Brute Force String Search / String Matching

1. Definition

Brute Force String Matching is the **simplest pattern matching technique**.
It checks whether a **pattern P** of length *m* appears in a **text T** of length *n* by comparing characters one-by-one.
It starts at **every possible position** in the text and checks if the pattern matches.
No preprocessing is done.
Worst-case performance is slow, but logic is very easy.

2. Key Points

- Compares pattern with text from **left to right**.
- Shifts pattern by **one position** each time.
- Time Complexity:
  - **Worst case:** O(n × m)
  - **Best case:** O(n)
- Also called **Naive String Matching**.

Algo BruteForceStringMatch(T, P)

```
// Algo BruteForceStringMatch(T, P)
// Input: Text T of length n, Pattern P of length m
// Output: Index where P starts in T (or -1 if not found)

For i = 0 to n - m:               // check each possible starting position
        j = 0
        while j < m and T[i + j] == P[j]:
               j = j + 1
        if j == m:                // complete pattern matched
```

```
            return i            // pattern found at index i
return -1                       // pattern not found
```

* Brute force String Matching

Shift pattern by next then match.

Text	a	b	b	b.	a	b	a	b	a	a	b
	0	1	2	3	4	5	6	7	8	9	10
Pattern 0th	a	b	a	a							
1st		a	b	a	a.						
2nd			a	b	a	a					
3rd				a	b	a	a.				
4th					a	b	a	b			
5th						a	b	a	~~b~~ a		
6th							a	b	a	a	

pattern found at index = 6 to 9

Text = n
Pattern = m

loop repeats $n-m+1$

→ is it efficient
Ans → no unnecessary steps

Time Complexity

Text = aaaab
Patt = b

Best Case follow Pat (W)

Best Case $O(n)$ mismatch at first case
Worst case. $O(n \times m)$
Avg Case. $O(n \times m)$

Eg MY CHOICE
T MY CHOICE IS GOOD
P : GOOD

KMP (Knuth–Morris–Pratt) String Matching

1. Definition

KMP is an efficient pattern matching algorithm that avoids rechecking characters.
It uses a **Pi table / LPS table (Longest Prefix which is also a Suffix)** to skip redundant comparisons.
Time Complexity: **O(n + m)**.

2. Why KMP is Better

- Does *not* backtrack in the text (T).
- Preprocesses pattern (P) to compute **LPS array**.
- Faster than brute force in worst case.

Algo KMP(T, P)

(with Pi/LPS table)

```
// Algo KMP(T, P)
// Input: Text T of length n, Pattern P of length m
// Output: Index where pattern starts (or -1 if not found)

1. Compute LPS[] for pattern P        // using the Pi-table
algorithm below
2. i = 0   // index for T
3. j = 0   // index for P

4. While i < n:
5.          if T[i] == P[j]:
6.                 i = i + 1
7.                 j = j + 1
8.                 if j == m:          // full pattern matched
9.                        return i - m
```

```
10.     else:                        // mismatch
11.         if j != 0:
12.             j = LPS[j - 1]   // shift using LPS table
13.         else:
14.             i = i + 1

15. return -1                        // pattern not found
```

Rabin–Karp String Matching

1. Rabin–Karp is a **pattern-matching algorithm** used to find a substring inside a text.
2. It uses **hashing** to compare the pattern with every substring of the text.
3. Instead of comparing characters one by one, it compares **hash values**, making it faster on average.
4. A **rolling hash** is used so that the next window's hash is computed efficiently.
5. If the **hash values match**, then a **direct character comparison** is done to avoid false matches.
6. Best/average-case time complexity is **O(n + m)**.
7. Worst-case occurs when many hash collisions happen → **O(nm)**.

Rabin–Karp Algorithm

```
// Algo RabinKarp(T, P)
// Input: Text T of length n, Pattern P of length m
// Output: All starting positions where P occurs in T

1. Compute hash of pattern P → hashP
2. Compute hash of first window of T (size m) → hashT
3. For i = 0 to n − m:
       if hashP == hashT:
           Compare characters of P with T[i … i+m-1]
           If all match → report match at i
```

```
        Compute next window hash using rolling hash
4. End
```

Boyer–Moore Algorithm (Bad Character Rule)

1. Definition

Boyer–Moore is an efficient string matching algorithm.
It compares the **pattern P** with text T **from right to left**.
Using the **Bad Character Table**, it shifts the pattern intelligently to skip unnecessary comparisons when a mismatch occurs.

2. Key Points

1. **Compare pattern from rightmost character** to left.
2. On mismatch, check the **Bad Character Table** for the mismatched character in the pattern.
3. Shift the pattern so that the mismatched character aligns with its **rightmost occurrence** in P.
4. If the mismatched character does **not exist in P**, shift the pattern completely past it.
5. Time Complexity:
    - Best case: **O(n/m)**
    - Worst case: **O(nm)**

Algo BoyerMoore(T, P)

```
// Input: Text T of length n, Pattern P of length m
// Output: Index of first occurrence of P in T (or -1)

1. Create BadChar table for all characters:
       BadChar[c] = -1 for all c
       For i = 0 to m-1:
            BadChar[P[i]] = i       // rightmost occurrence
```

```
2. s = 0   // shift of pattern in text
3. While s <= n - m:
       j = m - 1                   // start from right end of
pattern

       // Compare pattern with text
       While j >= 0 and P[j] == T[s + j]:
            j = j - 1

       if j < 0:                   // pattern matched
            return s
       else:                        // mismatch
            shift = max(1, j - BadChar[T[s + j]])
            s = s + shift

4. return -1   // pattern not found
```