EECS 214 Practice Quiz 2 Solutions

Question 1

Consider the following sort algorithm:

Sort(array)

T = make an empty red/black tree

for each element e in array

add e to T

for i=0 to array.length-1

e = T.FindMinimum

T.Remove(e)

array[i] = e

What is the time complexity (i.e. O of what?) of this sort?

**Answer:** Creating the empty tree takes constant time. Adding each element takes log n time, and runs n times. So adding the elements takes O(n log n) time. Extracting the minimum element and removing it each take O(log n) time, and also run n times, so the total time is O(n log n).

Question 2

Suppose you are given a hash table using chaining (i.e. linked lists) and it uses the following awful hash function:

h(k) = 0

What is the complexity (i.e. O of what?) of the following operation:

for i=1 to n

hashtable.Store(i, 0)

Assume that the hash table has to check whether the key i is already present when doing the store operation.

**Answer:** The problem is that since the hash function maps all keys to the same hash bucket, the hash table is really effectively just a linkedlistdictionary. So insertions require linear time (to check whether the key is already present). Since we run the insertion n time, the total time for the loop is quadratic, i.e. .

Question 3

Here's a random algorithm that operates on an array. Yes, I know it doesn't do anything useful, but tell me what its time complexity is anyway (i.e. is it linear, quadratic, log n, or what?).

BlaBlaBla(array, start, end)

if (start!=end)

for each i between start and end

sum += array[i]

BlaBlaBla(array, start, (start+end)/2)

BlaBlaBla(array, (start+end)/2, end)

**Answer:** The recursion runs level deep. At each level, the different calls to BlaBlaBla run their for each loops for different numbers of iterations, but the total number of iterations is always n, so the total number of iterations per level of recursion is n, so the total time is .

Question 4

Write a recursive algorithm to count the number of nodes in a connected graph. You may assume any graph representation you find convenient, however, if you use the array of adjacency lists representation, you can't cheat and just check the length of the array :-)

**Answer:** We assume a the array-of-adjacency-lists representation used in the books, so the input is an array of adjacency lists:

CountNodes(adj)

visited = new bool[adj.Length]

return CountNodesRecur(adj, 0)

CountNodesRecur(adj, nodeNumber)

visited[nodeNumber] = true

count = 1 // remember to include this node in the count

foreach neighbor in adj[nodeNumber]

if not visited[neighbor]

count += CountNodesRecur(adj, neighbor)

return count

This solution uses a depth-first walk of the graph, but a breadth-first walk would also work.

Question 5

Suppose you're given an undirected graph represented using the array-of-adjacency-lists representation used in the book. Give an algorithm for finding the distance from a given node to every other node in the graph. If a node is not reachable from the start node, it's distance should be marked as infinite. Be sure it will not loop infinitely if the graph has cycles.

**Answer:**

ComputeDistances(adj, start)

Q = new empty Queue

visited = new bool[adj.Length]

distance = new int[adj.Length]

Q.Enqueue(start)

visited[start] = true

distance[start] = 0

while Q not empty

node = Q.Dequeue()

for each neighbor in adj[node]

if not visited[neighbor]

visited[neighbor] = true

distance[neighbor] = distance[node] + 1

Q.Enqueue(neighbor)

Question 6

Suppose you're given a **weighted**, undirected graph represented using an adjacency matrix representation (i.e. a matrix of weights where the weight is ∞ if the nodes are unconnected). Give an algorithm for finding the distance **from every node to every other node** in the graph. If two nodes are unconnected, their distance should be marked as infinite. Your algorithm should run in O(V3) time.

**Answer: This is just the Floyd-Warshall algorithm. Assume w is the matrix of weights.**

**let dist be a |V| × |V| array of minimum distances initialized to ∞**

**for each vertex v**

**dist[v][v] ← 0**

**for each edge u,v**

**dist[u][v] ← w[u,v]**

**for k from 1 to |V|**

**for i from 1 to |V|**

**for j from 1 to |V|**

**if dist[i][j] > dist[i][k] + dist[k][j]**

**dist[i][j] ← dist[i][k] + dist[k][j]**

Question 7

What is the worst-case asymptotic time complexity of Dijkstra’s algorithm, assuming it is implemented using a binary heap?

**Answer: O((V+E)log V).**

Question 8

Explain the differences between a binary search tree and a binary heap.

**Answer: They are similar in that they’re both binary trees, however:**

* **A binary heap is a complete binary tree, a BST need not even be balanced, although it will hurt performance if it is not**
* **A binary heap satisfies the heap property (parents are larger than their children (max heap) or smaller than them (min heap)), whereas a BST satisfies the binary search tree property (all left descendants are <= the parent, all right descendants are >= it)**
* **A binary heap is typically embedded inside an array, a BST is not.**
* **A binary heap is used to implement a priority queue, a BST is used to implement a dictionary**