



CONTENTS

1	Tackling Difficult Problems: Positive Semidefinite Matrices	3
1.1	NP Problems	3
1.2	A Past Approach: Minimizing Errors in Neural Networks	4
1.3	Positive Definite and Semidefinite Matrices	5
1.4	Identifying and Constructing a Positive Semidefinite Matrix	6
1.5	The Max Cut Problem	7
1.6	The Max Cut Problem Solved in Python	9
2	Data Tables and pandas	11
2.1	Data types	11
2.2	pandas	12
2.3	Reading a CSV with pandas	12
2.4	Looking at a Series	13
2.5	Rows and the index	14
2.6	Changing data	15
2.7	Derived columns	17
3	Data tables in SQL	19
3.1	Using SQL from Python	22
4	Representing Natural Numbers	25
A	Answers to Exercises	27
	Index	29

Tackling Difficult Problems: Positive Semidefinite Matrices

With all the computing power available today, you would think no problem would be too difficult to tackle. However, this is not the case. There is a category of problems called NP (nondeterministic polynomial time) whose solution is easy to verify, but whose computation is difficult to perform. This is because there is no straightforward algorithm and any brute-force method would take too much time. For these problems, all we can do is to develop an efficient algorithm that can find a solution in a reasonable amount of time. While the solution might not be the most optimal, the goal is to find the best solution possible in a short time.

As you learn more about optimization techniques, you will come across many efficiency algorithms that have been used throughout the years. In the 1990s, the field of optimization changed with the discovery that algorithms based on semidefinite positive matrices can achieve a higher efficiency than seen in the past. Today, there is an entire field of programming called Semidefinite Programming that is based on the use of semidefinite positive matrices.

First we will take a look at what some of the NP problems are. Next, we will describe the intuition behind semidefinite positive matrices. We will take a look at one NP problem, then introduce the Python module to use for solving these problems.

1.1 NP Problems

In the world of mathematics, easy problems are referred to as P, or polynomial time, problems. In simple terms, this means the problem can be solved quickly, and it is easy to verify that the solution is correct. Addition, subtraction, division, multiplication, square roots, and matrix-vector multiplication are just a few examples. NP problems, as stated in the introduction, can't be computed in a reasonable amount of time, but solutions are usually easy to verify. The game of Sudoku is one such example. It is easy to verify a correct solution, but writing a generalized algorithm to solve any Sudoku game is an NP problem.

NP problems show up in many other situations, such as:

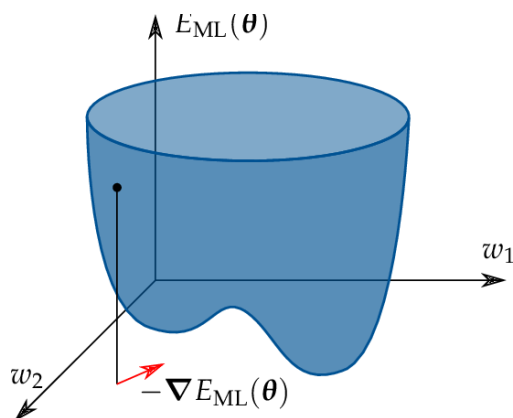
- Designing robust communication networks

- Scheduling tasks without conflicts
- Managing supply chains
- Detecting patterns in biological networks
- Figuring out subgroups in social networks
- Predicting the structure of proteins

For each of these, think of large scale problems for which there are many variables. A cloud computing company that provides AI services to thousands of clients must be able to schedule tasks efficiently and in a timely manner, as well as manage the power needed for the computers and cooling the data center. Supply chain management is crucial to figuring out how to pick up, transport, and distribute goods to help provide disaster relief. Understanding protein structure is important for designing drugs that can tackle specific conditions. All we can do for each of these situations is to find an optimal solution, but we cannot find the definitive solution.

1.2 A Past Approach: Minimizing Errors in Neural Networks

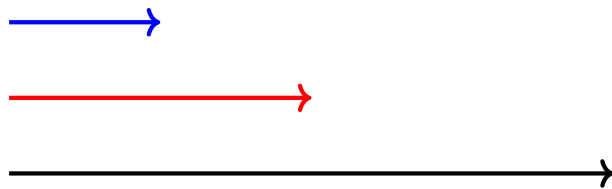
When neural networks were first being developed to recognize things (faces, letters, music, and so on), the goal was to calculate weights between network nodes that would minimize the recognition error. The error space could be visualized as a surface of valleys and hills. The lowest point would have the least error. The idea behind the iterative weight calculations for training the network was to descend down the gradient until reaching the low point. Without getting into the mathematics, you can see by looking at this figure that there are two valleys. Some neural network training resulted in ending at a low point, but not the lowest point. Wouldn't it be great if you could formulate an optimization problem so that you would be guaranteed to land at the lowest point? That is where semidefinite programming can help.



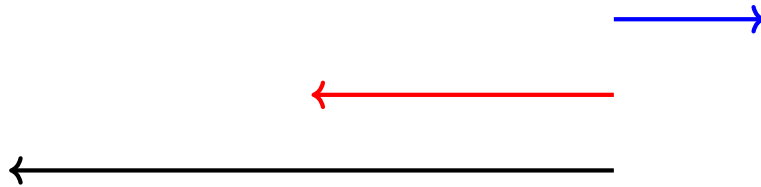
1.3 Positive Definite and Semidefinite Matrices

Unlike matrices that are defined by their content (such as identity matrix, zero matrix, and diagonal matrix), positive definite and semidefinite matrices are characterized by the result they produce. They have an analog in the scalar world, so let's first look at that. Let's take the scalars a and b and treat them as vectors. ab is then the dot product. If $a > 0$, then ab will take on the sign of b . If $a < 0$, then ab will have the opposite sign of b . If you look at this in a graph, you can see that in the first case, ab stays on the same side of the origin, but in the second case, ab flips

For $a = [2]$, $b = [4]$



For $a = [3]$, $b = [-4]$. the result of multiplication flips a to the other side of the graph.



The notion of “staying on the same side” is positive definite. The notion of flipping is negative definite. Positive definite means that $x > 0$, so the result is a positive number. Positive semidefinite means that $x \geq 0$, so the result is a non-negative number.

If you can formulate a problem as a positive semidefinite matrix, then you automatically constrain the result to the “same side.” This constraint results in higher algorithmic efficiency.

Let's look at the formal definition:

A matrix is positive definite if, and only if:

$$x^T A x > 0, x, x \neq 0$$

A matrix is positive semidefinite, if, and only if:

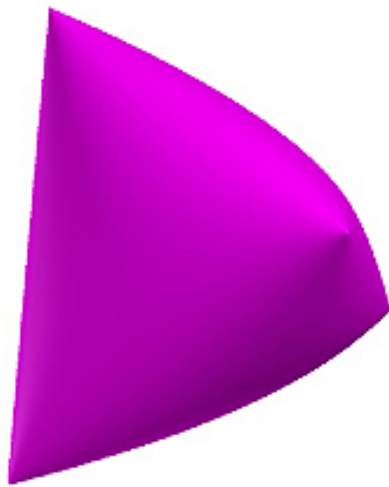
$$x^T A x \geq 0$$

Further, a positive semidefinite matrix had an important property. Its eigenvalues are ≥ 0 . The eigenvalues of a positive definite matrix are > 0 .

Take the triplet (a, b, c) and the symmetric matrix:

$$\begin{bmatrix} 1 & a & b \\ a & 1 & c \\ b & c & 1 \end{bmatrix} \geq 0$$

For what values of a, b, c is this matrix positive semidefinite? If you iterate through all possible combinations of a, b, c , then compute the eigenvalues for each matrix, you will find that some (like $0, 0, 0$) result in a positive semidefinite matrix and some (like $2, 2, 2$) are not positive semidefinite. If you plot the set of triplets that result in a semidefinite matrix, you will see an ellipsope. This shape guarantees an optimal solution.



1.4 Identifying and Constructing a Positive Semidefinite Matrix

In the last section, you saw that being symmetric does not guarantee a positive semidefinite matrix. You also saw that a matrix of positive values does not guarantee a positive semidefinite matrix. The only way to check for positive semidefinite is to calculate the eigenvalues, which you learned in a previous workbook.

A surefire way to construct a positive semidefinite matrix is:

$$AA^T$$

Exercise 1 Figuring out if a matrix is positive semidefinite

Is this matrix positive definite? Show your work.

$$\begin{bmatrix} 2 & 2 \\ 2 & 0 \end{bmatrix}$$

*Working Space**Answer on Page 27***Exercise 2 Creating a positive semidefinite matrix**

Using any 3 by 3 matrix, create a positive semidefinite matrix. Then show it is positive semidefinite by calculating its eigenvalues. You can either compute this by hand or using Python. In either case, show your work.

*Working Space**Answer on Page 27***1.5 The Max Cut Problem**

A famous NP problem is Max Cut. Given a graph of interconnected nodes, cut the graph to create two sets of nodes, such that the cut goes through as many edges as possible. (You can't cut an edge more than once.) Max Cut is important for binary classification in machine learning, circuit design, statistical physics, and more. There is no algorithm that will provide an exact solution. (If you could find one, you would be eligible to win a huge prize from the Clay Mathematics Institute!) Instead, you will see how to approximate a solution to this problem using a positive semidefinite matrix and a technique developed by mathematicians Michel Goemans and David Williamson.

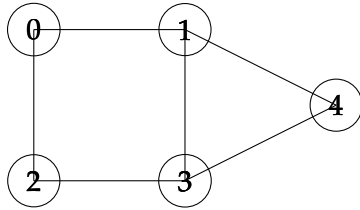
You won't see all the complete details here, as this section is meant to be a quick introduction to how you can apply positive semidefinite matrices.

Take this simple graph of five nodes and six edges. Each node in the graph will take on

one of two values (1 or -1) to show which set they fall into after a cut is made. For any two connected nodes, $x_i * x_j = 1$ if $x_i = x_j$ and -1 otherwise.

If you randomly assign 1 and -1 to the nodes, the chance of making the max cut is 0.5. By using semidefinite programming, you can achieve an algorithmic efficiency of 0.87.

The Goemans-Williamson technique can be used for any optimization problem where the variables take on the values of 1 and -1 .



As a list of edges the graph is:

$$\text{edges} = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 3), (3, 4)]$$

The optimization problem can be formulated as:

$$\text{Max} \sum_{\text{edges}(i,j)} \frac{1 - x_i x_j}{2}$$

for

$$x_i \in \{-1, 1\}$$

However, instead of allowing x_i to be scalar, Goemans-Williamson defines x_i as unit vectors.

$$x_i \in \mathbb{R}^n, \|x_i\| = 1$$

and that makes the optimization equation:

$$\text{Max} \sum_{\text{edges}(i,j)} \frac{1 - x_i^T x_j}{2}$$

It is this "relaxation" that gets us to a semidefinite matrix, because we can now rewrite the problem as a positive semidefinite matrix:

$$X = [x_i^T x_j]_{i,j}$$

Python has a module for solving optimization problems. Using this, you will get an

optimum matrix, but to get the unit vectors, you will need to take the square root of the matrix.

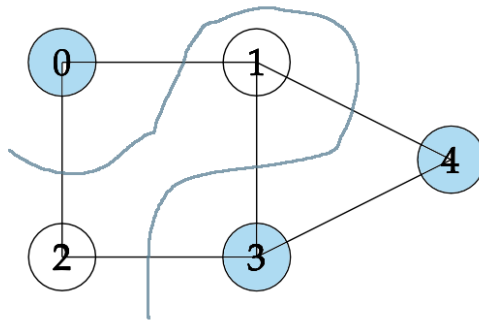
$$X = [x_1 \dots x_{1n}] [x_1 \dots x_{1n}]^T$$

Next we need to go from unit vectors to scalars, using a process called rounding.

$$x^i \in \mathbb{R}^n \rightarrow x^i \in \{-1, 1\}$$

Goemans-Williamson leveraged the fact that the end point of a unit vector is on a sphere. They generated a random plane to bisect the sphere. A vector on one side of the plane is assigned the value of 1 and a vector on the other side a value of -1 .

You can then assign the scalar values to the nodes and make the cut accordingly. This particular cut will be 5, as shown.



1.6 The Max Cut Problem Solved in Python

```
# MaxCut Problem
import numpy as np
import scipy
from scipy.linalg import sqrtm
# cvxpy is a python module for solving optimization problems
import cvxpy as cp

# define the edges of the graph
edges = [(0,1),
          (0,2),
          (1,3),
          (1,4),
          (2,3),
          (3,4)]

# Declare the matrix X to be positive semidefinite
X = cp.Variable((5,5),symmetric=True)
constraints = [X >> 0]
```

```
# Set diagonals to 1 to get unit vectors
constraints += [
    X[i,i] == 1 for i in range(5)
]

# Set the objective function
objective = sum(0.5*(1 - X[i,j]) for (i,j) in edges)

# Set up the problem to maximize using the objective function and
# keeping within the set constraints
prob = cp.Problem(cp.Maximize(objective), constraints)

# Returns a positive semidefinite matrix
print(prob.solve())

# To get the vectors, take square root of the matrix
x = sqrtm(X.value)

# Generate a random hyperplane
u = np.random.randn(5) # normal to random hyperplane

# Pick values according to which side of the hyperplane the vectors are on
x = np.sign(x @ u)
```

Data Tables and pandas

Much of the data that you will encounter in your career will come to you as a table. Some of these tables are spreadsheets, some are in relational databases, and some will come to you as CSV files.

Typically, each column will represent an attribute (like height or acreage) and each row will represent an entity (like a person or a farm). You might get a table like this:

property_id	bedrooms	square_meters	estimated_value
7927	3	921.4	\$ 294,393
9329	2	829.1	\$ 207,420

In most cases, one of the columns is guaranteed to be unique. We call this the *primary key*. In this table, `property_id` is the primary key; every property has one, and no two properties have the same `property_id`.

2.1 Data types

Each column in a table has a type, and these usually correspond pretty nicely with types in Python.

Here are some common datatypes:

Type	Python type	Example
Integer	<code>int</code>	910393
Float	<code>float</code>	-23.19
String	<code>string</code>	'Fred'
Boolean	<code>bool</code>	False
Date	<code>datetime.date</code>	2019-12-04
Timestamps	<code>datetime.datetime</code>	2022-06-10T14:05:22Z

Sometimes it is OK to have values missing. For example, if you had a table of data about employees, maybe one of the columns would be `retirement` — a date that tells you when the person retired. People who had not yet retired would have no value in this column. We would say that they have *null* for `retirement`.

Sometimes there are constraints on what values can appear in the column. For example, if the column were `height`, it would make no sense to have a negative value.

Sometimes a column can only be one of a few values. For example, if you ran a bike rental shop, each bicycle's status would be "available", "rented", or "broken". Any other values in that column would not be allowed. We often call these columns *categorical*.

2.2 pandas

The Python community works with tables of data *very* often, so it created the pandas library for reading, writing, and manipulating tables of data.

When working with tables, you sometimes need to go through them row-by-row. However, for large datasets, this is very slow. pandas makes it easy (and very fast) to say things like "Delete every row that doesn't have a value for height" instead of requiring you to step through the whole table.

In pandas, there are two datatypes that you use a lot:

- a `Series` is a single column of data.
- a `DataFrame` is a table of data: it has a `Series` for each column.

In the digital resources, you will find `bikes.csv`. If you look at it in a text editor, it will start like this:

```
bike_id,brand,size,purchase_price,purchase_date,status
5636248,GT,57,277.99,1986-09-07,available
4156134,Giant,56,201.52,2005-01-09,rented
7971254,Cannondale,54,292.25,1978-02-28,available
3600023,Canyon,57,197.62,2007-02-15,broken
```

The first line is a header; it tells you the name of each column. Next, the values are separated by commas. (Thus the name: CSV stands for "Comma-Separated Values".)

2.3 Reading a CSV with pandas

Let's make a program that reads `bikes.csv` into a pandas dataframe. Create a file called `report.py` in the same folder as `bikes.csv`.

First, we will read in the csv file. pandas has one series that acts as the primary key; it calls this one the index. When reading in the file, we will tell it to use the `bike_id` as the index series.

If you ask a dataframe for its shape, it returns a tuple containing the number of rows and

the number of columns. To confirm that we have actually read the data in, let's print those numbers. Add these lines to `report.py`:

```
import pandas as pd

# Read the CSV and create a dataframe
df = pd.read_csv('bikes.csv', index_col="bike_id")

# Show the shape of the dataframe
(row_count, col_count) = df.shape
print(f"*** Basics ***")
print(f"Bikes: {row_count:,}")
print(f"Columns: {col_count}")
```

Build it and run it. You should see something like this:

```
*** Basics ***
Bikes: 998
Columns: 5
```

Note that your table actually has six columns. The index series is not included in the shape.

2.4 Looking at a Series

Let's get the lowest, the highest, and the mean purchase price of the bikes. The purchase price is a series, and you can ask the dataframe for it. Add these lines to the end of your program:

```
# Purchase price stats
print("\n*** Purchase Price ***")
series = df["purchase_price"]
print(f"Lowest:{series.min()}")
print(f"Highest:{series.max()}")
print(f"Mean:{series.mean():.2f}")
```

Now when you run it, you will see a few additional lines:

```
*** Purchase Price ***
Lowest:107.37
Highest:377.7
Mean:249.01
```

What are all the brands of the bikes? Add a few more lines to your program that shows how many of each brand:

```
# Brand stats
print("\n*** Brands ***")
series = df["brand"]
series_counts = series.value_counts()
print(f"{series_counts}")
```

Now when you run it, your report will include the number of bikes for each brand from most to least common:

```
*** Brands ***
Canyon      192
BMC         173
Cannondale  170
Trek        166
GT          150
Giant       147
Name: brand, dtype: int64
```

`value_counts` returns a Series. To format this better, we need to learn about accessing individual rows in a series.

2.5 Rows and the index

In an array, you ask for data using an the location (as an int) of the item you want. You can do this in pandas using `iloc`. Add this to the end of your program:

```
# First bike
print("\n*** First Bike ***")
row = df.iloc[0]
print(f"{row}")
```

When you run it, you will see the attributes of the first row of data:

```
*** First Bike ***
brand      GT
size       57
purchase_price  277.99
purchase_date 1986-09-07
```

```
status          available
Name: 5636248, dtype: object
```

Notice that the data coming back is actually another series.

The last line says that the name (the value for the index column) for this row is 5636248. In pandas, we usually use this to locate particular rows. For example, there is a row with bike_id equal to 2969341. Let's ask for one entry from the

```
print("\n*** Some Bike ***")
brand = df.loc[2969341]['brand']
print(f"brand = {brand}")
```

Now, you will see the information about that bike:

```
*** Some Bike ***
brand = Cannondale
```

pandas has a few different ways of getting to that value. All of these get you the same thing:

```
brand = df.loc[2969341]['brand'] # Get row, then get value
brand = df['brand'][2969341]      # Get column, then get value
brand = df.loc[2969341, 'brand'] # One call with both row and value
```

2.6 Changing data

One of your attributes needs cleaning up. Every bike should have a status, and it should be one of the following strings: "available", "rented", or "broken". Get counts for each unique value in status:

```
print("\n*** Status ***")
series = df["status"]
missing = series.isnull()
print(f"{missing.sum()} bikes have no status.")
series_counts = series.value_counts()
for value in series_counts.index:
    print(f"{series_counts.loc[value]} bikes are \"{value}\"")
```

This will show you:

```
*** Status ***
7 bikes have no status.
389 bikes are "rented"
304 bikes are "broken"
296 bikes are "available"
1 bikes are "Flat tire"
1 bikes are "Available"
```

Right away, we can see two easily fixable problems: Someone typed “Available” instead of “available”. Right after you read the CSV in, fix this in the data frame:

```
mask = df['status'] == 'Available'
print(f"{mask}")
df.loc[mask, 'status'] = 'available'
```

When you run this, you will see that the mask is a series with bike_id as the index and False or True as the value, depending on whether the row’s status was equal to “Available”.

When you use loc with this sort of mask, you are saying “Give me all the rows for which the mask is True.” So, the assignment only happens in the one problematic row.

Let’s get rid of the mask variable and do the same for turning Flat tire into Broken:

```
df.loc[df['status'] == 'Available', 'status'] = 'available'
df.loc[df['status'] == 'Flat tire', 'status'] = 'broken'
```

Now those problems are gone:

```
7 bikes have no status.
389 bikes are "rented"
305 bikes are "broken"
297 bikes are "available"
```

What about the rows with no values for status? If we were pretty certain that the bikes were available, we could just set them to ‘available’:

```
missing_mask = df['status'].isnull()
df.loc[missing_mask, 'status'] = 'available'
```

Or maybe we would print out the IDs of the bikes so that we could go look for them:


```
missing_mask = df['status'].isnull()
missing_ids = list(df[missing_mask].index)
print(f"These bikes have no status:{missing_ids}")
```

However, let's just keep the rows where the status is not null:

```
missing_mask = df['status'].isnull()
df = df[~missing_mask]
```

At the end of your program, write out the improved CSV:

```
df.to_csv('bikes2.csv')
```

Run the program and open `bikes2.csv` in a text editor.

2.7 Derived columns

Let's say that you want to add a column with age of the bicycle in days:

```
bike_id,brand,size,purchase_price,purchase_date,status,age_in_days
5636248,GT,57,277.99,1986-09-07,available,13061
4156134,Giant,56,201.52,2005-01-09,rented,6362
7971254,Cannondale,54,292.25,1978-02-28,available,16174
```

Your first problem is that the `purchase_date` column looks like a date, but really it is a string. So, you need to convert it to a date. You can do this by applying a function to every item in the series:

```
df['purchase_date'] = df['purchase_date'].apply(lambda s: datetime.date.fromisoformat(s))
```

(With pandas, there is often more than one way to do things. pandas has a `to_datetime` function that converts every entry in a sequence to a datetime object. Here is another way to convert the string column in to a date column:

```
df['purchase_date'] = pd.to_datetime(df['purchase_date']).dt.date
```

You can look up `dt` and `date` if you are curious.)

Now, we can use the same trick to create a new column with the age in days:

```
today = datetime.date.today()
```

```
df['age_in_days'] = df['purchase_date'].apply(lambda d: (today - d).days)
```

When you run this, the new `bikes.csv` will have an `age_by_date` column.

Data tables in SQL

Most organizations keep their data as tables inside a relational database management system. Developers talk to those systems using a language called SQL (“Structured Query Language”).

Some relational database managers are pricey products you may have heard of before, such as Oracle or Microsoft SQL Server. Some are free, such as PostgreSQL or MySQL. These are server software that client programs talk to over the companies network.

There is a library, called `sqlite`, that lets us create files that hold tables. We can use SQL to create, edit, and browse those tables. `sqlite` is free, fast, and very easy to install. We will use `sqlite` instead of a networked database management system.

If you look in your digital resources, you will find a file called `bikes.db`. We created this file using `sqlite`, and now you will use `sqlite` to access it.

In the terminal, get to the directory where `bikes.db` lives. To open the `sqlite` tool on that file:

```
> textbfsqlite3 bikes.db
```

(If your system complains that there is no `sqlite3` tool, you need to install `sqlite`. See this website: <https://sqlite.org/>)

Please follow along: type each command shown here into the terminal and see what happens.

We mostly run SQL commands in this tool, but there are a few non-SQL commands that all start with a period. To see the tables and their columns, you can run `.schema`:

```
sqlite> .schema
CREATE TABLE bike (bike_id int PRIMARY KEY, brand text, size int,
                    purchase_price real, purchase_date date, status text);
```

That is the SQL command that we used to create the `bike` table. You can see all the columns and their types.

You want to see all the rows of data in that table?

```
sqlite> select * from bike;
4997391|GT|57|269.61|2009-05-03|rented
5429447|Cannondale|50|215.91|2002-02-17|broken
5019171|Trek|58|251.17|1985-07-11|rented
3000288|Cannondale|57|211.08|1993-01-05|broken
880965|GT|52|281.75|1995-08-02|available
...
```

You will see 1000 rows of data!

The SQL language is not case-sensitive, so you can also write it like this:

```
sqlite> SELECT * FROM BIKE;
```

Often, you will see SQL with just the SQL keywords in all caps:

```
sqlite> SELECT * FROM bike;
```

The semicolon is not part of SQL, but it tells sqlite that you are done writing a command and that it should be executed.

SQL lets you choose which columns you would like to see:

```
sqlite> SELECT bike_id, brand FROM bike;
4997391|GT
5429447|Cannondale
5019171|Trek
3000288|Cannondale
...
```

Using WHERE, SQL lets you choose which rows you would like to see:

```
sqlite> SELECT * FROM bike WHERE purchase_date > '2009-01-01' AND brand = 'GT';
4997391|GT|57|269.61|2009-05-03|rented
326774|GT|56|165.0|2009-06-27|available
264933|GT|52|302.43|2009-07-09|available
5931243|GT|55|173.56|2009-11-26|rented
4819848|GT|51|221.71|2009-12-11|rented
9347713|GT|52|232.32|2009-06-13|available
3019205|GT|58|262.94|2009-08-22|available
```

Using DISTINCT, SQL lets you get just one copy of each value:

```
sqlite> SELECT DISTINCT status FROM bike;
rented
broken
available

Busted
Flat tire
good
out
Rented
```

You can also edit these rows. For example, if you wanted every status that is Busted to be changed to broken, you can use an UPDATE statement:

```
sqlite> UPDATE bike SET status='broken' WHERE status='Busted';
sqlite> SELECT DISTINCT status FROM bike;
rented
broken
available

Flat tire
good
out
Rented
```

You can insert new rows:

```
sqlite> INSERT INTO bike (bike_id, brand, size, purchase_price, purchase_date, status)
...> VALUES (1, 'GT', 53, 123.45, '2020-11-13', 'available');
sqlite> SELECT * FROM bike WHERE bike_id = 1;
1|GT|53|123.45|2020-11-13|available
```

You can delete rows:

```
sqlite> DELETE FROM bike WHERE bike_id = 1;
sqlite> SELECT * FROM bike WHERE bike_id = 1;
```

To get out of sqlite, type `.exit`.

Exercise 3 SQL Query

Execute an SQL query that returns the `bike_id` (no other columns) of every Trek bike that cost more than \$300.

Working Space

Answer on Page 27

3.1 Using SQL from Python

The people behind `sqlite` created a library for Python that lets you execute SQL and fetch the results from inside a python program.

Let's create a simple program that fetches and displays the bike ID and purchase date of every Trek bike that cost more than \$300.

Create a file called `report.py`:

```
import sqlite3 as db

con = db.connect('bikes.db')
cur = con.cursor()

cur.execute("SELECT bike_id, purchase_date FROM bike WHERE purchase_price > 330 AND brand='Trek'")
rows = cur.fetchall()

today = datetime.date.today()
for row in rows:
    print(f"Bike {row[0]}, purchased {row[1]}")

con.close()
```

When you execute it, you should see:

```
> python3 report.py
Bike 4128046, purchased 2007-08-06
Bike 7117808, purchased 1995-03-12
Bike 7176903, purchased 1986-07-03
Bike 827899, purchased 2009-03-14
```

Bike 363983, purchased 1970-08-16

Representing Natural Numbers

Natural numbers are positive whole numbers, such as 1, 2, 3, and so on. -5 is not a natural number. π is not a natural number. $\frac{1}{2}$ is not a natural number.

You are used to seeing the natural numbers represented in a base-10 *Hindu-Arabic* numeral system. That is, when you see 2531 you think “2 thousands, 5 hundreds, 3 tens, and 1 one.” Rewritten, this is:

$$2 \times 10^3 + 5 \times 10^2 + 3 \times 10^1 + 1 \times 10^0$$

In any Hindu-Arabic system, the location of the digits is meaningful: 101 is different from 110. Here are those numbers in Roman numerals: CI and CX. Roman numerals didn’t have a symbol for zero at all.

The Hindu-Arabic system gave us really straightforward algorithms for addition and multiplication. For addition, you memorized the following table:

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15 16	17	18	

When you multiplied two number together, you simply multiplied each pair of digits. 254×26 might look like this:

2	5	4	
×	2	6	
	2	4	6×4
	3	0	6×5
1	2		6×2
		8	2×4
1	0		2×5
+	4		2×2
6 6 0 4			

For multiplication, you memorized this table:

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
9	0	9	18	27	36	45	54	63	72	81

Answers to Exercises

Answer to Exercise 1 (on page 7)

Yes. Its eigenvalues are

2, 2

Answer to Exercise 2 (on page 7)

The answer depends on the 3 by 3 matrix you chose.

Answer to Exercise 3 (on page 22)

```
SELECT bike_id FROM bike WHERE purchase_price > 330 AND brand='Trek'
```




INDEX

null, [11](#)