|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*; **class** GFG {  */\*\*  \* Given an array arr[] and a number K where K is smaller  \* than size of array, the task is to find the Kth smallest  \* element in the given array. It is given that all array elements  \* are distinct.  \*/   /\*\** ***@author*** *nitish-jindal  \* we can use MinHeap, but the complexity will O(n+klogn)  \* Hence we will use QuickSelect, which is a modification of HeapSort  \* its worst complexity is O(n^2) but avg is O(n)  \*/* **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while**(t != 0) {  **int** n = scan.nextInt();  **int** [] arr = **new int**[n];   **for**(**int** i = 0; i < n; i++) {  arr[i] = scan.nextInt();  }   **int** k = scan.nextInt();   System.***out***.println(*getKthSmallestElement*(arr, 0, n-1, k));  t--;  }  }   **public static int** getKthSmallestElement(**int** []arr, **int** low, **int** high, **int** k) {  **if**(k <= arr.**length** && low <= high) {  **int** pivotPosition = *partition*(arr, low, high);   **if**(pivotPosition == k-1) {  **return** arr[pivotPosition];  }   **if**(pivotPosition > k-1) {  **return** *getKthSmallestElement*(arr, low, pivotPosition-1, k);  }   **return** *getKthSmallestElement*(arr, pivotPosition+1, high, k-pivotPosition);  }   **return** Integer.***MAX\_VALUE***;  }   **public static int** partition(**int** [] arr, **int** low, **int** high) {  **int** pivot = arr[high];  **int** i = low;  **for**(**int** j = low; j < high; j++) {  **if**(arr[j] <= pivot) {  *swap*(arr, i, j);  i++;  }  }  *swap*(arr, i, high);  **return** i;  }   **public static void** swap(**int** [] arr, **int** i, **int** j) {  **int** temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*;   */\*\*  \* Given an incomplete Sudoku configuration in terms of a 9 x 9 2-D square matrix (mat[][]).  \* The task to print a solved Sudoku. For simplicity you may assume that there will be only one unique solution.*  <https://www.geeksforgeeks.org/sudoku-backtracking-7/> *\*/* **class** GFG {  **private static final int *SIZE*** = 9;  **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while**(t != 0) {  **int** [][] arr = **new int**[***SIZE***][***SIZE***];   **for**(**int** i = 0; i < ***SIZE***; i++) {  **for**(**int** j = 0; j < ***SIZE***; j++) {  arr[i][j] = scan.nextInt();  }  }   **int**[][] grid = {  {3, 0, 6, 5, 0, 8, 4, 0, 0},  {5, 2, 0, 0, 0, 0, 0, 0, 0},  {0, 8, 7, 0, 0, 0, 0, 3, 1},  {0, 0, 3, 0, 1, 0, 0, 8, 0},  {9, 0, 0, 8, 6, 3, 0, 0, 5},  {0, 5, 0, 0, 9, 0, 6, 0, 0},  {1, 3, 0, 0, 0, 0, 2, 5, 0},  {0, 0, 0, 0, 0, 0, 0, 7, 4},  {0, 0, 5, 2, 0, 6, 3, 0, 0}  };  *solveSuDuKo*(arr);  t--;  }  }   **private static void** solveSuDuKo(**int** [][] arr) {  **if**(*findSolution*(arr, 0, 0)) {  *printArr*(arr);  }  }   **private static boolean** findSolution(**int** [][] arr, **int** row, **int** col) {   **for** (**int** i = row; i < ***SIZE***; i++, col = 0) {  **for** (**int** j = col; j < ***SIZE***; j++) {  **if** (arr[i][j] == 0) {  **for**(**int** num = 1; num <= ***SIZE***; num++) {  **if**(*isSafe*(arr, i, j, num)) {  arr[i][j] = num;   **if**(*findSolution*(arr, i, j+1)) {  **return true**;  } **else** {  arr[i][j] = 0;  }  }  }  **return false**;  }  }  }   **return true**;  }   **private static void** printArr(**int** [][] arr) {  **for**(**int** i = 0; i < ***SIZE***; i++) {  **for**(**int** j = 0; j < ***SIZE***; j++) {  System.***out***.print(arr[i][j] + **" "**);  }  System.***out***.println(**""**);;  }  }   **private static boolean** isSafe(**int**[][] board, **int** row, **int** col, **int** val) {  **for** (**int** i = 0; i < 9; i++) {  **if** (board[i][col] == val) {  **return false**;  }  **if** (board[row][i] == val) {  **return false**;  }  **if** (board[3 \* (row / 3) + i / 3][3 \* (col / 3) + i % 3] == val) {  **return false**;  }  }  **return true**;  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*;  */\*\*  \* Given a 2D array, find the maximum sum subarray in it.  \* For example, in the following 2D array, the maximum sum subarray  \* https://practice.geeksforgeeks.org/problems/maximum-sum-rectangle/0/  \* https://www.youtube.com/watch?v=yCQN096CwWM  \*/  /\*\*  \* Algorithm:  \* 1. Use kadane Algo to find max sum in a array  \* 2. pick the xth column  \* 3. add the sum to the temp array  \* 4. mind the max sum in temp array using kadane algo  \* 5. repeat the process for all the columns   \*/* **class** GFG {  **public static void** main(String[] args) {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while** (t != 0) {  **int** n = scan.nextInt();  **int** m = scan.nextInt();   **int** arr[][] = **new int**[n][m];   **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < m; j++) {  arr[i][j] = scan.nextInt();  }  }   **int** result = Integer.***MIN\_VALUE***;   **for** (**int** x = 0; x < m; x++) {  **int** temp[] = **new int**[n];  **for** (**int** i = x; i < m; i++) {  **for** (**int** j = 0; j < n; j++) {  temp[j] = temp[j] + arr[j][i];  }  result = Math.*max*(result, *kadane*(temp));  }  }   System.***out***.println(result);   t--;  }  }   **private static int** kadane(**int**[] arr) {  **int** max\_sum = 0;  **int** max\_sum\_till\_now = 0;   **for** (**int** i = 0; i < arr.**length**; i++) {  max\_sum\_till\_now = max\_sum\_till\_now + arr[i];   **if** (max\_sum\_till\_now < 0) {  max\_sum\_till\_now = 0;  }   max\_sum = Math.*max*(max\_sum, max\_sum\_till\_now);  }   **return** max\_sum;  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*;  **class** Trie {  **boolean arr**[] = **new boolean**[26];  Trie **next**;  **boolean isEOI**[] = **new boolean**[26]; } **class** GFG {  **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while**(t != 0) {  **int** n = scan.nextInt();  scan.nextLine();  String data = scan.nextLine();   Trie trie = *insert*(data);   String patt = scan.nextLine();  **for**(**int** i=0; i<patt.length(); i++) {  **if**(!trie.**arr**[patt.charAt(i) - **'a'**]) {  System.***out***.println(**"0"**);  **break**;  }  **if**(i==patt.length()-1 && trie.**isEOI**[patt.charAt(i) - **'a'**]) {  System.***out***.println(**"1"**);  } **else if**(i==patt.length()-1 && !trie.**isEOI**[patt.charAt(i) - **'a'**]) {  System.***out***.println(**"0"**);  }  trie = trie.**next**;  }  t--;  }  }   **private static** Trie insert(String s) {  String arr[] = s.split(**" "**);   Trie node = **new** Trie();  Trie start = node;  **for**(String val : arr) {  **for**(**int** i = 0; i < val.length(); i++) {  **if**(i == 0) {  node.**arr**[val.charAt(i) - **'a'**] = **true**;  } **else** {  **if**(node.**next** == **null**) {  Trie newNode = **new** Trie();  node.**next** = newNode;  }  node = node.**next**;  node.**arr**[val.charAt(i) - **'a'**] = **true**;  }  }  node.**isEOI**[val.charAt(val.length() -1) - **'a'**] = **true**;  node = start;  }   **return** start;  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*;  */\*\*  \* Given a string of S as input.  \* Your task is to write a program to remove or delete minimum number of characters  \* from the string so that the resultant string is palindrome.  \*  \* We will use DP - find the length of longestPallindromicSubseq.  \* once we have the longest palindrome length, then sizeOfString - length will be our answer  \*  \* refer - https://www.geeksforgeeks.org/minimum-number-deletions-make-string-palindrome/  \* https://www.youtube.com/watch?v=\_nCsPn7\_OgI  \*/* **class** GFG {  **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();  scan.nextLine();   **while**(t != 0) {  String s = scan.nextLine();  **int** n = s.length();   **int** arr[][] = **new int**[n][n];   **for**(**int** i = 0; i < s.length(); i++) {  arr[i][i] = 1;  }   **for**(**int** index = 2; index <= n; index++) {  **for**(**int** i = 0; i < n-index+1; i++) {  **int** j = i + index - 1;   **if**(s.charAt(i) == s.charAt(j)) {  arr[i][j] = arr[i+1][j-1] + 2;  } **else** {  arr[i][j] = Math.*max*(arr[i+1][j], arr[i][j-1]);  }  }  }   System.***out***.println(n-arr[0][n-1]);   t--;  }  } } |

|  |
| --- |
| **import** java.security.KeyPair; **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*;   */\*\*  \* https://www.geeksforgeeks.org/find-the-point-where-maximum-intervals-overlap/  \*  \* Consider a big party where a log register for guest’s entry and exit times is maintained.  \* Find the time at which there are maximum guests in the party. Note that entries in register are not in any order.  \*/* **class** GFG {  **public static void** main(String[] args) {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while** (t != 0) {  **int** n = scan.nextInt();  **int** entry[] = **new int**[n];  **int** leave[] = **new int**[n];   **for**(**int** i = 0; i < n; i++) {  entry[i] = scan.nextInt();  }  **for**(**int** i = 0; i < n; i++) {  leave[i] = scan.nextInt();  }   Arrays.*sort*(entry);  Arrays.*sort*(leave);   **int** maxNum = 0;  **int** maxTime = 0;  **int** result = 0;   **int** i=0; **int** j=0;  **while**(i<n && j<n) {  **if**(entry[i] <= leave[j]) {  maxNum++;  **if**(maxNum >= result) {  result = maxNum;  maxTime = entry[i];  }  i++;  } **else** {  maxNum--;  j++;  }  }   **while** (i < n) {  maxNum++;  **if**(maxNum >= result) {  result = maxNum;  maxTime = entry[i];  }  i++;  }   System.***out***.println(result + **" "** + maxTime);  t--;  }  } |

|  |
| --- |
| */\*\*  \* Leaves to DLL   \* Given a Binary Tree of size N, extract all its leaf nodes  \* to form a Doubly Link List strating from the left most leaf.   \* Modify the original tree to make the DLL thus removing the leaf   \* nodes from the tree. Consider the left and right pointers of   \* the tree to be the previous and next pointer of the DLL respectively.  \*/* **class** Tree{  Node **head** = **null**;  Node **prev** = **null**;  Node **current** = **null**;  **public** Node convertToDLL(Node root)  {  getDLL(root);  **return head**;  }  **public** Node getDLL(Node root) {  **if**(root == **null** || (root.left == **null** && root.right == **null**)) {  **return** root;  }   **if**(root.left != **null** && root.left.left == **null** && root.left.right == **null**) {  **if**(**head** == **null**) {  **head** = root.left;  **prev** = root.left;  } **else** {  **prev**.right = root.left;  root.left.left = **prev**;  **prev** = root.left;  }  root.left = **null**;  }   root.left = getDLL(root.left);   **if**(root.right != **null** && root.right.left == **null** && root.right.right == **null**) {  **if**(**head** == **null**) {  **head** = root.right;  **prev** = root.right;  } **else** {  **prev**.right = root.right;  root.right.left = **prev**;  **prev** = root.right;  }  root.right = **null**;  }   root.right = getDLL(root.right);   **return** root;  } } |

|  |
| --- |
| */\*\*  Design a data-structure SpecialStack that supports all the stack operations   like push(), pop(), isEmpty(), isFull() and an additional operation getMin()   which should return minimum element from the SpecialStack.  Your task is to complete all the functions, using stack data-Structure. \*/* **class** GfG {  **public static** Integer *min* = Integer.***MAX\_VALUE***;   **public void** push(**int** a, Stack<Integer> s) {  **if** (isEmpty(s)) {  *min* = a;  s.push(a);  **return**;  }  **if** (*min* > a) {  s.push(2 \* a - *min*);  *min* = a;  } **else** {  s.push(a);  }  }   **public int** pop(Stack<Integer> s) {  **if** (isEmpty(s)) {  **return** -1;  } **else** {  **int** val = s.pop();  **if** (val < *min*) {  **int** a = *min*;  *min* = 2 \* *min* - val;  **return** a;  }  **return** val;  }  }   **public int** min(Stack<Integer> s) {  **if** (isEmpty(s)) {  **return** -1;  }  **return** *min*;  }   **public boolean** isFull(Stack<Integer> s, **int** n) {  **if** (isEmpty(s) && n > 0) {  **return false**;  }  **return** s.size() == n;  }   **public boolean** isEmpty(Stack<Integer> s) {  **return** s == **null** || s.size() == 0;  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*;  */\*\*  \* Given a string S with repeated characters (only lowercase).  \* The task is to rearrange characters in a string such that no  \* two adjacent characters are same.  \*  \* Input: aaabc  \* Output: abaca  \*  \* Input: aaabb  \* Output: ababa  \*  \*/  /\*\*  \* 1. Use priority queue/ maximum heap to store the elements accoring to freq such that hightest freq element should be root  \* 2. pop the root  \* 2. the the character to result  \* 3. decrease the freq  \* 4. push the prev element to queue  \* 5. assign prev to current root element  \*  \* https://www.geeksforgeeks.org/rearrange-characters-string-no-two-adjacent/  \*   \*/* **class** Pair {  **char key**;  **int value**;   **public** Pair(**char** key, **int** value) {  **this**.**key** = key;  **this**.**value** = value;  }   **public static** Comparator<Pair> *comparator* = **new** Comparator<Pair>() {  @Override  **public int** compare(Pair o1, Pair o2) {  **return** o2.**value** - o1.**value**;  }  }; } **class** GFG {    **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();  scan.nextLine();   **while**(t != 0) {  PriorityQueue<Pair> queue = **new** PriorityQueue<>(Pair.*comparator*);   String s = scan.nextLine();  Map<Character, Integer> map = **new** HashMap<>();   **for**(**int** i = 0; i < s.length(); i++) {  **if**(!map.containsKey(s.charAt(i))) {  map.put(s.charAt(i), 1);  } **else** {  map.put(s.charAt(i), map.get(s.charAt(i)) + 1);  }  }   map.forEach((key, value) -> {  queue.add(**new** Pair(key, value));  });   String res = **""**;  Pair prev = **null**;  **while**(!queue.isEmpty()) {  Pair current = queue.poll();   res += current.**key**;  current.**value** --;   **if**(prev != **null**) {  queue.add(prev);  }   prev = (current.**value** > 0) ? current : **null**;  }   **if**(res.length() == s.length()) {  System.***out***.println(1);  } **else** {  System.***out***.println(0);  }    t--;  }  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*;  */\*\*  \* Given inorder and postorder traversals of a Binary Tree in the arrays in[] and post[] respectively.  \* The task is to construct the binary tree from these traversals.  \* For example, if given inorder and postorder traversals are  \* In = {4, 8, 2, 5, 1, 6, 3, 7}  \* Post = {8, 4, 5, 2, 6, 7, 3, 1}  \* then your function should construct below tree.  \* pre = {1,2,4,8,5,3,6,7}  \*/* **class** Node {  **int data**;  Node **left**;  Node **right**;   **public** Node(**int** data) {  **this**.**data** = data;  **this**.**left** = **null**;  **this**.**right** = **null**;  } } **class** GfG {  **int size**;  Node buildTree(**int** in[], **int** post[], **int** n) {  **if**(n==0) {  **return null**;  }  **size** = n-1;   **return** getTree(in, post);  }   Node getTree(**int** in[], **int** post[]) {  **if**(**size** < 0 || in == **null** || in.**length** == 0) {  **size** ++; *// restore the size, as current element is not used yet* **return null**;  }  **if**(in.**length** == 1) {  **return new** Node(in[0]);  }   **int** lastEle = post[**size**];  **int**[] left = getLeftArray(in, lastEle);  **int** [] right = getRightArray(in, lastEle);   **size** --;   Node root = **new** Node(lastEle);  root.**right** = getTree(right, post);   **size**--;   root.**left** = getTree(left, post);   **return** root;  }   **public int**[] getLeftArray(**int** [] in, **int** ele) {  **int** pos = getPos(in, ele);  **if**(pos == -1) {  **return null**;  }  **int** [] arr = **new int**[pos];   **for**(**int** i = 0; i < pos; i++) {  arr[i] = in[i];  }  **return** arr;  }   **public int**[] getRightArray(**int** [] in, **int** ele) {  **int** pos = getPos(in, ele);  **if**(pos == -1) {  **return null**;  }  **int** [] arr = **new int**[in.**length** - 1 - pos];  **int** k = 0;  **for**(**int** i = pos+1; i < in.**length**; i++) {  arr[k++] = in[i];  }  **return** arr;  }   **public int** getPos(**int** [] arr, **int** ele) {  **for**(**int** i=0; i < arr.**length**; i++) {  **if**(ele == arr[i]) {  **return** i;  }  }  **return** -1;  } }  **better Approach**  **int search(int arr[],int start,int end,int data)**  **{**  **for(int i=start;i<=end;i++)**  **{**  **if(data==arr[i]) return i;**  **}**  **}**  **Node\* buildTreeUntil(int in[],int post[], int inStart, int inEnd, int \*postIndex)**  **{**  **if(inStart>inEnd) return NULL;**  **Node\* node=newNode(post[\*postIndex]);**  **(\*postIndex)--;**  **if(inStart==inEnd) return node;**  **int inIndex=search(in,inStart,inEnd,node->data);**  **//cout<<inIndex;**  **node->right=buildTreeUntil(in,post,inIndex+1,inEnd,postIndex);**  **node->left=buildTreeUntil(in,post,inStart,inIndex-1,postIndex);**  **return node;**  **}**  **Node \*buildTree(int in[], int post[], int n)**  **{**  **int pindex=n-1;**  **Node \*node=buildTreeUntil(in,post,0,n-1,&pindex);**  **return node;**  **}** |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*;  */\*\*  \* Find the largest rectangular area possible in a given histogram where the largest rectangle  \* can be made of a number of contiguous bars.  \* For simplicity, assume that all bars have same width and the width is 1 unit.  \* https://www.geeksforgeeks.org/largest-rectangle-under-histogram/  \** [*https://www.youtube.com/watch?v=ZmnqCZp9bBs*](https://www.youtube.com/watch?v=ZmnqCZp9bBs)   1. *If the element is greater or equal to top of stack then push into stack.* 2. *If Smaller then keep popping out from the stack until input[stackTop] is less or equal to currentElemment* 3. *Meanwhile perform -> {* 4. *Index = pop from stack* 5. *CurrentArea = input[Index] \* (isStackEmpty()) ? i : (i – stack.currentop() – 1);* 6. *Compare currentArea with maxArea*   *}*  *\*/* **class** GFG {  **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while**(t != 0) {  Stack<Integer> stack = **new** Stack<>();   **int** n = scan.nextInt();  scan.nextLine();  String [] arr = scan.nextLine().split(**" "**);   **int** current;  **int** max = -1;  **int** i = 1;  stack.push(0);   **while**(!stack.empty()) {  **if**(i < n) {  **while**(!stack.empty() &&  Integer.*parseInt*(arr[i]) < Integer.*parseInt*(arr[stack.peek()])) {  **int** index = stack.pop();  current = Integer.*parseInt*(arr[index]) \* (stack.empty() ? i : (i - stack.peek() -1));   **if**(current > max) {  max = current;  }  }  stack.push(i);  i++;  } **else** {  **int** index = stack.pop();  current = Integer.*parseInt*(arr[index]) \* (stack.empty() ? i : (i - stack.peek() -1));   **if**(current > max) {  max = current;  }  }  }   System.***out***.println(max);  t--;  }  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*;  */\*\*  \* https://www.geeksforgeeks.org/optimal-strategy-for-a-game-dp-31/  \*  \* You are given an array A of size N.  \* The array contains integers and is of even length.  \* The elements of the array represent N coin of values V1, V2, ....Vn. You play against an opponent in an alternating way.  \* In each turn,  \* a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin.  \* You need to determine the maximum possible amouint of money you can win if you go first.  \*   \* 2  \* 4  \* 5 3 7 10  \* 4  \* 8 15 3 7  \* Output:  \* 15  \* 22  \*  \* Explanation:  \* Testcase1: The user collects maximum value as 15(10 + 5)  \* Testcase2: The user collects maximum value as 22(7 + 15)  \*/* **class** GFG {  **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while**(t != 0) {  **int** n = scan.nextInt();  scan.nextLine();  String arr[] = scan.nextLine().split(**" "**);  System.***out***.println(*optimalStrategy*(arr));  t--;  }  }   **private static int** optimalStrategy(String arr[]) {  **int** n = arr.**length**;  **int** dp[][] = **new int**[n][n];   **for**(**int** gap = 0; gap < n; gap++) {  **int** i = 0;  **int** j = gap;  **while**(j < n) {  **int** x = (i+2 <= j) ? dp[i+2][j] : 0;  **int** y = (i+1 <= j-1) ? dp[i+1][j-1] : 0;  **int** z = (i <= j-2) ? dp[i][j-2] : 0;   dp[i][j] = Math.*max*(Integer.*valueOf*(arr[i]) + Math.*min*(x,y),  Integer.*valueOf*(arr[j]) + Math.*min*(y,z));   i++;  j++;  }  }   **return** dp[0][n-1];  } } |

|  |
| --- |
| */\*\*  \* Given a graph with N nodes and M directed edges. Your task is to complete the function kosaraju()  \* which returns an integer denoting the number of strongly connected components in the graph.  \*  \* https://www.youtube.com/watch?v=RpgcYiky7uw  \* https://github.com/mission-peace/interview/blob/master/src/com/interview/graph/StronglyConnectedComponent.java  \*/* / **class** Solution {  Stack<Integer> **stack** = **new** Stack<>();  **int result** = 0;  **public int** kosaraju(ArrayList<ArrayList<Integer>> adj, **int** N)  {  **boolean** visited[] = **new boolean**[N];  **for**(**int** i = 0; i < N; i++) {  **if**(!visited[i]) {  visited[i] = **true**;  doDFS(adj, i, visited);  }  }   adj = reverseGraph(adj);   **boolean** visitedV2[] = **new boolean**[N];  **while**(!**stack**.empty()) {  **int** root = **stack**.pop();  **if**(!visitedV2[root]) {  visitedV2[root] = **true**;  doDFS(adj, root, visitedV2);  **result**++;  }  }   **return result**;  }   **private void** doDFS(ArrayList<ArrayList<Integer>> adj, **int** i, **boolean** visited[]) {  **if**(adj.get(i) != **null**) {  adj.get(i).forEach((child) -> {  **if** (!visited[child]) {  visited[child] = **true**;  doDFS(adj, child, visited);  }  });  }  **stack**.push(i);  }   **private** ArrayList<ArrayList<Integer>> reverseGraph(ArrayList<ArrayList<Integer>> adj) {  ArrayList<ArrayList<Integer>> reverseGraph = **new** ArrayList<>();  **for**(**int** i = 0; i < adj.size(); i++) {  reverseGraph.add(**new** ArrayList<>());  }  **for**(**int** i = 0; i < adj.size(); i++) {  **final int** node = i;  **if**(adj.get(i) != **null**) {  adj.get(i).forEach(child -> {  reverseGraph.get(child).add(node);  });  }  }  **return** reverseGraph;  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*;  */\*\*  \* Given a weighted, undirected and connected graph.  \* The task is to find the sum of weights of the edges of the Minimum Spanning Tree.  \*  \* https://www.youtube.com/watch?v=fAuF0EuZVCk - minimum spanning tree  \* https://www.youtube.com/watch?v=ID00PMy0-vE - disjoint sets   \*   \*/* **class** Node {  **int data**;  **int rank**;  Node **parent**;   **public** Node(**int** data) {  **this**.**data** = data;  **rank** = 0;  **parent** = **this**;  } }  **class** DisjointSet {   **private** Map<Integer, Node> **map** = **new** HashMap<>();   **public void** makeSet(**int** data) {  Node node = **new** Node(data);  **map**.put(data, node);  }   **public** Node findSet(Node child) {  **if**(child.**data** == child.**parent**.**data**) {  **return** child;  } **else** {  child.**parent** = findSet(child.**parent**);  }  **return** child.**parent**;  }   **public void** union(**int** data1, **int** data2) {  Node parent1 = findSet(**map**.get(data1));  Node parent2 = findSet(**map**.get(data2));   **if**(parent1 == parent2) {  **return**;  }  **if**(parent1.**rank** >= parent2.**rank**) {  parent1.**rank** = parent1.**rank** == parent2.**rank** ? parent1.**rank** + 1 : parent1.**rank**;  parent2.**parent** = parent1;  } **else** {  parent1.**parent** = parent2;  }  }   Map<Integer, Node> getMap() {  **return this**.**map**;  } }  **class** Edge {  **int vertex1**;  **int vertex2**;  **int weight**;   **public** Edge(**int** vertex1, **int** vertex2, **int** weight) {  **this**.**vertex1** = vertex1;  **this**.**vertex2** = vertex2;  **this**.**weight** = weight;  }   **public static** Comparator *comparator* = **new** Comparator<Edge>() {  @Override  **public int** compare(Edge o1, Edge o2) {  **return** o1.**weight** - o2.**weight**;  }  };   **public** String toString() {  **return this**.**vertex1** + **" "** + **this**.**vertex2** + **" "** + **this**.**weight**;  } } **class** MST {   **public static void** main(String args[]) **throws** IOException {   BufferedReader read =  **new** BufferedReader(**new** InputStreamReader(System.***in***));  **int** t = Integer.*parseInt*(read.readLine());  **while** (t-- > 0) {  String str[] = read.readLine().trim().split(**" "**);  **int** V = Integer.*parseInt*(str[0]);  **int** E = Integer.*parseInt*(str[1]);   ArrayList<ArrayList<Integer>> graph = **new** ArrayList<>();  **for** (**int** i = 0; i < V; i++) {  ArrayList<Integer> temp = **new** ArrayList<>();  **for** (**int** j = 0; j < V; j++) temp.add(Integer.***MAX\_VALUE***);  graph.add(temp);  }  str = read.readLine().trim().split(**" "**);  **int** k = 0;  **int** i=0;  **while** (i++<E) {  **int** u = Integer.*parseInt*(str[k++]);  **int** v = Integer.*parseInt*(str[k++]);  **int** w = Integer.*parseInt*(str[k++]);  u--;  v--;  graph.get(u).set(v, w);  graph.get(v).set(u, w);  }   System.***out***.println(*spanningTree*(V, E, graph));  }  }   **static int** spanningTree(**int** V, **int** E, ArrayList<ArrayList<Integer>> graph) {  Object [] edgeList = *getAllEdgesSortedByWeight*(V, graph);   **int** minimumSpanningWeight = 0;   DisjointSet disjointSet = **new** DisjointSet();  **for**(**int** i = 0; i < V; i++) {  disjointSet.makeSet(i);  }   **for**(Object edge: edgeList) {  Edge currentEdge = (Edge) edge;   **if**(disjointSet.findSet(disjointSet.getMap().get(currentEdge.**vertex1**)) !=  disjointSet.findSet(disjointSet.getMap().get(currentEdge.**vertex2**))) {  disjointSet.union(currentEdge.**vertex1**, currentEdge.**vertex2**);  minimumSpanningWeight += currentEdge.**weight**;  }  }   **return** minimumSpanningWeight;  }   **public static** Object[] getAllEdgesSortedByWeight(**int** V, ArrayList<ArrayList<Integer>> graph) {  ArrayList<Edge> edgeList = **new** ArrayList<>();  **int** i = 0;  **for**(ArrayList<Integer> list : graph) {  **if**(i >= V) {  **break**;  }  **int** j = 0;  **for**(Integer weight : list) {  **if**(weight != Integer.***MAX\_VALUE***) {  edgeList.add(**new** Edge(i, j, weight));  }  j++;  }  i++;  }  Object[] array = edgeList.toArray();  Arrays.*sort*(array, Edge.*comparator*);  **return** array;  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*;  */\*\*  \* Implementing Dijkstra  \* Given a graph of V nodes represented in the form of the adjacency matrix.  \* The task is to find the shortest distance of all the vertex's from the source vertex.  \*  \** [*https://www.youtube.com/watch?v=lAXZGERcDf4*](https://www.youtube.com/watch?v=lAXZGERcDf4)  *Note – to get the path, we can store chile, parent map along with weight map in Solution class   \*/* **class** Vertex {  **int vertex**;  **int distance**;   Vertex(**int** vertex, **int** distance) {  **this**.**vertex** = vertex;  **this**.**distance** = distance;  }   **static** Comparator *comparator* = **new** Comparator<Vertex>() {  @Override  **public int** compare(Vertex o1, Vertex o2) {  **return** o1.**distance** - o2.**distance**;  };  }; }  **class** HeapMap {  **private** PriorityQueue<Vertex> **queue**;  **private** Map<Integer, Vertex> **map**;   HeapMap() {  **this**.**queue** = **new** PriorityQueue<>(Vertex.*comparator*);  **this**.**map** = **new** HashMap<>();  }   **void** add(Vertex v) {  **queue**.add(v);  **map**.put(v.**vertex**, v);  }   **boolean** contains(**int** v) {  **return map**.get(v) != **null**;  }   Vertex getMinimum() {  Vertex vertex = **queue**.poll();  **if**(vertex != **null**) {  **map**.remove(vertex.**vertex**);  }  **return** vertex;  }   **void** decrease(**int** key, **int** weight) {  Vertex current = getCurrentVertex(key);   **queue**.remove(current);   current.**distance** = weight;   **queue**.add(current);  }   Vertex getCurrentVertex(**int** key) {  **return map**.get(key);  }   **boolean** hasElement() {  **return** !**queue**.isEmpty();  } } **class** Solution {  **static int**[] dijkstra(ArrayList<ArrayList<Integer>> g, **int** src, **int** V)  {  HeapMap minHeap = **new** HeapMap();  Map<Integer, Integer> finalDistance = **new** HashMap<>();   *createMinHeap*(minHeap, src, V);   **while**(minHeap.hasElement()) {  Vertex vertex = minHeap.getMinimum();  finalDistance.put(vertex.**vertex**, vertex.**distance**);   **int** currentDistance = vertex.**distance**;  ArrayList<Integer> edges = g.get(vertex.**vertex**);  **int** i = 0;  **for**(**int** edgeWeight: edges) {  **if**(edgeWeight != 0 && minHeap.contains(i)) {  **if**(minHeap.getCurrentVertex(i).**distance** > edgeWeight + currentDistance) {  minHeap.decrease(i, edgeWeight+currentDistance);  }  }  i++;  }  }   **return** *createResultArray*(finalDistance, V);  }   **private static void** createMinHeap(HeapMap minHeap, **int** src, **int** V) {  **for**(**int** i = 0; i < V; i++) {  **if**(i == src) {  minHeap.add(**new** Vertex(src, 0));  } **else** {  minHeap.add(**new** Vertex(i, Integer.***MAX\_VALUE***));  }  }  }   **private static int**[] createResultArray(Map<Integer, Integer> finalDistance, **int** V) {  **int**[] resultDistnace = **new int**[V];  **for**(**int** i=0; i<V; i++) {  resultDistnace[i] = finalDistance.get(i);  }  **return** resultDistnace;  } } **class** Gfg {  **static void** printSolution(**int** dist[], **int** n)  {  **for**(**int** i = 0; i < n; i++)  System.***out***.print(dist[i] + **" "**);  }  **public static void** main(String args[])  {  Scanner sc = **new** Scanner(System.***in***);  **int** t = Integer.*parseInt*(sc.next());   **while**(t > 0)  {  **int** V = Integer.*parseInt*(sc.next());;  ArrayList<ArrayList<Integer>> list = **new** ArrayList<>(V);  **for**(**int** i = 0;i < V; i++)  {  ArrayList<Integer> temp = **new** ArrayList<>(V);  list.add(i, temp);  }   **for**(**int** i = 0; i < V; i++)  {  ArrayList<Integer> temp = list.get(i);  **for**(**int** j = 0; j < V; j++)  {  temp.add(Integer.*parseInt*(sc.next()));  }  list.add(temp);  }  **int** s = Integer.*parseInt*(sc.next());;  Solution T = **new** Solution();  **int**[] res = T.*dijkstra*(list, s, V);  *printSolution* (res, V);  System.***out***.println();  t--;  }  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*;  */\*\*  \* GRAPH ALGO - TOPOLOGICAL SORT  \* https://www.youtube.com/watch?v=ddTC4Zovtbc  \*  \* Given a sorted dictionary of an alien language  \* having N words and k starting alphabets of standard dictionary.  \* Find the order of characters in the alien language.  \*  \* 2  \* 5 4  \* baa abcd abca cab cad  \* 3 3  \* caa aaa aab  \*  \* Output  \* bdac  \* cab  \*  \* Algo -   \*   \* 1) Create a graph g with number of vertices equal to the size of alphabet in the given alien language.   \* For example, if the alphabet size is 5, then there can be 5 characters in words. Initially there are no edges in graph.  \*  \* 2) Do following for every pair of adjacent words in given sorted array.  \* .....a) Let the current pair of words be word1 and word2. One by one compare   \* characters of both words and find the first mismatching characters.  \* .....b) Create an edge in g from mismatching character of word1 to that of word2.  \*  \* 3) Print topological sorting of the above created graph.  \*/* **class** Pair<T> {  T **first**;  T **second**;   Pair(T first, T second) {  **this**.**first** = first;  **this**.**second** = second;  } }  **class** Graph {  **private** Map<Character, ArrayList<Character>> **adjList** = **new** HashMap<>();  **private** Map<Character, Boolean> **visited** = **new** HashMap<>();  Stack<Character> **topologicalResult** = **new** Stack<>();   **public void** createGraph(String[] dict, **int** count) {  initListAndVisited(dict, count);   **for**(**int** i = 0; i < dict.**length**-1; i++) {  Pair<Character> firstDifferentCharacter = getfirstDifferentCharacter(dict[i], dict[i+1]);  **if**(firstDifferentCharacter != **null**) {  **adjList**.get(firstDifferentCharacter.**first**).add(firstDifferentCharacter.**second**);  }  }  }   **private void** initListAndVisited(String[] dict, **int** count) {  **for**(String string: dict) {  **if**(**adjList**.size() == count) {  **break**;  }  **for**(Character c: string.toCharArray()) {  **if**(!**adjList**.containsKey(c)) {  **adjList**.put(c, **new** ArrayList<>());  **visited**.put(c, **false**);  }  }  }  }   **private** Pair<Character> getfirstDifferentCharacter(String s1, String s2) {  **int** i = 0;  **int** j = 0;  **while**(i < s1.length() && j < s2.length()) {  **if**(s1.charAt(i) != s2.charAt(j)) {  **return new** Pair<>(s1.charAt(i), s2.charAt(j));  }  i++;  j++;  }  **return null**;  }   **public void** topologicalSort() {  **adjList**.forEach((vertex, edges)-> {  **if**(!**visited**.get(vertex)) {  **visited**.put(vertex, **true**);  topologicalSortUtil(vertex);  }  });   }   **private void** topologicalSortUtil(Character vertex) {  **adjList**.get(vertex).forEach(child -> {  **if**(!**visited**.get(child)) {  **visited**.put(child, **true**);  topologicalSortUtil(child);  }  });  **topologicalResult**.push(vertex);  } }  **class** Solution {  **public** String findOrder(String [] dict, **int** N, **int** K)  {  Graph graph = **new** Graph();  graph.createGraph(dict, K);  graph.topologicalSort();   String result = **""**;  **while**(!graph.**topologicalResult**.empty()) {  result += graph.**topologicalResult**.pop();  }   **return** result;  } }  **class** GFG {  **public static void** main(String [] arg) {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while**(t != 0) {  **int** n = scan.nextInt();  **int** k = scan.nextInt();  scan.nextLine();  String arr[] = scan.nextLine().split(**" "**);  System.***out***.println(**new** Solution().findOrder(arr, n, k));  t--;  }  } } |

|  |
| --- |
| **// Important and Unique**  **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*;  */\*\*  \* Word break problem  \*   \* Given an input string and a dictionary of words,   \* find out if the input string can be segmented into a space-separated   \* sequence of dictionary words. See following examples for more details.  \*  \* Consider the following dictionary  \* { i, like, sam, sung, samsung, mobile, ice,  \* cream, icecream, man, go, mango}  \*  \* Input: ilike  \* Output: Yes  \*   \*   \* v.v.v Imp and unique  \* https://www.youtube.com/watch?v=WepWFGxiwRs - must watch  \*  \* solve - https://leetcode.com/problems/word-break-ii/  \*/* **class** GFG {  **public static void** main(String[] args) {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while** (t != 0) {  **int** n = scan.nextInt();  scan.nextLine();  String[] buffer = scan.nextLine().split(**" "**);   Map<String, Boolean> dictionary = **new** HashMap<>();  **for** (String s : buffer) {  dictionary.put(s, **true**);  }  String query = scan.nextLine();   **boolean** dp[][] = **new boolean**[query.length()][query.length()];   **for** (**int** i = 0; i < query.length(); i++) {  **int** start = 0;  **int** end = start + i;   **while** (end < query.length()) {  **if** (dictionary.get(query.substring(start, end + 1)) != **null**) {  dp[start][end] = **true**;  } **else** {  **for** (**int** index = start; index < end; index++) {  **if** (dp[start][index] && dp[index + 1][end]) {  dp[start][end] = **true**;  }  }  }  start++;  end++;  }  }   **int** res = dp[0][query.length() - 1] ? 1 : 0;  System.***out***.println(res);   t--;  }  } } |

|  |
| --- |
| **// DP – Longest Palindromic substring in a string**  **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*; **class** GFG {  **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);   **int** t = scan.nextInt();  scan.nextLine();   **while**(t != 0) {  String s = scan.nextLine();  **int** len = s.length();   **boolean** [][] arr = **new boolean** [len][len];   *// set all the length 1 substr as pallindromic* **for**(**int** i = 0; i < len; i++) {  arr[i][i] = **true**;  }   *// set all the length 2 substr as pallindromic* **for**(**int** i = 0; i < len-1; i++) {  **if**(s.charAt(i) == s.charAt(i+1)) {  arr[i][i+1] = **true**;  }  }   *// check for all the remaining in diagonal array (watch video for reference)* **for**(**int** i = len-1; i >= 0; i--) {  **for**(**int** j = i+1; j<len; j++) {  **if**(s.charAt(i) == s.charAt(j) && arr[i+1][j-1]) {  arr[i][j] = **true**;  }  }  }   **int** maxI = -12;  **int** maxJ = -12576;   **for**(**int** i = 0; i < len; i++) {  **for**(**int** j = 0; j < len; j++) {  **if**(maxJ - maxI + 1 < j - i + 1 && arr[i][j]) {  maxI = i;  maxJ = j;  }  }  }   System.***out***.println(s.substring(maxI, maxJ+1));   t--;  }  } } |

|  |
| --- |
| **// DP – Count Number of Palindromic substring in a string excluding single alphabets of string**  **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*;  */\*\*  \* Given a string, the task is to count all palindromic sub-strings present in it.  \*  \* Example:  \*  \* Input  \*  \* 2  \* 5  \* abaab  \* 7  \* abbaeae  \*  \* Output  \*  \* 3  \* 4  \*/* **class** GFG {  **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);   **int** t = scan.nextInt();   **while**(t != 0) {  **int** len = scan.nextInt();  scan.nextLine();  String s = scan.nextLine();   **boolean** [][] arr = **new boolean** [len][len];   *// set all the length 1 substr as pallindromic* **for**(**int** i = 0; i < len; i++) {  arr[i][i] = **true**;  }   *// set all the length 2 substr as pallindromic* **for**(**int** i = 0; i < len-1; i++) {  **if**(s.charAt(i) == s.charAt(i+1)) {  arr[i][i+1] = **true**;  }  }   *// check for all the remaining in diagonal array (watch video for reference)* **for**(**int** i = len-1; i >= 0; i--) {  **for**(**int** j = i+1; j<len; j++) {  **if**(s.charAt(i) == s.charAt(j) && arr[i+1][j-1]) {  arr[i][j] = **true**;  }  }  }   **int** PallindromicSubStr = 0;   **for**(**int** i = 0; i < len; i++) {  **for**(**int** j = 0; j < len; j++) {  **if**(arr[i][j]) {  PallindromicSubStr++;  }  }  }   System.***out***.println(PallindromicSubStr - len);   t--;  }  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*; */\*\*  \* Given an infix expression in the form of a string str. Convert this infix expression to postfix expression.  \*   \* Input:  \* 2  \* a+b\*(c^d-e)^(f+g\*h)-i  \* A\*(B+C)/D  \*  \* Output:  \* abcd^e-fgh\*+^\*+i-  \* ABC+\*D/  \*  \* ALGORITHM  \* 1. if alphabet - add to result  \* 2. if ( - add to stack  \* 3. if ) - pop stack till corrosponding ( is popped  \* 4. else if +,-,\*,/ -  \* 4.1 while top of stack is of higher or equal precedence, keep popping and adding to result  \* 4.2 at end add current character to stack  \* 5. when loop break add all the remaining characters of stack to results.  \*/* / **class** GFG {  **private static** Map<Character, Integer> *precedence* = **new** HashMap<>() {{  put(**'+'**, 1);  put(**'-'**, 1);  put(**'\*'**, 2);  put(**'/'**, 2);  put(**'^'**, 3);  put(**'('**, Integer.***MIN\_VALUE***);  put(**')'**, Integer.***MIN\_VALUE***);  }};   **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();  scan.nextLine();   **while**(t != 0) {  String infix = scan.nextLine();  String postfix = **""**;  Stack<Character> stack = **new** Stack<>();  **for**(Character c : infix.toCharArray()) {  **if**(*isAlphabet*(c)) {  postfix += c;  } **else** {  **if**(c == **'('**) {  stack.push(c);  } **else** {  **if**(c == **')'**) {  **while** (!stack.empty()) {  **if**(stack.peek() == **'('**) {  stack.pop();  **break**;  } **else** {  postfix += stack.pop();  }  }  } **else** {  **while** (!stack.empty() &&  *precedence*.get(stack.peek()) >= *precedence*.get(c)) {  postfix += stack.pop();  }  stack.push(c);  }  }  }  }  **while** (!stack.empty()) {  postfix += stack.pop();  }   System.***out***.println(postfix);  t--;  }  }   **private static boolean** isAlphabet(Character c) {  **return** *precedence*.get(c) == **null**;  } } |

|  |
| --- |
| **// DP – Box Stacking**  **import** java.util.ArrayList; **import** java.util.Collections; **import** java.util.Comparator; **import** java.util.Scanner;  */\*\*  \* Box Stacking  \*  \* You are given a set of N types of rectangular 3-D boxes,  \* where the ith box has height h, width w and length l.  \* You task is to create a stack of boxes which is as tall as possible,  \* but you can only stack a box on top of another box if the dimensions  \* of the 2-D base of the lower box are each strictly larger than those of  \* the 2-D base of the higher box. Of course, you can rotate a box so that any  \* side functions as its base.It is also allowable to use multiple instances of  \* the same type of box. You task is to complete the function maxHeight which returns  \* the height of the highest possible stack so formed.  \*  \* Input:  \* n = 4  \* height[] = {4,1,4,10}  \* width[] = {6,2,5,12}  \* length[] = {7,3,6,32}  \* Output: 60  \* Explanation: One way of placing the boxes is  \* as follows in the bottom to top manner:  \* (Denoting the boxes in (l, w, h) manner)  \* (12, 32, 10) (10, 12, 32) (6, 7, 4) (5, 6, 4)  \* (4, 5, 6) (2, 3, 1) (1, 2, 3)  \* Hence, the total height of this stack is  \* 10 + 32 + 4 + 4 + 6 + 1 + 3 = 60.  \* No other combination of boxes produces a height  \* greater than this.  \*  \* https://www.youtube.com/watch?v=9mod\_xRB-O0  \*  \* we will use same algo of longest increasing sub sequence  \*  \* 1. get all the eligible instances of boxes, such that length >= width  \* 2. sort this list according to base area  \* 3. apply longest increasing sub sequence on Height of boxes  \* 4. store a resul array to backtrack the boxes used  \*/* **class** BoxAndBaseArea {  **int length**;  **int width**;  **int height**;  **private int baseArea**;   BoxAndBaseArea(**int** length, **int** width, **int** height) {  **this**.**length** = length;  **this**.**width** = width;  **this**.**height** = height;  **this**.**baseArea** = length \* width;  }   @Override  **public** String toString() {  **return "Length = "** + **this**.**length** +  **" Width = "** + **this**.**width** +  **" Height = "** + **this**.**height**;  }   **static** Comparator *comparator* = **new** Comparator<BoxAndBaseArea>() {  @Override  **public int** compare(BoxAndBaseArea o1, BoxAndBaseArea o2) {  **return** o2.**baseArea** - o1.**baseArea**;  }  }; }  **class** Point3D {  **int x**;  **int y**;  **int z**;   **public** Point3D(**int** x, **int** y, **int** z) {  **this**.**x** = x;  **this**.**y** = y;  **this**.**z** = z;  } }  **class** Geeks {  **private static** ArrayList<BoxAndBaseArea> *boxList*;  **public static int** maxHeight(**int** height[], **int** width[], **int** length[], **int** n)  {  *boxList* = **new** ArrayList<>();  **for**(**int** i = 0; i < n; i++) {  *addBoxToArray*(height[i], width[i], length[i]);  }  Collections.*sort*(*boxList*, BoxAndBaseArea.*comparator*);   **int** [] maxHeight = **new int** [*boxList*.size()];  **for**(**int** i = 0; i < *boxList*.size(); i++) {  maxHeight[i] = *boxList*.get(i).**height**;  }   **int** [] result = **new int** [*boxList*.size()];  **for**(**int** i = 0; i < *boxList*.size(); i++) {  result[i] = i;  }  **for**(**int** i = 1; i < *boxList*.size(); i++) {  **for**(**int** j = 0; j < i; j++) {  **if**(*boxList*.get(i).**length** < *boxList*.get(j).**length** &&  *boxList*.get(i).**width** < *boxList*.get(j).**width** &&  maxHeight[i] < maxHeight[j] + *boxList*.get(i).**height**) {  maxHeight[i] = maxHeight[j] + *boxList*.get(i).**height**;  result [i] = j;  }  }  }   **int** index = *maxElemnet*(maxHeight);  **int** maxEle = maxHeight[index];   **while**(**true**) {  System.***out***.println(*boxList*.get(index).toString());   **if**(result[index] == index) {  **break**;  }   index = result[index];  }   *boxList* = **null**;  **return** maxEle;  }   **private static void** addBoxToArray(**int** x, **int** y, **int** z) {  Point3D point = **new** Point3D(x,y,z);  *sort*(point); *// x = minimum of 3, z = maximum of 3*  *boxList*.add(**new** BoxAndBaseArea(point.**z**, point.**x**, point.**y**));  *boxList*.add(**new** BoxAndBaseArea(point.**z**, point.**y**, point.**x**));  *boxList*.add(**new** BoxAndBaseArea(point.**y**, point.**x**, point.**z**));  }   **private static void** sort(Point3D point) {  **if**(point.**x** > point.**y**) {  **int** temp = point.**x**;  point.**x** = point.**y**;  point.**y**= temp;  }  **if**(point.**x** > point.**z**) {  **int** temp = point.**x**;  point.**x** = point.**z**;  point.**z** = temp;  }  **if**(point.**y** > point.**z**) {  **int** temp = point.**y**;  point.**y** = point.**z**;  point.**z** = temp;  }  }   **private static int** maxElemnet(**int** [] arr) {  **int** max = 0;  **int** index = 0;  **for**(**int** i = 0; i < arr.**length**; i++) {  **if**(arr[i] > max) {  max = arr[i];  index = i;  }  }  **return** index;  }   **public static void** main (String[] args) {  Scanner sc = **new** Scanner(System.***in***);  **int** t = sc.nextInt();   *//sc.next();* **while**(t-- > 0)  {  **int** n = sc.nextInt();   **int** A[] =**new int**[1000];  **int** B[] = **new int**[1000];  **int** C[] = **new int**[1000];   **for**(**int** i = 0; i < n; i++)  {  **int** a =sc.nextInt();  **int** b = sc.nextInt();  **int** c = sc.nextInt();   A[i] = a;  B[i] = b;  C[i] = c;  }   System.***out***.println(*maxHeight*(A,B,C,n));  }  } }  */\* 1 4 4 6 7 1 2 3 4 5 6 10 12 32 output = 60  \*/* |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*; */\*\*  \* An array A containing heights of building was given.  \* Its a rainy season. Calculate the amount of water collected between all the buildings.  \*  \* calculate left maximum of all and store in array  \* calculate right max of all and store in array  \* at every moment do min(leftMax, rightMax) - arr[i]  \*  \* space optimization  \* nstead of maintaing two arrays of size n for storing left and right max of each element,   \* maintain two variables to store the maximum till that point.   \* Since water trapped at any element = min(max\_left, max\_right) – arr[i].   \* Calculate water trapped on smaller element out of A[lo] and A[hi] first and move the pointers till lo doesn’t cross hi.  \*/* **class** GFG {  **public static void** main(String[] args) {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while** (t != 0) {  **int** n = scan.nextInt();  scan.nextLine();  String[] arr = scan.nextLine().split(**" "**);   **int** leftMax = 0;  **int** rightMax = 0;  **int** lo = 0;  **int** high = n - 1;  **int** res = 0;   **while** (lo <= high) {  **if** (Integer.*valueOf*(arr[lo]) < Integer.*valueOf*(arr[high])) {  **if** (Integer.*valueOf*(arr[lo]) > leftMax) {  leftMax = Integer.*valueOf*(arr[lo]);  } **else** {  res += (leftMax - Integer.*valueOf*(arr[lo]) > 0) ? leftMax - Integer.*valueOf*(arr[lo]) : 0;  }  lo ++;  } **else** {  **if** (Integer.*valueOf*(arr[high]) > rightMax) {  rightMax = Integer.*valueOf*(arr[high]);  } **else** {  res += (rightMax - Integer.*valueOf*(arr[high]) > 0) ? rightMax - Integer.*valueOf*(arr[high]) : 0;  }  high --;  }  }   System.***out***.println(res);  t--;  }  } } |

|  |
| --- |
| **import** java.util.Scanner; **import** java.util.Stack;  */\*Complete the function given below\*/  /\*\*  \* Max rectangle  \* Given a binary matrix. Find the maximum area of a rectangle formed only of 1s in the given matrix.  \*  \* matrix will look like  \* 0 1 1 0  \* 1 1 1 1  \* 1 1 1 1  \* 1 1 0 0  \* the max size rectangle is  \* 1 1 1 1  \* 1 1 1 1  \* and area is 4 \*2 = 8.  \*  \* 1. we will use the concept of max area in a histogram   \* 2. consider first row as histogram and calculate max area  \* 3. add second row and calculate/compare the area  \* 4. repeat for all of the rows to get the final solution   \* \*/* **class** GfG {  **public int** maxArea(**int** M[][], **int** m, **int** n) {  **int**[] arr = **new int**[m];  **int** res = 0;  **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < m; j++) {  arr[j] = M[i][j] == 1 ? arr[j] + M[i][j] : 0;  }   res = Math.*max*(res, findMaxAreaUnderHistogram(arr));  }  **return** res;  }   **private int** findMaxAreaUnderHistogram(**int**[] arr) {  **int** n = arr.**length**;  Stack<Integer> stack = **new** Stack<>();  **int** i = 0;  stack.push(i);  i++;   **int** maxArea = 0;  **while** (!stack.empty()) {  **while** (!stack.isEmpty() && (i == n || arr[i] < arr[stack.peek()])) {  **int** index = stack.pop();  **int** current = arr[index] \* (stack.empty() ? i : (i - stack.peek() - 1));  maxArea = Math.*max*(maxArea, current);  }  **if** (i < n) {  stack.push(i);  }  i++;  }   **return** maxArea;  } }  **class** Solution {  **public static void** main(String[] arg) {  Scanner scanner = **new** Scanner(System.***in***);   **int** n = scanner.nextInt();  **int** m = scanner.nextInt();   **int** arr[][] = **new int**[n][m];   **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < m; j++) {  arr[i][j] = scanner.nextInt();  }  }   System.***out***.println(**new** GfG().maxArea(arr, m, n));  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*; */\*\*  \* Rotten Oranges  \*  \* Given a matrix of dimension r\*c where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:  \* 0 : Empty cell  \* 1 : Cells have fresh oranges  \* 2 : Cells have rotten oranges  \*  \* So, we have to determine what is the minimum time required to rot all oranges.  \* A rotten orange at index [i,j] can rot other fresh orange  \* at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (up, down, left and right) in unit time.  \* If it is impossible to rot every orange then simply return -1.  \* \*/* **class** GFG {  **public static void** main(String[] args) {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while** (t != 0) {  **int** n = scan.nextInt();  **int** m = scan.nextInt();  scan.nextLine();  String[] input = scan.nextLine().split(**" "**);   **int**[][] arr = **new int**[n][m];  **int** index = 0;  **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < m; j++) {  arr[i][j] = Integer.*valueOf*(input[index++]);  }  }   **int**[] xDist = {1, 0, -1, 0};  **int**[] yDist = {0, 1, 0, -1};   **boolean**[][] visited = **new boolean**[n][m];  Queue<Point> queue = **new** LinkedList<>();  **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < m; j++) {  **if** (arr[i][j] == 2) {  **for** (**int** k = 0; k < 4; k++) {  Point newPoint = **new** Point(i + xDist[k], j + yDist[k]);  **if** (newPoint.isValid(n, m) &&  arr[newPoint.**x**][newPoint.**y**] == 1 &&  !visited[newPoint.**x**][newPoint.**y**]) {  visited[newPoint.**x**][newPoint.**y**] = **true**;  queue.add(newPoint);  }  }  }  }  }  queue.add(**null**);  **int** count = 0;   **while** (!queue.isEmpty()) {  Point current = queue.poll();   **if** (current == **null**) {  count++;  **if** (!queue.isEmpty()) {  queue.add(**null**);  }  } **else** {  **int** i = current.**x**;  **int** j = current.**y**;  **for** (**int** k = 0; k < 4; k++) {  Point newPoint = **new** Point(i + xDist[k], j + yDist[k]);  **if** (newPoint.isValid(n, m) &&  arr[newPoint.**x**][newPoint.**y**] == 1 &&  !visited[newPoint.**x**][newPoint.**y**]) {  visited[newPoint.**x**][newPoint.**y**] = **true**;  queue.add(newPoint);  }  }   }  }   **boolean** isMatrixComplete = *isMatrixComplete*(arr, visited);  **if** (isMatrixComplete) {  System.***out***.println(count);  } **else** {  System.***out***.println(**"-1"**);  }  t--;  }  }   **private static boolean** isMatrixComplete(**int**[][] arr, **boolean**[][] visited) {  **for** (**int** i = 0; i < arr.**length**; i++) {  **for** (**int** j = 0; j < arr[i].**length**; j++) {  **if** (arr[i][j] == 1 && !visited[i][j]) {  **return false**;  }  }  }  **return true**;  }   **static class** Point {  **int x**;  **int y**;   Point(**int** x, **int** y) {  **this**.**x** = x;  **this**.**y** = y;  }   **boolean** isValid(**int** n, **int** m) {  **return this**.**x** >= 0 && **this**.**x** < n && **this**.**y** >= 0 && **this**.**y** < m;  }  } } |

|  |
| --- |
| **import** java.util.Stack; **class** Node {  **int data**;  Node **left**;  Node **right**; } **class** Solution {   */\*\*  \* Given a Binary Search Tree and a target sum.   \* Check whether there's a pair of Nodes in the BST with value summing up to the target sum.   \*   \* We can do in O(n) time and O(logn) space   \* travers inorder from start and reverse together   \* believing, two values can never be equal in bst   \* \*/* **public int** isPairPresent(Node root, **int** target)  {  Stack<Node> stack1 = **new** Stack<>();  Stack<Node> stack2 = **new** Stack<>();   Node temp1 = root;  Node temp2 = root;  **while** (**true**) {  **while** (temp1 != **null**) {  stack1.push(temp1);  temp1 = temp1.**left**;  }   **while** (temp2 != **null**) {  stack2.push(temp2);  temp2 = temp2.**right**;  }   **if**(stack1.empty() || stack2.empty()) {  **break**;  }  temp1 = stack1.peek();  temp2 = stack2.peek();   **if**(temp1.**data** + temp2.**data** == target) {  **if**(temp1.**data** == temp2.**data**) {  **return** 0;  }  **return** 1;  }   **if**(temp1.**data** + temp2.**data** < target) {  stack1.pop();  temp1 = temp1.**right**;  temp2 = **null**;  } **else** {  stack2.pop();  temp2 = temp2.**left**;  temp1 = **null**;  }  }  **return** 0;  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*; */\*\*  \* Zero Sum Subarrays  \*  \* Idea is simple, to keep track of sum at every point  \* put the sum and index in map  \*  \* if the sum is already encountered then we have find a zero sum sub array  \* as if sum from i -> j is S and sum from i -> z is also S then sum from j -> z must be zero  \*  \** [*https://www.youtube.com/watch?v=C9-n\_H7dsvU&feature=youtu.be*](https://www.youtube.com/watch?v=C9-n_H7dsvU&feature=youtu.be) *\*/* **class** GFG {  **public static void** main (String[] args)  {  Scanner scan = **new** Scanner(System.***in***);  **int** t = scan.nextInt();   **while**(t != 0) {  **int** n = scan.nextInt();  scan.nextLine();  String [] arr = scan.nextLine().split(**" "**);   Map<Integer, ArrayList<Integer>> map = **new** HashMap<>();  **int** sum = 0;  **int** countOfZeroSubArray = 0;  **for**(**int** i =0; i <arr.**length**; i++) {  **if**(!*isParsable*(arr[i])) {  **continue**;  }  sum += Integer.*valueOf*(arr[i]);   **if**(sum == 0) {  countOfZeroSubArray++;  }   **if**(map.containsKey(sum)) {  countOfZeroSubArray += map.get(sum).size();  map.get(sum).add(i);  } **else** {  **final int** index = i;  map.put(sum, **new** ArrayList<>() {{  add(index);  }});  }  }  System.***out***.println(countOfZeroSubArray);  t--;  }  }   **private static boolean** isParsable(String s) {  **try** {  **int** i = Integer.*parseInt*(s);  **return true**;  } **catch** (Exception e) {  **return false**;  }  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*;   */\*\*  \* Strongly connected component using TARJANs' ALGO  \*  \*https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/  \*   \* candidates have same low point belog to same component  \* if B is child of A  \* 1. if not visited yet, then visit B and update A's lowPoint using B's lowPoint  \* 2. if already visited, then update A's lowPoint using B's TimeToReach  \*   \* If it is a root node then print the strongly connected vertex in this component  \*/* **class** Solution {   **int time**;   **public void** findStronglyConnectedComponent(ArrayList<ArrayList<Integer>> adj, **int** N) {  **time** = 0;  Stack<Integer> stack = **new** Stack<>();  **boolean**[] inStack = **new boolean**[adj.size()];   **int**[] timeToReach = **new int**[adj.size()];  **int**[] lowPoint = **new int**[adj.size()];   **for** (**int** i = 0; i < adj.size(); i++) {  timeToReach[i] = -1;  lowPoint[i] = -1;  }   **for** (**int** i = 0; i < adj.size(); i++) {  **if** (timeToReach[i] == -1) {  findSscUtil(adj, timeToReach, lowPoint, stack, inStack, i);  }  }  }   **private void** findSscUtil(ArrayList<ArrayList<Integer>> adj,  **int**[] timeToReach,  **int**[] lowPoint,  Stack<Integer> stack,  **boolean**[] inStack, **int** u) {  timeToReach[u] = **time**;  lowPoint[u] = **time**;  stack.push(u);  inStack[u] = **true**;   **for** (**int** v : adj.get(u)) {  **if** (timeToReach[v] == -1) {  findSscUtil(adj, timeToReach, lowPoint, stack, inStack, v);   lowPoint[u] = Math.*min*(lowPoint[u], lowPoint[v]);  } **else if** (inStack[v]) {  lowPoint[u] = Math.*min*(lowPoint[u], timeToReach[v]);  }  }   *// if we are on Head Node* **if** (lowPoint[u] == timeToReach[u]) {  **while** (!stack.empty() && stack.peek() != u) {  **int** x = stack.pop();  System.***out***.print(x + **" "**);  inStack[x] = **false**;  }   **if** (!stack.empty() && stack.peek() == u) {  **int** x = stack.pop();  System.***out***.print(x);  inStack[x] = **false**;  }  System.***out***.print(**","**);  }  } } |

|  |
| --- |
| **import** java.util.\*; **import** java.lang.\*; **import** java.io.\*; */\*\*  \* Given a singly linked list of size N, and an integer K.   \* You need to swap the Kth node from beginning and Kth node from the end in the linked list.  \* Note: You need to swap the nodes through the links and not changing the content of the nodes.  \*  \* Example 1:  \*  \* Input:  \* N = 4, K = 1  \* value[] = {1,2,3,4}  \* Output: 1  \* Explanation: Here K = 1, hence after  \* swapping the 1st node from the beginning  \* and end thenew list will be 4 2 3 1.  \*/* **class** GFG {  Node swapkthnode(Node head, **int** num, **int** k) {  *// Count nodes in linked list* **int** n = countNodes(head);   **if** (n < k)  **return** head;   *// If x (kth node from start) and  // y(kth node from end) are same* **if** (2 \* k - 1 == n)  **return** head;   *// Find the kth node from beginning of linked list.  // We also find previous of kth node because we need  // to update next pointer of the previous.* Node x = head;  Node x\_prev = **null**;  **for** (**int** i = 1; i < k; i++) {  x\_prev = x;  x = x.**next**;  }   *// Similarly, find the kth node from end and its  // previous. kth node from end is (n-k+1)th node  // from beginning* Node y = head;  Node y\_prev = **null**;  **for** (**int** i = 1; i < n - k + 1; i++) {  y\_prev = y;  y = y.**next**;  }   *// If x\_prev exists, then new next of it will be y.  // Consider the case when y->next is x, in this case,  // x\_prev and y are same. So the statement  // "x\_prev->next = y" creates a self loop. This self  // loop will be broken when we change y->next.* **if** (x\_prev != **null**)  x\_prev.**next** = y;   *// Same thing applies to y\_prev* **if** (y\_prev != **null**)  y\_prev.**next** = x;   *// Swap next pointers of x and y. These statements  // also break self loop if x->next is y or y->next  // is x* Node temp = x.**next**;  x.**next** = y.**next**;  y.**next** = temp;   *// Change head pointers when k is 1 or n* **if** (k == 1)  head = y;   **if** (k == n)  head = x;   **return** head;  }    **private int** countNodes(Node head) {  Node temp = head;  **int** count = 0;   **while** (temp != **null**) {  count++;  temp = temp.**next**;  }   **return** count;  } }  **class** Node {  **int data**;  Node **next**;   Node(**int** data)  {  **this**.**data** = data;  **next** = **null**;  } } |