

# **Combinatorics For Programmers**

**A Guide For Developers Looking To Improve Their Code**

**By Eliazar Matthew Nelson**

## **Table Of Contents**

**1.About This Book**

**2.What Is Combinatorics?**

**3Fundamentals of Programming**

**4.3D Graphics**

**5.Search Engines**

**6.Cryptography**

**7.Data Analysis**

**8.Computer Vision**

**9. Machine Learning**

## **About This Book**

Programming is one of the most interesting professions and hobbies of our time. It has created an entirely new industries and reinvigorated old industries. It has allowed geniuses such as Bill Gates and Mark Zuckerberg to make themselves billions at young ages. To understand our world it is essential to understand programming. To understand programming you need to understand the theory and the logic behind it. Much of that is going to involve the use of a certain field of mathematics known as combinatorics.

Essentially, combinatorics is the study of the combinations of finite sets of objects according to certain finite rules. Combinatorics has found usage in a variety of professions from hard sciences such as physics and chemistry to humanities such as linguistics. When we need to understand how to organize objects in an efficient manner this is how we are able to do so.

Many people have done some programming and may understand how to write quality code, but to truly separate yourself from the average coder you're going to need a strong understanding of the mathematics behind programming. How do binary search trees work? What is the logic behind computer graphics? What is behind a search engine? These questions are best explained through combinatorics.

Generally speaking, the study of combinatorics is something many people find intimidating and even many STEM students will try their best to avoid it. If you are the type that finds math intimidating or you are afraid you won't understand what's going on here don't worry. All you need is an understanding of algebra and statistics to understand what's going on here. Additionally, it is preferable to have some experience with coding javascript, C++, and PHP, but you don't need exceptional expertise to use this book. Last but not least it is important to have fun!

### **What You Will Need**

In order to create the examples in this book you're going to need to download certain libraries before hand. Among the libraries you'll be working with are

Languages Used:

PHP

Javascript

Python

C++

Numpy:

This is a Python library used by those who need to perform a number of analysis oriented tasks such as machine learning or computer vision. It provides support for very large matrices and arrays.

Webgl:

This is a library that takes some functions of OpenGL, an open source framework for computer graphics programming, and allows you to use them in Javascript.

Jsfeat: This is a library that allows you to use some computer vision functions in javascript as well. It isn't as deep as OpenCV but it gets the job done.

OpenCV: This is a C++ based computer vision library that helps you create programs designed to analyze visual information. It also contains some machine learning oriented functions that you may make use of if you have the desire to.

Biopython: Biopython is a library that allows you to use Python to search medical and scientific databases. It also allows you to perform certain tasks related to biological work such as determining the similarity of protein and DNA structure.

### **What Is Combinatorics?**

Combinatorics is essentially the mathematical study of the combinations of objects according to certain rules in finite sets. Combinatorics and its various subdisciplines may involve everything from probability to number theory. Like many fields of mathematics

combinatorics is frequently applied in many sciences and in the world of finance. Great examples of this can be found with ease. Combinatorial chemistry, for instance, has led to the discovery of entirely new drugs through the synthesis of many different combinations of various molecular groups.

To truly understand combinatorics you'll need to understand something known as set theory. Set theory is a way of organizing objects in a specific arrangement. For example, if you have Set A and it has five elements(a,b,c,d,e) you would write  $a \in A$  . This shows that a is an element of Set A.

When you're working with sets there are a number of things that you can do with sets. You can combine them together, you can isolate subsets, and you can work with specific elements. This table helps explain how each set operator works.

Operator	Meaning
$\in \ni$	Is an element of
$\cup$	Union Between
$\subseteq$	Is a subset of

$\notin$	Not element of
$\not\subseteq$	Not subset of
$\cap$	Intersection
$\supseteq$	Is A superset of
$\sqcap$	N-ary Intersection
$\sqcup$	N-ary Union

This may not seem to be clearly linked to programming, but much of what you do in programming is in fact linked to set theory. The concept of the database was originally based around set theory although array theory later created a superior means of designing databases. Even today array programming may make use of concepts in set theory.

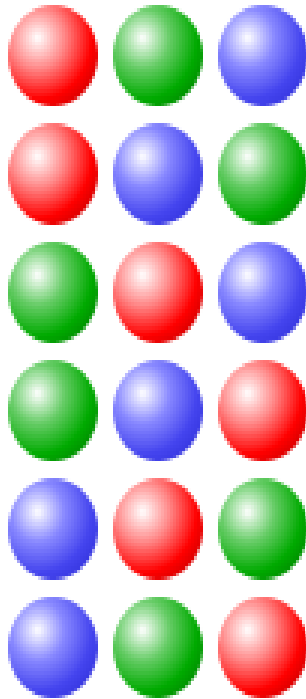
Within set theory a very important thing to understand is partitioning. **Partitions** are basically a way of turns sets into nonempty subsets. You can find examples of this in coding just about everywhere. For example, you might need to break down data for machine learning or you may find it useful for edge detection with working



with computer vision. The later would be very helpful for giving the computer the ability to understand how to detect faces using specific features.

Another area of importance for programmers is graph theory. Obviously, you'll have use for this if anything you want to make involves a graphical user interface. Since that covers quite few areas it's important to understand vertices, edges, and other key aspects of graph theory. From there you will learn more about rasters, meshes, and bitmaps.

Beyond set theory and graph theory you'll also find permutation, a key concept of combinatorics, is extremely important for improving your programming and making yourself a star coder. **Permutations** are possible ways you can arrange a set. The obvious potential uses of permutations in coding are plain as the light of day to see. When a you misspell a word and a word editor corrects you, that is an example of permutations at play. Properly securing a database may also involve the use of permutations to improve password security.



Permutations can be represented in a number of ways. You can represent them as a *word* or *function* depending on what you are planning to do. When you use a permutation as a word you are able to represent a set as a specific sequence. A word in this case does not necessarily mean what it might mean in the context of language. AABSDB is a word and so is 12453. However, depending on what you intend to use permutations for it can also be used to study the words within a formal language such as *cat* or *dog*.

When using words you can also use their inverses as well. So that 2 and  $\begin{smallmatrix} 1 & -1 & 2 \\ 2 & & 1 \end{smallmatrix}$  is possible. This makes permutations especially helpful for when you want to use them for the purpose of cryptography. We will be discussing that precise topic in a chapter covered in a

later section of this book!

Permutations may also be used to represent functions. A function in this case is simply an expression that represents the relationship between a number of variables. At this point you probably already worked on a number of functions through classes in algebra. In combinatorics the functions you create will generally produce more than one result. Most often you'll find yourself working with a continuous sequence of results from a specific function. Like calculus, combinatorics frequently deals with infinite series. We will use functions as a way of expressing the sum of those infinite series in a more convenient manner.

Much of the time when you are using combinatorics you are going to have to get familiar with the various constructions commonly used. These constructions are used to produce what are known as **generating functions**. Arriving out of partitioning of sets, a generating function is essentially the sum of an infinite series produced by applying certain rules. It is a way of compressing into a form that is easier to understand and more useful.

The infinite series produced by generating functions can be as simple as a single number or they may be involve something more interesting. Some generating functions produce fractions that go on for ever. You may already be familiar with some generating functions such as the generating function of the harmonic series.

In combinatorics the most basic generating functions you will be dealing with are known as **ordinary generating functions**. Essentially, you will be combining variables and coefficients in order to represent a sum of a series. Below is an example of one.

$$A_z = \sum_{n=0}^{\infty} A_n z^n$$

Although there are uses for ordinary generating functions, when you deal with labeled objects you might want to use **exponential generating functions**. Essentially, an exponential generating function expands by a power of X. There are few examples of them here.

$$A_z = \sum_{n \geq 0} A_n \frac{z^n}{|n|!} \bullet$$

There are situations where you might need to deal with things that aren't absolutely deterministic. You want to create a card game, generate textures for a 3D game, or design a predictive model using data. There are **probability generating functions** that will help you

create these combinations. Learn how to use them wisely because you'll need them if you're going to work with large amounts of data.

This is where **Probability Generating Functions** are useful. You can use them to help you understand combinations that involve randomness or something that simply isn't predictable. A probability generating function will look something like this.

$$G(z) = \sum_{x=0}^{\infty} p(x)z^x,$$

When you're working as a programmer you will probably find yourself dealing with something that requires you to deal with more than a single variable. There are in fact **bivariate generating functions** meant to help with this exact issue. They give you the ability to work with combinations involving two variables. A great example of this is here

$$\frac{1}{1 - (1+x)y}$$

A more complex version of this is the **multivariate**

**generating function.** These are especially helpful when you need to analyze data from a table. When you're dealing with computer vision and machine learning you might find yourself in need of them to better organize the information you'll be dealing with. See examples of multivariate generating functions [here](#). First an ordinary multivariate generating function

$$A(z, u) = \sum_{n, k} A_n k^{u^k} z^n$$

Now an exponential multivariate generating function.

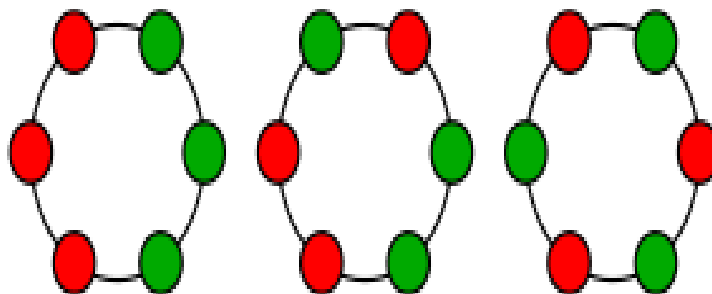
$$A(z, u) = \sum_{n, k} A_n k^{u^k} \frac{z^n}{n!}$$

There are more examples of generating functions but these are the most important examples of you'll be dealing with.

Programming is all about learning how to develop ways to systematically solve problems and how to probably organize information. Nothing is going to help you in this department more than the combinatorial convenience of **necklaces**. Essentially, when you have a necklace you have an equivalence class, or an arrangement of similar objects, of strings (sound familiar yet?) of  $n$  size within an alphabet of  $k$  size. The string is a sequence of elements found with the set which we call the alphabet. This can obviously include words in a dictionary but it may include shapes, colors, sounds, etc.

An alphabet might consist of a circle, a square, and a triangle. There are six strings possible in this situation as seen below you.

Those six strings combined are a necklace of 3 shapes.



Obviously we understand how strings are important to programming if you've ever had even a basic

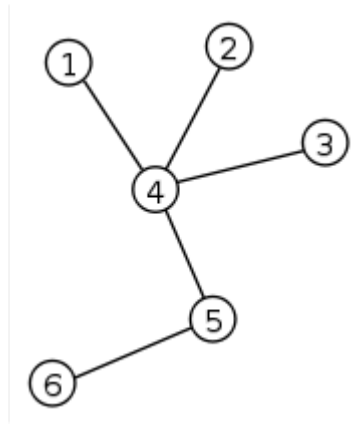
explanation of programming. Now you understand a little bit of the theory behind strings and you can now learn how to use that theory to better program.

Before we move on the fundamentals of programming one of the most important concepts you'll want to understand is that of trees. Descending from graph theory, trees are a great way to understand many abstract computer concepts and by extension many of the important concepts you'll be dealing with when you decide to improve your coding skills.

If you've ever taken a college course in algorithm theory you've likely already heard about trees and their importance to writing quality code. An idea borrowed from graph theory, trees help us understand many important concepts in coding. They are a sort of sequence designed in a way that prevents us from using them to create cycles. As long as two vertices are attaching to exactly one path you have a **tree**. A disconnected union of trees is called a forest. Typically, trees are undirected but there are rooted trees and these rooted trees may have directions. If the directions point away from the root of the tree it is **aborescence** or branching out. If they point towards the root it is **anti-aborescence**. You'll find this model is pretty useful for many of things you'll need to do.

Example of a tree here





These concepts cover the gist of what you'll need to learn in order to understand how to implement combinatorics in order to improve the quality of your code. There are few other concepts we'll discuss later to add upon the basics we learned here.

## **The Fundamentals Of Programming**

Now that you have an understanding of combinatorics we'll review the basics of programming . The applications of combinatorics are even easier to see when you look at the basics of programming. Queues, stacks, binary trees, arrays, loops, and everything in between has combinatorics just about every where. You just need to know where to look.

To begin to understand combinatorics in programming let's look at queues and stacks.

You already know that stacks are last in first out and queues are last out first in. What you might not know is that these concepts can clearly be understood through the lens of combinatorics. They are basically ways of turning a set orders into a specific sequence. Expanding upon what we know about combinatorics let's examine the peculiar case of the Fibonacci Heap.

The Fibonacci heap takes on a variety of obvious examples of combinatorics. Such as binary trees, the

use of big O notation to understand speed, and so much more. The min and max heaps essentially serve as specific sequences produced through generating functions used to create Fibonacci sequences. Finally, we use graph theory to provide a good representation of it. Combinatorics is absolutely essential to making faster code and more efficient code. Behind the scenes it appears just about everywhere and you'll find yourself using it to make everything run better.

While speed shows us many examples of how combinatorics can make us better coders, we can also see ways to improve our code by looking at examples of arrays and strings. As we saw in the previous chapter it is readily apparent that both concepts have ties to combinatorial concepts. Let's take this example of code written in Python.

```
<#!/usr/bin/python

import threading

import time

class myThread (threading.Thread):

def __init__(self, threadID, name,
counter):

threading.Thread.__init__(self)

self.threadID = threadID
```

```

self.name = name

self.counter = counter

def run(self):

    print "Starting " + self.name

    # Get lock to synchronize threads

    threadLock.acquire()

    print_time(self.name, self.counter, 3)

    # Free lock to release next thread

    threadLock.release()

def print_time(threadName, delay,
counter):

```

The threading concept and synchronization play right into something known in combinatorics as the sieve method. A sieve method is a way of splitting objects in such a way that we only need to deal with specific portions of the set. You lock the threads until there is a need to execute them. This improves the speed of your program because it allows the software to handle tasks one at a time.

Another area where you can clearly see the potential for combinatorics in programming is with the use of

strings. Strings, sequences of characters, are easily arranged through permutations. The alphabet is a great example of this.

There are 26 characters in the English alphabet. We can take the characters and place them into sequences we call words. Those words may make sense, apple for example, or they may be incoherent. To weed out incoherent words we can check them against a data set to see if they exist.

Even further down the line we can find ways to derive meaning from the words through their root words devoid of a suffix or prefix.

An extremely important concept for this book is the array structure. Oftentimes you'll find that applying combinatorics to software means finding a way to turn non-numerical data, such as natural language or colors, into something that a computer can process.

Near the end of this book you'll see an example of arrays in use with machine learning. In order to help the computer understand how to interpret data such as handwritten digits or the color of flower petals you need to represent that data in a numerical fashion. That enables the machine to understand what it to classify data properly.

One of the easiest ways to understand how combinatorics can be used in programming is random number generating. Here's an example of that below.

```
import random
```

```
import os
```

```
def main():
```

```
# winning_combo(lottery_numbers,  
power_ball)
```

```
generate_lotto_numbers(5)
```

```
def clear():
```

```
os.system('clear')
```

```
def pwr_ball():
```

```
lucky = random.randint(1, 85)
```

```
def pwr_ball2():  
  
    lucky2 = random.randint(4, 54)  
  
    def pwr_ball3():  
  
        lucky3 = random.randint(5, 31)  
  
        return lucky
```

```
def how_many_times_to_win(args, lucky):  
  
    # Call system clear  
  
    clear()  
  
    # Keep track of count  
  
    count = 1  
  
    # Infinite loop  
  
    while True:  
  
        clear()  
  
        # Display goal until user presses Enter
```

```
print("Here are your lucky lotto  
numbers: " + str(args) + " Power Ball: "  
+ str(lucky))
```

```
# Grab five non-repeating random numbers  
from x to y
```

```
lucky_digits = random.sample(range(23,  
64, 1), 5)
```

```
lucky_digits2 = random.sample(range(15,  
34, 1), 5)
```

```
lucky_digits3 = random.sample(range(6,  
49, 1), 5)
```

```
# Add Power Ball
```

```
power_ball = pwr_ball()
```

```
power_ball2 = pwr_ball2()
```

```
power_ball3 = pwr_ball3()
```

```
# Print out current attempt
```

```
print("You've just tried this " +
```



```

str(count) + " - " + str(lottery_num) +
" - Lucky number " + str(power_ball))

print("You've just tried this " +
str(count) + " - " + str(lottery_num) +
" - Lucky number " + str(power_ball2))

print("You've just tried this " +
str(count) + " - " + str(lottery_num) +
" - Lucky number " + str(power_ball3))
# Test if the amount of like numbers = 5

if len(set(lottery_num) & set(args)) ==
5 and power_ball == lucky:

print("It took " + str(count) + " times
to win the lottery!")

# Exit when found!

exit()

else:

count += 1

def generate_lotto_numbers(part):

```

```

correct_amount = part + 1

# Display lotto Numbers the amount of
times listed in the variable

for x in range(1, correct_amount):

print('Lottery Numbers: ' +
str(random.sample(range(1, 70, 1), 5)) +
' - Power Ball: ' + str(pwr_ball()))

def generate_lotto_numbers(part2):

correct_amount = part + 1

# Display lotto Numbers the amount of
times listed in the variable

for x in range(1, correct_amount):

print('Lottery Numbers 2: ' +
str(random.sample(range(1, 45, 1), 5)) +
' - Power Ball: ' + str(pwr_ball2()))

def generate_lotto_numbers(part3):

correct_amount = part + 1

# Display lotto Numbers the amount of
times listed in the variable

```

```
for x in range(1, correct_amount):

    print('Lottery Numbers 3: ' +
          str(random.sample(range(1, 80, 1), 5)) +
          ' - Power Ball: ' + str(pwr_ball3()))

    if part1==part2 v part1==part3 v
    part2==part3

    pause

main()
```

The code before you uses combinatorics to generate lottery numbers. Specifically, it uses combinatorial designs to create an algorithm that generates them randomly. There are other ways you could generate lottery numbers but this method stands out as exceptionally proficient.

In the example we just looked at we used partition theory to create a model for producing multiple winning lottery numbers and we made sure that those lottery numbers did not match up.

That example might seem trivial, but there are many

real world applications where you are going to want to make sure that your data isn't repeating itself. For example, if you were trying to track trends in share prices you would need to make sure you keep data organized according to date. Partitions are a perfect way to do just that.

With an understanding of how we're going to approach programming in this book let's start on one of our first topics.

## **First Example: Search Engines**

It may come as a surprise to some, but linguistics and computer science have had a strong relationship for quite some time. Linguists such as Noam Chomsky have used computer science to better understand language and combinatorics in particular has proved useful for linguists who want to better unknown languages.

Computer scientists have also used linguistics to provide some of the most important services we use today such as search engines, autocorrection, and translators. Much of this is achieved through the use of combinatorics to create a means for programmers to turn natural language into something that can be computed. In this chapter we will use combinatorics to help us understand how to create a search algorithm.

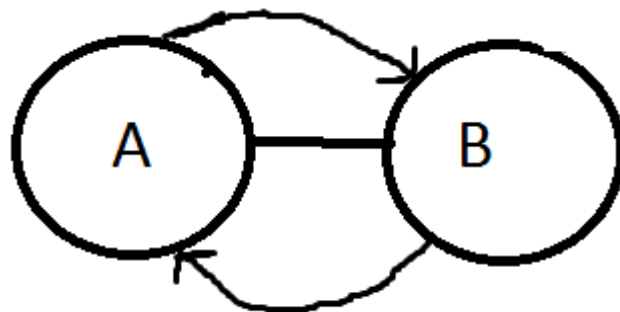
Two important concepts for you to understand is the concept of the alphabet and the word. The alphabet is the set of elements we will use to create the sequences we call words. Each element of the alphabet is referred

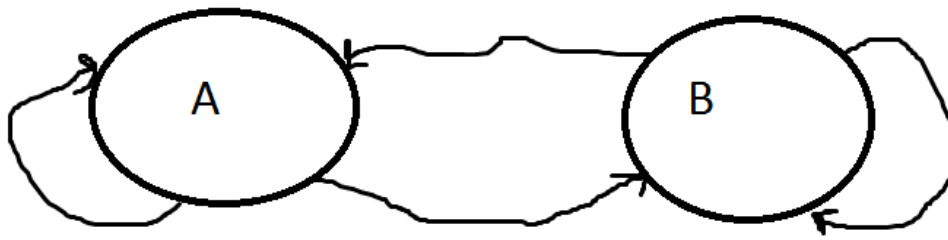
to as a letter. The letters in an alphabet do not have to necessarily be actual letters. They may be numbers, they be pictures, they maybe colors, and they may actually be long sentences. For example “Bird in the tree” can be a letter and so can “a” or “bc”.

ABABABAABAABAAAA

The concept of necklaces is also going to come up in this chapter too. We will find uses of necklaces to help make search engines more convenient and help users retrieve the content they consider most important.

Here are a few examples of automata for you to study.





Another concept we will be using very often is the idea of the unavoidable pattern. Essentially an unavoidable pattern is a pattern that will occur in a sequence if it is long enough. An avoidable sequence is a sequence that will occur in an infinite series of words but it will not actually match any of them. Obviously, one of the biggest issues when creating any search engine is making sure that the results given to your users are not irrelevant or pointless. If some is searching for “apps” you don't want to give them every result with “a” in it.

Now let's take a look at this example of a search engine in created in PHP. This example in particular will use unavoidable patterns and will allow us to optimize with automata.

First we start with the table

```
CREATE TABLE SEARCH_ENGINE (  
    `id` INT(11) NOT NULL  
    AUTO_INCREMENT,  
    `pageurl` VARCHAR(255) NOT  
    NULL,  
    `pagecontent` TEXT NOT NULL,  
    PRIMARY KEY (`id`))
```

Here we create our index file. We will treat the search

terms as sets and we match them to unavoidable patterns in our database.

```
<html>
    <head>
        <title> Your first
search engine </title>
    </head>
    <body>
        < form action =
'search.php' method = 'GET' >
            < center >
                <h1 > Your
first Search Engine </h1 >
                    < input
type = 'text' size='90' name =
'search' >
                                </ br >
                                </ br >
                                < input
type = 'submit' name = 'submit' value
= 'Search source code' >
                                    < option >
25 </ option >
                                    < option >
50 </ option >
                                    < option >
75 </ option >
                                </ center >
                            </ form >
                </ body >
</ html >
```



Here we create the database connection

```
mysql_connect ( "localhost",  
"USER_NAME", "PASSWORD" ) ;  
mysql_select_db  
( "DATABASE_NAME" );
```

The next block of code creates the query along with the tokens the user enters.

```
        $search_exploded = explode ( "  
", $search );  
        $x = 0;  
        foreach( $search_exploded as  
$search_each ) {  
            $x++;  
            $construct = " ";  
            if( $x == 1 )  
                $construct .=  
"keywords LIKE '%$search_each%' ";  
            else  
                $construct .=  
"AND keywords LIKE '%$search_each%'  
";  
        }  
        $construct = " SELECT * FROM  
SEARCH_ENGINE WHERE $construct ";  
        $run = mysql_query( $construct  
);
```

Here we receive the result and give it to the user. The results are presented as necklaces with only valid

necklaces displayed in the results.

```
if ($foundnum == 0)
    echo "Sorry, there are
no matching result for <b> $search
</b>."
    </ br >
    </ br > 1. Try using
more general words to improve your
search. for example: If you want to
search 'how to create a website' then
use general keyword like 'create'
'website'
    </ br > 2. Try different
words with similar meaning
    </ br > 3. Please check
your spelling";

else {
    echo
"$foundnum is in database !<p>";
    while
( $runrows =
mysql_fetch_assoc($run) ) {

$title = $runrows ['title'];

$description = $runrows
['description'];
$url = $runrows ['url'];

array preg_split (string pattern, string
```

```

string [, int limit [, int flags]]);

int preg_match (string pattern, string
string [, array pattern_array], [, int
$flags [, int $offset]]));

echo "<a href='$url'> <b> $title </b>
</a> <br> $desc <br> <a href='$url'>
$url </a> <p>";
    }
}

```

We can can apply our know how to help us build a decent web crawler as well. The crawler will use an algorithm based around necklaces in order to properly understand what it needs to do.

```

$file_handle = fopen( " Quantcast-
Top-Million.txt ", "r" );

    while ( !feof ( $file_handle )
) {
        $line =
fgets( $file_handle );

if( preg_match( '/^\d+/', $line ) )
{ # if it starts with some amount of
digits
        $tmp =
explode( "\t", $line );

```

```

$rank =
trim( $tmp[0] );
$url =
trim( $tmp[1] );
if( $url !=
'Hidden profile' ) { # Hidden profile
appears sometimes just ignore then
echo $
}
}
}
fclose( $file handle );

```

In that example we saw a few of the potential uses we may have for combinatorics in creating a search engine. Search engines don't just use natural language. In many situations you will want to use search engines for more specific purposes.

For example, let's say you are a biologist and you need to quickly access medical articles or research journals. You may want to derive specific information from these sources such as sequences of DNA or proteins, species names, and other important pieces of information. This is another area where you'll find combinatorics very powerful.

When biologists first took on the challenges of studying genetics they quickly discovered an ally in combinatorial mathematics. Essentially, every strand of DNA consist of a series of building blocks. In order to represent these strands of DNA we use the letters G, C, A, and T to represent them.

A Python library has been created specifically for this purpose. It is known as Biopython. Specifically created to help biologists and other scientists who need to work with sequenced information. Let's take this example of a distance tree calculator.

```
>>> from Bio.Phylo.TreeConstruction
import DistanceCalculator
>>> from Bio import AlignIO

>>> aln =
AlignIO.read('Tests/TreeConstruction/
msa.phy', 'phylip')

>>> print aln

SingleLetterAlphabet() alignment with
5 rows and 13 columns

AACGTGGCCACAT Alpha
AAGGTCGCCACAC Beta
GAGATTTCCGCCT Delta
GAGATCTCCGCCC Epsilon
CAGTTCGCCACAA Gamma

>>> calculator =
DistanceCalculator('identity')

>>> dm = calculator.get_distance(aln)
```

```
>>> dm

DistanceMatrix(names=['Alpha',
'Beta', 'Gamma', 'Delta', 'Epsilon'],
matrix=[[0], [0.23076923076923073,
0], [0.3846153846153846,
0.23076923076923073, 0],
[0.5384615384615384,
0.5384615384615384,
0.5384615384615384, 0],
[0.6153846153846154,
0.3846153846153846,
0.46153846153846156,
0.15384615384615385, 0]])

>>> print(dm)

Alpha      0
Beta       0.230769230769    0
Gamma      0.384615384615
0.230769230769    0
Delta      0.538461538462
0.538461538462    0.538461538462    0
```

```
Epsilon 0.615384615385
0.384615384615 0.461538461538
0.153846153846 0
      Alpha   Beta   Gamma   Delta
Epsilon
```

Here we see that we are using single letter alphabets and comparing the sequences to understand how much they match. Once we receive a decent idea about how similar each one is to the other 4 we can then determine the genetic distance of the matrix. The genetic distance is determined by automata.

## **3D Graphics And Combinatorics**



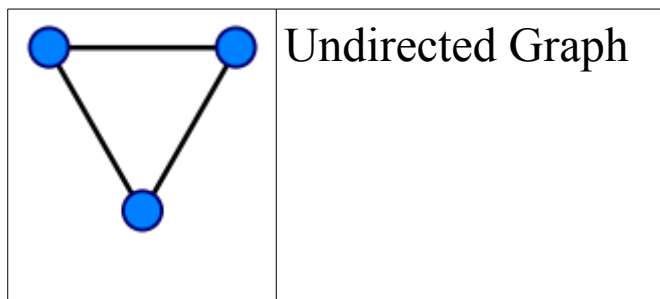
Graph theory is probably one of the most obvious ways that combinatorics is applicable in programming. Obviously to understand how to become a better graphics programmer you'll need to understand graph theory. Once you learn how combinatorics allows

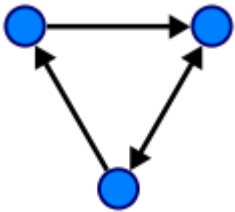
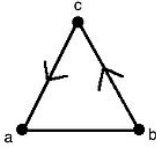
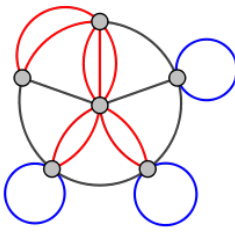
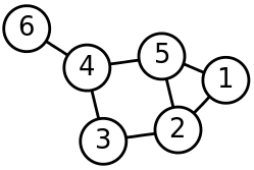


things such as meshes and polygon counts to work you'll know how to tackle potential issues in programming much better.

When we think of graphs we often think of a board room full of working stiffs discussing quarterly profits. In reality a graph is any visualization that involves various points, known as vertices, and the connections between them known as edges. This allows us to understand the relationship between objects. You have seen and used graphs for most of your life in one form or another.

There are different types of graphs and you can use them to represent different problems. If you want to understand the flow of liquid between various pipes in a sewage system you may use a directed graph to represent the relationship between the pipelines and the various portions of the waste removal system. However, you would not need to do this for something that does not require you to input direction such as a map of a city or pieces of a jigsaw puzzle.

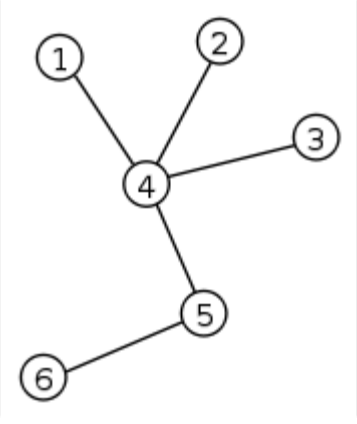
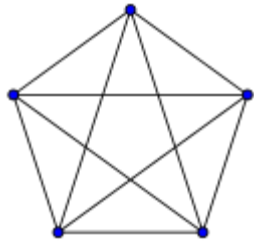
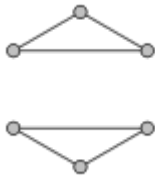
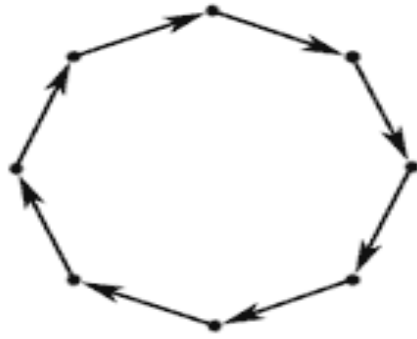


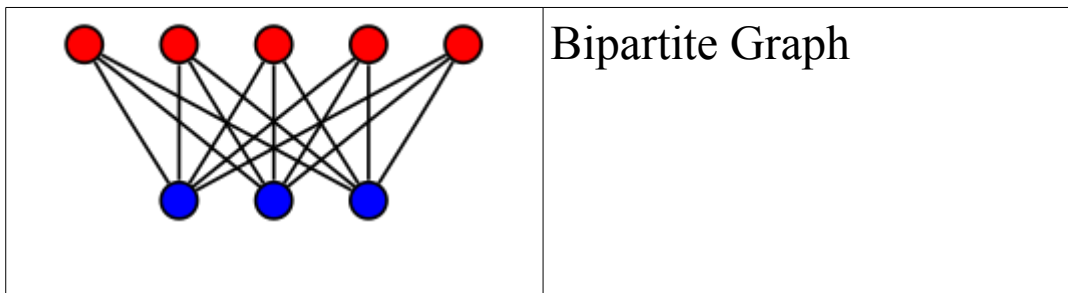
	Directed Graph
	Mixed Graph
	Multigraph
	Labeled Graph

Please make yourself familiar with these various types of graphs. We will be using them in the coding examples used in this book.

After making yourself familiar with the various types of graphs you might want to make yourself familiar with important classes of graphs. These are sets of specific graph types designed for certain purposes. Earlier on we used a special class of graphs known as binary trees to help understand some aspects of

programming. These classes of graphs are found in the table below.

	Tree
	Complete Graph
	Regular Graph
	Cycle Graph



With that in mind let's take a look at this WebGL javascript example. We will be designing a visual display of a Mandelbrot fractal.

```
<!doctype html>
```

```
<html>
```

```
<head>
```

```
<meta charset="utf-8">
```

```
<title>Mandelbrot set</title>
```

```
<script id="vertex-shader" type="x-shader/x-vertex">
```

```
attribute vec2 vertex_position;
```

```
varying vec2 fragment_position;
```

```
uniform vec2 offset;
```

```
uniform float zoom;
```

```
void main()
```

```
{
```

```
gl_Position = vec4(vertex_position, 0.0, 1.0);
```

```
fragment_position = zoom * vertex_position + offset;
```

```
}
```

```
</script>
```

```
<script id="argb-srgb" type="x-shader/x-fragment">
```

```
vec4 argb(float r, float g, float b)
```

```
{
```

```
    r = clamp(r, 0.0, 1.0);
```

```
    g = clamp(g, 0.0, 1.0);
```

```
    b = clamp(b, 0.0, 1.0);
```

```
    float K0 = 0.03928;
```

```
    float a = 0.055;
```

```
    float phi = 12.92;
```

```
    float gamma = 2.4;
```

```
    r = r <= K0 / phi ? r * phi : (1.0 + a) * pow(r, 1.0 /  
    gamma) - a;
```

```
    g = g <= K0 / phi ? g * phi : (1.0 + a) * pow(g, 1.0 /  
    gamma) - a;
```

```
    b = b <= K0 / phi ? b * phi : (1.0 + a) * pow(b, 1.0 /  
    gamma) - a;
```

```
    return vec4(r, g, b, 1.0);
```

```
}
```

```
</script>
```

This is where we define the variables we are going to use in this example. We want to first use an exponential generating function and then we will take those results and use them to create the fractal by placing them through the fractal generating function.

```
<script id="argb-linear" type="x-shader/x-fragment">
```

```
vec4 argb(float r, float g, float b)
```

```
{
```

```
return vec4(r, g, b, 1.0);
```

```
}
```

```
</script>
```

```
<script id="inside" type="x-shader/x-fragment">
```

```
float inside(vec2 z)
```

```
{
```

```
return ((z[0] <= 0.0 || 0.0 <= z[0]) && (z[1] <= 0.0 ||  
0.0 <= z[1]) && dot(z, z) < 4.0) ? 1.0 : 0.0;
```

```
}
```

```
</script>
```

```
<script id="sane-code" type="x-shader/x-fragment">
```

```
float inside(vec2 z)
```

```
{
```

```
return dot(z, z) < 4.0 ? 1.0 : 0.0;
```

```
}
```

```
</script>
```



```
<script id="fragment-shader" type="x-shader/x-fragment">
```

```
varying vec2 fragment_position;
```

```
vec2 cmul(vec2 a, vec2 b)
```

```
{
```

```
return vec2(a[0] * b[0] - a[1] * b[1], a[1] * b[0] + a[0] * b[1]);
```

```
}
```

This is where we'll get the colorful display that we want to use. Let's give them something bright and colorful to look at.

```
vec4 rainbow(float v)
```

```
{
```

```
float r = 6.0 * abs(v - 3.0 / 6.0) - 1.0;
```

```
float g = 2.0 - 6.0 * abs(v - 2.0 / 6.0);
```

```
float b = 2.0 - 6.0 * abs(v - 4.0 / 6.0);
```

```
return argb(r, g, b);
```

```
}
```

```
void main()
```

```
{
```

```
vec2 z = vec2(0.0, 0.0);
```

```
vec2 c = fragment_position;
```

```
float counter = 0.0;
```

```
for (int i = 0; i < 100; i++) {
```

```
    counter += inside(z);
```

```
    z = cmul(z, z) + c;
```

```
}
```

```
gl_FragColor = counter < 100.0 ? rainbow(0.01 *  
counter) : vec4(0.0, 0.0, 0.0, 1.0);
```

```
// gl_FragColor = rainbow(fragment_position[0]);
```

```
}
```

```
</script>
```

```
<script type="text/javascript">
```

```
var canvas, gl, vertex_position, quad_buffer;
```

```
var uniform_zoom, uniform_offset;
```

```
var zoom = 2.0, off_x = -0.5, off_y = 0.0;
```

```
function draw()
```

```
{
```

```
gl.clear(gl.COLOR_BUFFER_BIT);
```

```
gl.uniform2f(uniform_offset, off_x, off_y);
```

```
gl.uniform1f(uniform_zoom, zoom);
```

```
gl.bindBuffer(gl.ARRAY_BUFFER, quad_buffer);
```

```
gl.vertexAttribPointer(vertex_position, 2, gl.FLOAT,  
false, 0, 0);
```

```
gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
```

```
}
```

```
function mouse_down(event)
```

```
{
```

```
var x = 2.0 * (event.pageX - canvas.offsetLeft) /  
canvas.width - 1.0;
```

```
var y = 1.0 - 2.0 * (event.pageY - canvas.offsetTop) /  
canvas.height;
```

```
off_x += x * zoom;
```

```
off_y += y * zoom;
```

```
zoom /= 2.0;
```

```
draw();
```

```
}
```

```
function compile()
```

```
{
```

```
var vertex_shader =  
gl.createShader(gl.VERTEX_SHADER);
```

```
gl.shaderSource(vertex_shader,  
document.getElementById("vertex-shader").text);
```

```
gl.compileShader(vertex_shader);
```

```
if (!gl.getShaderParameter(vertex_shader,  
gl.COMPILE_STATUS)) {
```

```
alert("vertex shader:\n" +  
gl.getShaderInfoLog(vertex_shader));
```

```
return;
```

```
}
```

```
var fragment_shader =  
gl.createShader(gl.FRAGMENT_SHADER);
```

```
var source = "precision highp float;\n";
```

```
if (document.getElementById("workarounds-  
toggle").innerHTML == "disable")
```

```
source +=  
document.getElementById("workarounds").text;
```

```
else
```

```
source += document.getElementById("sane-  
code").text;
```

```
if (document.getElementById("srgb-  
toggle").innerHTML == "disable")
```

```
source += document.getElementById("argb-srgb").text;
```

```
else
```

```
source += document.getElementById("argb-  
linear").text;
```

```
source += document.getElementById("fragment-  
shader").text;
```

```
gl.shaderSource(fragment_shader, source);
```

```
gl.compileShader(fragment_shader);
```

```
if (!gl.getShaderParameter(fragment_shader,  
gl.COMPILE_STATUS)) {
```

```
    alert("fragment shader:\n" +  
    gl.getShaderInfoLog(fragment_shader));
```

```
return;
```

```
}
```

```
var program = gl.createProgram();

gl.attachShader(program, vertex_shader);

gl.attachShader(program, fragment_shader);

gl.linkProgram(program);


if (!gl.getProgramParameter(program,
gl.LINK_STATUS)) {

    alert("linker error:\n" +
gl.getProgramInfoLog(program));

    return;

}

gl.useProgram(program);
```



```
vertex_position = gl.getAttributeLocation(program,  
"vertex_position");
```

```
gl.enableVertexAttribArray(vertex_position);
```

```
uniform_offset = gl.getUniformLocation(program,  
"offset");
```

```
uniform_zoom = gl.getUniformLocation(program,  
"zoom");
```

```
}
```

```
function start()
```

```
{
```

```
canvas = document.getElementById("canvas");
```

```
gl = canvas.getContext("webgl fractal");
```

```
if (!gl)
```

```
gl = canvas.getContext("experimental-webgl fractal");
```

```
if (!gl) {  
  
    alert("could not get webgl context");  
  
    return;  
  
}
```

```
gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

For this exercise we're going to use a star to define what we want.

```
quad_buffer = gl.createBuffer();
```

```
gl.bindBuffer(gl.ARRAY_BUFFER, quad_buffer);
```

```
var vertices = [ 1.0, 1.0, -1.0, 1.0, 1.0, -1.0, -1.0, -1.0 ];
```

```
gl.bufferData(gl.ARRAY_BUFFER, new  
Float32Array(vertices), gl.STATIC_DRAW);
```

```
compile();
```

```
draw();
```

```
canvas.addEventListener("mousedown", mouse_down,  
false);
```

```
}
```

```
function toggle(button)
```

```
{
```

```
if (button.innerHTML == "disable") {
```

```
button.innerHTML = "enable";
```

```
} else {
```

```
button.innerHTML = "disable";
```

```
}
```

```
compile();
```

```
draw();
```

```
}
```

```
</script>
```

```
</head>
```

```
<body onload="start();" style="color: silver;  
background-color: black">
```

```
<p>toggle srgb correction: <button id="srgb-toggle"  
onclick="toggle(this);">disable</button>
```

```
toggle workarounds: <button id="workarounds-toggle"  
onclick="toggle(this);">enable</button></p>
```

```
<canvas id="canvas" style="border: none; cursor:  
default;" width="512" height="512">
```

WebGL not enabled?

</canvas>

<footer><small>

mandelbrot.html - visualize Mandelbrot set using  
GLSL in WebGL<br />

Written in 2013 by &lt;Ahmet Inan>  
&lt;xdsopl@googlemail.com><br />

</body>

Certain things stand out in that example and we need to look at how our example of a beautiful visual display used combinatorics and where that added know how helped make or coding much better than it would have been without it. First lets go over some of the mathematics we just did.

```
r = r <= K0 / phi ? r * phi : (1.0 + a) * pow(r,  
1.0 / gamma) - a;
```

Here see a formula of

$$r \leq \frac{K0}{\phi} \forall r \bullet \phi : (1+a) \bullet r^{1/\gamma} - a$$

This exercise used both combinatorics and something known as complex analysis. Before we do anything we used combinatorics to create an efficient function. After we did that we applied analytical techniques to produce the Mandelbrot fractal. There are other ways we could have done this. We could also have used a probability generating function to place into the Mandelbrot fractal. Such as the one here.

## **Using Combinatorics For Cryptography**

Cryptography is one area of programming where your experience with combinatorics will pay off big time. Good security requires you to make it as challenging as possible for a hacker to find a way to break into your database or invade your privacy.

A particularly useful subdiscipline of combinatorics we will use here is Ramsey theory. Ramsey theory is concerned with how many elements of a particular structure there must be in order to ensure that certain

properties are present. You'll use it to make sure the encryption algorithm is holding up to the quality we need.

Let's begin with crypto.js. First we need to include the necessary packages.

```
require.config({
  packages: [
    {
      name: 'crypto-js',
      location: 'path-
to/bower_components/crypto-js',
      main: 'index'
    }
  ]
});

require(["crypto-js/aes", "crypto-
js/sha256"], function (AES, SHA256) {
  console.log(SHA256("Message"));
});
```



There are two basic types of encryption we'll be dealing with. We will deal with text encryption and we will deal with object encryption. Here's an example of text encryption.

```
var CryptoJS = require("crypto-js");

// Encrypt
var ciphertext =
CryptoJS.AES.encrypt('my message',
'secret key 123');

// Decrypt
var bytes =
CryptoJS.AES.decrypt(ciphertext.toString
(), 'secret key 123');
var plaintext =
bytes.toString(CryptoJS.enc.Utf8);

console.log(plaintext);
```

An example of object encryption.

```
var CryptoJS = require("crypto-js");

var data = [{id: 1}, {id: 2}]

// Encrypt
var ciphertext =
CryptoJS.AES.encrypt(JSON.stringify(data
), 'secret key 123');

// Decrypt
var bytes =
CryptoJS.AES.decrypt(ciphertext.toString
()), 'secret key 123');
var decryptedData =
JSON.parse(bytes.toString(CryptoJS.enc.U
tf8));

console.log(decryptedData);
```

```
bool openssl_public_encrypt ( string $da
ta , string &$scripted , mixed $key [, in
t $padding = OPENSSL_PKCS1_PADDING ] )
```

To start off we begin with the example of the RSA public key system. Essentially, the encryption key is public but the decryption key is kept secret. Below you is a generic example of the RSA public key system.

```
function ssl_encrypt($source,$type,$key){
//Assumes 1024 bit key and encrypts
in chunks.

$maxlength=117;
$output='';
while($source){
    $input=
substr($source,0,$maxlength);
    $source=substr($source,$maxlength);
    if($type=='private'){
        $ok=
openssl_private_encrypt($input,
$encrypted,$key);
    }else{
        $ok=
openssl_public_encrypt($input,
$encrypted,$key);
    }

    $output.=$encrypted;
}
return $output;
```

```

}
function ssl_decrypt($source,$type,
$key){
// The raw PHP decryption functions
appear to work
// on 128 Byte chunks. So this
decrypts long text
// encrypted with ssl_encrypt().

$maxlength=128;
$output='';
while($source){
    $input=
substr($source,0,$maxlength);
    $source=substr($source,$maxlength);
    if($type=='private'){
        $ok=
openssl_private_decrypt($input,$out,
$key);
    }else{
        $ok=
openssl_public_decrypt($input,$out,
$key);
    }

    $output.=$out;
}
return $output;
}

```

This is somewhat useful in and of itself, but there is more we could do. Algebraic combinatorics gives us certain tools through the use of rings, essentially sets with either addition or multiplication operations, and modules, a sort of ring to put other rings in with a multiplication operation, to create something stronger than this. Before we place anything through the RSA system we will use a module to produce another layer.

Cryptography covers a wide array of issues. We use it when we send emails, governments use it to avoid having sensitive information exposed, and it is used to protect digital copyrights. A very important for of cryptography is something known as secret sharing. Basically, secret sharing is a way to kept information within a certain group and to keep others from gaining access to it. Below is a coding example of secret sharing. First we'll start with defining the basics of what we'll be working with.

```
(function(exports, global){  
  var defaults = {  
    bits: 8, // default number of bits  
    radix: 16, // work with HEX by default  
    minBits: 3,  
    maxBits: 20, // this permits 1,048,575 shares,  
    though going this high is NOT recommended in JS!  
  
    bytesPerChar: 2,
```

```

    maxBytesPerChar: 6, // Math.pow(256,7) >
    Math.pow(2,53)

    // Primitive polynomials (in decimal form) for
    // Galois Fields GF(2^n), for 2 <= n <= 30
    // The index of each term in the array corresponds
    // to the n for that polynomial
    // i.e. to get the polynomial for n=16, use
    primitivePolynomials[16]
    primitivePolynomials:
    [null,null,1,3,3,5,3,3,29,17,9,5,83,27,43,3,45,9,39,39,9,
    5,3,33,27,9,71,39,9,5,83],

    // warning for insecure PRNG
    warning: 'WARNING:\nA secure random number
    generator was not found.\nUsing Math.random(), which
    is NOT cryptographically strong!'
};

```

Here we construct the tables we'll need for multiplication.

```

var logs = [], exps = [], x = 1, primitive =
defaults.primitivePolynomials[config.bits];
for(var i=0; i<config.size; i++){
    exps[i] = x;
    logs[x] = i;
    x <<= 1;
    if(x >= config.size){
        x ^= primitive;
        x &= config.max;
    }
}

```

```

    }

    config.logs = logs;
    config.exps = exps;
};

```

This creates a random number generator for us to use.

```

// A totally insecure RNG!!! (except in Safari)
// Will produce a warning every time it is called.
config.unsafePRNG = true;
warn();

var bitsPerNum = 32;
var max = Math.pow(2,bitsPerNum)-1;
return function(bits){
    var elems = Math.ceil(bits/bitsPerNum);
    var arr = [], str=null;
    while(str===null){
        for(var i=0; i<elems; i++){
            arr[i] = Math.floor(Math.random()
* max + 1);
        }
        str = construct(bits, arr, 10, bitsPerNum);
    }
    return str;
};
};

```

This will create a random bits length for the information we are going to encrypt.

This is the part where we get to the meat of the secret sharing. You'll finally take all those random numbers and turn them into something you can use.

```
// Divides a `secret` number String str expressed in
radix `inputRadix` (optional, default 16)
// into `numShares` shares, each expressed in radix
`outputRadix` (optional, default to `inputRadix`),
// requiring `threshold` number of shares to reconstruct
the secret.
// Optionally, zero-pads the secret to a length that is a
multiple of padLength before sharing.
/** @expose */
exports.share = function(secret, numShares, threshold,
padLength, withoutPrefix){
    if(!isInitiated()){
        this.init();
    }
    if(!isSetRNG()){
        this.setRNG();
    }

    padLength = padLength || 0;

    if(typeof secret !== 'string'){
        throw new Error('Secret must be a string.');
```



```

        throw new Error('Number of shares must be
an integer between 2 and 2^bits-1 (' + config.max + '),
inclusive.')
    }
    if(numShares > config.max){
        var neededBits =
Math.ceil(Math.log(numShares + 1)/Math.LN2);
        throw new Error('Number of shares must be
an integer between 2 and 2^bits-1 (' + config.max + '),
inclusive. To create ' + numShares + ' shares, use at
least ' + neededBits + ' bits.')
    }
    if(typeof threshold !== 'number' || threshold%1 !
== 0 || threshold < 2){
        throw new Error('Threshold number of shares
must be an integer between 2 and 2^bits-1 (' +
config.max + '), inclusive. ');
    }
    if(threshold > config.max){
        var neededBits =
Math.ceil(Math.log(threshold + 1)/Math.LN2);
        throw new Error('Threshold number of shares
must be an integer between 2 and 2^bits-1 (' +
config.max + '), inclusive. To use a threshold of ' +
threshold + ', use at least ' + neededBits + ' bits. ');
    }
    if(typeof padLength !== 'number' || padLength
%1 !== 0 ){
        throw new Error('Zero-pad length must be an
integer greater than 1. ');
    }

```

Then you apply the polynomial generator

```
exports._getShares = function(secret, numShares,
threshold){
    var shares = [];
    var coeffs = [secret];

    for(var i=1; i<threshold; i++){
        coeffs[i] = parseInt(config.rng(config.bits),2);
    }
    for(var i=1, len = numShares+1; i<len; i++){
        shares[i-1] = {
            x: i,
            y: horner(i, coeffs)
        }
    }
    return shares;
};
```

We then apply Horner's method to the polynomial

```
function horner(x, coeffs){
    var logx = config.logs[x];
    var fx = 0;
    for(var i=coeffs.length-1; i>=0; i--){
        if(fx === 0){
            fx = coeffs[i];
            continue;
        }
        fx = config.exps[ (logx + config.logs[fx]) %
config.max ] ^ coeffs[i];
    }
}
```

```
    return fx;
};
```

```
function inArray(arr,val){
    for(var i = 0,len=arr.length; i < len; i++) {
        if(arr[i] === val){
            return true;
        }
    }
    return false;
};
```

```
function processShare(share){

    var bits = parseInt(share[0], 36);
    if(bits && (typeof bits !== 'number' || bits%1 !==
0 || bits<defaults.minBits || bits>defaults.maxBits)){
        throw new Error('Number of bits must be an
integer between ' + defaults.minBits + ' and ' +
defaults.maxBits + ', inclusive.')
    }

    var max = Math.pow(2, bits) - 1;
    var idLength = max.toString(config.radix).length;

    var id = parseInt(share.substr(1, idLength),
config.radix);
    if(typeof id !== 'number' || id%1 !== 0 || id<1 ||
id>max){
        throw new Error('Share id must be an integer
between 1 and ' + config.max + ', inclusive.');
```

```

share = share.substr(idLength + 1);
    if(!share.length){
        throw new Error('Invalid share: zero-length
share.')
    }
    return {
        'bits': bits,
        'id': id,
        'value': share
    };
};

exports._processShare = processShare;

```

Now we want to evaluate the polynomials for the bits length segments of the shares.

```

// Protected method that evaluates the Lagrange
interpolation
// polynomial at x=`at` for individual config.bits-length
// segments of each share in the `shares` Array.
// Each share is expressed in base `inputRadix`. The
output
// is expressed in base `outputRadix'
function combine(at, shares){
    var setBits, share, x = [], y = [], result = "", idx;

    for(var i=0, len = shares.length; i<len; i++){
        share = processShare(shares[i]);
        if(typeof setBits === 'undefined'){
            setBits = share['bits'];

```

```

    }else if(share['bits'] !== setBits){
        throw new Error('Mismatched shares:
Different bit settings.')
    }

    if(config.bits !== setBits){
        init(setBits);
    }

    if(inArray(x, share['id'])){ // repeated x
value?
        continue;
    }

    idx = x.push(share['id']) - 1;
    share = split(hex2bin(share['value']));
    for(var j=0, len2 = share.length; j<len2; j++){
        y[j] = y[j] || [];
        y[j][idx] = share[j];
    }
}

for(var i=0, len=y.length; i<len; i++){
    result = padLeft(lagrange(at, x,
y[i]).toString(2)) + result;
}

if(at===0){// reconstructing the secret
    var idx = result.indexOf('1'); //find the first 1
    return bin2hex(result.slice(idx+1));
}else{// generating a new share
    return bin2hex(result);
}

```

```

    }
};

// Combine `shares` Array into the original secret
/** @expose */
exports.combine = function(shares){
    return combine(0, shares);
};

```

Now we have the id generator at work here.

```

// Generate a new share with id `id` (a number between
1 and 2^bits-1)
// `id` can be a Number or a String in the default radix
(16)
/** @expose */
exports.newShare = function(id, shares){
    if(typeof id === 'string'){
        id = parseInt(id, config.radix);
    }

    var share = processShare(shares[0]);
    var max = Math.pow(2, share['bits']) - 1;

    if(typeof id !== 'number' || id%1 !== 0 || id<1 ||
id>max){
        throw new Error('Share id must be an integer
between 1 and ' + config.max + ', inclusive.');
```

```

    }

    var padding = max.toString(config.radix).length;

```

```

        return config.bits.toString(36).toUpperCase() +
        padLeft(id.toString(config.radix), padding) +
        combine(id, shares);
    };

```

Now we'll put the hex functions in

```

// Pads a string `str` with zeros on the left so that its
// length is a multiple of `bits`
function padLeft(str, bits){
    bits = bits || config.bits
    var missing = str.length % bits;
    return (missing ? new Array(bits - missing +
    1).join('0') : "") + str;
};

```

```

function hex2bin(str){
    var bin = "", num;
    for(var i=str.length - 1; i>=0; i--){
        num = parseInt(str[i], 16)
        if(isNaN(num)){
            throw new Error('Invalid hex character.')
        }
        bin = padLeft(num.toString(2), 4) + bin;
    }
    return bin;
}

```

```

function bin2hex(str){
    var hex = "", num;
    str = padLeft(str, 4);
    for(var i=str.length; i>=4; i-=4){

```

```

        num = parseInt(str.slice(i-4, i), 2);
        if(isNaN(num)){
            throw new Error('Invalid binary
character.')
        }
        hex = num.toString(16) + hex;
    }
    return hex;
}

```

This creates a character string for the hex representation.

```

// Converts a given UTF16 character string to the HEX
representation.
// Each character of the input string is represented by
// `bytesPerChar` bytes in the output string.
/** @expose */
exports.str2hex = function(str, bytesPerChar){
    if(typeof str !== 'string'){
        throw new Error('Input must be a character
string.');
```

```

    }
    bytesPerChar = bytesPerChar ||
defaults.bytesPerChar;

    if(typeof bytesPerChar !== 'number' ||
bytesPerChar%1 !== 0 || bytesPerChar<1 ||
bytesPerChar > defaults.maxBytesPerChar){
        throw new Error('Bytes per character must be
an integer between 1 and ' + defaults.maxBytesPerChar
+ ', inclusive.')
```



```

    }

    var hexChars = 2*bytesPerChar;
    var max = Math.pow(16, hexChars) - 1;
    var out = "", num;
    for(var i=0, len=str.length; i<len; i++){
        num = str[i].charCodeAt();
        if(isNaN(num)){
            throw new Error('Invalid character: ' +
str[i]);
        }else if(num > max){
            var neededBytes =
Math.ceil(Math.log(num+1)/Math.log(256));
            throw new Error('Invalid character code
(' + num + '). Maximum allowable is 256^bytes-1 (' +
max + '). To convert this character, use at least ' +
neededBytes + ' bytes.')
        }else{
            out = padLeft(num.toString(16),
hexChars) + out;
        }
    }
    return out;
};

```

Finally we convert the hex numbers into character strings.

```

// Converts a given HEX number string to a UTF16
character string.
/** @expose */

```

```

exports.hex2str = function(str, bytesPerChar){
    if(typeof str !== 'string'){
        throw new Error('Input must be a
hexadecimal string.');
```

```

    }
    bytesPerChar = bytesPerChar ||
defaults.bytesPerChar;

    if(typeof bytesPerChar !== 'number' ||
bytesPerChar%1 !== 0 || bytesPerChar<1 ||
bytesPerChar > defaults.maxBytesPerChar){
        throw new Error('Bytes per character must be
an integer between 1 and ' + defaults.maxBytesPerChar
+ ', inclusive.')
```

```

    }

    var hexChars = 2*bytesPerChar;
    var out = "";
    str = padLeft(str, hexChars);
    for(var i=0, len = str.length; i<len; i+=hexChars){
        out =
String.fromCharCode(parseInt(str.slice(i,
i+hexChars),16)) + out;
    }
    return out;
};

// by default, initialize without an RNG
exports.init();
})(typeof module !== 'undefined' && module['exports']
? module['exports'] : (window['secrets'] = {}), typeof
GLOBAL !== 'undefined' ? GLOBAL : window );
```

## **Understanding Data Analysis**

Most programmers are probably not interested in understanding how to use programming for data analysis but there are a number of things you can learn from this chapter that might just help you. Data analysis introduces you to concepts that can help produce more efficient algorithms. For the creatively inclined data analysis techniques are useful for creative endeavors. You'll probably find something in here to make a better app.

Much of what we'll be discussing in this chapter comes

from **complex analysis**. In complex analysis you will find your dealing with so **complex numbers**, or numbers involving both real and imaginary numbers. An example of a complex number is  $2+3i$ .

Once of the most interesting ones will be dealing with are **harmonic functions**. Essentially a function is harmonic if it satisfies the Laplace Equation and it has continuous second partials covering a specific region. Harmonic functions are great for modeling stochastic, or random, processes which makes them perfect for times when you need to model stock prices, consumer decisions, and other phenomena of that nature. Here is a great example of a Harmonic function at work.



Now we can move on to the subject of power series. A power series takes the form of

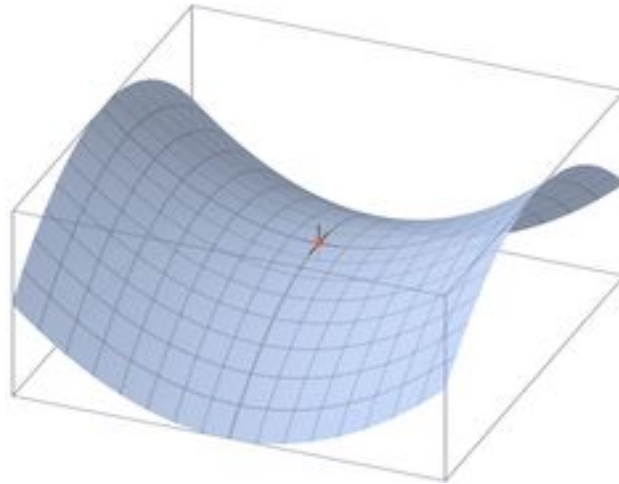
Where  $a_n$  represents the coefficient of the  $n$ th term and  $c$  is a constant. Later on you'll find that power series give rise to many other important concepts such as Taylor series and Laurent series. A Taylor series represents a

function as the sum of the terms based on the derivative of the function at a specific point. They usually look something like this

Taylor series are used very often in quantitative finance and other situations where you might need to come up with multiple possible projections. Each degree gives you a new idea about you need to do. Another type of power series you'll find yourself using quite a bit is the Laurent series.

The Laurent series is focused around the idea of a power series with negative degrees. You find combining this with combinatorics is a great way to work with programming software with applications in geography, in finance, in urban planning, architecture, and other areas. The Taylor series can be used to work with positive integers in a set but it can't be used for negative members of the set.

The last concept of data analysis you'll want to keep in mind is the singularity. A singularity is a point at which an object no longer behaves in a well behaved manner. Singularities are used all the time in science for things like black holes and they're used all the time for data analysis as well. You can use them to understand when humans are no longer suited for certain tasks in decision making, you can use them to in graphics to determine draw distance, and there are many more examples. This is a great example of what the singularity for a saddle point might look like



It just so happens that analytic combinatorics, the use of combinatorics to solve issues in data analysis, is a great way to improve your coding skill and a great inspiration for creative work. In analytic combinatorics you learn how to find ways to count objects and how to find their asymptotes with complex analysis. Those asymptotes can help you in so many ways. Saddle points are great for generating landscapes in games, meromorphic functions are a wonderful way to understand data. It really is timeless. We'll use this WebGL example to better understand how to use meromorphic functions to our advantage.

Our first example will be this monte carlo simulation.

```

/* This code is released under the MIT License, see
./LICENSE.
*
* Copyright 2013 Monetate, Inc.
*
* Links against Double precision SIMD-oriented Fast
Mersenne Twister (dSFMT) library.
*   http://www.math.sci.hiroshima-u.ac.jp/~m-
mat/MT/SFMT/index.html
*
* build
*   gcc -O3 -std=gnu99 -msse2 -DHAVE_SSE2
-DDSFMT_MEXP=19937 -o simulate dSFMT.c
simulate.c
*
* run
*   simulate iterations group_weight_0
group_weight_1 ...
*/

```

```

#include <malloc.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

```

```

#define DSFMT_DO_NOT_USE_OLD_NAMES
#include "dSFMT/dSFMT.h"

```

```

typedef struct Summary {

```

```
double y0;
double y1;
double y2;
} Summary;
```

```
static double
parse_double(char const* s)
{
    char* end;
    double r = strtod(s, &end);
    if (*end) {
        fprintf(stderr, "Non floating point characters:
%s\n", s);
        exit(2);
    }
    return r;
}
```

```
/*
 * Parses and splits csv line and into Summary struct
 *
 * Arguments:
 * summary -- Summary struct to use
 * line -- Comma separated line of form
"sample_id,1,0.0,0.0"
 */
static void
parse_sample_summary_line(Summary* summary,
char* line)
{
```



```

    char* token_state;
    strtok_r(line, ",", &token_state); // sample_id, which
we don't care about
    summary->y0 = parse_double(strtok_r(NULL, ",",
&token_state));
    summary->y1 = parse_double(strtok_r(NULL, ",",
&token_state));
    summary->y2 = parse_double(strtok_r(NULL, "\n",
&token_state));
}

```

```

/*
 * Pull in csv summaries, run random simulations and
write simulation results back out.
 *
 * Arguments:
 * in -- CSV file of sample summaries
 * out -- CSV file of simulation results
 * group_weights -- Array of weights as passed in from
command line. Ex: [25.0, 25.0, 50.0]
 * groups -- Number of groups
 * simulations -- Number of simulation iterations to run
 */

```

```

static void
simulate(FILE* in, FILE* out, double* group_weights,
int groups, int simulations)
{
    /*
     * Compute group cumulative distribution.
     *
     * Given group_weights of [33, 33, 33], cdf will be

```

```
[0.3333, 0.6667, 1.0000]
```

```
    */  
    double cdf[groups];  
    double total_weight = 0;  
    for (int g = 0; g < groups; ++g) {  
        total_weight += group_weights[g];  
    }  
    double cumulative_weight = 0;  
    for (int g = 0; g < groups; ++g) {  
        cumulative_weight += group_weights[g];  
        cdf[g] = (double) cumulative_weight /  
total_weight;  
    }  
  
    /* Initialize group_summaries to zero. */  
    Summary *group_summaries = calloc(simulations *  
groups, sizeof(Summary));  
  
    /* Initialize random number generator. */  
    dsfmt_t dsfmt;  
    dsfmt_init_gen_rand(&dsfmt, 1234);  
  
    /* 128 bit alignment for sse2 ops. */  
    double *randoms = (double *) memalign(16,  
sizeof(double) * simulations);  
  
    /* For each input line:  
    *   Parse summary line.  
    *   Generate random doubles for each simulation.  
    *   Accumulate into random summary group for  
each simulation.  
    */
```

```

char line[128];
while (fgets(line, sizeof(line), in)) {
    Summary sample_summary;
    parse_sample_summary_line(&sample_summary,
line);

    /* Generate random doubles in the interval [0, 1).
*/
    dsfmt_fill_array_close_open(&dsfmt, randoms,
simulations);

    Summary *row = group_summaries;
    for (int i = 0; i < simulations; ++i) {
        /* Select group for this sample and this iteration
*/
        double rnd = randoms[i];
        int g = 0;
        while (g < groups - 1 && cdf[g] < rnd) { ++g; }

        Summary *group_summary = row + g;
        group_summary->y0 += sample_summary.y0;
        group_summary->y1 += sample_summary.y1;
        group_summary->y2 += sample_summary.y2;

        row += groups;
    }
}

/* Output group_summaries. */
Summary *group_summary = group_summaries;
for (int i = 0; i < simulations; ++i) {
    for (int g = 0; g < groups; ++g) {

```

```

        fprintf(out, "%d,%d,%lf,%lf,%lf\n", i, g,
                group_summary->y0, group_summary->y1,
group_summary->y2);
        ++group_summary;
    }
}

```

```

    free(randoms);
    free(group_summaries);
}

```

```

int
main(int argc, char** argv)
{
    if (argc < 3) {
        fprintf(stderr, "Usage: simulate iterations weight0
weight1 ...\n");
        exit(1);
    }

```

```

    int simulations = atoi(argv[1]);
    int groups = argc - 2;

```

```

    double group_weights[groups];
    for (int i = 0; i < groups; ++i) {
        group_weights[i] = parse_double(argv[i + 2]);
    }

```

```

    /* TODO: Assert simulations even number >= 382
for sse2 implementation. */
    simulate(stdin, stdout, group_weights, groups,
simulations);

```

```
    return 0;  
}
```

## **Computer Vision**

One of the most exciting areas of programming for anyone wanting to see some of the more exciting

aspects is computer vision. Many elements of combinatorics are applicable here. Obviously, graph theory can reveal a variety of improvements especially when you aim to work with photogrammetry. Feature detection and image segmentation are also areas where you will find your ability to work with combinatorics is of vital importance. We will use some of the things we learned with data analysis in this section so keep them in mind.

One particularly important area of computer vision where combinatorics is useful can be found in image pyramids. Think of the image pyramid as a set and think of the subsections of the image as elements. Basically, you can continue to divide until you reach the point of a single pixel.

You'll find two of the most common types of image pyramids are the Gaussian and Laplacian. Let's take an example from this JSFeat a library designed around computer vision for javascript,

```
<var levels = 3, start_width = 640, start_height = 480,
```

```
    data_type = jsfeat.U8_t | jsfeat.C1_t;
```

```
var my_pyramid = new jsfeat.pyramid_t(levels);
```

```
// this will populate data property with matrix_t
instances

my_pyramid.allocate(start_width, start_height,
data_type);

var level_0 = my_pyramid.data[0]; // cols = 640, rows =
480

var level_1 = my_pyramid.data[1]; // cols = 320, rows =
240

var level_2 = my_pyramid.data[2]; // cols = 160, rows =
120

// with build method you can draw input source to
levels

// skip_first_level is true by default

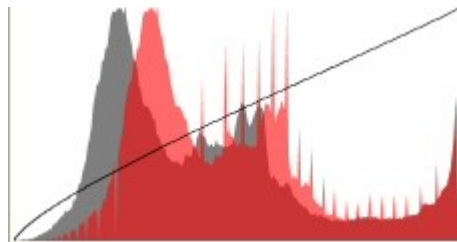
my_pyramid.build(source_matrix_t, skip_first_level =
true);>
```

The best way to understand this is as a series of graphs. As you slice the image into smaller sections with lower

resolutions, you will produce undirected graphs. This undirected graphs produces what is known as a geometric histogram.



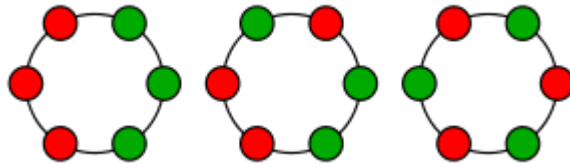
A sunflower image



The histogram for the sunflower

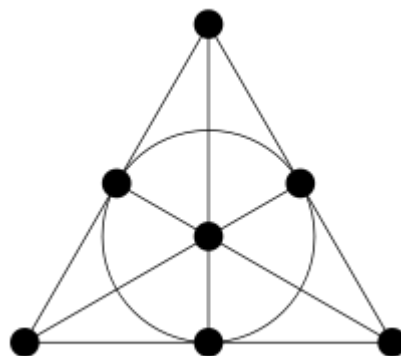
A **geometric histogram** gives you a way to represent the color data of the picture in a way that considers the configuration of that color data. Each time that you manage to drop down a rank on the image pyramid you are producing subsets that are then open to various combinatorial manipulations. You may treat them as **bracelets**, a chiral form of necklaces, and rotate them or reflect them about an axis. The middle necklace is an example of a bracelet.





Another area where you'll find interesting uses of combinatorics is in edge detection. Generally, you can simply use the brightness of an image to accurately define the outline of objects and from that outline you derive what you need to understand. You group pixels, or in some cases you group subpixels, together and you find the differences between these groups. Once you have the differences between these groups you can then determine where the edge should go.

Here we can find excellent use of matroid theory. Essentially a matroid gives us a way of understanding linear independence in vector spaces. Linear independence occurs when not even one vector in the set can be defined as a combination of two other vectors. For edge detection, this helps us avoid the mistake of placing edges where they do not belong.



Further use of matroids allows us to make a so called “greedy” algorithm. The greedy algorithm always finds the optimum choice to make in any situation. We can use it in this case to find the maximum number of edges we need for the object. We can do this with the Sobel derivative. Essentially the Sobel derivative gives us a simple way of creating an output image with a focus on edges.

// result is written as [gx0,gy0,gx1,gy1, ...]

sequence in 2 channel matrix\_t

```
jsfeat.imgproc.sobel_derivatives(source:matrix_t,  
dest:matrix_t);
```

Let's say you wanted to create a facial recognition app. One great way to do that is to use something in matroid theory known as hyperplanes. To understand exactly what a hyperplane is you're going to need a decent understanding of how it's parts.

First, every matroid has a finite number of elements in its set. Let's call our finite set A. The subsets are known as independent sets and we'll call the family of them B. A subset is independent if when compared to a larger set there exist an element that equals that larger set divided by the smaller set. If X is larger than Y, then there exists  $x=X/Y$  and it is within set A. The maximal independent set you can have before you have a dependent set is known as the base. Matroid theory

necessitates that every base of subset  $X$  have the same number of elements. This number is the rank of subset  $X$ .

With the rank of subset  $X$  we can get the closure of set  $A$ . The closure is an element of set  $A$  that equals both the rank of  $X$  and the rank of the union of  $X$  and that element. If the closure of the subset equals the subset itself that means it is a flat or subspace of the matroid. Finally, in a matroid of rank  $r$  the flat that equals  $r-1$  is the hyperplane.

Now we can focus on how to use those hyperplanes to for facial recognition. We can use the matroid to help us identify and isolate specific features and we can place each feature into a specific hyperplane. Each hyperplane will then give us a way of spotting a nose, a chin, eyes, and other important features. Here is the example code using Javascript.

```
<style>
    /* First we decide the image size */
    .thumb {
        height: 75px;
        border: 1px solid #000;
        margin: 10px 5px 0 0;
    }
</style>

<script type="text/javascript">
    /* Here we upload the picture */
    $(function () {
```

```

        $(":file").change(function () {
            if (this.files &&
this.files[0]) {
                var reader = new
FileReader();

                reader.onload =
imageIsLoaded;

reader.readAsDataURL(this.files[0]);
            }
        });
    });

    function imageIsLoaded(e) {
        $('#myImg').attr('src',
e.target.result);
        $('#yourImage').attr('src',
e.target.result);
    };
</script>

```

```

<input type='file' />
</br>
<script>
/* Now we produce the matroid */
var vert1 (x,y);
var vert2 (a,b);
var vert3 (c,d);

```

```
var vert4 (e,f);  
  
Set A(vert1, vert2, vert3, vert4);  
  x or y = vert1/vert2 or vert2/vert1  
</script>
```

Another area where combinatorics is useful is feature detection. Edge detection is great for helping us understand how to work with the shape of an object, but it does not allow use to understand the color of the object, depth, or other important features. With some good combinatorics we can easily create or modify algorithms in a way that allows us to make something truly useful.

One of the first feature tools we should look at is the ridge.

Another area where our understanding of graph theory and matroid theory can prove useful is in hand writing recognition. Everyone's hand writing is different from somebody else's, but with this we can understand exactly what we need to understand in to recognize the idiosyncrasies innate in each individual's writing. First let's start with this example in python.

First you're going to need to download the data sets we will use. You can find them here.

**Training set images:** train-images-idx3-ubyte.gz (9.9 MB, 47 MB unzipped, and 60,000 samples)

**Training set labels:** train-labels-idx1-ubyte.gz (29 KB, 60 KB unzipped, and 60,000 labels)

**Test set images:** t10k-images-idx3-ubyte.gz (1.6 MB, 7.8 MB, unzipped and 10,000 samples)

**Test set labels:** t10k-

Once you have that data on hand you will need to actually place that data into arrays the machine can understand.

```
import os
import struct
import numpy as np
def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
        '%s-labels-idx1-ubyte'
        % kind)
    images_path = os.path.join(path,
        '%s-images-idx3-ubyte'
        % kind)
    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
            lbpath.read(8))
        labels = np.fromfile(lbpath,
            dtype=np.uint8)
    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
```

```

imgpath.read(16))
images = np.fromfile(imgpath,
dtype=np.uint8).reshape(len(labels), 784)
return images, labels

```

Now that you've done that you'll want to actually load them. First you need to read the magic number of the data because it tells you about the file protocol and the number of items in the file. After that the file reads them

```

magic, n = struct.unpack('>II', lopath.read(8))
labels = np.fromfile(lopath, dtype=np.int8)

>>> X_train, y_train = load_mnist('mnist', kind='train')
>>> print('Rows: %d, columns: %d'
... % (X_train.shape[0], X_train.shape[1]))
Rows: 60000, columns: 784
>>> X_test, y_test =
load_mnist('mnist', kind='t10k')
>>> print('Rows: %d, columns: %d'
... % (X_test.shape[0],
X_test.shape[1]))
Rows: 10000, columns: 784

}

```

To understand how different people often write in different manners let's introduce an example that reveals the variety of ways people can write the number 7.

```
>>> fig, ax = plt.subplots(nrows=5,  
... ncols=5,  
... sharex=True,  
... sharey=True,)  
>>> ax = ax.flatten()  
>>> for i in range(25):  
... img = X_train[y_train == 7][i].reshape(28, 28)  
... ax[i].imshow(img, cmap='Greys',  
interpolation='nearest')  
>>> ax[0].set_xticks([])  
>>> ax[0].set_yticks([])  
>>> plt.tight_layout()  
>>> plt.show()
```



## **Machine Learning**

There probably isn't anything in programming more important today than that of machine learning. We are currently entering a second industrial age with artificial intelligence, as machine learning is often known to many, playing the central position. Most of the examples in this chapter will use Python but it shouldn't be too difficult to take the general gist of the ideas and place them into another language.

Before we get started we need to go over the basics of machine learning. In machine learning the data sets you will be presenting to the learning machine are known as epochs. Depending on the complexity of the learning machine you may find yourself in need of different amounts of epochs.

Several concepts from combinatorics work well in this area. Epochs can be understood through the lens of necklaces. We can then create algorithms designed to help the learning machine in question understand how to recognize patterns even when they are subjected to rotation, reflection, and other means of distortion.

Take this example of a perceptron.

```
import numpy as np
class Perceptron(object):
    """Perceptron classifier.
    Parameters
    -----
    eta : float
    Learning rate (between 0.0 and 1.0)
    n_iter : int
    Passes over the training dataset.
    Attributes
    -----
    w_ : 1d-array
    Weights after fitting.
    errors_ : list
    Number of misclassifications in every epoch.
    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter
    def fit(self, X, y):
        """Fit training data.
        Parameters
        -----
        X : {array-like}, shape = [n_samples, n_features]
        Training vectors, where n_samples
        is the number of samples and
        n_features is the number of features.
        y : array-like, shape = [n_samples]
        Target values.
```

## Returns

-----

```
self : object
"""
self.w_ = np.zeros(1 + X.shape[1])
self.errors_ = []
for _ in range(self.n_iter):
    errors = 0
    for xi, target in zip(X, y):
        update = self.eta * (target - self.predict(xi))
        self.w_[1:] += update * xi
        self.w_[0] += update
        errors += int(update != 0.0)
    self.errors_.append(errors)
    return self
def net_input(self, X):
    """Calculate net input"""
    return np.dot(X, self.w_[1:]) + self.w_[0]
def predict(self, X):
    """Return class label after unit step"""
    return np.where(self.net_input(X) >= 0.0, 1, -1)
```

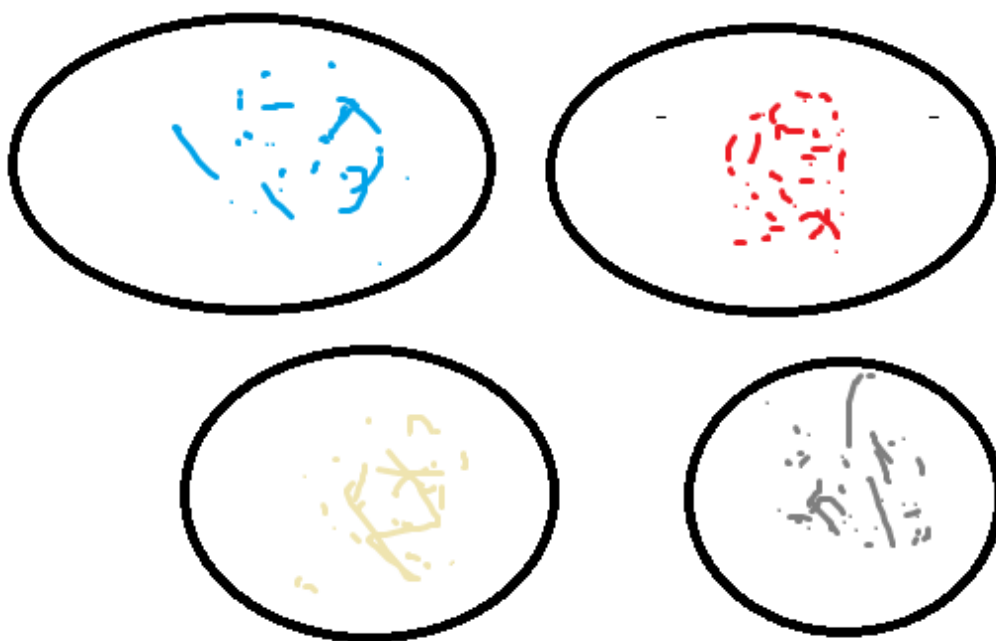
In this example we clearly see an algorithm that takes data from the epoch and put it in either 1 of 2 categories. We can look at this and see areas where we can put combinatorics to work. Specifically, we can look at the error rate for enhancements.

Let's start by taking a look at a classifier algorithm that uses color to help distinguish various flowers. After we are done

```

from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
def plot_decision_regions(X, y, classifier,
test_idx=None, resolution=0.02):
# setup marker generator and color map
markers = ('s', 'x', 'o', '^', 'v')
colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
cmap = ListedColormap(colors[:len(np.unique(y))])
# plot the decision surface
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() +
1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() +
1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max,
resolution),
np.arange(x2_min, x2_max, resolution))
Z = classifier.predict(np.array([xx1.ravel(),
xx2.ravel()]).T)
Z = Z.reshape(xx1.shape)
plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())
# plot all samples
X_test, y_test = X[test_idx, :], y[test_idx]
for idx, cl in enumerate(np.unique(y)):
plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
alpha=0.8, c=cmap(idx),
marker=markers[idx], label=cl)

```



In this representation of the machine's learning process we essentially see the machine splitting the data into sets based on color in a way very similar to some of the earlier experiments we conducted using necklaces. We can see that combinatorics can be used to greatly enhance any algorithm.

Now that we've seen a few examples of machine learning let's use some of the data analysis techniques we learned earlier on to enhance our machine learning.

First we'll use a Poisson generating function to create random sets of data that we can have our machine learn from. Then we will use complex analysis techniques to help us categorize that data.

First we're going to use movie reviews from IMDB to create an epoch of us to work with. Here we start with 50,000 reviews

```
>>> import pyprind
>>> import pandas as pd
>>> import os
>>> pbar = pyprind.ProgBar(50000)
>>> labels = {'pos':1, 'neg':0}
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
... for l in ('pos', 'neg'):
... path = './aclImdb/%s/%s' % (s, l)
... for file in os.listdir(path):
... with open(os.path.join(path, file), 'r') as infile:
... txt = infile.read()
... df = df.append([[txt, labels[l]]],
ignore_index=True)
... pbar.update()
>>> df.columns = ['review', 'sentiment']
0% 100%
[#####] | ETA[sec]:
0.000
Total time elapsed: 725.001 sec
```

In order to really improve the quality of our data we'll also use permutations to rearrange the data in a random manner.

```
>>> import numpy as np
```

```
>>> np.random.seed(0)
>>> df =
df.reindex(np.random.permutation(df.index))
>>> df.to_csv('./movie_data.csv', index=False)
```

Teaching a machine how to understand the opinions in a movie review is very similar algorithmically to the process behind a search engine. In order for the machine to understand the data you have to convert the words of the natural language into numbers it can understand by tokenizing them.

This is where the bag of words model comes in. First you create a vocabulary of certain words such as “funny” or “sad”. You then use an algorithm to help the machine understand how many times each word appears in the document. The words to be examined are converted into arrays we'll call feature vectors. “The clouds are blocking the sun” and “That was a real disaster” are examples of feature vectors.

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import
CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array([
... 'What a beautiful morning',
... 'I hope this isn't too bad!',
... 'I hated every minute of that movie didn't you?'])
>>> bag = count.fit_transform(docs)
```

Alright now we have a feature vector and we can use it to help us understand exactly what we're about to do.

Now we can partition the words into partitions. For example “What is this here?” and “what is that there” can be turned into “What is”, “this here”, and “that there”. This becomes pretty useful because it not only helps us understand which words are popping up but their context as well. If “this is” and “so sad” are found more often in feature features than “very happy” we can understand that the movie reviews suggest the films are melancholy. Yet another example of combinatorics improving our code.

Another importance aspect of this experiment will be term frequency. When you're trying to parse through documents you don't want relatively meaningless words carrying too much weight. A great way to stop this from happening is to give certain tokens less weight.

$$\text{tf-idf}(t,d)=\text{tf}(t,d) \times \text{idf}(t,d)$$

Here idf stands for inverse document frequency while tf represents term frequency with t represents t and d representing documents. The inverse document frequency formula is given by

$$\text{idf} = \log(Nd/1+df(d,t))$$



```
>>> from sklearn.feature_extraction.text import  
TfidfTransformer  
  
>>> tfidf = TfidfTransformer()  
>>> np.set_printoptions(precision=2)
```

After we do this we follow up by adding the tokenizer here.

```
>>> def tokenizer(text):  
... return text.split()  
>>> tokenizer('runners like running and thus they  
run')  
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

An effective way to help us improve our machine learning is to use something known as stemming. Essentially stemming is the process of turning each word into it's root form. Combinatorics provides an easy way to do this with a feature vector algorithm including unavoidable pattern recognition.

After you convert the word into a feature vector you specifically look to edit out every suffix and prefix you can. A typical stemmer would start with something like this java code example

```
var stemmer = (function(){  
    var step2list = {
```

"ational" : "ate",  
"tional" : "tion",  
"enci" : "ence",  
"anci" : "ance",  
"izer" : "ize",  
"bli" : "ble",  
"alli" : "al",  
"entli" : "ent",  
"eli" : "e",  
"ousli" : "ous",  
"ization" : "ize",  
"ation" : "ate",  
"ator" : "ate",  
"alism" : "al",  
"iveness" : "ive",  
"fulness" : "ful",  
"ousness" : "ous",  
"aliti" : "al",  
"iviti" : "ive",  
"biliti" : "ble",  
"logi" : "log"  
},

then you might have an algorithm to match the suffix of those word's to substrings including these endings.

We can enhance this by using an avoidability index to help us better understand the sentiment behind the words. Although none of the words will match the patterns “ing”, “er”, or “est”, we can learn much from seeing them in words. For example, If “eating” implies a current activity but “eaten” implies a past active. This sort of difference can make a huge difference when assessing the sentiment behind specific words. Take this javascript example for instance. First we begin by cutting the suffix of the word.

```
<script type="text/javascript">
var str = stemmer list;
var splitted = str.split(" ", -2);
return splitted;
</script>
```

The suffix we derive will be given an avoidability index and we'll be able to use that to better understand the sentiment. For example, the root word “sad” and the suffix “er” will give the impression that the movie review is comparing the movie to others and finding it to be more poignant. Let's match sliced to the list of suffixes.

```
<script type="text/javascript">
var str ;
var re = /( )/i;
var found = str.match( re );
</script>
```

Now we use the suffixes we grabbed we list them

according to their avoidability index here. A suffix like “al” or “er” would go in the 2 avoidability index while “est” and “ine” would be in the 3 avoidability index.

```
    if (!Array.reduce)
    {
    Array.reduce = function(fun /*, */)
    {
    var len = 2.length;
    if ( fun != "function")
    throw new wrong();
    // no value to return if no initial
    value and an empty array
    if (len == 0 && arguments.length == 1)
    throw new deadwrong();
    var i = 0;
    if (arguments.length >= 2)
    {
    var rv = arguments[1];
    }
    else
    {
    do
    {
    if (i in this)
    {
    rv = this[i++];
    break;
    }
```

```

}
// if array contains no values, no
initial value to return
if (++i >= len)
throw new wrong();
}
while (true);
}
for (; i < len; i++)
{
if (i in this)
rv = fun.call(null, rv, this[i], i,
this);
}
return rv;
};
}

```

Once we have placed them into the avoidability indexes the next thing to do is use them to assess the intention behind the reviews. For instance, ration plus “ale” would suggest persuasion is intended while ration plus “al” would suggest the review is making statements about logic rather than rhetoric. The code example here gives just that.

Now that we understand exactly what it takes to create good machine learning algorithms we can now focus on how to put those machine learning

algorithms into a web application. Machine learning is found in everything from search engines like Google to video streaming sites like Youtube to virtual assistance programs used by many businesses.

A great place to start is with learning estimators. First let's load this movie classifier

```
>>> import pickle

>>> import os

>>> dest = os.path.join('movieclassifier',
'pkl_objects')

>>> if not os.path.exists(dest):
... os.makedirs(dest)

>>> pickle.dump(stop,
... open(os.path.join(dest, 'stopwords.pkl'), 'wb'),
... protocol=4)

>>> pickle.dump(clf,
... open(os.path.join(dest, 'classifier.pkl'), 'wb'),
... protocol=4)
```

Now we create our tokenizer and save it as vectorizer.py

```
from sklearn.feature_extraction.text import
HashingVectorizer
import re
import os
import pickle
cur_dir = os.path.dirname(__file__)
stop = pickle.load(open(
os.path.join(cur_dir,
```

```

'pkl_objects',
'stopwords.pkl'), 'rb'))
def tokenizer(text):
text = re.sub('<[^\>]*>', '', text)
emoticons = re.findall('(?:::|;|=)(?:-)?(?:\)|\(|D|P)',
text.lower())
text = re.sub('[\W]+', ' ', text.lower()) \
+ ' '.join(emoticons).replace('-', '')
tokenized = [w for w in text.split() if w not in stop]
return tokenized
vect = HashingVectorizer(decode_error='ignore',
n_features=2**21,
preprocessor=None,
tokenizer=tokenizer)

```

Now we will import from vectorizer.

```

>>> import pickle
>>> import re
>>> import os
>>> from vectorizer import vect
>>> clf = pickle.load(open(
... os.path.join('pkl_objects',
... 'classifier.pkl'), 'rb'))

```

Now that we have both we can use them to review select document samples.

```

>>> import numpy as np

```

```
>>> label = {0:'negative', 1:'positive'}
>>> example = ['I love this movie']
>>> X = vect.transform(example)
>>> print('Prediction: %s\nProbability: %.2f%%' %\
... (label[clf.predict(X)[0]],
... np.max(clf.predict_proba(X))*100))
Prediction: positive
Probability: 91.56%
```

The movie classifier and the object classifier are great by themselves, but we can improve on this model. The pickle module provides a great way create object hierarchies but using partitioning we can actually make the process more efficient. We'll add our own partitioning code here.