**Z algorithms Algorithm: TC: O(m+n)**

**Code:**

*package* StringProblems;  
*/\*\*  
 \* Created By Ravi on 28-01-2023  
 \*\*/  
public class* ZAlgorithm {  
 *public static void* main(String[] args) {  
 String s = "abcdefgkijklghifghsd";  
 String p = "ghi";  
 *int* idx = *isMatched*(s, p);  
 *if*(idx != -1)  
 {  
 System.***out***.println("Pattern Matched at index: " + idx);  
 }  
 *else* {  
 System.***out***.println("Pattern does not found !!!");  
 }  
 }  
 *private static int* isMatched(String s, String p)  
 {  
 String newStr = p + "$"+s;  
 *int*[] z = *zScore*(newStr);  
 *for*(*int* i =0; i<z.length; i++)  
 {  
 *if*(z[i] == p.length())  
 {  
 *return* i-p.length()-1; *// original = modified-pattern-$* }  
 }  
 *return* -1;  
 }  
 *private static int*[] zScore(String s)  
 {  
 *int* n = s.length();  
 *int*[] Z = *new int*[n];  
  
 *//window left, right pointer  
 int* left = 0;  
 *int* right =0;  
 Z[0] = n;  
 *for*(*int* i =1; i<n; i++)  
 {  
 *//case: i> right : nothing matches: will use brute force to calculate z score  
 if*(i > right)  
 {  
 left = right = i;  
  
 *while*(right <n && s.charAt(right-left) == s.charAt(right))  
 {  
 right++;  
 }  
 Z[i] = right-left;  
 right--;  
 }  
 *else* {  
 *//k = i-left: number of matches in [left, right] interval  
 int* k = i-left;  
 *if*(Z[k] < right -i +1)  
 {  
 Z[i] = Z[k];  
 }  
 *else* {  
 left =i;  
 *while*(right<n && s.charAt(right-left) == s.charAt(right))  
 {  
 right++;  
 }  
 Z[i] = right -left;  
 right--;  
 }  
 }  
 }  
 *return* Z;  
 }  
}

2. Rabin Carp Method:

*package* StringProblems;  
  
*/\*\*  
 \* Created By Ravi on 29-01-2023  
 \*\*/  
public class* RabinCarp {  
 *public static final int* ***d*** = 256;  
 *public static void* main(String[] args) {  
 String text = "abaabcdefaabaa";  
 String pattern = "abaa";  
 *int* q = 101;  
 *search*(pattern, text, q);  
   
 }  
  
 *private static void* patternMatch\_bf(String s, String p)  
 {  
 *int* m = s.length();  
 *int* n = p.length();  
 *for*(*int* i =0; i<=m-n; i++)  
 {  
 *int* j;  
 *for*(j =0; j<n; j++)  
 {  
 *if*(s.charAt(i+j) != s.charAt(j))  
 {  
 *break*;  
 }  
 }  
 *if*(j == n)  
 {  
 System.***out***.print(i + " ");  
 }  
 }  
 }  
 *private static void* search(String pat, String txt, *int* q)  
 {  
 *int* M = pat.length();  
 *int* N = txt.length();  
 *int* i, j;  
 *//find hash value of pat;  
 int* p =0;  
 *//hash value of txt  
 int* t =0;  
 *int* h = 1;  
  
 *//h -> pow(h,m-1)%q  
 for*(i =0; i<M-1; i++)  
 {  
 h = (h\****d***)%q;  
 }  
 *for*(i =0; i<M; i++)  
 {  
 p = (***d***\*p+pat.charAt(i))%q;  
 t =(***d***\*t+txt.charAt(i))%q;  
 }  
  
 *//slide  
 for*( i =0; i<=N-M; i++)  
 {  
 *if*(p == t)  
 {  
 *//compare char by char  
 for*(j =0; j<M; j++)  
 {  
 *if*(txt.charAt(i+j) != pat.charAt(j))  
 {  
 *break*;  
 }  
 }  
 *if*(j ==M)  
 {  
 System.***out***.println(i);  
 }  
 }  
 *if*(i<N-M)  
 {  
 t = (***d***\*(t-txt.charAt(i)\*h) + txt.charAt(i+M))%q;  
  
 *if*(t<0)  
 {  
 t = t+q;  
 }  
 }  
 }  
 }  
   
}

3. KMP Algorithm:

*package* StringProblems;  
  
*import* javax.swing.plaf.synth.SynthOptionPaneUI;  
*import* java.util.Arrays;  
  
*/\*\*  
 \* Created By Ravi on 30-01-2023  
 \*\*/  
public class* KMPAlgo {  
 *public static void* main(String[] args) {  
 String text = "abaabcdefaabaa";  
 String pattern = "xabaa";  
*// String txt = "abcdabcy";  
// String p ="bababaa$aababab";* String p = "aacecaaa$aaacecaa";  
 *int*[] lps = *createLPS*(p);  
 Arrays.*stream*(lps).forEach(i -> System.***out***.print(i + " "));  
*// KMPAlgorith(text, pattern);* }  
  
 *private static void* KMPAlgorith(String t, String p)  
 {  
 *int*[] lps = *createLPS*(p);  
 *char*[] text = t.toCharArray();  
 *int* m= text.length;  
 *char*[] pat = p.toCharArray();  
 *int* n = pat.length;  
 *int* i =0;  
 *int* j =0;  
 *while* (i<m && j<n)  
 {  
 *if*(text[i] == pat[j])  
 {  
 i++;  
 j++;  
 }  
 *else* {  
 *if*(j != 0)  
 {  
 j = lps[j-1];  
 }  
 *else* {  
 i++;  
 }  
 }  
 }  
 *if*(j ==n)  
 {  
 System.***out***.println("Pattern Matched !!");  
 }  
 *else* {  
 System.***out***.println("Pattern did not found !!");  
 }  
  
 }  
 *private static int*[] createLPS(String s)  
 {  
 *char*[] chars = s.toCharArray();  
 *int* n = chars.length;  
 *int*[] lps = *new int*[n];  
 *int* i =0;  
 *int* j =1;  
 *while*(j<n)  
 {  
 *if*(chars[i] == chars[j])  
 {  
 lps[j] = i+1;  
 i++;  
 j++;  
 }  
 *else* {  
 *if*(i>0)  
 {  
 i = lps[i-1];  
 }  
 *else* {  
 j++;  
 }  
  
 }  
 }  
 *return* lps;  
 }  
}

*private static void* KMPAlgorithm2(String t, String p)  
{  
 *//create LPS for pattern  
 int*[] lps = *createLPS*(p);  
 *char*[] text = t.toCharArray();  
 *int* m= text.length;  
 *char*[] pat = p.toCharArray();  
 *int* n = pat.length;  
 *int* i =0;  
 *int* j =0;  
  
 *//List to store Match index  
  
 while* (i<m)  
 {  
 *if*(text[i] == pat[j]) {  
 i++;  
 j++;  
 *if* (j == n) {  
 System.***out***.println(i - j);  
 j = lps[j - 1];  
 }  
 }  
 *else if*(i<m && text[i] != pat[j])  
 {  
 *if*(j != 0)  
 {  
 j = lps[j-1];  
 }  
 *else* {  
 i++;  
 }  
 }  
   
 }  
  
}

# Morris Traversal: InOrder

*class* Solution {  
 *public* List<Integer> inorderTraversal(TreeNode root) {  
 List<Integer> inorder = *new* ArrayList<Integer>();  
  
 TreeNode cur = root;  
 *while*(cur != *null*) {  
 *if*(cur.left == *null*) {  
 inorder.add(cur.val);  
 cur = cur.right;  
 }  
 *else* {  
 TreeNode prev = cur.left;  
 *while*(prev.right != *null* && prev.right != cur) {  
 prev = prev.right;  
 }  
  
 *if*(prev.right == *null*) {  
 prev.right = cur;  
 cur = cur.left;  
 }  
 *else* {  
 prev.right = *null*;  
 inorder.add(cur.val);  
 cur = cur.right;  
 }  
 }  
 }  
 *return* inorder;  
 }  
}

# Morris Traversal : PreOrder

*public class* Solution {  
 *static* ArrayList<Integer> preorderTraversal(Node root) {  
 ArrayList<Integer> preorder = *new* ArrayList<>();  
 Node cur = root;  
 *while* (cur != *null*) {  
 *if* (cur.left == *null*) {  
 preorder.add(cur.data);  
 cur = cur.right;  
 } *else* {  
 Node prev = cur.left;  
 *while* (prev.right != *null* && prev.right != cur) {  
 prev = prev.right;  
 }  
  
 *if* (prev.right == *null*) {  
 prev.right = cur;  
 preorder.add(cur.data);  
 cur = cur.left;  
 } *else* {  
 prev.right = *null*;  
 cur = cur.right;  
 }  
 }  
 }  
 *return* preorder;  
 }  
}

#Morris: PostOrder

*class* Solution {  
 *public*:  
 vector<*int*> postorderTraversal(TreeNode\* root) {  
 vector<*int*> ans;  
 *while*(root)  
 {  
 TreeNode \*curr = root;*//We will create thread from left most node of right subtree to present node and will travell to that node using curr  
  
 if*(curr->right)*//if root has right child  
 //We can't push directly this root node val to ans as we are not sure whether we are here  
 //thorough thread link after covering right subtree or we are here for the first time* {  
 curr = curr->right;  
 *while*(curr->left && curr->left != root)*//go to left most node of right subtree* curr=curr->left;  
 *if*(curr->left != root)*//not threaded yet* {  
 ans.push\_back(root->val);*//it means root was visited for first time and this is modified preorder hence   
 //push this node's val to ans* curr->left = root;*//create the thread* root = root->right;*//go to right to cover right subtree as modified preorder is root->right->left* }  
 *else//was threaded* {  
 curr->left = NULL;*//break the thread* root = root->left;*//right subtree has been covered hence now cover the left one  
 //no need to push this node value as we are here for the second time using thread  
 //link* }  
 }  
 *else//root hasn't right child* {  
 ans.push\_back(root->val);*//modified preorder is root->right->left hence push this value before going to left* root = root->left;  
 }  
 }  
 reverse(ans.begin(),ans.end());*//reversing root->right->left to left->right->root to make it post order  
 return* ans;  
 }  
};

#InOrder To PostFix:

public class Solution {

    public String solve(String A) {

        StringBuilder ans  = new StringBuilder();

        Stack<Character> st = new Stack<>();

        for(int i=0; i<A.length(); i++)

        {

            char ch = A.charAt(i);

            //case 1: if char is operand: add to results

            if(isOperand(ch))

            {

                ans.append(ch);

            }

            //case:02 open bracket: push to stack

            else if(ch=='(')

            {

                st.push('(');

            }

            //case: 3: closed bracket: pop the stack and append the result to ans

            else if(ch == ')')

            {

                while(st.peek() != '(')

                {

                    ans.append(st.peek());

                    st.pop();

                }

                st.pop(); //pop extra (

            }

            else //operator

            {

                while(!st.isEmpty() && precedence(ch) <= precedence(st.peek()))

                {

                    ans.append(st.peek());

                    st.pop();

                }

                //push to stack

                st.push(ch);

            }

        }

        while(!st.isEmpty())

        {

            ans.append(st.pop());

        }

        return ans.toString();

    }

    private int precedence(char ch)

    {

        if(ch == '^') return 3;

        if(ch == '\*' || ch == '/') return 2;

        if(ch =='+' || ch == '-') return 1;

        else return 0;

    }

    private boolean isOperand(char ch)

    {

        if(ch>='a' && ch<='z') return true;

        return false;

    }

}

#Quick Sort

*package* Sorting;  
  
*import* java.util.ArrayList;  
*import* java.util.Arrays;  
*/\*\*  
 \* Created By Ravi on 03-02-2023  
 \*\*/  
public class* QuickSortClass {  
 *public static void* main(String[] args) {  
 *int*[] arr = {1,2,3,6,5,4,3,-1,-0,91, 1000, -10000};  
 *int* n = arr.length;  
 *quickSort*(arr, 0, n-1);  
 Arrays.*stream*(arr).forEach(i -> System.***out***.print(i+" "));  
 }  
 *private static void* quickSort(*int*[] arr, *int* left, *int* right) {  
 *//base case  
 if*(left >= right)  
 {  
 *return*;  
 }  
 *int* pivot = *pivot*(arr, left, right);  
 *quickSort*(arr, left, pivot-1);  
 *quickSort*(arr, pivot+1, right);  
 }  
 *private static int* pivot(*int*[] arr, *int* left, *int* right)  
 {  
 *int* i =left;  
 *int* j = right;  
 *while*(i < j)  
 {  
 *if*(arr[i]<=arr[left])  
 {  
 i++;  
 }  
 *else if*(arr[j] > arr[left])  
 {  
 j--;  
 }  
 *else* {  
 *//swap  
 swap*(arr, i, j);  
 i++;  
 j--;  
 }  
 }  
 *//set i to prev* i--;  
 *swap*(arr, i, left);  
 *return* i;  
  
 }  
  
 *private static void* swap(*int*[] arr, *int* i, *int* j)  
 {  
 *int* temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 }  
}

#Merge Sort

*package* Sorting;  
*import* java.util.Arrays;  
*/\*\*  
 \* Created By Ravi on 02-02-2023  
 \*\*/  
public class* MergeSortClass {  
 *public static void* main(String[] args) {  
 *int*[] arr = {1,2,5,3,2,3,8,6,9, -1, -100, 100, -111, 0, 0, 0, -1, 1001,};  
 *int* n = arr.length;  
 *mergesort*(arr, 0, n-1);  
 System.***out***.print("\nAfter sorting: ");  
 Arrays.*stream*(arr).forEach(i-> System.***out***.print(i+" "));  
 }  
 *private static void* mergesort(*int*[] arr, *int* left, *int* right) {  
 *//base case  
 if*(left == right)  
 {  
 *return*;  
 }  
 *//find mid  
 int* mid = (left+right)/2;  
 *//sort left part  
 mergesort*(arr, left, mid);  
 *//sort right part  
 mergesort*(arr, mid+1, right);  
 *//merge two sorted part: left and right  
 merge*(arr, left, mid, right);  
 }  
 *private static void* merge(*int*[] arr, *int* left, *int* mid, *int* right)  
 {  
 *int*[] sorted = *new int*[right-left+1];  
 *int* k =0;  
  
 *int* i = left;  
 *int* j = mid+1;  
 *while*(i<=mid && j<=right)  
 {  
 *if*(arr[i]<=arr[j])  
 {  
 sorted[k++] = arr[i];  
 i++;  
 }  
 *else* {  
 sorted[k++] = arr[j];  
 j++;  
 }  
 }  
 *while*(i<=mid)  
 {  
 sorted[k++] = arr[i];  
 i++;  
 }  
 *while*(j<=right)  
 {  
 sorted[k++] = arr[j];  
 j++;  
 }  
 *//copy to arr  
 for*(*int* id =left; id<=right; id++)  
 {  
 arr[id] = sorted[id-left];  
 }  
 }

}

*private static void* insertionSort(*int*[] arr)  
{  
 *int* n = arr.length;  
  
 *for*(*int* i = 1; i<n; i++)  
 {  
 *//take a element from right(unsorted part) and put at sorted pos in left part(sorted)  
 int* val = arr[i];  
 *int* j = i;  
 *while*(j >0 && val<arr[j-1])  
 {  
 arr[j] = arr[j-1];  
 j--;  
 }  
 arr[j] = val;  
 }  
}

#Counting Sort

*package* Sorting;  
*import* java.util.Arrays;  
*/\*\*  
 \* Created By Ravi on 03-02-2023  
 \*\*/  
public class* CountingSortClass {  
 *public static void* main(String[] args) {  
  
 *int*[] arr = {1,2,1,2,1,2,1,4,3,3,3,5,5,5,4};  
 *countingSort*(arr);  
 Arrays.*stream*(arr).forEach(i -> System.***out***.print(i + " "));  
 }  
 *private static void* countingSort(*int*[] arr)  
 {  
 *int* n = arr.length;  
 *int* max = Arrays.*stream*(arr).max().getAsInt();  
 *int*[] count = *new int*[max+1];  
 *for*(*int* a: arr)  
 {  
 count[a]++;  
 }  
 *//commutative count  
 for*(*int* i =1; i<count.length; i++)  
 {  
 count[i] += count[i-1];  
 }  
  
 *int*[] sorted = *new int*[n];  
 *for*(*int* i = n-1; i>=0; i--)  
 {  
 sorted[count[arr[i]]-1] = arr[i];  
 count[arr[i]]--;  
 }  
 *//copy  
 for*(*int* i =0; i<n; i++)  
 {  
 arr[i] = sorted[i];  
 }  
 }  
}

#Radix Sort

*package* Sorting;  
*import* java.util.Arrays;  
*/\*\*  
 \* Created By Ravi on 04-02-2023  
 \*\*/  
public class* RadixSortClass {  
 *public static void* main(String[] args) {  
 *int*[] arr = {11,123,435,54,6,897,54};  
 *int* n = arr.length;  
 *radixSort*(arr, n);  
 Arrays.*stream*(arr).forEach(i -> System.***out***.print(i + " "));  
 }  
 *private static void* radixSort(*int*[] arr, *int* n) {  
 *int* max = Arrays.*stream*(arr).max().getAsInt();  
 *for*(*int* div = 1; max/div>0; div\*=10)  
 {  
 *countingSort*(arr, n, div);  
 }  
 }  
 *private static void* countingSort(*int*[] arr, *int* n, *int* div)  
 {  
 *//output array  
 int*[] output = *new int*[n];  
 *//count digit array  
 int*[] count = *new int*[10];  
 *//count digits  
 for*(*int* num: arr)  
 {  
 count[(num/div)%10]++;  
 }  
 *//commutative count  
 for*(*int* i =1; i<10; i++)  
 {  
 count[i] += count[i-1];  
 }  
 *//sorted array  
 int*[] sorted = *new int*[n];  
  
 *for*(*int* i = n-1; i>=0; i--)  
 {  
 *int* digit = (arr[i]/div)%10;  
 sorted[count[digit]-1] = arr[i];  
 count[digit]--;  
 }  
 *//copy to original aray  
 for*(*int* i=0; i<n; i++)  
 {  
 arr[i] = sorted[i];  
 }  
  
 }  
  
}

# Heap Sort and Heapify

*package* Sorting;  
*import* java.util.Arrays;  
*import* java.util.Vector;  
*/\*\*  
 \* Created By Ravi on 04-02-2023  
 \*\*/  
public class* HeapSortClass {  
 *public static void* main(String[] args) {  
 *int*[] test = {2,3,7,9,1,8,232, -1,0, 0,0,0,0,-1,-2};  
 *int* n = test.length;  
 *heapSort*(test, n);  
*// buildHeap(test, n);* Arrays.*stream*(test).forEach(i -> System.***out***.print(i+" "));  
  
 }  
 */\*\*  
 \* Function to heap sort a given array  
 \** ***@param arr****: input array  
 \** ***@param n*** *size of array  
 \*/  
 private static void* heapSort(*int*[] arr, *int* n)  
 {  
 *//build heap  
 buildHeap*(arr, n);  
 *//delete one by one  
 while*(n>0)  
 {  
 arr[n-1] = *deleteMin*(arr, n);  
 n--;  
 }  
 }  
 */\*\*  
 \* Function to delete min element(top) from min heap  
 \** ***@param*** *arr as heap  
 \** ***@param n****: size of array  
 \*/  
 private static int* deleteMin(*int*[] arr, *int* n)  
 {  
 *int* top = arr[0];  
 *swap*(arr, 0, n-1);  
 *heapify*(arr, n-1, 0);  
 *return* top;  
 }  
  
  
 */\*\*  
 \* Function to build heap from array  
 \** ***@param arr*** *to build heap  
 \** ***@param n****: size of array  
 \*/  
 private static void* buildHeap(*int*[] arr, *int* n)  
 {  
 *//start frm last non leaf node  
 for*(*int* i =n/2-1; i>=0; i--)  
 {  
 *heapify*(arr, n, i);  
 }  
 }  
  
 */\*\*  
 \* Function to convert array to heap  
 \** ***@param arr****: array to be headpiece  
 \** ***@param n****: size of array  
 \** ***@param i****: starting index : current parent(root)  
 \*/  
 private static void* heapify(*int*[] arr, *int* n, *int* i)  
 {  
 *int* largest = i; *//root  
 int* leftChild = 2\*i+1;  
 *int* rightChild = 2\*i+2;  
  
 *if*(leftChild<n && arr[leftChild]>arr[i])  
 {  
 largest = leftChild;  
 }  
 *if*(rightChild<n && arr[rightChild]>arr[largest])  
 {  
 largest = rightChild;  
 }  
  
 *//check if root is largest or not  
 if*(largest != i)  
 {  
 *swap*(arr, i, largest);  
  
 *heapify*(arr, n, largest);  
  
 }  
 }  
  
 */\*\*  
 \* Function to swap values of two indices in array  
 \** ***@param arr*** *array  
 \** ***@param i*** *index i  
 \** ***@param j*** *index j  
 \*/  
 private static void* swap(*int*[] arr, *int* i, *int* j)  
 {  
 *int* temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 }  
}

# Heap Sort

*package* Sorting;  
*import* java.util.Arrays;  
*import* java.util.Vector;  
*/\*\*  
 \* Created By Ravi on 04-02-2023  
 \*\*/  
public class* HeapSortClass {  
 *public static void* main(String[] args) {  
 *int*[] test = {2,3,7,9,1,8,232, -1,0, 0,0,0,0,-1,-2};  
 *int* n = test.length;  
 *heapSort*(test, n);  
*// buildHeap(test, n);* Arrays.*stream*(test).forEach(i -> System.***out***.print(i+" "));  
  
 }  
 */\*\*  
 \* Function to heap sort a given array  
 \** ***@param arr****: input array  
 \** ***@param n*** *size of array  
 \*/  
 private static void* heapSort(*int*[] arr, *int* n)  
 {  
 *//build heap  
 buildHeap*(arr, n);  
 *//delete one by one  
 while*(n>0)  
 {  
 arr[n-1] = *deleteMin*(arr, n);  
 n--;  
 }  
 }  
 */\*\*  
 \* Function to delete min element(top) from min heap  
 \** ***@param arr*** *as heap  
 \** ***@param n****: size of array  
 \*/  
 private static int* deleteMin(*int*[] arr, *int* n)  
 {  
 *int* top = arr[0];  
 *swap*(arr, 0, n-1);  
 *heapify*(arr, n-1, 0);  
 *return* top;  
 }  
  
  
 */\*\*  
 \* Function to build heap from array  
 \** ***@param arr*** *to build heap  
 \** ***@param n****: size of array  
 \*/  
 private static void* buildHeap(*int*[] arr, *int* n)  
 {  
 *//start frm last non leaf node  
 for*(*int* i =n/2-1; i>=0; i--)  
 {  
 *heapify*(arr, n, i);  
 }  
 }  
  
 */\*\*  
 \* Function to convert array to heap  
 \** ***@param arr****: array to be headpiece  
 \** ***@param n****: size of array  
 \** ***@param i****: starting index : current parent(root)  
 \*/  
 private static void* heapify(*int*[] arr, *int* n, *int* i)  
 {  
 *int* largest = i; *//root  
 int* leftChild = 2\*i+1;  
 *int* rightChild = 2\*i+2;  
  
 *if*(leftChild<n && arr[leftChild]>arr[i])  
 {  
 largest = leftChild;  
 }  
 *if*(rightChild<n && arr[rightChild]>arr[largest])  
 {  
 largest = rightChild;  
 }  
  
 *//check if root is largest or not  
 if*(largest != i)  
 {  
 *swap*(arr, i, largest);  
  
 *heapify*(arr, n, largest);  
  
 }  
 }  
  
 */\*\*  
 \* Function to swap values of two indices in array  
 \** ***@param arr*** *array  
 \** ***@param i*** *index i  
 \** ***@param j*** *index j  
 \*/  
 private static void* swap(*int*[] arr, *int* i, *int* j)  
 {  
 *int* temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 }  
}

**DijKastra Algorithm:**

*public class* Main {  
 *static class* Edge {  
 *int* src;  
 *int* nbr;  
 *int* wt;  
 Edge(*int* src, *int* nbr, *int* wt) {  
 *this*.src = src;  
 *this*.nbr = nbr;  
 *this*.wt = wt;  
 }  
 }  
 *static class* Pair *implements Comparable*<Pair>{  
 *int* vtx;  
 String psf;  
 *int* wsf;  
 Pair(*int* vtx,String psf,*int* wsf){  
 *this*.vtx=vtx;  
 *this*.psf=psf;  
 *this*.wsf=wsf;  
 }  
 *public int* compareTo(Pair o){  
 *return this*.wsf-o.wsf;  
 }  
 }  
   
 *public static void* main(String[] args) *throws* Exception {  
 BufferedReader br = *new* BufferedReader(*new* InputStreamReader(System.***in***));  
  
 *int* vtces = Integer.*parseInt*(br.readLine());  
 ArrayList<Edge>[] graph = *new* ArrayList[vtces];  
 *for* (*int* i = 0; i < vtces; i++) {  
 graph[i] = *new* ArrayList<>();  
 }  
 *int* edges = Integer.*parseInt*(br.readLine());

*for* (*int* i = 0; i < edges; i++) {  
 String[] parts = br.readLine().split(" ");  
 *int* v1 = Integer.*parseInt*(parts[0]);  
 *int* v2 = Integer.*parseInt*(parts[1]);  
 *int* wt = Integer.*parseInt*(parts[2]);  
 graph[v1].add(*new* Edge(v1, v2, wt));  
 graph[v2].add(*new* Edge(v2, v1, wt));  
 }  
  
 *int* src = Integer.*parseInt*(br.readLine());  
 *boolean*[] visited=*new boolean*[vtces];

PriorityQueue<Pair> pq=*new* PriorityQueue<>();  
 pq.add(*new* Pair(src, src+"", 0));  
 *while*(pq.size()>0){  
 Pair rem=pq.remove();  
 *if*(visited[rem.vtx]==*true*){  
 *continue*;  
 }  
 visited[rem.vtx]=*true*;  
 System.***out***.println(rem.vtx+" via "+ rem.psf+" @ "+ rem.wsf);  
  
 *for*(Edge e:graph[rem.vtx]){  
 *if*(visited[e.nbr]==*false*){  
 pq.add(*new* Pair(e.nbr,rem.psf+e.nbr,rem.wsf+e.wt));  
 }  
 }  
 }  
  
 }

#Bellman Ford

*public class* BellMan {  
 *public static void* main(String[] args) {  
   
 }  
 *private static int*[] bellmanFord(*int* edges[][],*int* V,*int* src){  
  
 *// Step 1 - Creating a V sized array/list,  
 // and initializing it with a very big value.  
 // Creating a vector dis of size V with values as INT\_MAX.  
 int* dis[]=*new int*[V];  
 Arrays.fill(dis,Integer.***MAX\_VALUE***);  
 dis[src]=0; *// Since we are already on source vertex, we can reach it within no time.  
  
 // Step 2 - For V-1 times, traversing over,  
 // all the edges and checking if a shorter  
 // path between any edge u to v is possible.   
 int* u,v,wt;  
 *for*(*int* i=0;i<n-1;i++) *// Iterating V-1 times* {  
 *for*(*int* j=0;j<edges.length;j++) *// Iterating over all the edges.* {  
 u=edges[j][0]; *// Source vertex.* v=edges[j][1]; *// Destination vertex.* wt=edges[j][2];*// Weight of the edge.   
  
 // If we can reach v from u in less time it   
 // is currently required to reach v then update   
 // the value.  
 if*(dis[u]!=Integer.***MAX\_VALUE***&&dis[u]+wt<dis[v])  
 dis[v]=dis[u]+wt;  
 }  
 }  
 *// Step 3 - Checking for negative edge weight cycle,   
 // by checking if the underliying condition satifies.  
 for*(*int* j=0;j<edges.length;j++)  
 {  
 u=edges[j][0];  
 v=edges[j][1];  
 wt=edges[j][2];  
  
 *// If the below condition satisfies, it means negative   
 // edge weight cycle exists. Because traversing again   
 // is reducing the cost and in order to minimize the   
 // cost we can traverse till infinity and hence a proper   
 // answer can't be calculated.   
 if*(dis[u]!=Integer.***MAX\_VALUE***&&dis[u]+wt<dis[v])  
 *return new int*[0];  
 }  
  
 *return* dis; *// returning our answer vector/array.* }  
  
}

class DSU {

        public int[] parent, rank, size;

        public DSU(int n) {

            parent = new int[n];

            rank = new int[n];

            size = new int[n];

            for (int i = 0; i < n; i++) {

                parent[i] = i;

                rank[i] = 0;

                size[i] = 1;

            }

        }

        public int find(int p) {

            // path compression

            if(p == parent[p]) return p;

            return parent[p] = find(parent[p]);

        }

        public void unionByRank(int u, int v) {

            int pu = find(u);

            int pv = find(v);

            if (pu == pv)

                return;

            // union by size

            if (rank[pu] < rank[pv]) {

                parent[pu] = pv;

            } else if(rank[pv] < rank[pu]) {

                parent[pv] = pu;

            }

            else

            {

                parent[pv] = pu;

                rank[pu] += 1;

            }

        }

        public void unionBySize(int u, int v) {

            int pu = find(u);

            int pv = find(v);

            if (pu == pv)

                return;

            // union by size

            if (size[pu] < size[pv]) {

                parent[pu] = pv;

                size[pv] += size[pu];

            } else  {

                parent[pv] = pu;

                size[pu] += size[pv];

            }

        }

             }