Flip Game II

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: `+` and `-`, you and your friend take turns to flip two **consecutive** `"++"` into `"--"`. The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to determine if the starting player can guarantee a win.

For example, given `s = "++++"`, return true. The starting player can guarantee a win by flipping the middle `"++"` to become `"+--+"`.

Follow up:
Derive your algorithm's runtime complexity.

Solution 1

At first glance, backtracking seems to be the only feasible solution to this problem. We can basically try every possible move for the first player (Let's call him 1P from now on), and recursively check if the second player 2P has any chance to win. If 2P is guaranteed to lose, then we know the current move 1P takes must be the winning move. The implementation is actually very simple:

```
int len;
string ss;
bool canWin(string s) {
    len = s.size();
    ss = s;
    return canWin();
}
bool canWin() {
    for (int is = 0; is <= len-2; ++is) {
        if (ss[is] == '+' && ss[is+1] == '+') {
            ss[is] = '-'; ss[is+1] = '-';
            bool wins = !canWin();
            ss[is] = '+'; ss[is+1] = '+';
            if (wins) return true;
        }
    }
    return false;
}
```

For most interviews, this is the expected solution. Now let's check the time complexity: Suppose originally the board of size N contains only '+' signs, then roughly we have:

```
T(N) = T(N-2) + T(N-3) + [T(2) + T(N-4)] + [T(3) + T(N-5)] + ...
        [T(N-5) + T(3)] + [T(N-4) + T(2)] + T(N-3) + T(N-2)
     = 2 * sum(T[i])  (i = 3..N-2)
```

You will find that T(N) = 2^(N-1) satisfies the above equation. Therefore, this algorithm is at least exponential.

Can we do better than that? Sure! Below I'll show the time complexity can be reduced to O(N^2) using Dynamic Programming, but the improved method requires some non-trivial understanding of the game theory, and therefore is not expected in a real interview. If you are not interested, please simply skip the rest of the article:

> Concept 1 (**Impartial Game**): Given a particular arrangement of the game board, if either player have exactly the same set of moves should he move first, and both players have exactly the same winning condition, then this game is called **impartial game**. For example, chess is not impartial because the players can control only their own pieces, and the +- flip game, on the other

> hand, is impartial.

--

> Concept 2 (**Normal Play vs Misere Play**): If the winning condition of the game is that the **opponent has no valid moves**, then this game is said to follow the **normal play convention**; if, alternatively, the winning condition is that the **player himself has no valid moves,** then the game is a **Misere** game. Our +- flip has apprently normal play.

Now we understand the the flip game is an impartial game under normal play. Luckily, this type of game has been extensively studied. Note that our following discussion only applies to normal impartial games.

In order to simplify the solution, we still need to understand one more concept:

> Concept 3 (**Sprague-Grundy Function**): Suppose x represents a particular arrangement of board, and x*0, x*1, x*2, ... ,x*k represent the board after a valid move, then we define the Sprague-Grundy function as:

```
g(x) = FirstMissingNumber(g(x_0), g(x_1), g(x_2), ... , g(x_k)).
```

> where FirstMissingNumber(y) stands for the smallest positive number that is not in set y. For instance, if g(x*0) = 0, g(*x1) = 0, g(x_k) = 2, then g(x) = FMV({0, 0, 2}) = 1.

Why do we need this bizarre looking S-G function? Because we can instantly decide whether 1P has a winning move simply by looking at its value. I don't want to write a book out of it, so for now, please simply take the following theorem for granted:

> Theorem 1: If g(x) != 0, then 1P must have a guaranteed winning move from board state x. Otherwise, no matter how 1P moves, 2P must then have a winning move.

So our task now is to calculate g(board). But how to do that? Let's first of all find a way to numerically describe the board. Since we can only flip ++ to --, then apparently, we only need to write down the lengths of consecutive ++'s of length >= 2 to define a board. For instance, ++--+-++++-+----- can be represented as (2, 4).

(2, 4) has two separate '+' subsequences. Any operation made on one subsequence does not interfere with the state of the other. Therefore, we say (2, 4) consists of two **subgames**: (2) and (4).

Okay now we are only one more theorem away from the solution. This is the last theorem. I promise:

> Theorem 2 (**Sprague-Grundy Theorem**): The S-G function of game x = (s1, s2, ..., sk) equals the XOR of all its subgames s1, s2, ..., sk. e.g. g((s1, s2, s3)) = g(s1) XOR g(s2) XOR g(s3).

With the S-G theorem, we can now compute any arbitrary g(x). If x contains only one number N (there is only one '+' subsequence), then

```
g(x) = FMV(g(0, N-2), g(1, N-3), g(2, N-4), ... , g(N/2-1, N-N/2-2));
     = FMV(g(0)^g(N-2), g(1)^g(N-3), g(2)^g(N-4)), ... g(N/2-1, N-N/2-2));
```

Now we have the whole algorithm:

```
Convert the board to numerical representation: x = (s1, s2, ..., sk)
Calculate g(0) to g(max(si)) using DP.
if g(s1)^g(s2)^...^g(sk) != 0 return true, otherwise return false.
```

Calculating g(N) takes O(N) time (N/2 XOR operations plus the O(N) First Missing Number algorithm). And we must calculate from g(1) all the way to g(N). So overall, the algorithm has an O(N^2) time complexity.

Naturally, the code is bit more complicated than the backtracking version. But it reduces the running time from ~128ms to less than 1ms. The huge improvement is definitely worth all the hassle we went through:

```
int firstMissingNumber(unordered_set<int> lut) {
    int m = lut.size();
    for (int i = 0; i < m; ++i) {
        if (lut.count(i) == 0) return i;
    }
    return m;
}

bool canWin(string s) {
    int curlen = 0, maxlen = 0;
    vector<int> board_init_state;
    for (int i = 0; i < s.size(); ++i) {
        if (s[i] == '+') curlen++;              // Find the length of all continu
ous '+' signs
        if (i+1 == s.size() || s[i] == '-') {
            if (curlen >= 2) board_init_state.push_back(curlen);    // only lengt
h >= 2 counts
            maxlen = max(maxlen, curlen);       // Also get the maximum continuou
s length
            curlen = 0;
        }
    }          // For instance ++--+--++++-+ will be represented as (2, 4)
    vector<int> g(maxlen+1, 0);    // Sprague-Grundy function of 0 ~ maxlen
    for (int len = 2; len <= maxlen; ++len) {
        unordered_set<int> gsub;    // the S-G value of all subgame states
        for (int len_first_game = 0; len_first_game < len/2; ++len_first_game) {
            int len_second_game = len - len_first_game - 2;
            // Theorem 2: g[game] = g[subgame1]^g[subgame2]^g[subgame3]...;
            gsub.insert(g[len_first_game] ^ g[len_second_game]);
        }
        g[len] = firstMissingNumber(gsub);
    }

    int g_final = 0;
    for (auto& s: board_init_state) g_final ^= g[s];
    return g_final != 0;    // Theorem 1: First player must win iff g(current_sta
te) != 0
 }
```

written by stellari original link here

Solution 2

The idea is try to replace every `"++"` in the current string `s` to `"--"` and see if the opponent can win or not, if the opponent cannot win, great, we win!

For the time complexity, here is what I thought, let's say the length of the input string `s` is `n`, there are at most `n - 1` ways to replace `"++"` to `"--"` (imagine `s` is all `"+++..."`), once we replace one `"++"`, there are at most `(n - 2) - 1` ways to do the replacement, it's a little bit like solving the N-Queens problem, the time complexity is `(n - 1) x (n - 3) x (n - 5) x ...`, so it's `O(n!!)`, double factorial.

That's what I thought, but I could be wrong :)

```
public boolean canWin(String s) {
  if (s == null || s.length() < 2) {
    return false;
  }

  for (int i = 0; i < s.length() - 1; i++) {
    if (s.startsWith("++", i)) {
      String t = s.substring(0, i) + "--" + s.substring(i + 2);

      if (!canWin(t)) {
        return true;
      }
    }
  }

  return false;
}
```

written by jeantimex original link here

Solution 3

```
public boolean canWin(String s) {
    List<String> list = new ArrayList<>();
    for(int i = 0; i < s.length() - 1; i++){
        if(s.charAt(i) == '+' && s.charAt(i + 1) == '+')
            list.add(s.substring(0, i) + "--" + s.substring(i + 2, s.length()));
// generate all possible sequence after every attempt
    }
    /*if(list.isEmpty())
        return false;*/
    for(String str : list){
        if(!canWin(str))             // if there is any one way the next player c
an't win, take it and you'll win
            return true;
    }
    return false;
}
```

written by skyflash original link here