7.9 PROJECTS

Project 7.1 Climate change

The causes and consequences of global warming are of intense interest. The consensus of the global scientific community is that the primary cause of global warming is an increased concentration of "greenhouse gasses," primarily carbon dioxide (CO_2) in the atmosphere. In addition, it is widely believed that human activity is the cause of this increase, and that it will continue into the future if we do not limit what we emit into the atmosphere.

To understand the causes of global warming, and to determine whether the increase in CO_2 is natural or human-induced, scientists have reconstructed ancient climate patterns by analyzing the composition of deep ocean sediment core samples. In a core sample, the older sediment is lower and the younger sediment is higher. These core samples contain the remains of ancient bottom-dwelling organisms called *foraminifera* that grew shells made of calcium carbonate ($CaCO_3$).

Virtually all of the oxygen in these calcium carbonate shells exists as one of two stable isotopes: oxygen-16 (^{16}O) and oxygen-18 (^{18}O). Oxygen-16 is, by far, the most common oxygen isotope in our atmosphere and seawater at about 99.76% and oxygen-18 is the second most common at about 0.2%. The fraction of oxygen-18 incorporated into the calcium carbonate shells of marine animals depends upon the temperature of the seawater. Given the same seawater composition, colder temperatures will result in a higher concentration of oxygen-18 being incorporated into the shells. Therefore, by analyzing the ratio of oxygen-18 to oxygen-16 in an ancient shell, scientists can deduce the temperature of the water at the time the shell formed. This ratio is denoted $\delta^{18}O$; higher values of $\delta^{18}O$ represent lower temperatures.

Similarly, it is possible to measure the relative amounts of carbon isotopes in calcium carbonate. Carbon can exist as one of two stable isotopes: carbon-12 (^{12}C) and carbon-13 (^{13}C). (Recall from Section 4.4 that carbon-14 (^{14}C) is radioactive and used for radiocarbon dating.) $\delta^{13}C$ is a measure of the ratio of carbon-13 to carbon-12. The value of $\delta^{13}C$ can decrease, for example, if there were a sudden injection of ^{12}C-rich (i.e., ^{13}C-depleted) carbon. Such an event would likely cause an increase in warming due to the increase in greenhouse gasses.

In this project, we will examine a large data set [68] containing over 17,000 $\delta^{18}O$ and $\delta^{13}C$ measurements from deep ocean sediment core samples, representing conditions over a period of about 65 million years. From this data, we will be able to visualize deep-sea temperature patterns over time (based on the $\delta^{18}O$ measurements) and the accompanying values of $\delta^{13}C$. We can try to answer two questions with this data. First, is there a correspondence between changes in carbon output and changes in temperature? Second, are recent levels of carbon in the atmosphere and ocean "natural," based on what has happened in the past?

Part 1: Read and plot the data

First, download the file named `2008CompilationData.csv` from the book website.[1] Examine the file and write a function that creates four parallel lists containing information about δ^{18}O and δ^{13}C readings: one list containing the δ^{18}O measurements, one list containing the δ^{13}C measurements, one list containing the sites of the measurements, and a fourth list containing the ages of the measurements. (Note that the `Age (ma)` column represents the ages of the samples in millions of years.) If a row is missing either the δ^{18}O value or the δ^{13}C value, do not include that row in your lists.

Using `matplotlib`, create scatter plots of the δ^{18}O and δ^{13}C measurements separately with respect to time (we will use the list of sites later). Be sure to appropriately label your axes. Because there is a lot of data here, your plots will be clearer if you make the dots small by passing `s = 1` into the `scatter` function. To create two plots in separate windows, precede the first plot with `pyplot.figure(1)` and precede the second plot with `pyplot.figure(2)`:

```
pyplot.figure(1)

# plot d18O data here

pyplot.figure(2)

# plot d13C data here

pyplot.show()  # after all the figures
```

Part 2: Smooth and plot the data

Inconsistencies in sampling, and local environmental variation, result in relatively noisy plots, but this can be fixed with a smoothing function that computes means in a window moving over the data. This is precisely the algorithm we developed in Section 7.2. Write a function that implements this smoothing algorithm, and use it to create three new lists — for δ^{18}O, δ^{13}C, and the ages — that are smoothed over a window of length 5.

To compare the oxygen and carbon isotope readings, it will be convenient to create two plots, one over the other, in the same window. We can do this with the `matplotlib` `subplot` function:

```
pyplot.figure(3)
pyplot.subplot(2, 1, 1) # arguments are (rows, columns, subplot #)

# plot d18O here

pyplot.subplot(2, 1, 2)

# plot d13C here
```

[1]This is a slightly "cleaned" version of the original data file from `http://www.es.ucsc.edu/~jzachos/Data/2008CompilationData.xls`

Part 3: The PETM

Looking at your smoothed data, you should now clearly see a spike in temperature (i.e., rapid decrease in δ^{18}O) around 55 ma. This event is known as the Palaeocene-Eocene Thermal Maximum (PETM), during which deep-sea temperatures rose 5° to 6° Celsius in fewer than 10,000 years!

To investigate further what might have happened during this time, let's "zoom in" on this period using data from only a few sites. Create three new lists, one of ages and the other two of measurements, that contain data only for ages between 53 and 57 ma and only for sites 527, 690, and 865. Sites 527 and 690 are in the South Atlantic Ocean, and site 865 is in the Western Pacific Ocean. (Try using list comprehensions to create the three lists.)

Plot this data in the same format as above, using two subplots.

Question 7.1.1 *What do you notice? What do your plots imply about the relationship between carbon and temperature?*

Part 4: Recent history

To gain insight into what "natural" CO_2 levels have looked like in our more recent history, we can consult another data set, this time measuring the CO_2 concentrations in air bubbles trapped inside ancient Antarctic ice [48]. Download this tab-separated data file from the book website, and write code to extract the data into two parallel lists — a list of ages and a list of corresponding CO_2 concentrations. The CO_2 concentrations are measured in parts per million by volume (ppmv) and the ages are in years.

Next, plot your data. You should notice four distinct, stable cycles, each representing a glacial period followed by an interglacial period. To view the temperature patterns during the same period, we can plot the most recent 420,000 years of our δ^{18}O readings. To do so, create two new parallel lists — one containing the δ^{18}O readings from sites 607, 659, and 849 for the last 420,000 years, and the other containing the corresponding ages. Arrange a plot of this data and your plot of CO_2 concentrations in two subplots, as before.

Question 7.1.2 *What do you notice? What is the maximum* CO_2 *concentration during this period?*

Part 5: Very recent history

In 1958, geochemist Charles David Keeling began measuring atmospheric CO_2 concentrations at the Mauna Loa Observatory (MLO) on the big island of Hawaii. These measurements have continued for the past 50 years, and are now known as the *Keeling curve*. Available on the book website (and at `https://scrippsco2.ucsd.edu/assets/data/atmospheric/stations/in_situ_co2/weekly/weekly_in_situ_co2_mlo.csv`) is a CSV data file containing weekly CO_2 concentration readings (in ppmv) since 1958. Read this data into two parallel lists, one containing the CO_2 concentration readings and the other containing the dates of the readings. Notice that the file contains a long header in which each line starts with a quotation mark. To read past this header, you can use the following `while` loop:

```
line = inputFile.readline()  # inputFile is your file object name
while line[0] == '"':
    line = inputFile.readline()
```

Each of the dates in the file is a string in `YYYY-MM-DD` format. To convert each of these date strings to a fractional year (e.g., `2013-07-01` would be `2013.5`), we can use the following formula:

```
year = float(date[:4]) + (float(date[5:7]) + float(date[8:10])/31) / 12
```

Next, plot the data.

Question 7.1.3 *How do these levels compare with the maximum level from the previous 420,000 years? Is your plot consistent with the pattern of "natural"* CO_2 *concentrations from the previous 420,000 years? Based on these results, what conclusions can we draw about the human impact on atmospheric* CO_2 *concentrations?*

Project 7.2 Does education influence unemployment?

Conventional wisdom dictates that those with more education have an easier time finding a job. In this project, we will investigate this claim by analyzing 2018 data from the U.S. Census Bureau.[2]

Part 1: Get the data

Download the data file from the book website and look at it in a text editor. You will notice that it is a tab-separated file with one line per U.S. metropolitan area. In each line are 60 columns containing population data broken down by educational attainment and employment status. There are four educational attainment categories: less than high school graduate (i.e., no high school diploma), high school graduate, some college or associate's degree, and college graduate.

Economists typically define unemployment in terms of the "labor force," people who are available for employment at any given time, whether or not they are actually employed. So we will define the unemployment rate to be the fraction of the (civilian) labor force that is unemployed. To compute the unemployment rate for each category of educational attainment, we will need data from the following ten columns of the file (column numbers start at 1):

Column	Contents
2	Name of metropolitan area
3	Total population of metropolitan area
11	No high school diploma, total in civilian labor force
15	No high school diploma, unemployed in civilian labor force
25	High school graduate, total in civilian labor force
29	High school graduate, unemployed in civilian labor force
39	Some college, total in civilian labor force
43	Some college, unemployed in civilian labor force
53	College graduate, total in civilian labor force
57	College graduate, unemployed in civilian labor force

Read this data from the file and store it in six lists that contain the names of the metropolitan areas, the total populations, and the unemployment rates for each of the four educational attainment categories. Each unemployment rate should be stored as a floating point value between 0 and 100, rounded to one decimal point. For example, 0.123456 should be stored as 12.3.

Next, use four calls to the `matplotlib plot` function to plot the unemployment rate data in a single figure with the metropolitan areas on the x-axis, and the unemployment rates for each educational attainment category on the y-axis. Be sure to label each plot and display a legend. You can place the names of the metropolitan areas on the x-axis with the `xticks` function:

[2] *Educational Attainment by Employment Status for the Population 25 to 64 Years* (Table B23006), U.S. Census Bureau, 2018.

```
pyplot.xticks(range(len(names)), names, rotation = 270, fontsize = 'small')
```

The first argument is the locations of the ticks on the x-axis, the second argument is a list of labels to place at those ticks, and the third and fourth arguments optionally rotate the text and change its size.

Part 2: Filter the data

Your plot should show data for 519 metropolitan areas. To make trends easier to discern, let's narrow the data down to the most populous areas. Create six new lists that contain the same data as the original six lists, but for only the thirty most populous metropolitan areas. Be sure to maintain the correct correspondence between values in the six lists. Generate the same plot as above, but for your six shorter lists.

Part 3: Analysis

Write a program to answer each of the following questions.

Question 7.2.1 *In which of the 30 metropolitan areas is the unemployment rate higher for HS graduates than for those without a HS diploma?*

Question 7.2.2 *Which of the 30 metropolitan areas have the highest and lowest unemployment rates for each of the four categories of educational attainment? Use a single loop to compute all of the answers, and do not use the built-in* `min` *and* `max` *functions.*

Question 7.2.3 *Which of the 30 metropolitan areas has the largest difference between the unemployment rates of college graduates and those with only a high school diploma? What is this difference?*

Question 7.2.4 *Print a formatted table that ranks the 30 metropolitan areas by the unemployment rates of their college graduates. (Hint: use a dictionary.)*

Project 7.3 Maximizing profit

Part 4 of this project assumes that you have read Section 7.6.

Suppose you are opening a new coffee bar, and need to decide how to price each shot of espresso. If you price your espresso too high, no customers will buy from you. On the other hand, if you price your espresso too low, you will not make enough money to sustain your business.

Part 1: Poll customers

To determine the most profitable price for espresso, you poll 1,000 potential daily customers, asking for the maximum price they would be willing to pay for a shot of espresso at your coffee bar. To simulate these potential customers, write a function

```
randCustomers(n)
```

that returns a list of `n` normally distributed prices with mean \$4.00 and standard deviation \$1.50. Use this function to generate a list of maximum prices for your 1,000 potential customers, and display of histogram of these maximum prices.

Part 2: Compute sales

Next, based on this information, you want to know how many customers would buy espresso from you at any given price. Write a function

```
sales(customers, price)
```

that returns the number of customers willing to buy espresso if it were priced at the given `price`. The first parameter `customers` is a list containing the maximum price that each customer is willing to pay. Then write another function

```
plotDemand(customers, lowPrice, highPrice, step)
```

that uses your `sales` function to plot a *demand curve*. A demand curve has price on the x-axis and the quantity of sales on the y-axis. The prices on the x-axis should run from `lowPrice` to `highPrice` in increments of `step`. Use this function to draw a demand curve for prices from free to \$8.00, in increments of a quarter.

Part 3: Compute profits

Suppose one pound (454 g) of roasted coffee beans costs you \$10.00 and you use 8 g of coffee per shot of espresso. Each "to go" cup costs you \$0.05 and you estimate that half of your customers will need "to go" cups. You also estimate that you have about \$500 of fixed costs (wages, utilities, etc.) for each day you are open. Write a function

```
profits(customers, lowPrice, highPrice, step, perCost, fixedCost)
```

that plots your profit at each price. Your function should return the maximum profit, the price at which the maximum profit is attained, and the number of customers who buy espresso at that price. Do not use the built-in `min` and `max` functions.

Question 7.3.1 *How should you price a shot of espresso to maximize your profit? At this* *price, how much profit do you expect each day? How many customers should you expect each day?*

**Part 4: Find the demand function*

If you have not already done so, implement the linear regression function discussed in Section 7.6. (See Exercise 7.6.1.) Then use linear regression on the data in your demand plot in Part 2 to find the linear function that best approximates the demand. This is called the *demand* function. Linear demand functions usually have the form

$$Q = b - m \cdot P,$$

where Q is the quantity sold and P is the price. Modify your `plotDemand` function so that it computes the regression line and plots it with the demand curve.

Question 7.3.2 *What is the linear demand function that best approximates your demand curve?*

*Project 7.4 Admissions

This project is a continuation of the problem begun in Section 7.6, and assumes that you have read that section.

Suppose you work in a college admissions office and would like to study how well admissions data (high school GPA and SAT scores) predict success in college.

Part 1: Get the data

We will work with a limited data source consisting of data for 105 students, available on the book website.[3] The file is named `sat.csv`.

Write a function

```
readData(fileName)
```

that returns four lists containing the data from `sat.csv`. The four lists will contain high school GPAs, math SAT scores, verbal (critical reading) SAT scores, and cumulative college GPAs. (This is an expanded version of the function in Exercise 7.6.3.)

Then a write another function

```
plotData(hsGPA, mathSAT, crSAT, collegeGPA)
```

that plots all of this data in one figure. We can do this with the `matplotlib` `subplot` function:

```
pyplot.figure(1)
pyplot.subplot(4, 1, 1) # arguments are (rows, columns, subplot #)

# plot HS GPA data here

pyplot.subplot(4, 1, 2)

# plot SAT math here

pyplot.subplot(4, 1, 3)

# plot SAT verbal here

pyplot.subplot(4, 1, 4)

# plot college GPA here
```

Question 7.4.1 *Can you glean any useful information from these plots?*

Part 2: Linear regression

As we discussed in Section 7.6, a linear regression is used to analyze how well an independent variable predicts a dependent variable. In this case, the independent variables are high school GPA and the two SAT scores. If you have not already, implement the linear regression function discussed in Section 7.6. (See Exercise 7.6.1.)

[3]Adapted from data available at `http://onlinestatbook.com/2/case_studies/sat.html`

Then use the `plotRegression` function from Section 7.6 to individually plot each independent variable, plus combined (math plus verbal) SAT scores against college GPA, with a regression line. (You will need four separate plots.)

Question 7.4.2 *Judging from the plots, how well do you think each independent variable predicts college GPA? Is there one variable that is a better predictor than the others?*

Part 3: Measuring fit

As explained in Exercise 7.6.4, the coefficient of determination (or R^2 coefficient) is a mathematical measure of how well a regression line fits a set of data. Implement the function described in Exercise 7.6.4 and modify the `plotRegression` function so that it returns the R^2 value for the plot.

Question 7.4.3 *Based on the R^2 values, how well does each independent variable predict college GPA? Which is the best predictor?*

Part 4: Additional analyses

Choose two of the independent variables and perform a regression analysis.

Question 7.4.4 *Explain your findings.*

*Project 7.5 Preparing for a 100-year flood

This project assumes that you have read Section 7.6.

Suppose we are undertaking a review of the flooding contingency plan for a community on the Snake River, just south of Jackson, Wyoming. To properly prepare for future flooding, we would like to know the river's likely height in a 100-year flood, the height that we should expect to see only once in a century. This 100-year designation is called the flood's *recurrence interval*, the amount of time that typically elapses between two instances of the river reaching that height. Put another way, there is a 1/100, or 1%, chance that a 100-year flood happens in any particular year.

River heights are measured by stream gauges maintained by the U.S. Geological Survey (USGS)[4]. A snippet of the data from the closest Snake River gauge, which can be downloaded from the USGS[5] or the book's website, is shown below.

```
#
# U.S. Geological Survey
# National Water Information System
 ⋮
#
agency_cd▷site_no▷peak_dt▷peak_tm▷peak_va▷peak_cd▷gage_ht▷...
5s▷15s▷10d▷6s▷8s▷27s▷8s▷...
USGS▷13018750▷1976-06-04▷▷15800▷6▷7.80▷...
USGS▷13018750▷1977-06-09▷▷11000▷6▷6.42▷...
USGS▷13018750▷1978-06-10▷▷19000▷6▷8.64▷...
 ⋮
USGS▷13018750▷2011-07-01▷▷19900▷6▷8.75▷...
USGS▷13018750▷2012-06-06▷▷16500▷6▷7.87▷...
```

The file begins with several comment lines preceded by the hash (`#`) symbol. The next two lines are header rows; the first contains the column names and the second contains codes that describe the content of each column, e.g., 5s represents a string of length 5 and 10d represents a date of length 10. Each column is separated by a tab character, represented above by a right-facing triangle (▷). The header rows are followed by the data, one row per year, representing the peak event of that year. For example, in the first row we have:

- `agency_cd` (agency code) is `USGS`
- `site_no` (site number) is `13018750` (same for all rows)
- `peak_dt` (peak date) is `1976-06-04`
- `peak_tm` (peak time) is omitted
- `peak_va` (peak streamflow) is `15800` cubic feet per second
- `peak_cd` (peak code) is `6` (we will ignore this)

[4] `http://nwis.waterdata.usgs.gov/nwis`
[5] `http://nwis.waterdata.usgs.gov/nwis/peak?site_no=13018750&agency_cd=USGS&format=rdb`

- `gage_ht` (gauge height) is `7.80` feet

So for each year, we essentially have two gauge values: the peak streamflow in cubic feet per second and the maximum gauge height in feet.

If we had 100 years of gauge height data in this file, we could approximate the water level of a 100-year flood with the maximum gauge height value. However, our data set only covers 37 years (1976 to 2012) and, for 7 of those years, the gauge height value is missing. Therefore, we will need to estimate the 100-year flood level from the limited data we are given.

Part 1: Read the data

Write a function

```
readData(filename)
```

that returns lists of the peak streamflow and gauge height data (as floating point numbers) from the Snake River data file above. Your function will need to first read past the comment section and header lines to get to the data. Because we do not know how many comment lines there might be, you will need to use a `while` loop containing a call to the `readline` function to read past the comment lines.

Notice that some rows in the data file are missing gauge height information. If this information is missing for a particular line, use a value of 0 in the list instead.

Your function should return two lists, one containing the peak streamflow rates and one containing the peak gauge heights. A function can return two values by simply separating them with a comma, .e.g.,

```
return flows, heights
```

Then, when calling the function, we need to assign the function call to two variable names to capture these two lists:

```
flows, heights = readData('snake_peak.txt')
```

Part 2: Recurrence intervals

To associate the 100-year recurrence interval with an estimated gauge height, we can associate each of our known gauge heights with a recurrence interval, plot this data, and then use regression to extrapolate out to 100 years. A flood's recurrence interval is computed by dividing $(n + 1)$, where n is the number of years on record, by the rank of the flood. For example, suppose we had only three gauge heights on record. Then the recurrence interval of the maximum (rank 1) height is $(3 + 1)/1 = 4$ years, the recurrence interval of the second largest height is $(3 + 1)/2 = 2$ years, and the recurrence interval of the smallest height is $(3 + 1)/3 = 4/3$ years. (However, these inferences are unlikely to be at all accurate because there is so little data!)

Write a function

```
getRecurrenceIntervals(n)
```

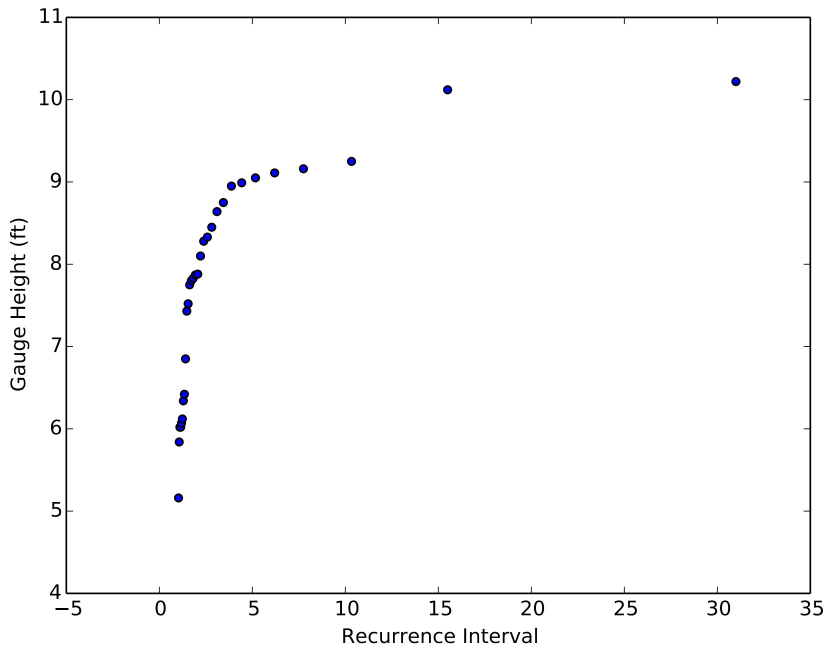


Figure 1 The peak gauge height for each recurrence interval.

that returns a list of recurrence intervals for `n` floods, in order of lowest to highest. For example if `n` is 3, the function should return the list `[1.33, 2.0, 4.0]`.

After you have written this function, write another function

```
plotRecurrenceIntervals(heights)
```

that plots recurrence intervals and corresponding gauge heights (also sorted from smallest to largest). Omit any missing gauge heights (with value zero). Your resulting plot should look like Figure 1.

Part 3: Find the river height in a 100-year flood

To estimate the gauge height corresponding to a 100-year recurrence interval, we need to extend the "shape" of this curve out to 100 years. Mathematically speaking, this means that we need to find a mathematical function that predicts the peak gauge height for each recurrence interval. Once we have this function, we can plug in 100 to find the gauge height for a 100-year flood.

What we need is a regression analysis, as we discussed in Section 7.6. But linear regression only works properly if the data exhibits a linear relationship, i.e., we can draw a straight line that closely approximates the data points.

Question 7.5.1 *Do you think we can use linear regression on the data in Figure 1?*

This data in Figure 1 clearly do not have a linear relationship, so a linear regression

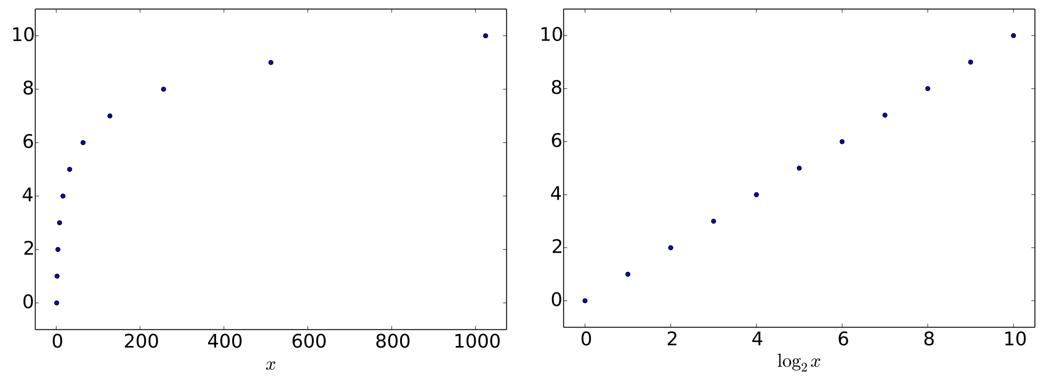


Figure 2 On the left is a plot of the points $(2^0, 0), (2^1, 1), (2^2, 2), \ldots, (2^{10}, 10)$, and on the right is a plot of the points that result from taking the logarithm base 2 of the x coordinate of each of these points.

will not produce a good approximation. The problem is that the x coordinates (recurrence intervals) are increasing multiplicatively rather than additively; the recurrence interval for the flood with rank $r + 1$ is $(r + 1)/r$ times the recurrence interval for the flood with rank r. However, we will share a trick that allows us to use linear regression anyway. To illustrate the trick we can use to turn this non-linear curve into a "more linear" one, consider the plot on the left in Figure 2, representing points $(2^0, 0), (2^1, 1), (2^2, 2), \ldots, (2^{10}, 10)$. Like the plot in Figure 1, the x coordinates are increasing multiplicatively; each x coordinate is twice the one before it. The plot on the right in Figure 2 contains the points that result from taking the logarithm base 2 of each x coordinate ($\log_2 x$), giving $(0, 0), (1, 1), (2, 2), \ldots, (10, 10)$. Notice that this has turned an exponential plot into a linear one.

We can apply this same technique to the `plotRecurrenceIntervals` function you wrote above to make the curve approximately linear. Write a new function

```
plotLogRecurrenceIntervals(heights)
```

that modifies the `plotRecurrenceIntervals` function so that it makes a new list of logarithmic recurrence intervals, and then computes the linear regression line based on these x values. Use logarithms with base 10 for convenience. Then plot the data and the regression line using the `linearRegression` function from Exercise 7.6.1. To find the 100-year flood gauge height, we want the regression line to extend out to 100 years. Since we are using logarithms with base 10, we want the x coordinates to run from $\log_{10} 1 = 0$ to $\log_{10} 100 = 2$.

Question 7.5.2 *Based on Figure 2, what is the estimated river level for a 100-year flood? How can you find this value exactly in your program? What is the exact value?*

Part 4: An alternative method

As noted earlier in this section, there are seven gauge height values missing from the data file. But all of the peak streamflow values are available. If there is a linear correlation between peak streamflow and gauge height, then it might be more accurate to find the 100-year peak streamflow value instead, and then use the linear regression between peak streamflow and gauge height to find the gauge height that corresponds to the 100-year peak streamflow value.

First, write a function

```
plotFlowsHeights(flows, heights)
```

that produces a scatter plot with peak streamflow on the x-axis and the same year's gauge height on the y axis. Do not plot data for which the gauge height is missing. Then also plot the least squares linear regression for this data.

Next, write a function

```
plotLogRecurrenceIntervals2(flows)
```

that modifies the `plotLogRecurrenceIntervals` function from Part 3 so that it find the 100-year peak streamflow value instead.

Question 7.5.3 *What is the 100-year peak streamflow rate?*

Once you have found the 100-year peak streamflow rate, use the linear regression formula to find the corresponding 100-year gauge height.

Question 7.5.4 *What is the gauge height that corresponds to the 100-year peak streamflow rate?*

Question 7.5.5 *Compare the two results. Which one do you think is more accurate? Why?*

Project 7.6 Voting methods

Although most of us are used to simple plurality-based elections in which the candidate with the most votes wins, there are other voting methods that have been proven to be fairer according to various criteria. In this project, we will investigate two other such voting methods. For all of the parts of the project, we will assume that there are four candidates named Amelia, Beth, Caroline, and David (abbreviated A, B, C, and D).

Part 1: Get the data

The voting results for an election are stored in a data file containing one ballot per line. Each ballot consists of one voter's ranking of the candidates. For example, a small file might look like:

```
B A D C
D B A C
B A C D
```

To begin, write a function

```
readVotes(fileName)
```

that returns these voting results as a list containing one list for each ballot. For example, the file above should be stored in a list that looks like this:

```
[['B', 'A', 'D', 'C'], ['D', 'B', 'A', 'C'], ['A', 'B', 'C', 'D']]
```

There are three sample voting result files on the book website. Also feel free to create your own.

Part 2: Plurality voting

First, we will implement basic plurality voting. Write a function

```
plurality(ballots)
```

that prints the winner (or winners if there is a tie) of the election based on a plurality count. The parameter of the function is a list of ballots like that returned by the `readVotes` function. Your function should first iterate over all of the ballots and count the number of first-place votes won by each candidate. Store these votes in a dictionary containing one entry for each candidate. To break the problem into more manageable pieces, write a "helper function"

```
printWinners(points)
```

that determines the winner (or winners if there is a tie) based on this dictionary (named `points`), and then prints the outcome. Call this function from your `plurality` function.

Part 3: Borda count

Next, we will implement a vote counting system known as a Borda count, named after Jean-Charles de Borda, a mathematician and political scientist who lived in 18th century France. For each ballot in a Borda count, a candidate receives a number of points equal to the number of lower-ranked candidates on the ballot. In other words, with four candidates, the first place candidate on each ballot is awarded three points, the second place candidate is awarded two points, the third place candidate is awarded one point, and the fourth place candidate receives no points. In the example ballot above, candidate B is the winner because candidate A receives $2 + 1 + 2 = 5$ points, candidate B receives $3 + 2 + 3 = 8$ points, candidate C receives $0 + 0 + 1 = 1$ points, and candidate D receives $1 + 3 + 0 = 4$ points. Note that, like a plurality count, it is possible to have a tie with a Borda count.

Write a function

```
borda(ballots)
```

that prints the winner (or winners if there is a tie) of the election based on a Borda count. Like the `plurality` function, this function should first iterate over all of the ballots and count the number of points won by each candidate. To make this more manageable, write another "helper function" to call from within your loop named

```
processBallot(points, ballot)
```

that processes each individual ballot and adds the appropriate points to the dictionary of accumulated points named `points`. Once all of the points have been accumulated, call the `printWinners` above to determine the winner(s) and print the outcome.

Part 4: Condorcet voting

Marie Jean Antoine Nicolas de Caritat, Marquis de Condorcet was another mathematician and political scientist who lived about the same time as Borda. Condorcet proposed a voting method that he considered to be superior to the Borda count. In this method, every candidate participates in a virtual head-to-head election between herself and every other candidate. For each ballot, the candidate who is ranked higher wins. If a candidate wins every one of these head-to-head contests, she is determined to be the Condorcet winner. Although this method favors the candidate who is most highly rated by the majority of voters, it is also possible for there to be no winner.

Write a function

```
condorcet(ballots)
```

that prints the Condorcet winner of the election or indicates that there is none. (If there is a winner, there can only be one.)

Suppose that the list of candidates is assigned to `candidates`. (Think about how you can get this list.) To simulate all head-to-head contests between one candidate named `candidate1` and all of the rest, we can use the following `for` loop:

```
for candidate2 in candidates:
    if candidate2 != candidate1:
        # head-to-head between candidate1 and candidate2 here
```

This loop iterates over all of the candidates and sets up a head-to-head contest between each one and `candidate1`, as long as they are not the same candidate.

Question 7.6.1 *How can we now use this loop to generate contests between all pairs of candidates?*

To generate all of the contests with all possible values of `candidate1`, we can nest this `for` loop in the body of another `for` loop that also iterates over all of the candidates, but assigns them to `candidate1` instead:

```
for candidate1 in candidates:
    for candidate2 in candidates:
        if candidate2 != candidate1:
            # head-to-head between candidate1 and candidate2 here
```

Question 7.6.2 *This nested* `for` *loop actually generates too many pairs of candidates. Can you see why?*

To simplify the body of the nested loop (where the comment is currently), write another "helper function"

```
head2head(ballots, candidate1, candidate2)
```

that returns the candidate that wins in a head-to-head vote between `candidate1` and `candidate2`, or `None` is there is a tie. Your `condorcet` function should call this function for every pair of different candidates. For each candidate, keep track of the number of head-to-head wins in a dictionary with one entry per candidate.

Question 7.6.3 *The most straightforward algorithm to decide whether* `candidate1` *beats* `candidate2` *on a particular ballot iterates over all of the candidates on the ballot. Can you think of a way to reorganize the ballot data before calling* `head2head` *so that the* `head2head` *function can decide who wins each ballot in only one step instead?*

Part 5: Compare the three methods

Execute your three functions on each of the three data files on the book website and compare the results.

Project 7.7 Heuristics for traveling salespeople

Imagine that you drive a delivery truck for one of the major package delivery companies. Each day, you are presented with a list of addresses to which packages must be delivered. Before you can call it a day, you must drive from the distribution center to each package address, and back to the distribution center. This cycle is known as a ***tour***. Naturally, you wish to minimize the total distance that you need to drive to deliver all the packages, i.e., the total length of the tour. For example, Figures 3 and 4 show two different tours.

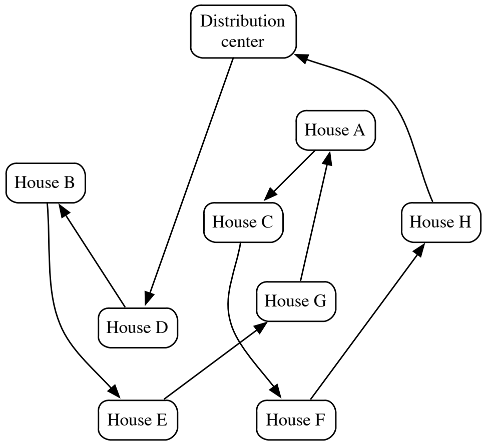


Figure 3 An inefficient tour.

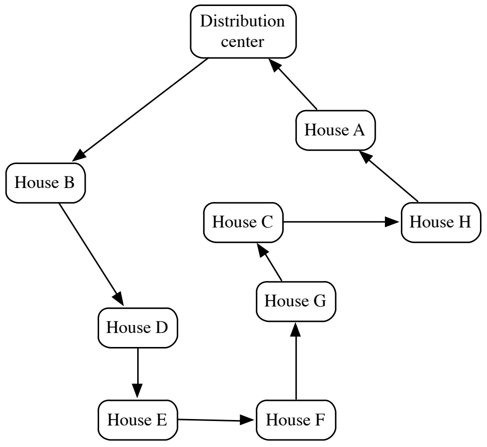


Figure 4 A more efficient tour.

This is known as the *traveling salesperson problem (TSP)*, and it is notoriously difficult. In fact, as far as anyone knows, the only way to come up with a guaranteed correct solution is to essentially enumerate all possible tours and choose the best one. But since, for n locations, there are $n!$ (n factorial) different tours, this is practically impossible.

Unfortunately, the TSP has many important applications, several of which seem at first glance to have nothing at all to do with traveling or salespeople, including circuit board drilling, controlling robots, designing networks, x-ray crystallography, scheduling computer time, and assembling genomes. In these cases, a heuristic must be used. A heuristic does not necessarily give the best answer, but it tends to work well in practice

For this project, you will design your own heuristic, and then work with a genetic algorithm, a type of heuristic that mimics the process of evolution to iteratively improve problem solutions.

Part 1: Write some utility functions

Each point on your itinerary will be represented by (x, y) coordinates, and the input to the problem is a list of these points. A tour will also be represented by a list of points; the order of the points indicates the order in which they are visited. We will store a list of points as a list of tuples.

The following function reads in points from a file and returns the points as a list of tuples. We assume that the file contains one point per line, with the x and y coordinates separated by a space.

```
def readPoints(filename):
    inputFile = open(filename, 'r')
    points = []
    for line in inputFile:
        values = line.split()
        points.append((float(values[0]), float(values[1])))
    return points
```

To begin, write the following three functions. The first two will be needed by your heuristics, and the third will allow you to visualize the tours that you create. To test your functions, and the heuristics that you will develop below, use the example file containing the coordinates of 96 African cities (`africa.tsp`) on the book website.

1. `distance(p, q)` returns the distance between points `p` and `q`, each of which is stored as a two-element tuple.
2. `tourLength(tour)` returns the total length of a tour. The tour is stored as a list of tuples. Remember to include the distance from the last point back to the first point.
3. `drawTour(tour)` draws a tour using turtle graphics. Use the `setworldcoordinates` method to make the coordinates in your drawing window more closely match the coordinates in the data files you use. For example, for the `africa.tsp` data file, the following will work well:

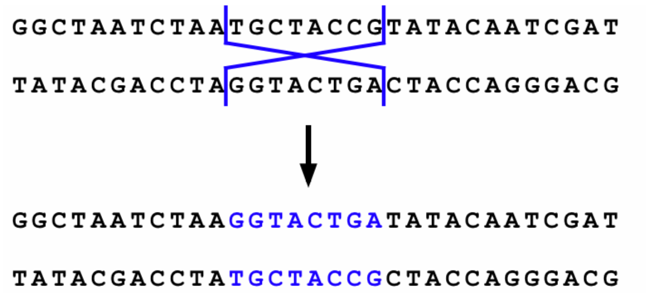
   ```
   screen.setworldcoordinates(-40, -25, 40, 60)
   ```

Part 2: Design a heuristic

Now design your own heuristic to find a good TSP tour. There are many possible ways to go about this. Be creative. Think about what points should be next to each other. What kinds of situations should be fixed? Use the `drawTour` function to visualize your tours and help you design your heuristic.

Part 3: A genetic algorithm

A ***genetic algorithm*** attempts to solve a hard problem by emulating the process of evolution. The basic idea is to start with a population of feasible solutions to the problem, called individuals, and then iteratively try to improve the fitness of this population through the evolutionary operations of recombination and mutation. In genetics, recombination is when chromosomes in a pair exchange portions of their DNA during meiosis. The illustration below shows how a crossover would affect the bases in a particular pair of (single stranded) DNA molecules.


GGCTAATCTAATGCTACCGTATACAATCGAT
TATACGACCTAGGTACTGACTACCAGGGACG
GGCTAATCTAAGGTACTGATATACAATCGAT
TATACGACCTATGCTACCGCTACCAGGGACG

Mutation occurs when a base in a DNA molecule is replaced with a different base or when bases are inserted into or deleted from a sequence. Most mutation is the result of DNA replication errors but environmental factors can also lead to mutations in DNA.

To apply this technique to the traveling salesperson problem, we first need to define what we mean by an individual in a population. In genetics, an individual is represented by its DNA, and an individual's fitness, for the purposes of evolution, is some measure of how well it will thrive in its environment. In the TSP, we will have a population of tours, so an individual is one particular tour — a list of cities. The most natural fitness function for an individual is the length of the tour; a shorter tour is more fit than a longer tour.

Recombination and mutation on tours are a bit trickier conceptually than they are for DNA. Unlike with DNA, swapping two subsequences of cities between two tours is not likely to produce two new valid tours. For example, suppose we have two tours `[a, b, c, d]` and `[b, a, d, c]`, where each letter represents a point. Swapping the two middle items between the tours will produce the offspring `[a, a, d, d]` and `[b, b, c, c]`, neither of which are permutations of the cities. One way around this is to delete from the first tour the cities in the portion to be swapped from the second tour, and then insert this portion from the second tour. In the above example, we would delete points `a` and `d` from the first tour, leaving `[b, c]`, before inserting `[a, d]` in the middle. Doing likewise for the second tour gives us children `[b, a, d, c]` and `[a, b, c, d]`. But we are not limited in genetic programming to recombination that more or less mimics that found in nature. A recombination operation can be anything that creates new offspring by somehow combining two parents. A large part of this project involves brainstorming about and experimenting with different techniques.

We must also rethink mutation since we cannot simply replace an arbitrary city with another city and end up with a valid tour. One idea might be to swap the positions of two randomly selected cities instead. But there are other possibilities as well.

Your mission is to improve upon a baseline genetic algorithm for TSP. Be creative! You may change anything you wish as long as the result can still be considered a genetic algorithm. To get started, download the baseline program from the book website. Try running it with the 96-point instance on the book website. Take some time to understand how the program works. Ask questions. You may want to refer to the Python documentation if you don't recall how a particular function works. Most of the work is performed by the following four functions:

- `makePopulation(cities)`: creates an initial population (of random tours)
- `crossover(mom, pop)`: performs a recombination operation on two tours and returns the two offspring
- `mutate(individual)`: mutates an individual tour
- `newGeneration(population)`: update the population by performing a crossover and mutating the offspring

Write the function

```
histogram(population)
```

that is called from the `report` function. (Use a Python dictionary.) This function should print a frequency chart (based on tour length) that gives you a snapshot of the diversity in your population. Your histogram function should print something like this:

```
Population diversity
   1993.2714596455853 : ****
   2013.1798076309087 : **
   2015.1395212505120 : ****
   2017.1005248468230 : ********************************
   2020.6881282400334 : *
   2022.9044855489917 : *
   2030.9623523675089 : *
   2031.4773010231959 : *
   2038.0257926528227 : *
   2040.7438913120230 : *
   2042.8148398732630 : *
   2050.1916058477627 : *
```

This will be very helpful as you strive to improve the algorithm: recombination in a homogeneous population is not likely to get you very far.

Brainstorm ways to **improve the algorithm**. Try lots of different things, ranging from tweaking parameters to completely rewriting any of the four functions described above. You are free to change anything, as long as the result still resembles a genetic algorithm. Keep careful records of what works and what doesn't to include in your submission.

On the book website is a link to a very good reference [51] that will help you think of new things to try. Take some time to skim the introductory sections, as they will give you a broader sense of the work that has been done on this problem. Sections 2 and 3 contain information on genetic algorithms; Section 5 contains information on various recombination/crossover operations; and Section 7 contains information on possible mutation operations. As you will see, this problem has been well studied by researchers over the past few decades! (To learn more about this history, we recommend *In Pursuit of the Traveling Salesman* by William Cook [10].)