|  |  |
| --- | --- |
| **Merge Sort**  **Time Complexity: O(nlogn)** in all 3 cases (worst, average and best)  **Auxiliary Space:** **O(n)** | **Quick Sort**  **Time Complexity:**  **O(nlogn)** in best case  **O(N2)** in worst case  **Auxiliary Space:** **no extra space needed** |
| public class MyMergeSort {  **private int[] array;**  **private int[] tempMergArr;**  **private int length;**  **public void sort(int arr[]) {**          this.array = arr;          this.length = arr.length;          this.tempMergArr = new int[length];          doMergeSort(0, length - 1);      }    **private void doMergeSort(int low, int high) {**          if (low < high) {              int mid = low + (high - low) / 2;              // Below step sorts the left side of the array              doMergeSort(low, mid);    // Below step sorts the right side of the array              doMergeSort(mid + 1, high);    // Now merge both sides              mergeParts(low, mid, high);          }      }    **private void mergeParts(int low, int mid, int high) {**          for (int i = low; i <= high; i++) {              tempMergArr[i] = array[i];          }          int i = low;          int j = mid + 1;          int k = low;          while (i <= mid && j <= high) {              if (tempMergArr[i] <= tempMergArr[j]) {                  array[k] = tempMergArr[i];                  i++;              } else {                  array[k] = tempMergArr[j];                  j++;              }              k++;          }          while (i <= mid) {              array[k] = tempMergArr[i];              k++;              i++;          }        }  }     public static void main(String a[]){          int[] inputArr = {45,23,11,89,77,98,4,28,65,43};          MyMergeSort mms = new MyMergeSort();          mms.sort(inputArr);          for(int i:inputArr){              System.out.print(i);              System.out.print(" ");          }      } | public class MyQuickSort {    **private int[] array;**  **private int length**;    **public void sort(int[] arr){**  this.array = arr;  this.length = arr.length;  this.quickSort(0, length-1);  }    **public void quickSort(int low, int high){**  int i = low;  int j = high;    int pivotIdx = low + (high - low) / 2; //Middle Element as pivot  int pivot = array[pivotIdx];    while(i <= j) {  while(array[i] < pivot){  i++;  }  while(array[j] > pivot){  j--;  }    if(i <= j){  swap(i,j);  i++;  j--;  }  }    if(low < j)  quickSort(low, j);  if(i < high)  quickSort(i, high);    }    **public void swap(int i, int j){**  **int temp = array[i];**  **array[i] = array[j];**  **array[j] = temp;**  **}**    **public void printArr(int[] arr){**  **for(int i : arr)**  **System.out.print(i + " ");**  **System.out.println();**  **}**    public static void main(String s[]){  MyQuickSort mqs = new MyQuickSort();  int[] input = {23,56,1,34,4};  mqs.printArr(input);  mqs.sort(input);  mqs.printArr(input);    } |

|  |  |
| --- | --- |
| **STACK**  public class **MyArrayStack**<T> implements Stack<T> {    public int **stackSize**;  public T[] **arrayStack**;  public int **top**;      public **MyArrayStack**(int size) {  super();  this.arrayStack = (T[]) new Object[size];  this.stackSize = size;  this.top = -1;  }  public **boolean push(T value)** {  if(this.isStackFull())  this.increaseCapacity();  this.arrayStack[++top] = value;  System.out.println("Push : "+value);  return true;  }  public T **pop()** {  if(isStackEmpty())  System.out.println("Stack is empty");  T value = this.arrayStack[top];  top--;  System.out.println("Pop : "+value);  return value;  }  public T **peek()** {  System.out.println("Pop : "+this.arrayStack[top]);  return this.arrayStack[top];  } | public void **increaseCapacity()** {  this.stackSize = this.stackSize \* 2;  T[] newArrayStack = (T[]) new Object[stackSize];    for(int i=0; i< arrayStack.length;i++)  newArrayStack[i] = arrayStack[i];    this.arrayStack = newArrayStack;  System.out.println("Stack is full - increasing the size ");  }    public **boolean isStackFull()**{  return top == stackSize-1;  }    public **boolean isStackEmpty()**{  return top == -1;  }  public static void main(String a[]){  MyArrayStack<String> stringStack = new MyArrayStack<String>(2);  stringStack.push("java2novice");  stringStack.push("1");  stringStack.push("2");  stringStack.push("3");  stringStack.pop();  stringStack.peek();  stringStack.pop();  stringStack.pop();  stringStack.peek();    MyArrayStack<Integer> integerStack = new MyArrayStack<Integer>(2);  integerStack.push(23);  }  }  **public interface Stack<T> {**  **public boolean push(T value);**  **public T pop();**  **public T peek();**  **}** |

|  |  |
| --- | --- |
| **Stack using Java Util**  import java.util.Stack;  public static void main(String[] args) {  // Creating a Stack  Stack<String> stackOfCards = new Stack<>();  // Pushing new items to the Stack  stackOfCards.push("Jack");  stackOfCards.push("Queen");  stackOfCards.push("King");  stackOfCards.push("Ace");  System.out.println("Stack => " + stackOfCards);  System.out.println();  // Popping items from the Stack  String cardAtTop = stackOfCards.pop(); // Throws EmptyStackException if the stack is empty  System.out.println("Stack.pop() => " + cardAtTop);  System.out.println("Current Stack => " + stackOfCards);  System.out.println();  // Get the item at the top of the stack without removing it  cardAtTop = stackOfCards.peek();  System.out.println("Stack.peek() => " + cardAtTop);  System.out.println("Current Stack => " + stackOfCards);    System.out.println(stackOfCards.isEmpty());  System.out.println(stackOfCards.search("Jack"));  System.out.println(stackOfCards.search("Queen"));  System.out.println(stackOfCards.search("King"));  System.out.println(stackOfCards.search("Kingdd"));  } | **QUEUE Using Java Util**  import java.util.Queue;  import java.util.LinkedList;  public static void main(String[] args) {  // Create and initialize a Queue using a LinkedList  Queue<String> waitingQueue = new LinkedList<>();  // Adding new elements to the Queue (The Enqueue operation)  waitingQueue.add("Rajeev");  waitingQueue.add("Chris");  waitingQueue.add("John");  waitingQueue.add("Mark");  waitingQueue.add("Steven");  System.out.println("WaitingQueue : " + waitingQueue);  // Removing an element from the Queue using remove() (The Dequeue operation)  // The remove() method throws NoSuchElementException if the Queue is empty  String name = waitingQueue.remove();  System.out.println("Removed from WaitingQueue : " + name + " | New WaitingQueue : " + waitingQueue);  // Removing an element from the Queue using poll()  // The poll() method is similar to remove() except that it returns null if the Queue is empty.  name = waitingQueue.poll();  System.out.println("Removed from WaitingQueue : " + name + " | New WaitingQueue : " + waitingQueue);  }  // Check is a Queue is empty  System.out.println("is waitingQueue empty? : " + waitingQueue.isEmpty());  // Find the size of the Queue  System.out.println("Size of waitingQueue : " + waitingQueue.size());  // Check if the Queue contains an element  name = "John";  if (waitingQueue.contains(name)) {  System.out.println("WaitingQueue contains " + name);  } else {  System.out.println("Waiting Queue doesn't contain " + name);  }  // Get the element at the front of the Queue without removing it using element()  // The element() method throws NoSuchElementException if the Queue is empty  String firstPersonInTheWaitingQueue = waitingQueue.element();  System.out.println("First Person in the Waiting Queue (element()) : " + firstPersonInTheWaitingQueue);  // Get the element at the front of the Queue without removing it using peek()  // The peek() method is similar to element() except that it returns null if the Queue is empty  firstPersonInTheWaitingQueue = waitingQueue.peek();  System.out.println("First Person in the Waiting Queue : " + firstPersonInTheWaitingQueue);  //Iterate through the list  for(String s: waitingQueue){  System.out.println(s);  } |

|  |  |
| --- | --- |
| **QUEUE** |  |

|  |  |
| --- | --- |
| **Linkedlist Using Java Util** |  |

**Linear data structure**

|  |  |  |
| --- | --- | --- |
| **LINKED LIST (Singly)**  public class **SinglyLinkedList**<E>  implements LinkedList<E> {  public SinglyLinkedList() {  super();  }    **private int currentSize;**  **private Node<E> head;**  **private Node<E> tail;**  public void **addFirst**(E obj) {  Node<E> newNode = new Node<E>(obj);  if(head == null){  head = newNode;  tail = newNode;  } else {  newNode.next = head;  head = newNode;  }  currentSize++;  }  public void **addLast**(E obj) {  Node<E> newNode = new Node<E>(obj);  System.out.println("Adding Last : " +obj);  if(head == null){  head = newNode;  tail = newNode;  } else {  tail.next = newNode;  tail = newNode;  }  currentSize++;  }  public E **removeFirst**() {  if(head == null) {  return null;  }  E data = head.data;  System.out.println("Removing First : " +data);  if(head == tail)  head = tail = null;  else  head = head.next;  currentSize--;  return data;  }    public int **size**() {  return currentSize;  } | public E **removeLast**() {  if(head == null){  return null;  }  if(head == tail){  E data = head.data;  head = tail = null;  currentSize--;  System.out.println("Removed Last :" + data);  return data;  }else {  Node<E> current= head, prev = null;  while(current != tail){  prev = current;  current = current.next;  }  E data = current.data;  prev.next = null;  tail = prev;  currentSize--;  System.out.println("Removed Last :" + data);  return data;  }  }  public E **remove**(E obj) {  Node<E> current = head, prev = null;  while(current != null){  if(((Comparable<E>)obj).compareTo(current.data) == 0){  if(current == head)  return this.removeFirst();  if(current == tail)  return this.removeLast();    prev.next = current.next;  currentSize--;  return current.data;  }  prev = current;  current = current.next;  }  return null;  }  public Boolean **contains**(E obj) {  Node<E> current = head;  while(current != null){  if(((Comparable<E>)obj).compareTo(current.data) == 0){  return true;  }  current = current.next;  }  return false;  }  public void **traverse()** {  Node<E> temp = head;  while(true){  if(temp == null){  break;  }  System.out.print(temp.data);  temp = temp.next;  } } | **//Inner Class**  **private class Node<T> implements Comparable<T> {**  **private T data;**  **private Node<T> next;**  **//Constructor**  **public Node(T value){**  **this.data = value;**  **}**    **@Override**  **public int compareTo(T obj) {**  **if(obj == this.data)**  **return 0;**  **else**  **return 1;**  **}**  **}**  public E **peekFirst**() {  if(head == null)  return null;  return head.data;  }  public E **peekLast**() {  if(tail == null)  return null;  return tail.data;  }  public static void main(String s[]){  SinglyLinkedList<Integer> intList =  new SinglyLinkedList<Integer>();  intList.addFirst(3);  intList.addLast(180);  intList.removeFirst();  intList.removeLast();  intList.remove(180);  intList.contains(1);  intList.peekFirst();  intList.peekLast();  intList.traverse();  intList.size();  }  }  public **interface LinkedList**<E> {  void addFirst(E obj);  void addLast(E obj);  E removeFirst();  E removeLast();  E remove(E obj);  Boolean contains(E obj);  E peekFirst();  E peekLast();  int size();  void traverse();  } |

|  |  |  |
| --- | --- | --- |
| **LINKED LIST (Doubly)**  public class **DoublyLinkedList**<E> implements  LinkedList<E> {    **private Node<E> head;**  **private Node<E> tail;**  **private int currentSize;**    public DoublyLinkedList() {  super();  }  public void **addFirst**(E obj) {  Node<E> newNode = new Node<E>(obj,head,null);  if(head != null){  head.prev = newNode;  }  head = newNode;  if(tail == null)  tail = newNode;  currentSize++;  }  public void **addLast**(E obj) {  Node<E> newNode = new Node<E>(obj,null,tail);  if(tail != null) {  tail.next = newNode;  }  tail = newNode;  if(head == null)  head = newNode;  currentSize++;  }  public E **removeFirst**() {  if(currentSize == 0) return null;    Node<E> temp = head;  head = head.next;  head.prev = null;  currentSize--;  System.out.println("Removed: "+temp.data);  return temp.data;    }  public E **removeLast**() {  if(currentSize == 0) return null;    Node<E> temp = tail;  tail = tail.prev;  tail.next = null;  currentSize--;  System.out.println("Removed: "+temp.data);  return temp.data;  } | public E **remove**(E obj) {  Node<E> current = head;  while(current != null){  if(((Comparable<E>)obj).compareTo(current.data)  == 0){    if(current == head)  return this.removeFirst();  else if(current == tail)  return this.removeLast();    current.prev.next = current.next;  current.next.prev = current.prev;  currentSize--;  System.out.println("Removed: "+current.data);  return current.data;  }  current = current.next;  }  System.out.println("Not found");  return null;  }  public Boolean **contains**(E obj) {  Node<E> current = head;  while(current != null){  if(((Comparable<E>) obj).compareTo(current.data)  == 0){  return true;  }  current = current.next;  }  return false;  }  public E **peekFirst**() {  if(head == null)  return null;  return head.data;  }  public E **peekLast**() {  if(tail == null)  return null;  return tail.data;  }  public int **size**() {  return this.currentSize;  }  public void **traverse**() {  Node<E> current = head;  while(current != null){  System.out.print(current.data + " ");  current = current.next;  }  }  **public void traverseBack(){**  Node<E> current = tail;  while(current!= null){  System.out.print(current.data + " ");  current = current.prev;  }  } | **//Node - Inner Class for doubly linkedlist**  **private class Node<T> implements Comparable<T> {**    **private T data;**  **private Node<T> next;**  **private Node<T> prev;**  **//Constructor**  **public Node(T value,**  **Node nextRef, Node prevRef){**  **this.data = value;**  **this.next = nextRef;**  **this.prev = prevRef;**  **}**    **public int compareTo(T obj) {**  **if(obj == this.data)**  **return 0;**  **else**  **return 1;**  **}**  **}**  public static void main(String s[]){  DoublyLinkedList <Integer> intList =  new DoublyLinkedList <Integer>();  intList.addFirst(3);  intList.addLast(180);  intList.removeFirst();  intList.removeLast();  intList.remove(180);  intList.contains(1);  intList.peekFirst();  intList.peekLast();  intList.traverse();  intList.traverseBack();  intList.size();  }  }  public **interface LinkedList**<E> {  void addFirst(E obj);  void addLast(E obj);  E removeFirst();  E removeLast();  E remove(E obj);  Boolean contains(E obj);  E peekFirst();  E peekLast();  int size();  void traverse();  } |

|  |  |
| --- | --- |
| **Binary Search Tree – Non-linear data structure**   Time Complexity: O(log n),  public class **BinarySearchTree**<E extends Comparable<E>> implements Tree<E>{  **private Node<E> root;**    public BinarySearchTree() {  super();  }  @Override  **public void insert(E data) {**  System.out.println("[input: "+data+"]");  if(root == null){  root = new Node(data);  System.out.println(" -> inserted : "+data);  return ;  }  insertNode(root,data);  System.out.print(" -> inserted: "+data);  System.out.println();  }  @Override  **public boolean isEmpty() {**  return this.root == null;  }  @Override  **public Node insertNode(Node node, E data) {**  Node<E> tempNode = null;  System.out.print(" ->"+node.getData());  int compareResult = node.getData().compareTo(data); // comparing node value with data  if(compareResult == 0 || compareResult > 0){  System.out.print("[L]");  if(node.getLeft() == null){  node.setLeft(new Node(data));  return node.getLeft();  } else {  tempNode = node.getLeft();  }  } else {  System.out.print("[R]");  if(node.getRight() == null){  node.setRight(new Node(data));  return node.getRight();  } else {  tempNode = node.getRight();  }  }  return insertNode(tempNode, data);  }  } | **public static void main(String s[]){**  BinarySearchTree<Integer> bst = new BinarySearchTree<Integer>();  bst.insert(10);  bst.insert(20);  bst.insert(21);  bst.insert(8);  bst.insert(6);  bst.insert(16);  bst.insert(23);  }  public **interface Tree** <E> {  public void insert(E data);  public boolean isEmpty();  public Node insertNode(Node node, E data);  }  import java.util.Comparator;  public **class Node**<E extends Comparable<E>> {  **private Node<E> left;**  **private Node<E> right;**  **private E data;**  **//Constructor**  **public Node(E value) {**  **this.data = value;**  **}**  public int compareTo(E obj){  if(this.data == obj)  return 0;  else  return 1;  }  public void setLeft(Node left) {  this.left = left;  }  public Node getLeft() {  return left;  }  public void setRight(Node right) {  this.right = right;  }  public Node getRight() {  return right;  }  public void setData(E data) {  this.data = data;  }  public E getData() {  return data;  }  } |

|  |  |
| --- | --- |
| **Height, Minvalue and MaxValue of BST**  **public Integer findHeight(){**  return this.getNodeHeight(this.root);  }    public Integer **getNodeHeight**(Node<E> node){  if(node == null)  return -1;    return Math.max(getNodeHeight(node.getLeft()),  getNodeHeight(node.getRight())) + 1;  } | **public E findMinValue(){ return getMinValue(this.root); }**  **public E getMinValue(Node<E> node){**  if(node.getLeft() != null){  return (E)this.getMinValue(node.getLeft());  }  return node.getData();  }    **public E findMaxValue(){ return getMaxValue(this.root); }**  **public E getMaxValue(Node<E> node){**  if(node.getRight()!= null){  return (E)this.getMaxValue(node.getRight());  }  return node.getData();} |

Tree always has**n-1**edges.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Pre-order traversal (Depth first Traversal)**   |  |  | | --- | --- | | Binary Search Tree (pre-order traversal) | In pre-order traversal- first we visit the  **current node, then left subtree and then right subtree**   1. Start with root node. 2. Check if the current node is empty / null. 3. Display the data part of the root (or current node). 4. Traverse the left subtree by recursively calling the pre-order function. 5. Traverse the right subtree by recursively calling the pre-order function. |   //Pre-Order Traversal (Depth First Traversal)  public void preOrderTraversal(){  this.doPreOrderTraversal(this.root);  }    public void doPreOrderTraversal(Node<E> node){  if(node == null)  return;  System.out.print(node.getData() + " ");  doPreOrderTraversal(node.getLeft()); //recursive  doPreOrderTraversal(node.getRight());//recursive  } | **In-order traversal (Depth first Traversal)**   |  |  | | --- | --- | | Binary Search Tree (in-order traversal) | In-order traversal - first we visit the  **left subtree, then current node and then right subtree**   1. Start with root node. 2. Check if the current node is empty / null. 3. Traverse the left subtree by recursively calling the in-order function. 4. Display the data part of the root (or current node). 5. Traverse the right subtree by recursively calling the in-order function. |   //In-Order Traversal (Depth First Traversal)  public void inOrderTraversal(){  this.doInOrderTraversal(this.root);  }    public void doInOrderTraversal(Node<E> node){  if(node == null)  return;  doInOrderTraversal(node.getLeft());//recursive  System.out.print(node.getData() + " ");  doInOrderTraversal(node.getRight());//recursive  } |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Post-order traversal (Depth first Traversal)**   |  |  | | --- | --- | | Binary Search Tree (post-order traversal) | In post-order traversal, first we visit  **the left subtree, then the right subtree, and then current node.**   1. Start with root node. 2. Check if the current node is empty / null. 3. Traverse the left subtree by recursively calling the post-order function. 4. Traverse the right subtree by recursively calling the post-order function. 5. Display the data part of the root (or current node). |     //Post-Order Traversal (Depth First Traversal)  public void postOrderTraversal(){  this.doPostOrderTraversal(this.root);  }  public void doPostOrderTraversal(Node<E> node){  if(node == null)  return;  doPostOrderTraversal(node.getLeft()); //recursive  doPostOrderTraversal(node.getRight()); //recursive  System.out.print(node.getData() + " ");  }  public static void main(String s[]){  BinarySearchTree<Integer> bst = new BinarySearchTree<Integer>();  bst.insert(10); bst.insert(20);bst.insert(21); bst.insert(8); bst.insert(6);  bst.insert(16); bst.insert(23);  System.out.println("Is Empty: "+bst.isEmpty());  System.out.println("Min Value : " +bst.findMinValue());  System.out.println("Max Value : "+bst.findMaxValue());  System.out.println("Tree Height :" +bst.findHeight());  System.out.print("\nPre-Order Traversal:"); bst.preOrderTraversal();  System.out.print("\nIn-Order Traversal:"); bst.inOrderTraversal();  System.out.print("\nPost-Order Traversal:"); bst.postOrderTraversal();}  System.out.print("Level Order Traversal:"); bst.levelOrderTraversal(); | **Level Order Traversal (Breadth First Traversal)**   |  |  | | --- | --- | | Binary Search Tree (Level order traversal) | 1. Since we have to traverse in same level until we cover all nodes in the same level, it is hard to keep the node references, we will maintain a queue for storing all discovered nodes. 2. Initially the queue will be empty. 3. If the root node is not null, push it to the queue. 4. Use a while loop to visit all nodes in the queue until the queue is empty. 5. Inside while loop, pop out the node. 6. Add the left node to the queue if it is not null. 7. Add the right node to the queue if it is not null. 8. Read the node data and display it. 9. Repeat from step 5, until the queue is empty. |   public void **levelOrderTraversal**(){  Queue<Node> discoveredNodeQueue = new LinkedList<Node>();  if(root == null){  System.out.println("The Tree is Empty");  return;  }  discoveredNodeQueue.add(this.root);  //while queue is not empty  while(!discoveredNodeQueue.isEmpty()) {  Node tempNode = discoveredNodeQueue.remove();  if(tempNode.getLeft() != null){  discoveredNodeQueue.add(tempNode.getLeft());  }  if(tempNode.getRight() != null){  discoveredNodeQueue.add(tempNode.getRight());  }  System.out.print(tempNode.getData() + " ");  }  } |

|  |  |
| --- | --- |
| Delete a node from Binary Search Tree //Delete a Node  public void delete(E data){  deleteNode(this.root, data);  }    public Node deleteNode(Node<E> node, E data){  if(node == null)  return null;  int compareResult = data.compareTo(node.getData());  System.out.println(compareResult);  if(compareResult == -1){ // data is less than node.getData()  node.setLeft(deleteNode(node.getLeft(),data));  } else if (compareResult == 1) { // data is greater than node.getData()  node.setRight(deleteNode(node.getRight(),data));  } else {  // node with no leaf nodes  if(node.getLeft() == null && node.getRight() == null){  System.out.println("Deleting: " +data);  return null;  } else if (node.getLeft() == null){  // node with one node (no left node)  System.out.println("Deleting: "+data);  return node.getRight();  } else if (node.getRight() == null){  // node with one node (no right node)  System.out.println("Deleting: "+data);  return node.getLeft();  } else {  //node with two nodes,  //search for min number in right subtree  E min = (E)this.getMinValue(node.getRight());  node.setData(min);  node.setRight(deleteNode(node.getRight(), min));  System.out.println("Deleting: "+data);  }  }  return node;  } | Deleting a node from Binary search tree is little complicated compare to inserting a node. It includes two steps:   1. Search the node with given value. 2. Delete the node.   The algorithm has **3 cases** while deleting node:   1. Node to be deleted has is a leaf node (no children). 2. Node to be deleted has one child (either left or right child node). 3. Node to be deleted has two nodes.   We will use simple recursion to find the node and delete it from the tree.  Here is the steps to delete a node from binary search tree:  **Case 1: Node to be deleted is a leaf node (no children).**   1. This is very simple implementation. First find the node reference with given value. 2. Set corresponding link of the parent node to null. With this the node to be deleted lost its connectivity and eligible for garbage collection.   **Case 2: Node to be deleted has one child (either left or right child node).**   1. First find the node reference with given value. 2. Take the reference of the child node and assign its reference to the corresponding link of the parent node. With this the node to be deleted lost its connectivity and eligible for garbage collection.   **Case 3: Node to be deleted has two nodes.**   1. It is little complicated process. 2. First find the node reference with given value. 3. Find the minimum/maximum value of the right/left sub tree. 4. Replace the node value with the minimum/maximum value. 5. Now delete the minimum/maximum value from the nodes right/left sub tree. |

|  |  |
| --- | --- |
| **Graph Data Structure**  A Graph is a non-linear data structure consisting of nodes and edges.  The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.  A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.  https://www.geeksforgeeks.org/wp-content/uploads/undirectedgraph.png | public class GraphImpl {  public GraphImpl() {  super();  }    static class Graph {  int noOfVertices;  LinkedList<Integer> adjListArray[];    Graph(int noOfVertices){  this.noOfVertices = noOfVertices;  adjListArray = new LinkedList[noOfVertices];    for(int i=0;i<adjListArray.length;i++ ){  adjListArray[i] = new LinkedList<Integer>();  }  }  }    public static void addEdge(Graph graph, int src, int dest) {  graph.adjListArray[src].add(dest);  graph.adjListArray[dest].add(src); // for undirected graph  }    public static void printGraph(Graph graph){  for(int i=0; i< graph.noOfVertices;i++){  System.out.println("Adjacency list of vertex "+ i);  System.out.print("head");  for(Integer pCrawl: graph.adjListArray[i]){  System.out.print(" -> "+pCrawl);  }  System.out.println("\n");  }  }    public static void main(String s[]){  int vertices = 5;  Graph graph = new Graph(vertices);  addEdge(graph, 0, 1);  addEdge(graph, 0, 4);  addEdge(graph, 1, 2);  addEdge(graph, 1, 3);  addEdge(graph, 1, 4);  addEdge(graph, 2, 3);  addEdge(graph, 3, 4);    // print the adjacency list representation of  // the above graph  printGraph(graph);  }  } |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |
| --- |
| 1. CoinDenominations - <https://www.geeksforgeeks.org/coin-change-dp-7/> 2. Count\_Ones\_In\_Integer - <https://www.geeksforgeeks.org/count-set-bits-in-an-integer/> 3. Dutch\_National\_Flag- imp - <https://www.youtube.com/watch?v=CNVN76UWpBo> 4. Find\_Two\_Elements\_with\_Given\_Sum - <https://www.geeksforgeeks.org/given-an-array-a-and-a-number-x-check-for-pair-in-a-with-sum-as-x/> 5. Find\_Median\_Among\_Arrays - <https://www.geeksforgeeks.org/median-two-sorted-arrays-different-sizes-ologminn-m/> 6. Fix\_Sorted\_Arrays - <https://www.geeksforgeeks.org/sort-an-almost-sorted-array-where-only-two-elements-are-swapped/> 7. Int\_to\_English - imp - <https://www.programcreek.com/2014/05/leetcode-integer-to-english-words-java/> 8. Intersection\_of\_two\_sorted\_arrays\_of\_integers - <https://www.geeksforgeeks.org/union-and-intersection-of-two-sorted-arrays-2/> 9. List\_Max\_Contiguous\_Sum - <https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/> 10. Math\_Int\_Divide\_without\_div\_operator - <https://www.geeksforgeeks.org/divide-two-integers-without-using-multiplication-division-mod-operator/> 11. Monotonic\_Array - imp - <https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n-simple-implementation/> 12. PalindromeGenerator - <https://www.geeksforgeeks.org/print-all-palindrome-permutations-of-a-string/>   <https://www.geeksforgeeks.org/print-palindromic-permutations-given-string-alphabetic-order/>     1. Palindrome - <https://www.geeksforgeeks.org/string-palindrome/> 2. Phone\_Number\_to\_String – imp - <https://www.programcreek.com/2014/04/leetcode-letter-combinations-of-a-phone-number-java/>   <https://www.geeksforgeeks.org/find-possible-words-phone-digits/>   1. Points\_Nearest - very imp - <https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/>   <https://www.geeksforgeeks.org/closest-pair-of-points-onlogn-implementation/>   1. Remove\_Zeros\_From\_Array - imp - <https://www.geeksforgeeks.org/remove-leading-zeros-from-an-array/> [search more] 2. Sorted\_Iterator\_over\_K\_Sorted\_Lists - <https://www.geeksforgeeks.org/merge-k-sorted-arrays/> 3. String\_Remove\_Duplicate\_Chars - imp - <https://www.geeksforgeeks.org/remove-duplicates-from-a-given-string/> 4. Task\_Scheduler - very imp - <https://www.geeksforgeeks.org/scheduling-priority-tasks-limited-time-minimizing-loss/> [search more]   https://leetcode.com/articles/task-scheduler/   1. MinimumWordBreaking - <https://www.geeksforgeeks.org/minimum-word-break/> |

Fibonacci

Prime numbers

Factorial

Valid Anagram

|  |  |  |  |
| --- | --- | --- | --- |
| **1.Coin Denominations**  [**https://www.geeksforgeeks.org/coin-change-dp-7/**](https://www.geeksforgeeks.org/coin-change-dp-7/)  public class CoinDenomination {  public CoinDenomination() {  super();  }  // Returns the count of ways we can  // sum S[0...m-1] coins to get sum n  static int count( int S[], int m, int n )  {  // If n is 0 then there is 1 solution  // (do not include any coin)  if (n == 0)  return 1;    // If n is less than 0 then no  // solution exists  if (n < 0)  return 0;    // If there are no coins and n  // is greater than 0, then no  // solution exist  if (m <=0 && n >= 1)  return 0;    // count is sum of solutions (i)  // including S[m-1] (ii) excluding S[m-1]  return count( S, m - 1, n ) +  count( S, m, n-S[m-1] );  }    // Driver program to test above function  public static void main(String[] args) {  int arr[] = {1, 2, 3};  int m = arr.length;  System.out.println( count(arr, m, 4));  }      } | **2. Count\_Ones\_In\_Integer**  <https://www.geeksforgeeks.org/count-set-bits-in-an-integer/>  **Count set bits in an integer**  Write an efficient program to count number of 1s in  binary representation of an integer.  Examples :   |  |  | | --- | --- | | Input : n = 6  Output : 2  Binary representation of 6 is 110  and has 2 set bits | Input : n = 13  Output : 3  Binary representation of 11 is 1101  and has 3 set bits |   public class CountSetBits {  public CountSetBits() {  super();  }    public static int countSetBits(int n){  int count = 0;  while(n > 0){  n = n & (n -1);  count++;  }  return count;  }  public static void main(String s[]){  System.out.println(countSetBits(9));  }  }  **Brian Kernighan’s Algorithm: Time Complexity:** O(logn)  Subtraction of 1 from a number toggles all the bits (from right to left) till the rightmost set bit (including the rightmost set bit). So if we subtract a number by 1 and do bitwise & with itself (n & (n-1)), we unset the rightmost set bit. If we do n & (n-1) in a loop and count the no of times loop executes we get the set bit count.  The beauty of this solution is the number of times it loops is equal to the number of set bits in a given integer.    1. Initialize count: = 0  2. If integer n is not zero  (a) Do bitwise & with (n-1) and assign the value back to n  n: = n&(n-1)  (b) Increment count by 1  (c) go to step 2  3. Else return count |

|  |  |
| --- | --- |
| **3. Dutch National Flag (3-Way Partition)**  <https://www.geeksforgeeks.org/sort-an-array-of-0s-1s-and-2s/>  public class DutchNationalFlag {  public DutchNationalFlag() {  super();  }  public static void createDutchNationalFlag(int[] input){  int low = 0, mid=0 ;  int high = input.length -1;  while(mid <= high) {  if(input[mid] == 0){  swap(input, low , mid);  low++; mid++;  } else if(input[mid] == 1){  mid++;  }else if(input[mid] == 2){  swap(input,mid,high);  high--;  }  }  }    public static void swap(int[] input, int i, int j){  int temp = input[i];  input[i] = input[j];  input[j] = temp;  }  public static void main(String s[]){  int[] input= {1,2,1,2,0,0,1,2,0};  createDutchNationalFlag(input);  for(int i : input)  System.out.print(i+" ");  }  }  We can change the swap function to do a check that the values being swapped are same or not, if not same, then only swap values. | **4. Find\_Two\_Elements\_with\_Given\_Sum**  <https://www.geeksforgeeks.org/given-an-array-a-and-a-number-x-check-for-pair-in-a-with-sum-as-x/>  public class PairSum {  public PairSum() {  super();  }    //Using hashing  **Time Complexity:**O(n) **Auxiliary Space:** O(n) where n is size of array.  public static void printpairsUsingHash(int input[], int sum){  HashSet<Integer> hash = new HashSet<Integer>();  for(int i=0;i<input.length;i++) {  int temp = sum - input[i];    if(temp >= 0 && hash.contains(temp)) {  System.out.println("Pair with given sum " + sum + " is (" + input[i] + ", "+temp+")");  }  hash.add(input[i]);  }  }    //Using sorting  **Time Complexity:**O(n logn) //depends on sorting alg **Auxiliary Space:** O(n) for merge sort and O(1) for Heap Sort.  public static void printPairsUsingSort(int input[], int sum){  Arrays.sort(input);  int low = 0, high = input.length -1;  for(int i = 0;i < input.length; i++){  if(input[low] + input[high] == sum){  System.out.println("The pairs are:"+input[low]+","+input[high]);  return;  } else if(input[low] + input[high] < sum){  low++;  } else if(input[low] + input[high] > sum){  high--;  }  }  }      public static void main(String s[]){  int input[] = {1, 4, 45, 6, 10, 8};  int n = 51;  printpairsUsingHash(input,n);  printPairsUsingSort(input,n);  }  } |

|  |  |
| --- | --- |
| 5.  Find\_Median\_Among\_Arrays - <https://www.geeksforgeeks.org/median-two-sorted-arrays-different-sizes-ologminn-m/> | **6. Fix\_Sorted\_Arrays**  <https://www.geeksforgeeks.org/sort-an-almost-sorted-array-where-only-two-elements-are-swapped/>  **Time complexity** is O(n) and only one swap operation to fix the array.  The idea is to traverse from rightmost side and find the first out of order element (element which is smaller than previous element). Once first element is found, find the other out of order element by traversing the array toward left side.  public class FixSortedArrays {  public FixSortedArrays() {  super();  }    public static void sortByOneSwap(int[] arr){  for(int i = arr.length-1;i > 0; i--){  if(arr[i] < arr[i-1]){  int j= i-1;  while(j >= 0 && arr[i] < arr[j]){  j--;  }  //SWAP  int temp = arr[i];  arr[i] = arr[j+1];  arr[j+1] = temp;  break;  }  }  }    public static void printArray(int[] input){  for(int i: input){  System.out.print(i+" ");  }  System.out.println();  }    public static void main(String s[]){  int[] input = {10, 30, 20,40, 50, 60, 70};  printArray(input);  sortByOneSwap(input);  printArray(input);  }  } |

|  |  |
| --- | --- |
| 7. **Int to English – Imp**  <https://leetcode.com/problems/integer-to-english-words/discuss/70627/Short-clean-Java-solution>  import java.util.Scanner;  public class IntToEnglish {  private final String[] belowTen = new String[] {"", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine" };  private final String[] belowTwenty = new String[] {"Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"};  private final String[] belowHundred = new String[] {"", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};  public String numberToWords(int num) {  if (num == 0)  return "Zero";  return helper(num);  }  private String helper(int num) {  String result = new String();  if (num < 10)  result = belowTen[num];  else if (num < 20)  result = belowTwenty[num - 10];  else if (num < 100)  result = belowHundred[num / 10] + " " + helper(num % 10);  else if (num < 1000)  result = helper(num / 100) + " Hundred " + helper(num % 100);  else if (num < 1000000)  result = helper(num / 1000) + " Thousand " + helper(num % 1000);  else if (num < 1000000000)  result = helper(num / 1000000) + " Million " + helper(num % 1000000);  else  result = helper(num / 1000000000) + " Billion " + helper(num % 1000000000);  return result.trim();  }  public static void main(String s[]){  Scanner scanner = new Scanner(System.in);  int input = scanner.nextInt();  IntToEnglish ite = new IntToEnglish();  System.out.println(ite.numberToWords(input));  }  } | **Int to English – Imp (Best)**  <https://leetcode.com/problems/integer-to-english-words/discuss/70625/My-clean-Java-solution-very-easy-to-understand>  import java.util.Scanner;  public class IntToEnglish\_BEST {  public IntToEnglish\_BEST() {  super();  }    private final String[] LESS\_THAN\_20 = {"", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"};  private final String[] TENS = {"", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};  private final String[] THOUSANDS = {"", "Thousand", "Million", "Billion"};  public String numberToWords(int num) {  if (num == 0)  return "Zero";  int i = 0;  String words = "";    while (num > 0) {  if (num % 1000 != 0)  words = helper(num % 1000) +THOUSANDS[i] + " " + words;  num /= 1000;  i++;  }    return words.trim();  }  private String helper(int num) {  if (num == 0)  return "";  else if (num < 20)  return LESS\_THAN\_20[num] + " ";  else if (num < 100)  return TENS[num / 10] + " " + helper(num % 10);  else  return LESS\_THAN\_20[num / 100] + " Hundred " + helper(num % 100);  }    public static void main(String s[]){  Scanner scanner = new Scanner(System.in);  int input = scanner.nextInt();  IntToEnglish\_BEST ite = new IntToEnglish\_BEST();  System.out.println(ite.numberToWords(input));  }  } |

|  |  |
| --- | --- |
| 8. Union and Intersection\_of\_two\_sorted\_arrays\_of\_integers -  <https://www.geeksforgeeks.org/union-and-intersection-of-two-sorted-arrays-2/>  Union of arrays arr1[] and arr2[] - **O(m+n)**  To find union of two sorted arrays, follow the following merge procedure :  1) Use two index variables i and j, initial values i = 0, j = 0 2) If arr1[i] is smaller than arr2[j] then print arr1[i] and increment i. 3) If arr1[i] is greater than arr2[j] then print arr2[j] and increment j. 4) If both are same then print any of them and increment both i and j. 5) Print remaining elements of the larger array.  static int printUnion(int arr1[], int arr2[], int m, int n) {  int i = 0, j = 0;  while (i < m && j < n) {  if (arr1[i] < arr2[j])  System.out.print(arr1[i++] + " ");  else if (arr2[j] < arr1[i])  System.out.print(arr2[j++] + " ");  //To Handle Duplicates as well below else if will take care  else if (arr2[j] == arr1[i]){  System.out.print(arr1[i++] + " ");  System.out.print(arr2[j++] + " ");  }  //If not To Handle Duplicates – ignore above else if and put below else stmt  // else {  // System.out.print(arr2[j++] + " ");  // i++;  // }  }  /\* Print remaining elements of  the larger array \*/  while (i < m)  System.out.print(arr1[i++] + " ");  while (j < n)  System.out.print(arr2[j++] + " ");  return 0;  } | Intersection of arrays arr1[] and arr2[]  To find intersection of 2 sorted arrays, follow the below approach :  1) Use two index variables i and j, initial values i = 0, j = 0 2) If arr1[i] is smaller than arr2[j] then increment i. 3) If arr1[i] is greater than arr2[j] then increment j. 4) If both are same then print any of them and increment both i and j.  static void printIntersection(int arr1[], int arr2[], int m, int n)  {  int i = 0, j = 0;  while (i < m && j < n)  {  if (arr1[i] < arr2[j])  i++;  else if (arr2[j] < arr1[i])  j++;  else  {  System.out.print(arr2[j++]+" ");  i++;  }  }  }    public static void main(String args[]) {  int arr1[] = { 1, 2, 4, 5, 6 };  int arr2[] = { 2, 3, 5, 7 };  int m = arr1.length;  int n = arr2.length;  printUnion(arr1, arr2, m, n);  System.out.println();  printIntersection(arr1, arr2, m, n);  } |

|  |  |
| --- | --- |
| 9. **List\_Max\_Contiguous\_Sum**  public class MaxContiguousSubArray {  public MaxContiguousSubArray() {  super();  }  public static void main(String s[]) {  int [] a = {-2, -3, 4, -1, -2, 1, 5, 3, -3};  System.out.println("Maximum contiguous sum is " + maxSubArraySum(a));  }  public static int maxSubArraySum(int a[]){  int size = a.length;  int maxSoFar = 0, maxEndingHere = 0;  for(int i=0;i<size; i++) {  maxEndingHere = maxEndingHere + a[i];    if(maxSoFar < maxEndingHere) {  maxSoFar = maxEndingHere;  }  if(maxEndingHere < 0){  maxEndingHere = 0;  }  }  return maxSoFar;  }  }  // this program uses Kadane’s algorithm  Lets take the example:  {-2, -3, 4, -1, -2, 1, 5, -3}  max\_so\_far = max\_ending\_here = 0  for i=0, a[0] = -2  max\_ending\_here = max\_ending\_here + (-2)  Set max\_ending\_here = 0 because max\_ending\_here < 0  for i=1, a[1] = -3  max\_ending\_here = max\_ending\_here + (-3)  Set max\_ending\_here = 0 because max\_ending\_here < 0  for i=2, a[2] = 4  max\_ending\_here = max\_ending\_here + (4)  max\_ending\_here = 4  max\_so\_far is updated to 4 because max\_ending\_here greater  than max\_so\_far which was 0 till now  for i=3, a[3] = -1  max\_ending\_here = max\_ending\_here + (-1)  max\_ending\_here = 3  for i=4, a[4] = -2  max\_ending\_here = max\_ending\_here + (-2)  max\_ending\_here = 1    for i=5, a[5] = 1  max\_ending\_here = max\_ending\_here + (1)  max\_ending\_here = 2    for i=6, a[6] = 5  max\_ending\_here = max\_ending\_here + (5)  max\_ending\_here = 7  max\_so\_far is updated to 7 because max\_ending\_here is  greater than max\_so\_far    for i=7, a[7] = -3  max\_ending\_here = max\_ending\_here + (-3)  max\_ending\_here = 4 | 10. **Math\_Int\_Divide\_without\_div\_operator** <https://www.geeksforgeeks.org/divide-two-integers-without-using-multiplication-division-mod-operator/>  package com.hl.problems;  public class DivideWithoutDivOperator {  public DivideWithoutDivOperator() {  super();  }  public static int divideAppr1(int dividend, int divisor) {  //Keep subtracting the divisor from dividend until dividend becomes less than divisor.  //The dividend becomes the remainder, and the number of times subtraction is done becomes the quotient.  //Time complexity : O(a) & Auxiliary space : O(1)  int sign = ((dividend < 0) ^ (divisor < 0)) ? -1 : 1;  dividend = Math.abs(dividend);  divisor = Math.abs(divisor);  int quotient = 0;  while (dividend >= divisor) {  dividend -= divisor;  ++quotient;  }  return sign \* quotient;  }  public static long divideAppr2(long dividend, long divisor) {  //Efficient Approach : Use bit manipulation in order to find the quotient. The divisor and dividend can be written as  //dividend = quotient \* divisor + remainder  //As every number can be represented in base 2(0 or 1), represent the quotient in binary form by using shift operator as given below :  //Determine the most significant bit in the quotient. This can easily be calculated by iterating on the bit position i from 31 to 1.  //Find the first bit for which divisor << i is less than dividend and keep updating the ith bit position for which it is true.  //Add the result in temp variable for checking the next position such that (temp + (divisor << i) ) is less than dividend.  //Return the final answer of quotient after updating with corresponding sign.  //Time complexity : O(log(a))  //Auxiliary space : O(1)  long sign = ((dividend < 0) ^ (divisor < 0)) ? -1 : 1;    dividend = Math.abs(dividend);  divisor = Math.abs(divisor);    long quotient = 0, temp = 0;  for (int i = 31; i >= 0; --i)  {  if (temp + (divisor << i) <= dividend)  {  temp += divisor << i;  quotient |= 1 << i;  }  }  return (sign \* quotient);  }    public static void main(String s[]) {  int a = 10, b = 3;  System.out.println(divideAppr1(a, b));  System.out.println(divideAppr2(a, b));  int a1 = 43, b1 = -8;  System.out.println(divideAppr1(a1, b1));  System.out.println(divideAppr2(a1, b1));  }  } |

|  |  |
| --- | --- |
| 11. **Monotonic Array**  [**https://leetcode.com/articles/monotonic-array/**](https://leetcode.com/articles/monotonic-array/)  public class MonotonicArray {  public MonotonicArray() {  super();  }    /\*  Complexity Analysis  Time Complexity: O(N), where N is the length of Array.  Space Complexity: O(1).  \*/    public static boolean isMonotonic(int arr[]){  boolean inc = true;  boolean dec = true;    for(int i=0;i<arr.length-1;i++){  if(arr[i] > arr[i+1])  inc = false;  if(arr[i] < arr[i+1])  dec = false;  }  return inc || dec;  }      public static void main(String s[]){  //int input[] = {1,2,4,5};  //int input[] = {1,3,1};  int input[] = {5,4,2,2,3};  //int input[] = {1,2,4,5};  System.out.println(isMonotonic(input));  }  } | 12. PalindromeGenerator  - <https://www.geeksforgeeks.org/print-all-palindrome-permutations-of-a-string/>  - <https://www.geeksforgeeks.org/print-palindromic-permutations-given-string-alphabetic-order/>  Check code in laptop |

|  |
| --- |
| 13.  <https://www.geeksforgeeks.org/recursive-function-check-string-palindrome/>  public class PalindromeRecursive {  public PalindromeRecursive() {  super();  }    public static boolean checkPalindromeRecursive (String s, int low, int high){  if(low == high)  return true;  if(s.charAt(low) != s.charAt(high))  return false;  if(low <= high)  return checkPalindromeRecursive (s,low+1,high-1);  return true;  }    public static boolean isPalindrome(String s){  int n = s.length();  if(n == 0)  return false;  return checkPalindromeRecursive(s,0,n-1);  }    public static void main(String s[]){  String input = "MALAYALAM";  System.out.println(isPalindrome(input));  input = "MADAM";  System.out.println(isPalindrome(input));    }  } |

|  |  |
| --- | --- |
| 14. Phone\_Number\_to\_String – imp <https://www.programcreek.com/2014/04/leetcode-letter-combinations-of-a-phone-number-java/>  <https://www.geeksforgeeks.org/find-possible-words-phone-digits/>  **Time Complexity:**Time complexity of above code is **O(4n)** where n is number of digits in input number.  import java.util.ArrayList;  import java.util.HashMap;  import java.util.List;  import java.util.Scanner;  public class PhoneNumberToString\_1 {  public PhoneNumberToString\_1() {  super();  }    public static List<String> letterCombinations(String digits) {  HashMap<Character,char[]> phoneNoMap = new HashMap<Character,char[]>();  phoneNoMap.put('2',new char[] {'A','B','C'});  phoneNoMap.put('3',new char[] {'D','E','F'});  phoneNoMap.put('4',new char[] {'G','H','I'});  phoneNoMap.put('5',new char[] {'J','K','L'});  phoneNoMap.put('6',new char[] {'M','N','O'});  phoneNoMap.put('7',new char[] {'P','Q','R','S'});  phoneNoMap.put('8',new char[] {'T','U','V'});  phoneNoMap.put('9',new char[] {'W','X','Y','Z'});    List<String> result = new ArrayList<String>();  if(digits == null || digits.length() == 0){  return result;  }    char[] arr = new char[digits.length()];  helper(digits,0,phoneNoMap,result,arr);    return result;  }    public static void helper(String digits,int index, HashMap<Character,char[]> phoneNoMap, List<String> result, char[] arr){  if(index == digits.length()){  result.add(new String(arr));  return;  }  char number = digits.charAt(index);  char[] candidates = phoneNoMap.get(number);  for(int i=0;i<candidates.length; i++){  arr[index] = candidates[i];  helper(digits,index+1,phoneNoMap,result,arr);  }  }    public static void main(String s[]){  Scanner scan = new Scanner(System.in);  String input = scan.nextLine();  System.out.println(input);  System.out.println(letterCombinations(input));    }  } | **Given a keypad as shown in diagram, and a n digit number, list all words which are possible by pressing these numbers.** *For example if input number is 234, possible words which can be formed are (Alphabetical order):* *adg adh adi aeg aeh aei afg afh afi bdg bdh bdi beg beh bei bfg bfh bfi cdg cdh cdi ceg ceh cei cfg cfh cfi*  Mobile-keypad  Explanation:  Let’s do some calculations first.  How many words are possible with seven digits with each digit representing n letters?  For first digit we have at most four choices, and for each choice for first letter, we have at most four choices for second digit and so on. So it’s simple maths it will be O(4n).  Since keys 0 and 1 don’t have any corresponding alphabet and many characters have 3 characters, 4n would be the upper bound of number of words and not the minimum words.  Now let’s do some examples. For number above 234.  Do you see any pattern?  Yes, we notice that the last character always either G,H or I and whenever it resets its value from I to G, the digit at the left of it gets changed.  Similarly whenever the second last alphabet resets its value, the third last alphabet gets changes and so on.  First character resets only once when we have generated all words. This can be looked from other end also.  That is to say whenever character at position i changes, character at position i+1 goes through all possible characters and it creates ripple effect till we reach at end.  Since 0 and 1 don’t have any characters associated with them. we should break as there will no iteration for these digits.  Let’s take the second approach as it will be easy to implement it using recursion.  We go till the end and come back one by one. Perfect condition for recursion.  let’s search for base case. When we reach at the last character, we print the word with all possible characters for last digit and return. Simple base case |

|  |  |
| --- | --- |
| 15. | 16. Remove\_Zeros\_From\_Array - imp  <https://www.geeksforgeeks.org/remove-leading-zeros-from-an-array/>  **Below code Removes all the founded values in the array O(N)**  public class RemoveZerosFromArray {  public RemoveZerosFromArray() {  super();  }  public static int removeElement(int[] nums, int val) {  int numVals = 0;  for (int i = 0; i < nums.length; i++) {  if (nums[i] == val) {  numVals++;  } else {  nums[i - numVals] = nums[i];  }  }  int[] temp = new int[nums.length - numVals];  for(int i=0;i<temp.length;i++){  temp[i] = nums[i];  }  for(int i: temp)  System.out.print(i + " ");  System.out.println();  return nums.length - numVals;  }    public static void main(String s[]){  int[] input = {1,0,0,2,3,1,0,4,0,9};  System.out.println(removeElement(input,0));  }  **Below code Removes all the leading zeros in the array**  public static void removeLeadingZeros(int[] nums){  int nonzeroIdx = -1;  for (int i = 0; i < nums.length; i++) {  if (nums[i] != 0) {  nonzeroIdx = i;  break;  }  }  for (int i = nonzeroIdx; i < nums.length; i++) {  System.out.print(nums[i] + " ");  }  } |

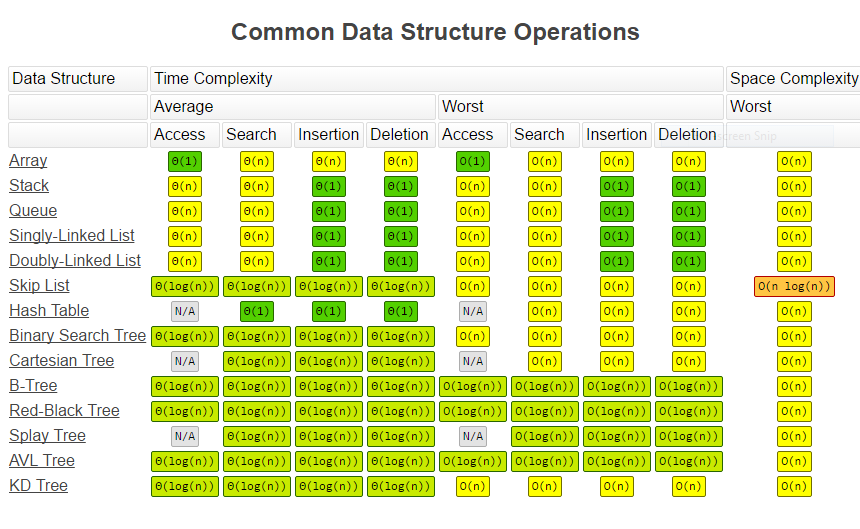
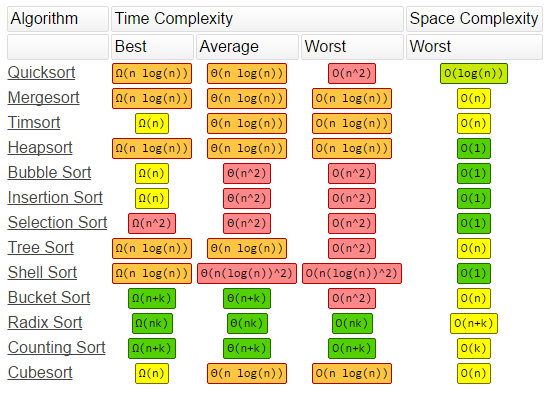
|  |  |
| --- | --- |
| 17. | 18. **Remove Duplicate characters from string**    //**Using hashing – O(N)**  public static void removeDup(String str){  LinkedHashSet<Character> lhs = new LinkedHashSet<Character>();  for(Character i: str.toCharArray())  lhs.add(i);  for(Character c: lhs)  System.out.print(c);  }  **//Using HashMap/HashSet – O(N)**  public static void removeDup1(String str){  //HashMap<Character, Boolean> hm = new HashMap<>();  HashSet<Character> hs = new HashSet<>();  StringBuilder sb = new StringBuilder();  for(Character i: str.toCharArray()){  // if(!hm.containsKey(i)){  // sb.append(i);  // hm.put(i,true);  // }  if(!hs.contains(i)){  sb.append(i);  hs.add(i);  }  }  System.out.println(sb.toString());  } |

|  |  |
| --- | --- |
| 19. Task Scheduler  https://leetcode.com/articles/task-scheduler/ | 20. |

|  |  |
| --- | --- |
| **Fibonacci Series (with Recursion and Memoization)**  public class FibonacciWithRecursionMemoization {  public FibonacciWithRecursionMemoization() {  super();  }  public static Map<Long,Long> memo = new HashMap<>();    public static long fibWithMemo(long n){  if (n < 0)  throw new IllegalArgumentException("Index was negative. No such thing as a negative index in a series.");  if(n == 0 || n == 1)  return n;    if(memo.containsKey(n)){  //System.out.printf("grabbing memo[%d]\n", n);  return memo.get(n);  }  //System.out.printf("computing fib(%d)\n", n);  long result = fibWithMemo(n-1) + fibWithMemo(n-2);  memo.put(n,result);  return result;  } | public static void main(String s[]){  Scanner scanner = new Scanner(System.in);  int input = scanner.nextInt();    //To print the n-th fibonacci element - with memoization  System.out.print(fibWithMemo(input));  //To print the first n fibonacci series - with memoization  int i =0;  long preTime=System.currentTimeMillis();  while(i != input){  System.out.print(fibWithMemo(i) + " ");  i++;  }  long postTime=System.currentTimeMillis();  System.out.println();  System.out.println("Time taken to compute in milliseconds->"+(postTime-preTime)); // 152695ms for 51 fibonacci numbers    }  } |

|  |  |
| --- | --- |
| **Prime Numbers**  **Check if given Number is a Prime Number or not**  The number which is only divisible by itself and 1 is known as **prime number**, for example 7 is a prime number because it is only divisible by itself and 1.  This program takes the number (entered by user) and then checks whether the input number is prime or not. The program then displays the result.  class GFG {        // function check whether a number      // is prime or not      static boolean isPrime(int n)      {          // Corner case          if (n <= 1)              return false;            // Check from 2 to n-1          for (int i = 2; i < n; i++)              if (n % i == 0)                  return false;            return true;      }        /\* Driver program  \*/      public static void main(String[] args)      {           if(isPrime(11))           System.out.println(" true") ;             else           System.out.println(" false");        }  } | **Displays the prime number between two intervals**  public class Prime {  public static void main(String[] args) {  int low = 20, high = 50;  while (low < high) {  boolean flag = false;  for(int i = 2; i <= low/2; ++i) {  // condition for nonprime number  if(low % i == 0) {  flag = true;  break;  }  }  if (!flag)  System.out.print(low + " ");  ++low;  }  }  } |
| **Factorial of given number**  class Factorial {   public static int factorial(int n) {      // single line to find factorial      return (n == 1 || n == 0) ? 1 : n \* factorial(n - 1);      }  // Driver Code    public static void main(String args[])    {  int num = 5;      System.out.println("Factorial of " + num +                        " is " + factorial(num));     }  } | **Valid anagram**  **/\*Given two strings s and t, write a function to determine if t is an anagram of s.**  **Anagram: reorder of letters.**  **s = "anagram", t = "nagaram", return true.**  **s = "rat", t = "car", return false.**    **Use HashMap<charactor, count> to store the frequency of chars of 1st string, and check aginst 2nd string.**  **s character: +1;**  **t character: -1;**  **check count of each index in the map; they should all be 0\*/**    public boolean isAnagram(String s, String t) {  if (s == null || t == null) {  return s == null && t == null;  } else if (s.length() != t.length()) {  return false;  }    HashMap<Character, Integer> map = new HashMap<Character, Integer>();  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);  }  if (!map.containsKey(t.charAt(i))) {  map.put(t.charAt(i), -1);  } else {  map.put(t.charAt(i), map.get(t.charAt(i)) - 1);  }  }  for (int i = 0; i < s.length(); i++) {  if (map.get(s.charAt(i)) != 0) {  return false;  }  }  return true;  } |

|  |  |
| --- | --- |
| **Java get Input**  //import java.util.Scanner  Scanner in = new Scanner(System.in)  int i = in.nextInt  String s = in.nextLine();  StringBuilder sb = new StringBuilder();  sb.append(“SomeString”);  **String**  str.length()  str.substring(0,str.length()-2)  str.charAt(3)  str.trim()  **Array**  Arr.length  **Sort a String**  // convert input string to char array          char tempArray[] = inputString.toCharArray();  // sort tempArray          Arrays.sort(tempArray);  // return new sorted string          return new String(tempArray);  **Int to String / String to Int**  String a = String.valueOf(2); //integer to numeric string  int i = Integer.parseInt(a); //numeric string to an int  **Iterate Map**  Map<Integer, Integer> map = new HashMap<Integer, Integer>();  for (Map.Entry<Integer, Integer> entry : map.entrySet()) {  System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());  }  throw new NoSuchElementException();  **Comparable/Comparator**  if(((Comparable<E>)obj).compareTo(current.data) == 0){}  private class Node<T> implements Comparable<T> {  @Override  public int compareTo(T obj) {  if(obj == this.data)  return 0;  else  return 1;  }   PriorityQueue<String> pQueue =new PriorityQueue<String>();  }  Math.abs(intvalue)  **Calculcate running time in Millisecond**:  long preTime=System.currentTimeMillis();  long postTime=System.currentTimeMillis();  System.out.println("Time taken to compute in milliseconds->"+(postTime-preTime));   ASCII characters from 0 to 127. | **Reverse a String:**  public static String reverse(String str) {  char[] charArray = str.toCharArray();  char temp;  int i = 0, j = charArray.length;    while (i < j) {  temp = charArray[i];  charArray[i] = charArray[j];  charArray[j] = temp;  i++;  j--;  }  return String.valueOf(charArray);  }  **SWAP**  public static String swap(String a, int i, int j) {  char[] charArray = a.toCharArray();  char temp = charArray[i];  charArray[i] = charArray[j];  charArray[j] = temp;  return String.valueOf(charArray);  } |
|  |  |



|  |  |
| --- | --- |
| 12. PalindromeGenerator  - <https://www.geeksforgeeks.org/print-all-palindrome-permutations-of-a-string/>  - <https://www.geeksforgeeks.org/print-palindromic-permutations-given-string-alphabetic-order/>  package com.hl.problems;  public class PalindromeGeneratorGod {  public PalindromeGeneratorGod() {  super();  }  public static final int MAX\_CHAR = 26;  public static int[] freq = new int[MAX\_CHAR];  public static char oddChar;  public static String half="";  public static void printAllPermutations(char[] c, int n) {  if (!canMakePalindrome(c, n)) {  System.out.println("Cant Make a Palindrome with the given string");  return;  } else {  System.out.println("Proceed");  }    findOddAndRemoveItsFreq();  System.out.println("OddChar : "+ oddChar);    for (int i = 0; i < n / 2; i++) {  half += c[i];  }  System.out.println("Half : " + half);    System.out.println("List of Permutations: ");  permute(half, 0, half.length() - 1);  }  private static void permute(String str, int l, int r) {  if (l == r){  if(oddChar == ' ')  System.out.println(str + reverse(str));  else  System.out.println(str + oddChar + reverse(str));  }else {  for (int i = l; i <= r; i++) {  str = swap(str, l, i);  permute(str, l + 1, r);  str = swap(str, l, i);  }  }  }  public static String reverse(String str) {  char[] charArray = str.toCharArray();  char temp;  int i = 0, j = charArray.length-1;    while (i < j) {  temp = charArray[i];  charArray[i] = charArray[j];  charArray[j] = temp;  i++;  j--;  }  return String.valueOf(charArray);  } | public static String swap(String a, int i, int j) {  char[] charArray = a.toCharArray();  char temp = charArray[i];  charArray[i] = charArray[j];  charArray[j] = temp;  return String.valueOf(charArray);  }  public static void findOddAndRemoveItsFreq() {  oddChar = ' ';  for (int i = 0; i < MAX\_CHAR; i++) {  if (freq[i] % 2 != 0) {  freq[i]--;  oddChar = (char) (i + 'A');  break;  }  }  }  public static boolean canMakePalindrome(char[] c, int n) {  int oddCharCount = 0;  for (int i = 0; i < n; i++) {  int idx = c[i] - 'A';  freq[idx]++;  if (freq[idx] % 2 != 0)  oddCharCount++;  else  oddCharCount--;  }  /\* Palindrome condition :  \* if length is odd then only one letter's frequency must be odd  \* if length is even no letter should have odd frequency \*/  if ((n % 2 == 1 && oddCharCount == 1) || (n % 2 == 0 && oddCharCount == 0)) {  return true;  } else {  return false;  }  }  public static void main(String s[]) {  String str = "MALAYALAM";  str = str.toUpperCase();  char[] input = str.toCharArray();  printAllPermutations(input, input.length);  }  } |