Olympiads in Informatics, 2016, Vol. 10, 231–235

DOI: 10.15388/ioi.2016.15

Problem Solving, Presenting, and Programming: A Matter of Giving and Taking

Tom VERHOEFF
Dept. of Math. and CS, Eindhoven University of Technology
Eindhoven, The Netherlands
e-mail: t.verhoeff@tue.nl

Abstract. Nim is a well-known two-person game, where players alternate *taking* items from one of multiple piles. Finding a winning strategy for such games is a nice exercise in *problem solving*. Typically, the winning strategy for classical nim is explained in terms of nim sums, involving binary notation of numbers. I explain how to understand and play a winning strategy without prior knowledge of binary notation, which is useful when *presenting* this strategy in primary school. *Programming* that strategy is also an interesting challenge. This can be done elegantly in a functional language that supports patterns, such as the Wolfram Language. I conclude by *giving* you a variant of nim to work out yourself.

Keywords: impartial games, nim, functional programming.

1. Introduction of Problems to Solve

I teach enrichment classes in a primary school. The focus is on problem solving. One of the themes is puzzles and games that allow a mathematical analysis. Sometimes we work on 'exotic' (less known) puzzles and games, such as 'the princess on a graph', Perfect (2011), and others, Dore *et al.* (2010). But of course it is also important to address the classics.

One of these all-time classics is the following two-person game, known as *misère nim*[1], played as follows:

- Start with several piles of items (go stones, pennies, toothpicks, etc.).
- Players alternate turns.
- At each turn, a player takes one or more items from one pile.
- The player taking the last item loses.

Nim is an *impartial* game, in the sense that the set of available moves depends only on that state of the piles, and not on whose turn it is. The problem is to find a winning strategy: decide which positions are a win for the first player, and how to play in such positions.

[1] `https://en.wikipedia.org/wiki/Nim`

This can be formulated as a programming problem, where you need design and implement a function to select a move for a given game position. It can also be offered as a *reactive task*, Verhoeff (2009).
However, the problem that I first want to address is how to present a winning strategy that does not require prior knowledge of binary notation, so that it can be explained to pupils in primary school.

2. Presenting a Simple Solution

Impartial games are mathematically well understood via the *Sprague-Grundy Theorem*[2], especially for *normal* play, where the player who cannot move loses. But in this article we use the misère rule: the player who cannot move wins, making it a bit more interesting.

When only piles of size 1 remain, it is easy to see how the game proceeds. When the number of size-1 piles is even, the player to move wins, and loses when it is odd. Therefore, when all piles *except one* have size 1, the player to move can win, viz. by taking away from the larger pile such that an odd number of piles of size 1 remain, i.e., by taking away all or all-but-one items from the larger pile.

What remains to analyze are game positions with at least two piles whose size is larger than 1. It turns out (some explanation will follow) that in this case we need to break each pile into distinct groups whose size is a power of 2. Traditionally, this is accomplished by writing the pile size in binary notation.

However, I felt that too complicated to explain in primary school. And then it struck me that there is an easy way to avoid binary notation. For each pile, we are going to form groups as follows:

1. Break the pile down into groups of size 1.
2. Repeatedly combine two groups of equal size into one group.
3. This terminates when all group sizes are different.

Observe that it is an *invariant* of this process that the group sizes are powers of 2: Initially, in Step 1, the group sizes are $1 = 2^0$, and in Step 2 the group size doubles: $2 \times 2^i = 2^{i+1}$. The number of groups is a suitable *variant function*, which decreases in each iteration of Step 2, proving loop termination. Note that this is a non-deterministic algorithm.

Once all piles have been broken down into such groups, proceed as follows:

1. For each group size, determine how often such groups occur.
2. Consider the largest group size G that occurs an *odd* number of times.
3. If no such group exists, the position is lost; otherwise, proceed with Step 4.
4. Start by removing already 1 item from a pile P that has a size-G group.
5. Regroup the remaining items of that group as explained above.
6. For all group sizes $S < G$ in pile P, do the following:

[2] https://en.wikipedia.org/wiki/Sprague?Grundy_theorem

# size-S groups in P	in total	Remove from P
0	...	does not happen
1	even	0 size-S groups
1	odd	1 size-S group
2	even	2 size-S groups
2	odd	1 size-S group

7. After this move, all group sizes occur an *even* number of times. Hence, it leaves a lost position.

Breaking down the group of size $G - 1 = 2^k - 1$ in Step 5, you obtain k groups of all sizes 2^i with $i < k$. Therefore, in Step 6 every group size $S < G$ occurs at least once.

By the way, this also explains why these groups sizes, being powers of 2, are useful. But do note that there is no need to know about powers of 2 to carry out these algorithms.

3. Programming the Simple Solution

In a functional programming language that supports patterns, the first algorithm, which splits a pile into groups, can be elegantly expressed. Here it is using the Wolfram Language, Wolfram (2016):

```
(* Create groups of 1, given the pile size n *)
singletons[n_Integer] := Table[{1}, n];

(* Combine two equal groups, using a pattern *)
combine[{x___, a_, y___, a_, z___}] := {x, Join[a, a], y, z};
(* Do nothing if no equal groups are present *)
combine[list_List] := list;

(* Repeatedly combine equal groups until no more change *)
combineStar[list_List] := FixedPoint[combine, list];

(* Split a pile of size n *)
split[n_Integer] := combineStar @ singletons @ n;

(* Split all piles in a position *)
split[position_List] := Map[split, position];
```

Some *examples*:

```
singletons[5]
   {{1}, {1}, {1}, {1}, {1}}

combine[{{1}, {1}, {1}, {1}, {1}}]
   {{1, 1}, {1}, {1}, {1}}
```

```
split[5]
   {{1, 1, 1, 1}, {1}}

split[{3, 4, 5}]
   {{{1, 1}, {1}}, {{1, 1, 1, 1}}, {{1, 1, 1, 1}, {1}}}
```

The second algorithm, which determines whether a position is won and how to move is equally elegant.

```
(* Reduce by removing a pair of equal groups *)
reduce[{x___, a_, y___, a_, z___}] := {x, y, z};
(* Do nothing if no equal groups are present *)
reduce[list_List] := list;

(* Repatedly reduce a list until no more change *)
reduceStar[list_List] := FixedPoint[reduce, list];

(* Determine the nimsum of a list *)
nimsum[list_List] := reduceStar @ Catenate @ list;

(* Determine whether a position is won *)
won[position_List] := nimsum @ split @ position != {};

(* Move to make, if position is won and
   has at least two groups of size > 1 *)
move2[position_] :=
   Block[{s, ns, mx, i},
      s = split @ position;
      ns = nimsum @ s;
      (* Determine largest group in ns *)
      mx = First @ MaximalBy[ns, Length];
      (* Determine index of pile with group mx *)
      i = First @ FirstPosition[s, mx];
      (* Replace pile i with nimsum of that pile and ns *)
      ReplacePart[position, i -> Total[nimsum @ {ns, s[[i]]}, 2]]
   ];
```

An *example*

```
nimsum @ split @ {3, 4, 5}]
   {{1, 1}}

won[{3,4,5}]
   True

move2[{3,4,5}]
   {1, 4, 5}
```

Programming `move1` that moves optimally when at most one group has a size greater than 1 is left as an exercise.

I admit that it takes some time to get acquainted with functional programming. But once you do, it does pay off.

4. Conclusion

Finding a simple way for presenting a solution to a problem can in itself benefit from problem solving. I have applied this to presenting an optimal strategy for classical nim, a well-known taking game. Finally, I have shown how this strategy can be expressed in the Wolfram Language using functional programming.

Let me finish by giving you a new challenge. In *classical nim*, a player must take one or more items from *exactly one* pile. Here is a less well-known variant: at each turn, the player must take one or more items from *one or two* piles. Find the winning positions for the first player and how to determine a winning move.

While you are at it, the game *Chomp*[3], is still unsolved, in the sense, that we do not know how to determine a winning move for the first player (although it has been proven that the first player can win).

References

Dore, R. *et al.* (2010). "Math puzzles for dinner". *MathOverow*, 24 Jun. 2010. (Accessed 19 Jun. 2016) `http://mathoverflow.net/questions/29323/math-puzzles-for-dinner/`

Perfect, C. (2011). *Solving the* "*princess on a graph*" *puzzle*. Blog, 15 Dec. 2011. (Accessed 19 Jun. 2016) `http://checkmyworking.com/2011/12/solving-the-princess-on-a-graph-puz`

Verhoeff, T. (2009) 20 Years of IOI competition tasks. *Olympiads in Informatics*, 3, 149–166.

Wolfram (2016). *Wolfram Language*. (Accessed 20 Jun. 2016) `http://www.wolfram.com/language`

T. Verhoeff is Assistant Professor in Computer Science at Eindhoven University of Technology, where he works in the group Software Engineering & Technology. His research interests are support tools for verified software development and model driven engineering. He received the IOI Distinguished Service Award at IOI 2007 in Zagreb, Croatia, in particular for his role in setting up and maintaining a web archive of IOI-related material and facilities for communication in the IOI community, and in establishing, developing, chairing, and contributing to the IOI Scientific Committee from 1999 until 2007.

[3] `https://en.wikipedia.org/wiki/Chomp`