Can I Win

In the "100 game," two players take turns adding, to a running total, any integer from 1..10. The player who first causes the running total to reach or exceed 100 wins.

What if we change the game so that players cannot re-use integers?

For example, two players might take turns drawing from a common pool of numbers of 1..15 without replacement until they reach a total >= 100.

Given an integer `maxChoosableInteger` and another integer `desiredTotal`, determine if the first player to move can force a win, assuming both players play optimally.

You can always assume that `maxChoosableInteger` will not be larger than 20 and `desiredTotal` will not be larger than 300.

Example

```
Input:
maxChoosableInteger = 10
desiredTotal = 11

Output:
false

Explanation:
No matter which integer the first player choose, the first player will lose.
The first player can choose an integer from 1 up to 10.
If the first player choose 1, the second player can only choose integers from 2 up
to 10.
The second player will win by choosing 10 and get a total = 11, which is >= desired
Total.
Same with other integers chosen by the first player, the second player will always
win.
```

Solution 1

After solving several "Game Playing" questions in leetcode, I find them to be pretty similar. Most of them can be solved using the **top-down DP** approach, which "brute-forcely" simulates every possible state of the game.

The key part for the top-down dp strategy is that we need to**avoid repeatedly solving sub-problems**. Instead, we should use some strategy to "remember" the outcome of sub-problems. Then when we see them again, we instantly know their result. By doing this, ~~we can always reduce time complexity from **exponential** to **polynomial**~~.
(**EDIT:** Thanks for @billbirdh for pointing out the mistake here. For this problem, by applying the memo, we at most compute for every subproblem once, and there are `O(2^n)` subproblems, so the complexity is `O(2^n)` after memorization. (Without memo, time complexity should be like `O(n!)`)

For this question, the key part is: `what is the state of the game`? Intuitively, to uniquely determine the result of any state, we need to know:

1. The unchosen numbers
2. The remaining desiredTotal to reach

A second thought reveals that **1)** and **2)** are actually related because we can always get the **2)** by deducting the sum of chosen numbers from original desiredTotal.

Then the problem becomes how to describes the state using**1)**.

In my solution, I use a **boolean array** to denote which numbers have been chosen, and then a question comes to mind, if we want to use a Hashmap to remember the outcome of sub-problems, can we just use `Map<boolean[], Boolean>` ? **Obviously we cannot**, because the if we use boolean[] as a key, the reference to boolean[] won't reveal the actual content in boolean[].

Since in the problem statement, it says `maxChoosableInteger` will not be larger than `20`, which means the length of our **boolean[] array** will be less than `20`. Then we can use an `Integer` to represent this boolean[] array. How?

Say the boolean[] is `{false, false, true, true, false}`, then we can transfer it to an Integer with binary representation as `00110`. Since Integer is a perfect choice to be the key of HashMap, then we now can "memorize" the sub-problems using `Map<Integer, Boolean>`.

The rest part of the solution is just simulating the game process using the top-down dp.

```
public class Solution {
    Map<Integer, Boolean> map;
    boolean[] used;
    public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
        int sum = (1+maxChoosableInteger)*maxChoosableInteger/2;
        if(sum < desiredTotal) return false;
        if(desiredTotal <= 0) return true;

        map = new HashMap();
        used = new boolean[maxChoosableInteger+1];
        return helper(desiredTotal);
    }

    public boolean helper(int desiredTotal){
        if(desiredTotal <= 0) return false;
        int key = format(used);
        if(!map.containsKey(key)){
    // try every unchosen number as next step
            for(int i=1; i<used.length; i++){
                if(!used[i]){
                    used[i] = true;
     // check whether this lead to a win (i.e. the other player lose)
                    if(!helper(desiredTotal-i)){
                        map.put(key, true);
                        used[i] = false;
                        return true;
                    }
                    used[i] = false;
                }
            }
            map.put(key, false);
        }
        return map.get(key);
    }

// transfer boolean[] to an Integer
    public int format(boolean[] used){
        int num = 0;
        for(boolean b: used){
            num <<= 1;
            if(b) num |= 1;
        }
        return num;
    }
}
```

Updated: Thanks for @ckcz123 for sharing the great idea. In Java, to denote `boolean[]`, an easier way is to use `Arrays.toString(boolean[])`, which will transfer a `boolean[]` to sth like `"[true, false, false, ....]"`, which is also not limited to how `maxChoosableInteger` is set, so it can be generalized to arbitrary large `maxChoosableInteger`.

written by leogogogo original link here

Solution 2

```
public class Solution {

    private Boolean[] win;
    int choosen = 0;

    public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
        if (desiredTotal == 0) {
            return true;
        }
        if ((1 + maxChoosableInteger) * maxChoosableInteger / 2 < desiredTotal) {
            return false;
        }
        win = new Boolean[1 << maxChoosableInteger];
        return canWin(maxChoosableInteger, desiredTotal, 0);
    }

    private boolean canWin(int n, int total, int now) {
        if (win[choosen] != null)
            return win[choosen];
        if (now >= total) {
            win[choosen] = false;
            return false;
        }
        for (int i = 1; i <= n; i++) {
            int bit = 1 << (i - 1);
            if ((choosen & bit) == 0) {
                choosen ^= bit;
                boolean ulose = !canWin(n, total, now + i);
                choosen ^= bit;

                if (ulose) {
                    win[choosen] = true;
                    return true;
                }
            }
        }
        win[choosen] = false;
        return false;
    }
}
```

written by wzdxt original link here

Solution 3

The solution is quite strightforward. First off we have to eliminate primitive cases. So,

- if the first player can choose a number, which is already greater than or equal to the desired total obviously it wins.
- If max choosable integer is less than the desired total, but if it exceeds the desired total in sum with any other number then the first player looses anyway.
- If the sum of all number in the pool cannot exceed or reach the desired total, then no one can win.

Now, for the other cases we can useMiniMax logic to reveal the winner. Because both player play optimally, In order to win, the first player has to make a choice, that leaves the second player no chance to win. Thus, at each step we consider all the possible choices by the current player and give turn to the second player recursively. If we find a move, after which the second player looses anyway or we have already exceed the desired total by adding the chosen number, we return true, i.e the current player wins. This way the game looks like the following tree:

```
     player1 ->  0
              /| ...\
  player2 -> 1 2 ....max
            /|\ ..../ | \
player1 -> 2 3...  1  2 ..max-1
           ...                \
player1 ->   /      |     \   loose
player2 -> loose   win    loose
```

The figure above helps to imagine how the algorithm considers all possible scenarios of the game. The leafs of the game tree are loose or win states for one of the players. Finally the logic concludes to the idea, that if some branch does not contain any leaf that ends with win state for player2, the move associated with that branch is the optimal one for the first player.

P.S: Time complexity of naive implementation will work for O(n!). Therefore we have to memorize branch states after traversing once.

```
public class Solution {
    Map<Integer, Boolean> set[];
    public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
        if(maxChoosableInteger >= desiredTotal) return true;
        if(maxChoosableInteger+1 >=desiredTotal) return false;
        set = new Map[301];
        for(int i  =0 ;i<301;i++) set[i] = new HashMap<>();
        if(maxChoosableInteger*(maxChoosableInteger+1)/2 < desiredTotal) return f
alse;
        return canWin((1<<maxChoosableInteger+1)-1, desiredTotal);
    }

    public boolean canWin(int set1, int total){
        if(set[total].containsKey(set1)) return set[total].get(set1);
        for(int i = 20;i>=1;i--){
            int p = (1<<i);
            if((p&set1) == p){
                int set1next = (set1^p);
                int totalNext = total - i;
                if(totalNext<=0) return true;
                boolean x;
                if(set[totalNext].containsKey(set1next)) x = set[totalNext].get(s
et1next);
                else x = canWin(set1next, totalNext);
                if(!x){
                    set[total].put(set1, true);
                    return true;
                }
            }
        }
        set[total].put(set1, false);
        return false;
    }
}
```

written by ZhassanB original link here