CS 1332 Studypalooza Worksheet ANSWERS

The final exam is cumulative and covers everything taught this semester! This document has many, many questions but does not touch on every topic, so we would recommend reviewing any missing topics.

Document outline:

1. Big-O Review Tables
   a. Data Structures Table
   b. Sorting Algorithms Table
   c. Pattern Matching Table
   d. Graph Algorithms Table

2. Multiple Choice Questions

3. Diagramming Questions
   a. Tree Diagramming (BST, AVLs, 2-4 Trees)
   b. BFS + DFS
   c. Dijkstra's Algorithm
   d. Prim's Algorithm
   e. Kruskal's Algorithm
   f. LCS

4. Coding Questions
   a. Linked List Coding Question
   b. BST Coding Question
   c. Sorting Algorithms Coding Question
   d. Pattern Matching Coding Question
   e. Graphs Coding Question

5. Scenario Questions

If you notice a mistake on this answer key, please let us know!
# Data Structures Big-O Table

Note: Since these data structures all do different things, some of the categories may not apply (i.e. search for Stack) - write "N/A" if it does not apply. Some of the table is already filled out. We are assuming **worst-case time complexity with amortized analysis** (denoted with an asterisk). Feel free to copy this table and fill it out for average-case analysis.

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>first index</td>
<td>last index</td>
<td>any index</td>
<td>first index</td>
<td>last index</td>
<td>any index</td>
<td>at index</td>
<td>for specific value</td>
</tr>
<tr>
<td>Arrays</td>
<td>O(1)</td>
<td>O(1)*</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>ArrayLists</td>
<td>O(n)</td>
<td>O(1)*</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>SLL, no tail</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>N/A</td>
</tr>
<tr>
<td>SLL, tail</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>N/A</td>
</tr>
<tr>
<td>CSL</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>N/A</td>
</tr>
<tr>
<td>DLL</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>N/A</td>
</tr>
<tr>
<td>Stack</td>
<td>O(1)</td>
<td>O(1)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>O(n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td>O(1)</td>
<td>O(1)</td>
<td>N/A</td>
<td>N/A</td>
<td>O(n)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>O(n)</td>
<td>O(n)</td>
<td>N/A</td>
<td>O(n)</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>N/A</td>
<td>N/A</td>
<td>O(n)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Map</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>N/A</td>
<td>O(n)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVL</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>N/A</td>
<td>O(log n)</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skip List</td>
<td>O(n)</td>
<td>O(n)</td>
<td>N/A</td>
<td>O(n)</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-4 Tree</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>N/A</td>
<td>O(log n)</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Sorting Algorithms Table

<table>
<thead>
<tr>
<th>Sorting Algorithm</th>
<th>Big-O</th>
<th>stable?</th>
<th>in-place?</th>
<th>adaptive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble sort</td>
<td>O(n^2)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cocktail shaker sort</td>
<td>O(n^2)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>selection sort</td>
<td>O(n^2)</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>insertion sort*</td>
<td>O(n^2)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>merge sort</td>
<td>O(n log n)</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>quick sort</td>
<td>O(n^2)</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>radix sort</td>
<td>O(kn)</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

*Insertion sort is not on the final exam for HB

## Pattern Matching Table

<table>
<thead>
<tr>
<th></th>
<th>best-case</th>
<th>worst-case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>single occur.</td>
<td>all occur.</td>
</tr>
<tr>
<td>Brute Force</td>
<td>O(m)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Boyer-Moore (no Galil rule)</td>
<td>O(m)</td>
<td>O(m + n/m)</td>
</tr>
<tr>
<td>KMP</td>
<td>O(m)</td>
<td>O(m + n)</td>
</tr>
<tr>
<td>Rabin-Karp</td>
<td>O(m)</td>
<td>O(m + n)</td>
</tr>
</tbody>
</table>
For simplicity, represent \(|E|\) as \(E\) and \(|V|\) as \(V\). However, on the exam, make sure to include the cardinality sign!

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (or Space) Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacency Matrix (space complexity)</td>
<td>(O(V^2))</td>
</tr>
<tr>
<td>Adjacency List (space complexity)</td>
<td>(O(V + E))</td>
</tr>
<tr>
<td>Edge List (space complexity)</td>
<td>(O(E))</td>
</tr>
<tr>
<td>Depth-First Search</td>
<td>(O(V + E))</td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>(O(V + E))</td>
</tr>
<tr>
<td>Dijkstra's Algorithm</td>
<td>(O((V + E) \log V))</td>
</tr>
<tr>
<td>Prim's MST Algorithm</td>
<td>(O((V + E) \log V))</td>
</tr>
<tr>
<td>Kruskal's MST Algorithm</td>
<td>(O(E \log E))</td>
</tr>
</tbody>
</table>
Multiple Choice Questions

1. Binary Tree and BST
2. A
3. C
4. E
5. B, D, E
6. D
7. D

Diagramming Questions:

Tree Diagramming:

1a.
1b. Any BST with nodes 13, 21, 27, 31, 49

1c.
1d.
Graphs Diagramming

For the DFS, BFS, and Dijkstra's problems below, if there exists a tie on the next vertex to traverse, always choose the vertex that comes first alphabetically.

1. B A C D F G E H
2. H D A B F C G E

Visited Set:
G B C A E F H D
Current vertex:
Priority Queue: (D, 14)

Distance Map:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

3.

4. Vertices: F B C D A G H E
   Edges: FB, FC, BD, CA, FG, GH, GE
5. BD, CF, CG, BC, AD, CE, DH are edges visited as well as MST

6. rone or roze

7. LCS: CAGTTA, CAGTCA, CGATCA, GGATCA
public class LinkedList {
    int size;
    Node head;
    Node tail;

class Node {
    int data;
    Node next;

    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }
}

public void addLast(int data) {
    Node newNode = new Node(data, null);
    this.tail.next = newNode;
    this.tail = newNode;
}

public int removeLast() {
    Node current = this.head;
    int data = this.tail.data;
    // traverse to second-to-last Node
    for (int i = 0; i < size - 1; i++) {
        current = current.next;
    }
    current.next = null;
    this.tail = current;
    return data;
}
BST Coding Question

A node in a BST is a proud parent if the sum of all of its children (including grand-children, great-grand-children, etc.) is divisible by 4 and greater than 0.

For example, in the following tree, the nodes at 10 and 12 are both proud parents:
- Node 10: $3 + 1 + 4 + 12 + 11 + 13 = 44$ (divisible by four!)
- Node 12: $11 + 13 = 24$ (divisible by four!)

Write a function, countProudParents(), that counts the number of proud parents in a BST. You must implement this recursively. Notice the class IntegerWrapper - we'll have to keep track of two values during recursion, the sum of a node's children as well as the total number of proud parents. Using the IntegerWrapper will let you keep an integer value across different function calls, how should you use this class for your solution? Here is some starter code:

```java
public class BST {
    int size;
    Node root;

    class Node {
        int data;
        Node right;
        Node left;
        // constructor not shown
    }

    // counts the number of nodes whose children's sum is nonzero and divisible by four.
    public int countProudParents() {
        IntWrapper wrapper = new IntWrapper();
        wrapper.value = 0;
        proudParentsHelper(this.root, wrapper);
        return wrapper.value;
    }

    // helper function for recursive counting proud parents
    private void proudParentsHelper(Node node, IntWrapper wrapper) {
        // implementation details
    }
}
```

```java
class IntWrapper {
    int value;
}
```
public int proudParentsHelper(Node current, IntWrapper wrapper) {
    if (current == null) {
        return 0;
    }
    int childrenSum = proudParentsHelper(current.left, wrapper)
        + proudParentsHelper(current.right, wrapper);
    if (childrenSum > 0 and childrenSum % 4 == 0) {
        wrapper.value++;
    }
    return childrenSum + current.data;
}

class IntWrapper {
    int value;
}
You are given the following starter code for the QuickSort algorithm. Fill in the code so that this code is a valid QuickSort algorithm.

```java
import java.util.Random;
public class Sorting {

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

    public static void quickSort(int[] arr) {
        if (arr == null || arr.length <= 1) {
            return;
        } else {
            quickSort(arr, 0, arr.length);
        }
    }

    private static void quickSort(int[] arr, int start, int end) {
        if (Math.abs(start - end) <= 1) {
            return;
        }
        // Select pivot index and swap pivot
        int pivotIndex = rand.nextInt(end - start) + start;
        int pivot = arr[pivotIndex];
        swap(arr, start, pivotIndex);
        int i = start + 1;
        int j = end - 1;
        while (i <= j) {
            while (i <= j && arr[i] <= pivot) {
                i++;
            }
            while (i <= j && arr[j] > pivot) {
                j--;
            }
            if (i <= j) {
                // i and j are changed AFTER the swap occurs
                swap(arr, i++, j--);
            }
        }
    }
}
```
// Restore pivot and recurse
swap(arr, start, j);
quickSort(arr, start, j);
quickSort(arr, j + 1, end);
}
Pattern Matching Coding Question

KMP Failure Table Coding

Given the following code snippets for the KMP Failure Table algorithm, fill in the blanks in the provided code with the appropriate answer. The method should build and return the failure table used in the KMP pattern matching algorithm. You can use more than 1 line of code for question 4 and question 5 if necessary, but only for those questions.

```java
public static int[] buildFailureTable(CharSequence pattern, CharacterComparator comparator) {

    //null checks for pattern and comparator are omitted, you can assume a valid input.
    //recall that the character in the pattern is accessed with the method .charAt() and the comparator has a .compare method
    int[] failureTable = new int[pattern.length()];
    failureTable[0] = 0
    int i = 0;
    int j = 1;
    while (j < pattern.length()) {
        if (comparator.compare(pattern.charAt(i), pattern.charAt(j)) == 0) {
            i++;
            failureTable[j] = i;
            j++;
        } else if (i == 0) {
            failureTable[j] = 0;
            j++;
        } else {
            i = failureTable[i - 1];
        }
    }

    return failureTable;
}
```

Hint: try to make the failure table for a pattern, i.e. ababcdcd, and think of what is happening in terms of code to get those values

Graphs Coding Question
public class Graph {

    class Vertex {
        int data; // constructors and other methods not shown
    }

    class Edge {
        Vertex u;
        Vertex v;
        int weight; // constructors and other methods not shown
    }

    class Graph {
        Set<Vertex> vertices;
        Set<Edge> edges;
        Map<Vertex, List<Vertex>> adjacencyList; // constructors and other methods not shown
    }

    class Queue<T> {
        public void push(T data) { /* implementation not shown */ }
        public int pop() { /* implementation not shown */ }
        public int size() { /* implementation not shown */ }
    }

    // returns an ArrayList with vertices in order of exploration for the BFS algorithm.
    public static ArrayList<Vertex> bfs(Vertex start, Graph graph) {
        if (start == null || graph == null || !graph.getVertices().contains(start)) {
            throw new IllegalArgumentException();
        }
        ArrayList<Vertex> visitedVertices = new ArrayList<>();
        Queue<Vertex> queue = new Queue<>();
        queue.push(start);
        visitedSet.add(start);
        while (queue.size() > 0) {
            Vertex current = queue.pop();
            for (Vertex neighbor : graph.adjacencyList.get(current)) {
                if (!visitedSet.contains(neighbor)) {
                    visitedSet.add(neighbor);
                    queue.add(neighbor);
                }
            }
        }
    }
Graphs Coding Question Pt 2

Prim's/Kruskal's Coding

Below you are given a bank of unordered code lines from A to Q where some of the code lines from the code bank complete Prim's or Kruskal's algorithm as implemented in the homework.

Students in HB's section should write the order of the steps for Prim's algorithm.
Students in Moss' section should write the order of the steps for Kruskal's algorithm.

Write the letter corresponding to each line in the order they are to be called within the method. Separate individual letters in the list using spaces. For example, your answer should be in the format of: “R S T U V W X Y”

NOTE: You will not need to use every step. No step is used more than once. There are multiple correct orderings.

A few points to note:
- Your ordering should be based on the homework implementation
- Each edge stores an edge weight and a pair of vertices (u, v).
- Assume that the edge set output is initialized for both Prim’s and Kruskal’s.
- Assume the disjoint set have already been initialized for Kruskal’s.
- For Kruskal’s, recall that two sets are disjoint if they have no element in common.
  This is equivalent to checking if the representative of the sets are different (using the find() method).

A. Create a while loop that terminates when the MST has 2* (|V| - 1) or the data structure that stores edges is empty
B. Create a queue and add all edges adjacent to the start vertex to the queue
C. For all edges adjacent to u, add to the data structure that stores edges only if the corresponding pair v has not already been visited
D. Return the edge set output if the size is 2 * (|V| - 1), else return null
E. Create a priority queue and add all edges in the graph to the priority queue
F. Union each vertex in the edge
G. Traverse back to the beginning of the while loop and recheck the terminating condition
H. Create a priority queue and add all edges adjacent to the start vertex to the priority queue
I. Return the edge set output
J. Add edge and reverse edge to MST edge set output if the vertex u and vertex v are in disjoint sets
K. Create a while loop that terminates when the data structure that stores edges is empty
L. Create a while loop that terminates when the MST has $2^*(|V| - 1)$
M. Remove the first element in the data structure that stores edges
N. Add edge and reverse edge to MST edge set output if the vertex v has not been visited before (you have already visited vertex u)
O. Create a queue and add all edges in the graph to the queue
P. Add edge and reverse edge to MST edge set output
Q. Add edge and reverse edge to MST edge set output if the vertex u and vertex v are not in disjoint sets
R. Initialize a visited set and add the start vertex
S. Add the edge vertex v to the visited set (you have already visited vertex u)

There are multiple valid answers, just be careful with the ordering.

Scenario Questions:

1. BFS
2. Dijkstra's
3. Prim's
4. n
5. m
6. q
7. j
8. h