## **What is Pattern Searching?**

**Pattern searching** is a fundamental operation in data structures and algorithms. It involves finding a specific pattern or substring within a given text or string. Efficient pattern searching algorithms are essential for various applications, including text processing, information retrieval, and bioinformatics.

## **Important Pattern Searching Algorithms:**

### **1. Naive String Matching**

The naive string matching algorithm is the simplest pattern searching algorithm. It compares the pattern with every possible substring of the text. The **time complexity** of this algorithm is **O(mn)**, where **m** is the length of the pattern and **n** is the length of the text.

### **2. Knuth-Morris-Pratt (KMP) Algorithm**

The **KMP algorithm** is a heuristic algorithm that uses a precomputed failure function to skip unnecessary comparisons. The failure function indicates the length of the **longest proper prefix** of the pattern that is also a **suffix** of the current substring. The **time complexity** of the KMP algorithm is **O(m + n)**, where **m** is the length of the pattern and **n** is the length of the text.

### **3. Rabin-Karp Algorithm**

The **Rabin-Karp algorithm** is a heuristic algorithm that uses hashing to compare the pattern with the text. It computes a hash value for the pattern and for each substring of the text. If the hash values match, it performs a character-by-character comparison to confirm the match. The **time complexity** of the Rabin-Karp algorithm is **O(m + n)**, where **m** is the length of the pattern and **n** is the length of the text.

# **Boyer Moore Algorithm for Pattern Searching**

Boyer Moore algorithm also preprocesses the pattern.   
Boyer Moore is a combination of the following two approaches.

1. **Bad Character Heuristic**
2. **Good Suffix Heuristic**

Both of the above heuristics can also be used independently to search a pattern in a text.

## **Applications of Pattern Searching**

Pattern searching algorithms have numerous applications, including:

* **Text Processing:** Searching for keywords in a document, finding and replacing text, spell checking, and plagiarism detection.
* **Information Retrieval:** Finding relevant documents in a database, web search, and data mining.
* **Bioinformatics:** Searching for DNA sequences in a genome, protein analysis, and gene expression analysis.
* **Network Security:** Detecting malicious patterns in network traffic, intrusion detection, and malware analysis.
* **Data Mining:** Identifying patterns in large datasets, customer segmentation, and fraud detection.

# **Naive algorithm for Pattern Searching**

Given **text** string with length **n** and a **pattern** with length **m,** the task is to prints all occurrences of **pattern**in **text**.  
**Note:** You may assume that n>m.

**Examples:**

***Input:*** *text = “THIS IS A TEST TEXT”, pattern = “TEST”****Output:*** *Pattern found at index 10*

***Input:*** *text = “AABAACAADAABAABA”, pattern = “AABA”****Output:*** *Pattern found at index 0, Pattern found at index 9, Pattern found at index 12*

***Time Complexity:*** *O(N2)****Auxiliary Space:*** *O(1)*

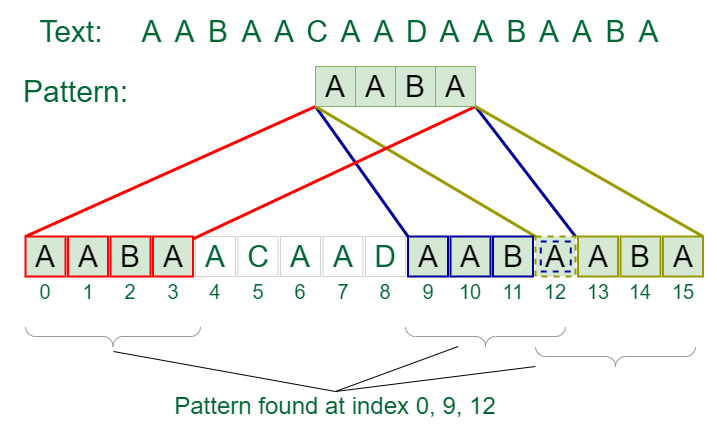
## ***Complexity Analysis of Naive algorithm for Pattern Searching:***

### ***Best Case: O(n)***

* *When the* ***pattern*** *is found at the very beginning of the* ***text*** *(or very early on).*
* *The algorithm will perform a constant number of comparisons, typically on the order of* ***O(n)*** *comparisons, where n is the length of the* ***pattern****.*

### ***Worst Case: O(n2)***

* *When the* ***pattern*** *doesn’t appear in the* ***text*** *at all or appears only at the very end.*
* *The algorithm will perform* ***O((n-m+1)\*m)*** *comparisons, where* ***n*** *is the length of the* ***text*** *and* ***m*** *is the length of the* ***pattern****.*
* *In the worst case, for each position in the* ***text****, the algorithm may need to compare the entire* ***pattern*** *against the text.*



# **Rabin-Karp Algorithm for Pattern Searching**

Given a text T**[0. . .n-1]** and a pattern P**[0. . .m-1]**, write a function search(char P[], char T[]) that prints all occurrences of P[] present in T[] using Rabin Karp algorithm. You may assume that **n > m**.

**Examples:**

***Input:*** *T[] = “THIS IS A TEST TEXT”, P[] = “TEST”****Output:*** *Pattern found at index 10*

***Input:*** *T[] = “AABAACAADAABAABA”, P[] = “AABA”****Output:*** *Pattern found at index 0  
 Pattern found at index 9  
 Pattern found at index 12*

## **Rabin-Karp Algorithm:**

In the [Naive String Matching](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/) algorithm, we check whether every substring of the text of the pattern’s size is equal to the pattern or not one by one.

*Like the Naive Algorithm, the Rabin-Karp algorithm also check every substring. But unlike the Naive algorithm, the Rabin Karp algorithm matches the* ***hash value*** *of the* ***pattern*** *with the* ***hash value*** *of the current substring of* ***text****, and if the* ***hash values*** *match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for the following strings.*

* ***Pattern*** *itself*
* *All the substrings of the* ***text*** *of length* ***m*** *which is the size of pattern.*

## **How is Hash Value calculated in Rabin-Karp?**

**Hash value** is used to efficiently check for potential matches between a **pattern** and substrings of a larger **text**. The hash value is calculated using a **rolling hash function**, which allows you to update the hash value for a new substring by efficiently removing the contribution of the old character and adding the contribution of the new character. This makes it possible to slide the pattern over the **text** and calculate the hash value for each substring without recalculating the entire hash from scratch.

Here’s how the hash value is typically calculated in Rabin-Karp:

**Step 1:** Choose a suitable **base** and a **modulus**:

* Select a prime number ‘**p**‘ as the modulus. This choice helps avoid overflow issues and ensures a good distribution of hash values.
* Choose a base ‘**b**‘ (usually a prime number as well), which is often the size of the character set (e.g., 256 for ASCII characters).

**Step 2:** Initialize the hash value:

* Set an initial hash value ‘**hash**‘ to **0**.

**Step 3:** Calculate the initial hash value for the **pattern**:

* Iterate over each character in the **pattern** from **left** to **right**.
* For each character **‘c’** at position **‘i’**, calculate its contribution to the hash value as **‘c \* (bpattern\_length – i – 1) % p’** and add it to ‘**hash**‘.
* This gives you the hash value for the entire **pattern**.

**Step 4:** Slide the pattern over the **text**:

* Start by calculating the hash value for the first substring of the **text** that is the same length as the **pattern**.

**Step 5:** Update the hash value for each subsequent substring:

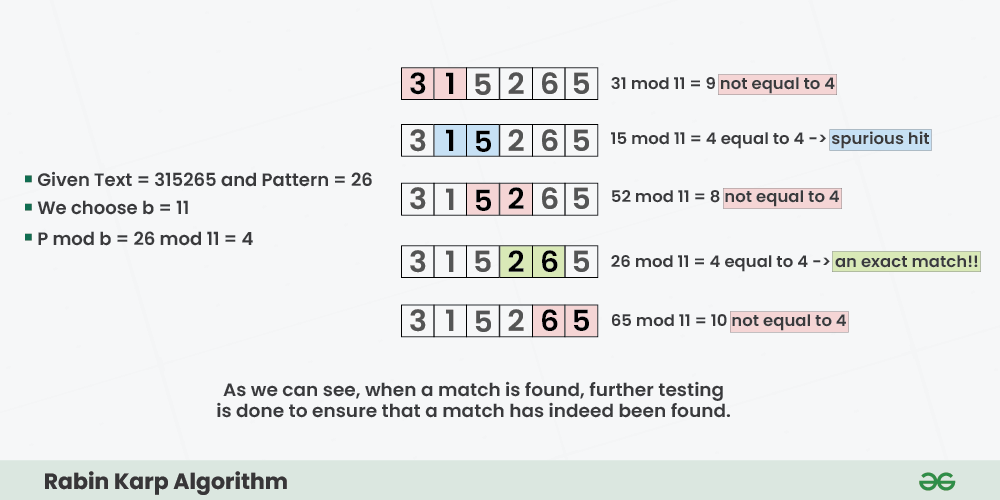
* To slide the **pattern** one position to the right, you remove the contribution of the leftmost character and add the contribution of the new character on the right.
* The formula for updating the hash value when moving from position **‘i’** to **‘i+1’** is:

**hash = (hash - (text[i - pattern\_length] \* (bpattern\_length - 1)) % p) \* b + text[i]**

**Step 6:** Compare hash values:

* When the hash value of a substring in the **text** matches the hash value of the **pattern**, it’s a **potential match**.
* If the hash values match, we should perform a character-by-character comparison to confirm the match, as [hash collisions](https://www.geeksforgeeks.org/what-is-hashing/) can occur.

Below is the Illustration of above algorithm:



Step-by-step approach:

* Initially calculate the hash value of the pattern.
* Start iterating from the starting of the string:
  + Calculate the hash value of the current substring having length **m**.
  + If the hash value of the current substring and the pattern are same check if the substring is same as the pattern.
  + If they are same, store the starting index as a valid answer. Otherwise, continue for the next substrings.
* Return the starting indices as the required answer.

**Time Complexity:**

* The average and best-case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm).
* The worst case of the Rabin-Karp algorithm occurs when all characters of pattern and text are the same as the hash values of all the substrings of T[] match with the hash value of P[].

**Auxiliary Space:** O(1)

## **Limitations of Rabin-Karp Algorithm**

**Spurious Hit:** When the hash value of the pattern matches with the hash value of a window of the text but the window is not the actual pattern then it is called a **spurious hit**. Spurious hit increases the time complexity of the algorithm. In order to minimize spurious hit, we use good [hash function](https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/). It greatly reduces the spurious hit.