Day 34 / 100 :

Topic - Strings

1 Problem statement: Predict the Winner (Medium)

You are given an integer array nums. Two players are playing a game with this array: player 1 and player 2.
Player 1 and player 2 take turns, with player 1 starting first. Both players start the game with a score of 0. At each turn, the player takes one of the numbers from either end of the array (i.e., nums[0] or nums[nums.length - 1]) which reduces the size of the array by 1. The player adds the chosen number to their score. The game ends when there are no more elements in the array.
Return true if Player 1 can win the game. If the scores of both players are equal, then player 1 is still the winner, and you should also return true. You may assume that both players are playing optimally.

Example 1:
Input: nums = [1,5,2]
Output: false
Explanation: Initially, player 1 can choose between 1 and 2.
If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If player 2 chooses 5, then player 1 will be left with 1 (or 2).
So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.
Hence, player 1 will never be the winner and you need to return false.

Example 2:
Input: nums = [1,5,233,7]
Output: true
Explanation: Player 1 first chooses 1. Then player 2 has to choose between 5 and 7. No matter which number player 2 choose, player 1 can choose 233.
Finally, player 1 has more score (234) than player 2 (12), so you need to return True representing player1 can win.

Solutions :

Approach 1 - Using DP

Recursive Approach

```
class Solution {
public:
    int dfs(int i , int j ,vector<int>& nums , int turn){
        if(i == nums.size() || j == -1) return 0;
```

```
        if(i > j) return 0;
        if(turn == 0){
            return max(nums[i] + dfs(i + 1 , j , nums , 1),
                       nums[j] + dfs(i , j - 1 , nums , 1));
        }
        else{
            return min(-nums[i] + dfs(i + 1 , j , nums , 0),
                       -nums[j] + dfs(i , j - 1 , nums , 0));
        }
    }
    bool PredictTheWinner(vector<int>& nums) {
        int n = nums.size();
        int val = dfs(0 , n - 1 , nums , 0);
        return val >= 0;
    }
};
```

Memorization Approach

```
class Solution {
public:
    vector<vector<vector<long>>> dp;
    int dfs(int i , int j ,vector<int>& nums , int turn){
        if(i == nums.size() || j == -1) return 0;
        if(i > j) return 0;
        if(dp[i][j][turn] != -1e10) return dp[i][j][turn];
        if(turn == 0){
            return dp[i][j][turn] = max(nums[i] + dfs(i + 1 , j , nums , 1),
                                        nums[j] + dfs(i , j - 1 , nums , 1));
        }
        else{
            return dp[i][j][turn] = min(-nums[i] + dfs(i + 1 , j , nums , 0),
                                        -nums[j] + dfs(i , j - 1 , nums , 0));
        }
    }
    bool PredictTheWinner(vector<int>& nums) {
        int n = nums.size();
        dp.resize(n , vector<vector<long>>(n , vector<long>(2 , -1e10)));
        int val = dfs(0 , n - 1 , nums , 0);
        return val >= 0;
    }
};
```

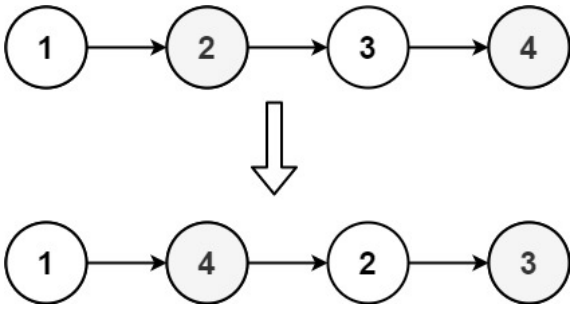
2 Problem statement: Record List (Medium)

You are given the head of a singly linked-list. The list can be represented as:
$L_0 \rightarrow L_1 \rightarrow \ldots \rightarrow L_{n-1} \rightarrow L_n$

Reorder the list to be on the following form:
$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \ldots$

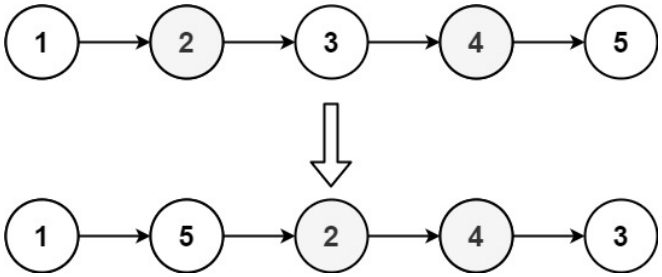
You may not modify the values in the list's nodes. Only nodes themselves may be changed.

Example 1:



Input: head = [1,2,3,4]
Output: [1,4,2,3]

Example 2:



Input: head = [1,2,3,4,5]
Output: [1,5,2,4,3]

Solutions :

Approach 1 - Two pointers

Core ideas:

- we have to run through the list to find the penultimate, not the last node;
- once we have it, we can also access the last and we set the the second node to follow the last, last to follow the head and NULL the penultimate;
- after that, we call reorderList on the third node;
- if the list has less than 3 elements, no point in proceeding, so we return (our base case);

Time performance is rather low (I go more or less quadratic here), but unless you change the values and/or use a bunch of extra memory to store the most of the list in some collection, I don't see many other ways around than iterating through it a bunch of times; I might try the other approach later, but that is clearly less fun.
The recursive code:

```
class Solution {
public:
    void reorderList(ListNode* head) {
        // base case handled here
        if (!head || !head->next || !head->next->next) return;
        // we need to find the penultimate node in order to proceed
        ListNode* penultimate = head;
        while (penultimate->next->next) penultimate = penultimate->next;
        // then we move it in the second spot
        penultimate->next->next = head->next;
        head->next = penultimate->next;
        // and set penultimate to be the last
        penultimate->next = NULL;
        // and then we proceed with the rest, same way
        reorderList(head->next->next);
    }
};
```

The iterative code:

```
class Solution {
public:
    void reorderList(ListNode* head) {
        // we need to find the penultimate node in order to proceed
        ListNode* penultimate;
        while (head && head->next && head->next->next) {
            penultimate = head;
            while (penultimate->next->next) penultimate = penultimate->next;
            // then we move it in the second spot
            penultimate->next->next = head->next;
            head->next = penultimate->next;
            // and set penultimate to be the last
            penultimate->next = NULL;
            head = head->next->next;
        }
    }
};
```