NOI.PH Training: Game Theory

Combinatorial Game Theory

Cisco Ortega

Contents

1 Introduction

Game Theory is a rich and diverse field of mathematics, because hey, who doesn't love playing games? Problems in Game Theory usually involve, naturally, some sort of game, and to solve them you would need to find some optimal winning strategy. Some games are simply Win/Lose, while others are more nuanced with some sort of 'point system' or objective function that you want to maximize.

By now, you've already dealt with many optimization problems. For example, given an array, find the subarray that has the largest sum. The heart of what makes a problem part of Game Theory, however, is that it tackles situations with *multiple players* competing against one another.

Think of even something as simple as Tic-Tac-Toe. It would certainly be very boring as a single player game. What makes it interesting to analyze is the fact that there is a back-and-forth, an interplay between the two players. The players respond to each other and adjust their strategies accordingly based on the changing options available to them. Amidst the complex web of interactions presented by all the moves available to each player, game theory problems ask you to find which is the optimal move.

Combinatorial Game Theory will be the main topic of this module, focusing on a specific subclass of Game Theory problems which we will later define. These are the types of problems that typically show up in contests, and typically involve two or more rational actors playing some sort of mathematically-flavored game.

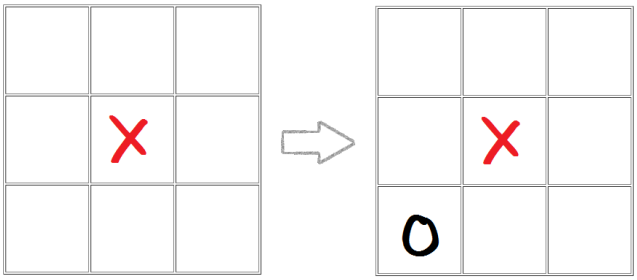
2 Games

2.1 What is a game?

We will begin by presenting an intentionally broad and abstract definition of a game, then try to solidify the concept with some concrete examples. Then, we will proceed to look at games with specific properties which allow us to solve them more efficiently. I apologize for the spam of definitions that will follow, but don't worry, because they're more of terms that you should read and understand rather than anything that you actually need to memorize. The concepts should be intuitive, too, and we're just giving them names.

What is a game in the first place? The kinds of games that we usually see in programming competitions typically have the following elements.

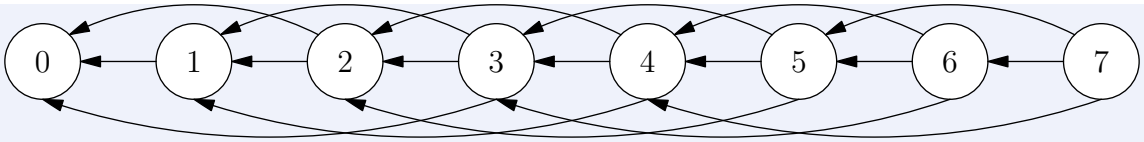
- **Players**. These are the actors who take turns making decisions, trying to win the game. In most games we will deal with, there are usually only two of them.
- **States**. Rather abstractly, these are really just all the possible states that your game could be in at any given moment. For instance, in Tic-Tac-Toe, a state would be a 3×3 grid filled with Xs and Os in some way.
- **Moves**. Moves allow the players to transition from state to state. For instance, in Tic-Tac-Toe again, a valid move would be to place an X (or an O) on one of the free spaces on the grid; filling up that space transitions us from one game state to another. The set of available moves that the current turn player can choose from often varies from state to state.



Two game states in Tic-Tac-Toe are pictured above, with the move "place a O in the bottom-right corner" transitioning one state into the other.

Personally, I find it nice to visualize a game as a **directed graph**. The vertices represent each of the game states, while the directed edges indicate the moves which transition one state to another. When presented with an unusual game, I highly encourage you to sketch out a diagram; visualization is key.

Example 2.1. Let's examine one of the simplest games, the **Subtraction Game**. The game begins with n coins on the table. Alvin and Barbie take turns removing exactly 1, 2, or 3 coins from the table[a]; Alvin always goes first. The winner is the one who takes the last coin from the table; alternatively, the loser is the first one who cannot take a coin on their turn.



Here, each game state is simply how many coins there are left on the table. Our moves would be the taking of 1, 2, or 3 coins. Suppose we started a game with $n = 7$ coins. So, we draw 8 vertices labeled from 0 to 7, and draw directed edges from each vertex to the ones whose values are 1, 2, or 3 less than it. For example, vertex 5 is connected to vertices 4, 3, and 2.

[a]Subtraction games are a general class of games characterized by removing some number of coins from the table, not necessarily 1, 2, and 3. Other introduction to Game Theory courses might use 1 and 2, or 1, 2, 3 and 4. It is understood, though, that for this module, when I say "the subtraction game", I am referring to the case where you can only take 1, 2, or 3 coins at a time.

There are lots of games out there, but this module is specifically about **Combinatorial Game Theory**, so let's specify some assumptions we will be making for the remainder of this module about the specific kinds of games we will be dealing with. Unless stated otherwise, assume that

- The games are only between two players. Also, I will be referring to the two players as Alvin and Barbie. There's no real reason why, it's just cleaner and a lot more fun to have names to refer to instead of 'this player' and 'that other player'. Since these two players will be taking turns, the game is sequential.
- Both players are playing optimally. The players will never make any mistakes, and will always take a move that will make them win if possible.
- The games are finite. They will always end after a finite number of moves.
- The games are deterministic and both players have complete information about the game state at all times. We will not be dealing with games like poker, where one player has information the other player doesn't (the cards in their hand).
- The games are **impartial**, meaning that the set of moves available to each player at each state is completely identical. Tic-Tac-Toe is not an impartial game since one player can only draw Xs while the other player can only draw Os; non-impartial games are called **partisan** games. The subtraction game, meanwhile, *is* impartial, since Alvin and Barbie *both* can only take 1, 2, or 3 coins at each state.

Exercise 2.2. Consider the following game that is similar to the subtraction game. The game begins with n coins on the table. Alvin and Barbie take turns, and can either remove 1, 2, or 3 coins from the table or add 1, 2, or 3 coins to the table. Which, if any, of our assumptions is not satisfied by this game?

Finally, let's talk about how to *win*. The winner is typically given as part of the game rules; in the subtraction game, we said that the winner is the one who takes the last coin from the table. Notice that given the restrictions above, we should already be able to tell with 100 percent certainty whether the first player will win, or if the second player will, assuming that they both play their cards right. It's sort of like how good chess players already know when a game is decided; although the game may not be in checkmate just yet, if the winning player knows what they are doing, then they will be able to execute a series of moves such that the opponent, no matter what they do, is guaranteed to eventually fall to checkmate.

The difference is that rather than merely seeing only a few moves into the future, we assume that Alvin and Barbie have supreme supercomputer-level insight and can explore every single possibility and timeline that results from every possible series of moves they could make. If there is a winning line of play, they will always be able to see it and take it. Thus, given the information about some state and whose turn it currently is, it will already be a foregone conclusion whether Alvin or Barbie is the winner.

2.2 Finding Win States and Loss states using Dynamic Programming

A **win state**, which I will typically indicate with a `W`, is a state where the current turn player has a winning strategy—that is, there is some series of moves that they can take to guarantee a win from that spot. If no matter what the current turn player does, they are destined to lose, then we call that state a **loss state** instead, which I will indicate with a `L`. [1]

For example, in the subtraction game, $n = 4$ is a loss state. Suppose it is Alvin's turn right now. If he takes 1 coin, then there will be 3 coins left and Barbie can take all of them for the win; if he takes 2 coins, Barbie can also take the remaining 2, and if he takes 3, Barbie can still take the remaining 1. So, no matter what Alvin does, there's no way he can win this, so $n = 4$ is a loss state. However, $n = 5$ is a Win state. If it is Alvin's turn, she can take 1 coin from the pile, and then Barbie will start her turn with 4 coins—*but*, we already argued earlier that 4 was a loss state. No matter how many coins Barbie takes, Alvin can take the remaining coins for his win. Since there existed a line of play that led to his victory, we know that $n = 5$ is a win state. Note that there only has to exist *at least* one path to victory. Certainly not *all* moves are winning moves; if at $n = 5$, Alvin had chosen to take 3 coins instead, then Barbie would have just taken the remaining 2 coins for her win. However, as we said earlier, we assume that Alvin and Barbie are always playing optimally, and Alvin would always have been able to spot that winning line.

Exercise 2.3. Alvin and Barbie play the subtraction game with $n = 8$ coins. If Alvin goes first, who among the two has a winning strategy?

Okay, so this revealed something nice about win states and loss states, and how we can calculate them with dynamic programming. We can recursively determine whether a given state is a win state or a loss state with the following procedure,

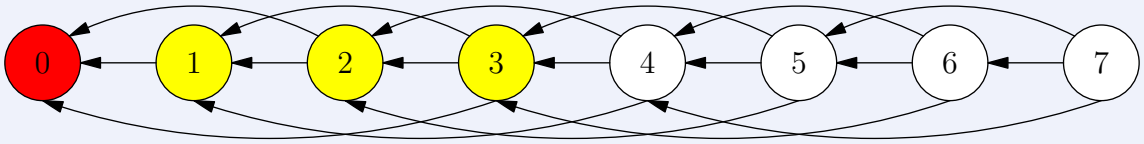
- The terminal states (the ones that end the game) are our base cases. In most games, they are often loss states. For instance, in the subtraction game, $n = 0$ is a terminal state because the game ends when there are no coins left on the table; it is also a loss state because if your turn starts with 0 coins on the table, you lose.
- If a state points to[2] **at least** one loss state, it is a win state. The current turn player will simply take that winning move and force their opponent into a loss state.
- If a state **only** points to win states, then that state is a loss state. If all the moves available to the current turn player lead to wins for the opponent, then they are completely helpless. All timelines lead to failure, so to speak.

[1] The choice of `W` and `L` is certainly not standard. I've also seen the use of `P` and `N`, for whether the **p**revious or **n**ext player will win. Go with whatever makes you comfortable.

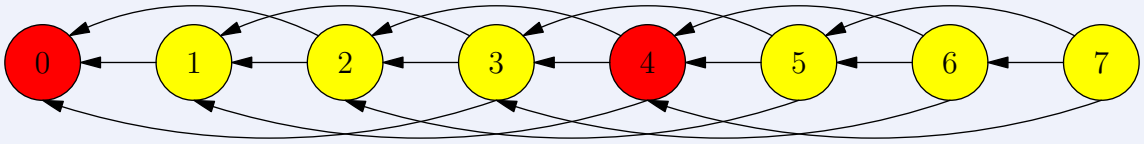
[2] I informally use 'points to' in order to mean "there exists a move that lets me transition from this state to that one". You get it, though, right? Just picture the directed graph diagram.

An important assumption here is that there are no cycles in our constructed graph; this is why it was important when we said earlier that our games should be finite.

Example 2.4. Let's revisit the subtraction game with this knowledge. I'm going to color the loss states in red, and the win states in yellow.



By definition, we set our terminal state $n = 0$ to be a loss state. Then, we notice that the states $n = 1, 2, 3$ all point to $n = 0$, a loss state. Therefore, they must be win states!



Notice that $n = 4$ only points to $n = 1, 2, 3$, which we know are all win states. All the states that $n = 4$ points to are win states, therefore $n = 4$ itself is a loss state. Then, since $n = 5, 6, 7$ all point to $n = 4$, a loss state, $n = 5, 6, 7$ are all win states.

In general, the general pseudocode for determining the winning and losing states looks something like this:

```
list_of_states <- list of states
toposort(list_of_states) #toposort using the reverse edges; put the terminal
↪  states first in the list

for each non-terminal state u in list_of_states:
    evaluate all states v that u points to
    if any of the v are loss states:
        u is a win state
    else u only points to win states:
        u is a loss state
```

Exercise 2.5. Why do the states have to be topologically sorted?

This is how you would answer it bottom-up. We can also do things top-down with memoization:

```
memo <- memoization table
def is_win(u):
    if u is in memo:
        return memo[u]
    else:
```

```
        evaluate is_win(v) for all v that u points to
        if any of the v is a loss state:
            return memo[u] = win
        else u only points to win states:
            return memo[u] = lose
```

For example, I used the following code to generate the win states and loss states in the subtraction game:

```cpp
#include<bits/stdc++.h>
using namespace std;

bool is_win[N+1];
void play_game() {
    is_win[0] = false;
    for(int n = 1; n <= N; n++) {
        is_win[n] = false; // assume that it doesn't point to a loss state
        for(int i = 1; i <= 3 && n-i >= 0; i++) {
            if(!is_win[n-i]) { // if it does, then it is actually a win state
                is_win[n] = true;
                break;
            }
        }
    }
}
int main() {
    play_game();
    for(int i = 1; i <= N; i++)
        cout << i << ": " << (is_win[i]? "W" : "L") << '\n';
}
```

This is only one of many possible implementations; go with what suits you best. One thing is that I filled my table bottom-up, though you may choose to do it top-down.

Note that with the DP solution, you can handle partisan games as well. However, you would then also need to include **the current turn player** as part of the game state. As an example, consider that in chess, whether it is White or Black's turn right now is just as important as where the pieces themselves are on the board.

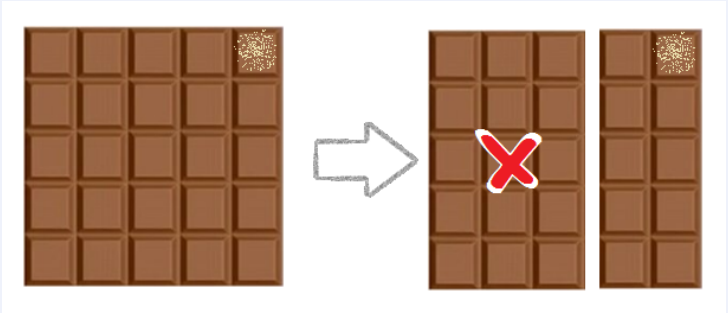
Exercise 2.6. The game begins with n coins on the table. Alvin and Barbie take turns removing exactly 1, 3, or 5 coins from the table; Alvin always goes first. The winner is the one who takes the last coin from the table; alternatively, the loser is the first one who cannot take a coin on their turn.

Write a program that outputs the winner of each game (either Alvin or Barbie) for each n from 1 to 10^5.

Exercise 2.7. The game begins with n coins on the table. Alvin and Barbie take turns removing exactly s coins from the table, where s is any perfect square (and not greater than the number of coins currently on the table); Alvin always goes first. The winner is the one who takes the last coin from the table; alternatively, the loser is the first one who cannot take a coin on their turn.

Write a program that outputs the winner of each game (either Alvin or Barbie) for each n from 1 to 10^5.

Exercise 2.8. Suppose that Alvin and Barbie have a chocolate bar divided into an $m \times n$ grid, and only the top-right-most square is poisoned. Alvin and Barbie take turns 'snapping' the bar in two—i.e., they make a single straight cut through one of the gridlines—and eating one of the two pieces. Then, they pass the remaining piece to the other player, and so on. Alvin always goes first. The loser is the one who has to eat the poisoned square.



Write a program that outputs the winner of each game (either Alvin or Barbie) for each m, n from 1 to 10^3.

Exercise 2.9. The game begins with 1 red coin, 2 blue coins, and 3 green coins on the table. Alvin and Barbie take turns removing as many coins as they want from the table, as long as they remove at least one coin each turn, and all coins removed that turn are the same color; Alvin always goes first. The winner is the one who takes the last ball from the table; alternatively, the loser is the first one who cannot take a coin on their turn.

Who wins, Alvin or Barbie? Draw the graph representing the states and how they transition to each other.

Exercise 2.10. The game begins with a row of n coins. Alvin and Barbie take turns removing coins from the table; Alvin always goes first. At any move, they may take away one or two *consecutive* coins from anywhere, not *just* at the beginning or the end. Two coins are considered "consecutive" if they are consecutive in the original state; that is, two far-apart coins don't become consecutive even when all coins between them are taken. The first one who cannot make a valid move on their turn is the loser.

When $n = 5$, who wins, Alvin or Barbie? Draw the graph representing the states and how they transition to each other.

2.3 Pattern Recognition

Knowing how to implement a game with Dynamic Programming is essential to tackling all the weirder games out there. However, suppose that I asked you whether Alvin or Barbie would win the subtraction game for $n = 10^{18}$. Obviously, we can no longer directly use the DP solution to solve this.

Such constraints usually are going to hint to you that there is some sort of pattern, or underlying mathematical structure behind the particular rules of this game. These problems end up boiling down to pattern recognition—if you can spot the pattern, you can probably figure out from there how to evaluate a game state quickly.

And, as we all know, the best way to tackle a pattern-recognition problem is to **actually write down the first few terms in the sequence**, and see if you can spot it there. The ruleset itself is rather abstract, so it can be hard to think about what the pattern could possibly be just from looking at it. That's why I find it helpful to actually have some numbers I can stare at.

Now, you *could* write this out by hand, but seriously, why would you? You have a computer! Use it. Even if the bounds are ludicrously high, **writing the DP solution is still valuable** because it will provide you with a springboard for your ideas. Some patterns are astonishingly simple if you see the values themselves, but are not at all obvious from the rules of the game.

Example 2.11. Let's actually run our code from earlier for the subtraction game and see what comes out.

```
0: L
1: W
2: W
3: W
4: L
5: W
6: W
7: W
8: L
9: W
10: W
11: W
12: L
13: W
14: W
15: W
16: L
17: W
18: W
19: W
20: L
```

As you might have suspected by now, it seems that loss states occur at multiples of 4, and every other state is a win state.

If this were in a contest and you were reasonably certain about the pattern that you spotted, you could just go ahead and submit your code without proving that the pattern you spotted is actually true, the legendary "Proof by AC". That is a perfectly reasonable thing to do, and you will see in the problemset later that between spotting a pattern and proving that it holds, the latter can sometimes be a *lot* more involved than the former. Just make sure the pattern you think you spotted is actually correct!

The following example should illustrate a case where an attempt at pattern-finding can go wrong.

Example 2.12. Let's look at Exercise 2.7. Here's the output for n up to 20.

```
0: L
1: W
2: L
3: W
4: W
5: L
6: W
7: L
8: W
9: W
10: L
11: W
12: L
13: W
14: W
15: L
16: W
17: L
18: W
19: W
20: L
```

Aha! So the pattern goes `L, W, L, W W` periodically, so all we have to do is check the value of n mod 5. If it is 1, 3, or 4, then it is a win state, and if it is 0 or 2, it is a loss state. Problem solved.

Except... let's print out more values. Here's the values from 20 to 30.

```
20: L
21: W
22: L
23: W
24: W
25: W
26: W
27: W
28: W
29: W
30: W
```

Yikes! Our pattern fell apart. We falsely identified 25 and 30 as loss states, when they really should have been win states (it should be fairly obvious in the case of 25 for why this is a win state).

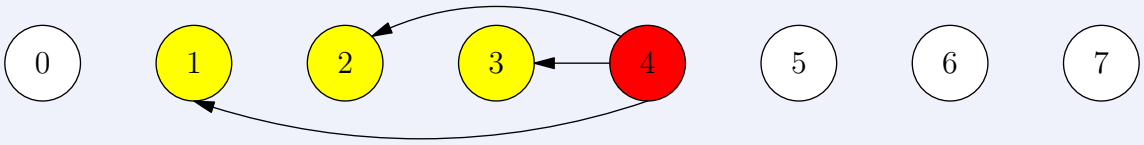
Situations like this show that it never hurts to prove your answer, just to be really certain there isn't something you're missing. Since whether our game states are win states or loss states is recursively defined, it most naturally lends itself to be proven with **induction**, specifically usually **strong induction**. Seriously, induction is going to be your best friend in proving these kinds of problems.

Example 2.13. Let's prove that in our first example of the subtraction game in Example 2.1, n is a loss state when it is a multiple of 4, and a win state otherwise.

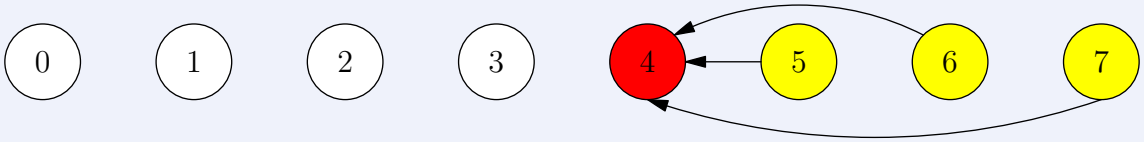
First we establish our base cases, which are n from 0 to 3. By hand, we can verify that they satisfy our pattern.

Next, the strong induction. Assume that our pattern holds for all from 0 to $n-1$. Let's show that it holds true for n. If n is a multiple of 4, then none of $n-1$, $n-2$, or $n-3$ are

multiples of 4; by inductive hypothesis, these are all win states. Therefore, if n is a multiple of 4, then it only points to win states, therefore it is a loss state, just like what our pattern promised.



If n is not a multiple of 4, then at least one of $n-1$, $n-2$, or $n-3$ is (picture how they look on the number line). By inductive hypothesis, this multiple of 4 must be a loss state. Therefore, if n is not a multiple of 4, it points to a loss state, therefore it is a win state, just like what our pattern promised.



Exercise 2.14. For Exercise 2.6 and Exercise 2.8, spot the pattern in the win/loss states and come up with a $\mathcal{O}(1)$ formula for each that determines whether a given n (or m, n, for Exercise 2.8) is a win for Alvin or a win for Barbie. Prove these formulas with strong induction.

3 Nim games

3.1 Nim

Let's move on to the archetypal Combinatorial Game Theory game—Nim. In Nim, there are n piles on the table that each contain some number of coins. The number of coins in each of the piles is given as $p_1, p_2, \cdots p_n$. On each of Alvin and Barbie's turns, they can remove any number of coins that they want, as long as **they all come from the same pile**. Again, Alvin always goes first. The winner is the one who takes the last coin from the table; alternatively, the loser is the first one who cannot take a coin on their turn.

Given the initial number of coins in each of the piles, determine whether Alvin or Barbie will win the game.

I highly encourage you to play a few games of Nim yourself, first, before proceeding. Take some physical coins or whatever tokens you have and, even if you don't know the winning strategy just yet, just play around with it and try to get a feel for how it works and maybe an idea of what states are winning and which are losing. If you can, maybe enlist a friend to play against you!

Dynamic Programming doesn't exactly sound like the best idea here. If a pile can contain at most p coins, then the number of states we'd have to keep is $\mathcal{O}(p^n)$, which looks gross. However, it turns out that there's a rather beautiful and simple solution to Nim that runs in only $\mathcal{O}(n)$ time. This fantastic result is one of the main reasons why mathematicians—and problem setters—love Nim so much.

I recommend this wonderful video by Mathologer[3] about Nim. You get to watch an example of a game of Nim, and it also contains a good overview on what the winning strategy is, and a proof for why it works. Later in this module I will in fact be using the same terminology that Mathologer uses in his video because I find it very nice. One thing to note is that the game played in the video is **Misère Nim**, which means that the last person to take a card is the *loser*, instead of the winner, but we will see that this doesn't actually make too much of a difference at all in Nim.

First, recall the bitwise XOR operation. The XOR of two numbers is often written as $a \oplus b$ in literature, although in C++ and Java, the operator uses the ^ symbol. We get the XOR of two numbers by writing them in binary and then performing "addition without carrying"; in other words, we perform a logical XOR on each pair of bits. For example, $10 \oplus 12 = 6$, because

$$\begin{array}{rcccc} & 1 & 0 & 1 & 0 \\ \oplus & 1 & 1 & 0 & 0 \\ \hline & 0 & 1 & 1 & 0, \end{array}$$

and 0110_2 converted back into decimal notation is 6. Because of the notion that XOR is "addition without carrying", I will call the result of an XOR operation to be the XORsum (like sum for addition, or product for multiplication).

For a more visual understanding of XOR, imagine that we represented each number as a pile of coins, and took each pile and separated the coins into subpiles whose sizes are powers of 2, matching each pile's binary representation. Then, if we see two subpiles of the same size, we pair them together and they cancel out. Thus, at the end, for each power of 2, there should be at most 1 subpile of each size left unpaired. Taking the sum of all unpaired subpiles gives us the XOR.

With that out of the way, let us present one of the most widely known theorems in Combinatorial Game Theory.

[3] https://www.youtube.com/watch?v=niMjxNtiuu8

Theorem 3.1. Suppose we are playing a game of Nim with n piles, and the piles contain $p_1, p_2, \cdots p_n$ stones, respectively. Let the **Nim-sum** of the game be the XORsum of all the piles, $p_1 \oplus p_2 \oplus \cdots \oplus p_n$. This game state is a loss state if and only if the Nim-sum is 0; if it is nonzero, then the game state is a win state.

Let's look at some concrete examples. Suppose Alvin goes first and Barbie goes second in each of the following games.

Example 3.2. If there are three piles with 1, 2, and 3 coins, then Alvin definitely loses, because $1 \oplus 2 \oplus 3 = 0$.

If there are four piles with 2, 4, 5, and 6 coins, then Alvin definitely wins, because $2 \oplus 4 \oplus 5 \oplus 6 = 5$, which is nonzero. He can remove the entire pile with 5 coins. Then, the game state that Barbie has is a losing one because $2 \oplus 4 \oplus 6 = 0$.

We also note that although the winning strategy itself is slightly different for Misère Nim, the check for whether Alvin or Barbie wins is exactly the same—check if the XORsum of the piles is zero or nonzero. We will delve more into this later when we discuss the strategy for Misère Nim.

Here is an example of code that finds the winner of a Nim game in $\mathcal{O}(n)$.

```
#include<bits/stdc++.h>
using namespace std;
using ll = long long;
int main() {
    int n; cin >> n;
    ll nimsum = 0;
    for(int i = 0; i < n; i++) {
        ll p; cin >> p;
        nimsum ^= p; // ^ is XOR
    }
    cout << (nimsum!=0 ? "Alvin" : "Barbie") << '\n';
}
```

We shall also list some of the properties of the XOR operator that may be helpful when solving problems involving Nim.

- The XOR is commutative, i.e. $a \oplus b = b \oplus a$.
- The XOR is associative, i.e. $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
- The XOR identity is 0, i.e. $a \oplus 0 = a$.
- The XOR inverse of every integer is itself, i.e $a \oplus a = 0$. This is one of the unique properties of XOR that sets it apart from other operations, and it is in my opinion very important to keep in mind. Almost all problems I've solved involving XOR, not just the Nim games, end up abusing this fact in one way or another.

Exercise 3.3. Justify, even if just to yourself, why each of these properties is true.

Exercise 3.4. Note that $a \oplus a = 0$ implies that if there are exactly two piles, then Barbie has a winning strategy if and only if the two piles have an equal number of coins. Describe what Barbie's winning strategy is, i.e. what she should do in response to Alvin, in order to win.

Does this remind you of any of the past exercises in this module?

Now, let's prove why the XOR works!

Proof. Let's temporarily switch our terminology. If a game state has an Nim-sum of 0, instead of a loss state, let's called it a *balanced* state. If a game state has a nonzero Nim-sum, instead of a win state, let's called it an *unbalanced* state. This is mainly for clarity in the mental images that will follow.

First we establish that the empty game is a balanced state. Obviously, if there are no piles, the Nim-sum is 0 and you lose.

Next, note that if you have a balanced state, any move you make to it will make it unbalanced, i.e. if the Nim-sum is 0, changing any of the piles must make it nonzero. This shouldn't be too hard to see by playing around with physical piles, and its formal proof is left as an exercise to the reader.

Finally, note that if you have an unbalanced state, it is always possible to make it balanced, i.e. if the Nim-sum is non-zero, there will always exist a move that makes it zero. Let the Nim-sum here be N. I will start with a visual take on the argument, using Mathologer's imagery of pairing up powers of 2 together, then follow it up with a differently-phrased argument that works with bits instead.

First, the visual argument. Separate each pile into subpiles whose sizes are powers of 2, according to their binary representations. Pair together subpiles of the same size, and since the Nim-sum is non-zero, we are guaranteed that there is at least one subpile left unpaired. Determine the largest unpaired subpile, and locate a pile which has a subpile of this size. At least one such pile is guaranteed to exist (why?).

Now, let's focus our attention on this pile. All of its subpiles that are greater than the largest unpaired subpile, we ignore and keep as is; they're already fine. Then, all of its subpiles smaller than the largest unpaired subpile, just throw them away. Finally, subtract one coin from the largest unpaired subpile itself, destroying it.

Now the largest unpaired subpile is no longer unpaired. The only thing left that needs to be taken care of is that there may still be smaller unpaired subpiles. We can redistribute the coins from what used to be the largest unpaired subpile so that every smaller unpaired subpile now has a partner. Then, once we're done, we can throw away all the excess coins that were unused. We are always going to have enough coins to accomplish this because if there were 2^g coins in the largest unpaired subpile, then there are at most only $2^0 + 2^1 + ... + 2^{g-1} = 2^g - 1$ coins in the subpiles smaller than it. Now, every subpile has a partner, and so we are back in the balanced state.

This is basically the same argument, but made by directly manipulating bits instead of pairing up piles.

We need to choose a pile p_i and a number of coins to take away from it s such that $N \oplus p_i \oplus (p_i - s) = 0$ (we XOR by p_i first to remove it from the XORsum, then the XOR by $p_i - s$ is to add the new value in). Now, if N is nonzero, then it must have a greatest significant bit—i.e. the 'leftmost' 1 in the binary representation of N—which we'll call the gth bit. At

least one of the piles must also have a 1 in its gth bit, since that bit had to have come from somewhere after all. Let's choose one of those piles as our p_i; then, the new number of coins in that pile, $p_i - s$, can be constructed in the following manner,

- All bits greater than the gth bit, we keep the same as p_i.
- The gth bit is set to 0.
- All bits less than the gth bit, we make it match the value of $N \oplus p_i$.

This new value $p_i - s$ is always less than p_i, since the first difference between the two is at the $g$the bit, where $p_i - s$ has a 0 and p_1 has a 1. Therefore, $s > 0$ and is thus a valid choice.

Let's show that $N \oplus p_i \oplus (p_i - s) = 0$. We know that bits greater than the gth bit in N are all zero, by definition. We also know that all bits greater than the gth bit in $p_i \oplus (p_i - s)$ are also zero, since we constructed $p_i - s$ so that they would match. Next, the gth bit is 1 in N, again by definition. The gth bit is 1 in p_i and 0 in $p_i - s$, so the XOR of all three gives us a 0 in the gth bit. Finally, for all bits less than the gth bit, we made the bits in $p_i - s$ match those in N, so their XOR is 0.

□

What's nice about this proof is that it actually tells you what the winning strategy is! If you can perform the XORs in your head (made easier with Mathologer's subpiles visualization), you could even try this game on your friends and they'll never be able to beat you! Well, unless they know the strategy themselves and are dealt a winning state, that is.

Exercise 3.5. Code an interactive Nim game! It should accept the number of piles n and the number of coins in each pile p_1, p_2, $\cdots$, p_n, as well as whether the player or the computer goes first. The computer then alternates between reading player input to process the player's moves, and deciding what move it should make on its own turn. Your program should know the winning move(s) (if one exists) on each of its turns, and should respond to the player's actions to try to win each time.

3.2 Misère Nim

In a lot of the games presented so far, we usually made it so that the first player who cannot make a move is the loser. In a **Misère game**, however, the first player who cannot make a move is instead the *winner*. Alternatively, the last player who makes a move is the loser.

Exercise 3.6. Restate the games from Exercise 2.6 to Exercise 2.10 as Misère games. How do they change?

For a DP solution, this doesn't really change all too much except for the fact that our terminal states are win states instead of loss states. However, what about Misère Nim in particular? Fantastically, we will have to change our winning strategy a little bit, but otherwise it is almost exactly the same as normal Nim!

Theorem 3.7. In Misère Nim, the rules are exactly the same as normal Nim, except the winner is the first one who can**not** take a stone on their turn. If all the piles are of size 1, then the

game state is a win state if n is even, and a loss state is n is odd. Otherwise, it is just like in normal Nim, where a given game state is a loss state if and only if the Nim-sum is 0; if it is nonzero, then the game state is a win state.

Proof. If all the piles are of size 1, then the players really don't have any choice but to take away the piles one by one. Then, the parity of the number of piles should determine the winner.

The winning strategy in Misère Nim is to play exactly the same as in normal Nim until there is only a single pile with more than one coin. Let's say there are m piles remaining, and without loss of generality, $p_1 > 1$, and $p_2 = p_3 = \cdots = p_m = 1$ for all other piles. Then, we shift strategies.

The winning player should leave an *odd* number of piles-of-size-1 on the field. That is, if m is even, they should take all the coins in p_i, and if m is odd, they should take all-but-1 of the coins in p_1. That way, we force our way into the only-piles-of-size-1 scenario described earlier, and we make sure to give our opponent an odd number of piles to guarantee that they have a losing state.

Unless the game already started with only piles of size 1, we will always end up entering this scenario, where there is only one pile left with more than 1 coin in it, since we can only remove coins from one pile at a time. Furthermore, we are guaranteed that the XOR of this state is nonzero (why?), i.e. it is an unbalanced state. Thus, if the winning player recycles the balanced-unbalanced tactic from normal Nim, then if it is an unbalanced state on their turn, they can ensure that it is going to be their turn when only one pile with more than 1 coin is left. Therefore, whether a given state is a win state or a loss state is still decided by the Nim-sum. □

Exercise 3.8. Modify your interactive Nim program so that it can play optimally in Misère Nim as well.

3.3 Reducing Other Games to Nim

This section is just a quick reminder that sometimes you may be given some sort of game with too many states to attack with DP. If so, it might be the case that the game is actually Nim in disguise! In these cases, the problem becomes identifying what these piles actually are. Once you figure that out, the task becomes a straightforward Nim implementation. Realizing that a game is actually Nim and finding out how to transform it into a game of Nim is an incredibly powerful tool that turns out to be more common than you might think.

Example 3.9. Consider the problem **Tower Breakers, Revisited!:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/tower-breakers-revisited-1.

This problem looks similar to our Nim, except with the added restriction that if we have a pile of size X, if we choose to take from this pile, then the remaining number of stones Y must be a proper divisor of X (i.e. a divisor strictly less than X).

Note one thing—if a pile is originally of size p, then at any point of time in the game, the size of that pile will always be a divisor of p. In fact, we can rewrite the 'taking away' operation. Instead of saying that we if we take from a pile of size X, we must leave behind a proper divisor Y, we can instead say that we choose some divisor $d > 1$ of X then replace X with $\frac{X}{d} = Y$.

If we think about this as dividing instead of subtracting, then we should be able to see that

we are simply playing a normal game of Nim on the **number of prime factors** of each tower as the size of our piles. Try to think about why that is in fact the case.

Exercise 3.10. Consider Exercise 2.8. Can this problem be rephrased to be about a Nim game?

3.4 Some Variants of Nim

3.4.1 Nim with Addition

I don't think this variant really has a name, so I'm going to give it the rather uninspired named of "Nim with Addition". Nim with Addition is exactly like a normal game of Nim, except players are given an extra option. On each of their turns, instead of removing any number of coins from a pile, they may *add* any number of coins they want, as long as all the coins are added to a single pile. To prevent the game from going on infinitely, suppose players are only allowed to perform the 'adding' operation a finite number of times. Who is expected to win?

Theorem 3.11. Nim with Addition is equivalent to a game of normal Nim, with the same piles.

Proof. It turns out that this game is *exactly* the same as Nim! How so? Suppose it's Alvin's turn right now. If he's already in a win state in Nim, then he can simply perform the same moves that he would have if this were just a normal game of Nim, ignoring the addition altogether. If he's in a loss state in Nim, then we already know that only doing the normal 'taking away' options will lead to his loss. So, his only hope is to add stones to one of the piles. But, if he adds s stones to one of the piles, then Barbie can just spend her turn removing those s stones from the same pile. Effectively, Barbie has just undone Alvin's last action and he is back in the same losing position that he started in! □

So, it turns out that adding in the option to add stones to one of the piles doesn't change things at all. This result, although simple, will end up being deceptively important for one of the most well-known results in Combinatorial Game Theory later on.

Exercise 3.12. Suppose we had some checkerboard with r rows and c columns. Each row contains 2 checkers, one red and one black. Alvin controls the red checkers while Barbie controls the black checkers. On each of their turns, they select one of the checkers of their color and move it any number of spaces to the left or to the right; Alvin always goes first. A checker cannot be on the same square as another, and also cannot jump over it. The loser is the first one who cannot make a move on their turn.

Given the initial positions of each player's checkers, who will win, Alvin or Barbie?

Hint: Although this game seems partisan, and could potentially go on infinitely, it can actually be reduced to an impartial game that will definitely end if the winning player knows what they are doing!

3.4.2 Staircase Nim

Suppose we have n 'steps' on a staircase labeled 0 to $n-1$. Each of the steps has p_0, p_1, $\cdots$, p_{n-1} coins on it. On each of Alvin and Barbie's turns, they choose a step $i > 0$ and may move any number of coins from step i to step $i-1$; Alvin always goes first. The loser is the

first one who cannot make a valid move on their turn.

Given n and the number of coins on each step p_0, p_1, $\cdots$, p_{n-1}, who will win, Alvin or Barbie? I encourage you to play around with this a bit before proceeding to see the solution.

Let's investigate. The first obvious thing is that all the coins on step 0 are pretty much useless, so we can just ignore them. If there is only $n = 1$ pile, then Alvin just loses immediately. If there are $n = 2$ piles, then the game is still rather straightforward. Alvin wins if there is at least one coin in step 1. He just dumps all the coins from step 1 into step 0, then Barbie loses because she doesn't have any more moves.

What about when $n = 3$? This is where it gets interesting. Notice that Alvin has the same winning strategy; if there is at least one coin in step 1, then he moves all the coins from step 1 into step 0. But, Barbie still has some valid moves left—she can still move coins from step 2 to step 1. But, Alvin is guaranteed to always have a response to that; whenever Barbie moves coins from pile 2 to pile 1, Alvin will just move those same coins from pile 1 to pile 0. Barbie is thus guaranteed to run out of moves first before Alvin, no matter how many coins are in pile 2.

This gives us an interesting result. It seems that all of the coins in piles 0 and 2 are useless, and it doesn't matter how many coins there are in each of those piles. Now, what if $n = 4$? Note that we can transfer coins from pile 3 to pile 2. But, since we said that the number of coins in pile 2 doesn't affect the final outcome, this is actually the same as if we had just **deleted** those coins from pile 3 which we had transferred over. So, the even-numbered steps are useless, and for the odd steps, we have some mechanism for 'deleting' an arbitrary number of them from the step. This sounds a lot like Nim, doesn't it!

Theorem 3.13. Staircase Nim is equivalent to playing a game of normal Nim, using the odd-numbered steps as our piles and disregarding the even-numbered steps completely.

Proof. Suppose it is Alvin's turn. He focuses his attention only on the odd-numbered steps and treats it like a game of Nim. If he is in a win state, and the winning strategy dictates that he take away s stones from step i, then he 'deletes' s stones from step i in Staircase Nim by moving them into step $i - 1$. Now, the XORsum of all the odd-numbered steps is zero.

Now it is Barbie's turn. If she makes a move on an odd-numbered step, then she has unbalanced the Nim-sum of the odd-numbered steps, and Alvin can always respond to this by re-balancing them again, according to the winning strategy dictated by Nim. If she makes a move on an even-numbered step, say by moving some coins from step i to step $i-1$, then Alvin can always respond by moving the same number of coins from step $i-1$ to $i-2$, and Barbie is back where she started! Eventually, Barbie will be the one to run out of moves first.

On the other hand, if on Alvin's turn, the odd-numbered steps give a loss state, then Barbie can use the same strategy to shut *him* out of the game, and Alvin will run out of moves first. □

I included this specifically just to show how sneakily Nim can worm its way into various games. It's something you should be aware of for when you tackle these kinds of combinatoric game theory problems.

Exercise 3.14. Suppose that instead of being a linear staircase, it branches off at one point. So, after the ith step, there's a Left branch whose steps go up to an nth step, $i+1$, $i+2$, $\cdots$, n, and a Right branch whose steps go up to an mth step, $i+1$, $i+2$, $\cdots$, m.

The strategy for the linear staircase was to play a Nim game with the coins on the odd-numbered steps. Does the strategy still apply here with the branching staircase?

4 Sprague-Grundy Theorem

4.1 Nimbers

In this section, we will describe the very powerful theorem of Sprague and Grundy that allows us to magically also know the optimal solutions to many kinds of games, even if we haven't seen them yet!

Theorem 4.1 (Sprague-Grundy Theorem)**.** Any impartial game under normal play conditions is equivalent to a game of Nim.

This is why a seemingly random game like Nim is so important. Essentially, it states that just about every impartial game that satisfies our assumptions from earlier can be treated like a game of Nim. The implication for contests is that the magic $\mathcal{O}(n)$ solution that we have for solving Nim games can actually be applied to a much wider variety of games than it initially might seem.

This seems rather unintuitive—I don't know about you, but when I first learned about Nim, it didn't feel like it had some sort of all-encompassing generality that lets all other combinatorial games end up being reduced to it. But, there are some things that it has going for it that actually seem promising. In Nim, we can take away as many coins as we want from a single pile, which definitely seems powerful. Additionally, Nim has that fantastic magical formula for quickly composing several piles together into one big game. Being able to 'add' games together into one big super-game *without* drastically increasing the complexity of our solution is also incredibly powerful, and something we definitely want to capitalize on. And, it turns out both of these facts are the keys to why Nim is the archetypal combinatorial game.

Let's introduce a concrete example of a game that requires us to use the Sprague-Grundy theorem in order to solve it.

Example 4.2. Consider the game from Exercise 2.7, where Alvin and Barbie could only remove a perfect square number of coins from the table, and the last person to take a coin wins. However, what if we made it more Nim-like? Instead of there simply being coins on the table, suppose there are n piles, containing p_1, p_2, $\cdots$, p_n coins respectively. Alvin or Barbie may remove a perfect square number of coins at a time on each of their turns, **as long as all those coins come from the same pile**. The last one to take a coin is the winner; alternatively, the first one who cannot take a coin is the loser. Let's call this game Nim Square.

Given n and each of the p_i, who wins, Alvin or Barbie?

Okay, so we're mixing the limitations of one of the subtraction games with the multiple pile nature of Nim. Unfortunately, the game as it is, is definitely **not** Nim; the proof of the winning strategy for Nim sort of hinged on the fact that we could remove an arbitrary amount of coins per pile, except now we're limited on the number of coins we can take away, so that's definitely not going to work anymore. If the s that we need to take away to return to a balanced state happens to not be a perfect square, then we're straight out of luck.

For lack of better options, let's fall back on the Dynamic Programming concept of games that we established earlier. A game is a finite directed graph, where the nodes are the states and the directed edges are the moves that transition one state to another. Then, to find out if a state u is a win state or a loss state, we check all of the other states that it points to. If u points to at least one loss state, then u is a win state. On the contrary, if u does not point to any loss states, and only points to win states, the u is itself a loss state.

Recall, however, in Nim, that in order to be more robust, rather than simply being assigned a boolean win/loss value, each game state was assigned a full non-negative integer to describe it. The game state would be considered a win state if that number was nonzero, and a loss state if it *was* zero. We call this integer assigned to each game its **nimber**[4]; for a normal game of Nim, the nimber of each pile is simply the number of coins in that pile. Then, to get the Nim-sum which describes the composite game state, we take the XORsum of all of the piles' nimbers.

What the Sprague-Grundy theorem is suggesting is that **every** game state has a nimber associated with it, even if the game itself isn't Nim. Let u be some game state; then, let $G(u)$[5] be the nimber associated with that state. Again, $G(u) \neq 0$ when u is a win state, and $G(u) = 0$ when u is a loss state. We compute the Grundy numbers in the following manner.

Let u be some game state. If u is a terminal state, then $G(u) = 0$. Otherwise, let V be the set of all states that u points to. Then, we recursively define $G(u)$ as

$$G(u) = \max_{v \in V} G(v),$$

where mex is the **m**inimum **ex**cludant function. That means that $\text{mex}\{S\}$ represents the smallest nonnegative integer missing from the set S. Hence, it is the **minimum** value that was **excluded** from the set. For instance, $\text{mex}\{0, 1, 3, 4\} = 2$, and $\text{mex}\, 1, 2, 9 = 0$.

Exercise 4.3. Suppose you have an arbitrary array of n non-negative integers, possibly with duplicates. What is the time complexity of finding the mex of this array?

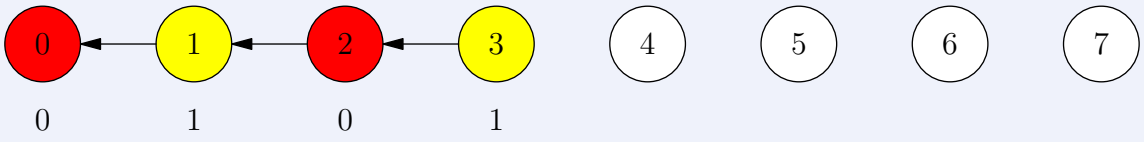
That's a lot to unpack, so let's take it one step at a time.

What we used to do in our old DP was that we would examine all of the states in V, and we simply checked if any of them were loss states.

Now, what we're doing instead is making a list of all the nimbers $G(v)$ that u points to. Then, we take the minimum excludant of that list, meaning, again, we search for the *smallest non-negative integer* that does **not** show up in that list. Then, $G(u)$ is the smallest non-negative integer that is **not** one of the nimbers that u points to.

That still might be kind of vague, so let's look at a concrete example.

Example 4.4. Let's look at game where you can only take a square number of coins, and focus on the version where there is still only one pile.

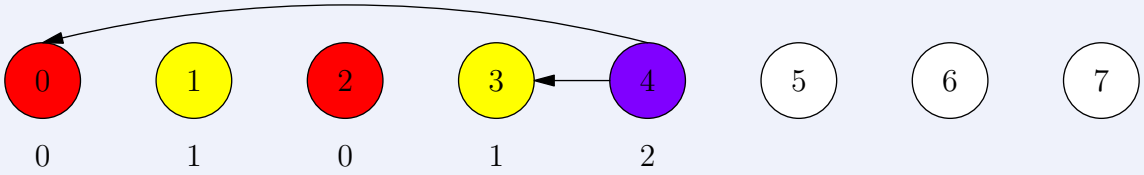


Since the game state here is completely captured by the number of coins in the pile, it is understood that $G(n)$ is the nimber of the state with n coins in it. Since $n = 0$ is a terminal state, we define $G(0) = 0$. Then, the state $n = 1$ only points to the state $n = 0$, so $G(1) = \text{mex}\{0\} = 1$, since the first non-negative integer not in the set $\{0\}$ is 1. The state $n = 2$ also only points to

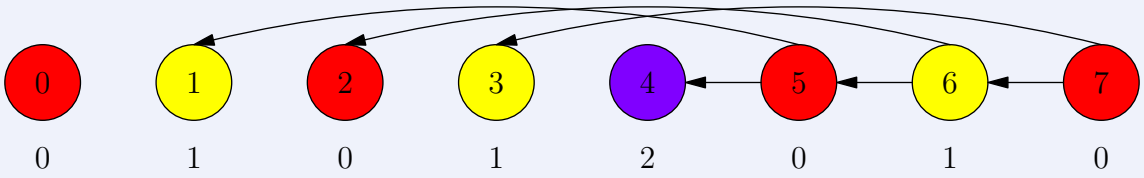
[4] You'll also see 'Grundy number' in the literature, but I think nimber is cute!

[5] The usage of 'G' as the function name comes from the fact that it's also called Grundy numbers, but come on, nimber is much cuter.

one state, $n = 1$, therefore $G(2) = \text{mex}\{1\} = 0$. Then, $n = 3$ also only points to one state, $n = 2$, and just like the case when $n = 1$, we have that $G(3) = \text{mex}\{0\} = 1$.



Now things get more interesting! When $n = 4$, we can choose to take 1 or 4 coins from the table, so it points to $n = 3$ and $n = 0$. Thus, $G(4) = \text{mex}\{G(0), G(3)\} = \text{mex}\{0, 1\} = 2$. Since both 0 and 1 are in the set, the next non-negative integer *not* in it is 2. So we already have some way to distinguish $n = 4$ from $n = 3$; although both are win states, they have different nimbers.



Let's fill up the last three states in our diagram.

$$G(5) = \text{mex}\{G(1), G(4)\} = \text{mex}\{1, 2\} = 0$$
$$G(6) = \text{mex}\{G(2), G(5)\} = \text{mex}\{0, 0\} = 1$$
$$G(7) = \text{mex}\{G(3), G(6)\} = \text{mex}\{1, 1\} = 0$$

Therefore the first 8 nimbers (starting at 0) are $0, 1, 0, 1, 2, 0, 1, 0$, respectively. If you compare it with the program you made earlier for this exercise, notice that both agree that $n = 0, 2, 5, 7$ are loss states.

Exercise 4.5. Find the nimbers for the first 10^5 values of n in this game. What is the runtime of your algorithm?

Exercise 4.6. Do all loss states have a nimber of 0?

What if there are multiple piles, though? Well, if there are multiple piles, then the resulting game state is a composite of multiple smaller states. The game is just a combination of several other games side-by-side, each one independent from the other. We say that the games are independent because a decision in one pile has no effect on any of the other piles; each pile is its own standalone thing. If in Nim, we took the Nim-sum of a composite game state by taking the XORsum of the piles, we will be using similar logic to get the Nim-sum of our nimbers.

Theorem 4.7. Suppose we have a composite game state consisting of n smaller independent states, s_1, s_2, $\cdots$, s_n. Then, the Nim-sum of the entire game state is equal to $G(s_1) \oplus G(s_2) \oplus \cdots G(s_n)$, where $\oplus$ is the bitwise XOR.

If the Nim-sum is nonzero, then that state is a win state. If the Nim-sum is equal to zero,

then that state is a loss state.

In other words, the game is basically just Nim! If there's one important thing that you need to memorize from this module, it should be Nim and the Sprague-Grundy theorem. The other techniques you can sort of derive on your own as an extension of DP and critical thinking, but these two are the magic results that are really hard to come up with on your own, yet still somewhat 'standard' in contests.

Example 4.8. Let's consider some game states in Square Nim. We know how to get the nimber of a single pile, so we can break down each game state into its piles then take the XOR of each of their nimbers.

- Suppose there are four piles whose sizes are $\{2, 3, 6, 7\}$. Then, the Nim-sum is $G(2) \oplus G(3) \oplus G(6) \oplus G(7) = 0 \oplus 1 \oplus 1 \oplus 0 = 0$, therefore this is a loss state.
- Suppose there are four piles whose sizes are $\{1, 3, 4, 6\}$. Then, the Nim-sum is $G(1) \oplus G(3) \oplus G(4) \oplus G(6) = 1 \oplus 1 \oplus 2 \oplus 1 = 3$, therefore this is a win state.

Exercise 4.9. Does Nim itself satisfy the fact that the nimber of each pile is the minimum excludant of all the states that pile points to?

So, our general game plan looks something like this.

- Take a game and break it down into its independent components, such as, say, a single pile. By independent, I mean that making a move in one component does not affect the state of the other components.
- Use the minimum excludant function to determine the nimbers for each of these simpler components, such as piles in our Nim games.
- Use the XOR logic from Nim to combine the information from each of the nimbers into the Nim-sum of the entire game.

It's rather math-y of an analogy, but compare it to how we deal with multiplicative functions. Usually, a multiplicative function f is far too complicated to evaluate at an arbitrary n. What we do instead is prime-factorize n into prime powers p^k (the individual piles). Then, we evaluate f at each of the prime powers p^k, which is usually much easier, similar to how it's usually much easier to get the minimum excludants and the nimbers of a single pile. Finally, we take all this information and compose it together. With a multiplicative function f, we multiply the results of the prime powers together, and with the game state, we XOR the results of each of the piles.

Proof. So, the question I'm sure you're thinking is **why**? **How**? What is this weird minimum excludant function and why does combining it with XOR somehow make these random other games work like Nim?

The key thing to note is that mex is a promise of sorts. Recall that the nimber $G(u)$ is the smallest nonnegative integer not being pointed to by u. That means that it promises that among all the states that u can transition to, every nimber from 0 to $G(u) - 1$ is represented. It doesn't say anything about the values greater than $G(u)$, but if we wanted to transition to any of the nimbers from o to $G(u) - 1$, we should be able to, since by definition, $G(u)$ as the mex is the *first* number we cannot transition to.

And if you think about it, isn't that sort of like what a pile in Nim is? If I have a pile with

n coins in it, that means that in one move, I can transition from n to any of the other pile-sizes from 0 to $n-1$.

So, you can imagine that playing a game that is a composite of states s_1, s_2, $\cdots$, s_n can pretty much be thought of as playing Nim with pile sizes $G(s_1)$, $G(s_2)$, $\cdots$, $G(s_n)$ in another universe. The two may have different names, but we can tell that they're essentially the same thing, because every action from one point of view can be translated to an analogous action in the other language.

Say that in Nim, I wanted to take away k stones from one of the piles. In the Nim world, pile i transitioned from p_i to $p_i - k$. In the actual game we are playing, this corresponds to the state s_i transitioning from a state that has nimber $G(s_i) = p_i$ to some other state that has nimber $p_i - k$. And, we are *promised* that this transition is always possible because $p_i - k < p_i$. Recall that nimber of s_i being $G(s_i)$ is a promise that if I wanted to transition to any nimber from 0 to $G(s_i) - 1$, I would always be able to find a state that s_i points to that would allow me to do so.

Ah, but there's a slight hiccup. Consider if in Square Nim, we had a pile of size 5, and we took away 1 coin. We transitioned from $G(5) = 0$ into $G(4) = 2$, so somehow the pile got bigger! The mex, after all, only promises that each number from 0 to $G(s_i) - 1$ is represented, but it has no promises about anything bigger than that. How does Nim account for that?

Well, recall one of the variants of Nim, that was *Nim with Addition*. Suppose someone made a move in the actual game that made the nimber go up from g to some larger h. Well, if this new state has a nimber of h, then it is possible to transition to any nimber from 0 to $h - 1$, again because of the minimum excludant function. Since $g < h$, it must definitely fall within that range, so there will always be a move that allows you to transition *back* to the nimber of g. Mind you, it definitely won't be the exact same state, but at least the nimber is back to what it was. So, just like in Nim with Addition, any addition operation on any of the 'piles' can always be undone.

So, since the actual game is just a reskinned version of Nim with Addition, it stands to reason that winning in Nim with Addition will also be the key to winning in our actual game. Thus, we can determine the winner or loser in any impartial combinatorial game by getting the XOR of each of the nimbers. Each of the nimbers really does quite directly correspond to some pile in a game of Nim (with Addition) that has that many coins in it. □

Example 4.10. Earlier, we said that in Square Nim, having four piles with sizes $\{1, 3, 4, 6\}$ was a win state because its Nim-sum was $1 \oplus 1 \oplus 2 \oplus 1$. Suppose Alvin is the current turn player. By assessing the game in terms of Nim, he realizes that to set the Nim-sum to 0, he should change the state with nimber 2 to a state with nimber 1.

So, he takes away 1 coin from the pile with 4 coins. Now there are $\{1, 3, 3, 6\}$ coins in each pile, which has a Nim-sum of 0, which is bad news for Barbie.

If Barbie takes 1 coin away from a pile with 3 stones, Alvin then responds by taking 1 coin away from the pile with 6 stones. The game state is now $\{1, 2, 3, 5\}$, whose Nim-sum is still $1 \oplus 0 \oplus 1 \oplus 0 = 0$.

Next Barbie takes 1 coin away from the pile with 5 stones, leaving 4 stones behind, and making the nimber go up from 0 to 2. Alvin then takes away 4 stones from that pile, leaving 0 stones behind, and making the nimber return from 2 to 0.

At this point, the two can only take away 1 stone at a time, and eventually enough, Alvin takes the last stone and Barbie loses. Unfortunately for her, if Alvin plays optimally, this is always the conclusion.

Exercise 4.11. Does the Sprague-Grundy still work with Misère games? Either prove or disprove it.

I want to stress the fact that the Sprague-Grundy theorem and XOR'ing the nimbers works with **any composition of several independent game states into one bigger game.** It doesn't have to be piles, although that is admittedly one of the more common types of game theory problems.

If the game ends up just being Nim, but with some weird condition on the number of stones you can take, then the bulk of the problem is finding the Grundy numbers for that single pile, which goes back to the pattern recognition tactics discussed earlier. Once you have a formula for the Grundy numbers, it is just a routine application of the Nim XOR formula.

Of course, that's assuming it's just asking who will win, and nothing spookier :D

4.2 Splitting a Game into Smaller Subgames

Recall Exercise 2.10. The game has n coins in a row, and in a move you can remove 1 or 2 *consecutive* coins from the row. If you drew the graph to represent this game, you'll know that it's certainly not as linear as the other examples in this module, because of the way that the row gets broken up into separate segments as more moves are made.

Moves that are not done at the ends will then end up splitting the row into two. Future moves cannot take coins from both rows at once, so a move that takes a coin in the middle actually splits the current game into two independent subgames. For instance, If I took the 3rd coin from a row of 10, we are now essentially playing **two** independent games, one with a row of 2 coins and one with a row of 7 coins. We started with just one game and ended up with more!

The directed graph will have an exponential number of vertices describing every possible way we can take coins from the row of n coins (Why?). Nim-games also had a plethora of states to consider, but our strategy to approaching those was to consider a much simpler directed graph, in the case where there was only one pile. Then, taking the XOR of the Grundy numbers of each pile is, in a way, sort of like 'adding' the graphs together.

Recall that in the Sprague-Grundy theorem, to get the nimber $G(u)$, we would take the mex over all possible nimbers that are possible results of moves from u. But, what if a move would split the game into several independent subgames? Well, the nimber of that entire move would be **the XOR of the nimbers of all the different subgames that the game would be split into**. This is us 'adding' all those subgames together to get the nimber associated with that collection of subgames as a whole.

Example 4.12. Suppose there are $n = 5$ coins in a row in the game from Exercise 2.10. Let $G(u)$ be the nimber of a row of u coins. You can verify that the nimbers for $n = 0$ to 4 are $0, 1, 2, 3, 1$, respectively.

We could take the 1st, 2nd, 3rd, 4th, or 5th coin. Taking the 1st coin splits it into a row of size 0 and a row of size 4, so we associate with this move a nimber of $G(0) \oplus G(4) = 0 \oplus 1 = 1$. Taking the 2nd coin splits it into a row of size 1 and a row of size 3, so we associate with this move a nimber of $G(1) \oplus G(3) = 1 \oplus 3 = 2$. Taking the 3rd coin splits it into a row of size 2 and another row of size 2, so we associate with this move a nimber of $G(2) \oplus G(2) = 0$. Taking the 4th or 5th coins are similar to taking the 2nd or 1st coins, respectively, due to symmetry.

We could also take the 1st and 2nd coin, which we associate with the nimber $G(0) \oplus G(3) = 3$,

or the 2nd and 3rd coin, which we associate with the nimber $G(1) \oplus G(2) = 3$. Again, the other two moves we could do are similar to these, due to symmetry.

Thus, we have that $G(5) = \text{mex}\{1, 2, 0, 3, 3\} = 4$.

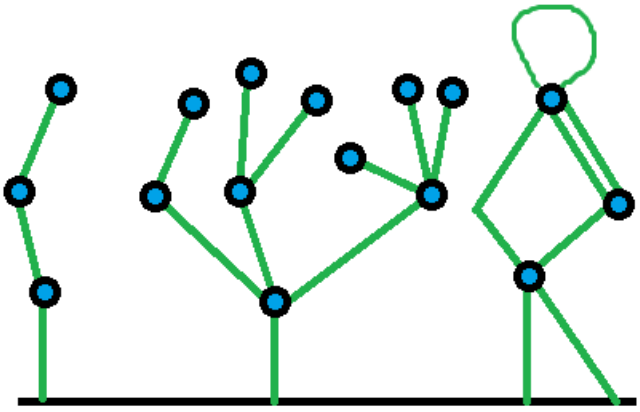
4.3 Hackenbush

Let's consider another famous combinatorial game, Hackenbush. We will specifically be focusing on the impartial version, usually called Green Hackenbush.

Picture a doodle that consists of a 'ground' line, and several pictures connected to the ground. The picture can be broken down into intersection points and segments connecting them together. Players take turns choosing a segment and erasing or 'cutting' it from the doodle, as well as erasing *any parts of the doodle that are no longer connected to the ground due to that segment getting cut off.* The winner is the last one who erases a segment; alternatively, the loser is the first one who cannot erase a segment on their turn.

This problem can be reduced to a formal graph problem. The intersection points are nodes, the segments between them are the edges, and the entire ground can be represented as a special 'root' node. Players take turns deleting edges from the graph, and when an edge is deleted, any vertices that are no longer in the same connected component as the root are also deleted.

Here's an example game I've drawn below. To capture the spirit of school whiteboard games, I've decided to render it in MS Paint.



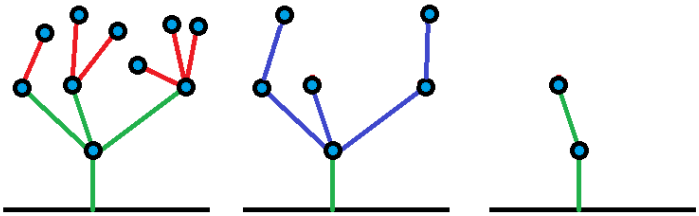
A full and more complete discussion of Hackenbush, including proofs, can be found here. For now, we will just look at the portions which are directly applicable to Nim and Sprague-Grundy.

Let's start with the simplest possible Hackenbush drawing, which is just linear chains of edges attached to the root. These are affectionately called 'bamboo stalks', and resemble the left-most item in the drawing. If the Hackenbush game merely consists of several bamboo stalks connected to the ground, then the game can be simply reduced to Nim.

Exercise 4.13. Reduce Hackenbush with only bamboo stalks into Nim.

How about examples like the middle drawing, though? What if our Hackenbush graph is a tree rooted in the ground? Then, cutting off an edge deletes a subtree in the graph.

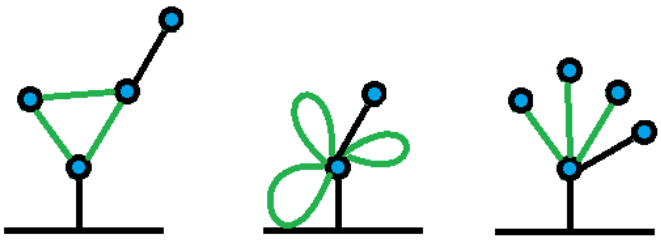
Now, we can use a principle commonly called the **Colon Principle**. For our purposes, the Colon Principle tells us that if several bamboo stalks are attached to a single node, then they can all be replaced by a single bamboo stalk whose length is the XOR of the lengths of all those stalks without changing the nimber of the game. We can use this to recursively reduce a tree into a single bamboo stalk. When we only have a collection of bamboo stalks, we can go back to applying the game of Nim. For instance, the middle picture in the game above can be reduced in the following manner,



The two stalks of length 1 can be paired off and deleted. The three stalks of length 1 can be replaced by a single stalk of length 1. Then, two stalks of length 2 can also be paired off and deleted. This finally leaves us with a single bamboo stalk of length 2. Repeat this strategy for every single tree in the doodle until you have a collection of bamboo stalks on which to apply Nim.

How about the last picture though, which is a general graph? There are cycles, parallel edges, and self loops. Well, for self-loop edges, they can be replaced with bamboo stalks of length 1 (due to the complete statement of the Colon Principle). Parallel edges can also be seen as cycles of length 2, since you can travel back and forth between the two connected nodes. So, it stands that we only need a way to deal with cycles.

The **Fusion Principle** says that any cycles can be condensed into a single node, with all the edges in the cycle being converted into self-loops attached to the condensed node, without changing the nimber of the graph.



Here we have a cycle of length 3. The entire cycle is condensed into a single node, with all the edges in the cycle converted into self loops. Then, we transform each of the self loops into bamboo stalks of length 1.

Therefore the general solution in impartial Hackenbush is to first use Fusion Principle to remove all cycles from the graph. Then, we repeatedly apply Colon Principle on the resulting trees, each time reducing the number of branches it has, unless we are left with a single bamboo stalk. If the final bamboo stalk consists of only the ground (length 0), then it is a loss state; otherwise, it is a win state. The example doodle given at the start of this section can be shown to be a loss state.

The full proofs for the Colon and Fusion Principles are included in the above link, as well as a discussion on the partisan version of the game, for those interested.

5 Minimax Games

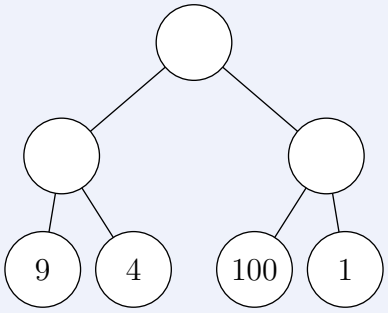
5.1 Games with Points

Up until now, the only possible outcomes of each game have been either a win or a loss. Even with the nimbers, they were only really an aid to determine a win or loss. Having a nimber of 100 is exactly the same win as a nimber of 1.

So, for this last section, let's consider games that actually have points! A common pattern in these games is to have one of the players be the **maximizer**, whose goal is to maximize some value, while the other is the **minimizer**, whose goal is to minimize that value, according to the rules of the game.

Example 5.1. A game can be represented as a branching decision tree. The leaves represent the terminal states of the game, and have some point value assigned to them. Imagine we have a game piece that initially rests on the root node.

Let's say goodbye to Alvin and Barbie and say hello to Maxi and Minnie. Maxi and Minnie take turns choosing one of the children of the current node and moving the game piece to that node. The game ends when the game piece is on a leaf, and the final score of the game is equal to the point value on that node. Maxi's goal is to **maximize** the final score, while Minnie's goal is to **minimize** the final score.



Consider the above decision tree. If Maxi goes first, what is the maximum value she can achieve? That 100 on the right seems very tempting for her, but if she goes to the right, Minnie is definitely going to move the piece to the 1 instead of the 100, which is Very Bad for Maxi. If she goes to the left though, Minnie's attempt to minimize the score will lead her to pick the 4 over the 9. So, Maxi is really choosing between a final score of 1 or 4, so she picks the bigger one, 4. The best final score she can get is 4.

If Minnie goes first, what then? For her, it is the 1 that seems tempting, but if she chooses to go to the right, Maxi will snatch that 100. If she goes to the left, though, then between 9 and 4, Maxi will maximize the score by picking 9. So, Minnie is really choosing between a final score of 9 or 100, so she picks the smaller one, 9. The best final score she can get is 9.

Note how the two cases are not equivalent.

You'll note that it's sort of a modification of the general DP solution that we had shown earlier. It roughly has the following pseudocode.

```
def minimax(u, is_maximizer):
    if u is terminal state:
```

```
        return value of u
    else:
        V = set of values of all states reachable from u
        if is_maximizer:
            return max(V)
        else:
            return min(V)
```

Maxi and Minnie will examine all their options and choose the largest or smallest one among them, respectively. What makes this bleed into game theory is that they have to 'play around' their opponent and take into consideration what they would pick. This interplay between Alvin picking the maximum of Barbie picking the minimum is why this is called the **minimax algorithm**.

Of course, this kind of exhaustive search is going to be really expensive, even with DP, so in practice, AI for complicated games only look a fixed number of moves into the future, then have some heuristic to assign point values to each state, where a higher point value is a more favorable state. For instance, a chess AI might consider how many pieces each player has, how powerful those remaining pieces are, who has control of the center, and a bunch of other factors.

But in competitive programming, we usually are concerned with problems which we can find the exact answers to, not just good enough answers. Let's then examine a family of minimax games that have some really interesting properties which we can study.

5.2 A Special Kind of Minimax Game

All the games in this section will be of the following form. Given an array, players take turns removing elements, until only one remains. One player maximizes this final number, while the other minimizes it. If both players play optimally, what's the final number?

Let's begin with an incredibly simple instance.

Example 5.2. The game begins with an array of n integers. Maxi and Minnie take turns choosing any element of the array and deleting it. The game ends when there is only a single integer left in the array. Maxi's goal is to maximize this integer, while Minnie's goal is to minimize the integer.

Given the array and assuming both are playing optimally, what is the maximum final value that Maxi can get if she goes first? What about the minimum final value that Minnie can get if she goes first?

Let's begin by assuming that Maxi always goes first. If we can answer this, then solving the case for when Minnie goes first is as simple as negating the entire array. Unfortunately for us, applying the generic minimax algorithm gives us a running time of $\mathcal{O}(n \cdot n!)$, which can be brought down to $\mathcal{O}(n \cdot 2^n)$ with DP, but that isn't really any much better, is it? But, you should probably already be able to tell just from how the problem looks that there must be some underlying structure that lets us solve it much, much faster.

Here is the key insight. Recall that we said that Maxi and Minnie are playing completely optimally, and since this game is deterministic and there are only a finite (albeit incredibly large) number of states to evaluate, there is already one exact foregone conclusion for what the result will be. So, let's assume that they're clairvoyant or something, and suppose that Maxi and Minnie already know that the answer is some `a[i]`.

We now reduce the game to an even simpler game: Maxi has some 'target' value `a[i]`. **Is**

it possible for Maxi to force the answer to be `a[i]` or greater[6]? If the answer is 'Yes', then we now know the answer in our original game is at least `a[i]`. If Maxi can force it to be at least that much, then we know she will.

If the answer to that question is 'No', then we seem to have a contradiction. How can the final answer be `a[i]` or greater if Maxi cannot force it to be so? Minnie certainly isn't going to help the answer be any larger than it needs to be. So, whatever the actual final answer is in our original game, it must be less than this impossible `a[i]`.

Now, notice the following. If it is possible for `a[i]` or greater to be an answer, then it is certainly also possible for a smaller number. If it is impossible for `a[i]` or greater to be an answer, then it is certainly also impossible for a larger number. With the right queries, we can hone in on an ever narrower set of possibilities until we know exactly what the answer is. These properties mean that we can binary search the answer!

You will see in Computer Science, in both competitive programming and outside it, that Binary Search is a powerful way to transform a decision "yes/no" no algorithm into an optimization solution, with minimal overhead. In our case, the final score that Maxi can get will be the greatest `a[i]` such that the answer to our above question is still 'yes'.

So, how do we answer this yes or no question? We can begin by transforming the original array into a binary array. We replace a number with `1` if it is greater than or equal to this candidate `a[i]`, and we replace a number with `0` if it is less than the candidate `a[i]`. For example, suppose we have the array $\{3, 1, 4, 1, 5, 9\}$, and we are testing to see if it is possible for Maxi to force the final answer to be 4 or greater. Thus, it transforms into the array $\{0, 0, 1, 0, 1, 1\}$. All the numbers greater than or equal to 4 are 'winning' final scores, because Maxi was successfully able to force the final answer to be at least 4. All the numbers less than 4 are the 'losing' final scores, since Maxi failed her goal. If the last number remaining is a 0, then Maxi 'lost' this game, and if the last number remaining is a 1, then Maxi 'won' this game.

Hey, so this goes back to our Win/Loss game from earlier! **Games with point values can just be reduced to games that are either win or lose**.

Now, back to the problem itself. Maxi's goal is to delete all the 0s in the list, while Minnie's goal is to delete all the 1s in the list. Maxi will always delete a 0 if she can, while Minnie will always delete a 1 if she can; if either cannot, then that means the array is all 0s or all 1s, in which case the answer would be obvious.

Now, notice that Minnie gets exactly $\lfloor \frac{n-1}{2} \rfloor$ moves. So, it stands to reason that if the number of 1s in the array is $\lfloor \frac{n-1}{2} \rfloor$ or less, then no matter what Maxi does, Minnie will be able to delete all of them simply by deleting a 1 on each of her turns, every chance she gets. On the flipside, if there are more than $\lfloor \frac{n-1}{2} \rfloor$ 1s in the array, then no matter what Minnie does, she won't be able to delete all of them. Maxi will delete all the 0s, and then win.

Therefore, Maxi wins the game if there are more than $\lfloor \frac{n-1}{2} \rfloor$ 1s in the transformed binary array, and if we recall which were the values that we set to 1, that means that there are more than $\lfloor \frac{n-1}{2} \rfloor$ elements that are greater than or equal to $a[i]$. Now we don't even need to do binary search; if we simply sort all the array's elements in order, then by their indices in the sorted array, we will be able to locate the largest integer with $\lfloor \frac{n-1}{2} \rfloor$ other integers greater than or equal to it. This will be the final answer.

Let's consider another variant of the game.

[6] We say "or greater" to cover the cases where Minnie does something stupid, but if she really is making the final answer as small as it can be, then the final answer will be `a[i]`

Example 5.3. The game begins with an array of n integers. Maxi and Minnie take turns choosing **any prefix or suffix** of the array and deleting all of those elements, as long as at least one element remains after this operation. By prefix, I mean some group of consecutive elements starting from the first element in the array, and by suffix, I mean some group of consecutive elements ending in the last element in the array. The game ends when there is only a single integer left in the array. Maxi's goal is to maximize this integer, while Minnie's goal is to minimize the integer.

Given the array and assuming both are playing optimally, what is the maximum final value that Maxi can get if she goes first? What about the minimum final value that Minnie can get if she goes first?

Let's apply a standard heuristic for when attacking a problem: looking at the special cases first. Suppose there is only one 1 in the transformed binary array, i.e. Maxi is aiming to get the absolute largest element in the array as the final score. Is this possible? Well, yes, if the largest element is either the first element or the last element. Maxi can simply delete every other element, and before Minnie even has time to respond, Maxi has already ended the game.

Okay, but what if the largest element *isn't* one of the outermost elements? What's Maxi going to do now? Try and play around with an array that only contains a single 1, while the rest are 0. You'll notice that whatever move you do, you're going to inescapably end up with a 0 on one of the endpoints. Why? Well, because we need two different 1s in order to have one on each endpoint! But, we only have 1. It's impossible for that single 1 to be on both endpoints, because that would mean an array of length 1, containing only a 1. But, since the lone 1 is in the middle, there's no way for us to isolate it.

So, after the first move, at least one of the endpoints will be a 0. That's bad, because Minnie can now easily isolate that 0. She will then delete everything except that 0, and now the game's over with the last remaining element being that 0.

Now, in the general case, suppose that for some goal `a[i]`, in Maxi's transformed binary array, either the first element is 1, the last element is 1, or both are. In this case, just like earlier, Maxi takes everything **except** the first or last element, and immediately ends the game with a 1 as the remaining element.

However, assume that both endpoints are 0 in the transformed binary array. No matter Maxi does, she cannot delete both endpoints, since she's not allowed to delete the entire array. So when Minnie's turn rolls around, at least one of the endpoints in the array handed to her will be a 0; then, she just deletes every other element except that 0, and wins.

So, it seems that the winner of the yes/no game is entirely dependent on the first element and the last element. If both are 0, then Minnie wins; otherwise, Maxi wins. Therefore, the final answer to the problem is the largest `a[i]` such that at least one of the endpoints is a 1, which turns out to be the bigger one of the two endpoints. So, rather anticlimactically, the optimal answer for Maxi is to always just pick the larger of the two endpoints, delete everything else, and end the game immediately, with that as the final score.

Let's tackle one last, more involved, example.

Example 5.4. The game begins with an array of n integers. Maxi and Minnie take turns choosing either the first or the last element of the array and deleting it. The game ends when there is only a single integer left in the array. Maxi's goal is to maximize this integer, while Minnie's goal is to minimize the integer.

Given the array and assuming both are playing optimally, what is the maximum final value

that Maxi can get if she goes first? What about the minimum final value that Minnie can get if she goes first?

Exercise 5.5. Can you see how this can be solved in $\mathcal{O}(n^2)$ using DP?

This turns out to be easier when we break it down into two cases. First, let's assume that n is even. If this is the case, then Minnie only gets to make $\frac{n}{2} - 1$ moves. I encourage you to play around as either role to get a feel for the dynamics of this game and how it works.

Minnie has a strategy which seems to be rather common in these kinds of Game Theory questions, so it's nice to familiarize yourself with it. Minnie, as the 2nd player, is simply going to *mirror* each of Maxi's moves, i.e. if Maxi takes the first element, then Minnie takes the last element in the array, and vice versa. What's so neat about this? Well, notice that no matter what Maxi does, Minnie's mirroring strategy will always end up deleting the first $\frac{n}{2} - 1$ and the last $\frac{n}{2} - 1$ elements of the array. Only the (1-indexed) $\frac{n}{2}$ and $\frac{n}{2} + 1$ th elements will manage to survive. Therefore, if both of these center elements are 0, Minnie is always able to win!

What about if at least one of the center elements is a 1? WLOG, let's assume that the first of the two (the $\frac{n}{2}$th element) is a 1. What Maxi can do is take the last element of the array. We are left with an odd-length array with the $\frac{n}{2}$th element perfectly in the center. Now, **Maxi** is the one to adopt the mirroring strategy. When Minnie takes the first element, Maxi takes the last, and vice versa. This allows Maxi to always be able to protect the center element, therefore Maxi can force the final remaining element to be a 1. If it is the $\frac{n}{2} + 1$ th element that is a 1, then Maxi does the same thing except she begins by removing the first element instead.

Thus, if n is even, we simply need to check the two center elements. If both of them are 0, then Minnie wins, otherwise Maxi wins.

What if n is odd though? Well, if the center element is 0, then, as we've established, Minnie is going to be able to protect that 0 by mirroring each of Maxi's moves. Therefore, if the center element is 0, then Minnie wins.

What if the center element is a 1 though? Whether Maxi takes the first or the last element, the remaining array will be even in length, so we can actually reuse or proof from earlier, except with Minnie going first this time. Recall that we said earlier that the case of Minnie going first is the same as the case of Maxi going first, except with all numbers replaced with their negative; verify that this means in the transformed array, this negating translates to us 'flipping' every value, i.e. 0s become 1s and 1s become 0s. Then, we simply apply our result for the even case and we are done. Maxi wins if and only if either the center and its left element or the center and its right element are both 1s. If the center is a 0, or if the center is a 1 sandwiched between two zeroes, then Minnie wins.

If you're having trouble thinking about that, we can more directly reuse our proof. Suppose the center of the odd-length array is 1, and the element directly to its left is a 1. Maxi then deletes the last element of the array, leaving us with an even-length array whose two center elements are both 1. Then, with the mirroring strategy, no matter what Minnie does, we will be left with one 1 remaining as the final answer. If the element directly to the center's right is a 1, then again, Maxi can do the exact same thing except she begins by deleting the last element first. However, we can show that if the center element is a 1, and the elements on either side of it are both 0s, then Minnie will always be able to force a win. Verify this part yourself; its logic is incredibly similar to the one employed earlier.

I hope working through that case was able to illustrate that it really was just reusing the logic from the even case, with an extra tweak. Isn't it neat how the two cases help each other out?

Therefore, if n is odd, we only need to check the three center elements. If the perfect center is a 1, or if the center three elements go `010`, then Minnie wins. Otherwise, Maxi wins.

Thus, to check whether a certain `a[i]` is a winner or not, we only need to inspect an array whose size is not more than 3. By iterating over all values of `a[i]`, we can find the greatest one which is still a winner, and this is going to be the optimal final answer.

Exercise 5.6. Code a simple interactive program for this game! Every other turn, you ask the human player whether they want to delete the first or last element, and you can print out the ever-shrinking array after each move the player and the computer make. Whether the computer is Maxi or Minnie, they should be able to play optimally.

Exercise 5.7. Consider the very first minimax game in this section. You are given a rooted tree with n vertices, and each of its leaves has a point value assigned to it. Maxi and Minnie take turns moving the game piece down from its current node to one of its children. The game ends when the piece is on a leaf node, and the final score of the game is the point value written on that that.

Given the array and assuming both are playing optimally, what is the maximum final value that Maxi can get if she goes first? What about the minimum final value that Minnie can get if she goes first?

6 Problems

Solve as many as you can! Ask me if anything is unclear.[7] In general, the harder problems will be worth more points, although I won't be saying which ones are harder.

I recommend that all of you at least **attempt** the items marked red. Especially to the newcomers, don't worry so much about solving every single one of them, but I think that there is something to gain from thinking about them a bit and giving them a good shot.

6.1 Warmup problems

These problems are straightforward applications of basic Game Theory concepts. They have barely any other frills and can be solved by direct application of the methods discussed above. I recommend practicing on these until the concepts are routine to you.

W1 A Chessboard Game: https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/a-chessboard-game-1

W2 Game of Stones: https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/game-of-stones-1

W3 Introduction to Nim Game: https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/nim-game-1

W4 Misère Nim: https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/misere-nim-1

W5 Poker Nim: https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/poker-nim-1

W6 John: UVa 1566

W7 Bachet's Game: UVa 10404

W8 Godsend: https://codeforces.com/problemset/problem/841/B

6.2 Non-coding problems

No need to be overly formal in your answers; as long as you're able to convince me, it's fine! First timers, solve at least [30★]. Veterans, solve at least [50★].

1. [3★] Exercise 2.14.
2. [5★] Exercise 4.11
3. Provide a proof for your solutions to the following problems in the Coding section.
   - [3★] Floor Division Game
   - [4★] Box Game
   - [8★] Playing with Stones

[7] Especially for ambiguities! Otherwise, you might risk getting fewer points even if you *technically* answered the question correctly.

4. The following is a classic game in Combinatorial Game Theory called **Chomp**. Alvin and Barbie have a chocolate bar divided into an $m \times n$ grid, and only the bottom-left-most square is poisoned. Alvin and Barbie take turns eating some of the chocolate in the following manner.

   If we label the bottom-left-most square as $(1, 1)$ and the top-right-most square as (m, n), ala Cartèsian coordinates, then on each of their turns, they choose some remaining uneaten square at (a, b), and eat all squares (x, y) where $x \geq a$ or $y \geq b$. Basically, they choose an uneaten square and eat it **and** all squares above or to the right of that square.

   Alvin always goes first. The loser is the one who has to eat the poisoned square.

   (a) [2★] In the special case when $m = n$ (the grid is a square), prove that eating $(2, 2)$ is always a winning strategy for Alvin.

   (b) [12★] One of the coding problems asks you to write a program that should be able to determine the winner, Alvin or Barbie, given the initial size of the chocolate bar m, n. Observe who the winner is for different values of m and n and try to spot a pattern in the data. Then, prove that this pattern holds.

   Hint: The expected proof uses a proof by contradiction and **does not actually construct the winning strategy**, unlike for Nim. Do not attempt to figure out what the actual winning move should be from a given game state.

5. Wythoff's Game is another combinatorial game, performed on a chessboard that extends infinitely up and to the right. We denote the bottom-left-most square as $(0, 0)$, and number the rows increasingly from left to right and the columns increasingly from bottom to top, ala Cartèsian coordinates.

   Suppose we begin with a Queen piece from chess placed on somewhere on the chessboard, (m, n). Alvin and Barbie take turns moving the Queen any number of steps south, west, or southwest (at a 45° angle); Alvin always goes first. The winner is the player who moves the Queen into the bottom-left corner $(0, 0)$.

   (a) [2★] Wythoff originally did not pose this question with a chessboard. He originally framed it in the following manner. Suppose you have two piles, one with m coins and one with n coins. The players take turns removing any number of coins from the first pile, the second pile, or both piles, but if they remove coins from both piles, they must remove the same amount from each pile. Is this equivalent to the game that uses the infinite chessboard?

   (b) [3★] Prove that every column and every row has exactly one position in that column/row such that if the Queen begins in that position, Barbie can force a win.

   (c) [7★] Prove your solution to the Wythoff's game coding exercise with bounds up to 10^7.

   It should be noted that there is an explicit formula for constructing the loss states, but the math involved in proving it is beyond the scope of this book. You may look it up if you are curious, but the above items can be done without it.

6. [8★] We will prove a generalization of Nim, which I'll call k-Nim. Suppose that there are n piles containing p_1, p_2, $\cdots$, p_n coins, respectively. Rather than being restricted to just one pile, the players may choose up to k piles and remove any number of coins from those piles, with possibly different amounts of coins taken from each pile. Note that our original Nim-game is just 1-Nim. We said that a given game state in Nim is a lose state if and only if the Nim-sum is equal to 0. For 1-Nim, the Nim-sum is simply the XORsum of the number of coins in each pile.

Recall that the XOR is considered 'addition without carrying' because what we do to compute it is write our integers in binary, then independently get the sum of each column, modulo 2. We get the k-XOR $\oplus_k$ of a set of integers by writing them out in binary, then independently getting the sum of each column, modulo $k+1$. In this case, our standard XOR is the 1-XOR.

For example, with the 2-XOR, $4 \oplus_2 6 \oplus_2 12 \oplus_2 14$ gives us

$$\begin{array}{rcccc} & 0 & 1 & 0 & 0 \\ & 0 & 1 & 1 & 0 \\ & 1 & 1 & 0 & 0 \\ \oplus_2 & 1 & 1 & 1 & 0 \\ \hline & 2 & 1 & 2 & 0 \end{array}$$

since the additions in each column are now done modulo $2+1=3$.

Prove that in k-Nim, a given game state is a lose state if and only if the k-XOR of all the piles has a 0 in the result of every column. For example, if there were $n=4$ piles with 4, 6, 12, and 14 coins, then it would be a loss state in 1-Nim since their normal XOR is 0, but it is a win state in 2-Nim since their 2-XOR is nonzero.

I suggest going back to the proof for Nim and trying to extend that to the generalized k-Nim.

7. [8★] What is the winning strategy in Misère k-Nim? Does the k-XOR still reveal who the winner is, just like it does in 1-Nim?

8. [7★] Let's play a game of Multistack Nim! Suppose we have a grid of cubbyholes with m columns and n rows, with $1 \le mn \le 10^6$. Label their columns from 1 to m, left to right, and their rows from 1 to n, bottom to top. Denote by (i, j) the cubbyhole that is found at the intersection of the ith column and the jth row, ala Cartèsian coordinates, so for example the bottom-left-most cubbyhole is $(1, 1)$ and the top-right-most cubbyhole is (m, n).

   The cubbyhole (i, j) initially contains $p_{i,j}$ coins, ranging from 0 to 10^9. Alvin and Barbie take turns removing coins from the cubbyholes, with the condition that

   - They may remove any number of coins from the **same** cubbyhole, but they must always remove at least one coin on their turn.
   - They may only remove coins from a cubbyhole if all cubbyholes above it in its column are empty. In other words, they may only remove coins from cubbyhole (i, j) if all (i, k) are already empty, for all $k > j$.

   Alvin always goes first. The one who takes the last coin is the winner; alternatively, the first one who cannot make a valid move is the loser.

   Given m, n, and the initial number of coins in each cubbyhole, describe and prove a method that will allow us to tell whether Alvin or Barbie wins the game.

9. The following problem is known as the 'Buckets of Fish' game, discussed by Joel David Hamkins. Suppose we have a row of n buckets in the sand, each initially containing some amount of fish (it is possible that some, or all, of the buckets are empty). Nearby is a virtually infinite supply of fish, freshly caught from the West Philippine Sea.

   Taking turns, each player selects a bucket and removes exactly one fish from it; then, if desired, they may add any finite number of fish from the nearby supply to the buckets to the left of the chosen one. The winner is whoever takes the very last fish from the buckets, leaving them all empty.

For instance, say Alvin and Barbie are playing a game with 3 buckets, containing 1, 3, and 5 fish, from left to right. Say Alvin goes first, then he could remove 1 fish from the 2nd bucket then place 3 fish in the leftmost bucket; now there are 4, 2, 5 fish in the buckets. Barbie could then remove a fish from the leftmost bucket, and since there are no buckets to the left of that, she ends her turn; now there are 3, 2, 5 fish in the buckets. Alvin could then, if he chooses, take a fish from the rightmost bucket, then place 1 fish in the leftmost bucket and 10^{10^6} fish in the middle bucket. This is a perfectly valid series of moves.

(a) [10★] Prove that no matter what the players do, the game is going to end within a finite number of moves. This might not seem inherently obvious since the players could place an absurd number of fish in the buckets. **Hint:** You might want to use induction.

(b) [7★] Prove that the second player can force a win if and only if at the start of the game, all the buckets contain an even number of fish.

6.3 Coding problems

First timers, solve at least [40★]. Veterans, solve at least [65★]. Assume that your programs should complete within 'the standard time limits', i.e. 1-3 seconds.

C1 [5★] Exercise 5.7. Suppose that there can be up to $n = 2 \cdot 10^5$ vertices.

C2 Consider Project Euler 301 (Nim), which, as is, has an upper bound of $n = 2^{30}$.

(a) [6★] Can you still solve the problem for $n = 2^{60}$?

(b) [8★] How about for $n = 2^{2^{60}}$? Give the answer modulo $10^9 + 7$ since it is incredibly large.

(c) [10★] How about finding the number of valid n in the range $[L, R]$, where $1 \leq L \leq R \leq 10^{18}$?

C3 [8★] (Adapted from a problem from Singapore ICPC 2018 Preliminaries) The game begins with n coins on the table. Alvin and Barbie take turns removing one less than a prime number of coins from the table—that is, each turn, they choose some prime p then remove $p - 1$ coins from the table ($p - 1$ cannot be more than the number of coins currently on the table). Alvin always goes first. The loser is the first person who cannot take a coin on their turn.

You will be given 10^5 values of n, where n is an integer ranging from 1 to 10^9. For each n, indicate whether Alvin or Barbie wins the game. **Hint:** This problem was given during a 5-hour ICPC-style contest.

For testing purposes, it is given that the first five positive values of n for which Barbie can force a win are 3, 8, 11, 32, and 35.

C4 Check the non-coding problems for the description of the game **Chomp**. Although it is actually rather easy to show whether Alvin or Barbie wins the game given the initial dimensions m, n, constructing the winning strategy is a much more difficult affair. So, unlike Nim, you probably won't be able to play this optimally against your friends, but a computer might!

Write an interactive program that lets you 'play' against the computer in a game of Chomp. The computer should be playing optimally, meaning that if it has a chance to win, it should. Even if the computer is in a lose state, it should still continue to play, and if the human player messes up at any point, it should be able to capitalize on the mistake and go for the win.

The bare bones I'm expecting are that

- Your program accepts as input the dimensions m, n and whether the player or computer goes first.
- It accepts input from the player on their turns to know which square the player wishes to chomp on.
- It outputs, on its turns, which square the computer decides to chomp on.
- After either the player or the computer makes a move, it prints out a simple ASCII-graphic grid of the remaining board state (just to make sure that you're actually processing the turns correctly internally). If you want to use flashier graphics than an ASCII grid or put an actual GUI, that's up to you.
- A simple message of "You win!" or "Computer wins!" or anything similar once the game is over.

Also, please include a short readme.txt so I know how to feed it input for the player interaction parts. Other than that, you can add whatever cute stuff or dialogue that you want. I won't give you any extra points for making a Javascript applet or an Android app but I will be very impressed.

(a) [5★] The program works for $1 \leq m, n \leq 4$.

(b) [7★] The program works for $1 \leq m, n \leq 10$.

C5 Check the non-coding problems for a description of Wythoff's game.

Write a program that, given an integer x as input, outputs the integer y such that if the Queen began at position (x, y), then Barbie would be able to force a win.

- [3★] The program works for $1 \leq x, y \leq 10^3$.
- [4★] The program works for $1 \leq x, y \leq 10^7$.

C6 Consider **Stone Game:** https://projecteuler.net/problem=260

- [3★] Any program that can find the correct answer.
- [4★] If your program uses $\mathcal{O}(n^2)$ space.
- [15★] If your program runs in faster than $\mathcal{O}(n^3)$ time.

C7 [2★] **Nim:** https://projecteuler.net/problem=301

C8 [10★] **Paper-strip Game:** https://projecteuler.net/problem=306

C9 [10★] **Nim Square:** https://projecteuler.net/problem=310

C10 [20★] **Hopping Game:** https://projecteuler.net/problem=391

C11 [12★] **Fibonacci Tree Game:** https://projecteuler.net/problem=400

C12 [15★] **Nim Extreme:** https://projecteuler.net/problem=409

First timers, solve at least [50★]. Veterans, solve at least [90★].

S1 [2★] **Exclusively Edible:** UVa 11311

S2 [2★] **Nimble Game:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/nimble-game-1

S3 [4★] **Game of Parity:** https://codeforces.com/problemset/problem/549/C

S4 [5★] **Floor Division Game:** https://www.codechef.com/problems/FDIVGAME

S5 [4★] **Stones:** UVa 12469

S6 [3★] **Chocolate in Box:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/chocolate-in-box

S7 [3★] **Industrial Nim:** https://codeforces.com/problemset/problem/15/C

S8 [4★] **Bob's Game:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/bobs-game

S9 [6★] **Box Game:** UVa 12293

S10 [10★] **Playing With Stones:** UVa 1482

S11 [11★] **Game of Stones:** https://codeforces.com/problemset/problem/768/E

S12 [8★] **Choosing Carrot:** https://codeforces.com/problemset/problem/794/E

S13 [8★] **Thanos Nim:** https://codeforces.com/problemset/problem/1147/E

S14 [7★] **Tokitsukaze, CSL and Stone Game:** https://codeforces.com/problemset/problem/1191/D

S15 [5★] **1-2-K Game:** https://codeforces.com/problemset/problem/1194/D

S16 [4★] **Tower Breakers, Again!:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/tower-breakers-again-1

S17 [5★] **The Prime Game:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/the-prime-game

S18 [5★] **Chessboard Game, Again!:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/chessboard-game-again-1

S19 [3★] **Deforestation:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/deforestation-1

S20 [8★] **A Game of Pawns:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/ngnl-3

S21 [12★] **Move the Coins:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/move-the-coins

S22 [5★] **Fun Game:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/fun-game-1

S23 [7★] **Powers Game:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/powers-game-1

S24 [10★] **Demiurges Play Again:** https://codeforces.com/problemset/problem/538/E

S25 [7★] **[2018 Algolympics] Cookie:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/2018-cookie

S26 [11★] **Dreamplay and the String Game:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/dreamplay-and-the-string-game

S27 [15★] **Spreadsheet Game:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/spreadsheet-game

S28 [18★] **Counting Games:** https://www.codechef.com/problems/GAMCOUNT

S29 [20★] **Tower Breakers, The Final Battle!:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/tower-breakers-the-final-battle-1

7 References

Helpful Reference: https://www.cs.cmu.edu/afs/cs/academic/class/15859-f01/www/notes/comb.pdf