Day 48 / 100 :

Topic - Array, Binary search

1 Problem statement: Coin Change II (medium)

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.
Return the number of combinations that make up that amount. If that amount of money cannot be made up by any combination of the coins, return 0.
You may assume that you have an infinite number of each kind of coin.
The answer is guaranteed to fit into a signed 32-bit integer.

Example 1:
Input: amount = 5, coins = [1,2,5]
Output: 4
Explanation: there are four ways to make up the amount:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1

Example 2:
Input: amount = 3, coins = [2]
Output: 0
Explanation: the amount of 3 cannot be made up just with coins of 2.

Solutions :

Approach 1 - DP

Intuition:
First two code is just take not take and add both will get total ways.third way is efficiently counting the number of combinations to achieve a specific amount using various coin denominations.

Approach:

- The **dp** array stores the number of combinations for each amount using different coin denominations.

- The solve function is a recursive function that calculates the number of combinations for a given amount and a specific coin i.
- Base cases:
- If amount is 0, return 1 as there's one way to make 0.
- If I exceed the number of coin denominations, return 0.
- If the result for this sub problem is already calculated, return it from **dp** array.
- Calculate two possibilities:
- Include the coin at index i and reduce the amount by coins[i].
- Exclude the coin at index i and move to the next coin.
- Return the sum of the two possibilities and store it in **dp** before returning.
- The change function initializes **dp** and calls solve to compute the answer.

```
class Solution {
public:
int dp[5001][301];
int solve(int amount,vector<int>&coins,int i){
    if(amount==0){
        return 1;
    }
    if(dp[amount][i]!=-1)
    return dp[amount][i];

    if(i==coins.size())
    return 0;

    int tk=0;
    if(coins[i]<=amount){
        tk=solve(amount-coins[i],coins,i);
    }
    int nt=solve(amount,coins,i+1);
    return dp[amount][i]=tk+nt;
}
    int change(int amount, vector<int>& coins) {
        memset(dp,-1,sizeof(dp));
         return solve(amount,coins,0);
    }
};
```

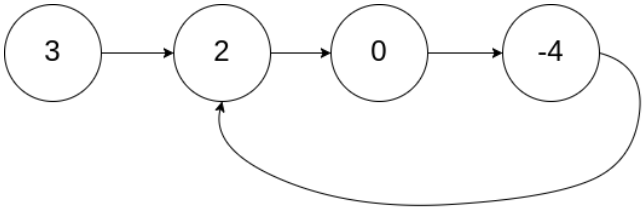
2 Problem statement: Linked List Cycle II (medium)

Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

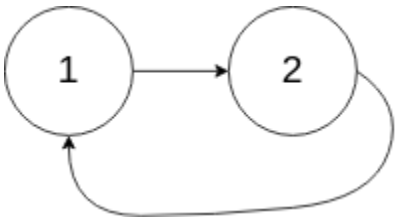
Do not modify the linked list.

Example 1:



Input: head = [3,2,0,-4], pos = 1
Output: tail connects to node index 1
Explanation: There is a cycle in the linked list, where tail connects to the second node.

Example 2:



Input: head = [1,2], pos = 0
Output: tail connects to node index 0
Explanation: There is a cycle in the linked list, where the tail connects to the first node.

Solutions :

Approach 1 - Two pointer + hash Table

Algorithm:

Step 1: determine whether there is a cycle

1.1) Using a slow pointer that move forward 1 step each time

1.2) Using a fast pointer that move forward 2 steps each time
1.3) If the slow pointer and fast pointer both point to the same location after several moving steps, there is a cycle;
1.4) Otherwise, if (fast->next == NULL || fast->next->next == NULL), there has no cycle.

Step 2: If there is a cycle, return the entry location of the cycle

2.1) L1 is defined as the distance between the head point and entry point
2.2) L2 is defined as the distance between the entry point and the meeting point
2.3) C is defined as the length of the cycle
2.4) n is defined as the travel times of the fast pointer around the cycle When the first encounter of the slow pointer and the fast pointer
According to the definition of L1, L2 and C, we can obtain:

- the total distance of the slow pointer traveled when encounter is L1 + L2
- the total distance of the fast pointer traveled when encounter is L1 + L2 + n * C
- Because the total distance the fast pointer traveled is twice as the slow pointer, Thus:
- 2 * (L1+L2) = L1 + L2 + n * C => L1 + L2 = n * C => L1 = (n - 1) C + (C - L2)*

It can be concluded that the distance between the head location and entry location is equal to the distance between the meeting location and the entry location along the direction of forward movement.
So, when the slow pointer and the fast pointer encounter in the cycle, we can define a pointer "entry" that point to the head, this "entry" pointer moves one step each time so as the slow pointer. When this "entry" pointer and the slow pointer both point to the same location, this location is the node where the cycle begins.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
ListNode *detectCycle(ListNode *head) {
    if (head == NULL || head->next == NULL)
        return NULL;

    ListNode *slow  = head;
```

```
    ListNode *fast  = head;
    ListNode *entry = head;

    while (fast->next && fast->next->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {                     // there is a cycle
            while(slow != entry) {              // found the entry
location
                slow  = slow->next;
                entry = entry->next;
            }
            return entry;
        }
    }
    return NULL;                                // there has no cycle
}
};
```