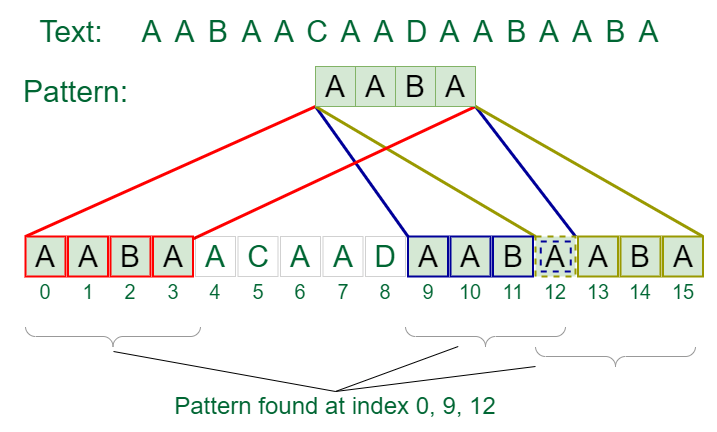
Pattern Searching

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*



*Pattern searching using* ***Naive algorithm***

|  |
| --- |
| **public** **class** Pattern {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  String str="AABAACAADAABAABA";  String pattern="AABA";  *findPattern*(str,pattern);    }  **private** **static** **void** findPattern(String str, String pattern) {  // **TODO** Auto-generated method stub  **int** main\_len=str.length();  **int** sub\_len=pattern.length();  **boolean** flag=**false**;  **for**(**int** i=0;i<=main\_len-sub\_len;i++)  {**int** j;  **for**( j=0;j<sub\_len;j++)  {  **if**(str.charAt(i+j)!=pattern.charAt(j))    **break**;  }  **if**(j==sub\_len)  {  System.***out***.println(i);  flag=**true**;  }  }    **if**(flag==**false**)  System.***out***.println("No matching found");        }  } |

Time complexity is o(m\*n)

m---main string;

n—pattern string length.

Space complexity---o(1)

Using KMP algorithm

|  |
| --- |
| **public** **class** Example1 {      **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  String text="ababcabcabababd";  String pattern="ababd";  *KMPsearching*(pattern, text);      }  **private** **static** **void** KMPsearching(String pattern, String text) {  // **TODO** Auto-generated method stub  **int** M=pattern.length();  **int** N=text.length();  **int** lps[]=*computeLPSArray*(pattern);    **int** j=0;  **int** i=0;  **while**(i<N)  {  **if**(pattern.charAt(j)==text.charAt(i))  {  i++;  j++;  }  **if**(j==M)  {  System.***out***.print((i-j)+ " ");  j=lps[j-1];  }  **else** **if**(i<N && pattern.charAt(j)!=text.charAt(i))  {  **if**(j!=0)  {  j=lps[j-1];  }  **else**  {  i++;  }  }    }    }  **private** **static** **int**[] computeLPSArray(String pattern) {  // **TODO** Auto-generated method stub  **int** M=pattern.length();  **int** lps[]=**new** **int**[M];  **int** length=0; // length of previous longest prefix or suffix  lps[0]=0;  **int** i=1;    **while**(i<M)  {  **if**(pattern.charAt(i)==pattern.charAt(length))  {  length++;  lps[i]=length;  i++;  }  **else**  {  **if**(length!=0)  {  length=lps[length-1];  }  **else** {  lps[i]=0;  i++;  }  }  }    **for**(**int** z=0;z<M;z++)  {  System.***out***.print(lps[z]+" ");  }      System.***out***.println();  **return** lps;        }  } |

Time

Main string -size m

Pattern string -size n

O(m+n)

Space-o(n)

**Prefix and suffix**

Given a string **s**, find the length of the longest prefix, which is also a suffix. The prefix and suffix should not overlap.

**Examples:**

***Input :****S = aabcdaabc****Output :****4****Explanation:****The string “aabc” is the longest prefix which is also suffix.*

***Input :****S****=****abcab****Output :****2*

***Input :****S = aaaa****Output :****2*

|  |
| --- |
| **public** **class** Prefix\_Suffix\_KMP {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  String str="aabcdaabc";  **int** res=*prefixSuffix*(str);  System.***out***.println(res);  }  **private** **static** **int** prefixSuffix(String str) {  // **TODO** Auto-generated method stub  **int** n=str.length();    **int** lps[]=**new** **int**[n];    **int** len=0;  **int** i=1;  lps[0]=0;  **while**(i<n)  {  **if**(str.charAt(i)==str.charAt(len))  {  len++;  lps[i]=len;  i++;  }  **else**  {  **if**(len!=0)  {  len=lps[len-1];  }  **else**  {  i++;  }  }  }  **return** lps[n-1];  }  } |

Find the all prime numbers using Sieve of Eranthosis

|  |  |
| --- | --- |
| public class PrimeNumbersNaive {  public static void main(String[] args) {  System.out.println("Prime numbers from 1 to 10:");  for (int num = 1; num <= 10; num++) {  if (isPrime(num)) {  System.out.print(num + " ");  }  }  }  // Function to check if a number is prime  public static boolean isPrime(int n) {  if (n < 2) return false; // 0 and 1 are not prime  for (int i = 2; i < n; i++) { // Check divisibility from 2 to n-1  if (n % i == 0) {  return false; // Not a prime number  }  }  return true; // Prime number  }  }  Time complexity-o(n^2)  Space complexity o(1) | **public** **class** Example {  **public** **static** **void** main(String[] args) {  **int** n=100;  *seiveAlgorithm*(n);  }  **private** **static** **void** seiveAlgorithm(**int** n) {  **boolean** primes[] = **new** **boolean**[n+1];  **for**(**int** i=0;i<=n;i++)  {  primes[i]=**true**;  }    **for**(**int** p=2;p\*p<=n;p++)  {  **if**(primes[p]==**true**)  {  **for**(**int** i=p\*p;i<=n;i=i+p)  {  primes[i]=**false**;  }    }  }    **for**(**int** i=2;i<=n;i++)  {  **if**(primes[i]==**true**)  System.***out***.print(i+" ");  }    }  } |

Time complexity is nloglogn.

Space complexity is o(n)

[**Minimum Number of Swaps to Make the String Balanced**](https://leetcode.com/problems/minimum-number-of-swaps-to-make-the-string-balanced/)

You are given a **0-indexed** string s of **even** length n. The string consists of **exactly** n / 2 opening brackets '[' and n / 2 closing brackets ']'.

A string is called **balanced** if and only if:

* It is the empty string, or
* It can be written as AB, where both A and B are **balanced** strings, or
* It can be written as [C], where C is a **balanced** string.

You may swap the brackets at **any** two indices **any** number of times.

Return *the****minimum****number of swaps to make*s ***balanced***.

**Example 1:**

**Input:** s = "][]["

**Output:** 1

**Explanation:** You can make the string balanced by swapping index 0 with index 3.

The resulting string is "[[]]".

**Example 2:**

**Input:** s = "]]][[["

**Output:** 2

**Explanation:** You can do the following to make the string balanced:

- Swap index 0 with index 4. s = "[]][][".

- Swap index 1 with index 5. s = "[[][]]".

The resulting string is "[[][]]".

**Example 3:**

**Input:** s = "[]"

**Output:** 0

**Explanation:** The string is already balanced.

**Constraints:**

* n == s.length
* 2 <= n <= 106
* n is even.
* s[i] is either '[' or ']'.
* The number of opening brackets '[' equals n / 2, and the number of closing brackets ']' equals n / 2.

|  |
| --- |
| public class Example {  public static void main(String[] args) {  //String s="[[[]]]";  String s="]]][[[";  int balance=0;  int maxunbalanced=0;  for(int i=0;i<s.length();i++)  {  if(s.charAt(i)=='[')  balance++;  else  balance--;  maxunbalanced=Math.min(balance, maxunbalanced);  //System.out.println(maxunbalanced);  }  int res=Math.abs(maxunbalanced);  System.out.println((res+1)/2);  }  }  Time complexity is o(n)  Space complexity is o(1) |

Count Number of Zeros in a Sorted Array

**Problem Statement:**

You are given a sorted binary array arr of size n, containing only 0s and 1s. Your task is to determine the number of 0s in the array using an efficient approach.

**Constraints:**

* The array is sorted in non-decreasing order.
* The array consists of only two distinct elements: 0 and 1.
* 1≤n≤10^6
* **Input:**
* A single integer n representing the size of the array.
* A sorted binary array arr of size n.

**Output:**

* An integer representing the number of 0s in the array.

**Requirements:**

* Your solution must use **binary search** for efficiency.
* The time complexity should be O(logn).

Input:

[0, 0, 0, 0, 1, 1, 1]

Output:

4

[1, 1, 1, 1, 1]

Output: 0

[0, 0, 0, 0,0]

Output: 5

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  **int** arr[]= {0,0,0,0,1,1,1,1};    **int** l=0;  **int** h=arr.length-1;  **int** firstIndex=-1;  **while**(l<=h)  {  **int** mid=(l+h)/2;    **if**(arr[mid]==0)  {  l=mid+1;    }  **else** **if**(arr[mid]==1)  {  firstIndex=mid;  h=mid-1;    }      }  System.***out***.println(firstIndex==-1?arr.length:firstIndex);      }  } |