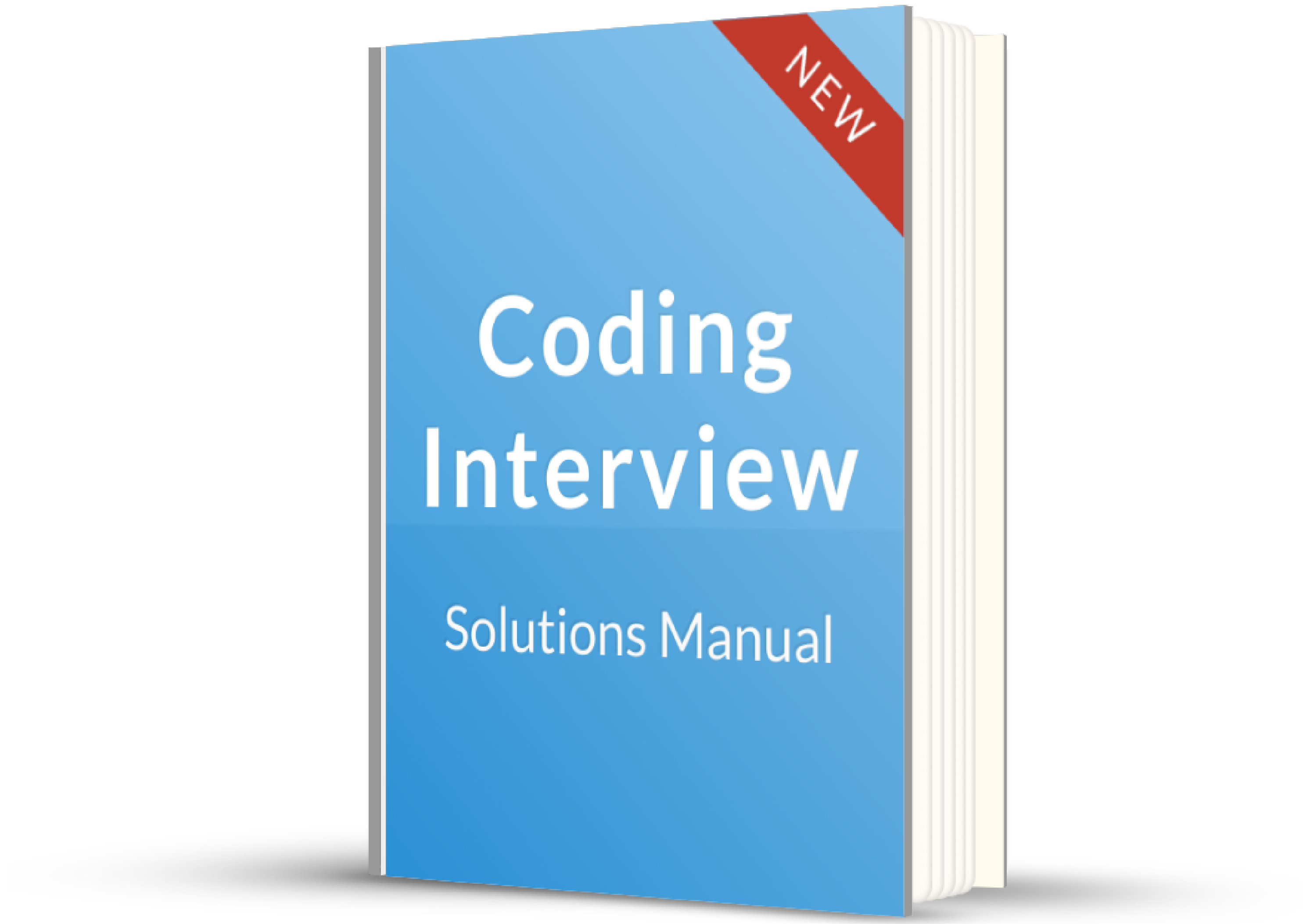

# Coding Interview

Solutions Manual

InterviewSolutionsManual.com

# Table of Contents

# Warm Ups

## FizzBuzz

**#basic** **#pulse test**

**Question:**
Write a function that iterates over the numbers 1 through 100. If the number is divisible by 3, output "Fizz". If the number is divisible by 5, output "Buzz". If the number is divisible by both 3 and 5, output "FizzBuzz". If the number is divisible by none of these, output the number.

**Answer:**

```java
public class FizzBuzz {
  public static void fizzBuzz() {
    for (int i = 1; i <= 100; i++) {
      if (i % 3 == 0 && i % 5 == 0)
        System.out.println("FizzBuzz");
      else if (i % 3 == 0)
        System.out.println("Fizz");
      else if (i % 5 == 0)
        System.out.println("Buzz");
      else
        System.out.println(i);
    }
  }
}
```

This is a basic "can this guy write a line of code" question that we should be able to do in our sleep. It's simply a for loop with an if-else chain. Don't get offended if asked this in an interview. Many times grad students applying for a software engineering job haven't written a line of code in years and struggle through this one. It should get us warmed up and in the coding mindset.

## Two are true

**#logic #basic**

**Question:**
Write a function that takes 3 booleans and return true if at least two of them are true.

**Answer:**

```java
public class Solution {
  public static boolean atLeastTwo(boolean a, boolean b, boolean c) {
    return a ? (b || c) : (b && c);
  }
}
```

The key here is to be succinct and think before we start blindly writing down code. We must avoid writing many lines of spaghetti code and if-else statements when the solution can be accomplished with a relatively straight forward ternary operator. If readability is more important to us, we can also achieve the same effect with a simple if-else statement.

**Complexity:**

- Time: O(1)
- Space: O(1)

# String Manipulation

## String reversal

**#string manipulation** **#basic**

**Question:**
Given a string, write a function that returns a string that contains the same characters in the reverse order.

**Answer:**

```java
public class Solution {
  public static String reverse(String input) {
    StringBuilder builder = new StringBuilder();
    char[] characters = input.toCharArray();
    for (int i = characters.length - 1; i >= 0; i--) {
      builder.append(characters[i]);
    }
    return builder.toString();
  }
}
```

This is a straight forward question used to test basic coding ability. The only slight trick is to use a string builder because Java strings are immutable. We simply iterate through the input string backwards, append the characters to a string builder, and output the results.

**Complexity:**

- Time: O(n)
- Space: O(n)

## String rotation

**#string manipulation** **#tricky**

**Question:**
Given two strings, write a function to check if one string is a rotation of the other

**Answer:**
```java
public class Solution {
  public static boolean isRotation(String one, String two) {
    return (one.length() == two.length()) &&
          ((one + one).indexOf(two) != -1);
  }
}
```

We first check to make sure that the two strings are the same length. We then concatenate the first string with itself, and see if the second string is a substring of this concatenated string. Take for example the inputs "tionssolu" and "solutions". When we concatenate the first input with itself we get "tionssolutionssolu". Now we can clearly see that this string contains "solutions".

**Complexity:**
- Time: O($n^2$)
- Space: O(n)

## Character uniqueness

**#string manipulation #hash map #hash set**

**Question:**
Given a string, write a function that checks to see if every character in the string is unique.

**Answer:**
```
import java.util.HashSet;

public class Solution {
  public static boolean isUnique(String str) {
    if (str.isEmpty())
      return false;

    HashSet<Character> map = new HashSet<Character>();
    char[] characters = str.toCharArray();
    for (int i = 0; i < characters.length; i++) {
      if (map.contains(characters[i]))
        return false;
      else
        map.add(characters[i]);
    }
    return true;
  }
}
```

This is a relatively straight forward hash map problem. Here we've opted to use a hash set because there is really no "value" that needs to be stored in the hash map. We simply iterate through the characters, adding each character to a hash set. If we come upon a character that already exists in the has set, we have a repeat character. Otherwise, all the characters in the string are unique.

**Complexity:**
- Time: O(n)
- Space: O(n)

## Remove duplicates from string

**#string manipulation** **#hash set** **#string builder**

**Question:**
Given a string, write a function that returns the string with all duplicate characters removed.

**Answer:**

```java
import java.util.LinkedHashSet;
import java.util.Set;

public class Solution {
  public static String removeDuplicates(String string) {
    char[] chars = string.toCharArray();
    Set<Character> charSet = new LinkedHashSet<Character>();
    for (char c : chars)
      charSet.add(c);

    StringBuilder sb = new StringBuilder();
    for (Character character : charSet)
      sb.append(character);
    return sb.toString();
  }
}
```

Here we use a data structure called a linked hash set. It allows us to add the characters from the string to the set efficiently in order to remove duplicates. We can then then iterate over the characters in the set efficiently to rebuild the string without duplicates. We opt to use a string builder in this case because strings in Java are immutable, so using a string builder is more memory efficient.

**Complexity:**

- Time: O(n)
- Space: O(n)

## Word order reversal

**#string manipulation** **#tricky**

**Question:**
Given a string that contains words separated by spaces, write a function that reverses the order of the words.

**Answer:**

```java
public class Solution {
  public static char[] reverseWordOrder(char[] words) {
    reverseCharacters(words, 0, words.length);
    int lastSpace = 0;
    for (int i = 0; i < words.length; i++) {
      if (words[i] == ' ') {
        reverseCharacters(words, lastSpace, i);
        lastSpace = i + 1;
      }
    }
    reverseCharacters(words, lastSpace, words.length);
    return words;
  }
  private static void reverseCharacters(char[] words, int min, int max) {
    for (int i = 0; i < (max - min) / 2; i++) {
      char temp = words[min + i];
      words[min + i] = words[max - i - 1];
      words[max - i - 1] = temp;
    }
  }
}
```

We first iterate over the characters in the string, reversing the characters by swapping each letter with the corresponding letter from the end of the string. We then make a second pass over the string, looking for spaces. Each time we encounter a space, reverse the characters we've seen since the last space. We must make sure to also flip the last chunk of characters, as there may be no space at the end of the string.

**Complexity:**

- Time: O(n)
- Space: O(1)

## String to integer

**#conversion** **#string manipulation**

**Question:**
Given an integer represented as a string, write a function to convert it to an integer.

**Answer:**
```java
public class Solution {
  public static int stringToInteger(String input) {
    int result = 0;
    int i = 0;
    boolean isNegative = false;
    if (input.charAt(i) == '-') {
      isNegative = true;
      i++;
    }
    for (; i < input.length(); i++) {
      result *= 10;
      result += ((int) input.charAt(i) - 48);
    }
    if (isNegative)
      result *= -1;
    return result;
  }
}
```

We must first check to see if the first character in the string is a minus sign. If it is, we'll have to multiply the number by -1 when we've parsed the rest of the number. We can then iterate over the rest of the characters in the string, multiplying the result by 10 at each step and adding the next digit. A digit can be obtained from a character by subtracting the value of the '0' character, which is 48.

**Complexity:**
- Time: O(n)
- Space: O(1)

## Longest common substring

**#string manipulation** **#dynamic programming**

**Question:**
Given two strings, find the length of the longest substring that they have in common.

**Answer:**

```java
public class Solution {
  public static int longestCommonSubstring(String one, String two) {
    int n = one.length();
    int m = two.length();

    if (n == 0 || m == 0)
      return 0;

    int maxLenth = 0;
    int[][] memoized = new int[n][m];

    for (int i = 0; i < n; i++)
      for (int j = 0; j < m; j++) {
        if (one.charAt(i) == two.charAt(j)) {
          if (i == 0 || j == 0)
            memoized[i][j] = 1;
          else
            memoized[i][j] = memoized[i - 1][j - 1] + 1;
          if (memoized[i][j] > maxLenth)
            maxLenth = memoized[i][j];
        }
      }
    return maxLenth;
  }
}
```

This is a classic dynamic programming problem. We compute the max length for smaller substrings, saving our results in the "memoized" table and building larger and larger solutions using the previously memoized results. We obtain our maximum length by finding the longest common substring out of all of the subproblems we compute substrings for.

**Complexity:**

- Time: O(nm)
- Space: O(nm)

# Bit Twiddling

## Computing $2^x$

**#bit manipulation**

**Question:**
Write a function to compute $2^x$ without using the multiplication operator or any looping structures.

**Answer:**

```java
public class Solution {
  public static int twoToThe(int x) {
    return 1 << x;
  }
}
```

When we shift a base 2 integer to the left by 1, it is equivalent to multiplying that number by 2. This means that if we shift 2 to the left x times, we get $2^x$.

## Power of 2 check

**#bit manipulation**

**Question:**
Given an integer, write a function to check if the number is a power of two using bitwise operators.

**Answer:**

```java
public class Solution {
  public static boolean isPowerOfTwo(int x) {
    return (x & (x - 1)) == 0;
  }
}
```

If the number we are checking is a power of two, the binary representation will be a one followed by zeros. If we subtract one from a power of two, it is equivalent to fliping every to the right of the 1, including the 1 itself. Bitwise anding these two numbers together will always result in zero if x is a power of two, and will always result in a non-zero number if x is not a power of two.

## Swap in place

**#bit manipulation #tricky**

**Question:**
Given two integers, write a function to swap them without using any temporary storage.

**Answer:**
```
public class Solution {
  public static void swapInPlace(int a, int b) {
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
  }
}
```

Because the XOR function is its own inverse, and it is commutative, we can perform the XOR operation 3 times, storing it in the first variable, then the second, then the first again. This results in swapping the two values without using any additional storage.

## Bit swap

**#bit manipulation**

**Question:**
Write a function that takes a byte and swaps the the first 4 bits with the last 4 bits.

**Answer:**
```java
public class Solution {
  public static byte swapBits(byte x) {
    return (byte) (((x & 0xf0) >> 4) | ((x & 0x0f) << 4));
  }
}
```

This is a classic bit twiddling problem. First we create a mask for the first 4 bits, then a mask for the last 4 bits. We then shift the result of ANDing the first mask with the byte to the right by 4, and the result of ANDing the second mask with the byte to the left by 4.

## Find the lone integer

**#bitwise** **#hash table** **#tricky**

**Question:**
Given an array of integers where each number appears exactly twice, except for one integer that appears exactly once, find the lone integer.

**Answer:**
```java
public class Solution {
  public static int findTheLoner(int[] input) {
    int value = 0;
    for (int i = 0; i < input.length; i++) {
      value = value ^ input[i];
    }
    return value;
  }
}
```

This can be solved using the standard hash table approach, but a better solution (in terms of space complexity) can be achieved using bitwise operations. The bitwise XOR operator is commutative, and it is its own inverse. This means that if we XOR every integer in the array together, each pair will cancel out, and we will be left with the number that only appears once. Tricky!

**Complexity:**
- Time: O(n)
- Space: O(1)

## Same sign check

**Question:**
Given two integers, write a function that checks to see if the two numbers are the same sign using bitwise operators.

**Answer:**

```java
public class Solution43 {
  public static boolean isSameSign(int x, int y) {
    return ((x ^ y) < 0);
  }
}
```

It is important to understand how negative numbers are represented in binary in order to solve this question. Negative numbers are represented using what's called "two's compliment." This means that the highest bit will be 1 in a negative number, and 0 in a positive number. This means when we XOR the two numbers together, if the sign is the same, we will get a negative number or 1 in the highest position. If they are different we will get a 0 in the highest position, which represents a positive number. This means we can check the sign of the result of XORing the two inputs together in order to check if the two inputs have the same sign or not.

# Linked Lists

## Linked list

**#linked list** **#explanation**

**Question:**
Explain how linked lists work.

**Answer:**
```
public class Node {
  public String data;
  public Node next;

  public Node(String data) {
    this(data, null);
  }
  public Node(String data, Node node) {
    this.data = data;
    next = node;
  }
}
```

A linked list is a simple data structure that is represented by a collection of nodes in which each node has a pointer to the next node. The last node can either have a pointer to null, or maintain a "sentinal node" that represents the end. A linked list can either be singly linked or doubly linked. A doubly linked list has a pointer to the previous node in addition to the next node. Linked lists have an insertion time of O(1) and a lookup time of O(n).

## Doubly linked list node removal

**#linked list** **#basic**

**Question:**
Given a doubly linked list and an integer, write a function that removes the first occurrence of that integer from the list.

**Answer:**

```java
public class Solution {
  public static void removeTargetNode(Node head, int target) {
    Node current = head;
    while (current != null) {
      if (current.value == target) {
        if (current.next != null)
          current.next.previous = current.previous;
        if (current.previous != null)
          current.previous.next = current.next;
        break;
      }
      current = current.next;
    }
  }

  private class Node {
    public Node next;
    public Node previous;
    public int value;
  }
}
```

We first iterate over the linked list until we find a node with the given target value. We then update the previous pointer in the next node to point to the node previous to the current node. We then update the next pointer in the previous node to point to the next node of the current pointer. Now we have effectively cut the current node out of the list.

**Complexity:**

- Time: O(n)
- Space: O(1)

## Finding cycles in singly linked lists

**#linked list #cycle #tricky**

**Question:**
Given a singly linked list, write a function to determine if the linked list contains a cycle.

**Answer:**

```java
public class Solution {
  public static boolean hasCycle(Node head) {
    if (head == null)
      return false;
    Node turtle, rabbit;
    turtle = rabbit = head;
    while (true) {
      turtle = turtle.next;
      if (rabbit.next != null)
        rabbit = rabbit.next.next;
      else
        return false;
      if (turtle == null || rabbit == null)
        return false;
      if (turtle == rabbit)
        return true;
    }
  }

  private class Node {
    public Node next;
    public int value;
  }
}
```

We can detect if the linked list has a cycle by creating two pointers starting at the beginning of the list that travel at different speeds. We'll call the fast pointer the "rabbit" who will travel twice as fast as the slow pointer, which we will call the "turtle." If the rabbit reaches the end of the list, we know there is no cycle. If the turtle and the rabbit meet, this means there has to be a cycle, because there is no other way for the turtle to catch up to the rabbit.

**Complexity:**

- Time: O(n)
- Space: O(1)

## Linked list integer addition

**#linked list**

**Question:**
Given two integers represented as linked lists, write a function that returns a linked list representing the sum of the two integers. The digits stored in the linked list are in reverse order (least significant digit to most significant digit). For example if the input were (1 ⇢ 5 ⇢ 8) + (1 ⇢ 2 ⇢ 3), the result would be (2 ⇢ 7 ⇢ 1 ⇢ 1) which represents 851 + 321 = 1172.

**Answer:**
```
public class Solution {
  public static Node addLinkedListNumbers(Node first, Node second) {
    int carry = 0;
    Node result = null;
    Node iter = null;
    while (first != null || second != null) {
      int firstValue = first == null ? 0 : first.value;
      int secondValue = second == null ? 0 : second.value;
      int sum = (firstValue + secondValue + carry) % 10;
      carry = (firstValue + secondValue + carry) / 10;

      Node node = new Node();
      node.value = sum;
      node.next = null;
      if (result == null) {
        iter = node;
        result = node;
      } else {
        iter.next = node;
        iter = node;
      }

      first = (first == null) ? null : first.next;
      second = (second == null) ? null : second.next;
    }
    if (carry != 0) {
      Node node = new Node();
      node.value = carry;
      node.next = null;
      iter.next = node;
      iter = node;
    }
    return result;
  }

  private static class Node {
    public Node next;
    public int value;
  }
}
```

We iterate over the linked list nodes in both lists, adding each digit together and keeping track of the carry. If we have a carry at the end, we must make sure to add an extra node ad the end of the list representing the carry digit.

**Complexity:**

- Time: O(n)
- Space: O(n)

# Binary Search Trees

**Question:**
Explain how binary search trees work.

**Answer:**

```
public class BinarySearchTree {
  public static void insert(Node head, Node newNode) {

    if (newNode.value < head.value) {
      if (head.left == null)
        head.left = newNode;
      else
        insert(head.left, newNode);
    } else {
      if (head.right == null)
        head.right = newNode;
      else
        insert(head.right, newNode);
    }
  }

  private class Node {
    public Node left;
    public Node right;
    public int value;
  }
}
```

A binary search tree is a tree data structure, in which elements are inserted and kept in order. Each node in the BST contains exactly two children, one or both of which can be null. The special property that makes a BST a search tree, and not just any old binary tree, is that the left child of any node is less than or equal to its parent, and the right child is greater than or equal to its parent. The left and right children of any node are also roots of BSTs themselves. Searching a binary search tree can be performed in O(log n) time, assuming a well balanced BST. This is performed by going down the left path if the target value is smaller than the current node, or going down the right path if the target value is larger than the current node.

## Binary search tree validity

**#binary search tree** **#basic** **#recursive**

**Question:**
Given the pointer to a node, determine if the node is the root of a binary search tree. We can assume the integers in the tree are distinct i.e. there are no repeating numbers.

**Answer:**
```java
public class Solution {
  public static boolean isValidBST(Node root) {
    return isValidBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
  }

  private static boolean isValidBST(Node current, int min, int max) {
    if (current.right != null) {
      if (current.right.value > max ||
          !isValidBST(current.right, current.value, max))
        return false;
    }
    if (current.left != null) {
      if (current.left.value < min ||
          !isValidBST(current.left, min, current.value))
        return false;
    }
    return true;
  }

  private class Node {
    public Node left;
    public Node right;
    public int value;
  }
}
```

In order to check if the node is the root of a binary search tree, we can first check if the left child's value is less than the current node's value. We can then recursively check if the left node is itself the root of a binary search tree. We can then do the same for the right child, making sure it's value is larger than the current node's value. We must also make sure that the values are within the min and max values allowed given the ancestors of the current node. This is accomplished by passing down allowed min and max values when we are making our recursive calls.

**Complexity:**
- Time: O(n)
- Space: O(1)

## Binary search tree successor

**#binary search tree** **#search**

**Question:**
Given the root of a binary search tree and a value, find the "successor" of that value, even if the value doesn't exist in the tree. The "successor" is defined as the node appearing immediately after the given node when performing an in-order traversal.

**Answer:**

```java
public class Solution {
  private static Node getLeftMost(Node head) {
    Node current = head;
    while (current.left != null)
      current = current.left;
    return current;
  }

  private static Node getRightMost(Node head) {
    Node current = head;
    while (current.right != null)
      current = current.right;
    return current;
  }

  public static int getSuccessor(Node head, int target) {
    Node current = head;
    int successor = 0;
    while (current != null) {
      if (current.value < target && current.right != null)
        current = current.right;
      else if (current.value > target) {
        if (current.left != null &&
            getRightMost(current.left).value > target)
          current = current.left;
        else {
          successor = current.value;
          current = null;
        }
      } else {
        if (current.right != null)
          successor = getLeftMost(current.right).value;
        current = null;
      }
    }
    return successor;
  }

  private static class Node {
    private int value;
    private Node left;
    private Node right;
  }
}
```

We start off by moving down our binary search tree looking for the target value. There are two cases that need to be considered when looking for the "successor" in a binary search tree. The first is when our target value has a right child, the successor is simply the leftmost node in our target's right subtree. If our target doesn't have a right child, or our target doesn't exist, we must check the rightmost node in our left subtree. If it is less than our target, we have found our successor. Otherwise we need to move down the left subtree.

**Complexity:**

- Time: O(log n)
- Space: O(1)

**Question:**
Given the root of a binary search tree and two values, find the least common ancestor of the two values.

**Answer:**
```java
public class Solution {
  public Node findLowestCommonAncestor(Node node, int value1,
      int value2) {
    if (node == null)
      return null;

    if (node.value > value2 && node.value > value1)
      return findLowestCommonAncestor(node.left, value1, value2);
    else if (node.value < value2 && node.value < value1)
      return findLowestCommonAncestor(node.right, value1, value2);
    else
      return node;
  }

  private class Node {
    public Node left;
    public Node right;
    public int value;
  }
}
```

This is a relatively straightforward recursive solution once we realize that the lowest common ancestor is the first node where each of the two values we are looking for lies in different child subtrees of the node we are looking for. We simply search recursively down as we would searching for a value in a binary search tree until our paths diverge. At that point, we return the node at which our paths diverge.

**Complexity:**
- Time: O(n)
- Space: O(1)

**Question:**
Given the root of a binary search tree and two values, write a function that returns the path between the two values.

**Answer:**
```java
import java.util.ArrayList;
import java.util.List;

public class Solution {
  public static List<Node> findShortestPath(Node head, int x, int y) {
    List<Node> result = new ArrayList<Node>();
    head = findLowestCommonAncestor(head, x, y);
    List<Node> left = findPath(head, x);
    List<Node> right = findPath(head, y);
    if (x < y) {
      left.remove(left.size() - 1);
      result.addAll(left);
      result.addAll(right);
    } else {
      right.remove(right.size() - 1);
      result.addAll(right);
      result.addAll(left);
    }
    return result;
  }

  public static List<Node> findPath(Node head, int x) {
    List<Node> result = new ArrayList<Node>();
    if (head.value == x) {
      result.add(head);
      return result;
    }
    if (head.value > x) {
      List<Node> left = findPath(head.left, x);
      result.addAll(left);
      result.add(head);
    }
    if (head.value < x) {
      result.add(head);
      List<Node> right = findPath(head.right, x);
      result.addAll(right);
    }
    return result;
  }

  private static Node findLowestCommonAncestor(Node node, int value1,
        int value2) {
    if (node == null)
      return null;

    if (node.value > value2 && node.value > value1)
      return findLowestCommonAncestor(node.left, value1, value2);
    else if (node.value < value2 && node.value < value1)
      return findLowestCommonAncestor(node.right, value1, value2);
    else
      return node;
  }

  private static class Node {
    public Node left;
    public Node right;
    public int value;
  }
}
```

We can reuse the code that we wrote in the least common ancestor problem, because the least common ancestor will be the root of our path. Once we have the least common ancestor, we can simply build a path to the value on the right, and a path to the value on the left, and combine the two to get the path between the two nodes.

**Complexity:**

- Time: O(n) where n is the length of the path
- Space: O(n)

# Searching & Sorting

## Searching a sorted array

**#basic** **#search** **#binary search**

**Question:**
Given a sorted array of integers and a target value, write a function that finds the index of the target value in the array.

**Answer:**

```
public class Solution {
  public static int findValue(int[] input, int target) {
    return findValue(input, target, 0, input.length);
  }

  public static int findValue(int[] input, int target, int min, int max) {
    int mid = (min + max) / 2;
    if (input[mid] == target)
      return mid;
    if (min >= max)
      return -1;
    if (target < mid)
      return findValue(input, target, min, mid);

    return findValue(input, target, mid + 1, max);
  }
}
```

The solution to this problem is a simple binary search, no tricks. If the middle number is our target, return its value. If the target is less than our middle number, recur to the left. If our target is greater than our middle number, recur to the right.

**Complexity:**

- Time: O(n log n)
- Space: O(1)

## Target sum

**#sorting**

**Question:**
Given an array of integers and a target value, write a function to determine whether the array contains two numbers that add up to the target value.

**Answer:**
```java
import java.util.Arrays;

public class Solution {
  public static int[] targetSum(int[] input, int target) {
    Arrays.sort(input);
    int start = 0;
    int end = input.length - 1;
    while (start < end) {
      int sum = input[start] + input[end];
      if (sum == target) {
        int[] answer = { start, end };
        return answer;
      }
      if (sum < target)
        start++;
      if (sum > target)
        end--;
    }
    return null;
  }
}
```

We start by sorting the array and creating a pointer at the beginning and one at the end. When the sum of the two values at each pointer is less than the target, move the first pointer forward. When the sum is greater than the target, move the second pointer backward. If the two pointers meet, there is no pair of numbers in the array that sum to the target.

**Complexity:**
- Time: O(n log n)
- Space: O(1)

## Search on a rotated sorted array

**#binary search** **#tricky**

**Question:**
Given a sorted array of integers that has been rotated, find a particular value in the array without finding the pivot.

**Answer:**

```
public class Solution {
  public static int rotatedSearch(int[] values, int min, int max, int x) {
    if (values[min] == x)
      return min;
    else if (values[max] == x)
      return max;
    else if (max - min == 1)
      return -1;

    int mid = (min + max) / 2;

    if (values[min] <= values[mid]) {
      if (x <= values[mid] && x >= values[min])
        return rotatedSearch(values, min, mid, x);
      else
        return rotatedSearch(values, mid, max, x);
    } else {
      if (x >= values[mid] && x <= values[max])
        return rotatedSearch(values, mid, max, x);
      else
        return rotatedSearch(values, min, mid, x);
    }
  }
}
```

At first this problem seems difficult since we can't explicitly find the pivot point. One key observation allows us to treat this problem as a slightly modified binary search. The key is to realize that when we split the array in half, one side will be in sorted order, while the other will contain the pivot. This means if we see that one side is in order, and our value should be on that side (based on that min and max, which will be located at each end), we can simply perform a standard binary search on that side. If our value is not in the sorted side, it must be in the out of order side, and we can recur on the out of order side.

**Complexity:**

- Time: O(n log n)
- Space: O(1)

## Quicksort

**#quicksort #sorting #explanation**

**Question:**
Explain how quicksort works.

**Answer:**

```java
public class QuickSort {
  public static void swap(int input[], int a, int b) {
    int temp = input[a];
    input[a] = input[b];
    input[b] = temp;
  }

  public static int partition(int input[], int front, int end) {
    int pivot = input[front];
    while (front < end) {
      while (input[front] < pivot)
        front++;
      while (input[end] > pivot)
        end--;
      swap(input, front, end);
    }
    return front;
  }

  public static void quickSort(int input[], int front, int end) {
    if (front >= end)
      return;
    int pivot = partition(input, front, end);
    quickSort(input, front, pivot);
    quickSort(input, pivot + 1, end);
  }
}
```

Quicksort is one of the the most commonly used sorts, because it is one of the O(n log n) sorts in the average case, and it can be performed in place, without and additional memory. Quicksort works by selecting a pivot and partitioning the array based on that pivot. This involves putting all the numbers less than the pivot on the left side of the pivot, and all the numbers greater on the right side of the pivot. This is accomplished by creating pointers at both the front and end of the partition, and then swapping numbers that are out of place. These left and right sides of the pivot are then sorted recursively until partitions are of size 1 and the array is sorted.

**Complexity:**

- Time: worst case O($n^2$), average case O(n log n)
- Space: O(1)

## K smallest elements

**#sorting #pivot #quicksort**

**Question:**
Given an array of n elements, write a function that returns the k smallest numbers.

**Answer:**

```java
import java.util.Arrays;

public class Solution {
  public static void swap(int input[], int a, int b) {
    int temp = input[a];
    input[a] = input[b];
    input[b] = temp;
  }

  public static int partition(int input[], int front, int end) {
    int pivot = input[front];
    while (front < end) {
      while (input[front] < pivot)
        front++;
      while (input[end] > pivot)
        end--;
      swap(input, front, end);
    }
    return front;
  }

  public static int[] findKSmallest(int input[], int front, int end,
      int k) {
    if (front >= end)
      return null;
    int pivot = partition(input, front, end);
    if (pivot == k)
      return Arrays.copyOfRange(input, 0, pivot);
    else {
      if (pivot > k)
        return findKSmallest(input, front, pivot, k);
      return findKSmallest(input, pivot + 1, end, k);
    }
  }

  public static int[] findKSmallest(int input[], int k) {
    return findKSmallest(input, 0, input.length - 1, k);
  }
}
```

This problem can be solved with a slight modification to quicksort. The modification is to check whether the pivot index at any given point is less than or greater than the k value. If it is less, we recur on the left side of the pivot. If the pivot is greater than the k value, we recur on the right side of the pivot. If the pivot index is equal to the k value, then the part of the array to the left of the pivot represents the k smallest values, and the pivot represents the kth number. A special case of this solution where k = n / 2 can be used to find the median in O(n) time.

**Complexity:**

- Time: O(n)
- Space: O(1)

# Hash Maps

## Hash tables vs hash maps

**#hash table** **#comparison**

**Question:**
Explain the difference between a hash table and a hash map.

**Answer:**

```java
import java.util.HashMap;
import java.util.Hashtable;

class Solution {
  HashMap<String, Integer> hashMap = new HashMap<String, Integer>();
  Hashtable<String, Integer> hashTable = new Hashtable<String, Integer>();
}
```

Hash tables and hash maps have three key differences:

- Hash tables are synchronized and thread safe while hash maps are not. Concurrent access is not allowed for hash tables, so only one thread can access a hash table at a time.
- Hash maps allow both null keys and null values, while hash tables do not. If we try to insert a null key or value into a hash table, we will get a null pointer exception.
- When iterating through a hash map, and we try to modify one of the values, we will get a concurrent modification exception. If we try to modify a hash table during iteration, we will not get an error.

## Generate all valid anagrams **#hash table #anagrams #string manipulation**

**Question:**
Given an string and an English dictionary, write a function to find all valid anagrams for the string. We can pre-compute and store whatever we'd like to speed up lookup time.

**Answer:**
```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;

public class Solution {
  private static HashMap<String, List<String>> map =
      new HashMap<String, List<String>>();
  public static void preCompute(List<String> englishDictionary) {
    for (String word : englishDictionary) {
      char[] letters = word.toCharArray();
      Arrays.sort(letters);
      String sortedWord = String.valueOf(letters);
      if (map.get(sortedWord) != null) {
        map.get(sortedWord).add(word);
      } else {
        List<String> words = new ArrayList<String>();
        words.add(word);
        map.put(sortedWord, words);
      }
    }
  }

  public static List<String> getAnagrams(String input) {
    char[] letters = input.toCharArray();
    Arrays.sort(letters);
    return map.get(String.valueOf(letters));
  }
}
```

When we have the option to pre-compute, we want to do as much work as possible before hand to make lookups faster. In this case we can pre-compute a hashmap from each string with its characters in sorted order to a list of other words that are anagrams of that word. For lookups, we can simply sort the characters of the given string and return the list of words we find in the hash map for that value.

**Complexity:**
- Time: O(n log n), were n is the length of the word
- Space: O(n)

## Counting gets and puts

**#hash map**

**Question:**
Extend HashMap to count the number of get and put operations across all instances of the data structure.

**Answer:**

```
import java.util.HashMap;

public class CountedMap<K, V> extends HashMap<K, V> {
  private static int numGets = 0;
  private static int numPuts = 0;

  @Override
  public V put(K key, V value) {
    numPuts++;
    return super.put(key, value);
  }

  @Override
  public V get(Object key) {
    numGets++;
    return super.get(key);
  }

  public int getNumGets() {
    return numGets;
  }

  public int getNumPuts() {
    return numPuts;
  }
}
```

In order to count the number of puts and gets that are performed, we extend HashMap, and overwrite the put and get methods. When either method is called, we increment the corresponding count and then call the correct super method. In order to keep count across all instances of the class, we make our counting variables static, so there is just one instance of each.

# Stacks & Queues

## Stacks vs queues

**#stack #queue #explanation**

**Question:**
Explain the differences between a stack and a queue.

**Answer:**

```java
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class StackQueue {
  Stack<Object> stack = new Stack<Object>();
  Queue<Object> queue = new LinkedList<Object>();
}
```

Stacks behave like stacks of dishes, while queues behave like lines of people waiting for tickets. The difference lies in the order in which elements are removed from each data structure. A stack uses what is called a first-in, first-out (FIFO) model. A queue on the other hand uses a last-in, first-out (LIFO) model. This means that with a stack, like a stack of dishes, when we remove a dish it will be the last dish that we placed on the stack. On the other hand with a queue, like a line of people waiting for tickets, the first person in line will get their tickets first. A stack typically uses the terminology push and pop for its add and remove methods, while queues typically use enqueue and dequeue.

## Queue using stacks

**#stack #queue #tricky**

**Question:**
Implement a queue data structure using only stacks. It must support the queue and dequeue operations. Don't worry about concurrency issues.

**Answer:**
```java
import java.util.Stack;

public class Solution {
  public class Queue<E> {
    private Stack<E> incoming = new Stack<E>();
    private Stack<E> outgoing = new Stack<E>();

    public void queue(E item) {
      incoming.push(item);
    }

    public E dequeue() {
      if (outgoing.isEmpty()) {
        while (!incoming.isEmpty()) {
          outgoing.push(incoming.pop());
        }
      }
      return outgoing.pop();
    }
  }
}
```

We can create a queue using two stacks, an incoming stack and an outgoing stack. When we queue a new item, simply push it onto the incoming stack. When we have to dequeue an item, we simply pop an item off of the outgoing stack. If the outgoing stack is empty, we pop all of the items off of the incoming stack, reversing their order, and place them on the outgoing stack.

**Complexity:**
- Time: Amortized O(1)
- Space: O(n)

## Minimum stack **#stack #minimum**

**Question:**
Create a data structure that extends Stack and supports findMin in O(1) time.

**Answer:**
```java
import java.util.Stack;

public class MinimumStack<E extends Comparable<E>> extends Stack<E> {
  private Stack<E> minStack;

  public MinimumStack() {
    super();
    minStack = new Stack<E>();
  }
  @Override
  public E push(E element) {
    if (element.compareTo(minStack.peek()) <= 0)
      minStack.push(element);
    return super.push(element);
  }
  @Override
  public E pop() {
    E element = super.pop();
    if (element.compareTo(minStack.peek()) == 0)
      minStack.pop();
    return element;
  }
  public E findMin() {
    return minStack.peek();
  }
}
```

We can keep track of the minimums with a seperate min stack. When we push a new element onto the stack, we check to see whether this element is smaller than the old min. If it is, add it to the min stack. When we pop an element, we check to see if it is the current minimum on the min stack. If it is, we pop it off of the min stack as well. When we want to perform findMin, we can simply call peek on the min stack.

**Complexity:**
- Time: O(1)
- Space: O(n)

# Probability & Randomness

## Random range expansion **#random #tricky #probability**

**Question:**
Given a random number generator that gives us an integer between 1 and 5 (inclusive), write a function that generates a random number between 1 and 7 (inclusive).

**Answer:**
```java
public class Solution {
  public static int randomNumber7() {
    int values[][] = {
      { 1, 1, 1, 2, 2 },
      { 2, 3, 3, 3, 4 },
      { 4, 4, 5, 5, 5 },
      { 6, 6, 6, 7, 7 },
      { 7, 0, 0, 0, 0 }
    };

    int result = 0;
    while (result == 0) {
      int i = randomNumber5();
      int j = randomNumber5();
      result = values[i-1][j-1];
    }
    return result;
  }

  public static int randomNumber5() {
    // implemented for us
  }
}
```

We need to make sure we maintain the same probability that we choose any number between 1 and 7. In order to accomplish this, we use our working random function to select two numbers between 1 and 5. This gives us 25 possibilities. Keeping the probabilities even, we can have 3 sets of possibilities between 1 and 7 (3*7 = 21 < 25). This means there are possible outcomes in which we need to try again. For the rest of the numbers, we simply assign 3 numbers that correspond with each number from 1 to 7. Programmatically, this can be achieved in the most straight forward fashion by creating a two dimensional array of values, and using the generated numbers to select a value in the array. If the selected value is one of the values that should be thrown away, we select another pair of random numbers.

**Complexity:**

- Time: Worst case it runs forever, but the probability of this is infinitesimally small. The average case is between 1 and 2 selections.
- Space: O(1)

## Random number shuffling

**#probability** **#random**

**Question:**
Given an array of integers, write a function to shuffle the integers randomly, so that each permutation is equally likely.

**Answer:**
```java
public class Solution {
  public static int[] shuffleArray(int[] input) {
    for (int i = 0; i < input.length; i++) {
      int swap = i + (int) (Math.random() * (input.length - i));
      int temp = input[swap];
      input[swap] = input[i];
      input[i] = temp;
    }
    return input;
  }
}
```

We have to be careful to not give one element a larger probability to be swapped than any other in order to make each possible permutation equally likely. In order to accomplish this we can iterate over the array in order, swapping each element with a random element that comes after the current element.

**Complexity:**
- Time: O(n)
- Space: O(1)

## Coin flips to die rolls **#probability #random**

**Question:**
Given a function that gives us the result of a coin flip (1 or 2), write a function that gives us the result of a dice roll (1 to 6 inclusive).

**Answer:**
```java
public class Solution {
  public static int dieRoll() {
    int result = 0;
    while (result < 1 || result > 6) {
      result = 0;
      if (coinFlip() == 1)
        result |= 1;
      if (coinFlip() == 1)
        result |= 2;
      if (coinFlip() == 1)
        result |= 4;
    }
    return result;
  }

  private static int coinFlip() {
    // implemented for us
  }
}
```

There are many solutions to this problem. One solution is to flip the coin three times and use the results of the coin flips to decide whether or not to turn on each of the three lowest bits in an integer. This will give us a result between 0 and 7. If we get 0 or 7, we'll need to roll again and continue to do so until we get a valid die roll.

**Complexity:**
- Time: This could run forever, but the chance of that is infinitesimally small
- Space: O(1)

# Optimization

## Find nearest neighbors **#sorting**

**Question:**
Given array of people, each represented by their name (a string) and their position in line (an integer). Write a function that takes a name and returns their 3 nearest neighbors. We can assume for the sake of simplicity that there are at least 4 people in the list.

**Answer:**
```java
import java.util.Arrays;

public class Solution {
  public static Person[] findNearestNeighbors(Person[] people,
      String target) {
    Person[] neighbors = new Person[3];
    Arrays.sort(people);
    for (int i = 0; i < people.length; i++) {
      if (people[i].name.equals(target)) {
        int before = i - 1;
        int after = i + 1;
        int index = 0;
        while (index < 3) {
          if (after < people.length && (before < 0 ||
              Math.abs(people[before].position - people[i].position) >
              Math.abs(people[after].position - people[i].position))) {
            neighbors[index] = people[after];
            after++;
          } else {
            neighbors[index] = people[before];
            before--;
          }
          index++;
        }
      }
    }
    return neighbors;
  }

  private static class Person implements Comparable<Person> {
    public String name;
    public int position;

    @Override
    public int compareTo(Person other) {
      return ((Person) other).position - this.position;
    }
  }
}
```

We can solve this quite easily by first sorting the array by each person's position. The 3 nearest neighbors will then be located in either in the three positions immediately before or immediately after the target.

**Complexity:**

- Time: O(n log n)
- Space: O(1)

## Stock market timing

**#stock market** **#optimization**

**Question:**
Given an array of integers, where each element represents the stock price on consecutive days. Write a function to find the maximum profit if we're allowed to buy exactly once and sell exactly once over the time period.

**Answer:**
```java
public class Solution {
  public static int findBiggestProfit(int[] days) {
    int minPrice = days[0];
    int maxProfit = 0;
    for (int i = 1; i < days.length; i++) {
      if (days[i] < minPrice)
        minPrice = days[i];
      int currentProfit = days[i] - minPrice;
      if (currentProfit > maxProfit)
        maxProfit = currentProfit;
    }
    return maxProfit;
  }
}
```

We can go over the stock price data day by day keeping track of the lowest price we've seen so far, and the best profit we've seen so far. Best profit is updated when we find a stock price that, when the min price is subtracted, is greater than the old max profit.

**Complexity:**
- Time: O(n)
- Space: O(1)

**Question:**
Given an array, describe an algorithm to identify the subarray with the maximum sum. For example, if the input is [1, -3, 5, 2, 9, -8, -6, 4], the output would be [5, 2, 9].

**Answer:**
```java
public class Solution {
  public static int maxSubarray(int[] input) {
    int maxSoFar = 0;
    int maxEndingHere = 0;
    for(int i = 0; i < input.length; i++) {
      maxEndingHere = Math.max(0, maxEndingHere + input[i]);
      maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }
    return maxSoFar;
  }
}
```

While this may seem like a dynamic programming problem, we can further optimize the memory usage by only keeping track of the maximum we've seen so far and the maximum ending at the current position instead of memoizing these values across the entire array. For the max ending here value, we add the current value to the last max ending value if the sum is greater than zero, otherwise we reset the value to zero. We then use this value and check it against the maximum we've seen so far, and update it if the current value is greater.

**Complexity:**
- Time: O(n)
- Space: O(1)

## Path finding

**#a star, path finding**

**Question:**
Given a two dimensional grid with weights representing the cost of moving through each square, write a function that finds the minimum path between a given start and end cell.

**Answer:**

```java
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class AStar {
  public class Node {
    List<Node> neighbors = new ArrayList<Node>();
    Node parent;
    int xPos;
    int yPos;
    int fCost;
    int gCost;
    int hCost;
    int cost;
  }

  public List<Node> aStar(Node start, Node goal) {
    Set<Node> open = new HashSet<Node>();
    Set<Node> closed = new HashSet<Node>();

    start.gCost = 0;
    start.hCost = estimateDistance(start, goal);
    start.fCost = start.hCost;

    open.add(start);
    while (true) {
      Node current = null;
      if (open.size() == 0)
        throw new RuntimeException("No path exists");
      for (Node node : open) {
        if (current == null || node.fCost < current.fCost)
          current = node;
      }

      if (current == goal)
        break;

      open.remove(current);
      closed.add(current);

      for (Node neighbor : current.neighbors) {
        if (neighbor == null)
          continue;
        int nextG = current.gCost + neighbor.cost;
        if (nextG < neighbor.gCost) {
          open.remove(neighbor);
          closed.remove(neighbor);
        }

        if (!open.contains(neighbor) && !closed.contains(neighbor)) {
          neighbor.gCost = nextG;
          neighbor.hCost = estimateDistance(neighbor, goal);
          neighbor.fCost = neighbor.gCost + neighbor.hCost;
          neighbor.parent = current;
          open.add(neighbor);
        }
      }
    }

    List<Node> nodes = new ArrayList<Node>();
    Node current = goal;
    while (current.parent != null) {
      nodes.add(current);
      current = current.parent;
    }
    nodes.add(start);

    return nodes;
  }

  public int estimateDistance(Node node1, Node node2) {
    return Math.abs(node1.xPos - node2.xPos) +
        Math.abs(node1.yPos - node2.yPos);
  }
}
```

This question is a set up for us to use the A* path finding algorithm.

**Complexity:**

- Time: O(N) where N is the number of cells in the grid

# Design Patterns

## Listener design pattern

**#design patterns** **#listener** **#explanation**

**Question:**
Explain the listener design pattern.

**Answer:**

```
public class Listener {
  public static void listener() {
    EventSource eventSource = new EventSource();
    EventListener eventListener = new EventListener();

    eventSource.addListener(eventListener);
  }
}
```

This is a design pattern that is often used when handling user interface actions. The idea is that there are a group of "listeners" who are interested in a certain event happening. These listeners register themselves as listeners with the event source. When the event occurs, the event source knows to go through its list of listeners for that event, and notify them that the event has occurred. An example of this would be if we had a button on a page that was supposed to make a ball bounce on the page when clicked. The object controlling the ball would subscribe to an "onClick" event for the button. When the object receives a click event from the button, it knows to make the ball bounce. This is a very flexible design pattern that can be used in many situations.

## Model view controller **#model view controller #mvc #design pattern #explanation**

**Question:**
Explain the model view controller design pattern.

**Answer:**

```
public class Model {
  // Contains program state, variables, etc
}

public class View {
  // Contains user interface, text fields, buttons, etc.
}

public class Controller {
  // Listens for inputs from view and makes changes to model
  // Listens for model changes and makes changes to view
}
```

The model view controller (MVC) design pattern is used when developing user facing applications in order to separate the application's data from the user interface. The idea is that the user interface should be able the change without affecting the underlying data model and vice versa. The model contains the application state, the view contains the application's user interface, and the controller handles communication between the two.

## Singleton pattern

**#singleton #design patterns #explanation**

**Question:**
Explain the singleton design pattern.

**Answer:**

```java
public class Singleton {
  private static final Singleton instance;

  static {
    instance = new Singleton();
  }

  public static Singleton getInstance() {
    return instance;
  }

  private Singleton() {
  }
}
```

The singleton pattern is used in order to maintain a single instance of an object in our program. Singletons contain private constructors that are never called externally. Instead, the "getInstance()" method is called when an instance of the singleton object is needed. Singletons can be initialized either lazily (when the getInstance method is first called) or eagerly (when the object is created).

# Parallelism & Concurrency

## Threads and processes

**#parallelism #explanation**

**Question:**
Explain threads and processes.

**Answer:**

```java
public class MyThread extends Thread {

    public void run() {
      // Code that we want to execute on the thread
    }

    public static void main(String args[]) {
        (new MyThread()).start();
    }
}
```

Threads and processes are both paradigms that allow computers to run more than one piece of code at the same time. If our computer has multiple cores it can actually execute them at the same time, while single core machines only simulate this simultaneous execution using context switching. A process is a program that is run by a user. Our web browser runs on one processes, while our text editor runs on another. This makes it appear as if both are running at the same time. Each process can have one or more threads. Our web browser may run each window on a separate thread, so if we have two windows open at once, we can see each one functioning at the same time. Likewise, a program that has a user interface and is performing some heavy computation at the same time (like rendering graphics) will likely take user input and perform the processing on different threads. This allows clicks and key presses to be registered very quickly, without having to wait for the heavy computation to finish. Threads can share memory, while processes have separate chunks of memory allocated to them.

## Mutex vs semaphore

**#parallelism** **#explanation**

**Question:**
Explain the difference between a mutex and a semaphore.

**Answer:**

```java
import java.util.concurrent.Semaphore;

class UseSemaphore {
  private static final int NUM_RESOURCES = 100;
  private final Semaphore semaphore = new Semaphore(NUM_RESOURCES, true);

  public void useSemaphore() throws InterruptedException {
    semaphore.acquire();
    // Do something that requires one of the resources
    semaphore.release();
  }
}
```

Both mutexes and semaphores are used in parallel programming to prevent multiple threads from accessing a shared resource at the same time. A semaphore is used when we have a limited pool of resources. Let's say for example we don't want to have more than 5 connections open to a server at one time. We can create a semaphore with the limit of 5. Each time a thread wants to open a connection, it must acquire a resource from the semaphore, and then release it when it is done. A mutex is like a semaphore with only a single resource. Mutexes are like locks and are used to prevent concurrent access on a single object or resource.

## Synchronized methods

**#parallelism** **#java**

**Question:**
Explain Java synchronized methods.

**Answer:**

```java
public class SynchronizedCounter {
  private int count = 0;

  public synchronized void increment() {
    count++;
  }

  public synchronized void decrement() {
    count--;
  }

  public synchronized int value() {
    return count;
  }
}
```

Objects in Java have their own mutexes, or locks. When a method is called that is synchronized, the object's mutex is locked. When the method is finished, the mutex is unlocked. This ensures only one synchronized method can be called on an object at a time.

## Preventing deadlocks

**#parallelism** **#deadlocks**

**Question:**
Explain one way to prevent deadlocks.

**Answer:**

```java
public class Deadlock {
  public static void deadLockThreadOne() {
    lockA();
    lockB();
  }

  public static void deadLockThreadTwo() {
    lockB();
    lockA();
  }
}
```

Deadlocks can be prevented by having a well defined access order when accessing multiple resources that require locks. This prevents the case where two threads are competing for the same resources, and one thread obtains a lock for resource A at the same time as another thread obtains a lock for resource B. Now thread one needs to obtain a lock for B, while the other thread is waiting to obtain the lock for A. The two threads will wait forever. If both threads obtained lock A first, then lock B, this situation would not happen.

# Matrix manipulation

## Matrix rotation in place

**#matrix manipulation** **#tricky**

**Question:**
Write a function that rotates an two dimensional square matrix 90 degrees using O(1) additional memory.

**Answer:**

```
public class Solution {
  public static void rotateMatrix(int input[][]) {
    int n = input.length;
    if (n <= 1)
      return;

    for (int i = 0; i < n / 2; i++) {
      for (int j = i; j < n - i - 1; j++) {
        int temp = input[i][j];
        input[i][j] = input[n - j - 1][i];
        input[n - j - 1][i] = input[n - 1 - i][n - 1 - j];
        input[n - 1 - i][n - 1 - j] = input[j][n - 1 - i];
        input[j][n - 1 - i] = temp;
      }
    }
  }
}
```

The key to this problem is that we have to rotate 4 numbers at once, storing the first value that is overwritten as a temporary variable and then restoring it in the end. We work our way from the outside to the inside in "layers" performing these four way swaps until we reach the center.

**Complexity:**

- Time: O(n)
- Space: O(1)

## Contact Us

Questions?

Comments?

Suggestions?

Email us at interviewsolutionsmanual@gmail.com!

Or visit us at www.InterviewSolutionsManual.com