**Brute-Force & Naive Algorithm**

The terms "brute-force" and "naive" are often used interchangeably, especially in the context of algorithms. Both refer to straightforward, simple methods of solving problems without employing more complex strategies or optimizations.

**### Brute-Force Algorithm:**

A brute-force algorithm exhaustively searches through all possible solutions without employing any specific knowledge about the problem. It relies on systematic trial and error. While this approach is generally simple to implement, it may not be the most efficient for all problems.

**### Naive Algorithm:**

The term "naive" is often used synonymously with "brute-force" in the context of algorithms. A naive algorithm doesn't make use of advanced techniques or optimizations. It solves a problem in a straightforward manner, typically following the most obvious or intuitive approach.

**### Brute-Force and Naive Algorithm in String Matching:**

In the context of string matching, a brute-force or naive algorithm would involve checking all possible positions in the main string for a match with the substring. If a match is found, it's reported; otherwise, the search continues.

Example:

Consider finding the substring "pattern" in the main string "This is a pattern matching example." Using a brute-force approach, you would check each position for a match, character by character.

**This is a pattern matching example.**

**^^^^^^^**

In this example, the substring "pattern" is found starting at position 10.

While this approach is simple to understand and implement, more sophisticated algorithms like the Knuth-Morris-Pratt (KMP) algorithm or the Boyer-Moore algorithm can offer better performance in certain scenarios by avoiding unnecessary character comparisons.

**find all occurrences of the substring in the given string and print their respective indices**

**void findAllOccurrences(String s, String subS) {**

**int n = s.length();**

**int m = subS.length();**

**int i = 0;**

**while (i <= n - m) {**

**int j;**

**for (j = 0; j < m; j++) {**

**if (s.charAt(i + j) != subS.charAt(j)) {**

**break;**

**}**

**}**

**if (j == m) {**

**int endIndex = i + m - 1;**

**System.out.println(subS + " found in index " + i + " to " + endIndex);**

**i = endIndex + 1; // Move to the next character after the found substring**

**} else { i++;}}**

**if (i == 0) {**

**System.out.println(subS + " not found"); } }**

The time complexity of this algorithm is O((n-m+1) \* m), where n is the length of the main string and m is the length of the substring. The worst-case scenario happens when you have to check almost all positions for a match.