ParaSet Final Report

Ellen Guo (ekg2134), Ryan Xu (rx2189), Cindy Zhu (cwz2102)

1 Introduction

Set® is a card game played with a deck of 81 unique cards. Each card has four properties: color, shape, shading, and number of figures. Each property has three values as follows:

- Color: Red, Green, or Purple.
- Shape: Oval, Diamond, or Squiggle.
- Shading: Solid, Shaded, or Open.
- Number of Figures: One, Two, or Three.

A valid set consists of three cards where for each property, the cards must either all have the same value or all different values.

To play the game, deal 12 cards. Any number of players try to find a valid set as fast as they can; when they find it, the three cards are removed and new cards are added.

We focus on a generalized game of Set® by creating a game where C cards are dealt, with p types of properties that each take on v possible values. Thus, the classic game of Set uses $C = 12$, $p = 4$, and $v = 3$ (and has a deck of $v^p = 3^4 = 81$ unique cards). A card c has p properties, each of which we will denote as $c[i]$ for $1 \leq i \leq p$.

A valid set therefore consists of v cards $c_1, c_2, \ldots, c_v$ such that for every property i, either $c_1[i] = c_2[i] = \cdots = c_v[i]$ or $c_1[i] \neq c_2[i] \neq \ldots \neq c_v[i]$.

The problem that we present is as follows: **Given C distinct cards (having different properties), determine all valid sets that can be made from a subset of the C cards.**

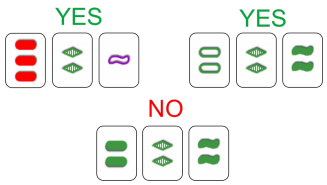


Figure 1: Examples of Sets and Non-Set. The first set (top left) has cards where the colors are different, the shapes are different, and shadings are different, and the number of figures is different. The second set (top right) has cards where the colors are the same, the shapes are different, the shadings are different, and the number of figures is the same. The last set is invalid because though the colors are the same, the shapes are different, and the number of figures is the same, the shadings are wrong: two are solid, but one is shaded (so not all the same or different).

2 Algorithmic Structure

Our algorithm across all variations takes the same basic structure. Our input is C unique cards (in practice, we randomly generate these within our program as well, which is trivial compared to the rest of the computation). Given these C cards, we first compute all combinations $\binom{C}{v-1}$ (we call each of these combinations a "preSet", for reasons which will soon become apparent).

Then, for each preSet, we can compute whether a valid Set can be created as follows: Notice that each preSet either cannot form a Set (it already violates the constraints of a Set) or that it has one *unique* card that can be added to form a valid Set. If such a card exists, check whether it exists within the cards that were dealt; if so, return the preSet combined with the missing card as a valid set.

Finally, note that except where noted, cards are represented in code as a list of p Ints, with each Int being the parameter's value at that index.

It should be observed that this algorithm grows very quickly, to the order of $\binom{C}{v-1}$. However, p has little effect on the runtime, and generally has more to do with how many valid solutions there are. Due to these observations, we generally stuck to c = [150, 200], v = 5 for testing (200 choose 5 = 2.5e9).

3 Algorithm 1

3.1 Sequential

In our baseline sequential implementation, we compute the permutations via recursive lists and the Cons operator.

```
type Card = [Int]

generatePreSets :: Int -> [Card] -> [[Card]]
generatePreSets v = generatePreSets' (v - 1)
  where
    generatePreSets' 0 _ = [[]]
    generatePreSets' _ [] = []
    generatePreSets' n (x : xs) = map (x :)
        (generatePreSets' (n - 1) xs) ++ generatePreSets' n xs
```

These preSets are then evaluated and combined into the final result. (The details of the evaluation are not important, but it notably involves a set membership check. For the full code, see the appendix.)

```
possibleSets :: [Card] -> Int -> [[Card]]
possibleSets dealtCards v =
  let preSets = generatePreSets v dealtCards
   in mapMaybe (getPossibleSet (Set.fromList dealtCards) v) preSets
```

These two sections of code are the focal points of our modification for parallelization.

3.2 Parallelization

We experimented with the parallelization of preset generation but had poor results similar to the sequential algorithm. In particular, we were dealing with low productivity, with a large percentage of time spent on garbage collection and low activity. Instead, we opted to avoid parallelizing the generation of preSets and focus on their evaluation in our next attempt. Here, we split the preSets into chunks of 10,000 and use one spark per chunk to evaluate them in parallel. We tested some different chunk sizes from 1000 to 100,000 and found that a chunk size of 10,000 provided the best speed up.

```
possibleSets :: [Card] -> Int -> [[Card]]
possibleSets dealtCards v =
  let preSets = generatePreSets v dealtCards
      preSetChunks = chunksOf 10000 preSets
   in concat $ parMap rseq (mapMaybe
    (getPossibleSet (Set.fromList dealtCards) v)) preSetChunks
```

3.3 Analysis

We ran our parallelization of Algorithm 1 with the default GHC settings and with garbage collector configuration using GC flags, both of which resulted in speed-ups from the sequential method, with the GC flag parallelization resulting in the largest speedup. For the run with the default GHC settings, Figure 2 left shows that the garbage collection was bad - lots of the total time was spent on garbage collection. This is verified when we take a look at the actual Threadscope run time breakdowns as well in Table 1, where we see that the productivity for the parallel implementation with no GC flag went down compared to the sequential productivity.

To address this garbage collection issue, we experimented with some GHC RTS options. We used -H to provide a suggested heap size for the garbage collector and tested values for -H from 0.5G to 8G. We found that -H2G, or a heap size of 2GB worked best for our tests. These GC flags resulted in less interleaving garbage collection work, especially in comparing the 2 parallel threadscopes in figure 2. We can also see that there is much higher activity overall, indicating that the parallelization is working much better with the GC flag as compared to without. This improvement from the GC flag allowed us to achieve a 2.16x speed-up when compared to sequential algorithm 1. In addition, we can see in figure 3 that when we increase the cores for the parallel with GC flag runs, we see a speed up of up to 2.86x compared to just 1 core.

	Sequential	Parallel no GC Flag	Parallel with GC Flag
Total Time	10.50s	9.13s	4.86s
Mutator Time	8.35s	5.58s	3.84s
GC Time	2.15s	3.55s	1.02s
Productivity (mutator vs total)	79.5%	61.1%	79.1%

Table 1: Table comparing Threadscope stats for versions of Algorithm 1. All using C=150, v=5, p=5, 8 cores.

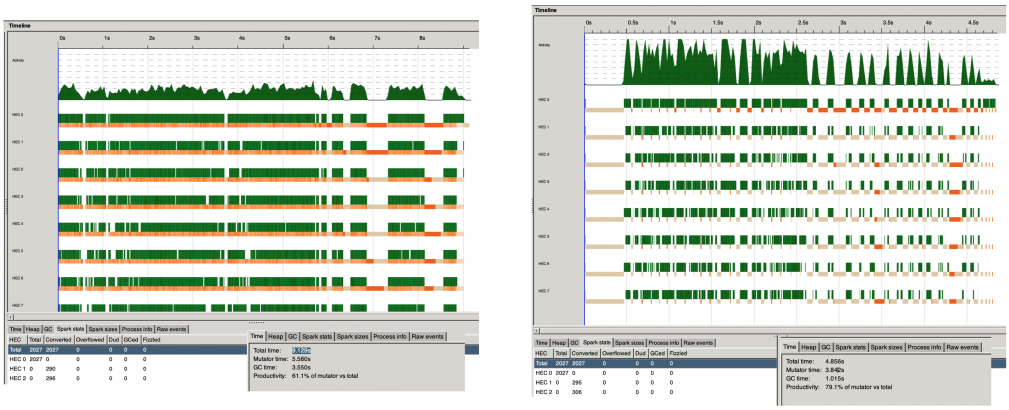


Figure 2: Threadscope snapshots parallelizations of Algorithm 1. Left: without GC flag; Right: GC flag -H2G

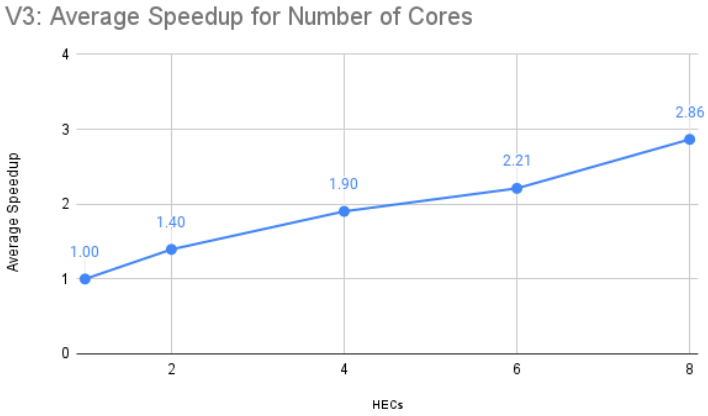


Figure 3: Average speedup for parallel implementation of Algorithm 1 using GC flag as a function of increasing cores.

4 Algorithm 2

4.1 Sequential

As the original preSet generation algorithm was plagued by garbage collection (likely due to the construction of large amounts of lists), we tried an alternative method without list concatenation. This method generates all bitstrings of C bits where $v-1$ bits are equal to 1; we then use this "one-hot" bitstring to determine which cards to include in the preSet.

```
getBitstrings :: Int -> Int -> [Integer]
getBitstrings n k = takeWhile (< bit (n + 1)) $ iterate next (bit k - 1)
  where
    next x =
```

```
      let smallest = x .&. negate x
          ripple = x + smallest
          new_smallest = ripple .&. negate ripple
       in ripple .|. (((new_smallest `div` smallest) `shiftR` 1) - 1)

bitStringToPreset :: [Card] -> Integer -> [Card]
bitStringToPreset dealtCards bitstring =
  [x | (x, i) <- zip dealtCards [0 ..], testBit bitstring i]

bitStringToMaybeSet :: [Card] -> Set Card -> Int -> Integer -> Maybe [Card]
bitStringToMaybeSet dealtCards dealtCardsSet v bitstring =
  getPossibleSet dealtCardsSet v $ bitStringToPreset dealtCards bitstring
```

Then we can map `bitStringToMaybeSet` over all the bitstring encodings to get our valid sets.

```
possibleSets :: [Card] -> Int -> [[Card]]
possibleSets dealtCards v =
  let c = length dealtCards
   in mapMaybe
        (bitStringToMaybeSet dealtCards (Set.fromList dealtCards) v)
        (getBitstrings c (v - 1))
```

4.2 Parallelization

This time, with less GC activity, the algorithm is more conducive to parallelization. It is difficult to parallelize the generation of bitstrings with the new algorithm, but we can still parallelize the evaluation of preSets through the same methods used before. First, we use the "chunking" method:

```
possibleSets :: [Card] -> Int -> [[Card]]
possibleSets dealtCards v =
  let c = length dealtCards
      bitStringChunks = chunksOf 5000 $ getBitstrings c (v - 1)
   in concat $
        parMap
          rseq
          (mapMaybe (bitStringToMaybeSet dealtCards (Set.fromList dealtCards) v))
          bitStringChunks
```

We also tried employing parBuffer as well:

```
possibleSets :: [Card] -> Int -> [[Card]]
possibleSets dealtCards v =
  let c = length dealtCards
   in catMaybes
        ( map
            (bitStringToMaybeSet dealtCards (Set.fromList dealtCards) v)
            (getBitstrings c (v - 1))
            `using` parBuffer 5000 rseq
        )
```

4.3 Analysis

Both parallelization algorithms were successful, getting 2.5x speedups compared to the raw sequential algorithm, and 2.8x speedups for one core versus 8. Running the parallel code on multiple cores, we see the results in Fig. 4: parallelization gains taper off after 4 cores, most likely as overhead and garbage collection start to compromise concurrency and parallelization gains. Fig. 5 gives us some more insight: for chunking, we see higher activity, but large segments of GC time, whereas for `parBuffer`, we don't have as high activity, but better distribution and less GC. This would make sense as chunking lists would force list elements to be in WHNF early; and for `parBuffer`, perhaps each spark is simply not doing enough work. This also lines up with the speedup graph, where we see that the chunking implementation (which struggles with memory) performs better on less cores, perhaps due to less memory pressure as there's less copies of data structures when there's less cores.

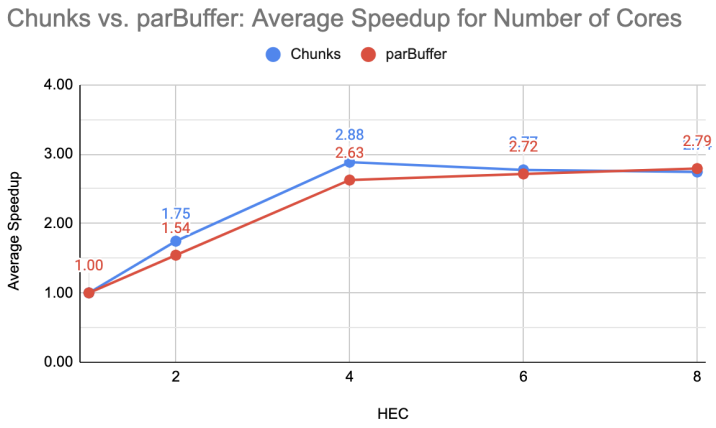


Figure 4: Average speedups for both parallel implementations of Algorithm 2 as a function of increasing cores.

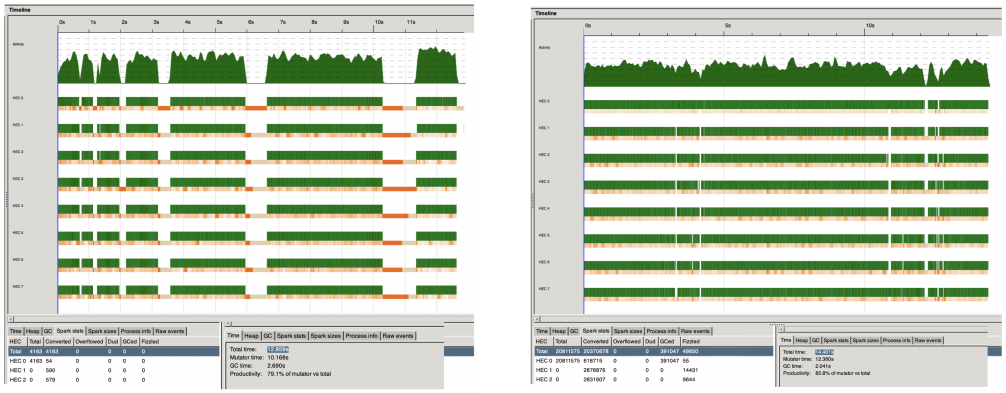


Figure 5: Threadscope snapshots of both parallelizations of Algorithm 2. Left: chunked implementation (chunksize 5000); Right: with `parBuffer` size 5000.

(Another small thing to note is that though there isn't any overflwo, there is some fizzling and GC of the sparks in the `parBuffer` version , which makes sense since there's a LOT of sparks being created. However, proportional to the total number of sparks, the rate of fizzle is only 0.25% and 1.5% GC'ed sparks, so we were pretty happy with the results.)

5 Conclusion

Overall, our first algorithm suffers from a high memory usage that leads to garbage collection dominating the runtime of the parallelized versions, although parallelization is still capable of creating significant improvements. Our second algorithm is slower, but it is lighter on memory and can be readily parallelized. In both algorithms, parallelization leads to a roughly 2.75x speedup for $c, v, p = 150, 5, 5$.

However, it should be noted that to achieve such a speedup for the first algorithm, the heap size argument must be tuned for the parameters given, while the second algorithm generally always achieves its speedup with more cores.

6 Usage

```
Usage: paraset [flags] <cards dealt> <number of values> <number of traits>
  -s     --silent       Silences all output.
  -d     --deck         Prints the deck generated.
  -n     --newline      Prints each solution on a separate line.
  -f     --file=        Optional filename containing dealt cards;
                        otherwise randomly generated
  -r 42  --randseed=42  Sets the random seed used.
  -v 6P  --version=6P   Sets the version used (latest = 6P).
                        Valid options: 6P, 6C, 6, 5, 4, 3, 2, 1
         --help         Print this help message
```

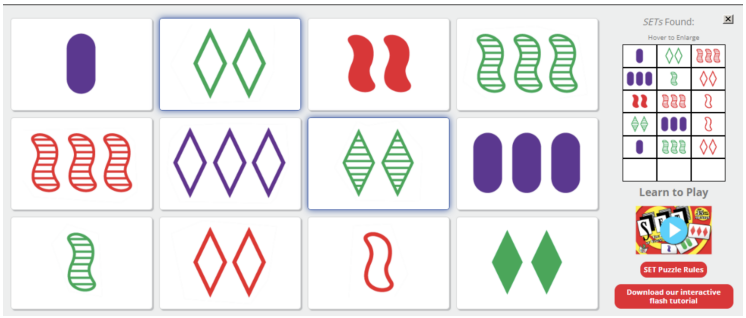
Note that the valid versions correspond as follows:

- 1: Sequential algorithm 1
- 2: Failed algorithm 1 parallelization
- 3: A1 Parallel preSet evaluation
- 4: (Failed) card data format experiment
- 5: Sequential algorithm 2
- 6: Naive parallelization of V5
- 6C: Chunked parallelization of V5
- 6P: ParBuf parallelization of V5

7 Sample input/output

```
$ stack exec -- paraset-exe -dn -v 5 -f "./test/sampleDeal.txt" 12 3 4 +RTS -N8
Dealt cards:
[[1,1,1,2],[1,2,1,1],[1,2,2,3],[1,2,3,2],[2,1,1,2],[2,1,2,2],[2,1,3,2],[2,2,2,1],
[2,2,2,3],[3,1,1,3],[3,3,3,1],[3,3,3,3]]
Solutions:
[[1,2,1,1],[1,2,2,3],[1,2,3,2]]
[[2,1,1,2],[2,1,2,2],[2,1,3,2]]
[[1,2,2,3],[2,1,1,2],[3,3,3,1]]
[[1,1,1,2],[2,2,2,3],[3,3,3,1]]
[[1,2,1,1],[2,1,2,2],[3,3,3,3]]
[[1,1,1,2],[2,2,2,1],[3,3,3,3]]
```

corresponds to the following deal and its solution (5 solutions are on the right hand side, the 6th solution is among the selected cards):



where the cards are encoded as such (so the left-top-most card would be [3,3,3,1]):

Property Index	Property Name	Value 1	Value 2	Value 3
0	Color	Red	Green	Purple
1	Shape	Diamond	Squiggle	Oval
2	Fill	Open	Shaded	Solid
3	Number	1	2	3

8 Code Listing

8.1 App/main.hs

```
module Main (main) where

import Control.DeepSeq (force)
import Control.Exception (IOException, catch, evaluate)
import Control.Monad (when)
import Data.List (sort)
import ParsetBase qualified
```

```
import System.Console.GetOpt
  ( ArgDescr (NoArg, ReqArg),
    ArgOrder (RequireOrder),
    OptDescr (..),
    getOpt,
    usageInfo,
  )
import System.Environment (getArgs)
import System.Exit (ExitCode (ExitFailure), die, exitSuccess, exitWith)
import System.IO (hPutStrLn, stderr)
import System.Random (StdGen, getStdGen, mkStdGen)
import V1 qualified
import V2 qualified
import V3 qualified
import V4 qualified
import V5 qualified
import V6Chunks qualified as V6C
import V6Naive qualified as V6
import V6Parbuffer qualified as V6P

type Card = [Int]

data Options = Options
  { optSilent :: Bool,
    optNewline :: Bool,
    optDeck :: Bool,
    optHelp :: Bool,
    optUsePresetSeed :: Bool,
    optRandSeed :: IO String,
    optVersion :: IO String,
    optDealtCardsFile :: Maybe String
  }

startOptions :: Options
startOptions =
  Options
    { optSilent = False,
      optNewline = False,
      optDeck = False,
      optHelp = False,
      optUsePresetSeed = False,
      optRandSeed = return "42",
      optVersion = return "6P",
      optDealtCardsFile = Nothing
    }

options :: [OptDescr (Options -> IO Options)]
options =
```

```
  [ Option
      ['s']
      ["silent"]
      (NoArg (\opt -> return opt {optSilent = True}))
      "Silences all output.",
    Option
      ['d']
      ["deck"]
      (NoArg (\opt -> return opt {optDeck = True}))
      "Prints the deck generated.",
    Option
      ['n']
      ["newline"]
      (NoArg (\opt -> return opt {optNewline = True}))
      "Prints each solution on a separate line.",
    Option
      ['f']
      ["file"]
      (ReqArg (\arg opt -> return opt {optDealtCardsFile = Just arg}) "")
      "Optional filename containing dealt cards; otherwise randomly generated",
    Option
      ['r']
      ["randseed"]
      (ReqArg (\arg opt -> return opt {optUsePresetSeed = True, optRandSeed = return arg}) "42")
      "Sets the random seed used.",
    Option
      ['v']
      ["version"]
      (ReqArg (\arg opt -> return opt {optVersion = return arg}) "6P")
      "Sets the version used (latest = 6P). Valid options: 6P, 6C, 6, 5, 4, 3, 2, 1",
    Option
      []
      ["help"]
      (NoArg (\opt -> return opt {optHelp = True}))
      "Print this help message"
  ]

getVersionResults :: Int -> Int -> Int -> StdGen -> [Card] -> String -> ([[Card]], [Card])
getVersionResults c v p g dealtCards version =
  case version of
    "6P" ->
      (force $ V6P.possibleSets dealtCards v, dealtCards)
    "6C" ->
      (force $ V6C.possibleSets dealtCards v, dealtCards)
    "6" ->
      (force $ V6.possibleSets dealtCards v, dealtCards)
    "5" ->
      (force $ V5.possibleSets dealtCards v, dealtCards)
```

```
    "4" ->
      (force $ V4.possibleSets dealtCardsV4 v p, map (V4.generateCardFromIndex v p) dealtCardsV4)
      where
        dealtCardsV4 = V4.dealCardsRandom c v p g
    "3" ->
      (force $ V3.possibleSets dealtCards v, dealtCards)
    "2" ->
      (force $ V2.possibleSets dealtCards v, dealtCards)
    "1" ->
      (force $ V1.possibleSets dealtCards v, dealtCards)
    _ -> error "Invalid version"

main :: IO ()
main = do
  argv <- getArgs
  case getOpt RequireOrder options argv of
    (actions, params, []) ->
      do
        opts <- foldl (>>=) (return startOptions) actions
        let Options
              { optSilent = silent,
                optNewline = newline,
                optDeck = deck,
                optHelp = help,
                optUsePresetSeed = presetSeed,
                optRandSeed = seed,
                optVersion = version
              } = opts
        when help $ do
          hPutStrLn stderr (usageInfo usage options)
          exitSuccess
        case params of
          [c, v, p] -> do
            inputSeed <- seed
            ver <- version
            let presetG = mkStdGen $ read inputSeed
            g <- if presetSeed then return presetG else getStdGen

            dealtCards <- case optDealtCardsFile opts of
              Just filename -> do
                contents <- catch (readFile filename) handleReadFileError
                let parsedContents = reads contents :: [([Card], String)]
                if null parsedContents
                  then handleParseError
                  else return $ sort $ fst $ head parsedContents
              Nothing -> return $ ParsetBase.dealCardsRandom (read c) (read v) (read p) g

            let (res, dealtCardsAsCards) = getVersionResults (read c) (read v) (read p) g dealtCa
```

```
            evalRes <- evaluate res
            when silent exitSuccess
            when deck $ do
              putStrLn "Dealt cards:"
              print dealtCardsAsCards
              putStrLn "Solutions:"
            if newline
              then mapM_ print evalRes
              else print res
          _ -> die $ usageInfo usage options
    (_, _, errs) -> do
      hPutStrLn stderr (concat errs ++ usageInfo usage options)
      exitWith (ExitFailure 1)
  where
    usage = "Usage: paraset [flags] <cards dealt> <number of values> <number of traits>"

handleReadFileError :: IOException -> IO String
handleReadFileError _ = do
  putStrLn "Error: Could not read the file."
  exitWith (ExitFailure 1)

handleParseError :: IO [Card]
handleParseError = do
  putStrLn "Error: File contents cannot be parsed as Cards. See test/sampleDeal.txt for an exampl
  exitWith (ExitFailure 1)
```

8.2 Src/V3.hs

```
module V3
  ( dealCardsRandom,
    possibleSets,
  )
where

import Control.Parallel.Strategies (parMap, rseq)
import Data.List (sort)
import Data.List.Split (chunksOf)
import Data.Map qualified as Map
import Data.Maybe (mapMaybe)
import Data.Set (Set)
import Data.Set qualified as Set
import System.Random (Random (randomR), StdGen)

{-
cards are represented as lists, where the index represents the trait, and
c[i] represents the value of trait i
-}
type Card = [Int]
```

```
{-
main function: given parameters
  C: number of cards dealt,
  v: number of values per trait,
  p: number of traits
  g: a random number generator
-}
possibleSets :: [Card] -> Int -> [[Card]]
possibleSets dealtCards v =
  let preSets = generatePreSets v dealtCards
      preSetChunks = chunksOf 10000 preSets
   in concat $
        parMap
          rseq
          (mapMaybe (getPossibleSet (Set.fromList dealtCards) v))
          preSetChunks

{-
  We generate c randomly dealt cards through generating random "swaps".
  The initial "deck" consists of all cards in sorted order (here, each card
  is represented as a single integer from 0 to v^p - 1, instead of its constituent parts.)
  Then we generate c "swaps", where for each of the first c positions in the deck,
  we swap the card with any subsequent card in the deck. After performing all swaps, we
  return the first c cards in the deck.

  This approach is inspired by https://wiki.haskell.org/Random_shuffle
  (Drawing without replacement).
-}

-- This generates all c swaps.
constructRandomList :: Int -> Int -> Int -> Int -> StdGen -> [(Int, Int)]
constructRandomList c v p sofar gen
  | c == sofar = []
  | otherwise =
      (sofar, num) : constructRandomList c v p (sofar + 1) newGen
  where
    (num, newGen) = randomR (sofar, v ^ p - 1) gen

-- This performs all swaps and constructs the list of returned cards.
constructCards :: Int -> Int -> Int -> [(Int, Int)] -> Map.Map Int Int -> [Card]
constructCards _ _ _ [] _ = []
constructCards c v p ((cardPosition, cardIndex) : nextCards) foundNums =
  generateCardFromIndex v p cardInSwapPos : nextCardList
  where
    cardInCurrentPos = Map.findWithDefault cardPosition cardPosition foundNums
    cardInSwapPos = Map.findWithDefault cardIndex cardIndex foundNums
    nextCardList = constructCards c v p nextCards
```

```
        (Map.insert cardIndex cardInCurrentPos foundNums)

-- This transforms the card from its index into its list form.
generateCardFromIndex :: Int -> Int -> Int -> Card
generateCardFromIndex _ 0 _ = []
generateCardFromIndex v remainingP index =
  remIndex + 1 : generateCardFromIndex v (remainingP - 1) num
  where
    (num, remIndex) = quotRem index v

dealCardsRandom :: Int -> Int -> Int -> StdGen -> [Card]
dealCardsRandom c v p g =
  sort $ constructCards c v p (constructRandomList c v p 0 g) Map.empty

{-
https://stackoverflow.com/questions/52602474/function-to-generate-the-unique-combinations-of-a-li
faster ones here https://stackoverflow.com/questions/26727673/haskell-comparison-of-techniques-fo
    are not great bc `subsequences` can get really huge if `length dealtCards` is large
-}
generatePreSets :: Int -> [Card] -> [[Card]]
generatePreSets v = generatePreSets' (v - 1)
  where
    generatePreSets' 0 _ = [[]]
    generatePreSets' _ [] = []
    generatePreSets' n (x : xs) =
      map
        (x :)
        (generatePreSets' (n - 1) xs)
        ++ generatePreSets' n xs

{-
  checks if a valid, correctly ordered set is possible
-}
getPossibleSet :: Set Card -> Int -> [Card] -> Maybe [Card]
getPossibleSet dealtCards v preSet =
  case getMissingCard preSet v of
    Nothing -> Nothing
    Just missingCard ->
      if Set.member missingCard dealtCards && missingCard < head preSet
        then Just $ missingCard : preSet
        else Nothing

getMissingValue :: Int -> [Int] -> Maybe Int
getMissingValue v values
  | Map.size m == 1 = Just $ head values
  | all eqOne (Map.elems m) = Just $ (v * (v + 1) `div` 2) - sum (Map.keys m)
  | otherwise = Nothing
  where
```

```
    eqOne :: Int -> Bool
    eqOne = (== 1)
    m = Map.fromListWith (+) [(val, 1) | val <- values]

getMissingCard :: [Card] -> Int -> Maybe Card
getMissingCard preSet v =
  mapM (getMissingValue v) transposedList
  where
    transpose :: [[a]] -> [[a]]
    transpose ([] : _) = []
    transpose x = map head x : transpose (map tail x)
    transposedList = transpose preSet

-- https://stackoverflow.com/questions/2578930/understanding-this-matrix-transposition-function-i
```

9 References

https://pbg.cs.illinois.edu/papers/set.pdf
SET Family Game on Amazon