|  |  |
| --- | --- |
| **Design an algorithm and write code to remove the duplicate characters in a string without using any additional buffer. Write the test cases for this method.** | **Implement a Doubly Linked List that has methods a) add b) remove.** |
| public string removeDuplicates(string str) {      //test cases : aaabbb, ababab, aaaa, abcd, null, "", aabc  if (string.IsNullOrEmpty(str) == true) return "";  var arr = str.ToArray();  int tail = 1;  for (int i = 1; i < arr.Length; i++) {  int j = 0;  for (; j < tail; j++) {  if (arr[i] == arr[j]) break;  }  if (j == tail) {  arr[j] = arr[i];  tail++;  }  }  var result = new char[tail];  for (int i = 0; i < result.Length; i++) {  result[i] = arr[i];  }  return new string(result);  } | public class DoublyLinkedList {  public Node head;  public Node tail;  public void add(int v) {  if (head == null) {  head = new Node(v);  tail = head;  return;  }  tail.next = new Node(v);  tail.next.prev = tail;  tail = tail.next;  }  public Node remove(int v) {  if (head == null) return null;  if (head.value == v) {  if (head == tail) {  var ret = head;  head = null;  tail = null;  return ret;  }  var ret2 = head;  head.next.prev = null;  head = head.next;  return ret2;  }  for (var node = head.next; node.next != null; node = node.next) {  if (node.value == v) {  node.prev.next = node.next;  node.next.prev = node.prev;  return node;  }  }  if (tail.value == v) {  var ret = tail;  tail.prev.next = null;  tail = tail.prev;  return ret;  }  return null;  }  } |
| **Write code to remove duplicates from an unsorted linked list without using a temporary buffer.** |
| public Node removeDuplicates(Node head) {  if (list == null || list.head == null) return;  var tail = list.head.next;  var prev = list.head;  for (var i = tail; i != null; i = i.next) {  Node j = list.head;  for (; j != tail; j = j.next) {  if (i.value == j.value) break;  }  if (j == tail) {  j.value = i.value;  prev = tail;  tail = tail.next;  }  }  prev.next = null;  return head;  } |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Implement a) Stack b) Queue.** | | **Write code for Tower of Hanoi.** | | | | | | | | | | | |
| public class Stack {  private Node top;  public void push(int v) {  var node = new Node(v);  node.next = top;  top = node;  }  public int pop() { {  if (top == null) return -1;  var ret = top.value;  top = top.next;  return ret;  }  }  public class Queue {  public Node head;  public Node tail;  public void enq(int v) {  if (head == null) {  head = new Node(v);  tail = head;  return;  }  var node = new Node(v);  tail.next = node;  tail = node;  }  public int deq() {  if (head == null) return -1;  var ret = head.value;  head = head.next;  tail = (head == null) ? null : tail;  return ret;  }  } | | public void hanoi(int n) {  if (n <= 0) return;  solve(n, 'A', 'B', 'C');  }  private void hanoi(int n, char from, char tmp, char to) {  if (n == 1) {  PrintLn("Moving " + n + " from " + from + " to " + to);  return;  }  solve(n - 1, from, to, tmp);  PrintLn("Moving " + n + " from " + from + " to " + to);  solve(n - 1, tmp, from, to);  } | | | | | | | | | | | |
| **Implement the following algorithms: a) In-Order b) Pre-Order c) Post-Order d) Level-Order e) Insert Node Method** | | | | | | | | | | | |
| public void PrintInOrder() {  PrintInOrder(Root);  } | | | | | | public void PrintPreOrder() {  PrintPreOrder(Root);  } | | | | | public void PrintPostOrder() {  PrintPostOrder(Root);  } |
| private void PrintInOrder(Node node) {  if (node == null) return;  PrintInOrder(node.Left);  Print(node.ToString() + ", ");  PrintInOrder(node.Right);  }  private void PrintPreOrder(Node node) {  if (node == null) return;  Print(node.ToString() + ", ");  PrintPreOrder(node.Left);  PrintPreOrder(node.Right);  }  private void PrintPostOrder(Node node) {  if (node == null) return;  PrintPostOrder(node.Left);  PrintPostOrder(node.Right);  Print(node.ToString() + ", ");  } | | | | | | | | public void Insert(int value) {  Root = Insert(Root, value);  }  private Node Insert(Node node, int value) {  if (node == null) {  return new Node(value);  }  if (value < node.value) {  node.Left = Insert(node.Left, value);  } else {  node.Right = Insert(node.Right, value);  }  return node;  } | | | |
| public void PrintLevelOrder() {  if (Root == null) return;  var q = new Queue<Node>(); q.Enqueue(Root);  while (q.Count > 0) {  var node = q.Dequeue();  Print(node.ToString() + ", ");  if (node.Left != null) q.Enqueue(node.Left);  if (node.Right != null) q.Enqueue(node.Right);  }  } | | | **Implement a) Depth First Search b) Breadth First Search** | | | | | | | public class AdjList {  private List<List<Edge>> vertexList;  public AdjList(int count) {  vertexList = new List<List<Edge>>();  for (int i = 0; i < count; i++) {  vertexList.Add(new List<Edge>());  }  }  } | | | |
| public class Edge {  public int vertex { get; set; }  public int cost { get; set; }  public Edge(int v, int c) {  vertex = v;  cost = c;  }  } | | | | | | |
| public void DepthFirstSearch() {  if (vertexList == null || vertexList.Count() <= 0) return;  var hashSet = new HashSet<int>();  DepthFirstSearch(0, hashSet);  }  public void DepthFirstSearch(int vertex, HashSet<int> hashSet) {  hashSet.Add(vertex);  var edgeList = vertexList[vertex];  foreach (var edge in edgeList) {  if (hashSet.Contains(edge.vertex) == true) continue;  DepthFirstSearch(edge.vertex, hashSet);  }  } | | | | | | public void BreadthFirstSearch() {  var hs = new HashSet<int>();  var q = new Queue<int>();  q.Enqueue(0);  hs.Add(0);  while (q.Count > 0) {  var vertex = q.Dequeue();  var edges = map[vertex];  printf(vertex + " - ");  foreach (var edge in edges) {  if (hs.Contains(edge.vertex) == true) continue;  q.Enqueue(edge.vertex);  hs.Add(edge.vertex);  }  }  } | | | | | | | |
| **Given a (decimal – e.g. 3.72) number that is passed in as a string, print the binary representation. If the number cannot be represented accurately in binary, print “ERROR”.** | | | | | | **Implement the following: a) Bubble Sort b) Selection Sort c) Merge Sort d) Quick Sort e) Bucket Sort** | | | | | | | |
| private string PrintBinary(string n) {  var split = n.Split('.');  int num = Convert.ToInt32(split[0]);  var dec = Convert.ToDouble("0." + split[1]);  var decString = "";  var stack = new Stack<char>();  while (num > 0) {  stack.Push((num % 2 == 0) ? '0' : '1');  num >>= 1;  }  var sbNum = new StringBuilder();  var sbDec = new StringBuilder();  while (stack.Count > 0) sbNum.Append(stack.Pop());  int cnt = 0;  while (dec != 0.0) {  if (cnt > 32) return "ERROR";  dec \*= 2.0;  if (dec >= 1.0) {  sbDec.Append("1");  dec -= 1.0;  } else {  sbDec.Append("0");  }  }  cnt++;  return sbNum.ToString();+ "." + sbDec.ToString();  } | | | | | | private void bubbleSort(int[] n) {  for (int i = n.Length - 1; i >= 0; i--) {  var swapped = false;  for (int j = 0; j < i; j++) {  if (n[j] > n[j + 1]) {  var tmp = n[j + 1];  n[j + 1] = n[j];  n[j] = tmp;  swapped = true;  }  }  if (swapped == false) break;  }  }  private void selectionSort(int[] n) {  for (int i = 0; i < n.Length; i++) {  int index = i;  for (int j = i + 1; j < n.Length; j++) {  if (n[j] < n[index]) {  index = j;  }  }  int tmp = n[i];  n[i] = n[index];  n[index] = tmp;  }  } | | | | | | | |
| private int[] mergeSort(int[] n) {  return mergeSort(n, 0, n.Length);  }  private int[] mergeSort(int[] n, int s, int e) {  if (e - s <= 1) return new int[] { n[s] };  int mid = (s + e) / 2;  int count = e - s;  var left = mergeSort(n, s, mid);  var right = mergeSort(n, mid, e);  var tmp = new int[count];  int li = 0;  int ri = 0;  for (int index = 0; index < count; index++) {  if (li < left.Length && (ri >= right.Length || left[li] < right[ri])) {  tmp[index] = left[li++];  } else {  tmp[index] = right[ri++];  }  }  return tmp;  } | | | | | | | | | public void quickSort(int[] n) {  quickSort(n, 0, n.Length - 1);  }  private void quickSort(int[] n, int s, int e) {  if (s >= e) return;  int p = partition(n, s, e);  quickSort(n, s, p - 1);  quickSort(n, p + 1, e);  }  private int partition(int[] n, int s, int e) {  int piv = n[e], p = e;  for (int i = s; i < p; i++) {  if (n[i] > piv) {  swap(n, i, p - 1);  swap(n, p, p - 1);  i--;  p--;  }  }  return p;  } | | | | |
| public void bucketSort(int[] n) {  int max = Int32.MinValue;  for (int i = 0; i < n.Length; i++) max = (n[i] > max) ? n[i] : max;  var buckets = new List<int>[max + 1];  for (int i = 0; i < buckets.Length; i++) buckets[i] = new List<int>();  for (int i = 0; i < n.Length; i++) {  var b = n[i] % (max + 1);  buckets[b].Add(n[i]);  }  int index = 0;  for (int i = 0; i < buckets.Length; i++) {  var bucket = buckets[i];  for (int j = 0; j < bucket.Count; j++) {  n[index++] = bucket[j];  }  }  } | | | | | | | | | **Implement binary search.** | | | | |
| public int BinarySearch (int[] arr, int key) {  int min = 0;  int max = arr.Length - 1;      while (min <= max) {          int mid = (min + max) / 2;          if (key == arr[mid]) {              return mid;          } else if (key < arr[mid]) {             max = mid - 1;          } else {              min = mid + 1;          }     }     return -1;  } | | | | |
| **Given an integer, write a function to determine if it is a power of three.** | | | | **Write a function to find a) gcd b) lcm of two numbers.** | | | | | | | | | |
| public bool IsPowerOfThree(int n) {  return (Math.Log10(n) / Math.Log10(3)) % 1 == 0;  } | | | | public int gcd(int a, int b) {  while (b != 0) {  int t = b;  b = a % b;  a = t;  }  return a; } | | | | | | | | //gcd(a,b) \* lcm(a,b) = ab  public int lcm(int a, int b) {  if (a == 0 || b == 0) return 0;  var t = gcd(a, b);  if (t == 0) return 0;  return (a \* b) / t;  } | |
| **Write an efficient algorithm that searches for a value in an *m x n* matrix. This matrix has the following property: it is sorted from left to right, top to bottom.** | | | | | **Implement a function to check if a tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that no two leaf nodes differ in distance from the root by more than one.** | | | | | | | | |
| public bool SearchMatrix(int[,] matrix, int target) {  if (matrix == null) return false;  var m = matrix.GetLength(1);  var n = matrix.GetLength(0);  if (m == 0 || n == 0) return false;  int start = 0, end = m \* n - 1;  while (start <= end) {  int mid = (start + end) >> 1;  int midX = mid / m;  int midY = mid % m;  if (matrix[midX, midY] == target) return true;  else if (target < matrix[midX, midY]) end = mid - 1;  else start = mid + 1;  }  return false;  } | | | | | public bool isBalanced() {  if (root == null) return false;  return getMaxDepth(root) - getMinDepth(root) <= 1;  }  public int getMinDepth(Node node) {  if (node == null) return 0;  return 1 + Math.Min(getMinDepth(node.left), getMinDepth(node.right));  }  public int getMaxDepth(Node node) {  if (node == null) return depth;  return 1 + Math.Max(getMaxDepth(node.left), getMaxDepth(node.right));  } | | | | | | | | |
| **Implement a full tree with *n* nodes.** | **Given a sorted (increasing order) array, write an algorithm to create a binary tree with minimal height.** | | | | | | | | | | | | |
| public Node createFullTree(int n) {  if (n <= 0) return null;  int cnt = 1;  var q = new Queue<int>();  var root = new Node(0);  q.Enqueue(root);  while (cnt < n && q.Count > 0) {  var node = q.Dequeue();  if (cnt++ < n) {  node.left = new Node(0);  q.Enqueue(node.left);  }  if (cnt++ < n) {  node.right = new Node(0);  q.Enqueue(node.right);  }  }  return root;  } | public Node createBinaryTree(int[] arr) {  if (arr == null || arr.Length == 0) return null;  return createBinaryTree(arr, 0, arr.Length - 1);  }  private Node createBinaryTree(int[] arr, int min, int max) {  if (min > max) return null;  int mid = (min + max) >> 1;  var node = new Node(arr[mid]);  node.left = createBinaryTree(arr, min, mid - 1);  node.right = createBinaryTree(arr, mid + 1, max);  return node;  } | | | | | | | | | | | | |
| **Write an algorithm to find the ‘next’ node of a given node in a binary search tree where each node has a link to its parent for a) In-Order Successor b) Pre-Order Successor c) Post-Order Successor** | | | | | | | | | | | | |
| public Node getInOrderSuccessor(Node node) {  if (node == null) return null;  if (node.parent == null || node.right != null)  return getLeftMost(node.right);  while (node.parent != null) {  if (node.parent.left == node) return node.parent;  node = node.parent;  }  return null;  } | | | | | | | | | | private Node getLeftMost(Node node) {  if (node == null) return null;  while (node.left != null) node = node.left;  return node;  } | | |
| public Node getPreOrderSuccessor(Node node) {  if (node == null) return null;  if (node.left != null) return node.left;  if (node.right != null) return node.right;  var cur = node;  while (node.parent != null) {  node = node.parent;  if (node.left == cur && node.right != null) {  return node.right;  }  cur = node;  }  return null;  } | | | | | | | public Node getPostOrderSuccessor(Node node) {  if (node == null) return null;  if (node.parent == null) return null;  if (node.parent.right == null) return node.parent;  if (node.parent.right == node) return node.parent;  node = node.parent.right;  while (node.Left != null || node.Right != null) {  node = (node.Left != null) ? node.Left : node.Right;  } return node.parent;  } | | | | | | |
| **Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.** | | | | | | | **Given a string, find the first non-repeating character in it and return its index. If it doesn't exist, return -1.** | | | | | | |
| public Node getCommonAncestor(Node n1, Node n2) {  if (n1 == null || n2 == null || root == null) return null;  if (n1 == root || n2 == root) return null;  return getCommonAncestor(root, n1, n2);  }  private Node getCommonAncestor(Node parent, Node n1, Node n2) {  if (parent == null) return null;  if ((contains(parent.left, n1) && contains(parent.right, n2)) ||  (contains(parent.left, n2) && contains(parent.right, n1))) {  return parent;  }  else if (contains(parent.left, n1) && contains(parent.left, n2)) {  if (parent.left == n1 || parent.left == n2) return parent;  else return getCommonAncestor(parent.left, n1, n2);  }  else if (contains(parent.right, n1) && contains(parent.right, n2)) {  if (parent.right == n1 || parent.right == n2) return parent;  else return getCommonAncestor(parent.right, n1, n2);  }  return null;  }  private bool contains(Node node, Node target) {  if (node == null) return false;  if (node == target) return true;  return contains(node.left, target) || contains(node.right, target);  } | | | | | | | public int FirstUniqChar(string s) {  var map = new int[26];  for (int i = 0; i < s.Length; i++) {  var c = s.ElementAt(i);  map[c - 'a']++;  }  for (int i = 0; i < s.Length; i++) {  if (map[s.ElementAt(i) - 'a'] == 1) return i;  }  return -1;  } | | | | | | |
| **Write a method to sort an array of strings so that all the anagrams are next to each other.** {"silent","abcdef","kfc","listen","tenlis"} 🡪 {,"abcdef","kfc","silent","listen","tenlis"} | | | | | | |
| public void sortChars(string[] s) {  Array.Sort(s, new AnagramComparer());  }  private class AnagramComparer : IComparer<String> {  public int Compare(String s1, String s2) {  var c1 = s1.ToArray();  var c2 = s2.ToArray();  Array.Sort(c1);  Array.Sort(c2);  var a1 = new string(c1);  var a2 = new string(c2);  return a1.CompareTo(a2);  }  } | | | | | | |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Given an integer *n*, return all distinct solutions to the *n*-queens puzzle. ‘Q’ and ‘.’ both indicate a queen and an empty space respectively.** | | | | | | | | **Given a collection of intervals, merge all overlapping intervals.**  **For example, Given**[1,3],[2,6],[8,10],[15,18]**, return**[1,6],[8,10],[15,18]**.** | | |
| public List<List<string>> SolveNQueens(int n) {  if (n <= 0) return null;  var rows = new int[n];  var result = new List<List<string>>();  for (int i = 0; i < n; i++) rows[i] = -1;  placeQueens(n, rows, 0, result);  return result;  }  private void placeQueens(int n, int[] rows, int i, List<List<string>> result) {  if (i == n) {  var list = new List<string>();  for (int x = 0; x < n; x++) {  var arr = new char[n];  for (int y = 0; y < n; y++) {  arr[y] = (rows[x] == y) ? 'Q' : '.';  }  list.Add(new string(arr));  }  result.Add(list);  return;  }  for (int j = 0; j < n; j++) {  if (canPlaceQueen(n, rows, i, j) == true) {  rows[i] = j;  placeQueens(n, rows, i + 1, result);  }  rows[i] = -1;  }  }  private bool canPlaceQueen(int n, int[] rows, int i, int j) {  for (int x = 0; x < n; x++) if (rows[x] == j) return false;  for (int x = 0; x < n; x++)  if (rows[x] != -1 && rows[x] - x == j - i) return false;  for (int x = 0; x < n; x++)  if (rows[x] != -1 && rows[x] + x == i + j) return false;  return true;  }   |  | | --- | | **Given a binary tree, find its depth.** | | public int getDepth(Node root) {  if (root == null) return 0;  return 1 + Math.Max(getDepth(root.left), getDepth(root.right));  } | | | | | | | | | public List<Interval> Merge(IList<Interval> intervals) {  if (intervals == null || intervals.Count == 0) return null;  if (intervals.Count == 1) return intervals.ToList();  var arr = intervals.ToArray();  Array.Sort(arr, new QComparer());  var result = new List<Interval>();  var last = arr[0];  //[1,3],[2,6],[8,10],[15,18] -> [1,6],[8,10],[15,18].  //[1,4],[2,3] //[1,4],[1,3] //[1,4],[4,5]  for (int i = 1; i < arr.Length; i++) {  var cur = arr[i];  if (cur.start <= last.end) {  last.end = Math.Max(cur.end, last.end);  } else {  result.Add(last);  last = cur;  }  }  result.Add(last);  return result;  }  private class QComparer : IComparer<Interval> {  public int Compare(Interval n1, Interval n2) {  return n1.start - n2.start;  }  }  public class Interval {  public int start;  public int end;  public Interval() { start = 0; end = 0; }  public Interval(int s, int e) { start = s; end = e; }  } | | |
| **In how many ways can you climb stairs to the top if you either climb 1 or 2 steps?** | | |
| public int ClimbStairs(int n) {  return ClimbStairs(n, new int[n + 1]);  }  private int ClimbStairs(int n, int[] map) {  if (n <= 1) return 1;  if (n == 2) return 2;  if (map[n] != 0) return map[n];  map[n] = ClimbStairs(n - 1) + ClimbStairs(n - 2);  return map[n];  } | | |
| public int ClimbStairs(int n) {  if (n <= 0) return 0;  if (n <= 2) return n;  int a = 1; int b = 2;  int sum = 0;  for (int i = 2; i < n; i++) {  sum = a + b;  a = b;  b = sum;  }  return sum;  } | **One way to serialize a binary tree is to use pre-order traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as #. For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,#,#,2,#,6,#,#", where # represents a null node. Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.** | | | | | | | | | |
| public bool IsValidSerialization(string preorder) {  if (string.IsNullOrEmpty(preorder)) return false;  if (preorder.Length == 1 &&  preorder.ElementAt(0) == '#') return true;  var arr = preorder.Split(',');  int nodeCnt = 0, nullCnt = 0;  foreach (var c in arr) {  if (nullCnt > nodeCnt) return false;  if (c == "#") nullCnt++;  else nodeCnt++;  }  return nullCnt == nodeCnt + 1;  } | | | | | | | | **Count the number of prime numbers less than a non-negative number, *n*.** | |
| public int CountPrimes(int n) {  if (n <= 2) return 0;  //Sieve of Eratosthenes  var isComp = new bool[n];  for (int i = 2; i \* i < n; i++) {  if (isComp[i] == false) {  for (int j = i; i \* j < n; j++) {  isComp[i \* j] = true;  }  }  }  int cnt = 0;  for (int i = 2; i < n; i++) {  if (isComp[i] == false) cnt++;  }  return cnt;  } | |
| **Write a method that calculates the mean of two input values.** |
| public int mean(int s, int e) {  return s + ((e - s) >> 1);  } |
| **Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.** | | | **You have two very large binary trees: T1, with millions of nodes, and T2, with hun­dreds of nodes. Create an algorithm to decide if T2 is a subtree of T1.** | | | | | | | |
| public Node DeleteDuplicates(Node head) {  if (head == null) return null;  Node dummy = new Node(0);  dummy.next = head;  var cur = head;  var prev = dummy;  while (cur != null && cur.next != null) {  if (cur != null && cur.val == cur.next.val) {  while (cur.next != null && cur.val == cur.next.val) {  cur = cur.next;  }  } else {  prev.next = cur;  prev = prev.next;  }  cur = cur.next;  }  prev.next = cur;  return dummy.next;  } | | | public bool isSub(Node t1, Node t2) {  if (t2 == null) return true;//empty tree is always a subtree  return isSubTree(t1, t2);  }  private bool isSubTree(Node n1, Node n2) {  if (n1 == null) return false;  if (n1.val == n2.val) {  if (isMatch(n1, n2)) return true;  }  return isSubTree(n1.left, n2) || isSubTree(n1.right, n2);  }  private bool isMatch(Node n1, Node n2) {  if (n2 == null && n1 == null) return true;  if (n2 != null && n1 == null) return false;  if (n2 == null && n1 != null) return true;  if (n1.val != n2.val) return false;  return isMatch(n1.left, n2.left) && isMatch(n1.right, n2.right);  } | | | | | | | |
| **Given a 2D board and a word, find if the word exists in the grid. The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once. For example, given board = [ ["ABCE"], ["SFCS"], ["ADEE"] ]**  **"ABCCED" returns true, "SEE" returns true, "ABCB" returns false.** | | | | | | | | | | **Given an array of integers, every element appears twice except for one. Find that single one. Note: Could you implement it without using extra memory?** |
| public bool Exist(char[,] board, string word) {  if (board == null || board.GetLength(0) == 0 || board.GetLength(1) == 0) return false;  if (string.IsNullOrEmpty(word) == true) return true;  int imax = board.GetLength(0), jmax = board.GetLength(1);  for (int i = 0; i < imax; i++) {  for (int j = 0; j < jmax; j++) {  if (board[i, j] == word.ElementAt(0)) {  if (Exist(board, word, i, j, 0, new bool[imax, jmax]) == true) return true;  }  }  }  return false;  }  private bool Exist(char[,] board, string word, int i, int j, int index, bool[,] visited) {  if (index == word.Length) return true;  if (i < 0 || i >= board.GetLength(0) || j < 0 || j >= board.GetLength(1)) return false;  if (visited[i, j] == true) return false;  var imax = board.GetLength(0);  var jmax = board.GetLength(1);  visited[i, j] = true;  if (board[i, j] == word.ElementAt(index)) {  if (Exist(board, word, i, j + 1, index + 1, visited) == true) return true;  if (Exist(board, word, i + 1, j, index + 1, visited) == true) return true;  if (Exist(board, word, i, j - 1, index + 1, visited) == true) return true;  if (Exist(board, word, i - 1, j, index + 1, visited) == true) return true;  }  visited[i, j] = false;  return false;}   |  | | --- | | **Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree where we allow a node to be a descendant of itself.** | | public Node LowestCommonAncestor(Node root, Node p, Node q) {  if (root == null) return null;  if (root == p || root == q) return root;  //post order  var left = LowestCommonAncestor(root.left, p, q);  var right = LowestCommonAncestor(root.right, p, q);  if (left != null && right != null) return root;  else return (left != null) ? left : right;  } | | | | | | | | | | | public int SingleNumber(int[] nums) {  var ret = 0;  foreach (var num in nums) ret ^= num;  return ret;  } |
| **Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad. You are given an API bool IsBadVersion(version) which will return whether version is bad. Implement a function to find the first bad version.** |
| /\* bool IsBadVersion(int version); \*/  public int FirstBadVersion(int n) {  int s = 1, e = n;  int v = s + ((e - s) >> 1);  for (; s < e; v = s + ((e - s) >> 1)) {  if (IsBadVersion(v) == true) {  e = v - 1;  } else {  s = v + 1;  }  }  return (IsBadVersion(v) == true) ? v : v + 1;  } |
| **Given a linked list, remove the *n*th node from the end of list and return its head.** |
| public Node RemoveNthFromEnd(Node head, int n) {  if (head == null) return null;  var dummy = new Node(0);  dummy.next = head;  var cur = dummy; var prev = dummy;  for (int i = 0; i < n; i++) {  if (cur == null) return head;  cur = cur.next;  }  while (cur.next != null) {  cur = cur.next;  prev = prev.next;  }  prev.next = prev.next.next;  return dummy.next;  } |
| **Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).**  **1. You may assume that the matrix does not change. 2. There are many calls to sumRegion function. 3. You may assume that row1 ≤ row2 and col1 ≤ col2.** | | | | | | | | | | |
| public class NumMatrix {  public NumMatrix(int[,] matrix) {}  public int SumRegion(int row1, int col1, int row2, int col2) {}  }  // Your NumMatrix object will be instantiated and called as such:  // NumMatrix numMatrix = new NumMatrix(matrix);  // numMatrix.SumRegion(0, 1, 2, 3);  // numMatrix.SumRegion(1, 2, 3, 4); | | | | | | | public class NumMatrix {  private int[,] m;  public NumMatrix(int[,] matrix) {  m = new int[matrix.GetLength(0), matrix.GetLength(1) + 1];  for (int i = 0; i < matrix.GetLength(0); i++) {  for (int j = 0; j < matrix.GetLength(1); j++) {  m[i, j + 1] = m[i, j] + matrix[i, j];  }  }  }  public int SumRegion(int row1, int col1, int row2, int col2) {  var sum = 0;  for (int i = row1; i <= row2; i++) {  sum += (m[i, col2 + 1] - m[i, col1]);  }  return sum;  }  } | | | |
| public class NumMatrix {  private int[,] m;  public NumMatrix(int[,] matrix) {  m = new int[matrix.GetLength(0) + 1, matrix.GetLength(1) + 1];  for (int i = 0; i < matrix.GetLength(0); i++) {  for (int j = 0; j < matrix.GetLength(1); j++) {  m[i+1,j+1] = m[i,j+1] + m[i+1,j] + matrix[i,j] - m[i,j];  }  }  }  public int SumRegion(int row1, int col1, int row2, int col2) {  return m[row2+1,col2+1]-m[row1,col2+1]-m[row2+1,col1]+m[row1,col1];  }  } | | | | | | |
| **Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node (recursively & iteratively).** | | | | | | | | public int MinDepth(Node root) {  if (root == null) return 0;  var q = new Queue<Node>();  int cur = 0, last = 1, depth = 1;  q.Enqueue(root);  while (q.Count > 0) {  var node = q.Dequeue();  if (node.left == null && node.right == null)  return depth;  last--;  if (node.left != null) {  q.Enqueue(node.left); cur++;  }  if (node.right != null) {  q.Enqueue(node.right); cur++;  }  if (last == 0) {  depth++; last = cur; cur = 0;  }  }  return 0;  } | | |
| public int MinDepth(Node root) {  if (root == null) return 0;  if (root.left == null) return 1 + MinDepth(root.right);  if (root.right == null) return 1 + MinDepth(root.left);  return 1 + Math.Min(MinDepth(root.left), MinDepth(root.right));  } | | | | | | | |
| **Sort a linked list using insertion sort.** | | | | | | | |
| public ListNode InsertionSortList(Node head) {  if (head == null || head.next == null) return head;  var dummy = new Node(0);  for (var cur = head; cur != null;) {  var next = cur.next; var node = dummy;  while (node.next != null && node.next.val < cur.val) node = node.next;  cur.next = node.next;  node.next = cur;  cur = next;  }  return dummy.next;  } | | | | | | | |
| **Given an m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.**  **Note: You can only move either down or right at any point in time.** | | | | | | | | | | |
| public int MinPathSum(int[,] grid) {  if (grid == null || grid.GetLength(0) == 0  || grid.GetLength(1) == 0) return 0;  return minPathSum(grid, 0, 0, 0);  }  private int minPathSum(int[,] grid, int i, int j, int sum) {  if (i >= grid.GetLength(0) || j >= grid.GetLength(1))  return Int32.MaxValue;  if (i == grid.GetLength(0) - 1 && j == grid.GetLength(1) - 1) {  sum += grid[i, j];  return sum;  }  sum += grid[i, j];  return Math.Min(minPathSum(grid, i, j + 1, sum),  minPathSum(grid, i + 1, j, sum));  }  **Given a binary tree, return the inorder/preorder/postorder traversal of its nodes' values without using recursion.**  public List<int> InorderTraversal(Node root) {  if (root == null) return new List<int>();  var result = new List<int>(); var stack = new Stack<Node>();  var cur = root;  while (cur != null || stack.Count > 0) {  while (cur != null) {  stack.Push(cur);  cur = cur.left;  }  cur = stack.Pop();  result.Add(cur.val);  cur = cur.right;  }  return result; } | | | | public int MinPathSum(int[,] grid) {  if (grid == null || grid.GetLength(0) == 0 || grid.GetLength(1) == 0) return 0;  var m = grid.GetLength(1);  var n = grid.GetLength(0);  var dp = new int[n, m];  dp[0, 0] = grid[0, 0];  for (int i = 1; i < m; i++) dp[0, i] = grid[0, i] + dp[0, i - 1];  for (int i = 1; i < n; i++) dp[i, 0] = grid[i, 0] + dp[i - 1, 0];  for (int i = 1; i < n; i++) {  for (int j = 1; j < m; j++) {  if (dp[i - 1, j] > dp[i, j - 1]) {  dp[i, j] = dp[i, j - 1] + grid[i, j];  } else {  dp[i, j] = dp[i - 1, j] + grid[i, j];  }  }  }  return dp[n - 1, m - 1];  } | | | | | | |
| **Given an integer *n*, return the number of trailing zeroes in *n*!** | | | | | | |
| public int TrailingZeroes(int n) {  if (n < 5) return 0;  var cnt = 0; var div = 0;  for (int i = n; i >= 5; i = div) {  div = i / 5;  cnt += div;  }  return cnt;  }  //5! --> 5x1 = 1; 25! = 5x5 + 25x1 = 6  //125! = 5x25 + 25x5 + 125x1 = 31 | | | | | | |
| public List<int> PreorderTraversal(Node root) {  if (root == null) return new List<int>();  var list = new List<int>(); var stack = new Stack<Node>();  stack.Push(root);  while (stack.Count > 0) {  var node = stack.Pop();  list.Add(node.val);  if (node.right != null) stack.Push(node.right);  if (node.left != null) stack.Push(node.left);  }  return list;  } | | **Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array. Formally the function should return true if there exists *i, j, k*such that *arr[i]* < *arr[j]* < *arr[k]* given**  **0 ≤ *i* < *j* < *k* ≤ *n*-1 else return false. Your algorithm should run in O(*n*) time and O(*1*) space complexity.** | | | | | | | | |
| public bool IncreasingTriplet(int[] nums) {  var a = Int32.MaxValue; var b = Int32.MaxValue;  foreach (var num in nums) {  if (num <= a) a = num;  else if (num <= b) b = num;  else return true;  }  return false;  } //*the\_smallest\_so\_far < the\_second\_smallest\_so\_far < current* | | | | | | | | |
| public List<int> PostorderTraversal(Node root) {  if (root == null) return new List<int>();  var list = new List<int>(); var stack = new Stack<Node>();  stack.Push(root); var prev = root;  while (stack.Count > 0) {  var node = stack.Pop();  if (node.left == null && node.right == null) {  prev = node; list.Add(node.val);  } else if (node.left == prev || node.right == prev) {  prev = node; list.Add(node.val);  } else {  stack.Push(node);  if (node.right != null) stack.Push(node.right);  if (node.left != null) stack.Push(node.left);  }  }  return list;  }  **Check whether a binary tree is symmetric recursively and iteratively. For example, this binary tree [1,2,2,3,4,4,3] is symmetric.**  public bool IsSymmetric(TreeNode root) {  if (root == null) return true;  return IsSymmetric(root.left, root.right);  }  private bool IsSymmetric(TreeNode n1, TreeNode n2) {  if (n1 == null && n2 == null) return true;  if (n1 == null || n2 == null) return false;  return n1.val == n2.val && IsSymmetric(n1.left, n2.right) && IsSymmetric(n1.right, n2.left);  }  public bool IsSymmetric(TreeNode root) {  if (root == null) return true;  var ls = new Stack<TreeNode>(); var rs = new Stack<TreeNode>();  ls.Push(root.left); rs.Push(root.right);  while (true) {  if (ls.Count == 0 && rs.Count == 0) break;  if (ls.Count == 0 || rs.Count == 0) return false;  var l = ls.Pop(); var r = rs.Pop();  if (l == null && r == null) continue;  if (l == null || r == null) return false;  if (l.val != r.val) return false;  ls.Push(l.left); ls.Push(l.right);  rs.Push(r.right); rs.Push(r.left);  }  return true;  } | | | | | | **Sort a linked list in *O*(*n* log *n*) time using constant space complexity.** | | | | |
| public Node SortList(Node head) {  if (head == null || head.next == null) return head;  return mergeSort(head);  }  private Node mergeSort(Node head) {  if (head.next == null) return head;  int cnt = 0;  for (var node = head; node != null; node = node.next) cnt++;  var mid = head;  Node tmp = null;  for (var i = 0; i < cnt >> 1; i++) {  tmp = mid;  mid = mid.next;  }  tmp.next = null;  var left = mergeSort(head);  var right = mergeSort(mid);  var dummy = new Node(0);  var cur = dummy;  for (int i = 0; i < cnt; i++) {  if (left != null && (right == null || left.val < right.val)) {  cur.next = left;  left = left.next;  } else {  cur.next = right;  right = right.next;  }  cur = cur.next;  }  return dummy.next;  } | | | | |
| **A sorted array is rotated at some pivot unknown to you beforehand. (i.e., 0 1 2 5 7 might become 5 7 0 1 2). Find the minimum element. Assume no duplicate exists in the array.** | | | | |
| public int FindMin(int[] nums) {  int s = 0, e = nums.Length - 1;  while (s < e) {  int m = s + ((e - s) >> 1);  if (nums[m] < nums[e]) {  e = m;  } else {  s = m + 1;  }  }  return nums[s];  } | | | | |
| **Given a string *s*, return all subsets of s. e.g. s = “abc”,**  **return [ [a,b,c], [a,bc], [ab,c], [abc] ]** | | | | | **Given a string *s*, partition *s* such that every substring of the partition is a palindrome. Return all possible palindrome partitioning of *s*. For example, given *s* = "aab", return [ ["aa","b"], ["a","a","b"] ]** | | | | | |
| List<List<string>> result = new List<List<string>>();  public List<List<string>> Subsets(string s) {  if (string.IsNullOrEmpty(s) == true) return result;  Subsets(s, new List<string>());  return result;  }  private void Subsets(string s, List<string> list) {  if (s.Length == 0) {  result.Add(new List<string>(list));  return;  }  for (int i = 1; i <= s.Length; i++) {  var sub = s.Substring(0, i);  list.Add(sub);  Subsets(s.Substring(i), list);  list.RemoveAt(list.Count - 1);  }  }  **Given two binary strings, return their sum (also a binary string).**  public string AddBinary(string a, string b) {  var result = new char[a.Length > b.Length ? a.Length : b.Length];  var carry = 0;  var ai = a.Length - 1;  var bi = b.Length - 1;  var ri = result.Length - 1;  while (ai >= 0 || bi >= 0) {  var v1 = (ai >= 0) ? a.ElementAt(ai) - '0' : 0;  var v2 = (bi >= 0) ? b.ElementAt(bi) - '0' : 0;  var sum = v1 + v2 + carry;  carry = (sum > 1) ? 1 : 0;  var sumStr = (sum < 2) ? sum + '0' : (sum - 2) + '0';  result[ri--] = (char)sumStr;  ai--;  bi--;  }  if (carry > 0) {  return "1" + new string(result);  }  return new string(result);  } | | | | | List<List<string>> result = new List<List<string>>();  public List<List<string>> Partition(string s) {  if (string.IsNullOrEmpty(s) == true) return result;  partition(s, new List<string>());  return result;  }  private void partition(string s, List<string> list) {  if (s.Length == 0) {  result.Add(new List<string>(list));  return;  }  for (int i = 1; i <= s.Length; i++) {  var sub = s.Substring(0, i);  if (isP(sub) == true) {  list.Add(sub);  partition(s.Substring(i), list);  list.RemoveAt(list.Count - 1);  }  }  }  private bool isP(string s) {  for (int i = 0; i < s.Length >> 1; i++) {  if (s.ElementAt(i) != s.ElementAt(s.Length - 1 - i))  return false;  }  return true;  }  **Given an m x n matrix, if an element is 0, describe how you would set its entire row and column to 0 in constant space. Do it in place.**  Use the first row and column to mark the rows and columns that have element 0. Place two boolean variables to store whether the first row and first column contain a zero.  **Calculate the sum of two integers *a* and *b*, without using + and - operators.**  public int GetSum(int a, int b) {  while (b != 0) {  int c = a & b; //carry  a ^= b; //add  b = c << 1; //a & b << 1 is equivalent to carry!  }  return a;  } | | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Reverse a linked list from position *m* to *n*. Do it in-place and in one-pass. For example: Given 1->2->3->4->5->NULL, *m* = 2 and *n* = 4, return 1->4->3->2->5->NULL. Note: Given *m*, *n* satisfy the following condition: 1 ≤ *m* ≤ *n* ≤ length of list.** | | | **Given a non-negative integer *n* less than or equal to 8, print all possible combinations of *n* number of 1s in a byte string. For example, *n* = 1 returns a total of eight byte strings: {10000000, 01000000, … , 00000001}.** | |
| public Node ReverseBetween(Node head, int m, int n) {  if (head == null) return null;  var dummy = new ListNode(0);  var node = head; var cur = dummy; var i = 1;  for (; i < m; i++) {  cur.next = node;  node = node.next;  cur = cur.next;  }  helper(node, i, n, cur, node);  return dummy.next;  }  private Node helper(Node node, int i, int n, Node start, Node end) {  if (i == n) {  start.next = node;  end.next = node.next;  return node;  }  var tmp = helper(node.next, i + 1, n, start, end);  tmp.next = node;  return node;  } | | | List<string> result = new List<string>();  public List<string> Combinations(int n) {  var bits = new int[8];  Combinations(n, 0, 0, bits);  return result;  }  private void Combinations(int n, int cnt, int i, int[] bits) {  if (n == cnt) {  var sb = new StringBuilder();  foreach (var bit in bits) sb.Append(bit + "");  result.Add(sb.ToString());  return;  }  if (i >= bits.Length) return;  bits[i] = 1;  Combinations(n, cnt + 1, i + 1, bits);  bits[i] = 0;  Combinations(n, cnt, i + 1, bits);  } | |
| **Find the total area covered by two rectilinear rectangles in a 2D plane. Each rectangle is defined by its bottom left corner and top right corner coordinates.** | | | | **Given an array of integers that is already *sorted in ascending order*, find two numbers such that they add up to a specific target number. Return the indices of the two numbers. Each input has exactly one solution.** |
| public int ComputeArea(int A, int B, int C, int D, int E, int F, int G, int H) {  var area1 = (C - A) \* (D - B);  var area2 = (G - E) \* (H - F);  return area1 + area2 - GetOverlap(A, B, C, D, E, F, G, H);  }  private int GetOverlap(int A, int B, int C, int D, int E, int F, int G, int H) {  if (E > C || G < A || F > D || H < B) return 0;  var width = 0; var height = 0;  if (A <= E && G <= C) width = G - E;  else if (E < A && C < G) width = C - A;  else if (G > C) width = C - E;  else width = G - A;  if (B <= F && H <= D) height = H - F;  else if (F < B && D < H) height = D - B;  else if (F < B) height = H - B;  else height = D - F;  return width \* height;  } | | | | Dictionary<int, int> map = new Dictionary<int, int>();  public int[] TwoSum(int[] numbers, int target) {  var result = new int[2];  int i = 0; int j = numbers.Length - 1;  while (i < j) {  var sum = numbers[i] + numbers[j];  if (sum == target) return new int[] { i + 1, j + 1 };  else if (sum > target) j--;  else i++;  }  return result;  } |
| **Given a singly linked list, group all odd nodes together followed by the even nodes where we are talking about the node number and not the value in the nodes. The program should run in O(1) space complexity and O(n) time complexity.** | | **A peak element is an element that is greater than its neighbors. Given an input array where num[i] ≠ num[i+1], find a peak element and return its index. The array may contain multiple peaks, in that case return the index to any one of the peaks. You may imagine that num[-1] = num[n] = -∞.** | | |
| public ListNode OddEvenList(ListNode head) {  if (head == null) return null;  var odd = head;  var evenStart = head.next;  var even = head.next;  while (odd != null && odd.next != null && even.next != null) {  odd.next = odd.next.next;  even.next = even.next.next;  odd = odd.next;  even = even.next;  }  odd.next = evenStart;  return head;  } | | public int FindPeakElement(int[] nums) {  int left = 0, right = nums.Length - 1;  while (left < right) {  int mid1 = (left + right) / 2;  int mid2 = mid1 + 1;  if (nums[mid1] < nums[mid2]) left = mid2;  else right = mid1;  }  return left;  } | | |
| **Implement an iterator over a binary search tree (BST). Implement the constructor and methods next() and hasNext()** | **Store all permutations of a string. For example, the string “123” has permutations**  **"123", "132", "213", "231", "312", "321".** | | | |
| public class BSTIterator {  Stack<TreeNode> stack = new Stack<TreeNode>();  public BSTIterator(TreeNode root) {  while (root != null) {  stack.Push(root);  root = root.left;  }  }  public bool HasNext() {  if (stack.Count == 0) return false;  return true;  }  public int Next() {  var ret = stack.Pop();  var node = ret.right;  while (node != null) {  stack.Push(node);  node = node.left;  }  return ret.val;  }  } | public List<string> GetPermutation(string s) {  List<string> list = new List<string>();  GetPermutation("", s, list);  return list;  }  private void GetPermutation(string left, string right, List<string> list) {  if (right.Length == 0) {  list.Add(left);  return;  }  for (int i = 0; i < right.Length; i++) {  var str = right.ElementAt(i) + right.Substring(0, i) + right.Substring(i + 1);  GetPermutation(left + str.ElementAt(0), str.Substring(1), list);  }  } | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Given a list of unique words, find all pairs of *distinct* indices (i, j) in the given list, so that the concatenation of the two words, i.e. words[i] + words[j] is a palindrome. Given words = ["abcd", "dcba", "lls", "s", "sssll"] return [[0, 1], [1, 0], [3, 2], [2, 4]] . The palindromes are ["dcbaabcd", "abcddcba", "slls", "llssssll"]** | | | |
| List<List<int>> result = new List<List<int>>();  Dictionary<string, int> map = new Dictionary<string, int>();  public List<List<int>> PalindromePairs(string[] words) {  if (words == null || words.Length == 0) return result;  for (int i = 0; i < words.Length; i++)  map.Add(words[i], i);  for (int i = 0; i < words.Length; i++) {  var a = words[i];  var ar = reverse(a);  if (isPalindrome(a) == true) {  if (map.ContainsKey("") == true && map[""] != i) {  add(i, map[""]);  add(map[""], i);  }  }  if (map.ContainsKey(ar) == true && map[ar] != i) {  add(i, map[ar]);  }  for (int j = 1; j < a.Length; j++) {  var l = a.Substring(0, j);  var r = a.Substring(j);  if (isPalindrome(l) == true) {  var rr = reverse(r);  if (map.ContainsKey(rr) == true && map[rr] != i) {  add(map[rr], i);  }  }  if (isPalindrome(r) == true) {  var rl = reverse(l);  if (map.ContainsKey(rl) == true && map[rl] != i) {  add(i, map[rl]);  }  }  }  }  return result;  } | | public string reverse(string s) {  char[] charArray = s.ToCharArray();  Array.Reverse(charArray);  return new string(charArray);  }  private bool isPalindrome(string word) {  int i = 0; int j = word.Length - 1;  while (i < j) {  if (word.ElementAt(i++) != word.ElementAt(j--))  return false;  }  return true;  }  private void add(int i, int j) {  var list = new List<int>();  list.Add(i);  list.Add(j);  result.Add(list);  } | |
| **Given a binary tree, determine if it is a valid binary search tree (BST). All nodes have unique values.** | |
| public bool IsValidBST(TreeNode root) {  return IsValidBST(root, Int32.MinValue, Int32.MaxValue);  }  private bool IsValidBST(TreeNode node, int min, int max) {  if (node == null) return true;  if (min >= node.val || node.val >= max) return false;  return IsValidBST(node.left, min, node.val)  && IsValidBST(node.right, node.val, max);  }  **Given a positive integer *num*, write a function which returns True if *num* is a perfect square else False. Note: Do not use any built-in library function such as sqrt.**  public bool IsPerfectSquare(int num) {  if (num <= 0) return false;  long l = 1; long r = (num >> 1) + 1;  while (l <= r) {  long m = l + ((r - l) >> 1);  long mul = m \* m;  if (mul == num) return true;  else if (mul > num) r = m - 1;  else l = m + 1;  }  return false;  } | |
| **The set [1,2,3,…,*n*] contains a total of *n*! unique permutations. Given *n* and *k*, return the *k*th permutation sequence. *n* will be between 1 and 9 inclusive.** | | **Suppose a sorted array is rotated at some pivot unknown to you beforehand. (i.e., 0 1 2 4 5 might become 4 5 0 1 2). Return the index of a value in the array or -1 if it doesn’t exist.** | |
| public string GetPermutation(int n, int k) {  var str = "";  for (int i = 1; i <= n; i++) str += i;  var result = "";  var s = str;  var p = factorial(n);  --k;  for (int i = n; i > 0; i--) {  p /= i;  var c = k / p;  var t = s.ElementAt(c) + s.Substring(0, c) + s.Substring(c + 1);  s = t.Substring(1);  result += t.Substring(0, 1);  k %= p;  }  return result;  }  private int factorial(int n) {  for (int i = n - 1; i >= 2; i--) n \*= i;  return n;  } | | public int Search(int[] nums, int target) {  int s = 0, e = nums.Length - 1;  while (s <= e) {  int m = s + ((e - s) >> 1);  if (nums[m] == target) return m;  if (nums[s] <= nums[m]) {  if (nums[s] <= target && target < nums[m]) e = m - 1;  else s = m + 1;  } else {  if (nums[m] < target && target <= nums[e]) s = m + 1;  else e = m - 1;  }  }  return -1;  } | |
| **There are *N* children standing in a line. Each child is assigned a rating value. You are giving candies to these children subjected to the following requirements: 1. Each child must have at least one candy. 2. Children with a higher rating get more candies than their neighbors. What is the minimum candies you must give?** | | **Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining. For example, given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.** | |
| public int Candy(int[] ratings) {  if (ratings == null || ratings.Length == 0) return 0;  var r = ratings;  var result = 0;  var left = new int[r.Length];  var right = new int[r.Length];  left[0] = 1;  right[r.Length - 1] = 1;  //scan from left to right  for (int i = 1; i < r.Length; i++)  left[i] = (r[i - 1] < r[i]) ? left[i - 1] + 1 : 1;  //scan from right to left  for (int i = r.Length - 2; i >= 0; i--)  right[i] = (r[i + 1] < r[i]) ? right[i + 1] + 1 : 1;  for (int i = 0; i < r.Length; i++)  result += Math.Max(left[i], right[i]);  return result;  } | | public int Trap(int[] height) {  if (height == null || height.Length == 0) return 0;  var h = height; int max = 0; var result = 0;  var left = new int[h.Length]; var right = new int[h.Length];  //scan from left to right  for (int i = 0; i < h.Length; i++) {  max = Math.Max(max, h[i]);  left[i] = max - h[i];  }  max = 0;  //scan from right to left  for (int i = h.Length - 1; i >= 0; i--) {  max = Math.Max(max, h[i]);  right[i] = max - h[i];  }  for (int i = 0; i < h.Length; i++)  result += Math.Min(left[i], right[i]);  return result;  } | |
| **Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.** | **Given a set of distinct integers, return all possible subsets. The solution set must not contain duplicate subsets. For example, [1,2,3], a solution is: [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]** | | |
| Dictionary<int, UndirectedGraphNode> map =  new Dictionary<int, UndirectedGraphNode>();  public UndirectedGraphNode CloneGraph(UndirectedGraphNode node) {  if (node == null) return null;  return clone(node);  }  private UndirectedGraphNode clone(UndirectedGraphNode node) {  var copy = new UndirectedGraphNode(node.label);  map.Add(node.label, copy);  foreach (var neighbor in node.neighbors) {  if (map.ContainsKey(neighbor.label) == false) {  clone(neighbor);  }  copy.neighbors.Add(map[neighbor.label]);  }  return copy;  } | List<List<int>> result = new List<List<int>>();  public List<List<int>> Subsets(int[] nums) {  if (nums == null || nums.Length == 0) return result;  Subsets(nums, 0, new List<int>());  return result;  }  private void Subsets(int[] nums, int i, List<int> list) {  if (i == nums.Length) {  result.Add(new List<int>(list));  return;  }  list.Add(nums[i]);  Subsets(nums, i + 1, list);  list.RemoveAt(list.Count - 1);  Subsets(nums, i + 1, list);  } | | |
| **Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses. For example, given *n* = 3, a solution set is: ["((()))","(()())","(())()","()(())","()()()"]** | | | **Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*. For example, If *n* = 4 and *k* = 2, a solution is: [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]** |
| List<string> result = new List<string>();  public List<string> GenerateParenthesis(int n) {  GenerateParenthesis("", n, n);  return result;  }  private void GenerateParenthesis(string s, int left, int right) {  if (left > right) return;  if (left == 0 && right == 0) {  result.Add(s);  return;  }  if (left > 0) GenerateParenthesis(s + "(", left - 1, right);  if (right > 0) GenerateParenthesis(s + ")", left, right - 1);  } | | | List<List<int>> result = new List<List<int>>();  public List<List<int>> Combine(int n, int k) {  Combine(n, k, 1, new List<int>());  return result;  }  private void Combine(int n, int k, int a, List<int> list) {  if (k == 0) {  result.Add(new List<int>(list));  return;  }  for (int i = a; i <= n; i++) {  list.Add(i);  Combine(n, k - 1, i + 1, list);  list.RemoveAt(list.Count - 1);  }  } |

|  |  |
| --- | --- |
| **Given an array of integers A and let *n* to be its length. Assume Bk to be an array obtained by rotating the array A *k* positions clock-wise, we define a "rotation function" F on A as follow: F(k) = 0 \* Bk[0] + 1 \* Bk[1] + ... + (n-1) \* Bk[n-1]. Calculate the maximum value of F(0), F(1), ..., F(n-1). A = [4, 3, 2, 6]**  F(0) = (0 \* 4) + (1 \* 3) + (2 \* 2) + (3 \* 6) = 0 + 3 + 4 + 18 = 25  F(1) = (0 \* 6) + (1 \* 4) + (2 \* 3) + (3 \* 2) = 0 + 4 + 6 + 6 = 16  F(2) = (0 \* 2) + (1 \* 6) + (2 \* 4) + (3 \* 3) = 0 + 6 + 8 + 9 = 23  F(3) = (0 \* 3) + (1 \* 2) + (2 \* 6) + (3 \* 4) = 0 + 2 + 12 + 12 = 26  So the maximum value of F(0), F(1), F(2), F(3) is F(3) = 26.  public int MaxRotateFunction(int[] A) {  if (A == null || A.Length == 0) return 0;  var cand = 0;  var sum = 0;  var n = A.Length;  for (int i = 0; i < n; i++) {  cand += (i \* A[i]);  sum += A[i];  }  var max = cand;  for (int i = 0; i < n - 1; i++) {  cand = cand + sum - n \* A[n - i - 1];  max = Math.Max(cand, max);  }  return max;  } | **Given a binary tree, determine if it is height-balanced. For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.** |
| public bool IsBalanced(TreeNode root) {  if (root == null) return true;  var right = getMaxDepth(root.right);  var left = getMaxDepth(root.left);  if (Math.Abs(right - left) > 1) return false;    return IsBalanced(root.left) && IsBalanced(root.right);  }    private int getMaxDepth(TreeNode node) {  if (node == null) return 0;  return 1 + Math.Max(getMaxDepth(node.left), getMaxDepth(node.right));  } |
| **Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level). For example: Given binary tree [3,9,20,null,null,15,7], return its level order traversal as: [ [3], [9,20], [15,7] ]**  List<IList<int>> result = new List<IList<int>>();  public List<IList<int>> LevelOrder(TreeNode root) {  if (root == null) return result;  var cur = new Queue<TreeNode>();  var next = new Queue<TreeNode>();  var list = new List<int>();  cur.Enqueue(root);  while (cur.Count > 0) {  var node = cur.Dequeue();  if (node.left != null) next.Enqueue(node.left);  if (node.right != null) next.Enqueue(node.right);  list.Add(node.val);  if (cur.Count == 0) {  result.Add(list);  cur = next;  next = new Queue<TreeNode>();  list = new List<int>();  }  }  return result;  } |
| **Find the sum of all left leaves in a given binary tree.** |
| public int SumOfLeftLeaves(TreeNode root) {  if (root == null) return 0;  return sum(root.left, true) + sum(root.right, false);  }  private int sum(TreeNode node, bool isLeft) {  if (node == null) return 0;  if (isLeft == true && node.left == null && node.right == null) {  return node.val;  }  return sum(node.left, true) + sum(node.right, false);  } |

|  |  |
| --- | --- |
| **Given a collection of distinct numbers, return all possible permutations. For example, [1,2,3] have the following permutations: [ [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1] ]** |  |
| List<IList<int>> result = new List<IList<int>>();  public List<IList<int>> Permute(int[] nums) {  Permute(nums, 0);  return result;  }  private void Permute(int[] nums, int a) {  if (a == nums.Length) {  var list = new List<int>();  for (int i = 0; i < nums.Length; i++) list.Add(nums[i]);  result.Add(list);  return;  }  for (int i = a; i < nums.Length; i++) {  swap(nums, a, i);  Permute(nums, a + 1);  swap(nums, a, i);  }  }  private void swap(int[] nums, int a, int b) {  var tmp = nums[a];  nums[a] = nums[b];  nums[b] = tmp;  } |  |