Amy Seo

Dis 1D
10 – 11:50AM
TA: Song

CS 180 Homework 2

1. **Algorithm:** Given *k* jars and *n* floors
   - If $k = 0$, return
   - Divide the number of floors into partitions of size $n^{(k-1)/k}$
   - Drop the first jar at $n^{(k-1)/k}$ and continue dropping it at multiples of $n^{(k-1)/k}$ until it breaks. Let us denote the multiples as an integer *m* starting from 1.
     - If the jar breaks at $mn^{(k-1)/k}$, the highest safe rung must be between $(m-1)n^{(k-1)/k}$ and $mn^{(k-1)/k}$.
     - Repeat the algorithm with *K* decremented by 1 (since the egg broke and we have one less egg now) in the interval $(m-1)n^{(k-1)/k}$ to $mn^{(k-1)/k}$.
   - Return m

   **Proof by Contradiction:** Suppose the algorithm returns the wrong highest safe rung *h*, when the correct highest safe rung is actually *b*.
   - Case 1: *h* is higher than *b*
     - *b* must be in a lower interval and/or floor by definition, and thus our algorithm will explore that interval first and find *b* before even reaching floor *h.*
     - This is a contradiction as our algorithm explored floor *h.*
   - Case 2: *h* is lower than *b*
     - This algorithm only breaks into smaller partitions immediately when a jar breaks, which means the jar had to break at some interval including *h*.
     - Since our algorithm visits floors from the bottom of the interval to the top, the first floor it will come across and break at is right above *h*.
     - This is a contradiction as the first floor it should break at is right above *b*.
   - Thus, by contradiction, the algorithm must return the highest safe rung.

   **Time Complexity & Justification:** The time complexity is $O(kn^{1/k})$
   - Each time we are dropping a jar, the floors are divided into intervals of size $n^{(k-1)/k}$
   - The worst-case number of drops per jar is is $n/n^{(k-1)/k} = n^{1/k}$ times
   - Since we have k jars, the worst case number of total drops is $kn^{1/k}$
   - As the number of jars increases, the time complexity grows asymptotically slower due to the 1/k exponent, which means the constraint $\lim_{n\to\infty} f_k(n)/f_{k-1}(n) = 0$ is satisfied.

2. **Proof by Induction:**
   - *Base Case:*
     When there is only one node with no children, there are 0 nodes with two children and 1leaf. Thus, the number of nodes with two children is exactly one less than the number of leaves.
   - *Inductive Step:*
     Let a binary tree with N nodes have one less number of nodes with two children than the number of leaves.

     Then, there are two cases for a binary tree with N+1 nodes:
     - The last node is inserted as a child of a node with no other children

        - The new binary tree has the same number of nodes with two children as the binary tree with N number of nodes. In the same way, there are the same number of leaves. Thus, by the inductive hypothesis, there are still one less number of nodes with two children than the number of leaves.
    - The last node is inserted as a child of a node with one other child
        - The new binary tree has one more node with two children, but also one more leaf. Since both the number of nodes with two children and the number of leaves increased by 1, by the inductive hypothesis, there is still one less number of nodes with two children than the number of leaves.
- Therefore, for any binary tree, the number of nodes with two children is exactly one less than the number of leaves.

**3. Proof by Contradiction:**

- Let there be a graph G with N nodes, where N is an even number, such that every node of G has a degree of at least N/2 and G is not connected.
- Since G is not connected, there must be at least 2 nodes V and W, such that V and W are not connected.
- Since every node has a degree of at least N/2, V and W must each have N/2 different nodes that are incident on each of them.
- That would mean there are N/2 + N/2 + 1 + 1 nodes, which is a contradiction as there are only N nodes.
- Therefore, by contradiction, if G is a graph with N nodes where N is an even number, and if every node of G has a degree of at least N/2, G must be connected.

4. **Algorithm:**

- Suppose we are given an undirected graph *G(V, E)* and we identify two nodes *V* and *W* in *G* such that we want to find the **number** of **shortest** paths between them
- Initialize an empty queue *Q* that holds tuples (*K, I*) where *I* represents the number of edges incident on some node *K*
- Push (*V,* 1) onto *Q*
- While *Q* is not empty
    - Let L be the length of Q at this point (this is the size of our "level", all nodes in this part of the queue are the same distance from V)
    - Iterate through *Q* for *L* number of times
        - Pop off the first node, let's denote this node as *C*
        - if *C* has not been visited yet:
            - mark *C* as visited
            - for all nodes adjacent to C that have not been visited yet:
                - if the adjacent node is in *Q*, increment its *I* value by the I value of C
                - if an adjacent node is not in *Q*, add it to *Q* with an *I* value of *C*'s *I* value
    - If *W* is now in *Q* (the shortest path has been found)
        - return the *I* value of *W*
- Return 0 because there is no valid path from *V* to *W*

**Proof of Algorithm by Contradiction:** Let there be two nodes V and W such that we found the number of shortest paths between them, but our number of shortest paths is incorrect. There are 2 cases:

- Case 1: The number of shortest paths returned accounted for a path that is longer than the shortest path.
    - Let there be some path *P2* that is longer than the shortest path *P1* that has been accounted for in *W*'s *I* value.
    - Then *P2* must have at least one more node in its path compared to *P1*.
    - Since *P1*'s path is shorter, it'll reach *W* first by the nature of our BFS algorithm.
    - When *W* is now in *Q* for the first time, the algorithm will return it's *I* value.
    - Since *P2* has not reached *W* yet, it cannot be accounted for. This is a contradiction as P2 has been accounted for. Thus, the algorithm only accounts for the number of **shortest** paths.
- Case 2: The number of shortest paths missed a shortest path.
    - Let *P2* be a shortest path that was not accounted for.
    - Let *P1* be a shortest path that is accounted for.
    - Since both paths are the shortest path, they must be the same distance from *V*. Thus, they will reach *W* at the same time (during the same iteration of the while loop). So when the *I* value of *W* is being returned, *P2* has been accounted for**.**
    - This is a contradiction as *P2* is not accounted for. Thus, the algorithm accounts for **every shortest path.**

**Time Complexity & Justification:** Since we are using BFS, we will visit all N nodes and traverse all M edges in the worst-case scenario. Therefore, the time complexity is O(M+N)

5. **Algorithm:**
    - Suppose S is the sequence of any real numbers and we have an integer K.
    - Let B() denote the blackbox function that gives us True(YES) or False(NO) depending on whether there is a subset in the input whose sum equals K
    - If the blackbox returns False for the inputted sequence S, end the algorithm and indicate that there is no valid subset whose sum is K.
    - For each number N in S from start to end of array:
        - remove N from S
        - if B(S) returns True, continue
        - else, add N back to S
    - Return S

**Proof by Contradiction:**

- Assuming that there is a valid subset, suppose our algorithm returns the sequence S whose sum does not equal K.
- Case 1: The sum is less than K.
    - By the nature of our algorithm, at the end of every iteration, the blackbox must respond with YES since we either remove or add back a number to satisfy the blackbox.
    - This means that S must receive a YES from the blackbox.
    - But the blackbox responds with NO when the sum of the entire subset is less than K since there is no possible subset that'll sum up to K if the entire sum is less than it.

    - This is a contradiction as the returned sequence S is guaranteed a YES from the blackbox.
    - Therefore, the sum of S can never be less than K.
- Case 2: The sum is greater than K.
    - There must be a number that can be removed from S as the blackbox must return YES for S as reasoned in Case 1.
    - By the nature of our algorithm, every number will be removed once. If the resulting sequence still gets a YES from the blackbox, the number will never be added back to the sequence.
    - It follows then that every number in S contributes to the sum of K. This is a contradiction as there is a number that can be removed from S.
    - Therefore, the sum of S can never be greater than K.
- By contradiction, our algorithm is guaranteed to return a subset whose sum is K, given that there is a valid subset.

**Time Complexity & Justification:** The time complexity is O(N) because we have to use the blackbox N times as we traverse through the whole array.

**6. An array of n elements contains all but one of the integers from 1 to n+1.**

**a. Algorithm:** Assuming the array is not empty, this is a binary search method.

- Let L be the first index and R be the last index of the input array A
- While L is less than R:
    - Let M be the midpoint between L and R
        - M = L + (R-L)/2
    - if A[M] is greater than M+1, search below M by setting R = M
    - else, search above by setting L = M + 1
- at this point, L = R
    - It follows that the midpoint M = L = R
    - if A(M) > M+1:
        - return M+1
    - else:
        - return M+2

**Proof by Contradiction:** Suppose we return the wrong missing number K when the correct answer was J.

- Case 1: K is greater than J.
    - Since the input array is sorted, J must come before K.
    - Whenever A[M] is greater than M+1, we know that the missing number must be below or at index M. Therefore, our algorithm searches index M and below.
    - This is a contradiction because we would have to search above M at some point to get to K.
    - Therefore, K can never be greater than J.
- Case 2: K is less than J.
    - Since the input array is sorted, J must come after K.
    - Whenever A[M] is less than M+1, we know the missing number must be above the current index M. Therefore, our algorithm searches above M.
    - This is a contradiction because we would have to search below M at some point to get to K.
    - Therefore, K can never be less than J
- Thus, by contradiction, our algorithm always returns the missing number.

**Time Complexity & Justification:** The algorithm runs in O(log N) time as this is a binary search, and the worst-case run time is the height of the tree.

**b. Algorithm:**

- Let N be the number of elements in the input array
- Let S be the summation of all elements from 1 to N+1 using the summation formula: $S = (N+1)(N+2)/2$
- Add up all the values in the array, let this value be denoted by V
- The missing number is S – V.

**Proof by Contradiction:**

- Suppose that:
  - S is the summation of all elements from 1 to N+1
  - V is the sum of all the elements in the input array
  - M is the missing number
  - D is the incorrect value the algorithm returned such that D does not equal M.
- Since V is the sum of all the elements in the array, V + M = S as M is the correct missing value.
- It follows that S – V = M. From our algorithm, S – V = D. This is a contradiction as M and D are not equal.
- Thus, by contradiction, D is always the missing value.

**Time Complexity & Justification:** This algorithm runs in O(N) time, where N is the number of values in the array because we need to traverse through each element in the array.