P THINK FAST S

Competitive Programming

From Problem 2 Solution in O(1)

Combinatorial Game Theory

Sprague – Grundy - Coin Turning Games

Mostafa Saad Ibrahim
PhD Student @ Simon Fraser University

Recall: Turning Turtles Game

- Given a horizontal line of N coins: Head/Tail
  - Move: Pick any head, and flip it to tail
  - Optionally, flip any coin on left of your chosen coin
- Solution
  - For every head at position k => pile of size k
  - Depdendent sub-games:
    - When the optional flipped coin goes from T to H, kind of dependency (e.h. TTHTTH**H** => **H**THTTHT)
    - We proved they are actually indpdendent
    - So HTTHH = HTTTT + TTTHT + TTTTH
    - That is, every H is independent sub-game

Coin Turning Game

- Coin Turning Games
  - Turning Turtles Game is one example for it
  - There are many variations, where Nim-analysis is hard
  - So grundy analysis makes things easier
    - But the nim analysis for the easy version is critical
  - Extending to **2D** variants create a **new theory** for nim-multiplication

Mock Turtles

- Variation of Turning Turtles Game
  - Now optionally turn up to 2 coins on your left
  - Assume N = 10^9
  - It is complex game now to prove nim equivalence
  - Better way, try to compute grundy value for the game
  - One might think, for every game, convert to mask
    - E.g.: HTTHH = 10011
    - Correct, but fits only up to small N (e.g. N = 20)
  - Recall, HTTHH is 3 independent subgames, each has 1 H
  - So compute grundy for a single position of head
    - Then game answer is xor of the H positions
  - Still N is so big? Try small N and find a **pattern**

Mock Turtles

- grundy(int pos)
  - Compute answer of single H (not whole input board)
  - We should try optionally flipping: 0, 1, 2 on our left
  - The tricky case, when flipping 2 positions
  - e.g. grundy(9) needs to flip 3 and 7
  - So a move created 2 independent sub-games
    - grundy(9) => grundy(3) ^ grundy(7) => insert for mex

Mock Turtles

```
// let a grid with no heads (TTTTT) has grundy = 0
// Compute grundy when 1 head at pos
int calcGrundyMockTurtle(int pos) {
  if (pos == 0)
    return 1;   // notice grundy(0) = 1

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

  unordered_set<int> sub_nimbers;

  // 1: flip 1 coin. Now state us TTTTT => grundy = 0
  sub_nimbers.insert(0);

  // 2: flip 2 coins: me and another.
    // e.g. TTTTTTH => TTTHTTT
  for (int i = 0; i < pos; ++i)
    sub_nimbers.insert(calcGrundyMockTurtle(i));

  // 3: flip 3 coins: me and other 2 coins (tricky)
     // e.g. TTTTTTH => THTHTTT
       // THTHTTT has 2 heads, 2 independent game from 1 a single move
  for (int i = 0; i < pos; ++i)  /// I turn another 2
    for (int j = i + 1; j < pos; ++j)
      sub_nimbers.insert(calcGrundyMockTurtle(i) ^ calcGrundyMockTurtle(j));

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

Mock Turtles

- Running the code, we get values:

position x :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$g(x)$:	1	2	4	7	8	11	13	14	16	19	21	22	25	26	28

- Seems g(x) = either 2x or 2x+1
  - These are called **odious** numbers
  - It depends on number of 1's in its binary values
  - Select the value that has odd 1s
  - E.g. g(4) = 8 (8 has 3 ones, 9 has 2 1s)
  - E.g. g(12) = 25 (25 has 3 ones, 24 has 2 1s)

Mock Turtles

```
void calcGrundyMockTurtle_mina() {
  clr(grundy, -1);

  for (int i = 0; i < 30; ++i) {
    int ans = calcGrundyMockTurtle(i);
    cout<<ans<<" ";
    int f = 1 - __builtin_popcount(i)%2;
    assert(ans == 2*i + f); // ith odious number
  }
  // g(x): 1 2 4 7 8 11 13 14 16 19 21 = odious sequence
  cout<<"\n";
}
```

Mock Turtles

- Your turn
  - **Prove that** if # of heads is odd => First always win
  - Let N be evil if # of binary 1s is even (e.g. 9, 24)
  - Investigate evil ^ evil, odious ^ odious, evil ^ odious
    - Is result evil or odious for above 3 cases?
    - Result of xoring odd # of odious numbers?
    - Recall: odious sequence elements > 0
  - **Prove that** if # of heads is even: result is the same as nim using the positions of heads
    - E.g. HHHH => nim piles = {0, 1, 2, 3}
  - Solution: See

Your turn: Ruler Turtles

- Variation of Turning Turtles Game
  - Now optionally turn up any number of coins but they must be consecutive to the original flipped H to T
  - TTTTH = {TTTTT, TTTHT, TTHHT, THHHT, HHHHT}
  - Your turn:
    - Code it, but use 1-based indexing (e.g. HTT = g(1))
    - Identify **the pattern function**
    - Solution in next slides
  - Output sequence: 1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16 1 2 1

Your turn: Ruler Turtles

```
// let a grid with no heads (TTTTT) has grundy = 0
// Compute grundy when 1 head at pos
// Assuming indexing from 1 (e.g. g(0) = 0)
// This is different handling when we assumed 0-indexing
int calcGrundyRulerTurtle(int pos) {
  if (pos == 0)
    return 0;   // notice grundy(0) = 1

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

  unordered_set<int> sub_nimbers;

  // 1: flip 1 coin. Now state us TTTTT => grundy = 0
  sub_nimbers.insert(0);

  // 2: flip any left, but consecutive
    // e.g. TTTTH = {TTTTT, TTTHT, TTHHT, THHHT, HHHHT}
  int xorVal = 0;
  for (int i = pos-1; i >= 0; --i)
  {
    // Each move create pos-i independent sub-games
    xorVal ^= calcGrundyRulerTurtle(i);
    sub_nimbers.insert(xorVal);
  }
  return ret = calcMex(sub_nimbers);
}
```

Your turn: Ruler Turtles

```
void calcGrundyRulerTurtle_main() {
  clr(grundy, -1);

  for (int i = 1; i < 30; ++i) {  // indexing from 1
    int ans = calcGrundyRulerTurtle(i);
    // g(x) is the largest power of 2 dividing x
    assert(ans == (i & -i) );
  }
  // g(x): 1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16 1 2 1
  cout<<"\n";
}
```

Your turn: Grunt Turtles

- Variation of Turning Turtles Game
  - In addition to your selected position H to T
  - Must select other 3 positions that makes the 4 positions are symmetric
    - One of these 3 positions is 0 position
  - E.g. one you selected n = 7
    - One group is: x--xx--x => {0, 3, 4, 7}
    - Notice symmetry of first 2 values to 2nd 2 values
  - Compare grundies with
    - 0 0 0 1 0 2 1 0 2 1 0 2 1 3 2 1 3 2 4 3 0
    - These are same grundies of [Grundy's game]
    - Don't compute pattern :) See link above

Your turn: Grunt Turtles

```
int calcGrundyGruntTurtle(int pos) {
  if (pos < 3)
    return 0;

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

  unordered_set<int> sub_nimbers;
  // E.g. 0, x, n - x, n for some 1 ≤ x < n/2
  // handle even/odd cases carefully
  for (int i = 1; i <= pos / 2; ++i)
    if (i != pos - i)
      sub_nimbers.insert(calcGrundyGruntTurtle(i) ^ calcGrundyGruntTurtle(pos - i));

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

Acrostic Twins

- 2D generalization of Turning Turtles Game
  - The grid of coins is 2D of head and tails
  - Move: Flip 2 coins
    - Pick a cell of H to flip to T
    - Flip another one (either above or left it)
  - Code is straightforward
    - F(x, y)
    - Try every row above it
    - Try every column before it
    - Each call creates one game with 1 flipped T to H

Acrostic Twins

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

int calcGrundyAcrosticTwins(int x, int y) {
  if (x == 0 && y == 0)
    return 0;

  int &ret = grundy2[x][y];
  if (ret != -1)
    return ret;

  unordered_set<int> sub_nimbers;

  for (int i = 0; i < y; i++)
    sub_nimbers.insert(calcGrundyAcrosticTwins(x, i));

  for (int i = 0; i < x; ++i)
    sub_nimbers.insert(calcGrundyAcrosticTwins(i, y));

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

Acrostic Twins (Nim addition)

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	0	3	2	5	4	7	6	9	8
2	2	3	0	1	6	7	4	5	10	11
3	3	2	1	0	7	6	5	4	11	10
4	4	5	6	7	0	1	2	3	12	13
5	5	4	7	6	1	0	3	2	13	12
6	6	7	4	5	2	3	0	1	14	15
7	7	6	5	4	3	2	1	0	15	14
8	8	9	10	11	12	13	14	15	0	1
9	9	8	11	10	13	12	15	14	1	0

Src: https://www.math.upenn.edu/~yuecheng/images/TomFergusonGametheory/GametheoryPartI.pdf

Acrostic Twins

- Analysis
  - Any connection between 1D case and 2D case?
  - 2D has the same 1D operations, once rows and other for columns
  - Intuition: may be the 2D answer is based on 1D answer
  - Recall 1D: pile(kth head) = k
  - Observation: F(x, y) = x^y (**Nim addition** (xor))
  - Game logic
    - We have 2 piles (N x M). Each time, we either reduce in 1st one (row) or reduce in 2nd one (column)
    - actually is is just normal nim game :)

Turning Corners

- 2D generalization of Turning Turtles Game
  - The grid of coins is 2D of head and tails
  - Move: Flip all coins in any rectangular block of coins
    - Pick a cell (x, y) of H to flip to T
    - Flip other 3 such that the 4 positions = rectangle
    - E.g. (a, b), (a, y), (x, b) and **(x, y)**,
    - where $0 \le a < x$ and $0 \le b < y$
  - Now, an H is flipped to T, and 3 T to H
    - So we have 3 recursive calls from a single move
    - xor the 3 calls first, before mex computation

Turning Corners

```
// aka nim multiplication
int calcGrundyTurningCorners(int x, int y) {  // O(N^4)
  if (x == 0 && y == 0)
    return 0;

  int &ret = grundy2[x][y];
  if (ret != -1)
    return ret;

  unordered_set<int> sub_nimbers;

  for (int a = 0; a < x; ++a)
    for (int b = 0; b < y; ++b)
      sub_nimbers.insert(calcGrundyTurningCorners(a, b) ^
                         calcGrundyTurningCorners(a, y) ^
                         calcGrundyTurningCorners(x, b));

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

Turning Corners (Nim multiplication)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	0	2	3	1	8	10	11	9	12	14	15	13	4	6	7	5
3	0	3	1	2	12	15	13	14	4	7	5	6	8	11	9	10
4	0	4	8	12	6	2	14	10	11	15	3	7	13	9	5	1
5	0	5	10	15	2	7	8	13	3	6	9	12	1	4	11	14
6	0	6	11	13	14	8	5	3	7	1	12	10	9	15	2	4
7	0	7	9	14	10	13	3	4	15	8	6	1	5	2	12	11
8	0	8	12	4	11	3	7	15	13	5	1	9	6	14	10	2
9	0	9	14	7	15	6	1	8	5	12	11	2	10	3	4	13
10	0	10	15	5	3	9	12	6	1	11	14	4	2	8	13	7
11	0	11	13	6	7	12	10	1	9	2	4	15	14	5	3	8
12	0	12	4	8	13	1	9	5	6	10	2	14	11	7	15	3
13	0	13	6	11	9	4	15	2	14	3	8	5	7	10	1	12
14	0	14	7	9	5	11	2	12	10	4	13	3	15	1	8	6
15	0	15	5	10	1	14	4	11	2	13	7	8	3	12	6	9

Src: https://www.math.upenn.edu/~yuecheng/images/TomFergusonGametheory/GametheoryPartI.pdf

Nim multiplication

- Previous table has multiplication properties
  - 0 acts like a zero for multiplication
    - $x \otimes 0 = 0 \otimes x = 0$ for all x
  - 1 acts like a unit for multiplication
    - $x \otimes 1 = 1 \otimes x = x$ for all x
  - **commutative** law obviously holds: $x \otimes y = y \otimes x$
  - **Associative** law holds $x \otimes (y \otimes z) = (x \otimes y) \otimes z$
  - Combined with nim addition, the **distributive** law holds
    - $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$
  - multiplicative inverse for non zeros: $6 \otimes 9 = 1$
  - We may consider as a **nim multiplication**
    - It can be computed efficintly by **Fermat 2-power**.

Recall: Twins game

- It was 1D variant
  - We must flip 2 coins, the rightmost must be H
  - Assume we have 2 1D twins games
  - Lets call them G1, G2
  - Let G1[x] = H and G2[y] = H
  - Let a < x, b < y, the 2nd coin to be flipped
  - E.g. in G1, make move at x, a, in G2, make move at y, b
- Let move in G = G1 x G2
  - G1 moves x G2 moves = all possible pairs of moves
  - E.g. (a, b), (a, y), (x, b) and **(x, y) = 2x2 moves**
  - Observe**: Turning Corners = Twins x Twins**

Games Multiplication

- Let
  - Given two 1D coin turning games, G1 and G
  - Define the tartan game G = G1 × G2, a 2D game
  - Let x = [x1 , x2, .. xm] is a legal move in G1
  - Let y = [y1 , y2, .. xn] is a legal move in G2
  - Let z = x * y is a legal move in G
    - positions (xi , yj) for all $1 \leq i \leq m$ and $1 \leq j \leq n$
    - Note: the **southeast** coin goes from heads to tails
- Note, such games are not intuitive
  - And nim multiplication too (for me)

Tartan Game Theorem

- If we know Turning Corners = Twins x Twins
  - Can we compute the 1D answer only of each game
  - And find the overall answer? Yes, **Nim Multiplication**
- The Tartan Theorem (proof)
  - Let x be move in G1 and y in G2
    - And their grundies, g1(x) and g2(y)
  - Let G = G1 x G2
  - Then grundy(x, y) = g1 (x) $\otimes$ g2 (y)
    - $\otimes$ = Nim multiplication
  - Such theorem can save computations or easier coding
    - Also finding the winning move easier

Rugs Game

- 2D generalization of Turning Turtles Game
  - The grid of coins is 2D of head and tails
  - Move: Flip **all** coins in **any rectangular block** of coins
    - Pick a cell (x, y) of H to flip to T
    - Find any rectangle with (x, y) in its **southeast**
  - Your turn: Code it (use 1-based)

Rugs Game

```
// Probably using table method will reduce order to O(N^4)
int calcGrundyRugsSlow(int x, int y) {  // 1-based
  if (x == 0 && y == 0)
    return 0;

  int &ret = grundy2[x][y];
  if (ret != -1)
    return ret;

  unordered_set<int> sub_nimbers;

  for (int a = 1; a <= x; ++a)
    for (int b = 1; b <= y; ++b) {
      int xorVal = 0;  // compute rect xor value
      for (int i = a; i <= x; ++i)
        for (int j = b; j <= y; ++j)
          if (i != x || j != y)
            xorVal ^= calcGrundyRugsSlow(i, j);

      sub_nimbers.insert(xorVal);
    }
  return ret = calcMex(sub_nimbers);
}
```

Rugs Game

- Observation: Rugs = Ruler x Ruler
  - Recall: Ruler is any consecutive row ending with H
    - Grundy =1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16 1 2 1
  - So each G1xG2 covers any rectangle based at H (x, y)
  - So compute 1D ruler
  - Compute Nim Multiplication
  - Rugs(x, y) = NimMultiplication(ruler(x), ruler(y))

Rugs Game

```
int solveTitanTheorem_rugs(int x, int y) {
  int grundy1 = calcGrundyRulerTurtle(x);
  int grundy2 = calcGrundyRulerTurtle(y);
  int grundy = calcGrundyTurningCorners(grundy1, grundy2);
  return grundy;
}
void calcGrundyRugsTheorem_main() {
  calcGrundyTurningCorners_main();  // Compute nim multiplication
  calcGrundyRulerTurtle_main(); // Compute 1D ruler
  int nimXor = 0, heads;
  cin>>heads;
  for (int d = 0; d < heads; ++d) {
    int x, y;
    cin>>x>>y;  // 1-based
    nimXor ^= solveTitanTheorem_rugs(x, y);
    // Recall, we xor among different grundies
    // We are doing Nim Additions over Nim Multiplications
  }
  if(nimXor != 0)   cout<<"First win";
  else              cout<<"Second win";
}
```

Rugs Game (Ruler x Ruler)

	1	2	1	4	1	2	1	8
1	1	2	1	4	1	2	1	8
2	2	3	2	8	2	3	2	12
1	1	2	1	4	1	2	1	8
4	4	8	4	6	4	8	4	11
1	1	2	1	4	1	2	1	8
2	2	3	2	8	2	3	2	12
1	1	2	1	4	1	2	1	8
8	8	12	8	11	8	12	8	13

Src: https://www.math.upenn.edu/~yuecheng/images/TomFergusonGametheory/GametheoryPartI.pdf

Finding a winning move

- Recall Mock Turtles
  - 1D game, turn H to T
  - Optionally turn up to 2 coins on your left
  - So overall turn 1, 2 or 3 coins
  - Recall its 1D sequence: 1, 2, 4, 7, 8, 11
- Defne Tartan Game = MockTur x MockTur
  - We can use tartan theorem to compute table trivially

Finding a winning move

T	H	T	T	T	T
T	T	T	T	T	T
T	T	T	T	T	T
T	T	T	T	T	T
T	T	T	T	T	H

	1	2	4	7	8	11
1	1	2	4	7	8	11
2	2	3	8	9	12	13
4	4	8	6	10	11	7
7	7	9	10	4	15	1
8	8	12	11	15	13	9

- Assume given grid on left, and its tartan grundies on left
- Are we on winning position? Yes. 2 ^ 9 = 11 (!= 0 ⇒ winning)
- Find 1 winning move?
- We need a move from 9 to make overall 2^2 = 0
    - E.g. move to 9 that creates 3 Heads and their **nim-sum** = 2
- How many available moves from H at (4, 5)
    - Using tartan style. 1D column has: 4C0 + 4C1 + 4C2 = 11
    - Same for 1D row: 5C0+5C1+5C2 = 16. Total moves: 11 x 16 = 176
    - Too much! Can tartan theorem help? Yes

Finding a winning move

- Here is an algorithm (based on proof)
  - suppose you are at position (x, y)
    - with SG-value $g1(x) \otimes g2(y) = v$
  - suppose you desire to replace v by grundy $u < v$
  - Let $v1 = g1(x)$ and $v2 = g2(y)$.
  - Find a move in **Turning Corners** that takes (v1, v2) into an SG-value u.
    - Denote the northwest corner of the move by(u1, u2)
    - Satisfy $(u1 \otimes u2) \oplus (u1 \otimes v2) \oplus (v1 \otimes u2) = u$.
    - This step can be $O(n^2)$ (or better order)

Finding a winning move

- Here is an algorithm (cont)
  - Find a move M1 in G1 from x to grundy(M1) = u1
    - E.g. $O(n^2)$ in Mock Turtles (i.e. 5C0+5C1+5C2)
  - Find a move M2 in G2 from y to grundy(M2) = u2
  - Then the move M1 × M2 in g1 × g2 moves to an SG-value u as desired
    - Notice, M1xM2 will contains (x, y)
    - So 1 (x, y) flips from H to T
    - Other coins flip from current to anti

Finding a winning move

```
// Find a move from (v1, v2) that has grundy target_u > 0
  // return only its top-left corner
// Fixed in our method
// We may implement this method in more efficient ways for queries
  // (u1⊕u2)⊕(u1⊕v2)⊕(v1⊕u2)=u.   // Add v1⊕v2 to both sides
  // (u1⊕u2)⊕(u1⊕v2)⊕(v1⊕u2)⊕(v1⊕v2)=u⊕(v1⊕v2)
  // (u1⊕v1)⊕(u2⊕v2)=u⊕(v1⊕v2) => Decoupled to 2 1d processing
  // See: http://www.stat.berkeley.edu/~mlugo/stat155-f11/tartan2.pdf
pair<int, int> findTurningCornersMove(int v1, int v2, int target_u) {
  for (int u1 = 0; u1 < v1; ++u1)
    for (int u2 = 0; u2 < v2; ++u2) {
      int grundy =  calcGrundyTurningCorners(u1, u2) ^
                    calcGrundyTurningCorners(u1, v2) ^
                    calcGrundyTurningCorners(v1, u2);
      if (grundy == target_u)
        return {u1, u2};
    }
  return {-1, -1};  // no solution
}
```

Finding a winning move

```
// Find a move from pos that has grundy target_u
// From problem to another, write yours
vector<int> findMockTurtleMove(int pos, int target_u) {
  vector<int> ret = {pos};
  for (int i = 0; i < pos; ++i) {
    int grundy = calcGrundyMockTurtle(i);
    if (grundy == target_u)
    {
      ret.push_back(i);
      return ret;
    }
  }

  for (int i = 0; i < pos; ++i)
    for (int j = i + 1; j < pos; ++j) {
      int grundy =  calcGrundyMockTurtle(i) ^
                    calcGrundyMockTurtle(j);
      if (grundy == target_u)
      {
        ret.push_back(i);
        ret.push_back(j);
        return ret;
      }
    }
  return ret;
}
```

Finding a winning move

```
vector< pair<int, int> > findMockTurtleSquaredMove(int x, int y, int u) {
  int v1 = calcGrundyMockTurtle(x);
  int v2 = calcGrundyMockTurtle(y);
  pair<int, int> p = findTurningCornersMove(v1, v2, u);

  if(p.first < 0 || p.second < 0)
    return {};

  vector<int> m1 = findMockTurtleMove(x, p.first);
  vector<int> m2 = findMockTurtleMove(y, p.second);

  vector< pair<int, int> > moves;
  int computed_u = 0;

  for(auto xx : m1) for(auto yy : m2) // move multiplication
  {
    moves.push_back({xx, yy});
    if(xx == x && yy == y)
      continue;
    computed_u ^= solveTitanTheoremMockTurtlesSquared(xx, yy);
  }
  assert(u == computed_u);
  return moves;
}
```

Finding a winning move

```
void calcGrundyMockTurtleSquaredTheorem_main() {
// Compute nim multiplication
  calcGrundyTurningCorners_main();
// Compute 1D Mock Turtle
  calcGrundyMockTurtle_main();

// Now solve whole input given H's
  int nimXor = 0, heads;

  cin >> heads;
  vector< pair<int, int> > inputPos;
  for (int d = 0; d < heads; ++d) {
    int x, y;
    cin >> x >> y;  // 0-based
    nimXor ^= solveTitanTheoremMockTurtlesSquared(x, y);
    inputPos.push_back({x, y});
  }
```

Finding a winning move

```
if (nimXor != 0)
{
  // Based on game, the closest H to (0, 0) won't have moves
  // Let's randomize as a general handling (hopefully faster)
  random_shuffle(inputPos.begin(), inputPos.end());
  bool foundMove = false;
  for (int d = 0; d < heads; ++d) {
    int x = inputPos[d].first, y = inputPos[d].second;
    int curg = solveTitanTheoremMockTurtlesSquared(x, y);
    vector< pair<int, int> > moves =
        findMockTurtleSquaredMove(x, y, nimXor ^ curg);

    if(moves.size() > 0) {
      foundMove = true;
      cout << "\n\n\nFirst win\n";
      for(auto p : moves)
        cout<<"Flip coin at "<<p.first<<", "<<p.second<<"\n";
      break;
    }
  }
  assert(foundMove);
}
else
  cout << "Second win";
```

Your turn

- Let G = G1 x G2
  - G1 to flip exactly 2 coins (most right is H), and **distance** between the 2 coins <= 4
    - E.g. Flip H at 10 and any of {9, 8, 7, 6, 5}
    - Do you notice correspondence to a nim variant?
    - its grundy in 1-based: $g1(x) = (x-1) \% 5$
  - G2: Ruler
  - Initially we have heads at (100, 100) and at (4, 1).
  - coin (100, 100) has value $4 \otimes 4 = 6$
  - coin at (4, 1) has value $(3 \otimes 1) = 3$
  - Validate this winning move:
    - $G1 = \{98, 100\} \times G2 = \{97, 98, 99, 100\}$

Games indexing

- Through the different examples we used different indexing (0 vs 1 based indexing)
  - E.g. Sometimes 1-based shows the pattern easily
  - May be coding is easier
  - If you are asked a new problem, try both and see
- When constructing Tartan Game, recall used indexing for each game
  - E.g. Let G = (Mock Turtles) x Ruler
  - Mock Turtles is 0-based
  - Ruler is 1-based

Solving impartial game

- If not impartial game (use search technique)
- Otherwise
  - Is it reasonable search space? E.g. do it with search
  - Is it a Nim game? Nim variant? Reduction?
  - Identify useless information (cancellation strategy, xor nature in cancelling equal piles, ..)
  - Think in concrete examples and come up with strategy
  - May use win/lose positions properties to prove solution
  - You may identify a pattern
  - If sub-games looks dependent, decouple them
  - Let grundy computation be your friend

Other Readings

- Winning Ways for Your Mathematical Plays
  - Major book in the field to read (I think vol2)
- Other Books: See1, See2, See3
  - See more coin turn games in see1
- Articles: See1, See2, See3, See4, See5
- Minimax / alpha beta: See1, See2

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً