Let’s go step by step — clear definitions, working, and why each is used.

**🔹 Sequential String Matching Algorithms**

When searching for a **pattern P** of length *m* inside a **text T** of length *n*, different algorithms exist.

**1️⃣ Brute Force (Naïve String Matching)**

**Definition:**

* Compares pattern with text one character at a time.
* If mismatch, shift pattern by **one position** and try again.

**Steps:**

1. Start at first position of text.
2. Compare characters one by one.
3. If mismatch → shift pattern by 1.
4. Repeat until match is found or end of text is reached.

**Time Complexity:**

* Worst case: **O(m·n)** (when many mismatches).
* Best case: **O(n)**.

**Usefulness:**

* Simple, easy to implement.
* Not efficient for large text.

**2️⃣ Knuth–Morris–Pratt (KMP) Algorithm**

**Definition:**

* Improves brute force by **avoiding rechecking characters** that have already matched.
* Uses **prefix function (failure function / lps array)** to decide how far to shift pattern after mismatch.

**Steps:**

1. Preprocess pattern → build LPS (Longest Proper Prefix which is also Suffix) array.
2. While matching, if mismatch happens after some matches, shift pattern so that already matched prefix doesn’t need to be checked again.

**Time Complexity:**

* Preprocessing: **O(m)**
* Searching: **O(n)**
* Total: **O(m+n)**

**Usefulness:**

* Efficient for **long texts and repeated searches**.
* Avoids unnecessary comparisons.

**3️⃣ Boyer–Moore Family of Algorithms**

**Definition:**

* Instead of scanning left-to-right, Boyer–Moore compares **from rightmost character of the pattern**.
* Uses two heuristics for skipping more characters:
  1. **Bad Character Rule** → If mismatch, shift so that the mismatched character in text aligns with last occurrence in pattern (or skip past if not present).
  2. **Good Suffix Rule** → If suffix matched but mismatch occurs, shift pattern to align with next occurrence of that suffix.

**Steps:**

1. Start aligning pattern at beginning.
2. Compare from right to left.
3. On mismatch, apply bad character or good suffix shift.
4. Skip multiple positions (much faster than brute force).

**Time Complexity:**

* Best case: **O(n/m)** (very fast).
* Worst case: **O(m·n)** (rare).
* Average: much better than brute force and KMP.

**Usefulness:**

* Very efficient in practice, especially for **large alphabets** (like English text).
* Used in text editors, search tools, etc.

**✅ Exam-Ready Summary**

* **Brute Force** → simple, but slow (**O(m·n)**).
* **KMP** → efficient (**O(m+n)**), avoids redundant checks with prefix table.
* **Boyer–Moore** → practical fastest, uses **bad character** + **good suffix** rules, skips more text.

**🔹 Brute Force Steps**

The brute force algorithm tries to **match P at every position in S** (from left to right).

**🔹 Problem**

* **Text S:** "two roads diverge in yellow wood" (length = 30)
* **Pattern P:** "roads" (length = 5)

We want to find if "roads" occurs in S.

**Step 1: Start with first 5 characters of S**

S[0..4] = "two r"  
P = "roads"  
Comparison: "two r" ≠ "roads" → mismatch

👉 Shift pattern by 1 position

**Step 2: Compare S[1..5] = "wo ro" with "roads"**

Mismatch → shift

**Step 3: Compare S[2..6] = "o roa" with "roads"**

Mismatch → shift

**Step 4: Compare S[3..7] = " road" with "roads"**

Mismatch → shift

**Step 5: Compare S[4..8] = "roads" with "roads"**

✅ Match found at position **4** (0-based index).

**🔹 Result**

* Pattern "roads" is found at index **4** in the document.
* **Number of comparisons made:**
  + Tried positions 0, 1, 2, 3 (all mismatches)
  + Success at position 4
  + Each trial compares up to length(P) = 5 characters

**🔹 Observations**

* Brute force wastes effort → notice how it **rechecked overlapping characters** every time.
* For small strings, this is fine.
* But for large texts, this becomes **slow** (O(m·n) time).

**Kumnth Moris Prat**

**🔹 Example Setup**

* **Text (S):** "abracadabracadabra" (length = 17)
* **Pattern (P):** let’s say "abracadabra" (length = 11)  
  (This is a classic KMP example.)

**🔹 Brute Force Approach**

* Start matching "abracadabra" from S[0].
  + First 11 chars match (abracadabra). ✅ Match at index 0.
* Then shift pattern by 1, recheck everything again → huge **repeated comparisons**.
* Brute force wastes time because "abra..." repeats inside text.

**🔹 KMP Approach**

**Step 1: Build LPS array for "abracadabra"**

We compute Longest Prefix = Suffix at each step:

Pattern = "abracadabra"

* "a" → 0
* "ab" → 0
* "abr" → 0
* "abra" → **1** (prefix "a" = suffix "a")
* "abrac" → 0
* "abraca" → 1
* "abracad" → 0
* "abracada" → 1
* "abracadab" → 2 ("ab")
* "abracadabr" → 3 ("abr")
* "abracadabra" → 4 ("abra")

👉 LPS = [0, 0, 0, 1, 0, 1, 0, 1, 2, 3, 4]

**Step 2: Search in Text**

* At S[0..10] → full match ✅ "abracadabra" found at index 0.
* Instead of shifting by 1, **KMP uses LPS[10] = 4** → shift pattern smartly.
* Jump directly to reuse "abra".
* Continue search → finds another match at index 7.
* And again at index 11.

**🔹 Why is this powerful?**

* Brute Force: keeps restarting from scratch → O(m·n).
* KMP: uses LPS to skip unnecessary checks → O(m+n).

✅ In "abracadabracadabra", **KMP will quickly find all occurrences of "abracadabra" at positions 0, 7, and 11**, without re-checking characters unnecessarily.

**🔹 Boyer–Moore Algorithm**

* **Text (S):** "abracadabracadabra"
* **Pattern (P):** "abracadabra"

**1. Main Idea**

* Instead of scanning **left to right** like Brute Force and KMP,  
  👉 Boyer–Moore compares the pattern **from right to left**.
* It uses two main heuristics to skip large parts of the text:
  1. **Bad Character Rule** – if mismatch happens, shift pattern so that mismatched character in text aligns with last occurrence of that char in the pattern. If it doesn’t exist in pattern → skip entire pattern length.
  2. **Good Suffix Rule** – if part of the pattern matched but then mismatched, shift so that this "good suffix" aligns with another occurrence in the pattern.

**2. Example Walkthrough**

**First alignment:**

Text: abracadabracadabra

Pattern: abracadabra

Compare from rightmost character:

* Text[10] = "a", Pattern[10] = "a" ✅
* Keep going backward until mismatch or full match.

Here: the first 11 characters match perfectly (abracadabra). ✅ Match at index 0.

**After first match:**

Now shift pattern. Suppose mismatch happens later (say when checking at index 1).

* Bad character rule:  
  If we mismatched on "b" in text but "b" appears last at position 1 in pattern → shift accordingly.
* Good suffix rule:  
  If suffix "abra" matched but mismatch happens earlier, align that "abra" with another "abra" in the pattern.

👉 This makes Boyer–Moore very efficient for **English text** or large alphabets, because mismatches allow **big jumps** (sometimes skipping 5–10 characters at once).

**3. Complexity**

* Best case: **O(n/m)** → very fast (skips big portions).
* Worst case: **O(m·n)** (but rare, mostly for repetitive texts like "aaaaaa...").

✅ For "abracadabracadabra" and "abracadabra", Boyer–Moore will also find matches at indices **0, 7, 11**, but it usually checks fewer characters than both Brute Force and KMP.

📌 **Summary so far:**

* **Brute Force:** Simple, but slow (checks every position).
* **KMP:** Avoids backtracking using LPS.
* **Boyer–Moore:** Uses clever heuristics to skip large portions → fastest in practice for large text.

**🔹 Boyer–Moore Step by Step**

👉 First, remember BM compares **from right → left**.  
We’ll use the **Bad Character Rule** here to show the shifts.

**Step 1: Initial alignment**

Text: abracadabracadabra

Pattern: abracadabra

* Compare last chars:  
  S[10] = "a", P[10] = "a" ✅  
  S[9] = "r", P[9] = "r" ✅  
  … all match → ✅ full match at index **0**.

**Step 2: Shift after match**

Move pattern to start at index 11 (since P has length 11, no overlap possible here).

Text: abracadabracadabra

Pattern: abracadabra

* Compare from right:  
  S[21] doesn’t exist → stop.  
  But since text length is 17, pattern has gone past → search ends.

**🔹 Another Case (with mismatch)**

Suppose instead we were searching for **Pattern = "cadabra"** (len = 7).

**Step 1: Alignment**

Text: abracadabracadabra

Pattern: cadabra

* Compare from right:  
  S[6] = "a", P[6] = "a" ✅  
  S[5] = "d", P[5] = "r" ❌ mismatch!

**Bad character = "d"** in text.

* "d" exists in pattern at position 3 (0-indexed).
* Shift pattern so "d" in text aligns with "d" in pattern.

**Step 2: Shift**

Text: abracadabracadabra

Pattern: cadabra

Now check again:

* From right to left → "cadabra" found at index **4**. ✅

**🔹 Key Takeaways**

* BM checks **from rightmost character first**.
* On mismatch → **Bad Character Rule** shifts the pattern (sometimes a big jump).
* If partial suffix matched → **Good Suffix Rule** helps skip further.
* Much faster in practice than Brute Force & often faster than KMP.