P THINK FAST S

# Competitive Programming

From Problem 2 Solution in O(1)

# Combinatorial Game Theory

## Sprague – Grundy Theorem

**Mostafa Saad Ibrahim**
PhD Student @ Simon Fraser University

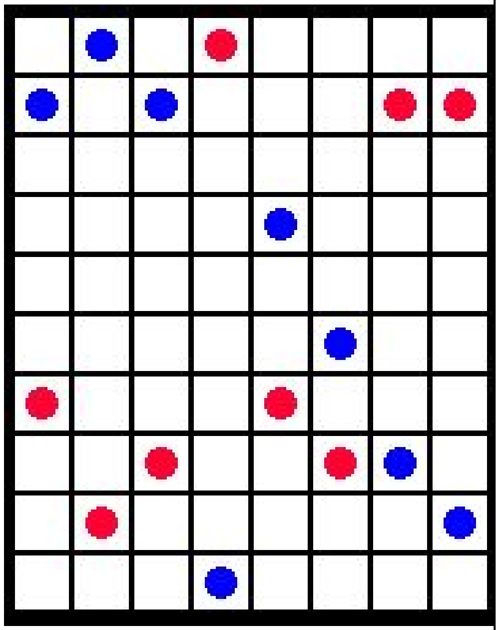
# Recall Nim game properties

- Impartial game
  - **Same set of moves** at any time allowed for both players
- 2 players play sequentially
- Each pile is independent sub-game
- Perfect information
- Finite game, No draws, No randomization
- Winner = last move
  - Loser = can’t make a move
- Such game => Sprague–Grundy theorem

# Recall Northcotts game

- Game Details
  - Board with red and blue opponents
  - Each column has 1 red and 1 blue ball
  - For a column, move only your color

    **toward** the other color (any # of steps)
  - BUT can't jump over opponent
  - Loser = Last to can't make a move
- Solution
  - Any **equivalence** to Nim game?
  - Convert each column to a pile size (# of cells between 2 colors). Then do xor for the pile sizes!

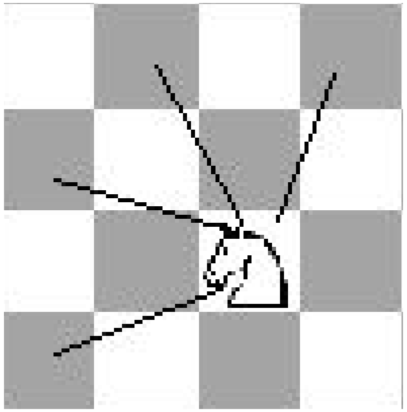


# Recall Northcotts game

- Equivalent game to Nim
  - Move is done to one specific column (= to 1 pile)
  - So our action is kind of N (for columns) **sub-games**
  - Moving toward your opponent minimize distance (= to remove stones from pile) => Finite steps (or poker trick)
  - All impartial game properties hold
- The core point
  - We managed to convert every sub-game to a single value
  - These values represents Nim-heaps.
  - Game logic was easy to find the sub-game=> value map
- What if the game is much more complex?

# Knights on chessboard game

- Given N x N chessboard with K knights on it
  - Knight Move is to 4 positions only
  - Player turn: Pick a knight and move it
  - *Allowed: Multiple knights in 1 position*
  - Loser: Player who can’t make a move to knight
- Equivalent game to Nim
  - K Nim piles but restricted stones to move
  - Knight move always makes sum of coordinates is decreased (= remove stones)
  - **But, how to compute Nim Heap size!**
    - **Sprague–Grundy theorem**



# Sprague–Grundy theorem

- Theorem
  - *Every impartial game is equivalent to a nim heap of a certain size (called Grundy number or Nimber)*
  - *So given such a game => convert to a unique number*
- Why important?
  - Convert any **sub-game** state to its grundy number
    - **Composite Game** = N independent sub-games
  - These numbers are equivalent to piles in Game of Nim
  - Just solve Normal Nim (e.g. xor piles sizes)
- Mex (Minimum excludant)
  - A simple function used to compute grundy numbers

# Composite Game

- Think in Nim
  - Say having N piles...player selects 1 pile to make move
  - Actually every pile is independent **sub-game**
  - **Composite Game = N <u>independent</u> sub-games**
- Northcotts game / Knights game
  - Every column/knight is sub-game
- Why important?
  - Solving a sub-game is easier than solve a complex one
  - E.g. find grundy of each knight, then xor values

# Mex (Minimum excludant)

- Mex of a set of numbers is the **smallest** x >= 0 is NOT in the set

$$\begin{aligned}
&\mathrm{mex}(\emptyset) = 0 \\
&\mathrm{mex}(\{1,2,3\}) = 0 \\
&\mathrm{mex}(\{0,2,4,6,\ldots\}) = 1 \\
&\mathrm{mex}(\{0,1,4,7,12\}) = 2 \\
&\mathrm{mex}(\{0,1,2,3,\ldots\}) = \omega \\
&\mathrm{mex}(\{0,1,2,3,\ldots,\omega\}) = \omega + 1
\end{aligned}$$

Src: http://www.geeksforgeeks.org/combinatorial-game-theory-set-3-grundy-numbersnimbers-and-mex/

# Mex (Minimum excludant)

```
int calcMex(unordered_set<int> hashtable) {
  int val = 0;

  while (hashtable.find(val) != hashtable.end())
    val++;
  return val;
}
```

# Grundy number

- Given a game that generates sub-games for every possible move
- The nimber of state G is equal to
  - 0 for Losing terminal position, and
  - mex{nimber(move1), nimber(move2).., nimber(moveN)}
  - Prove
- Note it is a recursive definition
  - For every possible move
    - Compute the grundy number for such move
  - Then get the mex(moves nimbers list)
  - Typically wrote as dp

# Grundy number for Nim variant

- Game: 1 pile of stones, Moves: Player takes 1, 2, 3
- *Grundy(0) = 0 => losing base case*
- Grundy(1) = mex(Grundy(0)) = mex({0}) = 1
- Grundy(2) = mex(Grundy(1), Grundy(0))
  = mex({1, 0}) = 2
  - From 2: take 1 stone (move to 1), take 2 stones(move to 0)
- Grundy(3) = mex({2, 1, 0}) = 3
- Grundy(4) = mex({3, 2, 1}) = 0
- Grundy(5) = mex({0, 3, 2}) = 1
  - 5 moves to sub-states {4, 3, 2}. Recall Grundy(4) = 0

# Grundy number for Nim variant

```
int grundy[100];  // initialize to -1

int calcGrundy(int n) {
  if (n == 0)
    return 0;

  int &ret = grundy[n];
  if (ret != -1)
    return ret;

  unordered_set<int> sub_nimbers;

  for (int i = 1; i <= 3; i++) if (n >= i)
      sub_nimbers.insert(calcGrundy(n - i));

  return ret = calcMex(sub_nimbers);
}
```

| N | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Grundy | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 |

One can observe that **answer** is: **grundy(n) = n%4**

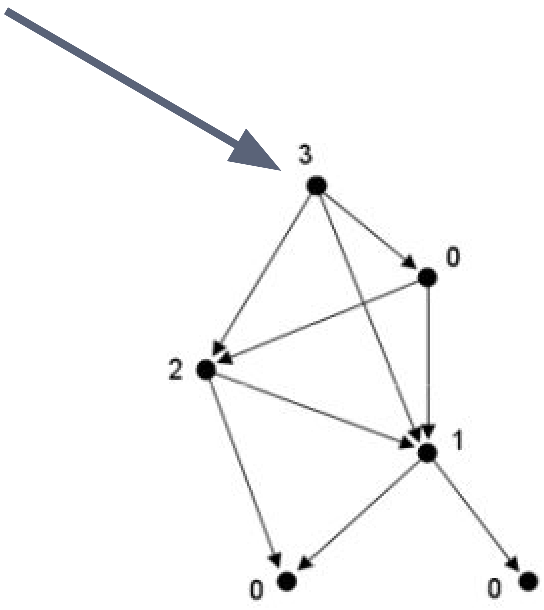
# Grundy number for simple game

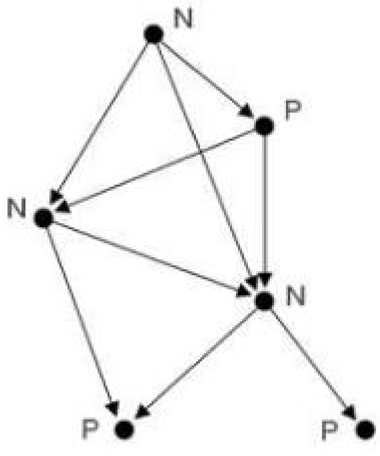
- Given number N.
  - Move is to divide by 2, 3 or 6 (and take floor)
  - Winner: Last to be able to divide (loser have N=0)

```
int calcGrundy(int n) {
  if (n == 0)
    return 0;

  int &ret = grundy[n];
  if (ret != -1)
    return ret;

  unordered_set<int> sub_nimbers;
  int moves[3] = { 2, 3, 6 };

  for (int i = 0; i < 3; i++)
    sub_nimbers.insert(calcGrundy1(n / moves[i]));

  return ret = calcMex(sub_nimbers);
}
```

# Grundy Graph of 1 game

Root (initial game setting)



# Grundy of several games

- Given N independent game
  - So every game has its own graph of sub-grundies
- Compute their grundies SG(game$_1$)
  - Generate moves. Recursively compute SG(sub-game)
  - Mex all sub-games
- Grundy of N games = xor their grundies
  - grundy(games sum) = SG(game$_0$) ^ SG(game$_1$) ….
  - Same handling as nim: grundy(games sum) != 0 => **win**

# Example: Knights on chessboard

- We can now write calcGrundy(x, y)
  - We have 4 possible positions as child move
- Given input some k pairs of (x, y)
  - Then just xor grundies of all knights positions

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 2 | 1 | 0 | 0 | 1 | 1 |
| 1 | 2 | 2 | 2 | 3 | 2 | 2 | 2 |
| 1 | 1 | 2 | 1 | 4 | 3 | 2 | 3 |
| 0 | 0 | 3 | 4 | 0 | 0 | 1 | 1 |
| 0 | 0 | 2 | 3 | 0 | 0 | 2 | 1 |
| 1 | 1 | 2 | 2 | 1 | 2 | 2 | 2 |
| 1 | 1 | 2 | 3 | 1 | 1 | 2 | 0 |

Src: https://www.topcoder.com/community/data-science/data-science-tutorials/algorithm-games/

# Example: Knights on chessboard

```
int grundy2[120][120];

bool valid(int v) {
  return v >= 0 && v < 8;
}

int calcGrundyChess(int r, int c) {
  int &ret = grundy2[r][c];
  if (ret != -1)
    return ret;

  unordered_set<int> sub_nimbers;

  const int DIR = 4;
  int dr[DIR] = { 1, -1, -2, -2 };
  int dc[DIR] = { -2, -2, 1, -1 };

  for (int d = 0; d < 4; ++d) {
    if (valid(r + dr[d]) && valid(c + dc[d]))
      sub_nimbers.insert(calcGrundyChess(r + dr[d], c + dc[d]));
  }
  return ret = calcMex(sub_nimbers);
}
```

# Example: Knights on chessboard

```
void chessMain() {
  clr(grundy2, -1);

  for (int i = 0; i < 8; ++i) {
    for (int j = 0; j < 8; ++j) {
      cout << calcGrundyChess(i, j) << " ";
    }
    cout << "\n";
  }

  int nimXor = 0, knights;

  cin>>knights;
  for (int d = 0; d < knights; ++d) {
    int x, y;
    cin>>x>>y;
    nimXor ^= calcGrundyChess(x, y);
  }
  if(nimXor != 0)    cout<<"First win";
  else               cout<<"Second win";
}
```

# Example: Sticks

- Example: Sticks (CEOI 2000)
  - Given N (<=10), and N values: Si <= 10
  - Given N rows (<=10), each row has numbers from 1 to $S_{row}$ <= 10 (e.g. list of list)
  - E.g. N = 3, S = {4, 8, 6}
  - So we have following:
  - 1 2 3 4
  - 1 2 3 4 5 6 7 8
  - 1 2 3 4 5 6

# Example: Sticks

- Example: Sticks (CEOI 2000)
  - In move: player pick 1 row, and remove up to 3 **consecutive** numbers
  - For example, if there is a row with 10 values
  - player1 removes [4, 5, 6] => remains [1, 2, 3, 7, 8, 9, 10]
  - Payer2 can remove [1, 2, 3] but NOT [3, 7, 8]
  - Loser: Can’t make a move
  - Who win? Simulate the game against computer?

# Example: Sticks

- Normal grundy, but efficiency concern
  - Inefficient: int grundy(vector<int> v)
- Let status be mask of 10 bits
  - E.g. 1110001111 represents [1, 2, 3, 7, 8, 9, 10]
  - For each mask, try all possible sub-masks of 1, 2, 3 consecutive bits on.
  - E.g. sub-mask 0000000110 represents [8, 9], 2 bits on
  - If submask exist, then it is valid move
  - do normal grundy
  - See iterative code here (first part only, 2nd simulate)
  - As small limits in stick => we used DP (**no game break**)

# Take-and-break game

- Recall divide n by 2, 3, 6 game
  - If N = 36, then we have 3 sub-states {36/2, 36/3, 36/6}
  - So any **single** move creates **one** sub-game only
  - SG(g) = mex{SC(g1), SC(g2), SC(g3)}
- What if a **single move** create more than **independent sub-games**?
  - Say move 1 creates *g1a, g1b, move2 creates g2 and move 3 creates g3a, g3b, g3c*
  - SG(g) = mex{**SG(g1a)^SG(g1b)**, SG(g2), **SG(g3a)^SG(g3b)^SG(g3c)**}
  - Apply the SG **xor** theorem on the independent sub-games

# Example: take-and-break game

- Given 2D chocolate bar, move can do
  - Remove last row or last column (and enjoy eating it)
  - Split it vertically or horizontally to 2 pieces
  - Any 1x1 pieces will be eaten
  - Loser: Last one with nothing to do

- Let Grundy(r, c) => grundy
  - Rule 1 => grundy(r-1, c) ⇒ remove last row
  - Rule 1 => grundy(r, c-1) ⇒ remove last row
  - Rule 2 can do rows-1 and cols-1 moves. But each moves generates **2 independent sub-games**
  - E.g. for col split at k: grundy(r, k) ^ grund(r, c-k)

# Grundy and losing/winning pos

- Assume a game state generates grundies
  - 2, 3, 7
  - It means we have 3 possible moves, each are winning
  - Then I am losing one
  - mex(2, 3, 7) = 0: So mex in this case works well
  - What if it generates: 0, 1, 2, 3, 7
  - It means 1 sub-game is losing and 4 are winning
  - mex(0, 1, 2, 3, 7) = 4 > 0: again works well
- Why xoring the final grundies works well?

  What proves that this number is a heap size!

# Grundy Mex and Heap size

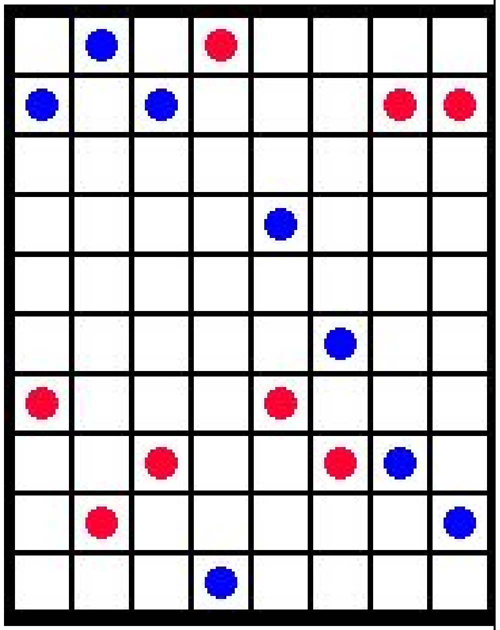
- Notes
  - Grundy size = a normal 1 pile nim heap size
    - E.g. if n = 5
    - Possible moves generates: 0, 1, 2, 3, 4
    - What if we allowed to add stones? Useless (poker)
  - Assume a state S generates grundies: {0, 1, 2, 3, 4, 7, 9}
  - Mex of {0, 1, 2, 3, 4, 7, 9} = 5 => why a nim size!
  - if 5 is a nim size => means we can generate substates 0-4
  - So finding first smallest number p, means [0, p-1] covered
  - What about possible values > p? Useless (poker)
    - I hope this is a correct intuition.

# Misère Grundy Number

- There is **NO** Sprague-Grundy Theorem for misère games
  - Grundy(terminal position) = 1 (NOT 0)
  - Some later winning state: Grundy({1}) = mex(1) = 0
  - So our overall values are 0 and 1
  - We can't represent nim in such limited values!
- So if a game is misère, don't think grundy
  - But still might be a nim game
  - Later in compound games, it might work

# Your turn: Northcotts game

- Game Details
  - One restriction on original game
  - Assume a move can be 1 or 2 steps only?
  - rows <= 20, columns <= 10^5



# Your turn: Too many stones Game

- Game
  - Example: Given k piles (< 50), each pile up to **$2^{90}$ stones**
    - In 1 move, user can take $2^m$ (for whatever m)
    - Loser: Can't make a move: Who is winner?
  - Hints:
    - Can you solve it if limits are much smaller?
    - Code it for small limits and see if there is **pattern**

# Your turn: Min # moves

- Doubloon Game problem:
  - Given 1 pile of size S coins (10^9)
  - Move: pick number that is $k^m$ (1 <= K <= 100)
  - Loser: Nothing to do
  - If first player will win, what is **the smallest number** of coins he may take?
  - Hints
    - Try to find the pattern for the # moves

# تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً