2.8b **Algorithm**: From the previous homework, we found that the optimal starting position for 2 eggs, n floors is $x = \lceil \frac{-1+\sqrt{1+8n}}{2} \rceil$ where the n is the number of floors. We will start from there:

1. Set step size $x = \lceil \frac{-1+\sqrt{1+8n}}{2} \rceil$.
2. While jar doesn't break, set new step size $x = x - 1$, step up and drop the jar.
3. Jar is now broken; set new floor count $n = x$ and return to last step.
4. Repeat steps 1-3 until there is 1 jar remaining, in which case set step size $x = 1$ and step up until the last jar breaks.

Proof of correctness: Since f_k is defined recursively as $f_k(n) = \frac{-1+\sqrt{1+8f_{k-1}(n)}}{2}$, by substitution we have $\lim_{n\to\infty} \frac{f_k(n)}{f_{k-1}(n)} = \lim_{n\to\infty} \frac{-1+\sqrt{1+8f_{k-1}(n)}}{2f_{k-1}(n)} = 0$. Therefore each function does indeed grow asymptotically slower than the previous one.

Complexity: $O(k \log n)$, since we need to traverse through k jars and the trials of each jar is done in $O(\log n)$.

3.5 **Answer**: Proof by induction;

Base case: Take a binary tree with exactly one node, then that node is a leave and there is no node with two children. Therefore the number of nodes with two children is exactly one less than the number of leaves for a binary tree with a single node.

Inductive step: Suppose that any binary tree with n leaves has $n-1$ nodes with two children, we want to show that any binary tree with $n+1$ leaves has exactly n nodes with two children. We can think of this as adding a node to the binary tree with n leaves. There are two possible scenarios from here:

- Add the new node to a leaf node: the new node becomes a leaf node, while its parent is no longer a leaf node. So the number of leaves remain at n and the number of nodes with two children remain at $n - 1$. So this scenario actually does not create a new leaf node, but we can observe that the rule still applies.
- Add the new node to a node with 1 child: the new node becomes a leaf node, while its parent becomes a node with two children. Then the number of leaves become $n + 1$ and the number of nodes with two children becomes n.

Therefore the number of nodes with two children is exactly one less than the number of leaves by mathematical induction.

3.7 **Answer**: True. Proof by contradiction:

Suppose that there exists a disconnected graph G, with n nodes, n even, where every node of G has degree at least $\frac{n}{2}$. Since G is disconnected, there must be at least two components. However, since every node has degree at least $\frac{n}{2}$, each component must have at least $\frac{n}{2} + 1$ nodes. But this implies G has at least $2(\frac{n}{2} + 1) = n + 2$ nodes, which contradicts with our assumption that G has n nodes, i.e. such disconnected G does not exist. Therefore by contradiction, if every node of G has degree at least $\frac{n}{2}$, then G is connected.

3.10 **Algorithm**:

1. Start at vertex w, traverse through its adjacent vertices. Store these as a list l.
2. If w is in l, return 1.
3. Traverse through the unvisited adjacent vertices of vertices in l, store them as the new l.
4. If w is in l, return the number of times it appears.
5. Repeat steps 3-4 until w is found in l.

Proof of correctness: We will first show that the algorithm does indeed find the shortest path by contradiction. Suppose that the algorithm finds paths of length k from v to w, but there exists at least one path of length j such that $j < k$. Since each iteration of the algorithm represents moving one length away, paths of length k are found in iteration k and paths of length j are found in iteration j. However, $j < k$ implies that we have already checked all paths of length j in a previous iteration, which means the algorithm would have returned paths of length j instead. Therefore such j does not exist and the algorithm does find the shortest paths.

Now we will show that the algorithm returns the correct number of shortest paths. Since by construction l_k contains all vertices length k away from u, any path from u to w would be contained in l_k. Therefore it is not possible for the algorithm to miss paths of length k.

Complexity: $O(m+n)$ as the worst case scenario is traversing through every edge and vertex, i.e. $m+n$.

P1 Suppose that you are given an algorithm as a blackbox. You cannot see how it is designed. The blackbox has the following properties: if you input any sequence of real numbers, and an interger k, the algorithm will answer YES or NO indicating whether there is a subset of the numbers whose sum is exactly k. Show how to use this blackbox to find the subset whose sum is k, if it exists. You should use the blackbox O(n) times, where n is the size of the input sequence.

Algorithm: Let $B(\cdot)$ denote the boolean blackbox algorithm and arr denote the unsorted sequence of real numbers.

1. If $B(arr, k)$ returns NO, exit program as such subset does not exist.
2. Set $subset = [\,]$.
3. Traverse through the sequence and check $B(arr, k - arr[i])$, where $arr[i]$ is the current element:
   - If $B(arr, k - arr[i])$ returns YES:
     - Append $arr[i]$ to $subset$.
     - Set $k = k - arr[i]$.
     - if $k == 0$, exit loop.
   - Else continue.
4. Return $subset$.

Proof of correctness: Let S be the desired subset. We will show that the algorithm will always return S by induction.

Base case: When the desired set has only one element, i.e. $|S| = 1$; let s_1 be the only element of S (note that $k = s_1$), then we have the following two possible scenarios:

- s_1 is not present in the sequence: this is handled by step 1; $B(arr, k)$ would return NO and exit the program as it is not possible to construct a subset with sum k from the sequence.
- s_1 is in the sequence: since step 3 traverses through every element in the sequence, it will eventually traverse to s_1. Since $k = s_1$, $B(arr, k - s_1)$ will always evaluate to true as it is always possible to construct a subset of sum 0 by selecting the empty set.

Inductive step: Suppose that given $k_n = s_1 + \ldots + s_n$, the algorithm successfully returns $S_n = \{s_1, \ldots, s_n\}$. We want to show that given $k_{n+1} = k_n + s_{n+1}$, the algorithm will return $S_{n+1} = S_n \cup \{s_{n+1}\}$. We can show that such s_{n+1} always exist and will be selected by the algorithm by examining the following scenarios:

- Such s_{n+1} does not exist: in this scenario, $B(arr, k_{n+1})$ would return NO and the program would exit after step 1.
- Such s_{n+1} exists: suppose s_{n+1} appears in the sequence after each $s_i \in S_n$ has already been traversed, which we can guarantee upon renumbering. Then $B(arr, k_{n+1} - s_{n+1})$ is equivalent to $B(arr, k_n)$ which is assumed true by inductive hypothesis. Then the algorithm will construct and return S_{n+1} by appending s_{n+1} to S, as desired.

Therefore the algorithm will always return a complete and correct result by mathematical induction.
Complexity: $O(n)$, assuming $B(\cdot)$ is $O(1)$, since the algorithm traverses through the sequence only once.

P2 An array of n elements contains all but one of the integers from 1 to $n + 1$.

(a) Give the best algorithm you can for determining which number is missing if the array is sorted, and analyze its asymptotic worst-case running time.

Algorithm:

1. Set lower search (inclusive) bound $lower = 0$ and upper search (exclusive) bound $upper = n$.
2. While $upper - lower > 1$, visit the element at index $i = \lfloor \frac{lower+upper}{2} \rfloor$ (assuming 0-based indexing) and check its value $arr[i]$:
   - If $arr[i] = i + 1$, the missing element is in the second half of the current search interval. Set $lower = i$.
   - If $arr[i] = i + 2$, the missing element is in the first half of the current search interval. Set $upper = i$.
3. The search area is now exactly one number (with *lower* as its index), meaning that we have found the neighbor of the missing number. Check which side the missing number is on:
   - If $arr[lower] = lower + 1$, return $arr[lower] + 1$ as the missing number.
   - If $arr[lower] = lower + 2$, return $arr[lower] - 1$ as the missing number.

Proof of correctness: By induction on the size of the array.
Base case: $n = 1$, then our array contains 1 element ranging from 1 to 2, i.e. either $arr = [1]$ or $arr = [2]$. We can examine the two possible scenarios separately:

- $arr = [1]$: we have $lower = 0$ and $upper = 1$, skipping step 2 as $upper - lower = 1 \not> 1$, then since $arr[lower] = arr[0] = 1 = lower + 1$, return 2 as the missing number, which is correct.
- $arr = [2]$: we have $lower = 0$ and $upper = 1$, skipping step 2 as $upper - lower = 1 \not> 1$, then since $arr[lower] = arr[0] = 2 = lower + 2$, return 1 as the missing number, which is correct.

Inductive step: Assume that the algorithm successfully finds the missing number for an array of size up to n. We want to show that it will also work for an array of size $n+1$. There are two possible scenarios here:

- $n+1$ is even: the array is halved into two search areas of length $\frac{n+1}{2}$. Since $\frac{n+1}{2} \leq n$, we know that the algorithm works by inductive hypothesis.
- $n+1$ is odd: the array is halved into two search areas of sizes $\frac{n}{2}$ and $\frac{n}{2}+1$. Since $\frac{n}{2} < n$ and $\frac{n}{2}+1 < n$, we know that the algorithm works by inductive hypothesis.

Therefore the algorithm will always return the missing number by induction.

Complexity: $O(\log n)$; the algorithm halves the search area until the search area is size 1, taking $\log_2 n$ iterations.

(b) Give the best algorithm you can for determining which number is missing if the array is not sorted, ana analyze its asymptotic worst-case running time.

Algorithm:

1. Set $sum = 0$.
2. Traverse through the array and add each element to the sum, i.e. $sum = sum + arr[i]$.
3. Return $\frac{1}{2}(n+1)(n+2) - sum$ as the missing number.

Proof of correctness: The accumulated sum includes every integer from 1 to $n+1$, excluding the missing number. Since we know that the sum of the first $n+1$ elements is $\frac{1}{2}(n+1)(n+2)$, we can simply subtract f.

Complexity: $O(n)$, since it requires traversing through an array of n elements.