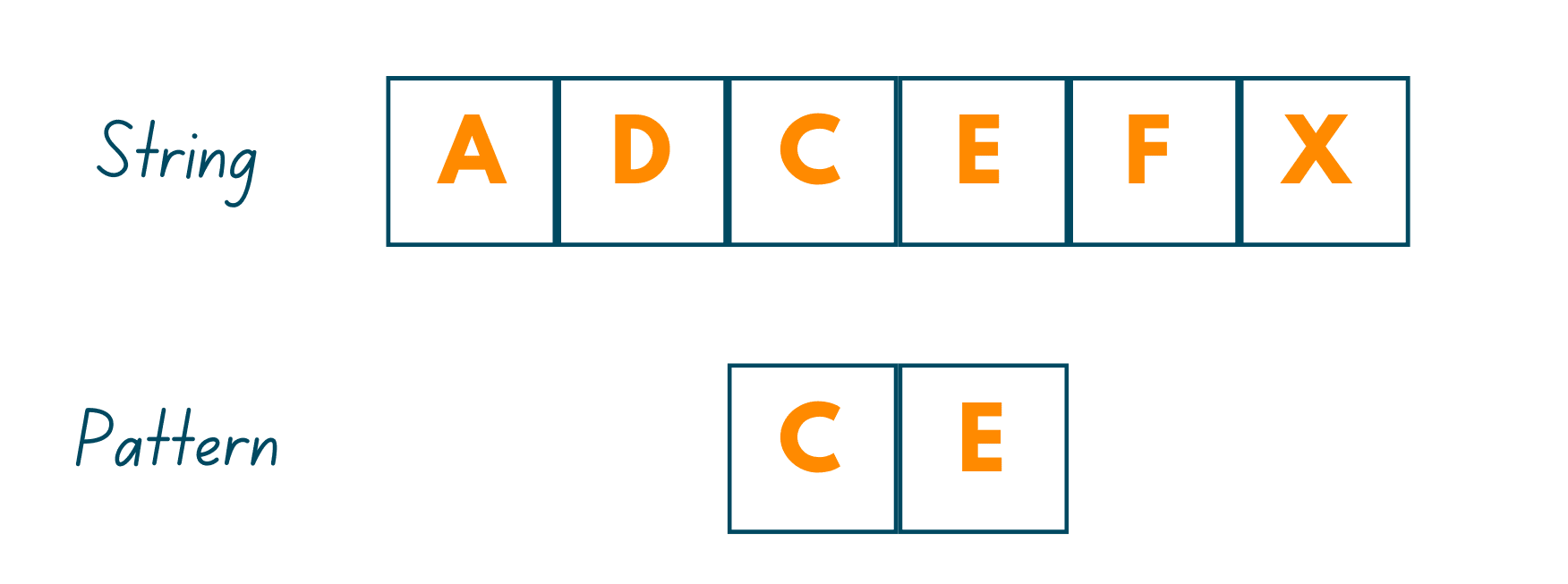
**Boyer Moore Algorithm (Strategy: Heuristic)**

**Boyer-Moore Algorithm**

In the year 1977, Robert Boyer and J Strother Moore established an algorithm that proved to be very effective for the various pattern-searching problems in the computing world. They named it the Boyer-Moore algorithm which has served as the benchmark for pattern-searching algorithms ever since.

Unlike the traditional way of pattern searching where we try to match the two strings in a forward manner, the Boyer-Moore advances the concept by beginning to match the last character of the string to be searched. In this way, the time complexity is reduced significantly.



As mentioned earlier, this algorithm takes the backward approach by aligning the pattern string P with text string T and thereafter, comparing the characters from right to left, starting with the rightmost character. It works on the principle that if a mismatched character is found, then there is no use in matching the other characters in the two strings.

To work effectively, there are two strategies/approaches we use in this algorithm:

1. Bad Character Heuristics
2. Good Suffix Heuristics

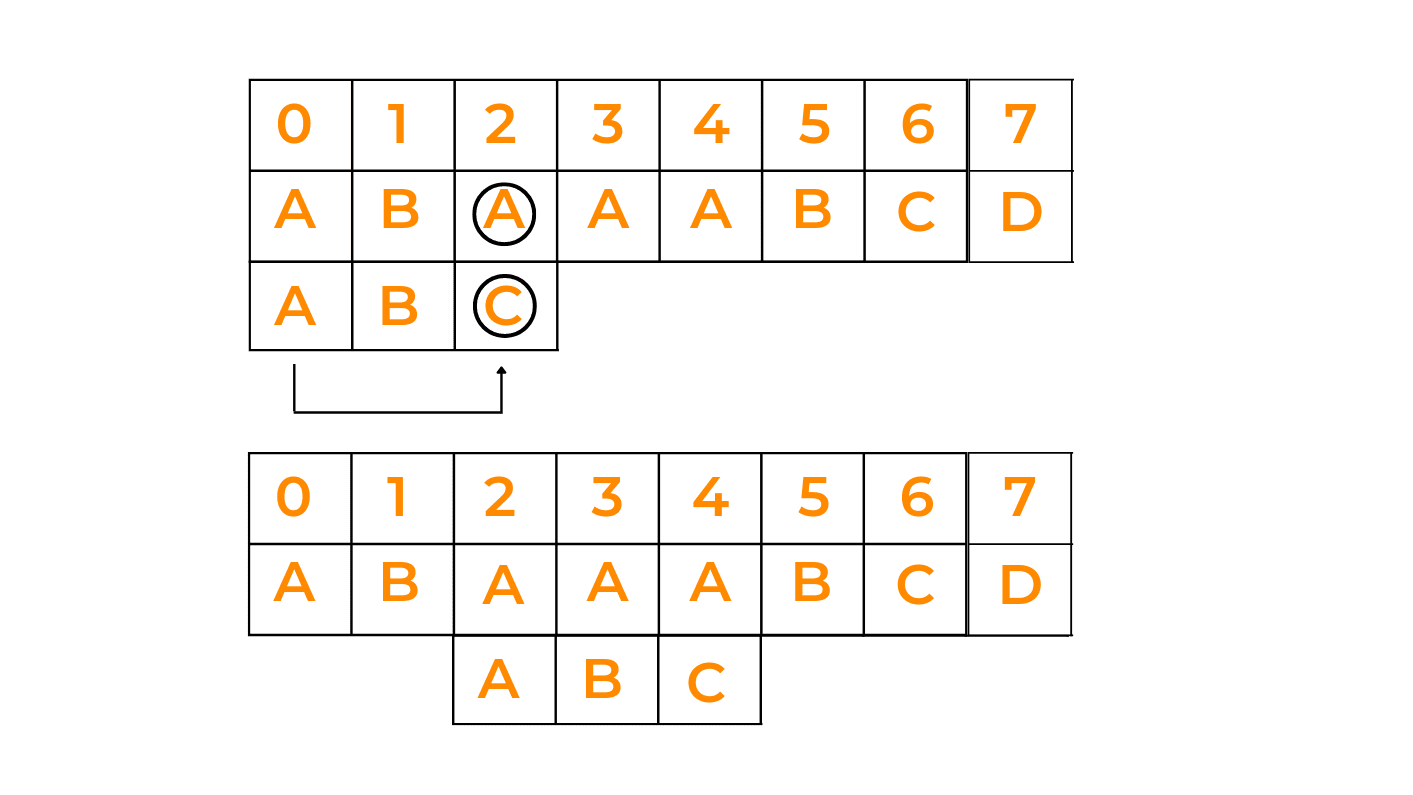
In the following, we have explained in detail both the strategies along with their implementation in C++, Java, and Python.

01) Bad Character Heuristics

This intuition behind solving the pattern-matching problem is quite simple. Just as its name suggests, here, the first character that we encounter which does not match the pattern we are searching for, is called the Bad Character. On encountering the bad character, we:

Case I: Shift the pattern till the mismatch becomes a match

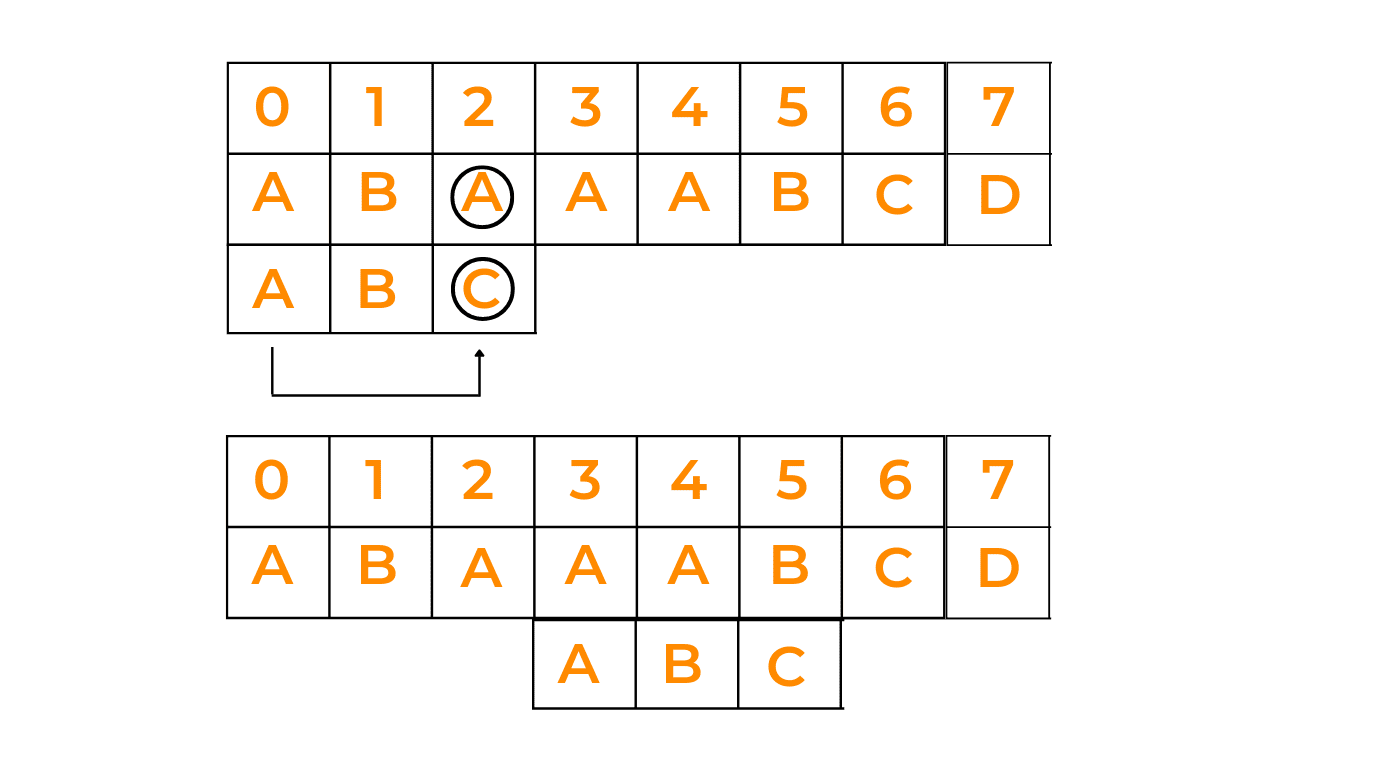
As mentioned earlier, we start checking the pattern occurrence from the rightmost character, so in this case, we'll find the bad character. Now, we'll shift the pattern in such a way that it gets aligned with the mismatched character in the text. For more clarity, you can see the illustration:



Explanation:  As we have already stated that in this algorithm, the matching begins from the rightmost character. So, we start matching the text with the pattern. On matching the first character of the pattern we observe that 'C' does not match with 'A'. So, we try to match the 'A' of the given text with 'A'(if present) in the pattern. So, we shift the pattern by 2 so that the rightmost character of both strings matches. Thereafter, we match the other 2 characters.

Case II: Shift the pattern till it moves past the bad character

In this case, we'll begin matching the pattern from the right end, and see if it matches one by one with all the characters of the given string. If at any time, we observe that the rightmost character is not matching with the rightmost character of the given string, we shift the whole pattern ahead of the last character of the string with which we were trying to match the pattern. Have a look at the below image:



Explanation: Now, in this case, observe that we have a mismatch at character 'C'. Now, after going through the whole pattern, we observe that it does not contain any 'C'. So, we take the whole pattern and move it right across the letter 'C' and later, check for it to match.

**class** Solution{

static int NO\_OF\_CHARS = 256;

static int max (int a, int b)

{

**return** (a > b)? a: b;

} //returns max of two numbers

static void badCharHeuristic( char []str, int size,int badchar[])

{

// Initialize all occurrences **as** -1

**for** (int i = 0; i < NO\_OF\_CHARS; i++)

badchar[i] = -1;

// Fill the actual value of last occurrence of a character

**for** (int i = 0; i < size; i++)

badchar[(int) str[i]] = i;

}

static void search( char txt[], char pat[])

{

int m = pat.length;

int n = txt.length;

int badchar[] = new int[NO\_OF\_CHARS];

badCharHeuristic(pat, m, badchar);

int s = 0;

**while**(s <= (n - m))

{

int j = m-1;

/\* Keep reducing index j of pattern **while**

characters of pattern **and** text are

matching at this shift s \*/

**while**(j >= 0 && pat[j] == txt[s+j])

j--;

/\* If the pattern **is** present at current

shift, then index j will become -1 after

the above loop \*/

**if** (j < 0)

{

System.out.println("Patterns occur at shift = " + s);

s += (s+m < n)? m-badchar[txt[s+m]] : 1;

}

**else**

s += max(1, j - badchar[txt[s+j]]);

}

}

public static void main(String []args) {

char txt[] = "ABAAABCD".toCharArray();

char pat[] = "ABC".toCharArray();

search(txt, pat);

}

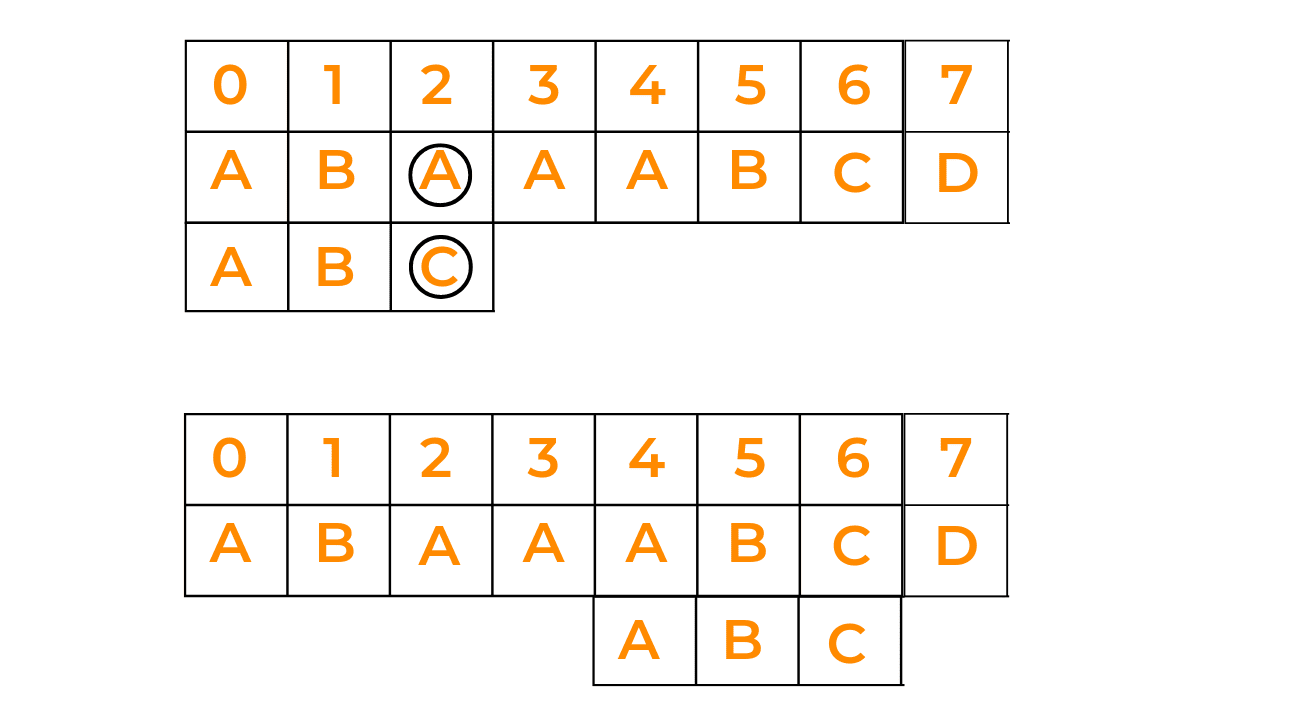
}

## **02) Good Suffix Heuristics**

Just like in the Bad Character approach preprocessing is done, in this strategy also, preprocessing is done based on certain scenarios. Here, we have 3 different cases that will tell which step we need to follow next to match the given pattern efficiently:

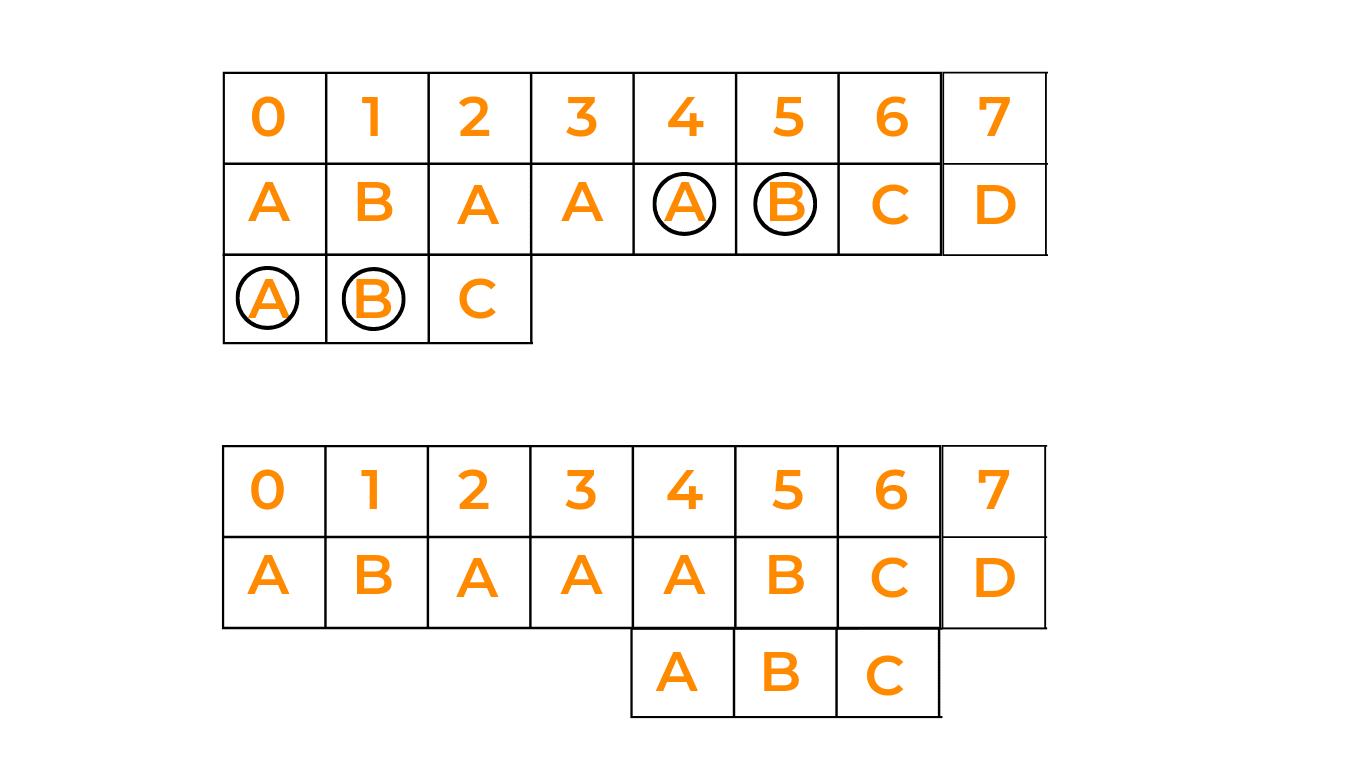
**Case I: Shift the pattern until the next occurrence is found**

Here, when we see that pattern is not matching with the given string, we shift the pattern to the right side to check whether that pattern is present in the further characters of the string. See the given image:



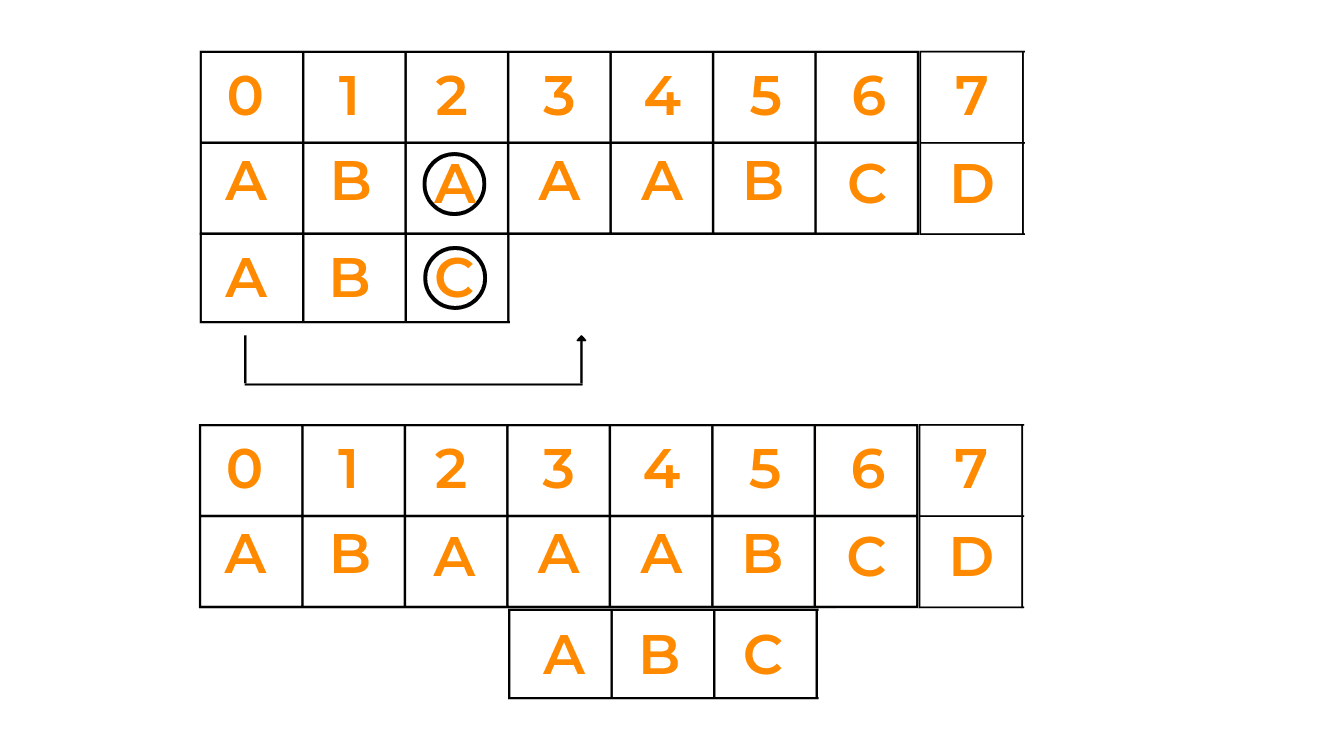
**Case II: Shift pattern till prefix of pattern matches with a suffix of text**

In this case, if we're not able to find the pattern in one go, we try to match the prefix of the pattern with a possible suffix of the given text, by shifting the pattern that many times as the length of the prefix. Have a look at the given image:



**Case III: Shift pattern till the pattern moves past the provided text**

This happens when there is no possibility of the pattern matching with the string. In this case, we just shift the pattern till the end. Look at the given illustration:



*/\* Java program for Boyer Moore Algorithm with*

*Good Suffix heuristic to find pattern in*

*given text string \*/*

**class** GFG

{

*// preprocessing for strong good suffix rule*

**static** **void** preprocess\_strong\_suffix(**int** []shift, **int** []bpos,

**char** []pat, **int** m)

{

*// m is the length of pattern*

**int** i = m, j = m + 1;

bpos[i] = j;

**while**(i > 0)

{

*/\*if character at position i-1 is not*

*equivalent to character at j-1, then*

*continue searching to right of the*

*pattern for border \*/*

**while**(j <= m && pat[i - 1] != pat[j - 1])

{

*/\* the character preceding the occurrence of t*

*in pattern P is different than the mismatching*

*character in P, we stop skipping the occurrences*

*and shift the pattern from i to j \*/*

**if** (shift[j] == 0)

shift[j] = j - i;

*//Update the position of next border*

j = bpos[j];

}

*/\* p[i-1] matched with p[j-1], border is found.*

*store the beginning position of border \*/*

i--; j--;

bpos[i] = j;

}

}

*//Preprocessing for case 2*

**static** **void** preprocess\_case2(**int** []shift, **int** []bpos,

**char** []pat, **int** m)

{

**int** i, j;

j = bpos[0];

**for**(i = 0; i <= m; i++)

{

*/\* set the border position of the first character*

*of the pattern to all indices in array shift*

*having shift[i] = 0 \*/*

**if**(shift[i] == 0)

shift[i] = j;

*/\* suffix becomes shorter than bpos[0],*

*use the position of next widest border*

*as value of j \*/*

**if** (i == j)

j = bpos[j];

}

}

*/\*Search for a pattern in given text using*

*Boyer Moore algorithm with Good suffix rule \*/*

**static** **void** search(**char** []text, **char** []pat)

{

*// s is shift of the pattern*

*// with respect to text*

**int** s = 0, j;

**int** m = pat.length;

**int** n = text.length;

**int** []bpos = **new** **int**[m + 1];

**int** []shift = **new** **int**[m + 1];

*//initialize all occurrence of shift to 0*

**for**(**int** i = 0; i < m + 1; i++)

shift[i] = 0;

*//do preprocessing*

preprocess\_strong\_suffix(shift, bpos, pat, m);

preprocess\_case2(shift, bpos, pat, m);

**while**(s <= n - m)

{

j = m - 1;

*/\* Keep reducing index j of pattern while*

*characters of pattern and text are matching*

*at this shift s\*/*

**while**(j >= 0 && pat[j] == text[s+j])

j--;

*/\* If the pattern is present at the current shift,*

*then index j will become -1 after the above loop \*/*

**if** (j < 0)

{

System.out.printf("pattern occurs at shift = %d\n", s);

s += shift[0];

}

**else**

*/\*pat[i] != pat[s+j] so shift the pattern*

*shift[j+1] times \*/*

s += shift[j + 1];

}

}

*// Driver Code*

**public** **static** **void** main(String[] args)

{

**char** []text = "ABAAAABAACD".toCharArray();

**char** []pat = "ABA".toCharArray();

search(text, pat);

}

}

**Bitwise Algorithms (Problem Type)**

**Count set bits (Strategy: Bit Manipulation)**

**Bit manipulation**

**It refers to the process of manipulating individual bits in a binary representation of data.**

**AND (&) - It performs the AND operation between the corresponding bits of two integers.**

**if both bits are 1 result is 1 otherwise 0.**

**0 1 = 0**

**1 0 = 0**

**1 1 = 1**

**0 0 = 0**

**OR (|) - It performs the OR operation between the corresponding bits of two integers.**

**if both bits are 0 result is 0 otherwise 1.**

**0 1 = 1**

**1 0 =1**

**1 1 =1**

**0 0 = 0**

**XOR (^) - It performs the XOR operation between the corresponding bits of two integers.**

**if both bits are different result is 1 otherwise it is 0.**

**1 0 = 1**

**0 1 = 1**

**1 1 = 0**

**0 0 = 0**

**NOT(~) - ( Inverts) 1 bit becomes 0 and 0 bit becomes 1**

**1 -->0**

**0 -->1**

**Left Shift (<<) - Moves the bits of a number to left by a specified number of positions.( multiply by 2 raised to that power)**

**Right Shift (>>) - - Moves the bits of a number to right by a specified number of positions (division by 2 raised to that power)**

**Count set bits (Strategy: Bit Manipulation)**

**Count Set - Counting the set bits (1 bit ) in an integer.**

**1 - set-bit**

**0 - unset-bit.**

**package bitmanipulation;**

**import java.util.Scanner;**

**public class CountSetBit1**

**{**

**public static int brian(int x)**

**{**

**int r =0;**

**while (x > 0)**

**{**

**x = x & (x-1);**

**r++;**

**}**

**return r;**

**}**

**public static int countsetbit(int x)**

**{**

**int r =0;**

**while (x > 0) {**

**if ((x & 1)== 1) {**

**r++;**

**}**

**x>>=1;**

**}**

**return r;**

**}**

**public static void main(String args[])**

**{**

**int n;**

**Scanner sc = new Scanner(System.in);**

**System.out.println("Enter an integer");**

**n = sc.nextInt();**

**System.out.println("Binary representation of n is : "+ Integer.toBinaryString(n));**

**System.out.println("Count Set Bit with simple bit manipulation (right shift) is : "+ countsetbit(n));**

**System.out.println("Count Set Bit with Briain bit manipulation is : "+ brian(n));**

**}**

**}**

**Find the two non-repeating elements in an array of repeating elements (Strategy: Bit Manipulation)**

Using XOR

Example --

arr = 1 2 3 2 1 4

1 = 0 0 1

2 = 0 1 0

3 = 0 1 1

2 = 0 1 0

1 = 0 0 1

4 = 1 0 0

1, 3 , 1

2 , 2 ,4

package bitmanipulation;  
  
import java.lang.reflect.Array;  
import java.util.Arrays;  
  
public class NonRepeating  
{  
 public static void nonRepeating(int[] arr)  
 {  
 int xor = 0;  
 for(int n:arr)  
 {  
 xor = xor ^ n;  
  
 }  
 int setBit = xor & ~(xor -1);  
 int x = 0, y= 0;  
 for(int num : arr)  
 {  
 if((num & setBit) != 0)  
 {  
 x = x^num;  
 }  
 else  
 {  
 y = y^num;  
 }  
 }  
 System.*out*.println("Non repeating elements in the array are : "+x +" "+y);  
 }  
 public static void main(String args[])  
 {  
 int []arr = {1,2,3,2,1,4};  
 System.*out*.println("Array is :"+ Arrays.*toString*(arr));  
 *nonRepeating*(arr);  
 }  
}

**Swap Bits (Strategy: Bit Manipulation)**

**Count total set bits in all numbers from 1 to n (Strategy: Bit Manipulation)**

**package bitmanipulation;**

**import java.util.Scanner;**

**public class TotalSetBit**

**{**

**public static int counts(int n)**

**{**

**int bitcount =0;**

**for(int i=1;i<=n;i++)**

**bitcount = bitcount +countsetbit(i);**

**return bitcount;**

**}**

**public static int countsetbit(int x)**

**{**

**int r =0;**

**while (x > 0) {**

**if (x % 2 != 0) {**

**r++;**

**}**

**x = x / 2;**

**}**

**return r;**

**}**

**public static void main(String args[])**

**{**

**int n;**

**Scanner sc = new Scanner(System.in);**

**System.out.println("Enter range ");**

**n = sc.nextInt();**

**// System.out.println("Binary representation of n is : "+ Integer.toBinaryString(n));**

**System.out.println(" Total Set Bit count is : "+ counts(n));**

**}**

**}**

**Optimize the Bitwise AND of all subsets of an array (Strategy: Bit Manipulation)**

**Find all Subsets in an Array**

**Arr = {1 ,2, 3}**

**Subsets – {},{1},{2},{3},{1,2},{2,3},{1,3},{1,2,3}**

**Arr.Size = 3.**

**So, subsets = 2 pow 3**

**Understanding Algorithms in Java and Analysing Algorithms**

**Classification of Algorithm: Design Strategies vs Problem Types**

 Classification of algorithms based on design strategies and problem types. Understanding these categories can help you choose the most suitable algorithm for specific tasks. Here are the main classifications:

1. **By Implementation Method**:
   * **Recursion or Iteration**: Recursive algorithms call themselves repeatedly until a base condition is met, while iterative algorithms use loops or data structures (like stacks and queues) to solve problems. For instance, the Tower of Hanoi problem is solved recursively, while the Stock Span problem is tackled iteratively.
   * **Exact or Approximate**: Exact algorithms find optimal solutions for problems, whereas approximation algorithms provide average outcomes based on sub-outcomes. Sorting algorithms fall under the exact category, while NP-Hard problems often use approximation algorithms.
   * **Serial, Parallel, or Distributed Algorithms**: Serial algorithms execute one instruction at a time. Parallel algorithms divide problems into subproblems and run them concurrently on different processors. If these parallel algorithms are distributed across multiple machines, they are called distributed algorithms.
2. **By Design Method**:
   * **Greedy Method**: Greedy algorithms make local decisions at each step without considering future consequences. For example, the Fractional Knapsack and Activity Selection problems are solved using the greedy approach.

**Brute Force Algorithms,**

**Definition**:

* + A brute force algorithm systematically examines all possible solutions until it finds the correct answer.
  + It’s like trying every combination of a lock until you find the right one.

1. **Examples**:
   * **Padlock Combination**: Imagine you have a small padlock with 4 digits (each from 0 to 9). If you forgot the combination, you’d use a brute force method to open it. You’d try all possible combinations: 0001, 0002, 0003, and so on, until it opens.
   * **Traveling Salesman Problem (TSP)**: Given 10 cities, the brute force solution calculates the total distance for every possible route and selects the shortest one.

**Divide and Conquer Algorithms**

**Divide and Conquer** is a powerful algorithmic technique used to solve complex problems by breaking them down into smaller, more manageable sub-problems.

**Basic Idea**:

* + Divide the original problem into two or more similar (but simpler) subproblems.
  + Solve each subproblem independently.
  + Combine the solutions of subproblems to obtain the solution for the original problem.

1. **Examples**:
   * **Merge Sort**: To sort a list of numbers, divide it into two halves, sort each half, and then merge them together.
   * **Quick Sort**: Choose a pivot element, partition the list into elements less than and greater than the pivot, and recursively sort the partitions.
   * **Binary Search**: Divide a sorted list in half and search in the appropriate half based on the target value.
2. **Benefits**:
   * Simplifies complex problems.
   * Enhances computational efficiency by solving smaller subproblems.

**Dynamic Programming Algorithms,**

1. **What is Dynamic Programming?**
   * DP solves problems by dividing them into simpler subproblems and storing their solutions for future use.
   * It avoids redundant computations, leading to efficient solutions for a wide range of problems.
2. **How Does Dynamic Programming Work?**
   * **Identify Subproblems**: Break the main problem into smaller, independent subproblems.
   * **Store Solutions**: Solve each subproblem and store the solution in a table or array.
   * **Build Up Solutions**: Use stored solutions to construct the solution for the main problem.
   * **Avoid Redundancy**: DP ensures each subproblem is solved only once.
3. **Examples of Dynamic Programming**:
   * **Fibonacci Sequence**:
     + Brute Force: Calculate Fibonacci numbers recursively, which can be inefficient.
     + DP Approach: Create a table to store intermediate results and efficiently compute Fibonacci numbers.
   * Other examples include finding the longest common subsequence, shortest path in a graph, and solving the knapsack problem.
4. **When to Use Dynamic Programming?**
   * DP is effective for problems with overlapping subproblems and optimal substructure.

**Greedy Algorithms,**

**Greedy algorithms** are a fascinating class of algorithms that make locally optimal choices at each step, aiming to find a global optimum solution

**Basic Idea**:

* + Greedy algorithms operate on the principle of “taking the best option now” without considering long-term consequences.
  + Decisions are made based on the information available at the current moment.

1. **Examples**:
   * **Fractional Knapsack**: Optimizes the value of items that can be fractionally included in a knapsack with limited capacity.
   * **Dijkstra’s Algorithm**: Finds the shortest path from a source vertex to all other vertices in a weighted graph.
   * **Kruskal’s Algorithm**: Finds the minimum spanning tree of a weighted graph.
   * **Huffman Coding**: Compresses data by assigning shorter codes to more frequent symbols.
2. **Applications**:
   * Greedy algorithms are used in various scenarios:
     + Assigning tasks to resources to minimize waiting time or maximize efficiency.
     + Selecting the most valuable items to fit into a knapsack with limited capacity.
     + Dividing an image into regions with similar characteristics.

**Backtracking Algorithms,**

**Backtracking algorithms** are problem-solving strategies that explore different options to find the best solution. They work by trying out different paths and, if one doesn’t work, backtracking and trying another until the right solution is found

**Definition**:

* + Backtracking involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end.
  + It’s commonly used in situations where you need to explore multiple possibilities, such as searching for a path in a maze or solving puzzles like Sudoku.

1. **General Steps**:
   * Choose an initial solution.
   * Explore all possible extensions of the current solution.
   * If an extension leads to a solution, return that solution.
   * If not, backtrack to the previous decision point and try a different extension.
   * Repeat until all possibilities are explored.
2. **Example**:
   * **Finding the Shortest Path Through a Maze**:
     + Input: A maze represented as a 2D array (0 for open space, 1 for walls).
     + Algorithm:
       - Start at the beginning.
       - For each of the four possible directions (up, down, left, right), try moving in that direction.
       - If moving in that direction leads to the ending point, return the path taken.
       - If not, backtrack to the previous position and try a different direction.
3. **When to Use Backtracking**:
   * Backtracking algorithms are best for problems with multiple possible solutions and where exhaustive search is necessary.

**Randomized Algorithms,**

**Recursive Algorithms,**

**Recursive algorithms** are a fascinating topic in computer science.

**What is Recursion?**

* + Recursion is a programming technique where a function calls itself within its own definition.
  + It allows a function to break down a complex problem into smaller subproblems, which are then solved recursively.

1. **How Does Recursion Work?**
   * Recursion creates a stack of function calls:
     + When a function calls itself, a new instance of the function is created and pushed onto the stack.
     + This process continues until a base case is reached (a condition that stops the recursion).
     + Once the base case is reached, the function calls start popping off the stack, returning their results.
2. **Types of Recursion**:
   * **Direct Recursion**: The method calls itself directly (e.g., factorial calculation).
   * **In-Direct Recursion**: Multiple methods create a circular call sequence (method A calls method B, which then calls method A).
   * **Head Recursion**: Recursive call at the beginning of the method.
   * **Tail Recursion**: Recursive call as the last statement.
3. **Examples of Recursion**:
   * **Factorial**: The factorial of a number (n) is the product of all integers from 1 to (n). It can be defined recursively as: ( \text{factorial}(n) = n \cdot \text{factorial}(n-1) ).
   * **Fibonacci Sequence**: Each number is the sum of the two preceding numbers. Recursively: ( \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) ).
4. **Applications of Recursion**:
   * **Tree and Graph Traversal**: Depth-first search (DFS) and breadth-first search (BFS).
   * **Dynamic Programming**: Solving optimization problems by breaking them into smaller subproblems.
   * **Divide-and-Conquer**: Solving problems by dividing them recursively and combining results.

**Selection of Data Structure and Algorithm to solve a problem**

When choosing a data structure and algorithm to solve a problem, consider the following steps:

1. **Analyze the Problem**:
   * Understand the problem requirements and constraints.
   * Identify the basic operations needed (e.g., insertion, deletion, search).
2. **Select an Appropriate Data Structure**:
   * **Arrays**: Use when you need constant-time access to elements by index.
   * **Linked Lists**: Useful for dynamic data with frequent insertions and deletions.
   * **Trees (BST, AVL, etc.)**: Ideal for hierarchical data or efficient searching.
   * **Hash Tables**: Efficient for key-value lookups.
   * **Graphs**: Solve problems involving relationships or networks.
3. **Consider Time Complexity**:
   * Choose data structures that optimize for the most frequent operations.
   * For example, if searching is common, use a hash table or a balanced tree.
4. **Algorithm Selection**:
   * Based on the problem type (sorting, searching, graph traversal, etc.), choose an appropriate algorithm.
   * Consider trade-offs between time complexity (big O notation) and space complexity.

Remember that there’s no one-size-fits-all solution. Each problem may require a different combination of data structures and algorithms