily mean you found the string.

Note: You have to do a string equality check just to be sure.

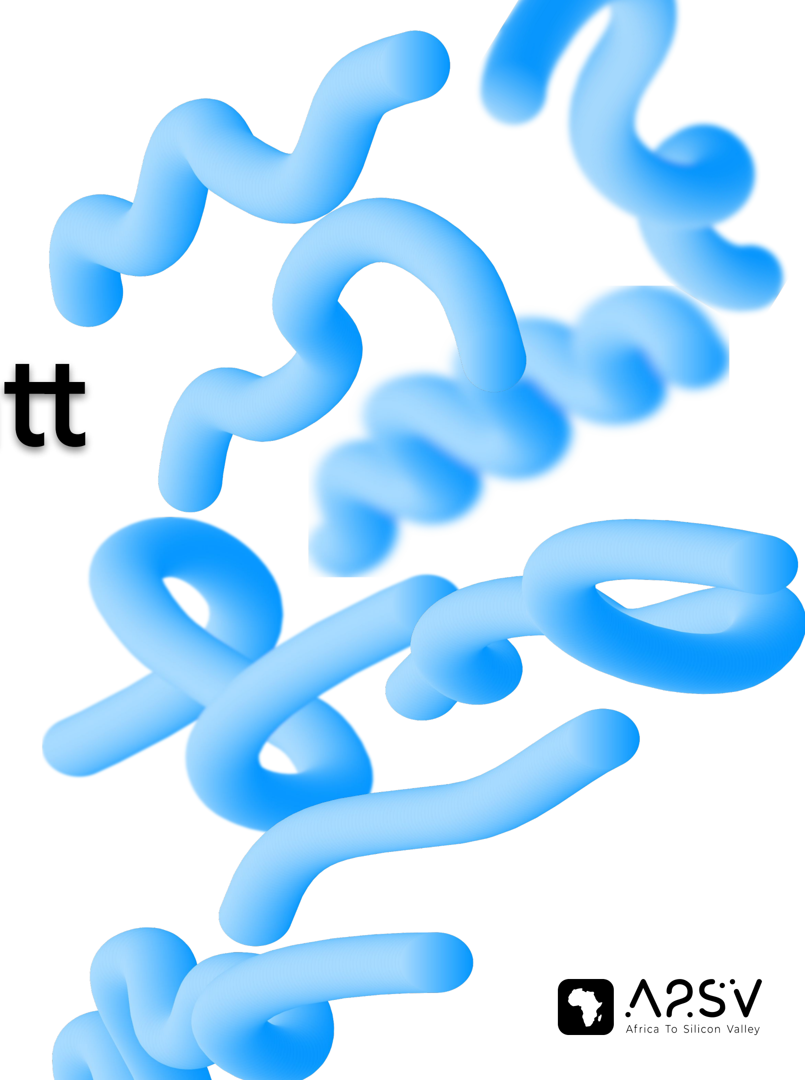
Most people don’t feel **confident** after writing a probabilistic algorithm such as **Rabin-Karp**, but the way you should see it is, if you can bring down the probability of your algorithm getting it wrong less than the probability of the hardware failing while running your code....

A2SV
Africa To Silicon Valley

you should be able to submit and be able to sleep at night.

# Knuth–Morris–Pratt algorithm

Guaranteed O(n + m) Time



This algorithm was invented by Donald Knuth, Von Pratt and independently by James Morris

Key Idea : Take advantage of the successful comparisons we make between the string and the pattern.

A2SV
Africa To Silicon Valley

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

Africa To Silicon Valley

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

The KMP algorithm wants to avoid going back in the string S and revert our progress in matching the pattern.

## So it looks for a suffix that is also a prefix in the matched substring before the mismatch

dsgwads

We know the substring `ds` exists in our string S before the mismatch. Due to this fact, the algorithm finds out how far it needs to go back in the string P to continue matching without reverting the progress that was made

In our example, we will jump back to `g` in the string P and we will not go back in our string S.

dsgwads

Africa To Silicon Valley

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

Since we don’t have any suffix that is prefix in the substring `ds`, we will now go back to the beginning in P

.A2SV
Africa To Silicon Valley

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

# Example

S = adsgwadsdsgwadsgz

P = dsgwadsgz

The algorithm mainly has two parts to achieve this efficiently.

1. Preprocessing
2. Matching

# 1. Preprocessing

## Some vocabularies first :)

**Prefix:** Substring of a string that starts from the beginning of the string. Empty string (`""`) is a prefix of every string.

- `"", "a", "ab", "aba", "abac", "abaca", "abacab"` `are prefix of` `"abacab"`
- `"", "a", "ab", "aba", "abab", "ababa", "ababab", "abababa"` are prefix of `"abababa"`

**Suffix:** Substring of a string that ends at the end of the string. Empty string (`""`) is a suffix of every string.

- `"abacab", "bacab", "acab", "cab", "ab", "b", ""` are suffix of `"abacab"`
- `"abababa", "bababa", "ababa", "baba", "aba", "ba", "a", ""` are suffix of `"abababa"`

A2SV
Africa To Silicon Valley

# 1. Preprocessing

**Proper Prefix:** Prefix that is not equal to the string itself.

- `"", "a", "ab", "aba", "abac", "abaca"` are proper prefix of `"abacab"`
- `"", "a", "ab", "aba", "abab", "ababa", "ababab"` are proper prefix of `"abababa"`

**Proper Suffix:** Suffix that is not equal to the string itself.

- `"bacab", "acab", "cab", "ab", "b", ""` are proper suffix of `"abacab"`
- `"bababa", "ababa", "baba", "aba", "ba", "a", ""` are proper suffix of `"abababa"`

**Border:** Substring of a string that is both proper prefix and proper suffix. The length of the border is often called the *Width of the Border*. Although, the term *Width* is rarely used.

- `"", "ab"` are borders of `"abacab"`
- `"", "aba", "ababa"` are borders of `"abababa"`

# 1. Preprocessing

**longest_border:** Array that stores the length of *Longest Proper Prefix that is also a Suffix* of every prefix of `string`. More precisely, `longest_border[i]` is the length of the longest border of the `string[0...i]`

# 1. Preprocessing

**The Longest Border Array (LPS, π-table, or Prefix Table)** is used in multiple algorithms. The naïve approach to built it is of $O(m^3)$ by adhering to the mathematical formula and searching for the longest proper prefix that is also a suffix, for every index.

```
for i = 1 to m-1
  for k = 0 to i
     if needle[0..k-1] == needle[i-(k-1)..i]
       longest_border[i] = k
```

However, we can follow the **greedy approach**, and can build it in **linear time**.

## 1. Preprocessing

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | | | | | | | | | |

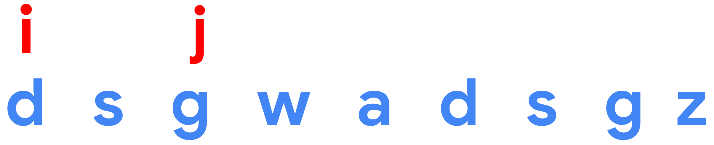
**LPS[i] = where to start matching in P after a mismatch at i + 1.**

**In other words, the length of the longest proper prefix that is a suffix in P[0….i]**

# 1. Preprocessing

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | i | j | | | | | | | |
| | d | s | g | w | a | d | s | g | z |
| LPS | 0 | | | | | | | | |

# 1. Preprocessing



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | | | | | | | |

# 1. Preprocessing

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | i | | | j | | | | | |
| | d | s | g | w | a | d | s | g | z |
| LPS | 0 | 0 | 0 | | | | | | |

# 1. Preprocessing

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | i | | | | j | | | | |
| | d | s | g | w | a | d | s | g | z |
| LPS | 0 | 0 | 0 | 0 | | | | | |

# 1. Preprocessing

|  | i |  |  |  |  | j |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  | d | s | g | w | a | d | s | g | z |
| LPS | 0 | 0 | 0 | 0 | 0 | 1 |  |  |  |

# 1. Preprocessing

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | i | | | | | j | | |
| | d | s | g | w | a | d | s | g | z |
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | | |

# 1. Preprocessing

|  |  |  |  | i |  |  |  |  | j |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | d | s | g | w | a | d | s | g | z |
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

Now that W and Z don’t match, i becomes LPS[i - 1]. This is because if we don’t have a border of three, we want to try out less wider borders before going back to zero.

# 1. Preprocessing

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
|  |  |  |  | i |  |  |  | j |
|  | a | a | a | c | a | a | a | a |
| LPS | 0 | 1 | 2 | 0 | 1 | 2 | 3 |  |

Here you can see, that c and a, don’t much and we can’t have a border of 4, but we clearly have a border of 3. That is why, we need to switch to i = LPS[i - 1] and then compare. Here LPS[i - 1] = 2.

# 1. Preprocessing

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
|  |  |  |  | i |  |  |  | j |
|  | a | a | a | c | a | a | a | a |
| LPS | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 3 |

And since a matches with a, **LPS[j] = LPS[i] + 1**

# 1. Practice: write the stub code for generating LPS table

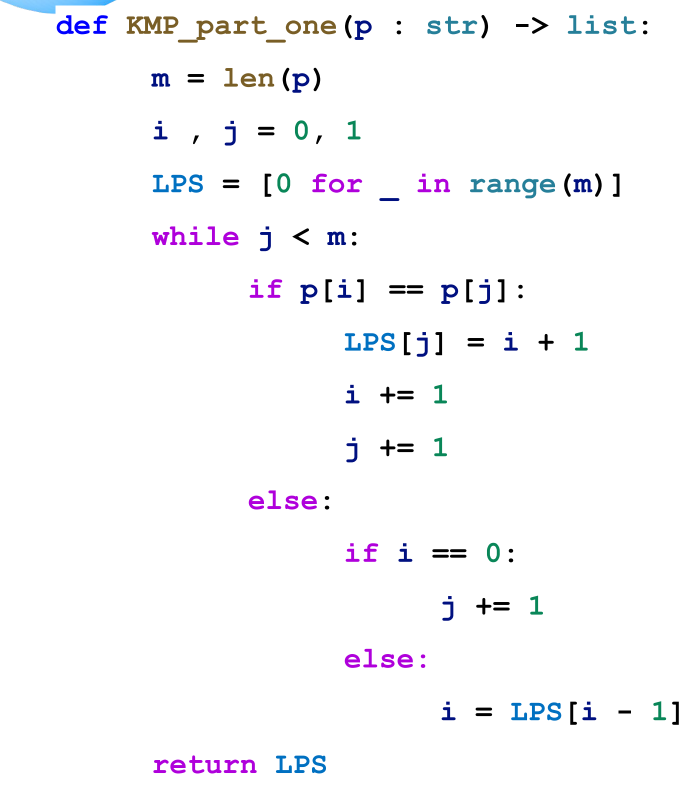
```
def KMP_part_one(p : str) -> list:
     # todo



assert KMP_part_one('aaacaaaa') == [0, 1, 2, 0, 1, 2, 3, 3]
assert KMP_part_one('dsgwadsgz') == [0, 0, 0, 0, 0, 1, 2, 3, 0]
```

A2SV
Africa To Silicon Valley

# Implementation



```
def KMP_part_one(p : str) -> list:
    m = len(p)
    i , j = 0, 1
    LPS = [0 for _ in range(m)]
    while j < m:
        if p[i] == p[j]:
            LPS[j] = i + 1
            i += 1
            j += 1
        else:
            if i == 0:
                j += 1
            else:
                i = LPS[i - 1]
    return LPS
```

# What is the time complexity of building the LPS table this way?

A2SV

Africa To Silicon Valley

Interestingly enough it’s linear.

O(length of the pattern)

A2SV
Africa To Silicon Valley

# Why O(M)?

- The total length of all "drops" (rollbacks in i (prevLPS)) is bounded by M, meaning no position is revisited unnecessarily.

- Even when i (prevLPS) drops after a mismatch, it cannot drop more than the interval already covered by j since the last time i was 0.

# 2. Matching

|  | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

|     | d | s | g | w | a | d | s | g | z |
|-----|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

## 2. Matching

|  | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |
|  |  |  |  |  | i |  |  |  |  |

S = adsgwadsdsgwadsgz

j

# 2. Matching

|  | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |
| | | | | | | | | i | |

S = adsgwadsdsgwadsgz

j

i = LPS[i - 1]

## 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

i = LPS[i - 1]

# 2. Matching

|     | d | s | g | w | a | d | s | g | z |
|-----|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

|  | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwads<u>dsgw</u>adsgz

j

# 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |
| | | | | | i | | | | |

S = adsgwadsdsgwadsgz

j

# 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

|     | d | s | g | w | a | d | s | g | z |
|-----|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

## 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

# 2. Matching

| | d | s | g | w | a | d | s | g | z |
|---|---|---|---|---|---|---|---|---|---|
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |
| | | | | | | | | | i |

S = adsgwadsdsgwadsgz

j

# 2. Matching

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  | d | s | g | w | a | d | s | g | z |
| LPS | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 0 |

i

S = adsgwadsdsgwadsgz

j

MATCH

# Implementation

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        LPS = KMP_part_one(needle)
        i, j = 0, 0
        while j < n:
            if needle[i] == haystack[j]:
                i += 1
                j += 1
            else:
                if i == 0:
                    j += 1
                else:
                    i = LPS[i - 1]
            if i >= m:
                return j - m

        return -1
```

# What is the time complexity of this Matching process?

# Once again it's linear.
# O(length of the text)

Hint: Notice the behavior of the pointers during the construction of the LPS array and compare it with the way the pointers move during the pattern matching process

# Practice Problem

## Rotate String

A2SV
Africa To Silicon Valley

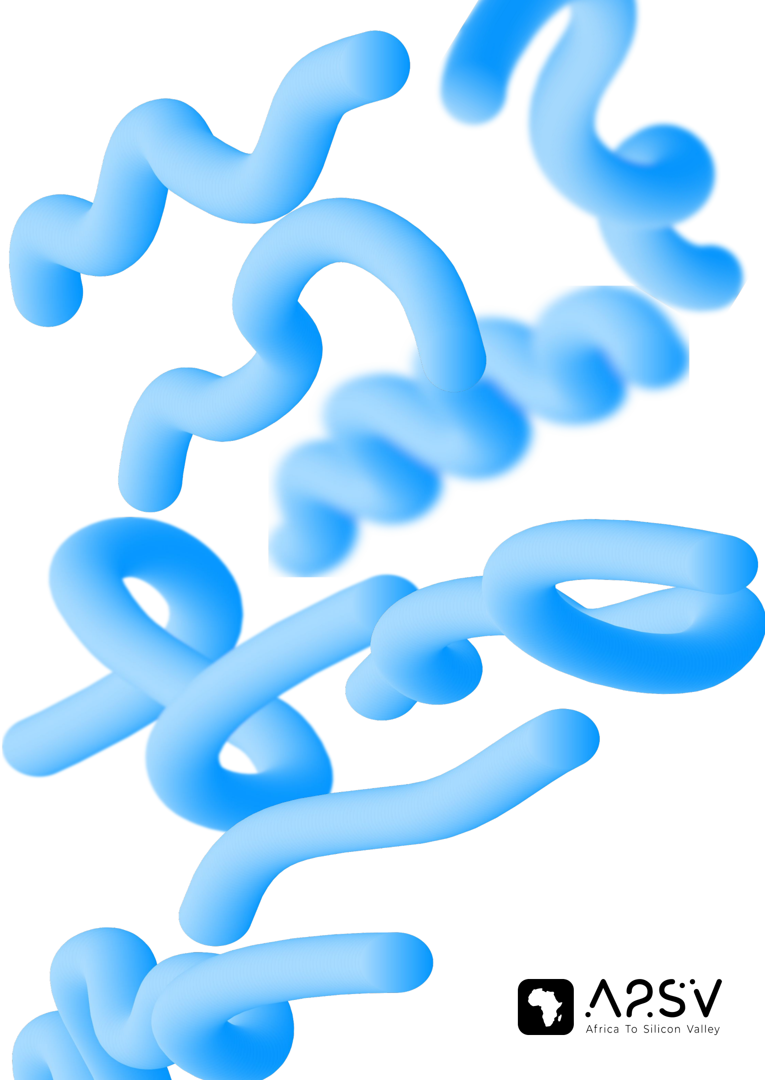
# Efficiency of the KMP algorithm

- Since the two portions of the algorithm have, respectively, complexities of O(m) and O(n), the complexity of the overall algorithm is O(m + n).
- These complexities are the same, no matter how many repetitive patterns are in P or S.

# Applications of RK and KMP

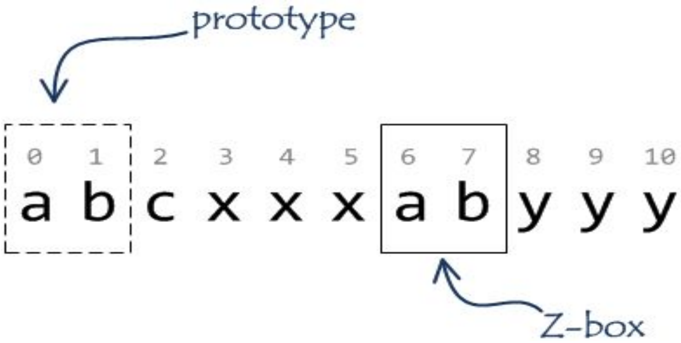
- Spell Checker
- Plagiarism Detection
- Text Editors
- Spam Filters
- Digital Forensics
- Matching DNA Sequences
- Intrusion Detection
- Search Engines
- Bioinformatics and Cheminformatics
- Information Retrieval System
- Language Syntax Checker

# Additional String Algorithms



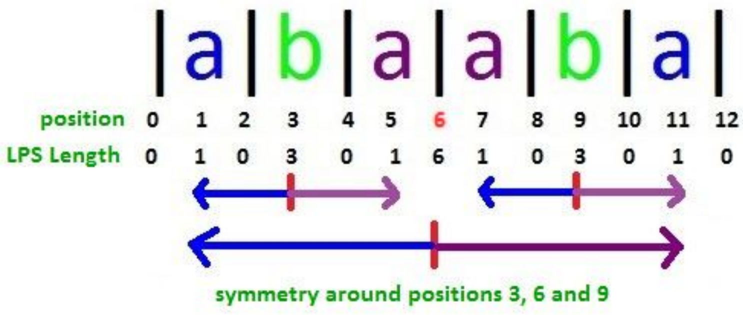
A2SV
Africa To Silicon Valley

# Z Algorithm



- Highly resembles KMP but simpler and versatile.
- Mostly used to find
    - Periodicity of a string
    - All Occurrences of a substring
- Relatively great at handling multiple patterns

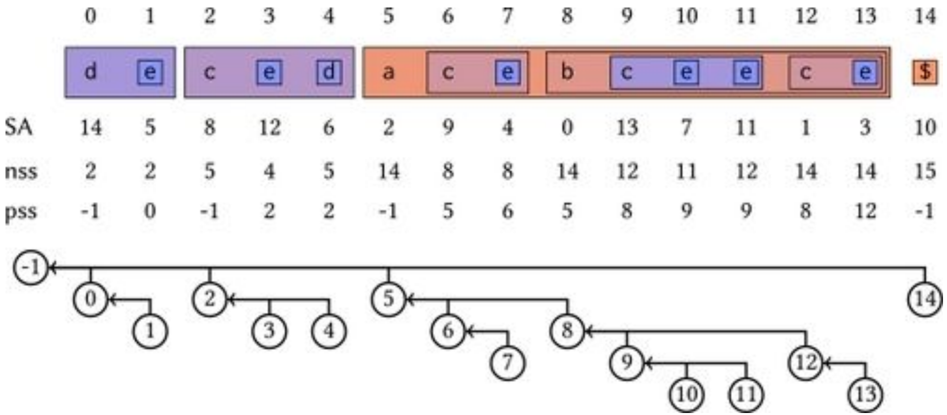
# Manacher's Algorithm


| a | b | a | a | b | a |
position 0 1 2 3 4 5 6 7 8 9 10 11 12
LPS Length 0 1 0 3 0 1 6 1 0 3 0 1 0
symmetry around positions 3, 6 and 9

- is used to find the longest palindromic substring in a given string in linear time.

- can be used to count all pairs (i, j) such that substring s[i...j] is a palindrome in linear time.

# Suffix Array



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | d | e | c | e | d | a | c | e | b | c | e | e | c | e | $ |
| SA | 14 | 5 | 8 | 12 | 6 | 2 | 9 | 4 | 0 | 13 | 7 | 11 | 1 | 3 | 10 |
| nss | 2 | 2 | 5 | 4 | 5 | 14 | 8 | 8 | 14 | 12 | 11 | 12 | 14 | 14 | 15 |
| pss | -1 | 0 | -1 | 2 | 2 | -1 | 5 | 6 | 5 | 8 | 9 | 9 | 8 | 12 | -1 |

- Efficiently solve pattern matching, lexicographic order problems, and LCP (Longest Common Prefix) queries.

- Applications: Fast substring queries, string compression, DNA sequence alignment.

# Practice Problems

- Repeated String Match
- Longest Happy Prefix
- Maximum Length of Repeated Subarray
- Repeated DNA Sequences
- Permutation in String
- Find Substring with a given hash value
- Division + LCP (easy version)

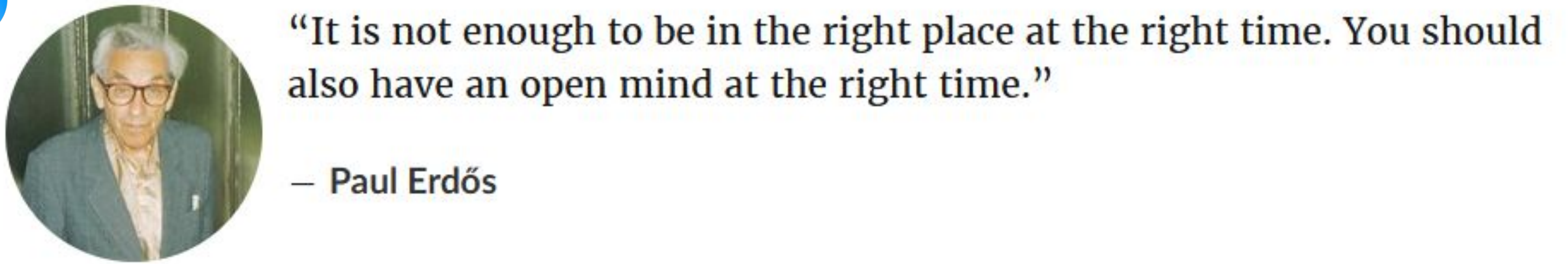
A2SV
Africa To Silicon Valley

# Resources

- Pattern Search with the Knuth-Morris-Pratt (KMP) algorithm
- Prefix function. Knuth–Morris–Pratt algorithm
- Knuth–Morris–Pratt (KMP) Pattern Matching Substring Search - First Occurrence Of Substring
- Algorithms live : Rolling hash and bloom filters
- String Searching | USACO GUIDE

A2SV
Africa To Silicon Valley

# Quote of the day



“It is not enough to be in the right place at the right time. You should also have an open mind at the right time.”

— Paul Erdős