Predict the Winner

Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by the player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player. This continues until all the scores have been chosen. The player with the maximum score wins.

Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.

Example 1:

```
Input: [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 c
hooses 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: [1, 5, 233, 7]
Output: True
Explanation: Player 1 first chooses 1. Then player 2 have 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 Tr
ue representing player1 can win.
```

Note:

1. 1
2. Any scores in the given array are non-negative integers and will not exceed 10,000,000.
3. If the scores of both players are equal, then player 1 is still the winner.

Solution 1

```
public class Solution {
    public boolean PredictTheWinner(int[] nums) {
        return helper(nums, 0, nums.length-1)>=0;
    }
    private int helper(int[] nums, int s, int e){
        return s==e ? nums[e] : Math.max(nums[e] - helper(nums, s, e-1), nums[s]
- helper(nums, s+1, e));
    }
}
```

Inspired by @sameer13, add a cache:

```
public class Solution {
    public boolean PredictTheWinner(int[] nums) {
        return helper(nums, 0, nums.length-1, new Integer[nums.length][nums.lengt
h])>=0;
    }
    private int helper(int[] nums, int s, int e, Integer[][] mem){
        if(mem[s][e]==null)
            mem[s][e] = s==e ? nums[e] : Math.max(nums[e]-helper(nums,s,e-1,mem),
nums[s]-helper(nums,s+1,e,mem));
        return mem[s][e];
    }
}
```

Explanation
So assuming the sum of the array it SUM, so eventually player1 and player2 will split the SUM between themselves. For player1 to win, he/she has to get more than what player2 gets. If we think from the prospective of one player, then what he/she gains each time is a **plus**, while, what the other player gains each time is a**minus**. Eventually if player1 can have a >0 total, player1 can win.

Helper function simulate this process. In each round:
if e==s, there is no choice but have to select nums[s]
otherwise, this current player has 2 options:
--> nums[s]-helper(nums,s+1,e): this player select the front item, leaving the other player a choice from s+1 to e
--> nums[e]-helper(nums,s,e-1): this player select the tail item, leaving the other player a choice from s to e-1
Then take the max of these two options as this player's selection, return it.

written by chidong original link here

Solution 2

The main idea is each player will play optimally,
so if there exist that my opponent will lose the game (false), I will return true, that's why I use '!' in the return;

For example : [1, 5, 2]

```
if (player1 pick 1) the rest is [5, 2];
 if (player2 pick 2) player1 win(true),
 if (player2 pick 5) player1 lose(false),
-> because player2 play optimally, so he choose to pick 5,
   player2 = !(pick 2) || !(pick 5)  = !true || !false = true;
   so when player1 first choose 1, he always loses.

if (player1 pick 2) the rest is [1, 5];
 if (player2 pick 1) player1 win(true),
 if (player2 pick 5) player1 lose(false),
-> because player2 play optimally, so he choose to pick 5,
   player2 = !(pick 1) || !(pick 5)  = !true || !false = true;
   so when player1 first choose 2, he always loses.

So, it this case, no matter player1 first choose 1 or 2, he always loses.
   player1 = !(pick 1) || !(pick 2)  = !true || !true = false;
```

Here is my code:

```
public boolean PredictTheWinner(int[] nums) {
 return first(0, 0, nums, 0, nums.length-1);
}

private boolean first(int s1, int s2, int[] nums, int start, int end) {
 if (start >= end ){
  if (s1 >= s2) return true;
  else return false;
 }
 return !second(s1+nums[start], s2, nums, start+1, end) || !second(s1+nums[end],
s2, nums, start, end-1);
}

private boolean second(int s1, int s2, int[] nums, int start, int end) {
 if (start >= end ){
  if (s1 < s2) return true;
  else return false;
 }
 return !first(s1, s2+nums[start], nums, start+1, end) || !first(s1, s2+nums[end]
, nums, start, end-1);
}
```

written by shawloatrchen original link here

Solution 3

The idea is that this is a minimax game, and if you went to MIT and took 6.046 then you would have seen something similar to this problem in class. And thanks to MIT OCW everyone can see the explanation

The DP solution

```
class Solution {
public:
    bool PredictTheWinner(vector<int>& nums) {
        if(nums.size()% 2 == 0) return true;

        int n = nums.size();
        vector<vector<int>> dp(n, vector<int>(n, -1));

        int myBest = utill(nums, dp, 0, n-1);
        return 2*myBest >= accumulate(nums.begin(), nums.end(), 0);
    }

    int utill(vector<int>& v, vector<vector<int>> &dp, int i, int j){
        if(i > j) return 0;
        if(dp[i][j] != -1) return dp[i][j];

        int a = v[i] + min(utill(v,dp, i+1, j-1), utill(v, dp, i+2, j));
        int b = v[j] + min(utill(v,dp,i, j-2), utill(v,dp, i+1, j-1));
        dp[i][j] = max(a, b);

        return dp[i][j];
    }
};
```

written by kevin36 original link here