The Joy of SET

Miriam Melnick
Bard College at Simon's Rock

May 2, 2011

"Pure mathematics is the world's best game. It is more absorbing than chess, more of a gamble than poker, and lasts longer than Monopoly." - Richard Trudeau, *Dots and Lines*.

1 Introduction

In the game of SET, players race to collect sets of 3 matching cards. Sometimes, they get stuck and the players claim there are no matches on the table. Is this true? Are there circumstances in which there are no matches visible? What conditions must be satisfied for this to occur? In this paper, we develop algebraic, geometric, and computational frameworks to answer these questions.

It is always important to have precise definitions for our terms. You are likely familiar with the common mathematical concept of a "set" as a collection of objects surrounded by curly brackets. Throughout this paper, we will use the following terminology to describe SET:

set An unordered collection of objects. The traditional mathematical notion of a set. Denoted with curly brackets.

SET A card game played with a special n-dimensional deck.

match[1] A set of three SET cards that conform to the SET rule as described below. Denoted using square brackets.

We introduce the game of SET by constructing the deck in a series of algorithmic steps (see Section 2.3). From there we will discuss how to play the game of SET and how to expand it from the standard 4 dimensions into n dimensions (see Section 2.5). Then we discuss the mathematical nature of SET and how we can model it using strings of 0's, 1's, and 2's (see Section 3). We will investigate the circumstances under which there are no matches visible (see Section 3.4). We will model n-dimensional SET using the vector space $\mathbb{Z}_3^n$ and discuss its algebraic structures (see Section 4). Then we will model n-dimensional SET using the projective affine space $AG(n, 3)$ and discuss the geometric consequences (see Section 5). Finally, we will examine how many cards can be dealt and still have no match. We will approach this problem primarily from a computational direction (see Section 6).

Figure 1: Three cards from the official SET deck that form a match.

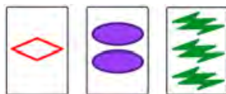


Figure 2: Three cards from my SET deck that form a match.

2 The Game of SET

2.1 What is SET?

SET is a fast-paced, multiplayer game played with a special deck of cards. The game tests pattern-matching skills as players survey a dozen cards at a time, their eyes searching for any three cards that form a match. SET's simplicity means that children as young as five years old often beat their parents. On the other end of the spectrum, avid SET fanatics can play through an entire deck in under ninety seconds. [4] The official SET deck consists of cards like those in Figure 1. [2]

For the purposes of this paper I have drawn my own SET deck which maintains all relevant characteristics of the official deck. Sample cards from my deck can be seen in Figure 2.

2.2 History

In 1974, Marsha Jean Falco, a population geneticist, was studying epilepsy in German Shepherds. As an organizational tool, she created a deck of cards with various genetic traits represented using symbols so that she could visually look for patterns. She soon realized that the cards had potential as a puzzle, and, with the encouragement of friends and family, she published the game in 1990. [17] Since then, it has become popular both in mathematical and lay environments.

2.3 Building SET

We will now build up the SET deck. Imagine we have an infinite stack of blank cards. As we develop the deck, we follow three rules:

1. For each attribute, there must be three possible values.
2. Every possible permutation of vaues must be represented.

[2]Cards and game ©1988, 1991 Cannei, LLC. All rights reserved. SET and the SET logo are registered trademarks of Cannei, LLC. Used with permission from Set Enterprises, Inc.

3. Every card must be unique.

We will call the number of attributes the "dimension" of the deck, for reasons that will become clear in the next chapter.

We begin with number. Pick a card off the deck and draw one symbol on it. Take another and draw two symbols, and one more to draw three symbols. Make all the symbols the same, so that the only variable is the number of symbols. We now have something like Figure 3. Notice that there are a total of 3 cards.

Figure 3: The 1-dimensional SET deck varies only number.

Continuing on, let's add another variable: fill. Now we make all the possible cards with number 1, 2, or 3 and fill solid, striped, or empty. Following the aforementioned rules that every permutation is represented once and only once, we now have a deck of 9 cards. Notice that $9 = 3^2$. The deck should now resemble Figure 4.

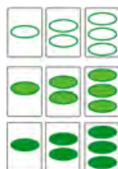


Figure 4: The 2-dimensional SET deck varies number and fill.

Now let's add the variable of shape. Let the three values be oval[3], squiggle, and diamond. We already have all of the oval cards, so we need to make a copy of the deck that has squiggles and a copy of the deck that has diamonds. This produces the deck shown in Figure 5, which has $3^3 = 27$ cards.

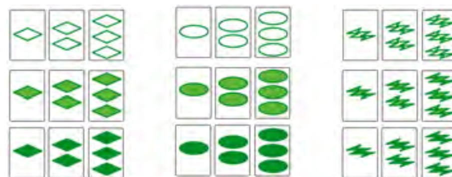


Figure 5: The 3-dimensional SET deck varies number, fill, and shape..

We're almost done. Now let's add the variable of color, with our three values being red, green, and purple. As before, we have already made $\frac{1}{3}$ of the necessary cards.

[3] In the official game, the shape is actually a stadium. I will call it an oval.

When we finish this stage, our deck looks like Figure 6. Notice that there are a total of $3^4 = 81$ cards.

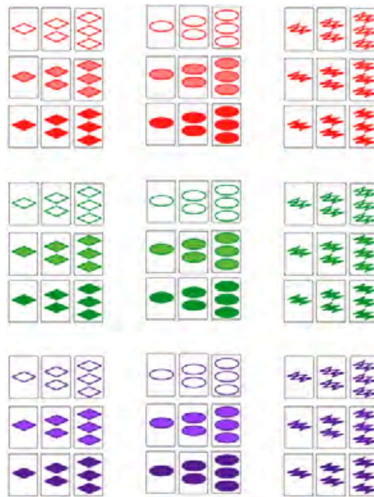


Figure 6: The 4-dimensional SET deck varies number, fill, shape, and color.

Congratulations, we have now built the standard SET deck. We summarize its features in the next section.

2.4 Playing SET

The Game of SET has one rule: a match is a collection of three cards such that, for each attribute, either all three are the same or all three are different. To begin, twelve cards are dealt face-up on the table. When a player sees a match, he or she says "SET" and picks up the three cards. If the cards form a valid match, the player gets a point, keeps the cards, and deals three new cards onto the table. If the cards do not form a valid match, the player loses a point and returns the cards to the table.

2.5 Expanding SET

The standard SET game is played with 4 attributes, as described in the previous section. However, it is also possible to play with other numbers of attributes. The SET rule is the same regardless of the dimension, and the game is still playable. If we pick out a particular subset of cards from our complete deck, we can play a lower-dimension game using a small deck like those shown in Figures 3, 4, and 5. However, these games are less intellectually stimulating. In particular, 1-dimensional SET games are very boring: you deal the three cards, someone says "SET", and they win.

Just as we built the 4-dimensional SET deck, we can add more complexity to play in higher dimensions. If we allow a new attribute, say, background color (with three possible values as always), then we would have $3^5 = 243$ cards. We could continue

this procedure to play n-dimensional SET, for $n \in \mathbb{Z}, n \geq 1$. In each case, we would have 3^n cards.

3 Representing SET

3.1 The SET Deck

The standard game of SET is played with a deck of 81 cards. Each card has four attributes, and each attribute has one of three possible values. The possibilities are shown in Table 1. We will encode the information in the numbers 0, 1, and 2.

color	0=	green	1=	red	2=	purple
number	0=	1	1=	2	2=	3
fill	0=	empty	1=	striped	2=	solid
shape	0=	oval	1=	diamond	2=	squiggle

Table 1: Attribute values for 4-Dimensional SET

Every possible combination of the above attributes is represented on exactly one card. There are no duplicates. The complete 4-dimensional deck is shown in Figure 6.

3.2 Basic Properties of SET

Before we dive in, let us establish some basic properties of the SET game. Recall the SET rule: three cards form a match if and only if for each attribute, the cards all have the same value or the cards have three different values.

Theorem 1. *There is exactly one match containing any two SET cards.*

Proof. Let us investigate the attributes one at a time. For each attribute, there are two cases:

1. If the two cards are the same in this attribute (ex. both green), then to form a match, the third card must also be the same (ex. also green).

2. If the two cards differ in this attribute (ex. one green and one purple), then to form a match, the third card must also be different (ex. red).

Thus the value for each attribute is forced by the first two cards. Since all SET cards are unique, there is exactly one card with that set of attributes. □

We can restate Theorem 1 as saying that two SET cards uniquely define the third in a match. In both the game of SET, it does not matter in what order you pick up the three cards. Therefore in our analysis, the match [A, B, C] is equivalent to the match [A, C, B]. How many ways can we order the 3 cards in a match? We can answer this question using combinatorics. Recall that a match must contain three distinct cards,

with no duplicates.

Lemma 1. *$\forall$ match [A, B, C] can be permuted in exactly 6 different ways.*

Proof. The first card can be any of the 3. The second card can be either of the 2 remaining cards. The final card is forced. The number of distinct permutations is $3 * 2 * 1 = 3! = 6$. $\square$

3.3 Representing 1-dimensional SET

In 1-dimensional SET, there are a total of 3 cards. Using the encoding schema given in Table 1, we can denote the three cards as the 1-tuples (0), (1), and (2).

Continuing in the same vein, how many total matches are there in the one-dimensional game? We can answer this question using combinatorics. Recall that a match must contain three distinct cards, with no duplicates.

Theorem 2. *In 1-dimensional SET there is only 1 possible match.*

Proof. There are 3 choices for the first card, 2 choices for the second card, and 1 choice for the last card. But a match is order-independent, so we must divide by the number of ways that three cards can be rearranged. By Lemma 1, there are 6 ways to arrange the cards in a match. So we have $\frac{3*2*1}{6} = 1$ possible match in 1-dimensional SET. $\square$

Lemma 2. *A line through 3 points in $\mathbb{Z}_3$ corresponds (by the schema given in Table 1) to a match in the 1-dimensional SET game.*

Proof. The proof is by (brief) exhaustion. There is only one line in $\mathbb{Z}_3$, and it goes through the three points (0), (1), and (2). By the schema given in Table 3, these 1-tuples correspond to the cards with 1, 2, and 3 symbols. These symbols do form a match by the SET rule because they are all distinct in their only attribute. $\square$

3.4 Representing Matchless Sets

During the course of a game of SET, the players often encounter a roadblock: they claim there are no matches on the table. This is not a mark of their inadequacy as players - they are often correct. We now move to the primary focus of this paper: under what conditions are there 0 matches on the table? What is the highest number of cards you can deal and still have no match visible? Is there a way to prove the existence of a match, even if you can't see it? We will look at three approaches to this problem: algebraic, geometric, and computational.

4 Algebra in SET

We can model SET in the n-dimensional vector space $\mathbb{Z}_3^n$.

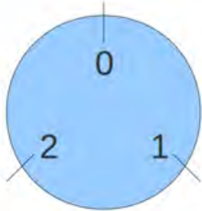


Figure 7: Modulus 3 as "clock arithmetic".

4.1 1-Dimensional SET in $\mathbb{Z}_3$

Modulus 3 (see Figure 7), which we denote $\mathbb{Z}_3$, is a group under modular addition. The operation is defined to be both associative and commutative and its operation table is given in Table 2.

$+_3$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Table 2: The addition table for modulus 3.

We can put the elements of $\mathbb{Z}_3$ into a 1-1 onto correspondence with the cards of the 1-dimensional SET deck, as shown in Figure 3. To do this we use the coordinates given in Table 1. A card in 1-dimensional SET contains only one piece of information, so it can be entirely described by one coordinate (ex. (1)). When we look at cards as points in this 1-dimensional vector space, it is natural to ask what a line represents. In $\mathbb{Z}_3$ there is only one line, and it passes through all three points. In Section 3, we noted that the three cards in the 1-dimensional deck form a match.

Lemma 3. *1-D SET cards A, B, and C form a match* $\Leftrightarrow$ *their corresponding points in* $\mathbb{Z}_3$ *sum to 0 mod 3* (0_3).

Proof. The proof is again by (brief) exhaustion.
$(\Leftarrow)$ Since 1-dimensional SET cards A, B, and C form a match, we know that they are either all the same or all distinct in each attribute. But they only have one attribute and there are no duplicate cards. Hence they are all distinct. Addition modulus 3 is associative and commutative, so we can say without loss of generality that their sum mod 3 is $0 + 1 + 2 = 0_3$.
$(\Rightarrow)$ By hypothesis $(a_1) + (b_1) + (c_1) = 0_3$. There are 3 choices for (a_1).

- $(a_1) = 0$: (b_1) cannot be 0 because that would make it identical to A in all attributes and there are no duplicate cards in SET, so it must be 1 or 2.
  - $(b_1) = 1$: $0 + 1 + c_1 = 0_3 \Leftarrow c_1 = 2$. Now A, B, C are (0), (1), (2), and those three cards form a match.

- $(b_1) = 2$: $0 + 2 + c_1 = 0_3 \Leftarrow c_1 = 1$. Now A, B, C are (0), (2), (1), and those three cards form a match.

- $(a_1) = 1$: (b_1) must be 0 (which forces $(c_1) = 2$) or 2 (which forces $(c_1) = 0$).
- $(a_1) = 2$: (b_1) must be 0 (which forces $(c_1) = 1$) or 1 (which forces $(c_1) = 0$).

□

Recall the SET rule: three cards form a match if and only if for each attribute, either all three values are the same or all three are different. Do these three cards form a match in 1-D SET?

Theorem 3. *The three cards in 1-dimensional SET form a match.*

Proof. The cards have only one attribute. In this attribute, they have three distinct values. By the SET rule, they form a match. □

4.2 2-Dimensional SET in $\mathbb{Z}_3^2$

We denote the Cartesian product $\mathbb{Z}_3 \times \mathbb{Z}_3$ as $\mathbb{Z}_3^2$. Just as we construct the complex plane by putting the reals on the horizontal axis and the reals on the vertical axis, we now put $\mathbb{Z}_3$ on each axis and see a total of $3^2 = 9$ discrete points. We can put these points into a 1-1 onto correspondence with the 2-dimensional SET cards, as seen in Figure 4.

We model 3- and 4-dimensional SET in the same way, and generalize the process below.

4.3 n-Dimensional SET in $\mathbb{Z}_3^n$

We model n-dimensional SET in $\mathbb{Z}_3^n$, an n-dimensional vector space. Points in the vector space are in a 1-1 onto correspondence with cards in the n-dimensional deck. We now formulate our first observations.

Lemma 4. *A,B,C make an n-dimensional SET* $\Leftrightarrow \vec{A} + \vec{B} + \vec{C} = \vec{0}$ *in* $\mathbb{Z}_3^n$.

Proof. ($\Rightarrow$) Let $A = (a_1, a_2, ..., a_n)$, $B = (b_1, b_2, ..., b_n)$, and $C = (c_1, c_2, ..., c_n)$. For each coordinate i ∈ (1, 2, ... n), we know that either $a_i = b_i = c_i$ or $(a_i \neq b_i, b_i \neq c_i, a_i \neq c_i)$. We split this into 4 cases.

- Case 1 $(a_i = b_i = c_i = 0)$: $a_i + b_i + c_i = 0 + 0 + 0 = 0$.
- Case 2 $(a_i = b_i = c_i = 1)$: $a_i + b_i + c_i = 1 + 1 + 1 = 0$ by the rules of addition mod 3.
- Case 3 $(a_i = b_i = c_i = 2)$: $a_i + b_i + c_i = 2 + 2 + 2 = 0$ by the rules of addition mod 3.
- Case 4 $(a_i \neq b_i, b_i \neq c_i, a_i \neq c_i)$: There are only 3 possible values, and all three values have to be distinct. Therefore we must have one of each. $0 + 1 + 2 = 0$ by the rules of addition mod 3.

(⇐) By hypothesis, for each i in {0,1...n}, $a_i + b_i + c_i = 0$ in modulus 3. There are 3 options for a_i:

- $a_i = 0$. Then there are three choices for b_i:
  - $b_i = 0$. The hypothesis forces $c_i = 0$, and by Case 1, above, this is a match.
  - $b_i = 1$. The hypothesis forces $c_i = 2$, and by Case 4, above, this is a match.
  - $b_i = 2$. The hypothesis forces $c_i = 1$, and by Case 4, above, and the associativity and commutativity of addition modulus 3, this is a match.
- $a_i = 1$. Then there are three choices for b_i:
  - $b_i = 0$. The hypothesis forces $c_i = 2$, and by Case 4, above, and the associativity and commutativity of addition modulus 3, this is a match.
  - $b_i = 1$. The hypothesis forces $c_i = 1$, and by Case 2, above, this is a match.
  - $b_i = 2$. The hypothesis forces $c_i = 0$, and by Case 4, above, and the associativity and commutativity of addition modulus 3, this is a match.
- $a_i = 2$. Then there are three choices for b_i:
  - $b_i = 0$. The hypothesis forces $c_i = 1$, and by Case 4, above, and the associativity and commutativity of addition modulus 3, this is a match.
  - $b_i = 1$. The hypothesis forces $c_i = 0$, and by Case 4, above, and the associativity and commutativity of addition modulus 3, this is a match.
  - $b_i = 2$. The hypothesis forces $c_i = 2$, and by Case 3, above, this is a match.

□

5 Geometry in SET

Let $AG(k, q)$ be the k-dimensional affine vector space with coefficients from modulus q. We will work with $AG(n, 3)$, which is isomorphic to $\mathbb{Z}_3^n$ (as defined in Section 4).

cap a set of k-tuples $\{A_1, A_2, ...A_m\}$ such that no three A_i are collinear.
Equivalently, a set of n points AG(k,q) such that no 3 are collinear.
We define the *size* of the cap to be m.

maximal cap the size of the largest cap in n-dimensional SET.

Affine Collinearity Property: Every line in $AG(n, 3)$ corresponds to a match in n-dimensional SET and every n-dimensional match corresponds to a line in $AG(n, 3)$.

5.1 Maximal Caps in Low Dimensions

What is the size of the maximal cap in $AG(n, 3)$? Let's start with 1-dimensional SET, which we model in $AG(1, 3)$.

Theorem 4. *In 1-dimensional SET, the maximal cap is 2.*

Proof. In 1-dimensional SET, there are only 3 cards: (0), (1), (2). If you deal any two of them, you don't have a match because a match requires 3 cards. If you deal all three of them, you DO have a match, by Lemma 3. Therefore the maximal cap is 2. □

This maximal cap can be represented graphically as in Figure 8.

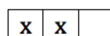


Figure 8: 1-D maximal cap.

Theorem 5. *In 2-dimensional SET, the maximal cap is 4.*

Proof. Here is an example of a cap of size 4: {(0,0), (0,2), (2,0), (2,2)}. No three of these cards make a match because they can't all be the same (there are only 2 0's and 2 2's in either slot) and they can't all be different (there are no 1's). It now suffices to show that there does not exist a cap of size 5. The proof proceeds by contradiction. Suppose there exists a 2-dimensional cap {A, B, C, D, E}. We can partition $\mathbb{Z}_3^2$ as the union of three parallel lines as in Figure 9(a). 2 of these lines contain 2 points each and (*) the remaining line contains 1 point. We call the line containing the lone point H. We now assume without loss of generality that E is the lone point. $\exists$ exactly 4 lines that contain E (L1, L2, L3, H). See Figure 9(b). By (*), above, none of {A, B, C, D} lie in H. That leaves 3 lines and 4 points. By the Pigeonhole Principle[4], two of the four points must lie in one line. But E also lies in that line. Three collinear points make a match. But by hypothesis, {A,B,C,D,E} is a cap (meaning that it contains no matches). We have reached a contradiction. Therefore $\nexists$ a cap of size 5. [15] □

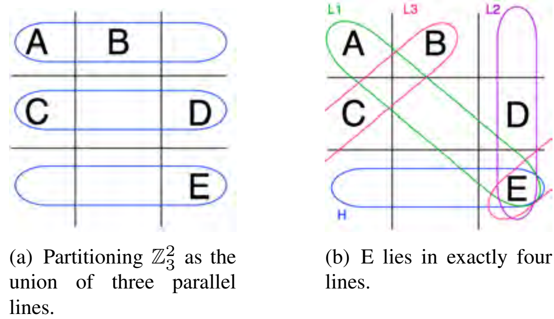
dimension	1	2	3	4	5	6	7
maximal cap	2	4	9	20	45	$112 \leq x \leq 114$	?

Table 3: All known maximal caps for $\mathbb{Z}_3^n$

6 A Computational Approach

The idea of our maximal cap algorithm is to check through all the possibilities, starting with small sets and moving up to bigger ones, until we find a size at which every set has a match. Recall that the goal is to find the largest collection (set) of cards that does not contain a match.

[4]If you have n pigeons and n+1 holes, you eventually have to put two holes in one pigeon.



(a) Partitioning $\mathbb{Z}_3^2$ as the union of three parallel lines.

(b) E lies in exactly four lines.

Figure 9: Proof: 2-dimensional maximal cap is 4.

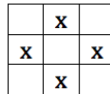


Figure 10: 2-D maximal cap

6.1 Computing Maximal Caps

The goal is to find the size of the largest collection of cards with 0 matches. In order to know that our maximal cap x is really maximal, we have to check all sets of x+1 cards and show that each one of those contains at least one match. So the basic outline of our program will be as follows. Let α denote a set of k cards. There is only one "operation": we can add a card to our set α. We can also check how many matches α contains.

6.2 First Algorithm

We first check all subsets of size 3. Some of them have a match, but some of them don't. Then we continue on to size 4, and so on. The complete computer code (written in Python) can be found in the Appendix. The algorithm follows the following instructions:

- Check all possible combinations of 3 cards. While examining a set, call it α.
  - If there exists a 3-set (a set of cardinality 3) with 0 matches, continue. We've found a cap (though it is not necessarily maximal).

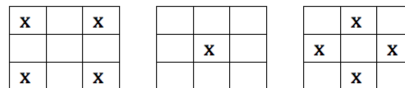


Figure 11: 3-D maximal cap

- If every 3-set contains at least 1 match, then we know the maximal cap must be 2.

- Next, check all possible combinations of 4 cards.
- Then all possible combinations of 5 cards.
- Continue until you find a match in every set of size k. Then k-1 is the maximal cap.

6.3 Improvements

My first algorithm ran too slowly to calculate the maximal caps in dimensions greater than 2. I made several improvements to it. The primary changes were switching to breadth-first search, pruning the tree, and checking for matches more efficiently.

6.3.1 Depth-first versus Breadth-first Search

We can model the search-space using a binary tree (see Figure 12). A binary tree starts with one node, which branches into two, which each branch into two, which each branch into two, and so on. In total, there are $2^{d+1} - 1$ nodes (where d is the depth of the tree), which we usually approximate as 2^{d+1} (as d gets large, the 1 becomes negligible). In reality, our tree is not *bi*nary but rather 3^n-ary (ternary for 1-dimensional SET, nonary for 2-dimensional SET, icosakaiheptary [18] for 3-dimensional SET, octacontakaihenary [19] for 4-dimensional SET, etc.). This means each branch splits into 3 (or 9, or 27, etc.) daughter branches. The consequence is that our functions grow even faster than functions on binary trees, but a binary tree still makes a good conceptual model.

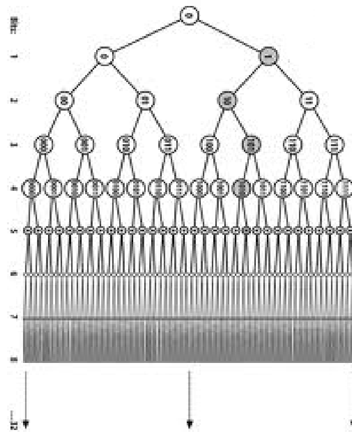


Figure 12: A binary tree.

There are two ways to traverse a binary tree: depth-first or breadth-first. In depth-first search, one starts at the top and goes all the way to the bottom of the first branch,

O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(2^n)	O(n!)

Table 4: Orders of growth of various algorithms, from fastest-running to slowest-running.

then back up to the top and all the way down to the bottom of the second branch, and so on until you reach the last node of the tree. This takes a *very* long time.

The alternative to a depth-first search is a breadth-first search. In a breadth-first search, you search all the first-level nodes, then all the second-level nodes, then all the third-level nodes, and so on. In general, breadth-first search is more efficient because you can often find what you need without canvassing every node. In Section 6.3.2, we discuss why breadth-first search is more efficient for this particular problem.

6.3.2 Pruning the Tree

prune To remove a node and all of its descendants from a binary tree.

The following is a key insight for the improved efficiency of the new program: if a set of cards contains a match, then no superset of it will contain 0 matches. In particular, if a collection α contains at least 1 match, then we do not need to waste time searching through its supersets. This means our algorithm can prune α. Now we are really making progress; pruning the tree dramatically reduces the number of collections we need to check for matches. The vast majority of 40-sets with matches also have a match in their 39-set, so we do not really need to check the 40-set at all.

6.3.3 Checking for Matches

The final improvement I made is to the way that I check for matches. Method I, which I used in the first algorithm, was to check if the new card makes a set with each possible pair of cards already in α. For example, if α (a set of n SET cards) is {A,B,C,D} with no matches and we are adding a card E, by the first method we check whether {A,B,E} make a match, whether {A,C,E} make a match, whether {B,C,E} make a match, and so on for all $\binom{n}{2}$ 3-sets.

Method II, new and improved, is to check whether the AE-card (the card that makes a match with A and E; uniquely defined by Theorem 1) is in α, whether the BE-card is in α, and so on for all n cards.

In Computer Science we talk about the order of growth (the "big O") of an algorithm, which describes how quickly the function grows when you increase the size of its input. Algorithms with O(1) grow in constant time, algorithms with O(n^2) grow with the square of the size of the input, et cetera. Table 4 shows some basic orders of growth from fastest-running (left) to slowest-running (right). The slowest-growing algorithms are the fastest-running and vice versa.

Method I has O(n!) while Method II has O(n log n). As shown in Table 4, O(n!) is a very slow algorithm and O(n log n) is a medium-speed algorithm. This demonstrates that Method II is more efficient (especially on larger inputs) than Method I.

6.4 Second Algorithm

My second algorithm begins with the creation of a first-in-first-out (FIFO) queue. We only put objects in the queue if they contain no matches. That way, whenever we pull something out of the queue, we can safely assume it contains no matches. What can we put in this queue? Well, we start with all the possible sets of 2 cards. So we put {A,B}, {A,C}, etc. all in the queue. We know these contain no matches because 2 cards is not sufficient to form a match according to the SET rule.

One the queue is initialized, we process it as follows:

- Pull something out of the bucket; call it α. Call its size k.
  - Add each possible card to α
  - Size is now $k+1$
  - If new set has no match, put it in queue
- Queue holds all combinations of 2, 3, 4... cards that contain 0 matches.
- Size of largest item from queue = maximal cap

This algorithm uses the three improvements discussed in Section 6.3. It uses breadth-first search by first checking all sets of one size, then all sets of the next size, et cetera. It uses pruning by only checking supersets of sets containing no matches. And it uses the more efficient match-checking algorithm.

7 Conclusions and Further Work

We have developed three frameworks for discussing maximal caps: algebraic, geometric, and computational. However, the problem remains unsolved for input larger than 5. The reason for this is that the order of growth (and its equivalent conceptual problems in algebra and geometry) is just too fast. But there is hope: in 2002, Edel, Ferret, Landjev, and Storme published the algebraic proof of the maximal cap in $\mathbb{Z}_3^5$. Perhaps their methods can be expanded to work in six dimensions as well. The trouble with the geometric proofs seems to be that hyperplanes and hypersolids are difficult to generalize. Difficult, but not impossible; they were used in the aforementioned proof [6]. Finally, there is plenty of room for improvement in the computational approach. Further optimization and parallelization will hopefully yield a computational proof of the 5-dimensional cap discovered only nine years ago.

References

[1] Bierbrauer, Jurgen, and Yves Edel. "Bounds on Affine Caps." *J. Combin. Des.* 10, no. 2 (July 15, 2002): 111-115.

[2] Bierbrauer, Jurgen, and Yves Edel. "Large caps in projective Galois Spaces." `http://www.mathi.uni-heidelberg.de/~yves/Papers/CapSurvey.pdf`. Accessed Feb 24, 2011.

[3] Conrey, Brian and Brianna Donaldson. "SET." http://www.mathteacherscircle.org/resources/materials/BConreyBDonaldsonSET.pdf Accessed Feb 26, 2011.

[4] Do, Norman. "The Joy of SET." Gazette of the Australian Mathematical Society. Sep 2005. http://www.austms.org.au/Publ/Gazette/2005/Sep05/mathellaneous.pdf. Accessed Feb 26, 2011.

[5] Holshouser, Arthur, Ben Klein, Harold Reiter, and Wayne Snyder. "The Commutative Equihoop and the Card Game SET." http://math.uncc.edu/~hbreiter/ComEquihoop.pdf. Accessed Feb 24, 2011.

[6] Edel, Yves, S. Ferret, I. Landjev, and L. Storme. "The Classification of the Largest Caps in AG(5,3)". *J. of Combinatorial Theory*, no. 99. A(2002): 95-110.

[7] Etingof, Pavel. "Groups Around Us." http://www-math.mit.edu/~etingof/groups.pdf. Accessed Feb 16, 2011.

[8] Fairbanks, Hillary. "The Game SET as $\mathbb{F}_3^4$." http://www.whitman.edu/mathematics/SeniorProjectArchive/2010/SeniorProject_HillaryFairbanks.pdf. Accessed Feb 24, 2011.

[9] Klee, Steven. "The Mathematics of SET." http://www.math.ucdavis.edu/~klee/SET.pdf. Accessed Feb 24, 2011.

[10] Magliery, Tom. "The SET®Home Page." http://magliery.com/Set/. Accessed Feb 4, 2011.

[11] McCullagh, Ellen. "SET and Affine Caps." http://www.math.uchicago.edu/~may/VIGRE/VIGRE2008/REUPapers/McCullagh.pdf. Accessed Feb 16, 2011.

[12] Mills, Melissa. "The Game SET and Finite Fields: An Algebraic Approach." Masters thesis, Oklahoma State University, 2006.

[13] Pellegrino, Giuseppe. "20 Is the Largest Size of a Cap in PG(4,3)." http://mint.sbg.ac.at/desc_CBoundB3M5Cap.html. Accessed Mar 3, 2011.

[14] Peterson, Ivars. "Ivars Peterson's MathTrek - SET Math." *Ivars Peterson's MathTrek - SET Math.* http://www.maa.org/mathland/mathtrek_08_25_03.html. Accessed Mar 3, 2011.

[15] Lent Davis, Benjamin, and Diane Maclagan. "The Card Game SET." http://galileo.stmarys-ca.edu/bdavis/set.pdf. Accessed Feb 24, 2011.

[16] Calderbank, A. R., and P. C. Fishburn. "Maximal three-independent subsets of $\{0,1,2\}^n$." Designs, Codes, and Cryptography, Volume 4, Number 4. http://www.springerlink.com/content/w261628463471418/. Accessed Feb 20, 2011.

[17] "The Set Game Company Homepage." *The Set Game Company Homepage*. http://www.setgame.com/. Accessed Jan 30, 2011.

[18] "What is the name of a 27-sided shape?" http://www.trueknowledge.com/q/what_is_the_name_of_a_27_sided_shape. Accessed May 3, 2011.

[19] "What is the name of an 81-sided shape?" http://www.trueknowledge.com/q/what_is_the_name_of_a_81_sided_shape. Accessed May 3, 2011.

[20] Zabrocki, Mike. "The Joy of Set or Some Combinatorial Questions Related to the Configuration of Points in $\mathbb{Z}_3^k$." *The Joy of Set*. http://www.math.yorku.ca/~zabrocki/set/. Accessed Feb 4, 2011.

8 Appendix

8.1 Programs

See attached pages.

```
import random

class Card:
    colors = {0:"red", 1:"green", 2:"purple"}
    numbers = {0:"1", 1:"2", 2:"3"}
    shapes = {0:"oval", 1:"diamond", 2:"squiggle"}
    fills = {0:"full", 1:"shaded", 2:"empty"}

    def __init__(self, c, n, s, f):
        self.c = c
        self.n = n
        self.s = s
        self.f = f
#        print 'Initialized Card'

    def ints(self):
        return [self.c, self.n, self.s, self.f]


def contains(array, element):
    for j in range(len(array)):
        if array[j] == element:
            return True
    return False

def check(A,B,C):
    '''checks three cards to see if they make a set.'''
    for a in (0,1,2,3):
        x = A.ints()[a]
        y = B.ints()[a]
        z = C.ints()[a]
        sum = x+y+z
        if (sum % 3) != 0:
            return False
    return True

def dealCard():
        ind = random.randint(0,80)
        if contains(dealt,ind):
            dealCard()
        else:
            dealt.append(ind)

def assembleDeck():
    for c in range(3):
        for n in range(3):
            for s in range(3):
                for f in range(3):
                    C = Card(c, n, s, f)
                    deck.append(C)

def chooseThree():
    for b in range(len(dealt)):
        for d in range(len(dealt)):
            for e in range(len(dealt)):
                if check(deck[dealt[b]],deck[dealt[d]],deck[dealt[e]]):
                    setCount +=1
```

```
def deal(cards):
    for i in range(cards):
        dealCard()
    #print dealt
    setCount = 0

def main():
    assembleDeck()
    setCount = 0
    sets = []
    deal(12)
#    chooseThree()
#    for k in dealt:
#        print k, deck[k].ints()
    for a in range(12):
        for b in range(12):
            for c in range(12):
                if (a!=b) and (b!=c) and (a!=c):
                    #print 'checking set'
                    A = dealt[a]
                    B = dealt[b]
                    C = dealt[c]
                    if check(deck[dealt[a]], deck[dealt[b]], deck[dealt[c]]):
                        if A<B and B<C:
                            newset = [A,B,C]
                        elif A<B and C<A:
                            newset = [C,A,B]
                        elif A<C and C<B:
                            newset = [A,C,B]
                        elif B<A and A<C:
                            newset = [B,A,C]
                        elif B<C and C<A:
                            newset = [B,C,A]
                        elif C<B and B<A:
                            newset = [C,B,A]
                        else: print A,B,C, "error"
                        if contains(sets,newset) == False:
                            sets.append(newset)
                            setCount += 1
    print sets
    print setCount, "sets"
    if setCount == 0:
        print "no set"

def menu():
    assembleDeck()
    deal(12)
    setCount = 0
    print " a: print all sets"
    print " b: count sets"
    print " c: redeal"
    print " d: check for sets"
    cmd = raw_input("Enter your choice: ")
    if cmd == "a":
        for a in range(12):
```

```
            for b in range(12):
                for c in range(12):
                    if (a!=b) and (b!=c) and (a!=c):
                        #print 'checking set'
                        if check(deck[dealt[a]], deck[dealt[b]], deck[dealt[c]]):
                            setCount += 1
                            print 'found set', deck[dealt[a]].ints(), deck[dealt[b]].ints(), dec
    elif cmd == "b":
        print setCount
    elif cmd == "c":
        deal(12)
        setCount = 0
    elif cmd == "d":
        for a in range(12):
            for b in range(12):
                for c in range(12):
                    if (a!=b) and (b!=c) and (a!=c):
                        #print 'checking set'
                        if check(deck[dealt[a]], deck[dealt[b]], deck[dealt[c]]):
                            setCount += 1
        if setCount > 0:
            print setCount, "sets"
        else: print "no set"

def iterate(times):
    assembleDeck()
    try:
        fout = open("out1.txt","w")
    except IOError, e:
        print "Error in file IO: ", e
    for k in range(times):
        undeal()
        deal(12)
        sets = []
        for a in range(12):
            for b in range(12):
                for c in range(12):
                    if (a!=b) and (b!=c) and (a!=c):
                        #print 'checking set'
                        A = dealt[a]
                        B = dealt[b]
                        C = dealt[c]
                        if check(deck[dealt[a]], deck[dealt[b]], deck[dealt[c]]):
                            if A<B and B<C:
                                newset = [A,B,C]
                            elif A<B and C<A:
                                newset = [C,A,B]
                            elif A<C and C<B:
                                newset = [A,C,B]
                            elif B<A and A<C:
                                newset = [B,A,C]
                            elif B<C and C<A:
                                newset = [B,C,A]
                            elif C<B and B<A:
                                newset = [C,B,A]
                            else: print A,B,C, "error"
                            if contains(sets,newset) == False:
```

```
                                sets.append(newset)
                                setCount += 1
        print k
        fout.write(str(setCount)+"\n")

    if fout:
        fout.close()
    print "all done"

def undeal():
    if len(dealt) > 0:
        for k in range(len(dealt)):
            dealt.remove(dealt[11-k])

deck, dealt = [],[]
setCount = -1
sets = []
main()
```

```
import random
import itertools

def contains(array, element):
    for j in range(len(array)):
        if array[j] == element:
            return True
    return False

def genDeck(dim):
        # make 3^dim empty strings
        # append each ternary dim-digit integer
        deck = []
        if dim > 0:
                for i in range(3):
                        if dim == 1:
                                deck.append(str(i))
                        if dim > 1:
                                for j in range(3):
                                        if dim == 2:
                                                deck.append(str(i)+str(j))
                                        if dim > 2:
                                                for k in range(3):
                                                        if dim == 3:
                                                                deck.append(str(i)+str(j)+str(k)
                                                        if dim > 3:
                                                                for l in range(3):
                                                                        if dim == 4:
                                                                                deck.append(str(
                                                                        if dim > 4:
                                                                                for m in range(3
                                                                                        if dim =

                                                                                        if dim >









        return deck

def check(A,B,C):
        for i in range(len(A)):
                if ((A[i] == B[i]) and (A[i] != C[i])):
                        return False
                elif ((A[i] == C[i]) and (A[i] != B[i])):
                        return False
                elif ((B[i] == C[i]) and (A[i] != B[i])):
                        return False
        return True

def makeSet(A,B):
    dim = len(A)
    C = ""
    i = 0
```

```
    while i < dim:
        print A[i], B[i]
        C = C + str(mod3(3-int(A[i]) -int(B[i])))
        i += 1
    return C

def mod3(x):
    return x % 3

def dealCard(deck,dealt,ind):
    newCard = deck[ind]
    if contains(dealt, newCard):
        dealCard(deck, dealt)
    else:
        dealt.append(newCard)
    return dealt

def assembleDeck(dim):
    for i in range(3**dim):
        deck.append(True)
    return deck


def deal(cards, deck, dealt):
        for i in range(cards):
                dealCard(deck, dealt, i)
    #print dealt
        setCount = 0
        return dealt

def main(cards, deck):
    #deck = genDeck(4)
    dealt = []
    sets = []
    deal(cards, deck, dealt)
#    chooseThree()
#    for k in dealt:
#        print k, deck[k].ints()
    for a in range(cards):
        for b in range(cards):
            for c in range(cards):
                if (a!=b) and (b!=c) and (a!=c):
                    #print 'checking set'
                    A = dealt[a]
                    B = dealt[b]
                    C = dealt[c]
                    if check(A,B,C):
                        if A<B and B<C:
                            newset = [A,B,C]
                        elif A<B and C<A:
                            newset = [C,A,B]
                        elif A<C and C<B:
                            newset = [A,C,B]
                        elif B<A and A<C:
                            newset = [B,A,C]
                        elif B<C and C<A:
                            newset = [B,C,A]
```

```
                        elif C<B and B<A:
                            newset = [C,B,A]
                        else: print A,B,C, "error"
                        if contains(sets,newset) == False:
                            sets.append(newset)
    print sets
    print len(sets), "sets"
    if setCount == 0:
        return True
    else: return False

def undeal():
    if len(dealt) > 0:
        for k in range(len(dealt)):
            dealt.remove(dealt[11-k])

def countSets(dealt):
        sets = []
        for a in range(len(dealt)):
            for b in range(len(dealt)):
                for c in range(len(dealt)):
                    if (a!=b) and (b!=c) and (a!=c):
                            #print 'checking set'
                            A = dealt[a]
                            B = dealt[b]
                            C = dealt[c]
                            if check(A,B,C):
                                    if A<B and B<C:
                                            newset = [A,B,C]
                                    elif A<B and C<A:
                                            newset = [C,A,B]
                                    elif A<C and C<B:
                                            newset = [A,C,B]
                                    elif B<A and A<C:
                                            newset = [B,A,C]
                                    elif B<C and C<A:
                                            newset = [B,C,A]
                                    elif C<B and B<A:
                                            newset = [C,B,A]
                                    else: print A,B,C, "error"
                                    if contains(sets,newset) == False:
                                            sets.append(newset)
        return len(sets)

def checkNoSet(small, newCard):
    for c in small:
        print makeSet(c, newCard)
        if contains(small, makeSet(c, newCard)):
            return False
    return True

def findMaxCap(dim):
        deck = genDeck(dim)
        cap = -1
        for n in range(3,(3**dim)):
            print n
            while cap < n:
```

```
                if (cap > 0) and (cap + 1 < n):
                    return cap
                for i in itertools.combinations(deck, n):
                        if cap == n:
                                pass
                        else:
                            print n, countSets(i)
                            if ((cap < n) and (countSets(i) == 0)):
                                cap = n
        return cap

def meta():
        for i in [4]:
                print i
                fout = file("maxcap4.txt","w")
                var = findMaxCap(i)
                print i, var
                fout.write(str(i)+str(var))
                fout.close()
        print "done"

def process(queue, deck):
    global rc
    rc += 1
    print "\n starting", rc
    small = queue.pop()
    print 'small=', small
    # small contains k cards with no set
    smind = deck.index(small[-1])
#   print 'smind =', smind
    for c in range(smind+1, len(deck)):
        #hand = []
        #for m in small:
        #   hand.append(m)
        #hand.append(deck[c])
        #print 'hand=', hand
        #if countSets(hand) == 0:
        #   queue.append(hand)
        if checkNoSet(small, deck[c]):
            hand = []
            for m in small:
                hand.append(m)
            hand.append(deck[c])
            queue.append(hand)
    print 'queue = ', queue
    if len(queue) > 0:
        return process(queue, deck)
    else:
        return len(small)


def primary(dim):
    queue = []
    deck = genDeck(dim)
    iters = itertools.combinations(deck, 2)
    for i in iters:
        queue.append(i)
```

```
    print process(queue, deck)
    print 'done'

rc = 0
primary(3)
```