String Matching Algorithms

- String matching operation is a core part in many text processing applications. The objective of this algorithm is to find pattern P from given text T. Typically |P|<< |T|. In the design of compilers and text editors, string matching operation is crucial. So locating P in T efficiently is very important.
- The problem is defined as follows: “Given some text string T[1….n] of size n, find all occurrences of pattern P[1…m] of size m in T.”

String Matching Algorithms can broadly be classified into two types of algorithms –

- Exact String Matching Algorithms
- Approximate String Matching Algorithms

Exact String Matching Algorithms:

Exact string matching algorithms is to find one, several, or all occurrences of a defined string (pattern) in a large string (text or sequences) such that each matching is perfect. All alphabets of patterns must be matched to corresponding matched subsequence.

Approximate String Matching Algorithms:

Approximate String Matching Algorithms (also known as Fuzzy String Searching) searches for substrings of the input string.

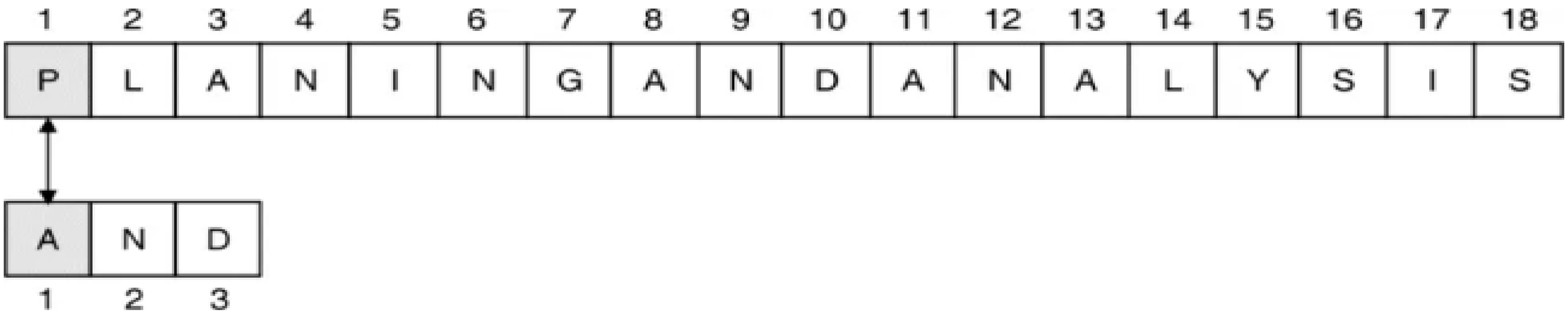
Applications of String Matching Algorithms:

- Plagiarism Detection
- Bioinformatics and DNA Sequencing
- Digital Forensics
- Spelling Checker
- Spam filters
- Search engines or content search in large databases

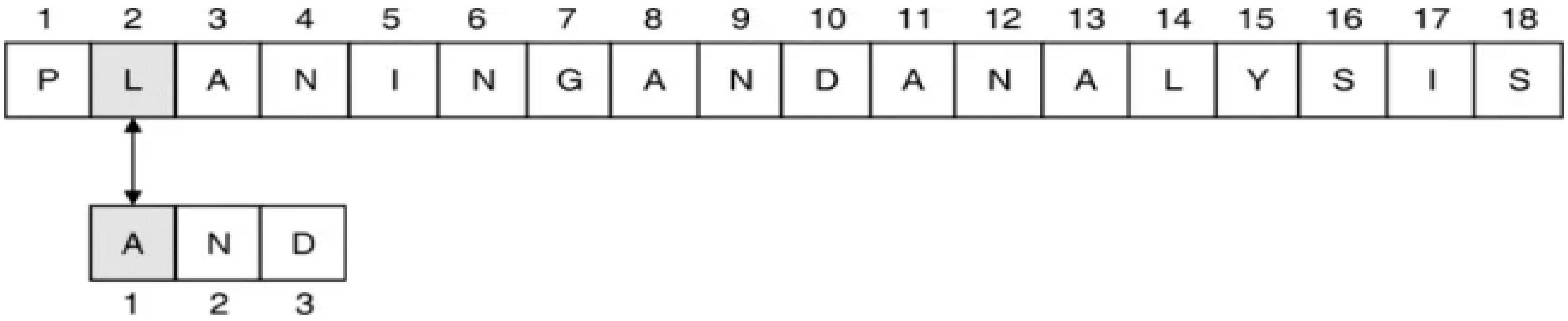
Naive String Matching Algorithm

- This is simple and efficient brute force approach. It compares the first character of pattern with searchable text. If a match is found, pointers in both strings are advanced. If a match is not found, the pointer to text is incremented and pointer of the pattern is reset. This process is repeated till the end of the text.
- The naïve approach does not require any pre-processing. Given text T and pattern P, it directly starts comparing both strings character by character.
- After each comparison, it shifts pattern string *one position* to the right.
- Following example illustrates the working of naïve string matching algorithm. Here,
  T = PLANINGANDANALYASIS and P = AND
- Here, t_i and p_j are indices of text and pattern respectively.

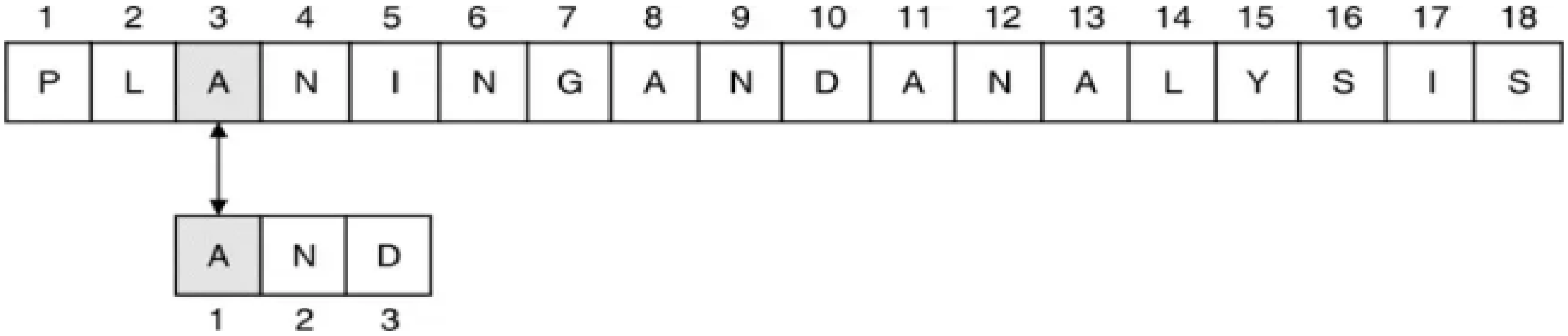
Step 1: T[1] ≠ P[1], so advance text pointer, i.e. t_i++.



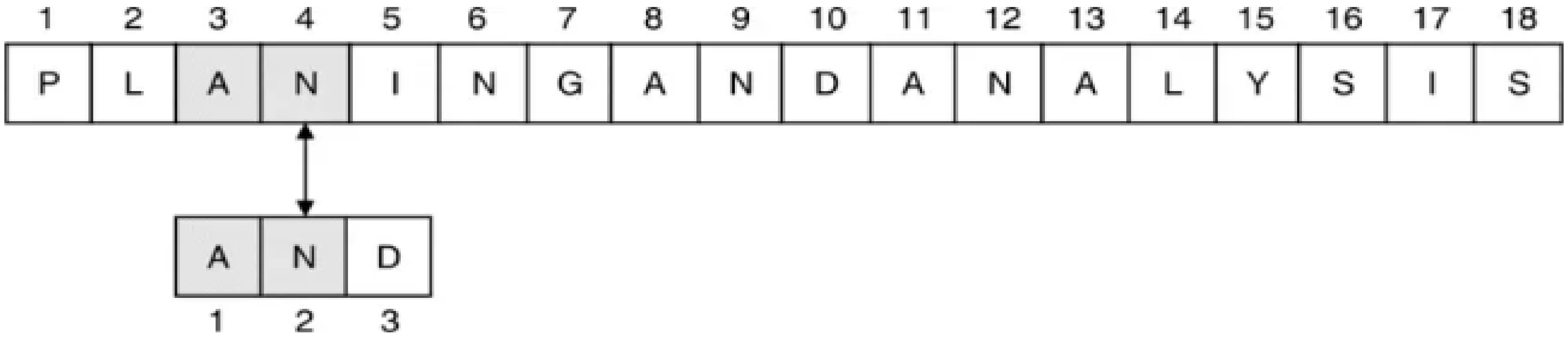
Step 2 : T[2] ≠ P[1], so advance text pointers i.e. t_i++



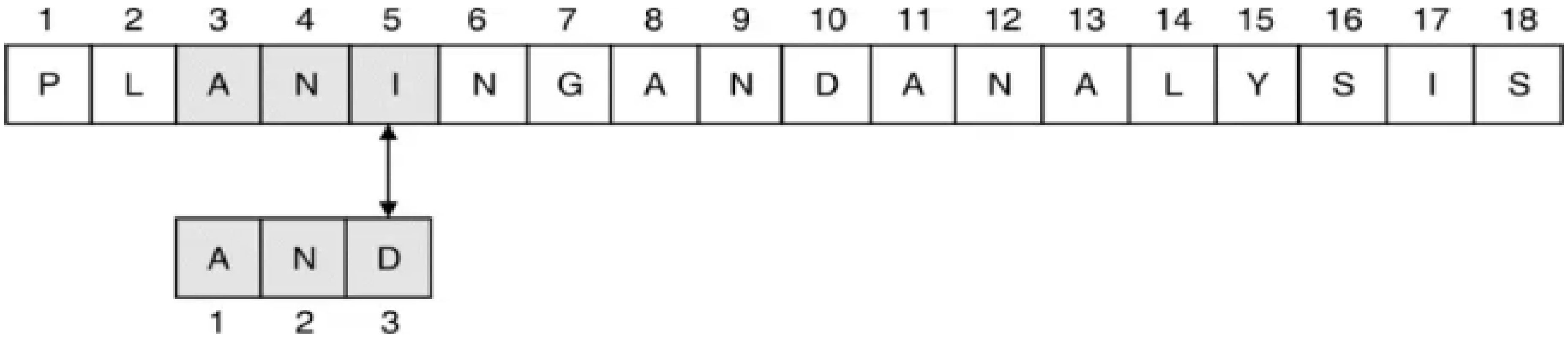
Step 3 : T[3] = P[1], so advance both pointers i.e. t_i++, p_j++



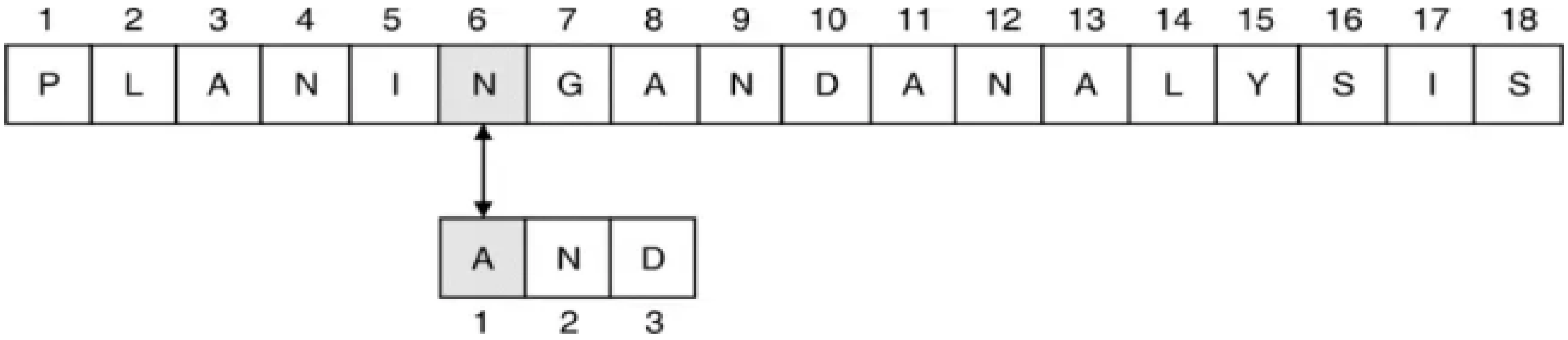
Step 4 : T[4] = P[2], so advance both pointers, i.e. t_i++, p_j++



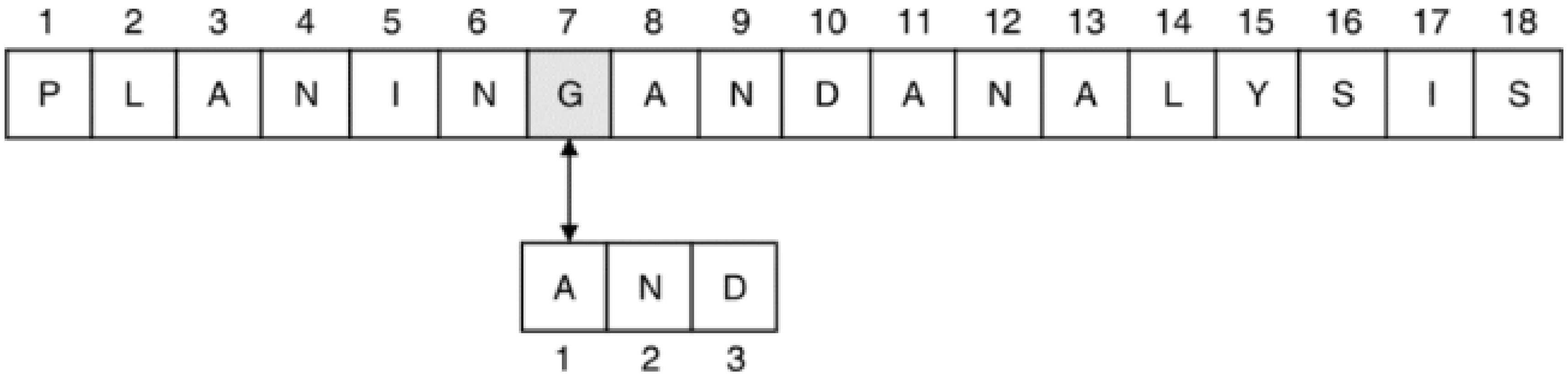
Step 5 : $T[5] \neq P[3]$, so advance text pointer and reset pattern pointer, i.e. t_i++, $p_j = 1$



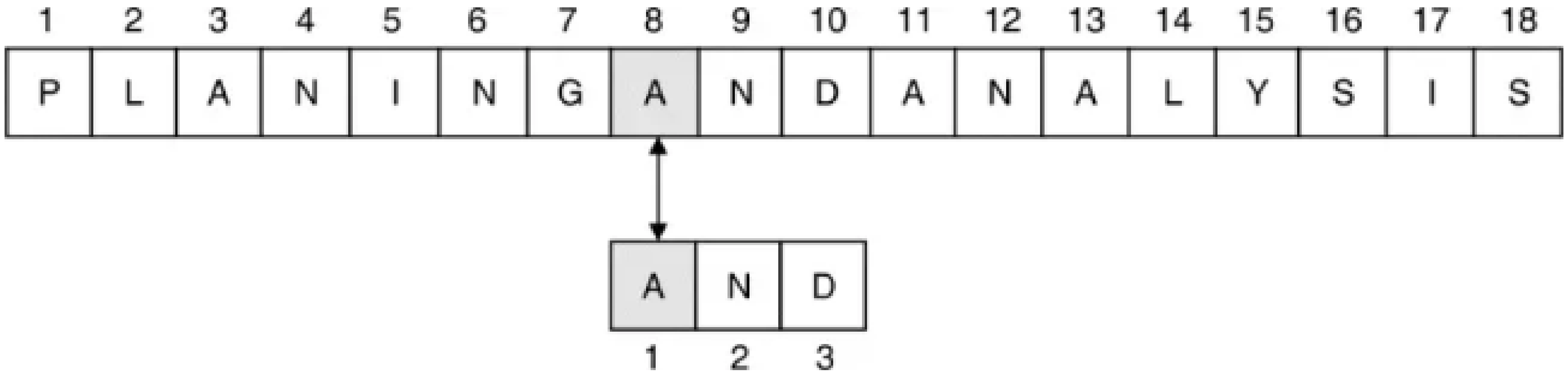
Step 6 : $T[6] \neq P[1]$, so advance text pointer, i.e. t_i++



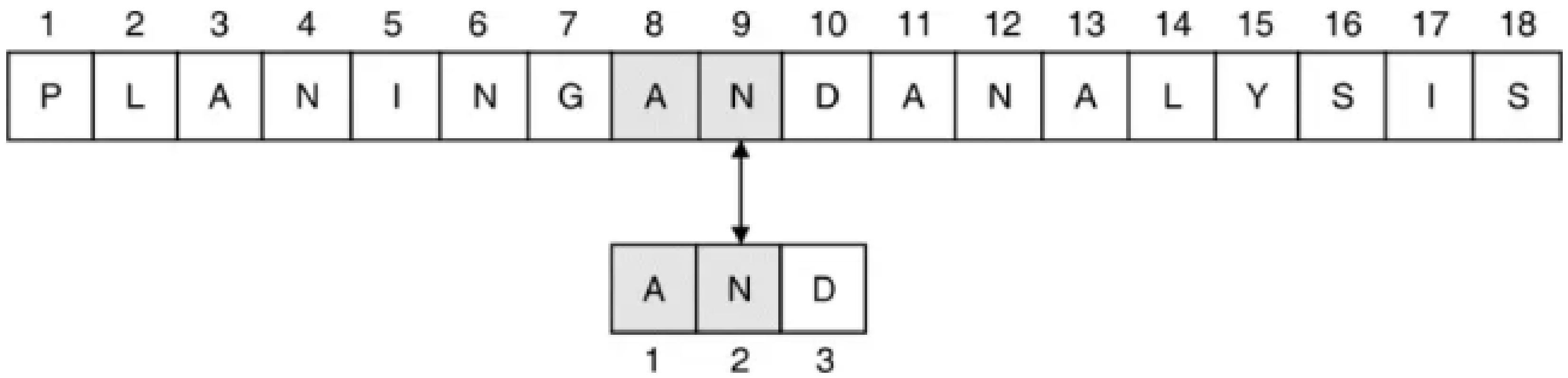
Step 7: T[7] ≠ P[1], so advance text pointer i.e. t_i++



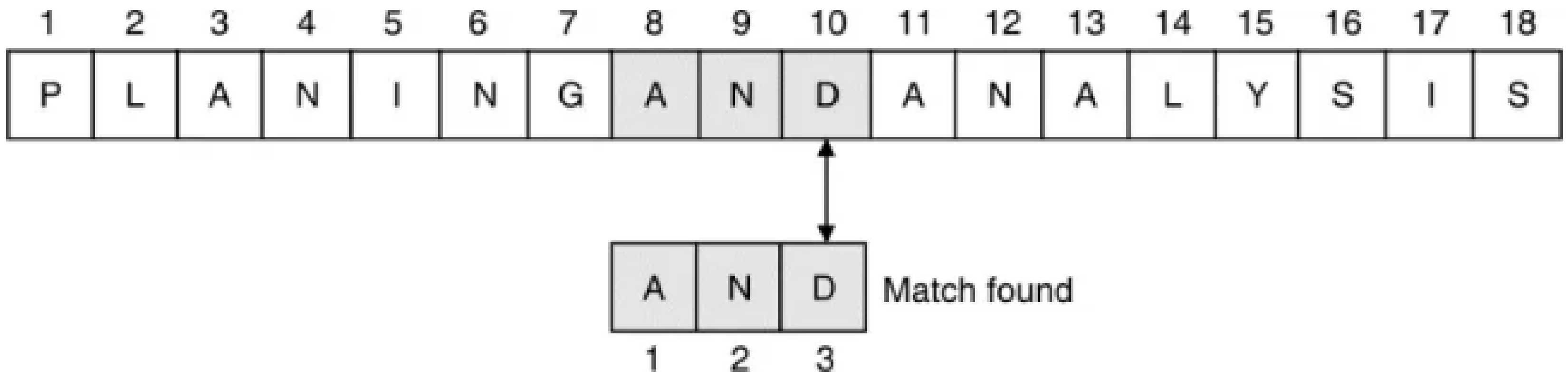
Step 8 : T[8] = P[1], so advance both pointers, i.e. t_i++, p_j++



Step 9 : T[9] = P[2], so advance both pointers, i.e. t_i++, p_j++



Step 10 : T[10] = P[3], so advance both pointers, i.e. t_i++, p_j++



This process continues till the possible comparison in the text string.

Algorithm

- Algorithm NAÏVE_STRING_MATCHING(T, P)
- // T is the text string of length n
- // P is the pattern of length m

- for i ← 0 to n – m do
-     if P[1… m] == T[i+1…i+m] them
-         print “Match Found”
-     end
- End

- Complexity = O(n*m)