Data Structures
CSCI H343, Fall 2022 [A]

Final

Name: _____

This exam has 12 questions, for a total of 100 points.

1. [10 points] You are given a hashtable with table size $m = 5$ that uses the multiply-add-and-divide (MAD) method for the hash function: $h(k) = ((ak + b) \mod p) \mod m$ (with $p = 13$, $a = 3$, $b = 8$) and uses separate chaining for collisions. Write down the equation for the MAD method hash function, specialized for this hashtable. Insert the keys 1, 2, 3, 4, 5, and 6 into the hashtable. Show the hash value for each key. Draw the resulting hashtable. How many collisions occurred?

Solution: The equation for the multiply-add-and-divide method for this hashtable is: **(1 point)**

$$h(k) = ((3k + 8) \mod 13) \mod 5$$

The results from hashing are:

- $h(1) = 1$ **(1/2 point)**
- $h(2) = 1$ **(1/2 point)**
- $h(3) = 4$ **(1/2 point)**
- $h(4) = 2$ **(1/2 point)**
- $h(5) = 0$ **(1/2 point)**
- $h(6) = 0$ **(1/2 point)**

The table is:

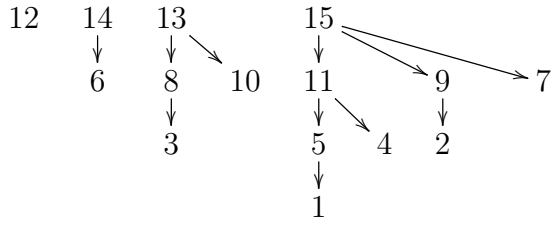
0	$\to 5 \to 6$
1	$\to 1 \to 2$
2	$\to 4$
3	$\to$
4	$\to 3$
5	$\to$

(4 points)

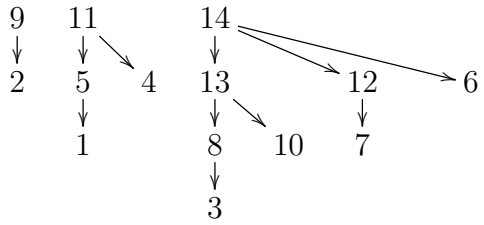
There were two collisions:

- 1 collided with 2,**(1 point)**
- 5 collided with 6. **(1 point)**

2. [8 points] Given the following Binomial Max-Heap, remove the maximum key and from the heap draw the updated heap.



Solution: Remove 15 and merge it's children into the heap. One of the correct solutions is:



3. [10 points] Fill in the blanks to complete the following implementation of Quicksort and its helper, the Partition algorithm. The parameters of the `quicksort` function an array of integers texttttA and a half-open range `[begin,end)`. The elements of `A` inside the specified range are to be rearranged into sorted order and the rest of `A` should not change.

```
static int partition(int[] A, int begin, int end) {
   int pivot = ___(a)___;
   int i = begin;
   for (int j = begin; j != end-1; ++j) {
      if (A[j] <= pivot) {
         ___(b)___
         ++i;
      }
   }
   swap(A, i, ___(c)___);
   return i;
}
static void quicksort(int[] A, int begin, int end) {
    if (begin != end) {
        int pivot = partition(A, begin, end);
        ___(d)___
        ___(e)___
    }
}
```

Solution: (2 points each)

```
(a) A[end-1]
(b) swap(A, i, j);
(c) end-1
(d) quicksort(A, begin, pivot);
(e) quicksort(A, pivot+1, end);
```

4. 6 points Your first task as a data analyst for the National Weather Service is to compute the median amount of precipitation in Seattle during each 1-hour interval of the previous century. You are given a large file with the amount of precipitation (in inches) that fell from the sky during each hour, so it contains $365 \times 24 \times 100 = 876,000$ entries. For example, on January 1, there was 0.14 inches of rain in the first hour of the new year, so 0.14 is the first entry in the file. (The number of digits of accuracy varies from hour to hour due to a glitch in the sensors, but you may assume that the entries are evenly distributed between 0 and 1 inches because it rains a lot in Seattle, but never pours.) You remember that the median is the value that would occur in the middle of a sequence once the sequence is sorted, so you decide to sort the precipitation data. Unfortunately, Congress has cut funding for the National Weather Service, so you only have a very old and slow computer on which to do the sorting.

   1. What sorting algorithm is the best for this situation and why?
   2. What is the time complexity of this sorting algorithm?

Solution:

1. Bucket Sort because we're sorting real numbers that are evenly distributed in a specified range, in this case between 0 and 1, and because Bucket Sort is fast. **(4 points)**
2. What is the time complexity of the sorting algorithm that you've chosen? Bucket Sort is linear time, in particular, $O(n)$ where n is the length of the array. **(2 points)**

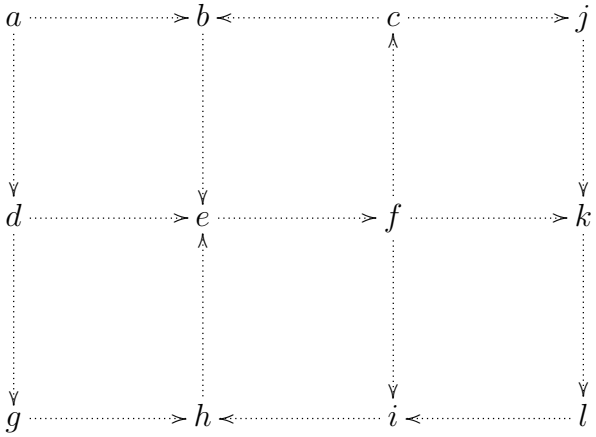
5. [10 points] Fill in the blanks to complete the following `AdjacencyMatrix` class for representing directed graphs.

```
public class AdjacencyMatrix implements Graph<Integer> {
    ArrayList<ArrayList<  (a)  >> is_adjacent;

    // Construct a graph with the specified number of vertices but no edges.
    // The vertices are numbered 0 through num_vertices - 1.
    public AdjacencyMatrix(int num_vertices) {
        is_adjacent = new ArrayList<>(num_vertices);
        for (int i = 0; i != num_vertices; ++i) {
            is_adjacent.add(new ArrayList<Boolean>(num_vertices));
            for (int j = 0; j != num_vertices; ++j)
                is_adjacent.get(i).add(  (b)  );
        }
    }

    public int numVertices() { return is_adjacent.size(); }

    // Add an edge between vertex u and v.
    public void addEdge(Integer u, Integer v) {
        (c)
    }

    // Return the vertices adjacent to u.
    public Iterable<Integer> adjacent(Integer u) {
        ArrayList<Integer> adj = new ArrayList<>();
        int v = 0;
        for (Boolean b :  (d)  ) {
            if (b)
                adj.add(v);
            (e)
        }
        return adj;
    }

    // Return all the vertices in the graph.
    public Iterable<Integer> vertices() {
        return IntStream.range(0, numVertices())::iterator;
    }

}
```

Solution: (2 points each)

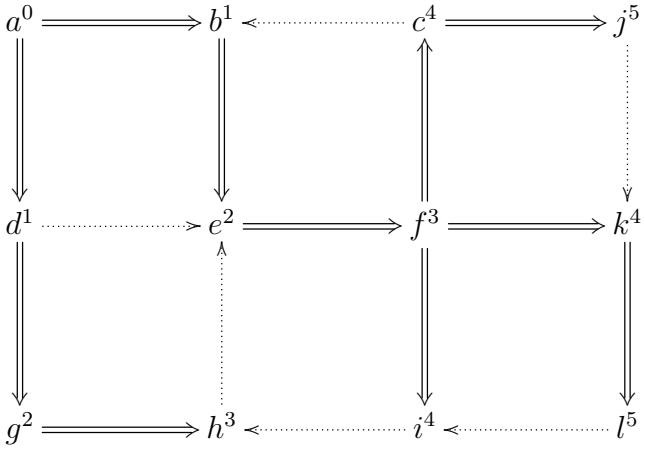
(a) Boolean
(b) false
(c) is_adjacent.get(u).set(v, true);
(d) is_adjacent.get(u)
(e) ++v;

6. 8 points Apply breadth-first search on the following graph starting at vertex a, marking the edges in the breadth-first tree. Give the length of the shortest path from a to all the vertices reachable from vertex a. When choosing the order in which to process and visit nodes, give priority to nodes that are earlier in the alphabet (`a` before `b`).



Solution:

(4 points for a correct breadth-first tree, 4 points for the correct distances)



7. [10 points] Complete the following code that computes the connected components of a graph by implementing the `DFS_visit` function.

```
public interface Graph<V> {
    int numVertices();
    void addEdge(V source, V target);
    Iterable<V> adjacent(V source);
    Iterable<V> vertices();
}

<V> void connected_components(Graph<V> G, Map<V, V> representative) {
    HashMap<V, Boolean> visited = new HashMap<V, Boolean>();
    DFS(G, representative, visited);
}

<V> void DFS(Graph<V> G, Map<V, V> reps, Map<V, Boolean> visited) {
    for (V u : G.vertices()) {
        visited.put(u, false);
        reps.put(u, u);
    }
    for (V u : G.vertices()) {
        if (! visited.get(u))
          DFS_visit(G, u, u, reps, visited);
    }
}

<V> void DFS_visit(Graph<V> G, V u, V representative, Map<V, V> reps,
                   Map<V, Boolean> visited) {
```

Solution:

```
<V> void DFS_visit(Graph<V> G, V u, V representative, Map<V, V> reps,
                   Map<V, Boolean> visited) {
    visited.put(u, true);                 // 1 point
    for (V v : G.adjacent(u)) {           // 2 points
        if (!visited.get(v)) {            // 1 point
            reps.put(v, representative); // 3 points
            DFS_visit(G, v, representative, reps, visited); // 3 points
        }
    }
}
```

8. [8 points] What is the tight big-O time complexity of the following algorithm? State your answer in terms of the number of vertices n and the number of edges m in the graph. Explain your answer. You may assume that all the method calls take $O(1)$ time.

```
static <V> void topo_sort(Graph<V> G,  Consumer<V> output, Map<V,Integer> num_in) {
    for (V u : G.vertices())
        num_in.put(u, 0);
    for (V u : G.vertices())
        for (V v : G.adjacent(u))
            num_in.put(v, num_in.get(v) + 1);
    LinkedList<V> zeroes = new LinkedList<V>();
    for (V v : G.vertices())
        if (num_in.get(v) == 0)
            zeroes.push(v);
    while (zeroes.size() != 0) {
        V u = zeroes.pop();
        output.accept(u);
        for (V v : G.adjacent(u)) {
            num_in.put(v, num_in.get(v) - 1);
            if (num_in.get(v) == 0)
                zeroes.push(v);
        }
    }
}
```

Solution: Each of the `for` loops is $O(n)$ **(2 points)**. The `while` loop processes every vertex in the graph just once because it only puts a vertex in the queue when its `num_in` is zero, and it decrements the `num_in` of a vertex before deciding whether to push it in the queue. **(1 point)** The `while` loop together with the inner `for` loop process each edge in the graph at most once because it processes each vertex once, and processes the out-edges of that vertex just once (with the inner `for` loop). **(1 point)** Thus, the `while` loop is $O(m+n)$ **(2 points)** and because $O(m+n)$ is greater than $O(n)$, the whole algorithm is $O(m+n)$. **(2 points)**

9. [10 points] Complete the following implementation of the union-find data structure (aka. disjoint sets). (You do **not** need to apply the path compression or union-by-rank optimizations.)

```
public class UnionFind<V> {
    Map<V, V> parent;

    public UnionFind(Map<N, N> p) {
        parent = p;
    }

    // Put x in a group by itself.
    void make_set(N x) {




    }

    // Merge the groups that x and y are in.
    public V union(V x, V y) {









    }

    // Return the representative of u.
    public V find(V u) {









    }
}
```

Solution:

```
public class UnionFind<V> {
    Map<V, V> parent;

    public UnionFind(Map<N, N> p) {
```

```
            parent = p;
        }

        void make_set(N x) {
            parent.put(x, x);
        }

        public V union(V x, V y) {
            V rx = find(x, parent); V ry = find(y, parent);
            parent.put(ry, rx);
            return rx;
        }
        public V find(V u) {
            while (u != parent.get(u)) {
                u = parent.get(u);
            }
            return u;
        }
    }
```

10. [8 points] The function `majorityElement`, shown below, returns an element of the array that occurs the largest number of times. (The array is required to be non-empty.) What is the time complexity of `majorityElement` in terms of the array size n? Explain your answer.

```
int countInRange(int[] nums, int num, int begin, int end) {
    int count = 0;
    for (int i = begin; i != end; ++i)
        if (nums[i] == num)
            ++count;
    return count;
}

int majorityElement(int[] nums, int begin, int end) {
    if (begin + 1 == end)
        return nums[begin];
    int mid = begin + (end - begin)/2;
    int left = majorityElement(nums, begin, mid);
    int right = majorityElement(nums, mid, end);
    if (left == right)
        return left;
    int leftCount = countInRange(nums, left, begin, end);
    int rightCount = countInRange(nums, right, begin, end);
    return leftCount > rightCount ? left : right;
}
```

Solution: The time complexity is $O(n \log n)$. Consider the recursive call tree. At each level of the tree, the algorithm does $O(n)$ work because of the calls to `countInRange` **(3 points)**. There are $\log n$ levels of the tree because the array size is cut in half with each recursive call to `majorityElement` **(3 points)**. Multiplying the time per level times the number of levels yields $O(n \log n)$. **(2 points)**

11. 9 points You're leading an expedition across a mountain range. The following grid of numbers is an abstraction of the map. Each number indicates how many hours it will take to hike through that region.

```
1 2 3 3 1      NW  N  NE            Example path:    3   Move:
3 3 1 4 1       W -|- E                           1       SW
1 1 2 3 3      SW  S  SE                           3     SE
4 4 3 5 3                                          5     S
```

Your expedition must start anywhere in the first row (the north side) and end anywhere in the last row (the south side). Each move in your path goes from the current row to the next row to the south. However, you must travel either 1) diagonally south-west, or 2) directly south, or 3) diagonally south-east. For example, one of the paths you could take is shown above and it would take 12 hours. The challenge is to find paths that take the least amount of hiking time.

Answer the following questions.

1. Is it better to apply a greedy algorithm or dynamic programming to this problem?
2. What path will give you the trip with the least hiking time across this mountain range? Show your work in applying a greedy algorithm or dynamic programming to solve this.
3. If you start in the north-west corner, what's the path with the least hiking time across the mountain range? Show your work.
4. If you start in the north-east corner, what's the path with the least hiking time across the mountain range? Show your work.

Solution:

1. There is no greedy choice that will give the optimal answer, so dynamic programming is better for this problem. **(2 points)**
2. The fastest path is the following, which takes 7 hours. **(3 points)**

```
2
  1
1
  3
```

3. 8 hours **(1 point)**
4. 8 hours **(1 point)**

Here's the dynamic programming table used to compute the answers. Each triple of numbers 9/9/7 are the trip times for the three options of going SW, S, or SE. **(2 points)**

```
-/8/8  9/9/7  10/8/12  8/12/10  10/8/-
-/8/7  8/7/8   5/6/7   9/10/10  7/7/-
-/5/5  5/5/4   6/5/7   6/8/6    8/6/-
  4      4       3       5        3
```

12. [3 points] What advice would you give a student taking Data Structures next year?

Solution: Open ended.