CS112 Final Review!!

Things to know from problem set: Isomorphic Trees, Graph Complement, HashPath

Interesting problem for today: Islands (count the number of Isolated nodes in a graph) / Topological sorting using bfs

QuickSort

Given the following Array of Integers

24 , 6, 72, 28, 12, 9, 3

Show the resulting QuickSort Tree
Using the first element as pivot
Draw the sort tree2
Derive the total running time (count item -> item comparisons)

HeapSort / HeapMerge

5. **Heap Merge (20 pts;5+5+3+7)**

Suppose that we have k heaps, with n elements in each. We wish to combine these into a single heap. Each of the following sub-parts describes one approach to merging the heaps, and you are asked to derive the big O running time. To get any credit, **you must show your work**.

a) We create a new, empty heap H. For each of the k heaps, we repeatedly remove the highest priority element and insert it into H, until the original heap is empty. What is the worst-case big O running time?

b) We create a new, empty linked list L. For each of the k heaps, we repeatedly remove the highest priority element and insert it at the front of L, until h_i is empty. We then transfer the elements of L into an array, and **heapify** the array (i.e. arrange in heap order) using the fastest known algorithm. What is the worst-case big O running time?

Solutions

a. First thing to keep in mind is the kind of operations we are doing and what we are given. So we are given K heaps and each of them have n elements. Now the proposed algorithm repeatedly removes the highest priority from each of the heaps (until they are empty) and inserts them into a master heap called H. Now a question you must ask yourselves in how long does it take to remove all elements from a heap of size n……..... The answer is nlogn. Now what if we were doing this for K heaps? In this case, the

answer will be k*nlogn since each of the K heaps will take nlogn time. Now the second part of the problem we should account for is how long it takes to insert all elements into the master heap. The total number of elements that will be contained in the master heap is k*n since we are inserting a total of k*n elements. This will give us a runtime of k*nlog(k*n). If you want to get a closer look at how we arrive at this, inserting the first, second, third.. . up until the k*nth element takes log(1) + log(2) + log(3) + ….. + log(kn) = knlog(kn). Now putting our two runtimes together, we get k*nlogn + knlog(kn) = O(knlog(kn)).

B. The first part of this problem is similar to part A since we remove the highest priority from all K heaps. This should give us the same runtime of k*nlog(n). Now the second part of this problem is that we insert each element that we removed from the heaps into the front of a linked list (Keep in mind that this is a constant time operation for a single insert). Since we will be inserting a total of k*n elements, the runtime for this part of the algorithm is kn. The third part of the algorithm is transfering the elements from the linked list to an array (The runtime for this part is also kn). The last time is running heapify on the array (which has kn elements). This part also takes kn. So in total we have k*nlog(n) + kn + kn + kn = O(knlog(n))

c) We create a new, empty linked list L. For each of the k heaps, we iterate over its array storage, remove each element, and add it to the front of L. We then transfer the elements of L into an array, and **heapify** the array using the fastest known algorithm. What is the worst-case big O running time?

d) We group the k heaps into $k/2$ pairs, and apply the algorithm from part (c) on each pair, leaving $k/2$ heaps. We then repeat this process until we are left with a single heap. What is the worst-case big O running time?

a. Knlgkn
b. Knlgn
c. Kn
d. Nlogk

C. This is a little different. Since we are removing the elements directly from the array storage of the heaps, we don't care about sifting down. To remove all elements from the array storage of k heaps (and insert them into a linked list) will take kn. To transfer the elements from the linked list

to an array will take kn. To heapify the array will take kn. So we have a total runtime of kn + kn + kn = O(kn)

d.

3. **Heapsort (20 pts;11+4+5)**

You are given the following `Heapsort` class outline - fill in the `siftDown`, `buildHeap`, and `sortHeap` methods.

```
public class Heapsort {

   // sorts an array in ascending order using heapsort
   public static <T extends Comparable<T>>
   void sort(T[] list) {
      buildHeap(list);
      sortHeap(list);
   }

   // sifts down in an array[0..n-1] starting at index k
   private static <T extends Comparable<T>>
   void siftDown(T[] list, int k, int n) {
       // FILL IN THIS METHOD




}
```

```
    // arranges items in list in heap order in LINEAR TIME, using repeated sift downs
    private static <T extends Comparable<T>>
    void buildHeap(T[] list) {
        // FILL IN THIS METHOD









    }

    // sorts a heap-ordered array
    private static <T extends Comparable<T>>
    void sortHeap(T[] list) {
        // FILL IN THIS METHOD









    }
}
```

2. **Shortest Path (20 pts)**

The following is an implementation of an *undirected* graph without any weights on the edges. Assume that the graph has already been loaded into the adjacency linked lists. Fill in the required method to find the shortest path from vertex numbered `x` to vertex numbered `y`. (Note: Since the graph is unweighted, the length of a path is simply the number of edges in it.) You may use, without implementation, any of the standard methods of the `Queue` and `Stack` classes. For anything else, you will need to implement your own code (including helper methods, if needed). You may add fields to the `Graph` class as necessary.

```
public class Neighbor {
    public int vnum;
    public Neighbor next;
    ...
}

public class Graph {
    Neighbor[] adjLists;   // adjacency linked lists for all vertices
    // add other fields as necessary




    // returns the length of the shortest path from x to y, (assume y different from x)
    // or -1 if y is not reachable from x
    public int shortestPath(int x, int y) {
      // FILL IN THIS METHOD
```

Public int shortestPath (int x, int y) {

boolean [] visited = new boolean[adjList.length];

Arrays.fill(visited,false);

Int [] dist = new int [adjList.length];

Arras.fill(dist, Interger.MAX_VALUE);

return bfs(x,y,visited,dist);

}

Int bsf (int x, int y, boolean [] visited, int [] dist) {

Explanation: Since we do not have weights on this graph, we can simulate weights by saying that the weight between two adjacent vertices (neighbors) is 1.

```
        Queue<Integer> nodes = new Queue<>();
        nodes.enqueue(x) // enqueue start node
        Dist[x] = 0;  // distance of start node is 0
        Visited[x] = true;  // mark start node as visited

        While ( ! nodes.isEmpty()) {

                Int v = nodes.dequeue(); // a. dequeue and visit its neighbors. If the neighbor
hasn't been visited, then set its distance to (the distance of the current vertex + 1)
                // If the neighbor has been visited, then check whether the distance of the current
vertex + 1 is less than the distance of the neighbor. If it is, then update it to the distance of the
current vertex + 1


                For ( Neighbor nbr = adjLis[v]; nbr != null; nbr = nbr.next) {

                        Int vertexNum = nbr.vnum;

                        If ( !visited[vertexNum] ) {
                                dist[vertexNum] = dist[v] + 1;
                                visited[vertexNum] = true;
                                nodes.enqueue(vertexNum);
                        }

                        else {
                                If ( dist[vertexNum] > dist[v] + 1) {
                                        dist[vertexNum] = dist[v] + 1;
                                }
                        }
                }

                return dist[y];
        } // end of bfs


}
```

Shortest Path, Connected Components

1. **Strongly Connected Directed Graph (25 pts)**

A directed graph is *strongly connected* if every vertex is reachable from every other vertex in the graph. (Note that reachablility is via a path that could have one or more edges.) You are given an adjacency linked lists representation of a directed graph. Complete the implementation below to tell if a directed graph is strongly connected. Do not worry about the efficiency of your implementation. You may add other helper methods (with full implementation) as needed.

```
public class DirectedGraph {
     class Edge {
          int vnum;    // vertex number of neighbor
          Edge next;   // pointer to next neighbor
          Edge(int v, Edge ptr) {vnum=v; next=ptr;}
     }
     int n;               // number of vertices
     Edge[] adjLists;     // adjacency linked lists
     // define additional fields as necessary



     // returns true if this graph is strongly connected, false otherwise
     public boolean isStronglyConnected() {
          // COMPLETE THIS METHOD
```

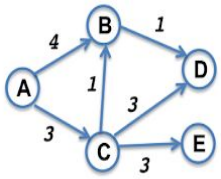
```
Public void topSortBfs(){
    Int [] indegrees = new int[adjLists.length];

   for(int = 0; i < adjLists.length; i++) {
            Edge temp = adjLists[i];
            while(temp!=null){
                    Indegrees[temp.vnum]++;
                    temp = temp.next;
            }
      }

      Queue<Integer> top = new LinkedList<>();
```

```
	for(int i = 0; i < indegrees.length; i++){

		if(indegrees[i] == 0){
			top.add(i);
		}
	}



while(!top.isEmpty()){

	Int vertex = top.poll();
	System.out.print(vertex);

	Edge temp = adjLists[vertex];

	while(temp != null){
		//Indegrees[temp.vnum]--;
		if(--Indegrees[temp.vnum] == 0){
			top.add(temp.vnum);
		}
		temp = temp.next;
	}
}


}
```

```
public boolean isStronglyConnected() {

        boolean [] visited = new boolean [adJLists.length];

        Arrays.fill(visited, false);

        for (int i = 0; i < visited.length; i++) {

                dfs(i, visited);
                // now check for false cells


                for (int j = 0; j < visited.length; j++){
                        If ( !visited[j] ) {
                                return false;
                        }
                }

                // re-initialize visited to false for next bfs call (next iteration of outer loop)
                Arrays.fill(visited,false);
        }


    return true;   // this means that all nodes are reachable from every other node

}


Void dfs ( int start, boolean [] visited) {

// I believe you can implement this yourselves

}
```

Dikjstra's algorithm:

2. **Dijkstra's Shortest Paths Algorithm (25 pts, 18+7)**



a) Dijkstra's shortest paths algorithm is executed on the graph above, starting at A. Assume that vertices A,B,C,D,E are given numbers 0,1,2,3,4 respectively in the implementation. Trace the execution of the algorithm as follows: at the end of every step, show the distance array and the fringe (see table below). The fringe is stored in a min-heap, in which distance updates can be made, apart from delete min, and insert. To show the fringe, draw the binary tree heap structure, with (vertex name,distance) information at each node. Every time there is a change to the heap, show the number of item-to-item comparisons needed to make that change, and also what operation resulted in that change. (Ignore the time needed to *locate* an item in the heap for a distance update.)

Step	Distance array	Fringe heap	Comparisons (number, for what operation)