ADVANCE DATA STRUCTURE NOTES

UNIT-IV

String Matching Algorithms

String matching algorithms have greatly influenced computer science and play an essential role in various real-world problems. It helps in performing time-efficient tasks in multiple domains. These algorithms are useful in the case of searching a string within another string. String matching is also used in the Database schema, Network systems.

Let us look at a few string matching algorithms before proceeding to their applications in real world. String Matching Algorithms can broadly be classified into two types of algorithms –

1. Exact String Matching Algorithms
2. 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. These are further classified into four categories:

1. **Algorithms based on character comparison:**

- Naive Algorithm: It slides the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.
- KMP (Knuth Morris Pratt) Algorithm: The idea is whenever a mismatch is detected, we already know some of the characters in the text of the next window. So, we take advantage of this information to avoid matching the characters that we know will anyway match.
- Boyer Moore Algorithm: This algorithm uses best heuristics of Naive and KMP algorithm and starts matching from the last character of the pattern.
- Using the Trie data structure: It is used as an efficient information retrieval data structure. It stores the keys in form of a balanced BST.

2. **Deterministic Finite Automaton (DFA) method:**

- Automaton Matcher Algorithm: It starts from the first state of the automata and the first character of the text. At every step, it considers next character of text, and look for the next state in the built finite automata and move to a new state.

3. **Algorithms based on Bit (parallelism method):**

- Aho-Corasick Algorithm: It finds all words in O(n + m + z) time where n is the length of text and m be the total number characters in all words and z is total number of occurrences of words in text. This algorithm forms the basis of the original Unix command fgrep.

4. Hashing-string matching algorithms:

- Rabin Karp Algorithm: It matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters.

Approximate String Matching Algorithms:

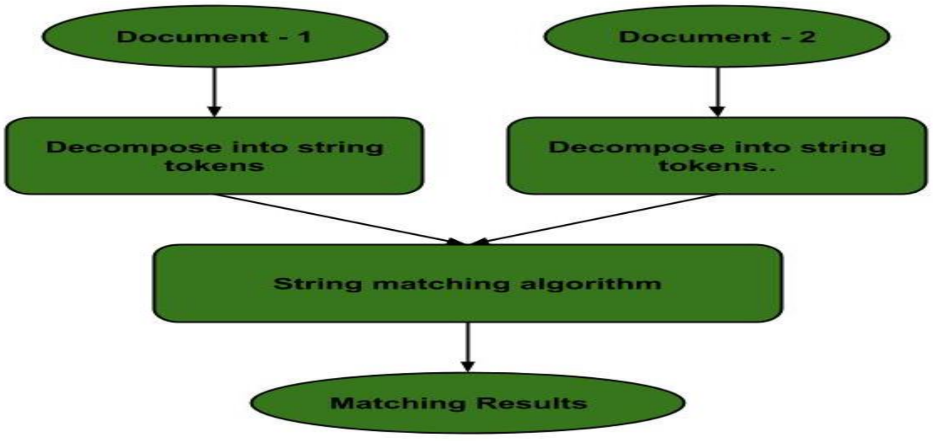
Approximate String Matching Algorithms (also known as Fuzzy String Searching) searches for substrings of the input string. More specifically, the approximate string matching approach is stated as follows: Suppose that we are given two strings, text T[1…n] and pattern P[1…m]. The task is to find all the occurrences of patterns in the text whose edit distance to the pattern is at most k. Some well known edit distances are – Levenshtein edit distance and Hamming edit distance.

These techniques are used when the quality of the text is low, there are spelling errors in the pattern or text, finding DNA subsequences after mutation, heterogeneous databases, etc. Some approximate string matching algorithms are:

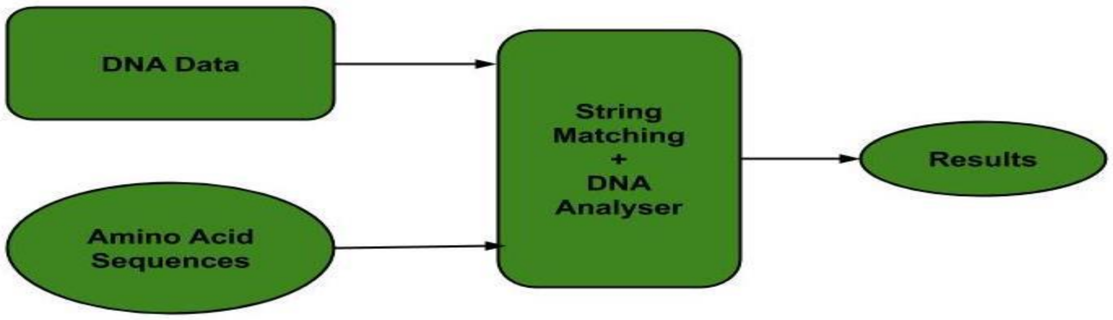
- **Naive Approach:** It slides the pattern over text one by one and check for approximate matches. If they are found, then slides by 1 again to check for subsequent approximate matches.
- Sellers Algorithm (Dynamic Programming)
- Shift or Algorithm (Bitmap Algorithm)

Applications of String Matching Algorithms:

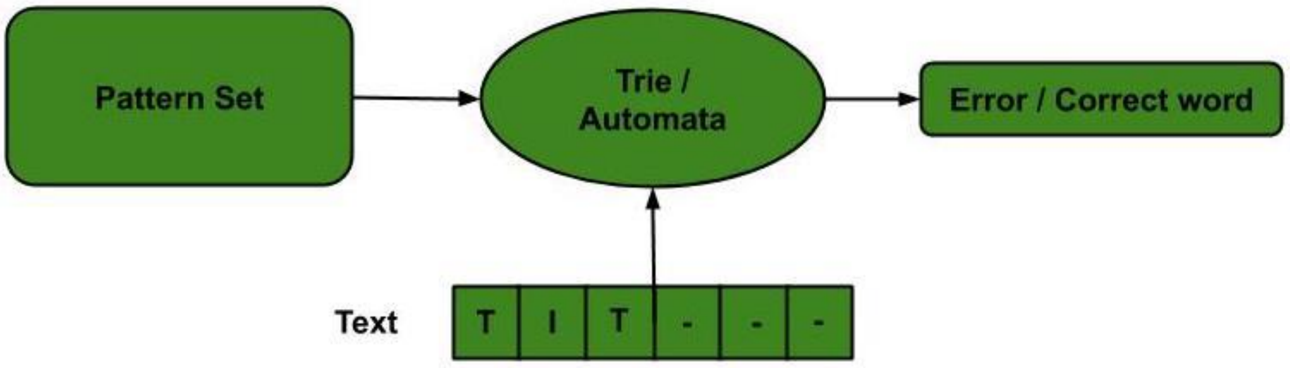
- **Plagiarism Detection:** The documents to be compared are decomposed into string tokens and compared using string matching algorithms. Thus, these algorithms are used to detect similarities between them and declare if the work is plagiarized or original.


Document - 1
Document - 2
Decompose into string tokens
Decompose into string tokens..
String matching algorithm
Matching Results

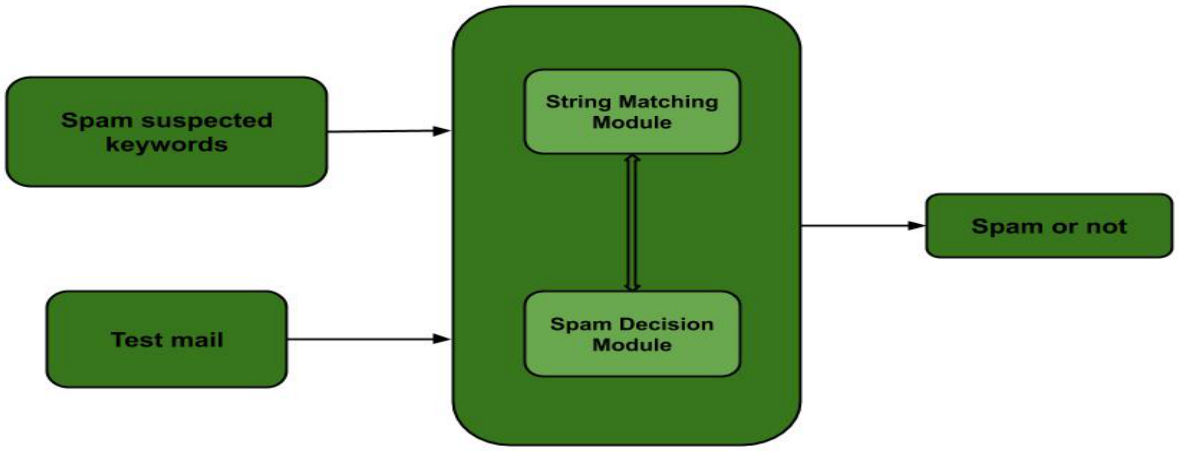
- **Bioinformatics and DNA Sequencing:** Bioinformatics involves applying information technology and computer science to problems involving genetic sequences to find DNA patterns. String matching algorithms and DNA analysis are both collectively used for finding the occurrence of the pattern set.



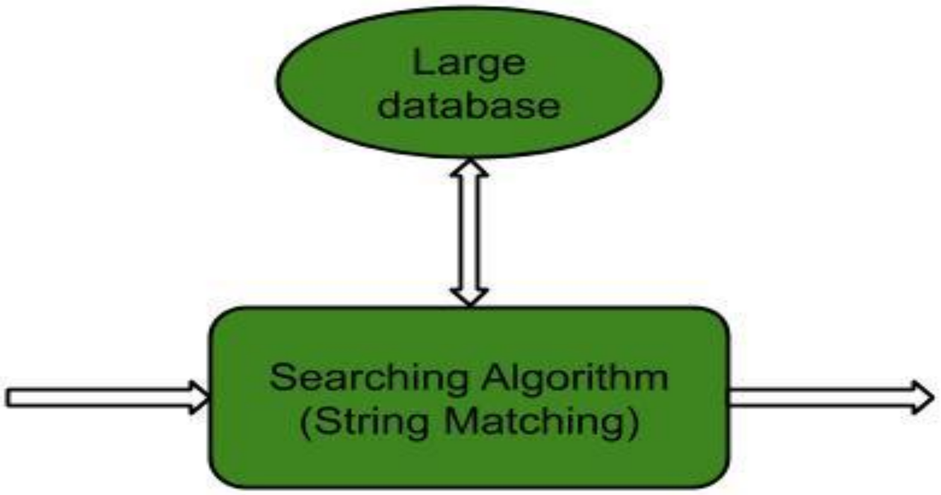
- **Digital Forensics:** String matching algorithms are used to locate specific text strings of interest in the digital forensic text, which are useful for the investigation.
- **Spelling Checker:** Trie is built based on a predefined set of patterns. Then, this trie is used for string matching. The text is taken as input, and if any such pattern occurs, it is shown by reaching the acceptance
state.



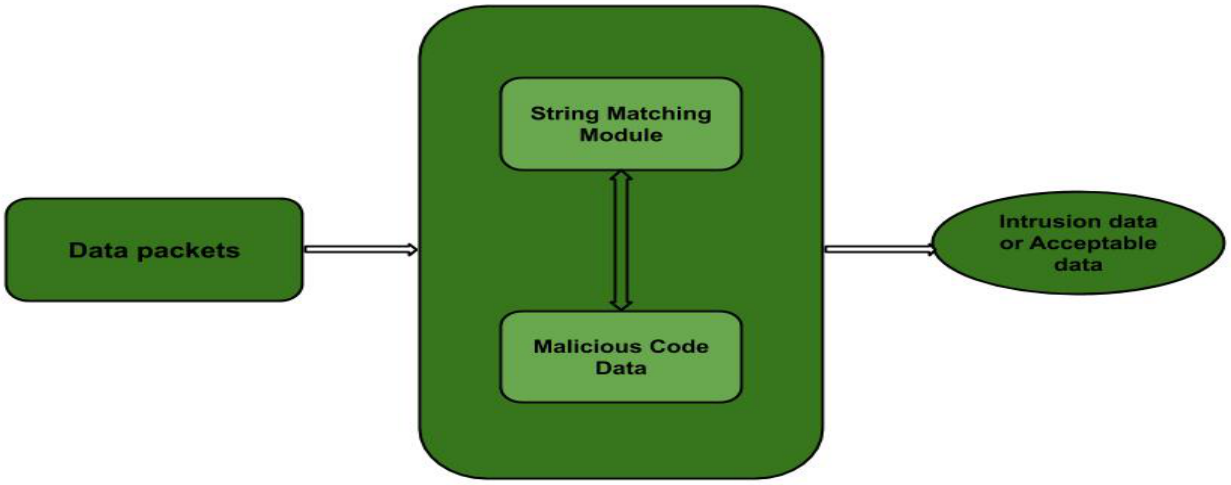
- **Spam filters:** Spam filters use string matching to discard the spam. For example, to categorize an email as spam or not, suspected spam keywords are searched in the content of the email by string matching algorithms. Hence, the content is classified as spam or not.



- **Search engines or content search in large databases:** To categorize and organize data efficiently, string matching algorithms are used. Categorization is done based on the search keywords. Thus, string matching algorithms make it easier for one to find the information they are searching for.



- **Intrusion Detection System:** The data packets containing intrusion-related keywords are found by applying string matching algorithms. All the malicious code is stored in the database, and every incoming data is compared with stored data. If a match is found, then the alarm is generated. It is based on exact string matching algorithms where each intruded packet must be detected.



Brute force Algorithm

A brute force approach is an approach that finds all the possible solutions to find a satisfactory solution to a given problem. The brute force algorithm tries out all the possibilities till a satisfactory solution is not found.

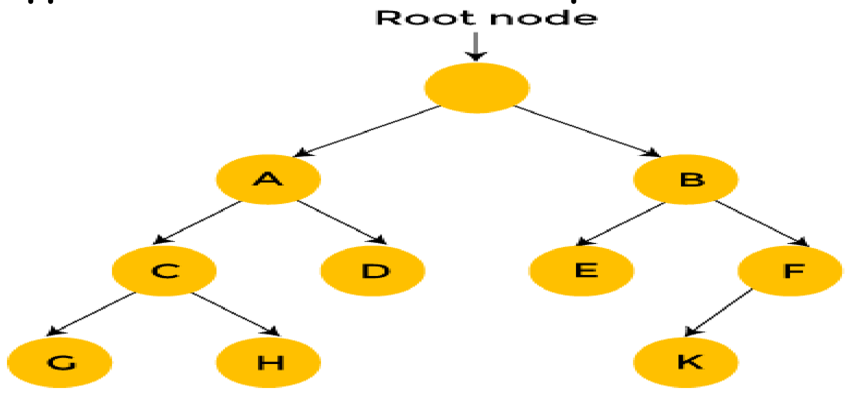
Such an algorithm can be of two types:

- **Optimizing:** In this case, the best solution is found. To find the best solution, it may either find all the possible solutions to find the best solution or if the value of the best solution is known, it stops finding when the best solution is found. For example: Finding the best path for the travelling salesman problem. Here best path means that travelling all the cities and the cost of travelling should be minimum.

- **Satisficing:** It stops finding the solution as soon as the satisfactory solution is found. Or example, finding the travelling salesman path which is within 10% of optimal.
- Often Brute force algorithms require exponential time. Various heuristics and optimization can be used:
- **Heuristic:** A rule of thumb that helps you to decide which possibilities we should look at first.
- **Optimization:** A certain possibilities are eliminated without exploring all of them.

Let's understand the brute force search through an example.

Suppose we have converted the problem in the form of the tree shown as below:



Brute force search considers each and every state of a tree, and the state is represented in the form of a node. As far as the starting position is concerned, we have two choices, i.e., A state and B state. We can either generate state A or state B. In the case of B state, we have two states, i.e., state E and F.

In the case of brute force search, each state is considered one by one. As we can observe in the above tree that the brute force search takes 12 steps to find the solution.

On the other hand, backtracking, which uses Depth-First search, considers the below states only when the state provides a feasible solution. Consider the above tree, start from the root node, then move to node A and then node C. If node C does not provide the feasible solution, then there is no point in considering the states G and H. We backtrack from node C to node A. Then, we move from node A to node D. Since node D does not provide the feasible solution, we discard this state and backtrack from node D to node A.

We move to node B, then we move from node B to node E. We move from node E to node K; Since k is a solution, so it takes 10 steps to find the solution. In this way, we eliminate a greater number of states in a single iteration. Therefore, we can say that backtracking is faster and more efficient than the brute force approach.

Advantages of a brute-force algorithm

The following are the advantages of the brute-force algorithm:

- This algorithm finds all the possible solutions, and it also guarantees that it finds the correct solution to a problem.
- This type of algorithm is applicable to a wide range of domains.
- It is mainly used for solving simpler and small problems.
- It can be considered a comparison benchmark to solve a simple problem and does not require any particular domain knowledge.

Disadvantages of a brute-force algorithm

The following are the disadvantages of the brute-force algorithm:

- It is an inefficient algorithm as it requires solving each and every state.
- It is a very slow algorithm to find the correct solution as it solves each state without considering whether the solution is feasible or not.
- The brute force algorithm is neither constructive nor creative as compared to other algorithms.

The Knuth-Morris-Pratt (KMP) Algorithm

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP Algorithm:

1. The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

The Prefix Function (Π)

Following pseudo code compute the prefix function, Π:

COMPUTE- PREFIX- FUNCTION (P)

```
1. m ←length [P]              //'p' pattern to be matched
2. Π [1] ← 0
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P [k + 1] ≠ P [q]
6. do k ← Π [k]
```

```
7. If P [k + 1] = P [q]
8. then k← k + 1
9. Π [q] ← k
10. Return Π
```

Running Time Analysis:

In the above pseudo code for calculating the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step1 to Step3 take constant time. Hence the running time of computing prefix function is O (m).

Example: Compute Π for the pattern 'p' below:

P :

a	b	a	b	a	c	a

Solution:

Initially: m = length [p] = 7

Π [1] = 0

k = 0

Step 1: q = 2, k = 0

Π [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

Step 2: q = 3, k = 0

Π [3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

Step3: q =4, k =1

Π [4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

Step4: q = 5, k =2

Π [5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: q = 6, k = 3

Π [6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: q = 7, k = 1

Π [7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

The KMP Matcher:

The KMP Matcher with the pattern 'p,' the string 'S' and prefix function 'Π' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

KMP-MATCHER (T, P)

```
1. n ← length [T]
2. m ← length [P]
3. Π← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0                  // numbers of characters matched
5. for i ← 1 to n         // scan S from left to right
6. do while q > 0 and P [q + 1] ≠ T [i]
7. do q ← Π [q]                 // next character does not match
```

```
8. If P [q + 1] = T [i]
9. then q ← q + 1              // next character matches
10. If q = m                   // is all of p matched?
11. then print "Pattern occurs with shift" i - m
12. q ← Π [q]                  // look for the next match
```

Running Time Analysis:

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is O (n).

Example: Given a string 'T' and pattern 'P' as follows:

T:															
	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P:							
	a	b	a	b	a	c	a

Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, ? was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

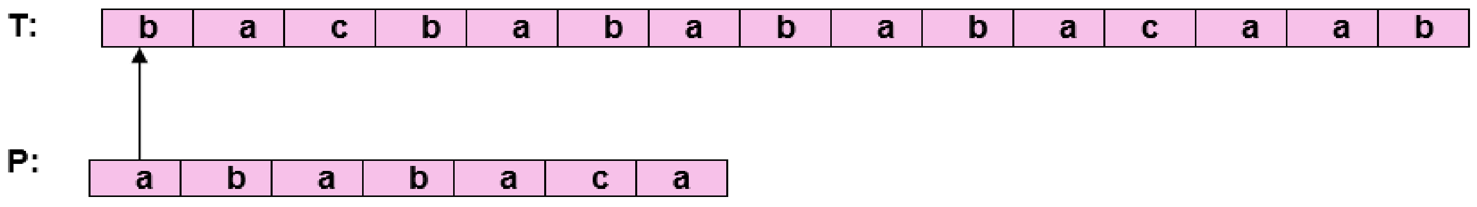
Solution:

Initially: n = size of T = 15

m = size of P = 7

Step1: i=1, q=0

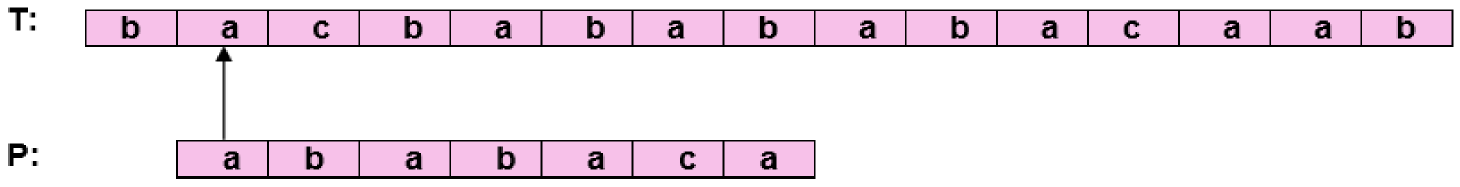
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

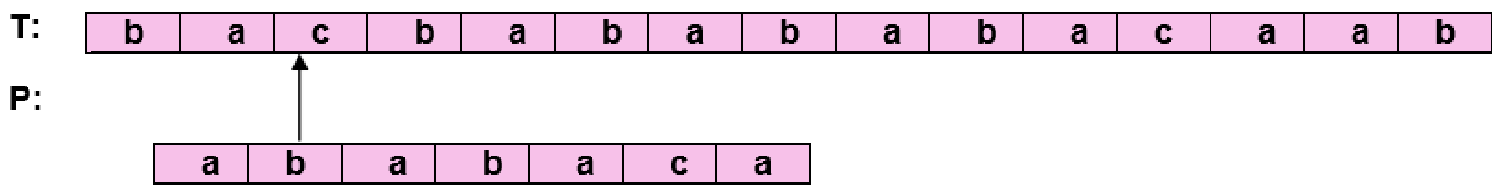
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

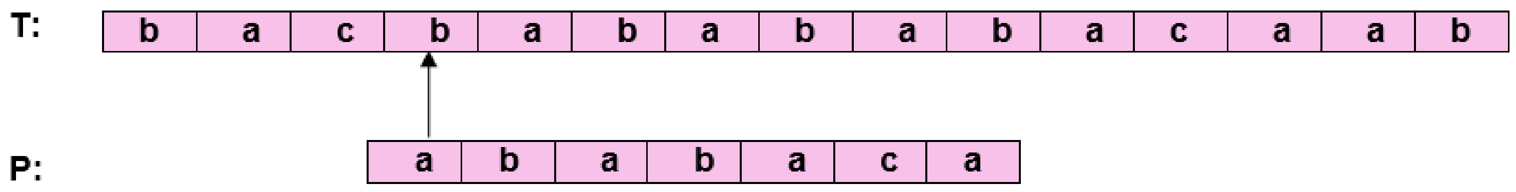
Comparing P [2] with T [3]   P [2] doesn't match with T [3]



Backtracking on p, Comparing P [1] and T [3]

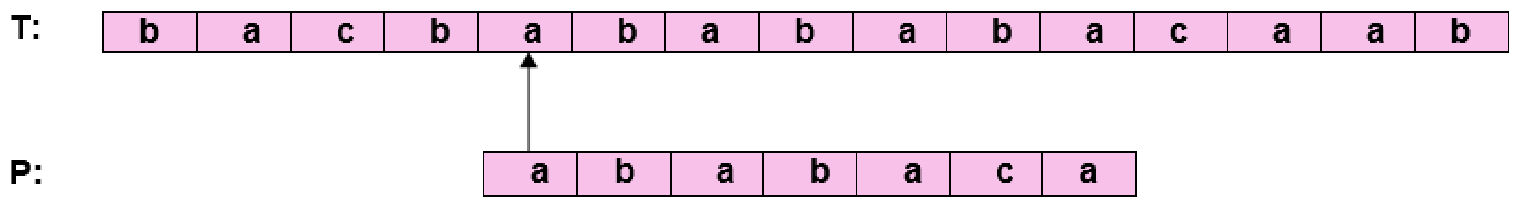
Step4: i = 4, q = 0

Comparing P [1] with T [4]   P [1] doesn't match with T [4]



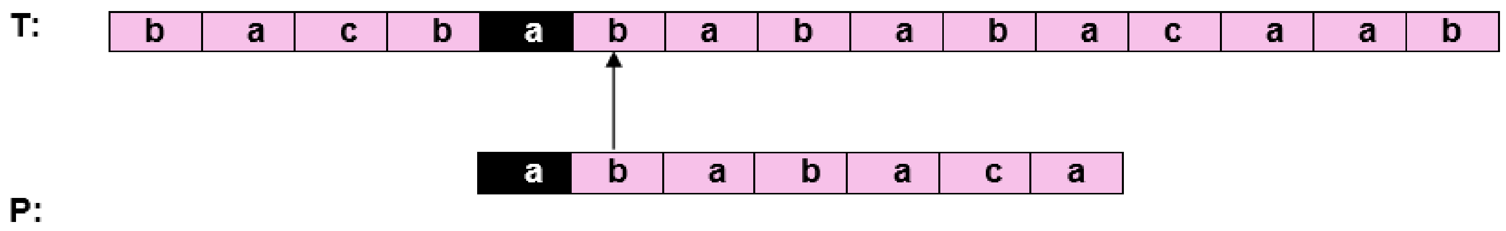
Step5: i = 5, q = 0

Comparing P [1] with T [5]   P [1] match with T [5]



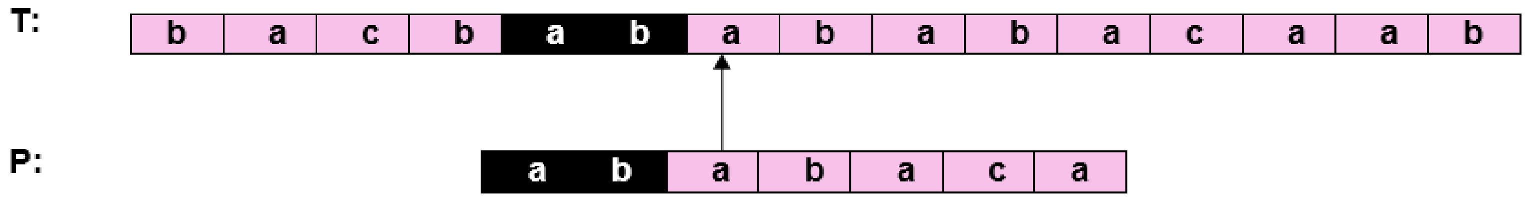
Step6: i = 6, q = 1

Comparing P [2] with T [6]   P [2] matches with T [6]



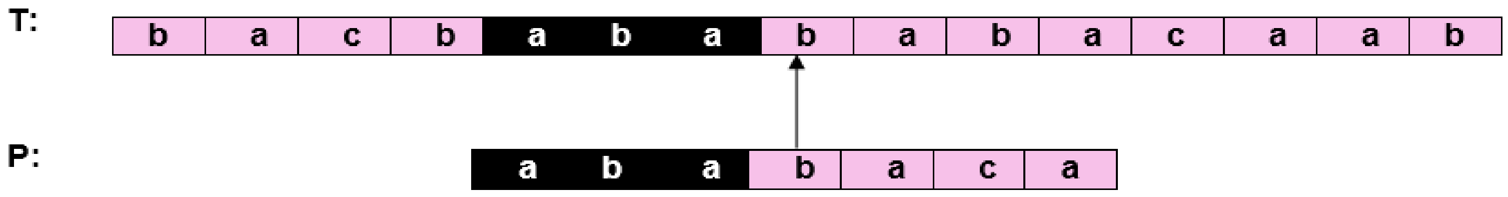
Step7: i = 7, q = 2

Comparing P [3] with T [7]   P [3] matches with T [7]



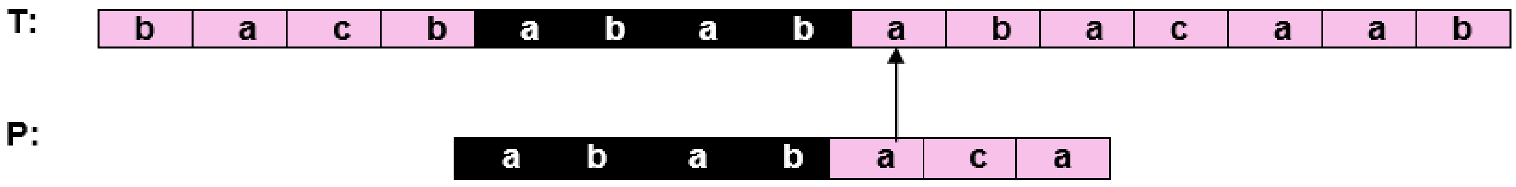
Step8: i = 8, q =3

Comparing P [4] with T [8]   P [4] matches with T [8]



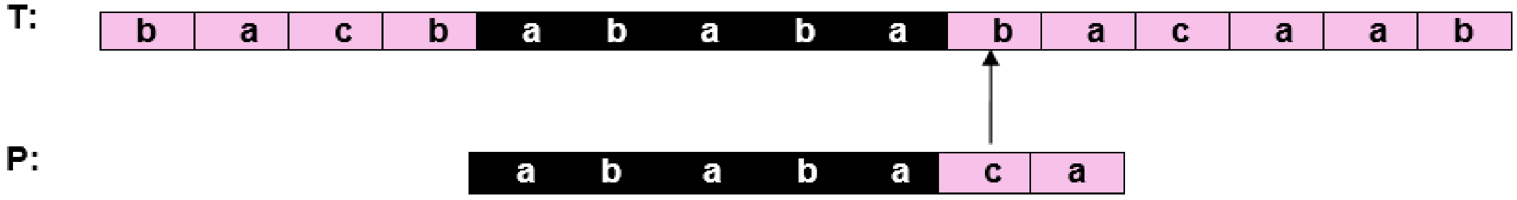
Step9: i = 9, q = 4

Comparing P [5] with T [9]   P [5] matches with T [9]



Step10: i = 10, q = 5

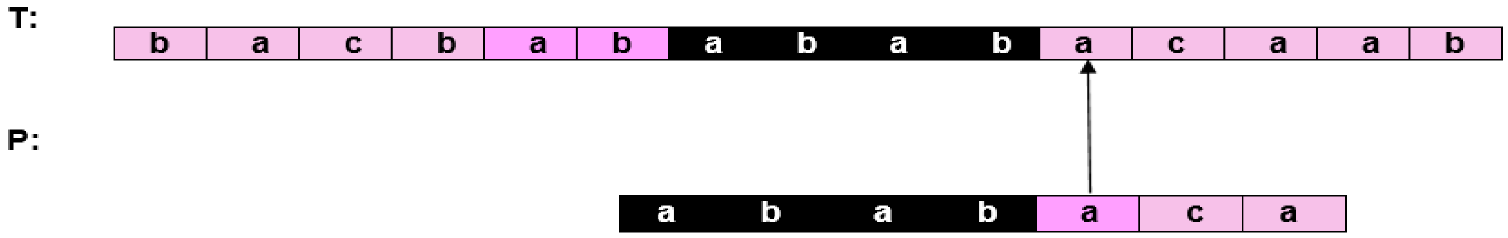
Comparing P [6] with T [10]   P [6] doesn’t match with T [10]



Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

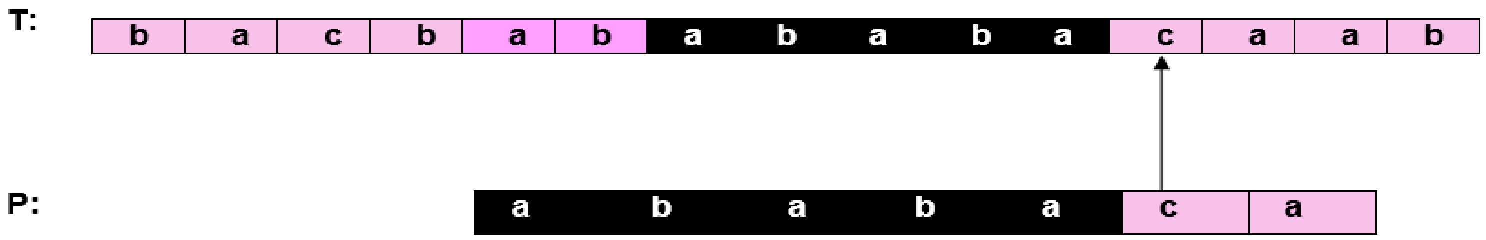
Step11: i = 11, q =4

Comparing P [5] with T [11]   P [5] match with T [11]

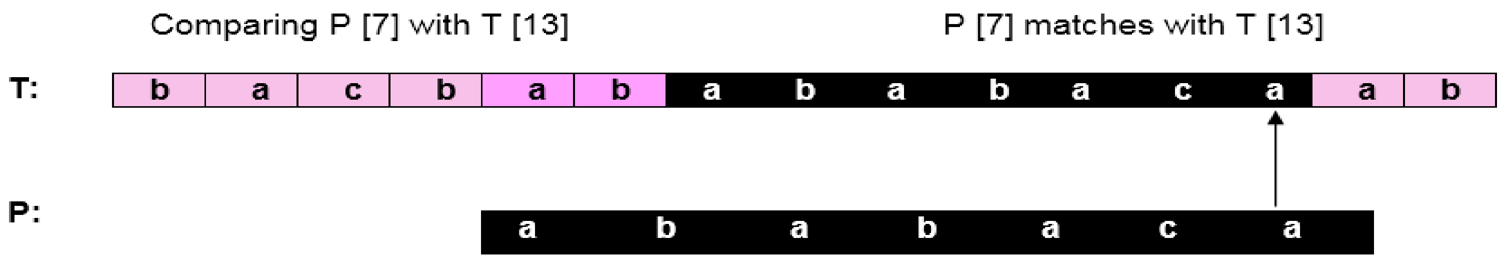


Step12: i = 12, q = 5

Comparing P [6] with T [12]   P [6] matches with T [12]



Step13: i = 3, q = 6


Comparing P [7] with T [13]
P [7] matches with T [13]
T:
b a c b a b a b a b a c a a b
P:
a b a b a c a

Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

Boyer-Moore strategies

Robert Boyer and J Strother Moore established it in 1977. The B-M String search algorithm is a particularly efficient algorithm and has served as a standard benchmark for string search algorithm ever since.

The B-M algorithm takes a 'backward' approach: the pattern string (P) is aligned with the start of the text string (T), and then compares the characters of a pattern from right to left, beginning with rightmost character.

If a character is compared that is not within the pattern, no match can be found by analyzing any further aspects at this position so the pattern can be changed entirely past the mismatching character.

For deciding the possible shifts, B-M algorithm uses two preprocessing strategies simultaneously. Whenever a mismatch occurs, the algorithm calculates a variation using both approaches and selects the more significant shift thus, if make use of the most effective strategy for each case.

The two strategies are called heuristics of B - M as they are used to reduce the search. They are:

1. Bad Character Heuristics
2. Good Suffix Heuristics

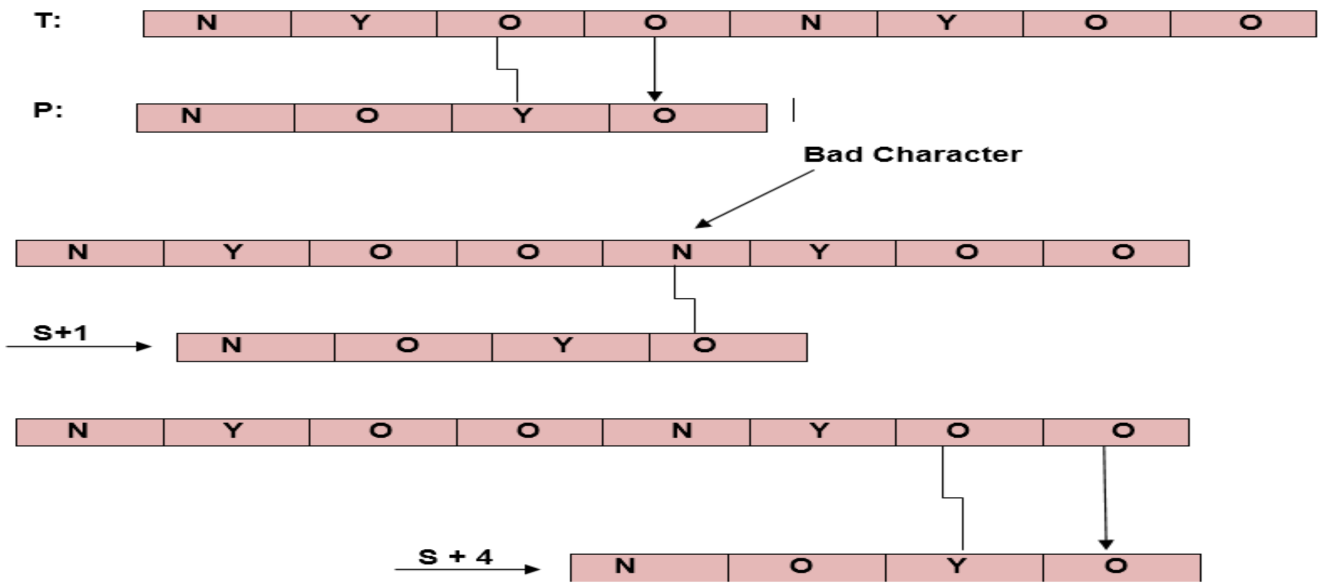
1. Bad Character Heuristics

This Heuristics has two implications:

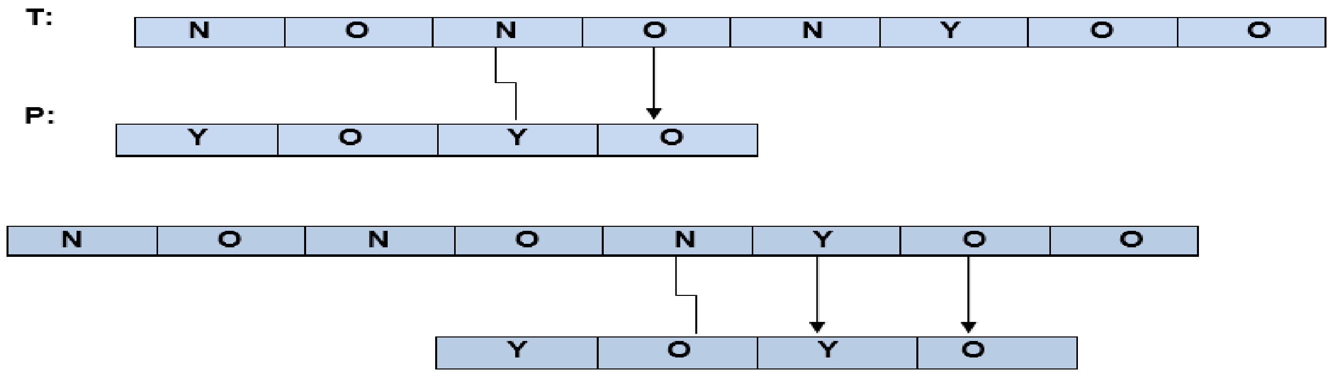
- Suppose there is a character in a text in which does not occur in a pattern at all. When a mismatch happens at this character (called as bad character), the whole pattern can be changed, begin matching form substring next to this 'bad character.'
- On the other hand, it might be that a bad character is present in the pattern, in this case, align the nature of the pattern with a bad character in the text.

Thus in any case shift may be higher than one.

Example1: Let Text T = <nyoo nyoo> and pattern P = <noyo>



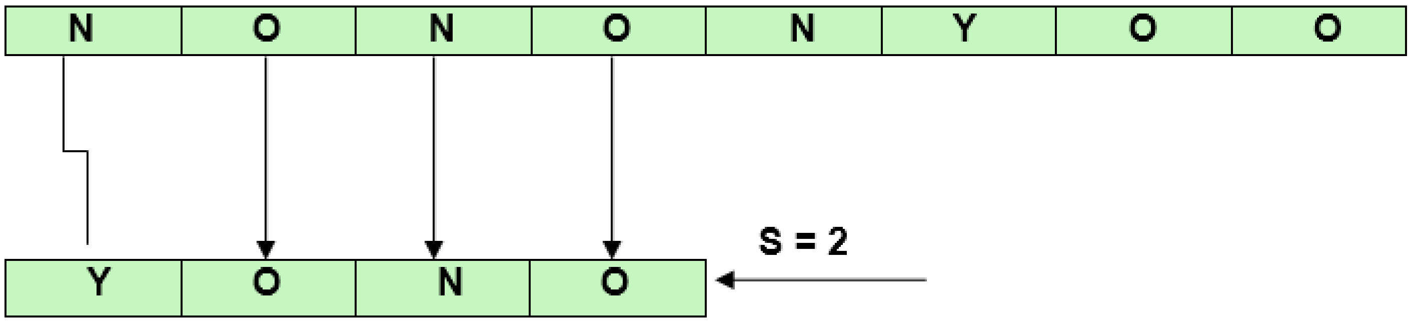
Example2: If a bad character doesn't exist the pattern then.



Problem in Bad-Character Heuristics:

In some cases, Bad-Character Heuristics produces some negative shifts.

For Example:



This means that we need some extra information to produce a shift on encountering a bad character. This information is about the last position of every aspect in the pattern and also the set of characters used in a pattern (often called the alphabet ∑of a pattern).

COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑)

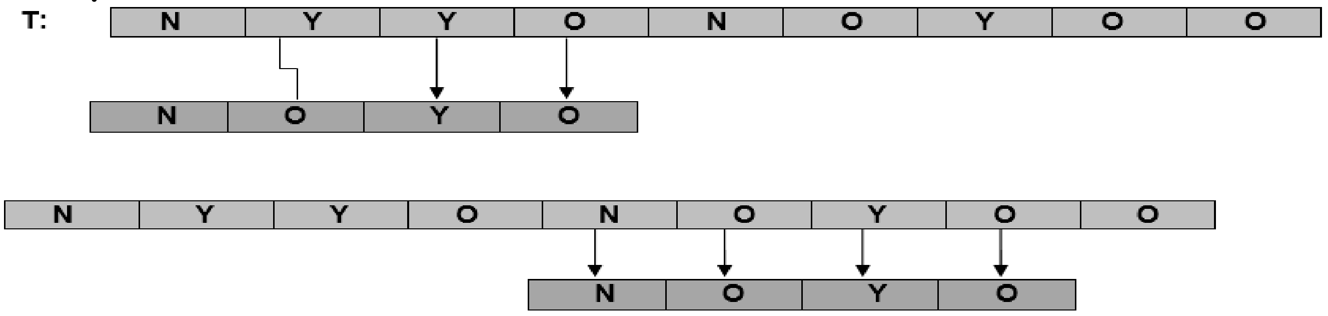
```
1. for each character a ∈ ∑
2. do ʎ [a] = 0
3. for j ← 1 to m
```

```
4. do ʌ [P [j]] ← j
5. Return ʌ
```

2. Good Suffix Heuristics:

A good suffix is a suffix that has matched successfully. After a mismatch which has a negative shift in bad character heuristics, look if a substring of pattern matched till bad character has a good suffix in it, if it is so then we have an onward jump equal to the length of suffix found.

Example:



COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)

```
1. Π ← COMPUTE-PREFIX-FUNCTION (P)
2. P'← reverse (P)
3. Π'← COMPUTE-PREFIX-FUNCTION (P')
4. for j ← 0 to m
5. do ɣ [j] ← m - Π [m]
6. for l ← 1 to m
7. do j ← m - Π' [L]
8. If ɣ [j] > l - Π' [L]
9. then ɣ [j] ← 1 - Π'[L]
10. Return ɣ
```

BOYER-MOORE-MATCHER (T, P, Σ)

```
1. n ←length [T]
2. m ←length [P]
3. ʌ← COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, Σ )
4. ɣ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
5. s ←0
6. While s ≤ n - m
7. do j ← m
8. While j > 0 and P [j] = T [s + j]
9. do j ←j-1
10. If j = 0
11. then print "Pattern occurs at shift" s
```

```
12. s ← s + γ[0]
13. else s ← s + max (γ [j], j - λ[T[s+j]])
```

Complexity Comparison of String Matching Algorithm:

Algorithm	**Preprocessing Time**	**Matching Time**
Naive	O	(O (n - m + 1)m)
Rabin-Karp	O(m)	(O (n - m + 1)m)
Finite Automata	O(m|∑|)	O (n)
Knuth-Morris-Pratt	O(m)	O (n)
Boyer-Moore	O(|∑|)	(O ((n - m + 1) + |∑|))

The Huffman Coding Algorithm

- (i) Data can be encoded efficiently using Huffman Codes.
- (ii) It is a widely used and beneficial technique for compressing data.
- (iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

Suppose we have 10^5 characters in a data file. Normal Storage: 8 bits per character (ASCII) - 8×10^5 bits in a file. But we want to compress the file and save it compactly. Suppose only six characters appear in the file:

	a	**b**	**c**	**d**	**e**	**f**	**Total**
Frequency	45	13	12	16	9	5	100

How can we represent the data in a Compact way?

(i) Fixed length Code: Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character:

For example:

a	000
b	001
c	010
d	011
e	100
f	101

For a file with 10^5 characters, we need 3×10^5 bits.

(ii) A variable-length code: It can do considerably better than a fixed-length code, by giving many characters short code words and infrequent character long codewords.

For example:

a	0
b	101
c	100
d	111

e	1101
f	1100

Number of bits = (45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4) x 1000

= 2.24 x 10^5bits

Thus, 224,000 bits to represent the file, a saving of approximately 25%.This is an optimal character code for this file.

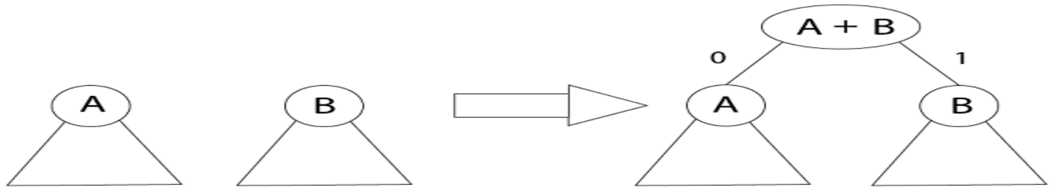
Prefix Codes:

The prefixes of an encoding of one character must not be equal to complete encoding of another character, e.g., 1100 and 11001 are not valid codes because 1100 is a prefix of some other code word is called prefix codes.

Prefix codes are desirable because they clarify encoding and decoding. Encoding is always simple for any binary character code; we concatenate the code words describing each character of the file. Decoding is also quite comfortable with a prefix code. Since no codeword is a prefix of any other, the codeword that starts with an encoded data is unambiguous.

Greedy Algorithm for constructing a Huffman Code:

Huffman invented a greedy algorithm that creates an optimal prefix code called a Huffman Code.



The algorithm builds the tree T analogous to the optimal code in a bottom-up manner. It starts with a set of |C| leaves (C is the number of characters) and performs |C| - 1 'merging' operations to create the final tree. In the Huffman algorithm 'n' denotes the quantity of a set of characters, z indicates the parent node, and x & y are the left & right child of z respectively.

Algorithm of Huffman Code

Huffman (C)

```
1. n=|C|
2. Q ← C
3. for i=1 to n-1
4. do
5. z= allocate-Node ()
6. x=  left[z]=Extract-Min(Q)
7. y= right[z] =Extract-Min(Q)
8. f [z]=f[x]+f[y]
```

9. Insert (Q, z)

10. return Extract-Min (Q)

Example: Find an optimal Huffman Code for the following set of frequencies:

1. a: 50 b: 25 c: 15 d: 40 e: 75

Solution:

Given that: C = {a, b, c, d, e}

f (C) = {50, 25, 15, 40, 75}

n = 5

Q $\leftarrow$ c

i.e.

c	15	b	25	d	40	a	50	e	75

for i $\leftarrow$ 1 to 4

i = 1 Z $\leftarrow$ Allocate node

x $\leftarrow$ Extract-Min (Q)

y $\leftarrow$ Extract-Min (Q)

c	15	b	25	d	40	a	50	e	75

Left [z] $\leftarrow$ x

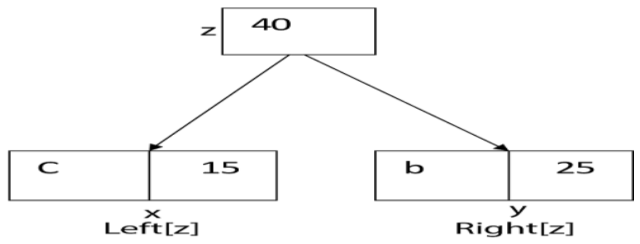
Right [z] $\leftarrow$ y

f (z) $\leftarrow$ f (x) + f (y) = 15 + 25

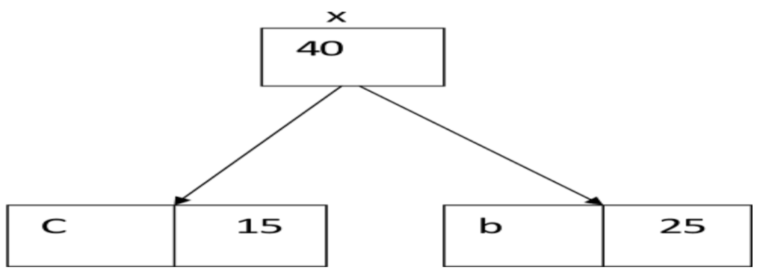
f (z) = 40

d	40	a	50	e	75

i.e.



Again for i=2



d	40	a	50	e	75

z $\leftarrow$ Allocate node

x $\leftarrow$ 40

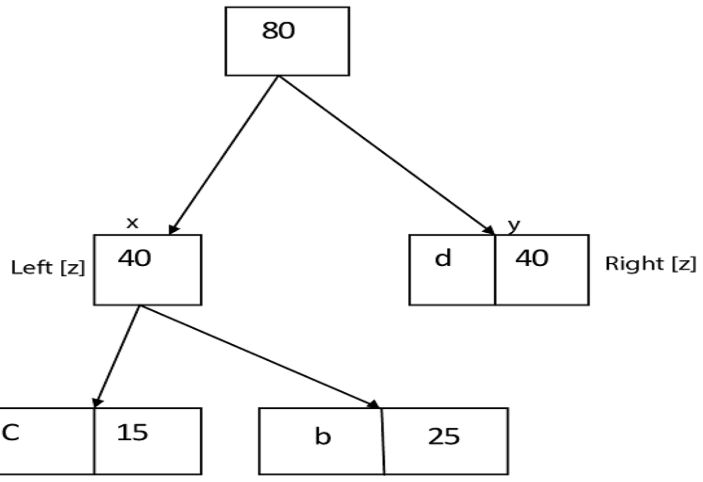
y $\leftarrow$ 40

left [z] $\leftarrow$ x

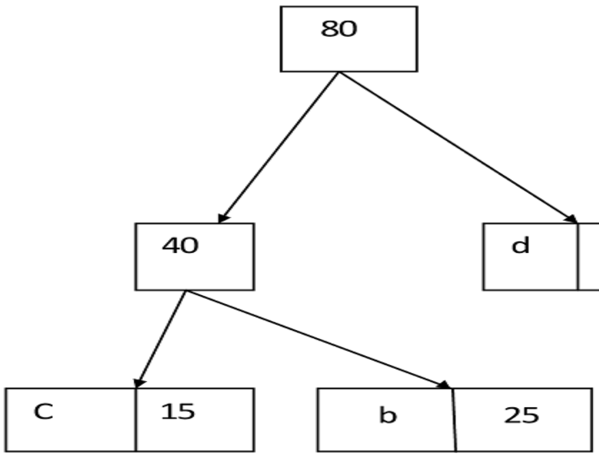
right [z] $\leftarrow$ y

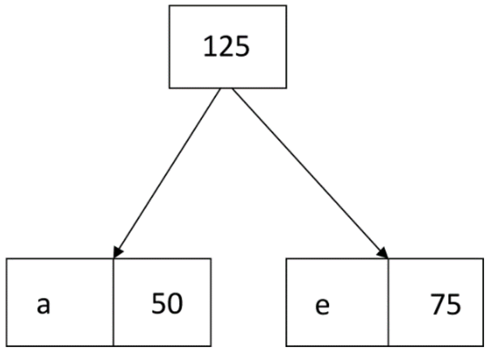
f (z) = 40 + 40 = 80

a	50	e	75

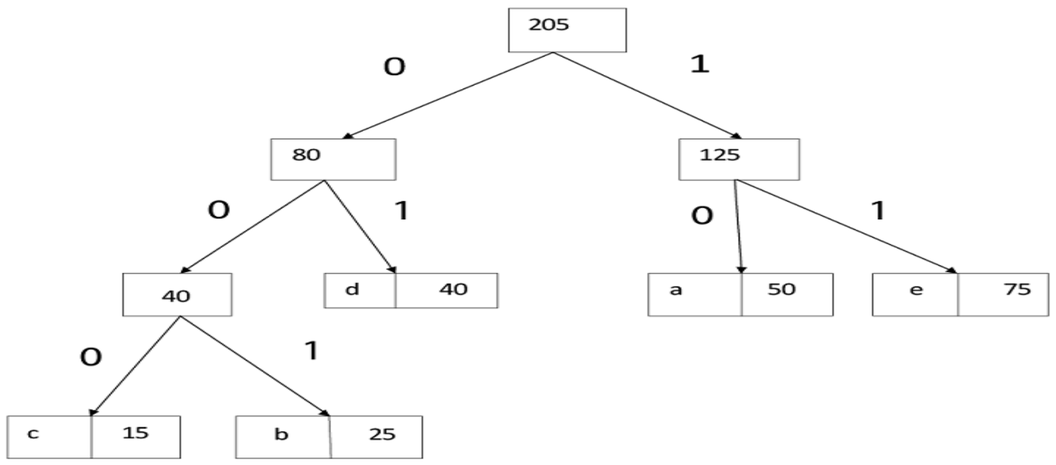


Similarly, we apply the same process we get





Thus, the final output is:



Longest Common Subsequence Problem (LCS)

Here longest means that the subsequence should be the biggest one. The common means that some of the characters are common between the two strings. The subsequence means that some of the characters are taken from the string that is written in increasing order to form a subsequence.

Let's understand the subsequence through an example.

Suppose we have a string 'w'.

W_1 = abcd

The following are the subsequences that can be created from the above string:

- ab
- bd
- ac
- ad
- acd
- bcd

The above are the subsequences as all the characters in a sub-string are written in increasing order with respect to their position. If we write ca or da then it would be a wrong subsequence as characters are not appearing in the increasing order. The total number of subsequences that would be possible is 2^n, where n is the number of characters in a string. In the above string, the value of 'n' is 4 so the total number of subsequences would be 16.

W_2= bcd

By simply looking at both the strings w1 and w2, we can say that bcd is the longest common subsequence. If the strings are long, then it won't be possible to find the subsequence of both the string and compare them to find the longest common subsequence.

Finding LCS using dynamic programming with the help of a table.

Consider two strings:

X= a b a a b a

Y= b a b b a b

(a, b)

For index i=1, j=1

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	← 0					
b	0						
a	0						
a	0						
b	0						
a	0						

Since both the characters are different so we consider the maximum value. Both contain the same value, i.e., 0 so put 0 in (a,b). Suppose we are taking the 0 value from 'X' string, so we put arrow towards 'a' as shown in the above table.

(a, a)

For index i=1, j=2

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	← 0	↖ 1				
b	0						
a	0						
a	0						
b	0						
a	0						

Both the characters are the same, so the value would be calculated by adding 1 and upper diagonal value. Here, upper diagonal value is 0, so the value of this entry would be (1+0) equal to 1. Here, we are considering the upper diagonal value, so the arrow will point diagonally.

(a, b)

For index i=1, j=3

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	← 0	↖ 1	← 1			
b	0						
a	0						
a	0						
b	0						
a	0						

Since both the characters are different so we consider the maximum value. The character 'a' has the maximum value, i.e., 1. The new entry, i.e., (a, b) will contain the value 1 pointing to the 1 value.

(a, b)

For index i=1, j=4

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	← 0	↖ 1	← 1	← 1		
b	0						
a	0						
a	0						
b	0						
a	0						

Since both the characters are different so we consider the maximum value. The character 'a' has the maximum value, i.e., 1. The new entry, i.e., (a, b) will contain the value 1 pointing to the 1 value.

(a, a)

For index i=1, j=5

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	←0	↖1	←1	←1	←1	
b	0						
a	0						
a	0						
b	0						
a	0						

Both the characters are same so the value would be calculated by adding 1 and upper diagonal value. Here, upper diagonal value is 0 so the value of this entry would be (1+0) equal to 1. Here, we are considering the upper diagonal value so arrow will point diagonally.

(a, b)

For index i=1, j=6

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	←0	↖1	←1	←1	←1	←1
b	0						
a	0						
a	0						
b	0						
a	0						

Since both the characters are different so we consider the maximum value. The character 'a' has the maximum value, i.e., 1. The new entry, i.e., (a, b) will contain the value 1 pointing to the 1 value.

(b, b)

For index i=2, j=1

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	←0	↖1	←1	←1	↖1	←1
b	0	↖1					
a	0						
a	0						
b	0						
a	0						

Both the characters are same so the value would be calculated by adding 1 and upper diagonal value. Here, upper diagonal value is 0 so the value of this entry would be (1+0) equal to 1. Here, we are considering the upper diagonal value so arrow will point diagonally.

(b, a)

For index i=2, j=2

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
		←	↖				
a	0	0	1	←1	←1	↖1	←1
b	0	↖1	←1				
a	0						
a	0						
b	0						
a	0						

Since both the characters are different so we consider the maximum value. The character 'a' has the maximum value, i.e., 1. The new entry, i.e., (a, b) will contain the value 1 pointing to the 1 value.
In this way, we will find the complete table. The final table would be:

	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	←0	↖1	←1	←1	↖1	←1
b	0	↖1	←1	↖2	↖2	←2	↖2
a	0	↑1	↖2	←2	←2	↖3	←3
a	0	↑1	↖2	←2	←2	↖3	←3
b	0	↖1	←1	↖3	↖3	←3	↖4
a	0	↑1	↖2	←2	←2	↖4	←4

In the above table, we can observe that all the entries are filled. Now we are at the last cell having 4 value. This cell moves at the left which contains 4 value.; therefore, the first character of the LCS is 'a'.
The left cell moves upwards diagonally whose value is 3; therefore, the next character is 'b' and it becomes 'ba'. Now the cell has 2 value that moves on the left. The next cell also has 2 value which is moving upwards; therefore, the next character is 'a' and it becomes 'aba'.
The next cell is having a value 1 that moves upwards. Now we reach the cell (b, b) having value which is moving diagonally upwards; therefore, the next character is 'b'. The final string of longest common subsequence is 'baba'.
Why a dynamic programming approach in solving a LCS problem is more efficient than the recursive algorithm?
If we use the dynamic programming approach, then the number of function calls are reduced. The dynamic programming approach stores the result of each function call so that the result of function calls can be used in the future function calls without the need of calling the functions again.
In the above dynamic algorithm, the results obtained from the comparison between the elements of x and the elements of y are stored in the table so that the results can be stored for the future computations.

The time taken by the dynamic programming approach to complete a table is O(mn) and the time taken by the recursive algorithm is $2^{max(m, n)}$.

Algorithm of Longest Common Subsequence

```
1.  Suppose X and Y are the two given sequences
2.  Initialize a table of LCS having a dimension of X.length * Y.length
3.  XX.label = X
4.  YY.label = Y
5.  LCS[0][] = 0
6.  LCS[][0] = 0
7.  Loop starts from the LCS[1][1]
8.  Now we will compare X[i] and Y[j]
9.     if X[i] is equal to Y[j] then
10.                         LCS[i][j] = 1 + LCS[i-1][j-1]
11.      Point an arrow LCS[i][j]
12.                    Else
13.                       LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
```