Name:____

CSE373 Winter 2014, Second Midterm Examination
February 26, 2014

Please do not turn the page until the bell rings.

Rules:

- The exam is closed-book, closed-note, closed calculator, closed electronics.
- **Please stop promptly at 3:20.**
- There are **106 points** total, distributed **unevenly** among **7** questions (many with multiple parts):

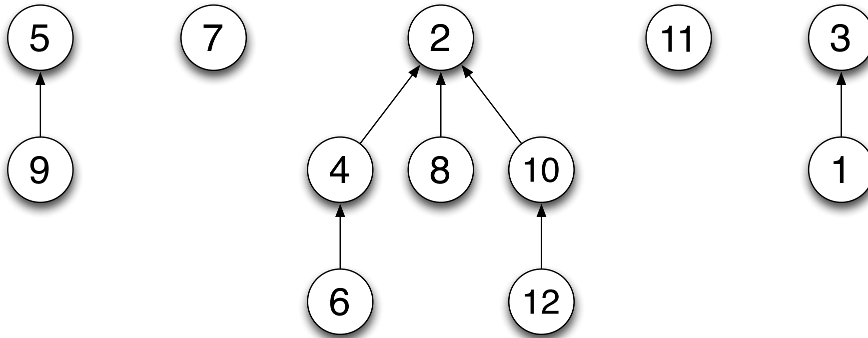
Question	Max	Earned
1	12	
2	13	
3	10	
4	27	
5	16	
6	14	
7	14	

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit. But clearly circle your final answer.**
- The questions are not necessarily in order of difficulty. **Skip around.** Make sure you get to all the problems.
- If you have questions, ask.
- Relax. You are here to learn.

Name:______________________________________

1. (**12** points) The following uptrees represent sets in an instance of the union-find ADT. They can be stored in a n-element array where each entry stores the parent of the node. If a node is a root (i.e. has no parent), the array stores the negative size of the uptree.



(a) Fill out the array below such that it corresponds with the uptrees pictured above.

1	2	3	4	5	6	7	8	9	10	11	12
3	-6	-2	2	-2	4	-1	2	5	2	-1	10

(b) Show the result of performing `union(11,3)` using union-by-size and `find(6)` using path compression by doing both of the following:

- Redraw below any uptrees (from above) that change as a result.
- Update the array representation (from part (a)) as appropriate by drawing a single slash ("/") through any numbers that change and writing the new number next to it.

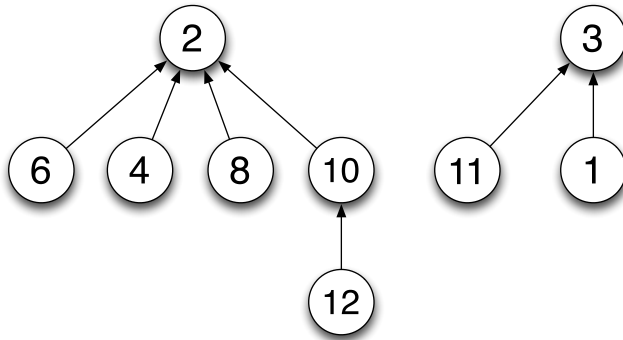
(c) For an instance of the union-find ADT initially containing n elements each in their own set, how many union operations must occur before all n elements are in the same set?

(d) If union-by-size and path compression are not used, what is the asymptotic worse-case running time for a `find` operation if there are n elements in the union-find?

Solution:

(b)

1	2	3	4	5	6	7	8	9	10	11	12
3	-6	~~-2~~ -3	2	-2	~~4~~ 2	-1	2	5	2	~~-1~~ 3	10



(c) $n-1$ union operations must occur

(d) $O(n)$

Name:______________________________________

2. (**13** points)

   (a) Fill in the contents of the hash table below after inserting the items shown. To insert the item k, use the hash function

   $$k \bmod \text{TableSize}$$

   and resolve collisions with **quadratic probing**.

   Insert: 13, 44, 103, 113, 2

0	1	2	3	4	5	6	7	8	9

   (b) We now consider looking up some items that are not in the table after doing the insertions above. For each, give the list of table locations that are looked at in order before determining that the item is not present. Include all the table locations examined, whether or not they contain an item. Note: these items are only being looked up, not inserted.

      i. 57
      ii. 42
      iii. 11

   (c) Give the load factor for the hash table.

Solution:

(a)

0	1	2	3	4	5	6	7	8	9
		113	13	44		2	103		

(b) i. 7, 8
    ii. 2, 3, 6, 1
    iii. 1

(c) $\lambda = 0.5$

Name:____

3. (**10** points) **Don't miss part (b).**

The Java code below provides an adjacency-matrix representation for a weighted directed graph where the n nodes correspond with the numbers 0, 1, ... $n-1$. The matrix locations store the edge weights. If the edge does not exist, a null value is stored instead. Remember that a 2D array in Java is simply an array of arrays. That is, every element of the outer array is itself just an array.

```
public class Graph {
    // Adjacency matrix representation with n nodes where
    // the 2D array is n by n.
    // Weight of the edge from node i to node j is in array index [i][j].
    // If the edge from node i to node j does not exist matrix[i][j] == null.
    private Double[][] matrix;
    public Graph() {
        // ... constructor not shown; assume it is correct
    }
    private void printCount(int node, int count) {
        System.out.println(node + ": " + count);
    }
    public void displayInCount(double threshold) {
        // YOUR CODE GOES HERE
    }
}
```

(a) You will implement the method `displayInCount(double threshold)` to print each node followed by the number of incoming edges to that node with a weigh greater than `threshold`. The line should have the node followed by a colon and the count, so we might see output like `4: 7`. No line should be printed twice. You should need somewhere around 9 lines of code. – not necessarily exactly 9, but to give you a sense if you are writing too much or far too little.

(b) Give tight asymptotic worst-case running-time bounds for your code in part (a) in terms of $|V|$, the number of nodes.

Solution:

(a)
```
public displayInCount(double threshold) {
  for (int i = 0; i < matrix.length; i++) {
    int count = 0;
    for (int j = 0; j < matrix.length; j++) {
      if (matrix[j][i] != null && matrix[j][i] > threshold) {
        count++;
      }
    }
    printCount(i, count);
  }
}
```

(b) $O(|V|^2)$.

Name:____

4. (**27** points) These three questions about graphs all have the same subparts. Note that for parts (iii), (iv), and (v), your answer should be in terms of an arbitrary k, <u>not</u> assuming $k = 4$.

   (a) Suppose a directed graph has k nodes, where there are two "special" nodes. One has an edge from itself to every non-special node and the other has an edge from every non-special node to itself. There are no other edges at all in the graph.

      i. Draw the graph (using circles and arrows) assuming $k = 4$.
      ii. Draw an adjacency list representation of the graph assuming $k = 4$.
      iii. In terms of k, exactly how many edges are in the graph?
      iv. Is this graph dense or sparse?
      v. In terms of k (if k is relevant), exactly how many correct results for topological sort that does this graph have?

   (b) Suppose a directed graph has k nodes, where each node corresponds to a number (1, 2, ..., k) and there is an edge from node i to node j if and only if $i \bmod 2 \neq j \bmod 2$

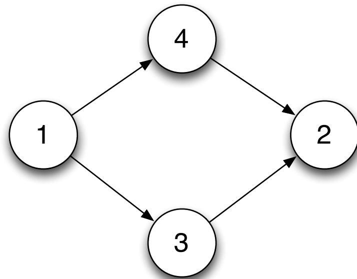
      i. Draw the graph (using circles and arrows) assuming $k = 4$.
      ii. Draw an adjacency list representation of the graph assuming $k = 4$.
      iii. In terms of k, exactly how many edges are in the graph assuming k is even?
      iv. Is this graph dense or sparse?
      v. In terms of k (if k is relevant), exactly how many correct results for topological sort that does this graph have?

   (c) Suppose a directed graph has k nodes, where each node corresponds to a number (1, 2, ..., k) and there is an edge from node i to node j if and only if $j = i + 1$.

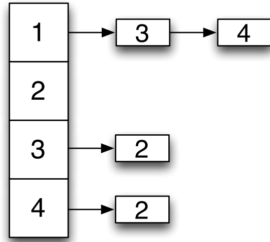
      i. Draw the graph (using circles and arrows) assuming $k = 4$.
      ii. Draw an adjacency list representation of the graph assuming $k = 4$.
      iii. In terms of k, exactly how many edges are in the graph?
      iv. Is this graph dense or sparse?
      v. In terms of k (if k is relevant), exactly how many correct results for topological sort that does this graph have?

Solution:

(a) i.



ii.

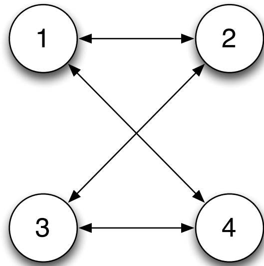


iii. $2(k-2)$

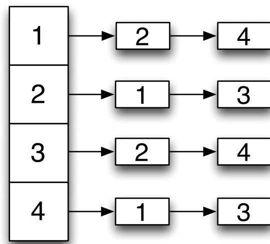
iv. sparse

v. $(k-2)!$, i.e., $1 \cdot 2 \cdot ... \cdot (k-2)$.

(b) i.



ii.

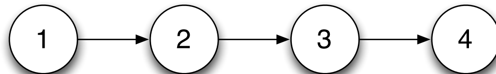


iii. $k^2/2$

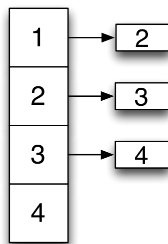
iv. dense

v. 0

(c) i.



ii.



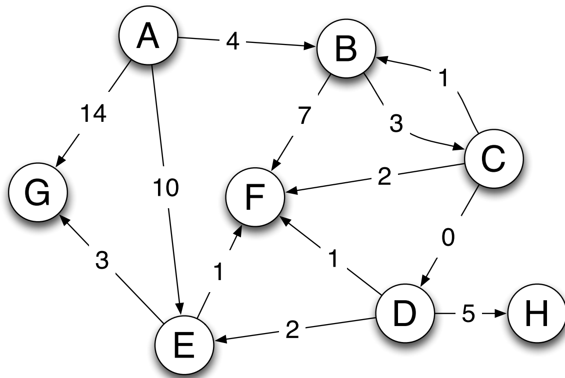
iii. $k-1$

iv. sparse

v. 1

Name:____

5. (**16** points) Consider the following directed, weighted graph:



(a) Step through Dijkstra's algorithm to calculate the single-source shortest paths from A to every other vertex. Show your steps in the table below. Cross out old values and write in new ones, from left to right within each cell, as the algorithm proceeds. Also list the vertices in the order which you marked them known. Finally, indicate the lowest-cast path from node A to node G.

Known vertices (in order marked known): A, B, C, D, F, E, G, H (or H, G)

Vertex	Known	Distance	Path
A	Y	0	—
B	Y	4	A
C	Y	7	B
D	Y	7	C
E	Y	~~10~~ 9	~~A~~ D
F	Y	~~11~~ ~~9~~ 8	~~B~~ ~~C~~ D
G	Y	~~14~~ 12	~~A~~ E
H	Y	12	D

Lowest-cost path from A to G: A to B to C to D to E to G

(b) Given the graph above, list one possible order that vertices in the graph above would be processed if a **breadth first traversal** is done starting at A.

Distance away	0	1	2	3	4
Vertex	A	B E G	F C	D	H

For a given distance, the vertices can be in any order.

Name:____

6. (**14** points)

   (a) Draw a weighted undirected graph with exactly 3 nodes that has exactly 0 minimum spanning trees.

   (b) Draw a weighted undirected graph with exactly 3 nodes that has exactly 1 minimum spanning tree.

   (c) Draw a weighted undirected graph with exactly 3 nodes that has exactly 2 minimum spanning trees.

   (d) Draw a weighted undirected graph with exactly 3 nodes that has exactly 3 minimum spanning trees.

   (e) Can a weighted undirected graph with 3 nodes have more than 3 minimum spanning trees? Why or why not?

   Solution:

   (a) Any unconnected, weighted, undirected graph

   (b) Solution needs to have either 2 or 3 non-self edges and if it has 3 non-self edges, then no two can have the same weight

   (c) Graph needs 2 non-self edges, with exactly 2 having the same weight and the third edge having a lower weight

   (d) Graph needs 3 non-self edges, all with the same weight

   (e) A 3-node graph can only have 3 spanning trees because it takes 2 edges (the number of nodes minus 1) to create a spanning tree. If the nodes are A, B, and C, the only possible spanning trees are:

   - (A,B), (A,C)
   - (B,C), (A,C)
   - (A,B), (B,C)

Name:____

7. (**14** points)

(a) Under what circumstances can you use perfect hashing?

(b) List what (if anything) is required in order for the following collision resolution methods to be guaranteed of finding a spot in the table to insert a new value (i.e. a value not previously in the table). Assume no rehashing (i.e. table resizing) occurs.

- Separate chaining

- Linear probing

- Quadratic probing

(c) Which collision resolution methods require lazy deletion?

(d) Double hashing is a collision resolution method that avoids the primary and secondary clustering caused by linear and quadratic probing. How does it accomplish this?

Name:____

(e) Briefly explain why we can claim an array-based queue implementation has amortized $O(1)$ running time for `enqueue` when any single `enqueue` operation can have $O(n)$ running time in the worst case.

(f) We have an array-based stack implementation that takes $O(n)$ time to perform a `push` operation when the array is not full. When the array is full, `push` takes $O(n^3)$ time (which includes doubling the size of the array). What is the amortized worst-case running time of `push` for our stack?

Solution:

(a) When all the keys are known ahead of time a "perfect" hash function can be constructed such that no collisions occur.

(b)
- Separate chaining requires nothing.
- For linear probing, the table cannot be full ($\lambda < 1$).
- For quadratic probing the table must be less than half full ($\lambda < 0.5$) and and the table size must be prime.

(c) All probing methods require lazy deletion (linear, quadratic, and double hashing).

(d) Double hashing avoids clustering by basing its probing interval on the key being hashed. It uses a second hash function to generate the interval. Since the interval is (probably) different for every key, clustering is less likely to occur.

(e) Since at least $n-1$ $O(1)$ enqueues must precede any $O(n)$ enqueue, the enqueue operation averages out to a $O(1)$ amortized running time. This is because in order for the array to need resizing, cheap enqueue operations must first fill it up.

(f) We will have $n - 1$ $O(n)$ push operations and 1 $O(n^3)$ push operation, which gives us a total of $(n - 1)n + n^3 = n^3 + n^2 - n$ for n operations. This averages to $n^2 + n - 1$ per operation, giving us an amortized running time of $O(n^2)$.