# Game Theory

## Introduction to Game Theory

Game theory is the science that involves mathematical strategies to determine the winner at the end of a game. When we talk about game theory, we are generally referring to a **Combinatorial Game** ( i.e. a game that has an outcome and doesn't get stuck in an infinite loop of moves or turns ).

For example, in chess, there may come a stage where the two players are left with no choice other than to repeat their moves infinitely.

In combinatorial game theory, there are two types of games:

1. **Impartial Games:** In an impartial game, the two players can play the same kind of moves.
2. **Partial Games:** In a partial game there is a restriction as to what moves the two players can play. For example, in the game of chess one player can only move pieces of a single color. That is, if player 1 is playing for the black side then he/she can't move the white pieces.

We have taken the example of chess above and seen how it is an impartial game and also that there may be a situation in the game where it is also not a combinatorial one.

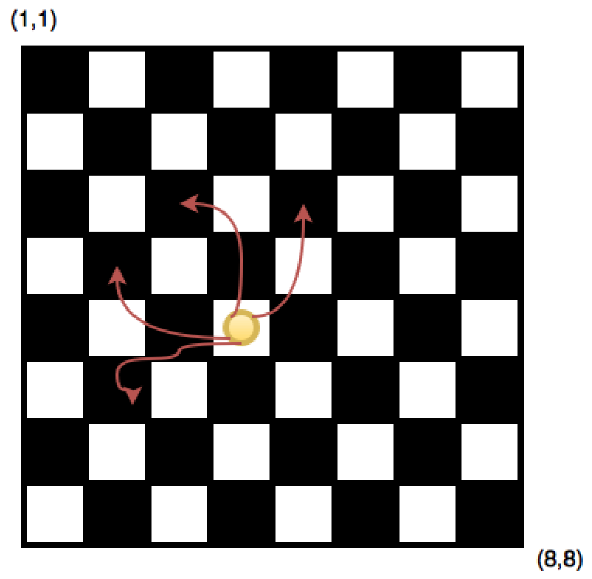
## Chessboard Game

**Problem Statement:** There is a chessboard of dimensions 15 X 15, and Alice and Bob start playing a game on it. The game is not chess and involves just one coin.

The coordinates of the top-left cell are (1,1) and the bottom right is (15,15). The coin is initially at (x,y). In each turn, the player whose turn it is can move the coin to any of the four cells (provided they are inside the board):

- (x-2,y+1)
- (x-2,y-1)
- (x+1,y-2)
- (x-1,y-2)

The figure below shows the possible moves of a coin at (5,4) (An 8 x 8 board is given in the image, but in the problem, it will always be a 15 x 15 board).



Alice makes the first move. Both Alice and Bob take alternate turns and move until it is not possible to move the coin anymore. The player unable to make a move loses.

**Approach:** From the starting point, we have at most four possible cells we could move to. If all 4 moves are not valid then the current state is losing. Also if any one of the cells out of 4 is losing then the current cell is winning. Now, we mark all the cells as winning or losing and then check the current state of the player and answer accordingly.

## Algorithm

- We will use the recursion in the following manner to solve the problem.
    - First of all we will construct a recursive function, let us say **f(x,y)** which will return as if the current cell **(x,y)** is winning or losing.
    - In the above function we will check if the value at the current cell has already been computed using the following condition **if(grid[x][y]!=-1)**. If this statement is **True** then we will simply return the previously computed value.
    - Otherwise, we will call the same function recursively on all the 4 possible coordinates where we can place the coin. **f(x-2,y-1), f(x-2,y-2), f(x-1,y-1), f(x-1,y-2).**
    - If atleast one of the four functions called above return **False** then we will mark the current grid **True** because then it will become the winning position for the second candidate
    - Otherwise, we mark the **grid[x][y]** False and return **grid[x][y]**.
- If **f(x,y)** is true
    - Return "**First**"
- Else
    - Return **"Second**"

## Code

```
#include<bits/stdc++.h>
using namespace std;
int grid[16][16];
/*
Possible Moves
      x-1, y-2
      x+1, y-2
      x-2, y-1
      x-2, y+1
*/

bool valid(int x, int y) {
      if(x<=0 or y<=0 or x>15 or y>15) {
            return false;
      }
      return true;
}

int f(int x, int y) {
      if(grid[x][y] != -1) return grid[x][y];

      else {
            bool isWin = false;
            if(valid(x-1, y-2)) {
                  if(f(x-1, y-2) == 0) isWin = true;
            }

            if(valid(x+1, y-2)) {
                  if(f(x+1, y-2) == 0) isWin = true;
            }

            if(valid(x-2, y-1)) {
                  if(f(x-2, y-1) == 0) isWin = true;
            }

            if(valid(x-2, y+1)){
                  if(f(x-2, y+1) == 0) isWin = true;
            }

            if(isWin) grid[x][y] = 1;
            else grid[x][y] = 0;
            return grid[x][y];
      }
}

int main() {
            int x, y;
            cin>>x>>y;
            for(int i=0; i<=15; i++) {
                  for(int j=0; j<=15; j++) {
                        grid[i][j] = -1;
                  }
            }

            grid[1][1] = grid[1][2] = grid[2][1] = grid[2][2] = 0;
            if(f(x, y) == 0) cout<<"Second\n";
            else cout<<"First\n";

      return 0;
}
```

### Time Complexity

**O($N^2$)**, where ‘N’ is the length of the chessboard.

Because there are **$N^2$** cells, and the function will be computed at most once, for each cell, the time complexity is **O($N^2$)**.

### Space Complexity

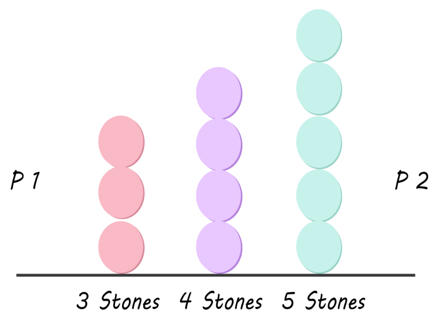
**O($N^2$),** where ‘N’ is the length of the chessboard.

Because there are **$N^2$** cells, and the function will be computed at most once, for each cell, the space complexity is **O($N^2$)**.

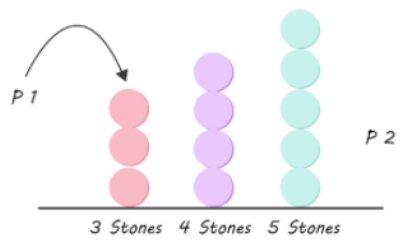
## Game of Nims

In this game, we are given numerous piles of stones and there are two players. The objective of the game is to remove a certain number of stones from one pile at a time in one move. The player may choose how many stones he/she wants to remove at their own convenience. The player who removes the last stone wins the game.

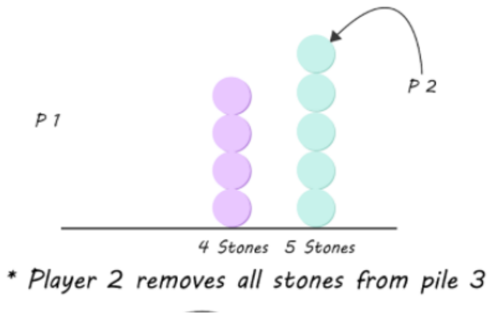
Let's take a look at the image given below for an example:



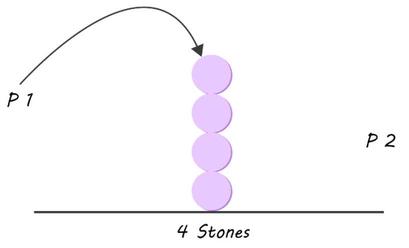
Now let's discuss one of the many possibilities in which player 1 wins the game.



* Player 1 removes all stones from pile 1

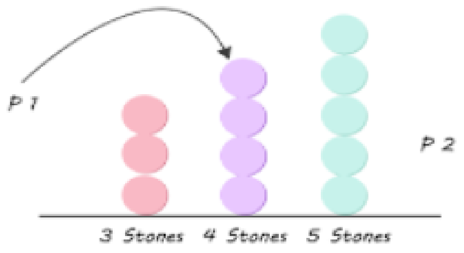


* Player 2 removes all stones from pile 3

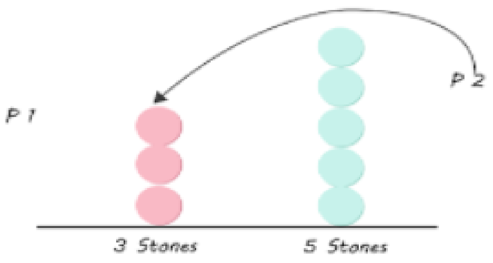


* Player 1 removes all stones from pile 2
& wins the game

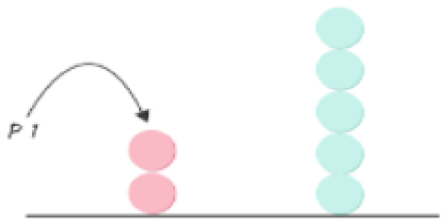
Now let's discuss one of the many possibilities in which **player 2 wins** the game.



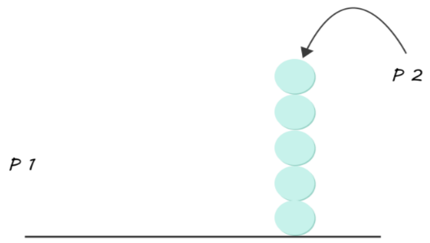
* Player 1 removes 4 stones from pile 2



* Player 2 removes 1 stones from pile 2



* Player 1 removes 2 stones from pile 1



* Player 2 removes all the stones from pile 3 & wins the game

## Nim Formula:

Game theory says that we can predict the outcome of the game even before it is played.

We have to calculate a **Nim Sum** which is equal to the **cumulative XOR of the stones in each pile at the initial stage.** If the value of Nim Sum is **zero** and player 1 and player 2 are both playing optimally, then player 1 always loses. If the Nim Sum is **non-zero** and player 1 and player 2 are both playing optimally then player 1 always wins.

In the above case Nim Sum = 3 ^ 4 ^ 5=2, which is a non-zero number.

## mex (Minimum Excludent)

Before jumping on to what grundy numbers are, let's discuss a small mathematical concept called **mex.**

The mex or Minimum Excludent Set of a subset of a well-ordered set is the smallest non-negative value from the whole set that does not belong to the subset.

For example, in a set of Non-Negative Integers, **mex { 0, 1, 3 } = 2** because **2** is the smallest non-negative integer that is not present in the subset **{0, 1, 3}**.

Similarly, **mex {1, 2, 3} = 0** because 0 is the smallest non-negative integer that is not present in the subset **{1, 2, 3}**.

## Grundy Numbers

Let us consider a state **N** of a two-player game and let **N-1** and **N-2** be the states reachable from N. We calculate the grundy value of **N** by

**Grundy (N) = mex {Grundy(N-1), Grundy(N-2)}**

Also, Grundy (0) is always 0.

Let us take an example of the game of nims in which we have **N** number of stones initially and we can reach states from **0** to **N-1**.

Then, **Grundy (N) = mex { Grundy (0), Grundy (1), Grundy (2) ...... Grundy (N-1) }**

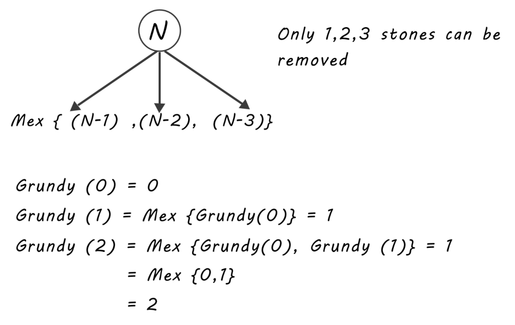
**Code**

**To find the Grundy Number**

```
#include<bits/stdc++.h>

using namespace std;

int mex(multiset<int> &vals){
	int MEX = 0;
	while(vals.find(MEX) != vals.end()){
		MEX++;
	}
	return MEX;
}

int main() {
	freopen("input.txt", "r", stdin);
	freopen("out.txt", "w", stdout);

	int n = 10;

	int grundy[n+1];

	multiset<int> vals;

	for(int i=0; i<=3; i++) {
		grundy[i] = i;
		vals.insert(i);
	}

	for(int i=4; i<=n; i++) {
		vals.erase(vals.find(grundy[i-4]));
		grundy[i] = mex(vals);
		vals.insert(grundy[i]);
	}

	for(int i=0; i<=n; i++) cout<<grundy[i]<<" ";

	return 0;
}
```

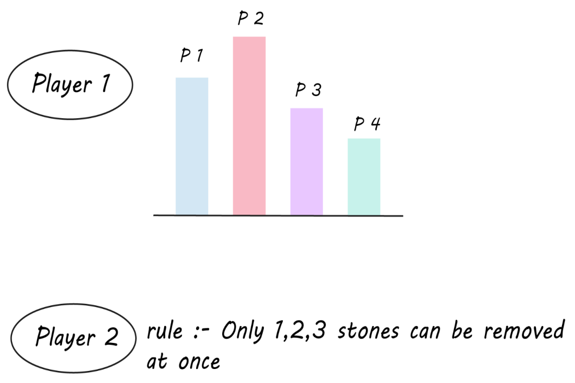
## Sprague Grundy Numbers

We've already looked at how the value of Grundy(N) is computed in the game of nim's, but what if the rules were changed so that a player may only remove 1, 2, or 3 stones in one turn? .



```
Grundy (0) = 0
Grundy (1) = Mex {Grundy(0)} = 1
Grundy (2) = Mex {Grundy(0), Grundy (1)} = 1
           = Mex {0,1}
           = 2
```

In such a game it would be very difficult to calculate the outcome of this game.



So Sprague- Grundy Theorem states that instead of taking the cumulative XOR of the piles of stones initially present, we take the cumulative XOR of the grundy value of the piles.

If **Cumulative XOR ( Grundy (p1), Grundy (p2), Grundy (p3), Grundy (p4) )** is Non-Zero, then Player 1 always wins. Else, Player 2 always wins.

**Code:**

**To find Sprague Grundy Number**

```
#include<bits/stdc++.h>

using namespace std;

const int N = 11;

int grundy[N];
multiset<int> vals;

int mex() {
	int MEX = 0;
	while(vals.find(MEX) != vals.end()) {
		MEX += 1;
	}
	return MEX;
}

int main() {
	freopen("input.txt", "r", stdin);
	freopen("out.txt", "w", stdout);

	int n;
	cin>>n;
	int stones[n];
	for(int i=0; i<n; i++) cin>>stones[i];

	for(int i=0; i<=3; i++) {
		grundy[i] = i;
		vals.insert(i);
	}

	for(int i=4; i<N; i++) {
		vals.erase(vals.find(grundy[i-4]));
		grundy[i] = mex();
		vals.insert(grundy[i]);
	}

	int ans = 0;
	for(int i=0; i<n; i++) {
		ans ^= grundy[stones[i]];
	}

	if(ans > 0) cout<<"Player 1";
	else cout<<"Player 2";

	return 0;
}
```

## MiniMax Algorithm

MiniMax is a kind of backtracking algorithm that is used in decision-making to find the optimal move for a player, assuming that your opponent also plays optimally.

In MiniMax there are two players, **Maximizer** (who wants to maximize his score in the current state) and **Minimizer** (who wants to maximize his score in the current state).

**Algorithm:**

Our goal is to determine the best course of action for the player. To accomplish so, we just select the node with the highest assessment score. We can also look ahead and analyze a prospective opponent's moves to make the process smarter.

We can look forward to as many movements as our processing capability allows for each move. The algorithm assumes the opponent is playing optimally.

From a technical standpoint, we begin with the root node and work our way up to the **best node possible**. The assessment scores of nodes are used to rank them. The **evaluation function** in our scenario can only award scores to outcome nodes (leaves). As a result, we reach leaves with scores in a recursive manner and propagate the scores backwards.

Starting with the **root node**, **Maximizer** selects the move with the highest score. Due to the fact that only leaves have evaluation scores, the algorithm must recursively approach leaf nodes. The **minimizer** is now choosing a move from the leaf nodes in the supplied game tree, thus the nodes with the lowest scores will be chosen. It continues to select the best nodes in the same manner until it reaches the root node.

Let's now formally define the **algorithm's steps**:

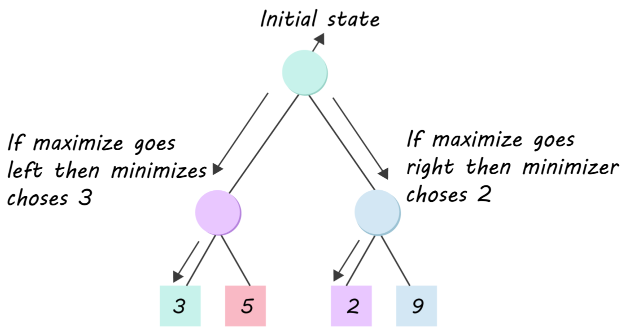
1. Construct the complete game tree
2. Using the evaluation function, evaluate the scores for the leaves.
3. Scores from the leaves to the roots, based on the player type:
    a. Select the youngster with the highest score for the maximum player.
    b. Choose the child with the lowest score as the minimum player.
4. At the root node, choose the node with the max value and perform the corresponding move.

Let’s take an example for more clarity.

**Example:**

Let’s consider a game that has 4 final states and you are the maximizer and you have the first chance.

Since this is a backtracking algorithm, we have to explore all the paths and determine which would work the best.


Initial state
If maximize goes left then minimizes choses 3
If maximize goes right then minimizer choses 2
3
5
2
9

Being the maximizer you would choose the larger value which is 3. Hence the optimal move for the maximizer is to go Left and the optimal value is 3.

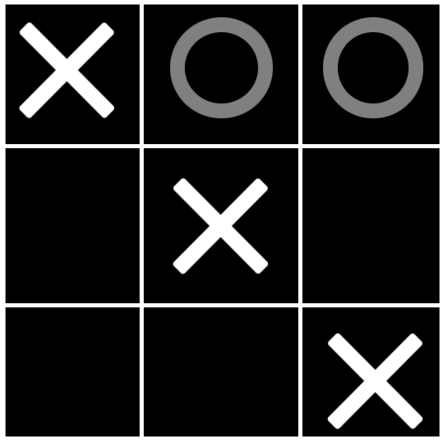
## Evaluation Function

The evaluation function is unique for every type of game. Evaluation functions are needed to determine the values for the given state of the game. Designing a good evaluation function is needed so that the minimax algorithm performs well. Let's design the evaluation function for the game Tic-Tac-Toe. The basic idea behind the evaluation function is to give a high value for a board if the maximizer's turn or a low value for the board if the minimizer's turn.

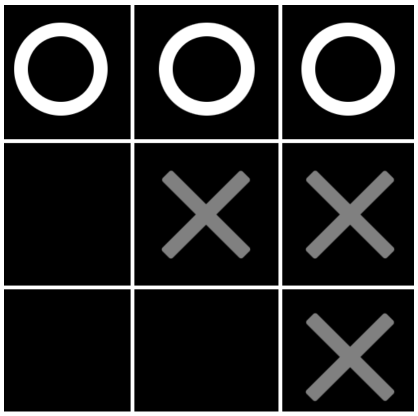
For this scenario let us consider X as the maximizer and O as the minimizer.

Let us build our evaluation function:

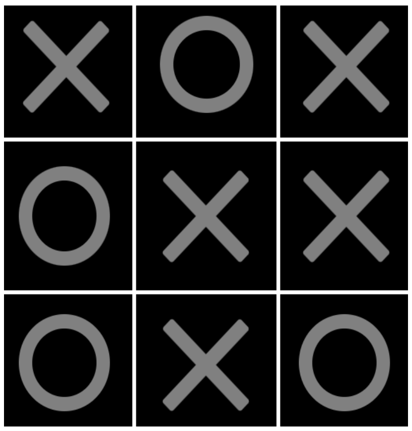
1. If X wins on the board we give it a positive value of +10.



2. If O wins on the board we give it a negative value of -10.



3. If no one has won or the game results in a draw then we give a value of +0.



Likewise, we can create the evaluation function for other games accordingly.

## Live Problem: Number Game

### Problem Statement

Ashishgup and FastestFinger play a game.

They start with a number n and play in turns. In each turn, a player can make **any one** of the following moves:

- Divide n by any of its odd divisors greater than 1.

- Subtract 1 from n if n is greater than 1.

Divisors of a number include the number itself.

The player who is **unable to make a move** loses the game.

Ashishgup moves first. Determine the winner of the game if both of them play optimally.

### Input

The first line contains a single integer t (1≤t≤100) — the number of test cases. The description of the test cases follows

The only line of each test case contains a single integer — n (1≤n≤109).

### Output

For each test case, print "Ashishgup" if he wins, and "FastestFinger" otherwise (without quotes).

**Approach:**

Let's analyse the problem for the following 3 cases:

- **Case 1: n is odd**

  Here Ashishgup can divide **n** by itself, since it is odd and hence **n/n=1**, and FastestFinger loses. Here **n=1** is an exception.

- **Case 2: n is even and has no odd divisors greater than 1**

  Here n is of the form **$2^x$**. As **n** has no odd divisors greater than **1**, Ashishgup is forced to subtract it by **1** making n odd. So if **x>1**, FastestFinger wins. For **x=1, n−1** is equal to **1**, so Ashishgup wins.

- **Case 3: n is even and has odd divisors**
  If **n** is divisible by **4** then Ashishgup can divide **n** by its largest odd factor after which **n** becomes of the form **$2^x$** where **x>1**, so Ashishgup wins.

Otherwise n must be of the form **2 · p**, where p is odd. If **p** is prime, Ashishgup loses since he can either reduce **n** by **1** or divide it by p both of which would be losing for him. If **p** is not prime then **p** must be of the form **p1 · p2** where **p1** is prime and **p2** is any odd number **>1**. Ashishgup can win by dividing **n** by **p2**.

## Code

```
#include<bits/stdc++.h>

using namespace std;

bool isPrime(int n) {
      for(int i=2; i*i<=n; i++) {
            if(n%i == 0) return false;
      }

      return true;
}

bool isFastestFinger(int n) {
      if(n == 1) return true;
      if(n == 2) return false;
      if(n % 2 != 0) return false;
      if((n&(n-1)) == 0) return true;
      if(n%4 != 0 and isPrime(n/2)) return true;
      return false;
}

int main() {
      freopen("input.txt", "r", stdin);
      freopen("out.txt", "w", stdout);
      int t;
      cin>>t;
      while(t--) {
            int n;
            cin>>n;

            if(isFastestFinger(n)) cout<<"FastestFinger\n";
            else cout<<"Ashishgup\n";
        }
        return 0;
}
```