Homework 1

Course: Introduction to applied data science (PHYS247) — Spring 2020

TA: Nima Chartab — Due Date: April 13, 12 p.m.

Problem 1: Linear algebra

We want to dial an N-digit number using a dial pad shown below. In every step, we are just allowed to move either vertically or horizontally to one of the neighbor numbers. For example, the red lines in the figure show three possible paths from number 2 and blues show two possible paths from number 7. So, if we start from number 2 and want to dial a 2-digit number, we just need one move and can only dial 21, 23 and 25. Starting from number 0, we have only one possible move to number 8. The question is if we start from number S and want to dial an N-digits number, how many distinct numbers, $\Omega(S, N)$, can be dialed? For instance, $\Omega(0, 2) = 1$, and $\Omega(2, 2) = 3$.
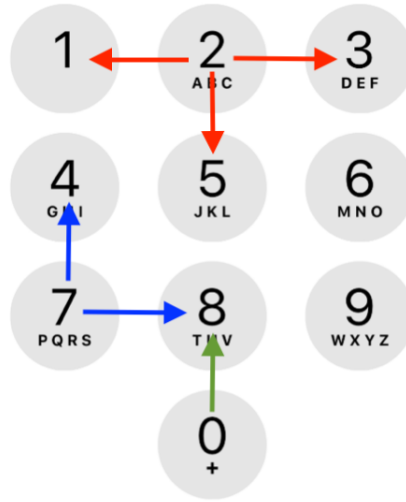
Let's solve this question step-by-step using linear algebra. Consider a 10×10 matrix which is defined as follows:

$$A_{ij} = \begin{cases} 1 & \text{if i to j is an allowed move} \\ 0 & \text{if i to j is a forbidden move} \end{cases}$$

For clarity, let's identify all elements of the matrix. $A_{00} = 0$, since moving from number 0 to itself is forbidden. The only possible move from 0 is to 8, so $A_{08} = 1$ and $A_{01} = A_{02} = A_{03} = A_{04} = A_{05} = A_{06} = A_{07} = A_{08} = 0$. Keep doing it for all other rows of the matrix and construct A as follows:

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Imagine we have started from number S, so $N - 1$ digits are left to be dialed. Starting from S, we can only dial an allowed set of numbers, $\{j | A_{Sj} = 1\}$. Therefore, $\Omega(S, N)$ can be calculated

1

recursively by adding $\Omega(\mathrm{k}, N-1)$ values for $k \in \{j|A_{Sj}=1\}$.

$$\Omega(S, N) = \sum_{k \in \{j|A_{Sj}=1\}} \Omega(k, N-1)$$

> **a) Show that the above equation can be written in a form of matrix such that $\Omega(S, N)$ is S^{th} element of a 10×1 matrix (Ω_N):**
> $$\Omega(N) = A \times \Omega(N-1)$$
> **Once $\Omega(N)$ is calculated, $\Omega(S, N)$ can be found by reading S^{th} element of $\Omega(N)$ matrix, $\Omega(S, N) = \Omega(N)_S$. For instance, 0th element of $\Omega(2)$ is $\Omega(0, 2) = 1$.**

Note that $\Omega(1)$ corresponds to the starting point before we move to the second digit of the number. So, It is a 10×1 matrix with all the elements equal to 1.

$$\Omega(1) = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

> **b) Show that $\Omega(N)$ can be found recursively using the following equation:**
> $$\Omega(N) = A \times ... \times A \times A \times \Omega(1) = A^{N-1} \times \Omega(1)$$

That's it! We solved the problem. Let's write a Python code to find $\Omega(N)$ for a given N.

> **c) Create a 10×10 NumPy matrix which contains all the elements we have found for A. Find the transpose of the matrix using `numpy.transpose`. Are the matrix and its transpose equal? Did you expect this result?**
>
> **d) Define a function with the name of "Dialer_Problem" which takes two parameters, S and N and returns $\Omega(S, N)$. Within the function, use a "for loop" to multiply $\Omega(1)$ to A and then A and so on. Test your function for a few examples. For instance, running `Dialer_Problem(5,10)` should return 18713.**
>
> **e) Rewrite the same code without using NumPy package. You will need to create a nested list which includes all the elements in A and also define a function which performs matrix multiplication.**

Let's evaluate the algorithmic efficiency of our code. We can use Python's time module to evaluate the time needed to execute the code for a given N. The figure below shows the execution time as a function of N while running the code on my own laptop. The time increases linearly with the number of digits, so in Big O notation, its performance can be written as $\mathcal{O}(N)$.

---
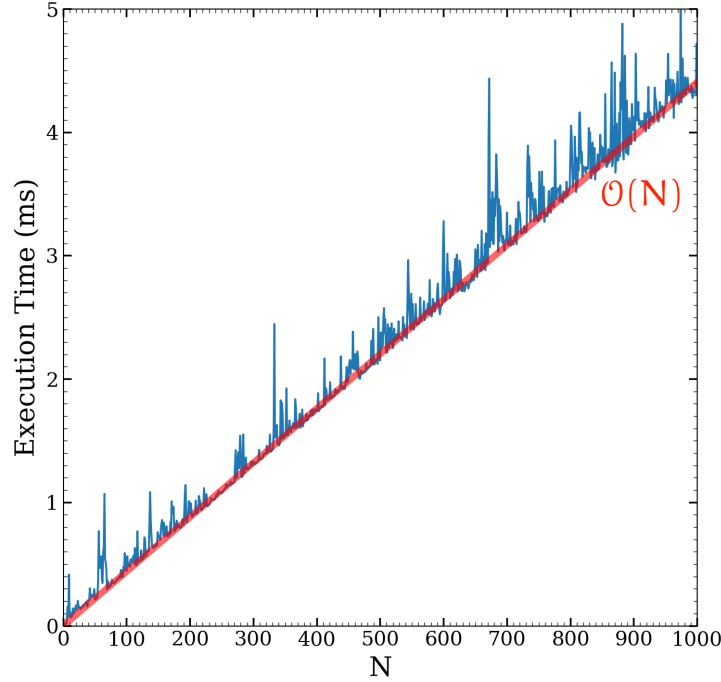
f) Using time module of Python, evaluate the algorithmic efficiency of your code in part d. You need to make the similar figure shown above on your computer.

g) How long will running your code for $N = 10^7$ take? You do not run your code for this big number of digits! Use extrapolation to estimate the running time. Do you think we need to improve the performance of our code to execute it for large numbers?

Let's improve the performance of the code. We can express any number in the base-2 numeral system ("0" and "1"). Any decimal number can be expressed with a series of zeroes and ones (a_i):

$$\text{Decimal number} = \sum a_i 2^i$$

For example, $22 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$. So, 22 in the binary system is 10110.

h) Write your own function which takes a decimal number and return the number in the binary system. Do not use `bin()` syntax of python.

In part b, you derived that $\Omega(N) = A^{N-1} \times \Omega(1)$, so writing $N - 1$ in the binary system, $N - 1 = \sum a_j 2^j$, we get:

$$\Omega(N) = A^{N-1} \times \Omega(1) = A^{\sum a_j 2^j} \times \Omega(1) = \prod (A^{2^j})^{a_j} \times \Omega(1) = \ldots \times (A^4)^{a_2} \times (A^2)^{a_1} \times \Omega(1)$$

For clarity if we want to find $\Omega(23)$, it can be expanded as follows:

$$\Omega(23) = A^{22} \times \Omega(1) = (A^{16})^1 \times (A^8)^0 \times (A^4)^1 \times (A^2)^1 \times \Omega(1)$$

So, we just need to start from computing A^2 and multiplying it by itself to get A^4 and again multiply by itself to get A^8 and so on. Instead of doing 22 matrix multiplications, we only do 6 multiplications to get A^{22}.

i) Implement the mentioned algorithm to rebuild your function in part d. This algorithm will allow us to speed up the execution time of the program. You will need to use your binary convert function in part h.

j) Evaluate the algorithmic efficiency of your code in part h in the same way you did in part f. I implemented the algorithm and ran it on my own computer and I have added my result below for your reference.

k) How long will running your code for $N = 10^7$ take using this algorithm? Compare your finding with part g. Algorithm matters!
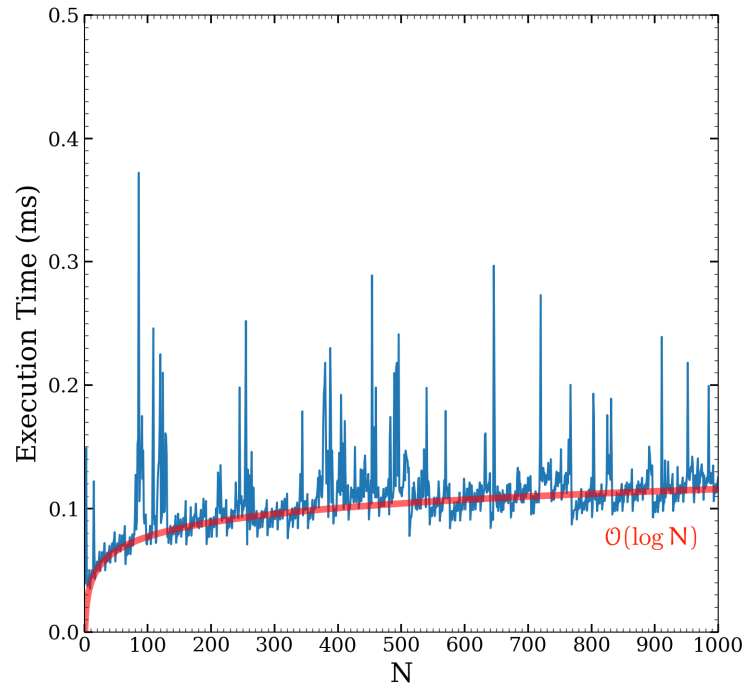
Problem 2: A simple dimensionality reduction

In this problem, we will use a principal component analysis(PCA) to map a 2-D data set to 1-D space. Consider a set of 10, 2-D data points given in a 10×2 matrix below:

$$D = \begin{pmatrix} 1.72 & 0.10 \\ -0.58 & 0.31 \\ 1.99 & 1.54 \\ 1.60 & 4.97 \\ 2.72 & 2.43 \\ 2.83 & 3.59 \\ 5.95 & 7.71 \\ 4.75 & 7.63 \\ 5.57 & 5.60 \\ 9.82 & 7.91 \end{pmatrix}$$

The covariance matrix describes how the two variables (first column and second column of matrix D) change together. We want to map our data to a vector which directs towards the most variance.

This direction can be translated to finding the eigenvector of covariance matrix with the largest eigenvalue.

a) Create a 10×2 NumPy array which contains all elements in D. Find the covariance matrix of D using `numpy.cov`.

b) Find eigenvalues and corresponding eigenvectors of the covariance matrix using NumPy linear algebra function. Which eigenvector shows the direction with the most variance? Name this vector v.

c) Use the dot product of matrix D and v to map all the data points to a direction which corresponds to the most variance. Now, you have a 1-D data set instead of 2-D and you preserved most of the information regarding the relative position of data points in 2-D space. In other words, the data points which are close in 2-D space, are still close pairs in 1-D space. The figure below shows my own results for eigenvectors (red and blue) of the covariance matrix along with the data points. Mapping the data points to the red eigenvector (the one with larger eigenvalue) is shown in the right figure. Plotting your results is not required in this question but highly recommended.