HW 1 Algorithms

ADM 1-6:

Given a set of S subsets $S_1 \ldots S_m$ of the universal set of U = {1,…n}, find the smallest subset of subsets $T \subseteq S$ such that $\cup_{t_i \in T} t_i = U$. For example, consider the subsets S_1 = {1,3,5}, S_2 = {2,4}, S_3 = {1,4}, S_4 = {2,5}. The set cover of {1,…5} would then be S_1 and S_2. Find a counterexample for the following algorithm: Select the largest subset for the cover, and then delete all its elements from the universal set. Repeat by adding the subset containing the largest number of uncovered elements until all are covered.

Counterexample:

S = { {1,2,3}, {4,5,6}, {2,3,4,5} }
U = {1,2,3,4,5,6}

Algorithm:

S_3 = {2,3,4,5}, U = {1,6}

S_1 = {1,2,3}, U = {6}

S_2 = {4,5,6}, U= {}

The set cover of U would be S_3, S_1, S_2 (3 subsets). The most optimal number of subsets would be 2: S_1, S_2

ADM 1-19:

Prove by induction that a tree with n vertices has exactly n – 1 edges.

Proof:

Call the scenario with a tree with n vertices having exactly n-1 edges P(n). Assume P(n) is true. We can prove that for P(n+1) vertices in a tree there are n edges: (n+1) – 1 = n edges. Therefore, a tree with n vertices has exactly n – 1 edges.

ADM 2-12

Prove that $n^3 - 3n^2 - n + 1 = \Theta(n^3)$

Proof:

To prove that $n^3 - 3n^2 - n + 1 = \Theta(n^3)$, show that $n^3 - 3n^2 - n + 1 = O(n^3)$ AND $n^3 - 3n^2 - n + 1 = \Omega(n^3)$:

1. To prove that $n^3 - 3n^2 - n + 1 = O(n^3)$, find constant c and n_0 such that $n^3 - 3n^2 - n + 1 <= c \cdot n^3$ Let c = 3 and n_o = 1. Then for all $n >= n_o$, $n^3 - 3n^2 - n + 1 <= c \cdot n^3$. Since the dominant term is n^3 and $c \cdot n^3$ will always be at least 3 times the size of the left side of our function.

2. To prove that $n^3 – 3n^2 – n + 1 = \Omega(n^3)$, find constant c and n_0 such that $n^3 – 3n^2 – n + 1 >= c \cdot n^3$

   Let c = .5 and $n_o = 1$. Then for all $n >= n_o$, $n^3 – 3n^2 – n + 1 >= c \cdot n^3$. Since $c \cdot n^3$ is only half the size of the left side of our function.

   Therefore, by 1 and 2, $n^3 – 3n^2 – n + 1 = \Theta(n^3)$.

ADM 2-52:

You have 8 balls that are all the same size. Seven of them weigh the same and one of them weighs slightly more. How can you find a ball that's heavier by using a balance and only two weighing's?

Algorithm:

1. Divide the eight balls into two groups of four each: call them group 1 and group 2
2. Take 3 balls from group 1 and 3 balls from group 2 and weigh them.
3. If the 3 balls in group 1 = the 3 balls in group 2 then weigh the two remaining balls: The heavier of the 2 remaining balls is the heaviest of all eight. Else if the 3 balls from group 1 > the 3 balls from group 2 OR the 3 balls from group 1 < the three balls from group 2: then take two balls from the heavier subset and weigh them: If the balls are the same weight: then the remaining unweight ball is the heaviest of the 8. Else: The heavier of the two balls is the heaviest of the 8.

Apple SDE1:

You have 100 coins lying flat on the table, each with a head side and a tail. 10 of them are heads up and 90 of them are tails up. You can't see, feel, or in any other way find out which side is up. Split the coins into two piles such that there are the same number of heads in each pile.

Algorithm:

Split the 100 coins into two piles randomly: Pile A (90 coins) and pile B (10 coins). We know that there are 10-n heads in pile B where n represents the number of heads in pile A. We also know that there are 90 – n tails in pile A where n represents the number of tails in pile B in this case. We can then directly calculate the number of tails in pile B as 10 – (10 – n) which equals n tails. Therefore, there are the same number of tails in pile B as there are heads in pile A. Therefore, to get the same number of heads in both piles, you need to flip all of the coins over in pile B.

Time Complexity:

The time complexity of this algorithm would be O(1) or constant time since you are randomly separating the coins into two piles and flipping over all of the coins in one of them. Even in the worse case scenario, pile B just simply need to be flipped once.

Amazon SDE1

Given a string s, find the first non-repeating character in it and return its index. If it does not exist, return -1.

Algorithm:

First create a dictionary to record the frequency of each character in the string using conditional statements:

For char in s:

    If character is in dictionary:

        Add 1 to the value of that character in the dictionary.

    Else set the value of the character in the dictionary to 1

Next, iterate through the dictionary and return the index of the character who's value is 1:

    For i in range of the dictionary length:

        If s[i] in dictionary and the frequency of the character is one:

            Return that character's index

Else if no character in the dictionary has a value of 1, return -1.

Time complexity:

The time complexity of this algorithm would be O(n) since you initially need to iterate through the whole string and record the frequency of each character.

Microsoft Reverse String: (See mock interview)