Homework 2

Tags	Homework
Date	

PROBLEM 1

ALGORITHM

We have k > 2 jars, and want to find the number of times we have to drop the jars to find the highest safe rung.

- set i = 1
- drop jar from i * n^((k-1)/k)
- if jar breaks
  - jar is between (i-1) * n^((k-1)/k) and i * n^((k-1)/k)
  - repeat with k-1 jars
- if jar doesn't break
  - i += 1
  - repeat with k jars

PROOF

- We can observe that for the first jar, we can drop it at most 2n / n^((k-1)/k) = 2n^(1/k) times
- Therefore, fk(n) ≤ 2kn^(1/k)
- This means the highest rung that is n^((k-1)/k) for the base condition k=1
- Assume that what above is true, it must also be true for k-1 jars in the interval (i-1) * n^((k-1)/k) and i * n^((k-1)/k)
  - Therefore, it should take less than 2(k-1)(n^((k-1)/k))^(1/(k-1)) = 2(k-1)n^(1/k) which is the same statement we found above with k-1

- Hence, by induction the algorithm requires less than 2kn^(1/k) drops which meets the problem specification

RUNTIME + PROOF

- This algorithm has a runtime of O(2kn^(1/k)) because it takes at worst that many drops or iterations of the recursive algorithm for us to determine the highest safe rung. The proof for this is the same as above.

PROBLEM 2

SETUP

Given an arbitrary binary tree T with n vertices, let us define the total number of nodes in T with two children as n_2 and total number of leaves in T as n_0.

We want to show that $n_2 = n_0 - 1$.

Base case:

Given a binary tree T such that there exists a single node with 0 children. In such binary tree, there exists no nodes with two children and exactly one leaf, as the node has no children.

Therefore, we have $n_2 = 0, n_1 = 1$. This affirms the statement that $n_2 = n_0 - 1$. Hence, the base condition is true.

Induction:

Assume, for induction, that for an arbitrary binary tree T with n vertices the statement above holds true. We want to prove that for an arbitrary binary tree T with n+1 vertices, the statement retains true.

Consider an arbitrary placement of a leaf l onto a node p. There are two cases that are achieved when this node is added:

- Case 1: Node p is a leaf
    - Node p becomes a parent node ⇒ number of leaves in tree reduces by 1 ⇒ n_0 -= 1

- Node p has a child with leaf l ⇒ number of leaves in tree increases by 1 ⇒ n_0 += 1
- Number of nodes with two children remains unchanged
- $n_2 = n_0 - 1 + 1 - 1 \Rightarrow n_2 = n_0 - 1$ remains

- Case 2: Node p is a parent node with 1 child
  - Node p becomes a parent node with 2 children ⇒ n_2 += 1
  - Number of leaves in tree increases by 1 ⇒ n_0 += 1
  - $n_2 + 1 = n_0 - 1 + 1 \Rightarrow n_2 = n_0 - 1$ remains

By Induction, when a node is is added to tree T with each node with at most two children, the statement retains true.

PROBLEM 3

Given a graph on n nodes, with n as an even number, let's assume that there exists a node with a degree < n/2 that could lead to a connected graph.

Consider 2 nodes n_1 and n_2, we know that for these two nodes to be connected, they must have an edge in common. In this case for n = 2, both of these nodes would need to have a degree of 1, that is $n/2 = 2/2 = 1$, which contradicts our assumption of a degree < n/2 above.

Henceforth, by way of contradiction, the claim is true.

PROBLEM 4

This turns out to be a problem that can be solved efficiently. Suppose we are given an undirected graph *G* = (*V* , *E*), and we identify two nodes *v* and *w* in *G*. Give an algorithm that computes the number of shortest *v*-*w* paths in *G*. (The algorithm should not list all the paths; just the number suffices.) The running time of your algorithm should be *O*(*m* + *n*) for a graph with *n* nodes and *m* edges.

ALGORITHM

- create set discovered with node s set as true and all other nodes set as false

- Initialize $L[0]$ to consist of the single element s
- Set the layer counter $i = 0$
- Set the current BFS tree T = null
- initialize path_lengths[v] to 0, and all other nodes to an arbitrarily large number
- initialize path_num_shortest[v] to 1
- While $L[i]$ is not empty
    - Initialize an empty list $L[i + 1]$
    - For each node a $\in L[i]$
        - Consider each edge (a,b) incident to a
        - If Discovered[b] = false then
            - Set Discovered[b] = true
            - Add edge (a, b) to the tree T
            - Add b to the list $L[i+1]$
        - if path_lengths[b] = path_lengths[a] + 1
            - path_num_shortest[b] += path_num_shortest[a]
        - else path_lengths[b] > path_lengths[a] + 1
            - path_num_shortest[b] = path_num_shortest[a]
            - path_lengths[b] = path_lengths[a] + 1
    - Increment the layer counter i by one
- return path_num_shortest[w]

PROOF

By work of contradiction, assume that there is a case while traversing the BFS tree where for two nodes a,b the length of node where path_lengths[b] < path_lengths[a] + 1.

- that means that there is a path that that goes from the node v to the b node that is larger than the a node, which would imply that b exists earlier in the tree than a which cannot be true for a BFS tree traversal

- By contradiction, that means there can only be two cases
    - path_lengths[b] < path_lengths[a] + 1
        - there exists a shorter path from v to b than v to a, update variables
    - path_lengths[b] = path_lengths[a] + 1
        - shortest path from v to a is added to path_num_shortest
- storing these paths and its lengths, we are able to get the number of shortest paths to find the node w and return it from the array

RUNTIME + PROOF

- A BFS algorithm has runtime of O(n+m) for n nodes and m edges.
- This is the case because each node is only iterated over once, meaning each node and each edge is iterated once in total
- The added components to this BFS algorithm don't add anything to the time complexity as there is only copying done within the if statements, which are all done in O(1) time.
- Therefore we have **O(n+m)**

PROBLEM 5

ALGORITHM

- Enter whole set into blackbox and check if sum = k, return YES if it is
- If the sum of the total set ≠ k
    - for each element in the subset
        - remove that element + get sum of sequence without that element
        - if sum = k. return YES
        - if sum ≠ k, then remove the last element of the set and re-calculate the sum of the set
            - if the sum = k, return YES

        - if sum ≠ k keep removing the last element of the set until sum = k or no elements remaining
- if sum still not = k, return NO

PROOF

- By way of contradiction, assume that there is a sum in a subset of the whole set that is equal to k but is lost by removing an arbitrary number.
- this would be a contradiction, because there would be another set where that subset exists by removing another arbitrary number
- this is due to the fact that the algorithm checks each possible subset of elements in the set, that is,
    - n elements ⇒ n^n combinations
    - for each element, check every subset ⇒ for each element, there are n elements then (n-1) elements then (n-2) elements until there is 0 elements ⇒ n^n combinations as well
- Hence if we check all combinations of the set, we are either sure to find a sum = k if it exists in the set

RUNTIME + PROOF

- worst case is no sum is found at end of algorithm
- only one search through the list with for loop = O(n)
- total removal of elements in list = O(n)
- assume sum of sequence found by blackbox is O(n)
- total is O(n) + O(n) + O(n) ~ **O(n)**

PROBLEM 7

ALGORITHM

Let array of of n elements that contain all but one of the integers from 1 to n+1 be defined as ‘arr’

[1,2,3,4,5,7]

- if arr[n/2] = n/2 + 1
  - missing element on elements with indices > n/2
  - if elements exist that are > n/2
    - recurse through algorithm for set of integers from [arr[0],arr[n/2]] until 1 number left, return index of that number in arr
- else arr[n/2] ≠ n/2 + 1
  - missing element on elements with indices ≤ n/2
  - recurse through algorithm for set of integers from [arr[n/2+1],arr[n]] until 1 number, return index of that number

PROOF OF ALGORITHM

- since list starts from 1, we know that index of element + 1 = value in list. If n/2+1 = n, then we know that the list is correct up until the n/2 index, so the missing element is on the right side. if n/2 + 1 ≠ n then the missing element is on the left side of the half.
- By way of contradiction, assume that splitting list by half doesn't have a side with missing element
  - means that right side has elements from n/2 to n with values n/2 + 1, n/2 + 2, ... , n+1 without a missing number
  - means that left side has elements from 0 to n/2 with values 1, 2, ..., n/2 without a missing number
  - this cannot exist by contradiction because missing element must exist in the set
- therefore, as we keep splitting the any n sized array by half, we eventually reach 2 elements that gives us the the missing number + 1, which can be found by taking the found number-1.

RUNTIME + PROOF

This algorithm uses binary search to find the missing number.

- The worst case would be if during the first pass, n/2 + 1 was the missing number
- In that case, we would achieve a runtime of **O(log n)**

- This is true because for each stage of the pass, we split the input array in half from n -> n/2 -> n/4 -> ... -> n/2^k. $n/2^k <= 1$ shows that k = log(n)

PROBLEM 8

ALGORITHM

- sum_arr = calculate total sum of array
- sum_all = calculate total sum of all numbers between 1 and n+1
- return sum_all - sum_arr

PROOF OF ALGORITHM

- we know that the array of elements are not sorted.
- let A = 1 + 2 + ... + n+1, B = 1 + 2 + ... + n+1 but B has a missing number
- A - B = (1-1) + (2-2) + ... + (n+1) - (n+1) but there will be one number in A that doesn't have a match, leaving A - B = missing number

RUNTIME + PROOF

- incremental search to calculate the sum of the array, and a sum of all numbers from 1 to n+1.
- search will increment through n elements in the worst case (every case because we are finding sum of the entire array of n elements once), that is O(n
- sum will take O(1) if a formula is derived or O(n) if calculated using a loop
- That gives us a worst case runtime of **O(n)** regardless