SOFTWARE ENGINEER SAMPLE INTERVIEW PROBLEMS AND SOLUTIONS

Problem 1: Strings and Permutations

Given a string, write an algorithm to determine if it is a permutation of a palindrome.

Solution

A palindrome is typically defined as a word or other sequence that reads the same both forwards and backwards; a permutation is one of many ways a sequence can be rearranged. Thus, a permutation of a palindrome is a set of characters rearranged in any order with none added or deleted, also known as an anagram.

For example, "racecar" is a palindrome, and "car race" is an anagram of it. In natural languages, anagrams may add or remove spaces to form new words, but in programming languages, spaces count as characters in strings, so the number of characters in permutations of a palindromic string must always remain the same.

Consequently, an algorithm in a program designed to find permutations of palindromes might begin by counting the number of characters in each string. If the number of characters in string1 equals the number in string2, for example, the program would move on. Otherwise, the program would return false to indicate string2 is not a permutation of string1.

If the program continued, it would then sort both strings into alphabetical order individually to check for matching characters. Finally, it would compare each character in string1 to the corresponding character in the same position of string2. If the program encountered a mismatch, the result would be false. If all characters match, the result would be true, and string2 would be a permutation of a palindrome of string1.

Pseudocode

```
Palindrome_Permutation_Test()

        length1 = lengthOfString1
        length2 = lengthOfString2

        IF (length1 ? length2)
        return false

        string1 = string1 sorted alphabetically
        string2 = string2 sorted alphabetically

        FOR (i = 0, i < length1, i++)
                IF (string1 at index i ? string2 at index i)
                        return false
                ELSE
                        return true
```

Problem 2: Linked Lists

Given two singly linked lists, determine if the two lists have an intersecting node (based on reference, not value).

Solution

A linked list is a set of elements organized linearly using pointers. In other words, each node of this data structure contains both an item and a reference to the location of the next item on the list. It is not necessarily arranged contiguously in the computer's memory, so two linked lists may intersect. If lists intersect at any point, they will always merge into one list, and the references at the end of each list will point to the same place in the computer's memory.

Thus, one way to solve this problem would be to iterate through both linked lists separately starting from head1 and head2 and using pointer1 and pointer2, respectively. When pointing to the next item returns null, the algorithm will have found the last item on the current list. Then, the program could compare the last item of each list for equality. If the comparison finds them equal, the lists will have indeed intersected.

Note: The following pseudocode assumes neither linked list is circular.

Pseudocode

```
Intersection_Test()

        IF head1 = null
                return false

        IF head2 = null
                return false

        pointer1 = head

        WHILE pointer1.next ? null
                pointer1 = pointer1.next

        pointer2 = head

        WHILE pointer2.next ? null
                pointer1 = pointer2.next

        IF pointer1 = pointer2
                return true
```

Problem 3: Stacks

Design a function called ‘Min()’ which returns the minimum element within a stack.

Solution

A stack is a data structure that works precisely as the name suggests. In a stack of books, a reader can only view the cover of the book on top and would need to remove it to see the cover of the next one. The same is true of a programming stack. A program can peek at the top item without removing it, but it must pop the item from the stack to access the next one.

To find the minimum element, an algorithm could peek at the first element and store its value as the current minimum. Next, the program could pop off the next element and compare the two values. The lower value would become the current minimum. When no elements remain on the stack, the program could then return the final minimum value.

Pseudocode

```
Min()

        min = stack.peek
        WHILE stack is not empty
                value = stack.pop
        if min < value
                min = value
```

Problem 4: Binary Trees

Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth (e.g., If you have a tree with depth *N*, you will have *N* linked lists).

Solution

A binary tree is a data structure that resembles an upside-down tree sprouting branches, limbs, and/or leaves. Like a real, organic tree, it begins from a single root, but each offshoot can have only zero, one, or two offshoots at most, hence its description of "binary." The root node and all other nodes that branch to at least one node are called "parents," and the nodes to which the parents branch are called "children."

There are four options for accessing each node of a binary tree: preorder, in-order, post-order, and level-order. It would be possible to use any of those methods to traverse the tree and create items in a linked list that correspond to each node.

Since the problem asks for a separate linked list for each level of depth, however, the best choice is level-order traversal, also known as a breadth-first search. Using level-order traversal means the algorithm would visit all the nodes at each level, store the value of each node as a corresponding element in a linked list, move to the level below it, make a new linked list based on the values of the nodes at that level, and continue until all levels have been traversed.

For example, if a program traversed the following binary tree by level order,

it would produce these linked lists.

1
2 ? 3
4 ? 5 ? 6 ? 7

As the example demonstrates, the resulting linked lists fulfill the problem's requirement to represent all nodes at each depth of the binary tree. One way to code this solution would be to iterate through all the nodes at one level, store their values in a queue, add each value in the queue to a new linked list, remove all the nodes from the queue, move to the next level, repeat until the queue is empty, and return all the linked lists.

Pseudocode

```
Binary_Tree_to_Linked_List()

        make queue1
        add root to queue1

        WHILE queue1 is not empty
                make linked list for current level
                        FOR each node in queue1
                                remove node
                        add to linked list and add children to queue2
                replace queue1 with queue2

        return all linked lists
```

Problem 5: Weighted Graphs

Given a directed and weighted graph, design an algorithm that will return the weight of the shortest path between two points, if one exists.

Solution

A directed graph consists of a set of points connected by lines that end in directional arrows, much like a map of one-way and/or two-way streets that may or may not have dead ends and/or circular routes in which the destination is the same as the point of departure. When a graph is weighted, that means each line or "street" has a numerical value (i.e., a weight).

A classic solution for finding the shortest path between two points on a directed, weighted graph is Dijkstra's shortest path algorithm. Although this approach is efficient, however, it does not work for a directed graph with negative weights. Since the variables in this problem are unknown, an extension of Dijkstra's algorithm that allows for the possibility of negative weights, the Bellman-Ford shortest path algorithm, would be a safer, more thorough solution. This algorithm progresses as follows.

1. Make a list of all the edges (or connecting lines) of the graph.

2. Find the number of iterations by subtracting one from the number of vertices (V - 1). The minimum distance to an edge can be updated no more than the total number of vertices minus the current one. As with Dijkstra's algorithm, the value for the current vertex, the point of departure, begins as zero, and the values of the rest of the vertices start at infinity.

3. Check the current value of each vertex, and compare it to the distance just determined. Reset the distance of each to the lesser value.

4. Ensure every existing path has been taken into consideration as a candidate for the shortest path.

5. If a cycle contains edges that add up to a negative value, each trip around such a path would artificially reduce the length of the entire path, so the results would be invalid. Prepare to detect and report that situation as an error.

The following pseudocode is an implementation of the Bellman-Ford algorithm revised from the Wikipedia entry on the subject into simpler language.

Pseudocode

```
Shortest_Path()

        FOR each vertex
                distance to all vertices == infinity
                distance from currentVertex to itself == 0
                predecessor list == 0

        FOR 1 to V ? 1
                FOR all edges
                        IF the distance from vertex1 + weight < distance from vertex2
                                the distance of vertex1 == distance of vertex2 + weight
                                the predecessor of vertex1 == vertex2

        FOR all edges
                IF the distance of vertex1 + weight < distance of vertex2
                        throw an error because the graph contains a negative-weight cycle

        return the predecessor and distances as the shortest path
```