**Homework 03 -- Questions 1 - 5**

1. Can we store a duplicate key in HashMap? How does HashMap handle collisions in Java?

We cannot store duplicate keys in a HashMap – all keys must be unique. A collision occurs when two different keys map to the same address in the HashMap. Collisions can be handled using either open addressing or closed addressing. With open addressing, each address in the HashMap can only store one item. As such, when two keys overlap, the collision is resolved by checking several alternate addresses to store the second key at. With closed addressing, each address in the HashMap stores a doubly-linked list of values. This means that when a collision occurs, the second key is appended to the linked list that already holds the first value. When retrieving items from the HashMap, the hash function tells us which linked list to check and traverses the linked list until the key is found.

In Java, collisions are handled with closed addressing. In previous versions of Java, this was done using linked lists for each position in the HashMap, but in more recent versions the linked lists were replaced with binary trees in order to improve the worst-case runtime of searching the chain of colliding keys from O(n) to O(log(n)).

2. Give some name of sorting algorithms. Among them which one you think is fastest?

Some examples of sorting algorithms are selection sort, insertion sort, bubble sort, merge sort, quick sort, and bucket sort. Among these, only merge sort, quick sort, and bucket sort have average runtimes of O(n\*log(n)), making these better than selection sort, insertion sort, and bubble sort. Bucket sort technically has a worst-case runtime of O(n2), but this is for the case where all items are placed in the same bucket. If we design our algorithm such that this scenario is statistically almost impossible, the bucket sort’s worst-case runtime falls to O(n\*log(n)), similar to merge sort and quick sort. However, bucket sort has a best-case runtime of O(n), making it slightly better than these two other sorts.

Between merge sort and quick sort, quick sort has the better asymptotic worst-case runtime for most scenarios, making it faster. For small input sizes, however, some sorts like insertion sort can perform better than quick sort since they avoid recursive function calls.

3. How merge sort works? What is the time complexity for merge sort?

Merge sort uses the idea of divide-and-conquer algorithm design to approach the problem of sorting recursively. This concept suggests that we start by dividing the input set into multiple smaller subsets and recursively sort each subset. This subset-division continues until we have subproblems of a constant size. For merge sort, this final subproblem occurs when there is only one item in a set. Once we have reached the final level of subproblems, we have many individual items in their own subsets. We then move back up the recursion stack by merging two subsets at a time, placing the items in their sorted order as the merging takes place (this step has time complexity of O(n), since we must iterate through each item in one of the subsets, merging its items into the other subset). With two recursive subproblems per function call, merge sort has a time complexity of O(n\*log(n)). This makes sense, since the merging has time complexity of O(n) and each recursive call reduces the input size from n to n/2, which implies O(log(n)) for the recursion step.

4. How can we traverse all the nodes of graph?

In general, to traverse all the nodes of a graph, we must traverse all of the nodes of each connected component of the graph (since starting at any one node will never allow us to traverse nodes outside of the starting node’s connected component). To traverse all of the vertices (nodes) of a single connected component of the graph, we can choose a starting vertex and visit each of the vertices connected to the starting vertex by its edges. When vertices or edges have been visited, we can mark them as such so that we do not visit them again. By doing this continually for each vertex, we will eventually traverse all of the vertices.

This general concept gives rise to two general traversal techniques for the nodes of graphs: Depth First Search and Breadth First Search. Each of these techniques produces a spanning forest of the original graph (that is, a subgraph containing all of the vertices in the graph, but not necessarily all of the edges, since each tree must maintain a hierarchical structure). For Depth First Search, we visit the starting vertex and mark it as visited, and then for each edge of that vertex, if the opposite vertex has not been visited, the edge leading to that vertex is marked as a “discovery edge.” We then call the function recursively for the vertex that has not been visited. As we discover vertices and edges, we create a spanning tree of the connected component formed by the vertices and the discovery edges (those that led to new vertices). This algorithm runs in O(n + m) time, where n is the number of vertices and m is the number of edges, since we must visit each vertex and edge once.

Breadth First Search uses a slightly different algorithm to visit all of the nodes of a graph while creating a spanning forest of the graph. For each connected component, the starting vertex is placed in a sequence, which we denote as a level (L0). Then, we create a new level (L1), and for each edge of the starting vertex, we check if the vertex opposite the starting vertex has been visited. If it has not, we add it to the new level (L1) and mark that edge as a discovery edge. Once we have finished checking each edge, we set the current level to the next level (moving on to L1 with new level L2). This process continues until the current level has no vertices in it (that is, we can only reach previously visited vertices). This processing from one level to the next is accomplished using a while loop that continually checks if the current level has no vertices, and thus ends when we have visited every vertex. Breadth First Search does not use recursion, but similarly has runtime O(n + m) for n vertices and m edges, since each is visited once. The discovery edges and visited vertices again form a spanning forest of the graph.

5. Write the applications of Breadth First Search (BFS) and Depth First Search (DFS).

Depth First Search can be used for pathfinding or for finding a cycle in a graph. For pathfinding, we modify the DFS such that its parameters are the input graph, the start vertex, and the end vertex. Then, we keep track of the visited vertices using a stack. If at any point the current vertex equals the end vertex, we return the elements of the stack. First, we add the current vertex to the stack. Then, for each incident edge, if the vertex opposite the current vertex is unexplored, we add that edge to the stack and then call the method recursively. After the recursive step, we pop the edge off the stack. This step only occurs if the path explored from that incident edge did not reach the end vertex. If each path from a vertex does not lead to the end vertex, the vertex itself is popped off the stack. This pushing and popping allows us to visit edges and vertices until a path is found.

For cycle finding, we similarly use a stack, but we visit unexplored edges until we find a back edge. Once an unexplored edge to a visited vertex is found, we know the stack contains a cycle, and thus we can return the elements of the stack.

Breadth First Search can be used to find a path between vertices with the minimum number of edges or find a simple cycle in a graph. Since BFS divides the graph into levels, we can find the shortest path between vertices by traveling from one level to the next until we reach the end vertex (that is, the shortest path has edges (L – 1) for L levels). We can also find a simple cycle by finding back edges, similar to how DFS works.

Both DFS and BFS can also be used to compute the spanning forest and connected components of a graph by definition.

**Programming Problems (6 – 10)**

Question 6:

Discussion:

HW3Question6.java contains the code for this problem. The algorithm for finding two numbers that add up to the target value starts by creating a HashMap of entries where the key is the number itself and the value is an ArrayList of all indices in the array of numbers where that number occurs. Then, for each value in the array of numbers, it computes the difference between the target value and that value. It then checks if the HashMap contains that difference as a key, and if it does, it finds the first index in the corresponding ArrayList that does not match the index of the first value (so that the same value is not used twice). If no pair of indices match this condition, the algorithm returns “No such pair exists.”

The time complexity to add each entry to the HashMap is O(n). The time complexity for checking each value in the original array is also O(n), since we check each value until a valid pair is found. Since O(n + n) = O(n), the overall time complexity of the algorithm is O(n). The space complexity of the algorithm is also O(n) since we store each of the values and their indices in the HashMap.

Verification:

We will verify the algorithm using several test cases where the target pair exists and doesn’t exist, and a few cases with duplicate values.

Input: [3,3], target: 6

Output: [0,1]

Input: [3], target: 6

Output: No such pair exists.

Input: [2,5,1,61,3], target: 66

Output: [1,3]

Input: [2,5,1,61,3,5,4], target: 10

Output: [1,5]

Question 7:

Discussion:

HW3Question7.java contains the code for this problem. The algorithm for finding the longest substring without repeating characters initializes a HashMap that will store each character in the input string. Using a while loop that checks that the current index in the string has not exceeded its length and that the character at that index does not exist in the HashMap, we add each non-repeating character to the HashMap, incrementing the length variable each time. Once we find a repeating character (that is, the while condition fails but the index is not yet equal to the length of the string), we recursively call the method for the substring after the first instance of the character that repeated. If that recursive call returns a length greater than the current one, it replaces the current value as the largest length. This way, we check every longest substring with non-repeating characters and find the length of the largest one.

The time complexity of this algorithm is O(n2). Each recursive call takes as input a substring excluding the characters up to the repeated character (each recursive call has input size (n – k) for some initial substring of length k). However, if a recursive call occurs, we know that the previous call had input size j for some substring of length j since it only checked the characters up to the repeating character. The value of j will always be less than or equal to the length of the longest substring, and the larger it is, the fewer recursive calls are needed. The worst case is that the substring repeats perfectly several times across the string (ex: abcabcabcabc), since we must recursively call the method for each character in the string until we find the final non-repeating string. Each recursive call has input size of the length of the substring (we will denote as m <= n), and the number of recursive calls is equal to the length of the string minus the length of the substring plus one (n – m + 1), since we call the function for each character until the final substring occurs at index (n – m). Thus, the number of operations required is m\*(n – m + 1). We can ignore the + 1, giving us m\*(n – m). Since m + (n – m) = n, we have a function where want to find the maximum product of two numbers whose sum is n. This generally occurs around the value (n / 2). Therefore, we have time complexity O(n / 2)2, or O(n2).

The space complexity of this algorithm is also O(n2), since we store a character in a HashMap for each operation of the algorithm.

Verification:

We will verify this algorithm using several strings with repeating characters in different locations.

Input: “abcjdhejabcfya”

Output: 9

Input: "abcabcabcabc"

Output: 3

Input: "bbbbb"

Output: 1

Input: "abcdefg"

Output: 7

Question 8:

Discussion:

HW3Question8.java contains the code for this problem. The algorithm for merging the intervals uses a while loop and a recursive function. The while loop calls the recursive function, adding its results to the merged list of intervals, until the original list of intervals is empty. The recursive function takes the first interval from the list and merges it with any overlapping intervals, removing the original intervals from the original list. If a merge occurs, the function calls itself with the new merged interval as input. Eventually, the original list becomes empty, ending the while loop.

The merging itself is done through methods of the Interval class found in the Interval package. The Interval class implements Comparable – the specific implementation considers two intervals “equal” if they have overlapping values. The intervals are merged by taking the smallest starting value and the largest ending value of the two intervals and creating a new interval from those.

The time complexity of this algorithm is O(n). No matter how many merges are necessary, each interval from the original list is checked only once. The space complexity is also O(n) since we store the intervals in several ArrayLists.

Verification:

We will verify the algorithm using several sets of intervals with overlapping and non-overlapping intervals.

Input: {{1,3},{2,6},{8,10},{15,18}}

Output: [[1, 6], [8, 10], [15, 18]]

Input: {{1,3},{2,6},{8,10},{15,18},{6,18}}

Output: [[1, 18]]

Input: {{1,500},{405,601},{700,800},{801,829},{814,900}}

Output: [[1, 601], [700, 800], [801, 900]]

Question 9:

Discussion:

HW3Question9.java contains the code for this problem. The algorithm uses a method called numIslands() that calls a recursive support method called exploreIsland(). The first method checks every coordinate in the grid, and if that coordinate has a value of “1”, it calls the second method. The second method marks a coordinate as explored if it contains a value of “1” (This repeated check is necessary for recursion) by changing its value to “2”. It then calls itself recursively for each valid adjacent coordinate (a coordinate is valid if its row and column indices are not outside the bounds of the grid). This way, the entire island will be marked as explored once the numIslands() method calls exploreIsland(). We can then increment a counter in numIslands() and continue to check cells in the grid. Once each cell has been checked, we return the number of islands.

The runtime complexity for this algorithm is O(m\*n), where m is the number of rows in the grid and n is the number of columns. The numIslands() method checks each cell in the grid, and as such has a runtime O(m\*n). The recursive exploreIslands() method will at worst visit each cell in the grid, giving it a worst-case runtime of O(m\*n) as well, and as such our total runtime is O(m\*n). The space complexity of this algorithm is O(1) since we do not need to create any data structures for this algorithm – the only memory space we use is for the temporary variables used in the for loops.

Verification:

We will verify this algorithm using several grids of different sizes with different numbers of islands.

Input: grid = {

{'1','0','1','1','0'},

{'0','1','0','1','0'},

{'1','1','0','0','1'},

{'0','0','1','1','0'}

};

Output: 5

Input: grid = {

{'1','0','1','1','1','1','0'},

{'1','1','1','0','0','0','1'},

{'1','0','1','0','0','1','0'},

{'1','0','1','0','0','1','0'},

{'1','0','1','0','0','1','0'},

{'1','1','0','1','0','1','1'},

{'1','0','1','1','1','1','0'}

};

Output: 3

Input: grid = {

{'1','1','1','0'},

{'0','1','1','0'},

{'1','1','0','1'},

{'0','1','1','0'}

};

Output: 2

Question 10:

Discussion:

HW3Question10.java contains the code for this problem. In order to determine a valid order to take courses in, we first initialize two HashMaps. The first HashMap stores as a value a list of courses that are “unlocked” by taking the course stored as the key. The second HashMap stores as a value a list of prerequisite courses for the course stored as the key. Once these have been initialized, the main function calls a recursive function called takeCourses(). This function starts by adding to the final course list all courses that have no remaining prerequisite courses to take. Then, for each course that was added, we check the courses unlocked by taking that course. For each unlocked course, the course that was taken is removed from the prerequisite list in the second HashMap. We can then call the function recursively until the number of courses in the final list matches the total number of courses to take.

Additionally, we need to be able to detect a cycle in the adjacency lists of the prerequisites to determine if the course sequence is impossible to create. *Geeks for Geeks* has a helpful implementation of cycle detection for graphs implemented as adjacency lists that I used for this problem (Source: “Detect Cycle”). The algorithm consists of an outer method and a recursive method. The outer method allows us to iterate over all the nodes in the graph. Since the recursive method only checks one connected component at a time, we need this outer method to be able to check all of the trees if the graph is a forest. The recursive method takes as input a current node, a Boolean array of visited indices, and a Boolean array for the recursion stack. At the beginning of the method, we check if the current node is marked in the recursion stack array or the visited array. If it is marked in the recursion stack array, we have a cycle and we return true. If it is marked in the visited array, we have already checked this node and its tree in an earlier function call, so we return false to return to the outer method.

When we visit a node, we mark it as visited in the visited array and in the recursion stack array. Then, we call the method recursively for the adjacent nodes. If we reach the end of the recursion for a particular path, we unmark the nodes from the recursion stack array for that path (so that we know those nodes are not part of a cycle). Thus, if all nodes are visited but no visited node was already part of the recursion stack array, we know there is no cycle.

The runtime complexity of this program is O(n). Initializing the Hashmaps has runtime O(n). Furthermore, since we only add each course to the final list once, this also has O(n) runtime. The time complexity of the cycle detection is O(n + m) for n vertices and m edges, although this graph implementation does not contain edges, so this simplifies to O(n). The space complexity of the program is also O(n), since the HashMaps, Boolean arrays, and final course list all store each course once.

Verification:

We will verify this program with several prerequisite lists with different numbers of courses and some that are impossible to complete.

Input: numCourses = 2; prerequisites = {{0,1},{1,0}}

Output: []

Input: numCourses = 5; prerequisites = {{1,0},{2,0},{3,4},{3,2}};

Output: [0,1,2,4,3]

Input: numCourses = 3; prerequisites = {{1,0},{2,1},{0,2}};

Output: []

Input: numCourses = 8; prerequisites = {{1,0},{2,1},{7,5},{7,4},{6,7},{3,0},{7,3}};

Output: [0,1,2,3,4,5,7,6]

Works Cited

“Detect Cycle in a Directed Graph.” *Geeks for Geeks*. 22 Mar, 2023, <https://www.geeksforgeeks.org/detect-cycle-in-a-graph/>. Accessed 11 Apr. 2023.