Problem Set 1

All parts are due Thursday, September 18 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 1-1. [20 points] **Asymptotic notation**

Arrange the following functions in increasing order of growth. That is, arrange them as a sequence $f_1, f_2, \ldots$ such that $f_1 = O(f_2), f_2 = O(f_3), f_3 = O(f_4), \ldots$. Also indicate if two functions f and g have the same asymptotic growth, that is if $f = \Theta(g)$.

$$\begin{array}{lll} \binom{n}{100}, & 3^n, & n^{100}, \\ 1/n, & 2^{2n}, & 10^{100}n, \\ 3^{\sqrt{n}}, & 1/5, & 4^n, \\ n\log n, & \log(n!), & n2^n, \\ 2^{n+1}, & n!, & 2^n. \\ \log(n), & n^{1/100}, & \log(\log(n)). \end{array}$$

Problem 1-2. [20 points] **Recurrence**

Derive the asymptotic growth of functions $T(n)$ defined by the each of the following recurrences (give the answer first, and then show your work).

(a) [5 points] What is the asymptotic expression for $T(n)$ defined as follows?

$$T(n) = 4T(n/2) + \log n$$

(b) [5 points] What is the asymptotic expression for $T(n)$ defined as follows?

$$T(n) = 9T(n/3) + n^2$$

(c) [5 points] What is the asymptotic expression for $T(n)$ defined as follows? Here, S is an auxiliary function, used to help define T.

$$T(n) = S(n, n)$$

$$S(x, y) = \begin{cases} \Theta(1) & : x \leq 2, y \leq 2 \\ \Theta(x) & : x > 2, y \leq 2 \\ \Theta(y) & : x \leq 2, y > 2 \\ S(x/2, y/2) + x + y & : x > 2, y > 2 \end{cases}$$

(d) [5 points] What is the asymptotic expression for $T(n)$ defined as follows? Here, S is an auxiliary function, used to help define T.

$$T(n) = S(n, n)$$

$$S(x, y) = \begin{cases} \Theta(1) & : x \leq 2, y \leq 2 \\ \Theta(x) & : x > 2, y \leq 2 \\ \Theta(y) & : x \leq 2, y > 2 \\ S(x, y/2) + x & : x > 2, y > 2 \end{cases}$$

Problem 1-3. [20 points] **Temperature recordings**

Let $a_0, a_1, \ldots, a_{n-1}$ be integers representing temperature measurements on n evenly distributed locations labeled $0, 1, \ldots, n-1$, around the Earth's equator. Location i is geographically close to location $i+1$, and since they lie in a circle, location $n-1$ is geographically close to location 0.

Given their proximity, it is reasonable that the temperatures a_i and a_{i+1} are close to each other. In fact, we will assume that their temperature difference is at most one. That is,

$$|a_i - a_{i+1}| \leq 1 \text{ (for } i = 0, \ldots, n-2) \text{ and also, } |a_{n-1} - a_0| \leq 1.$$

Suppose that n is even and call two points i, j opposite if $|i - j| = n/2 \mod n$. (Clearly, two opposite points are exactly opposite each other on the planet.)

(a) [5 points] For a point i, let $D(i)$ denote the difference between the temperature at point i and its opposite point $j := i + n/2 \mod n$. That is,

$$D(i) := a_i - a_j.$$

Argue that there are always opposite points with temperate difference bounded by 1. That is, that there is some i such that $|D(i)| \leq 1$.

Hint: It may be useful to picture a "continuous" version of the problem first, where the temperatures are recorded continuously on the perimeter of a circle. How does $D(i)$ relate to $D(j)$? Can $D(i)$ be continuous and avoid attaining value zero somewhere?

(b) [10 points] Describe an algorithm that finds such a pair in time $O(\log n)$, and argue that it achieves the correct running time.

(c) [5 points] Prove that the algorithm you described in the previous part is correct.

Hint: You may want to identify an invariant maintained throughout the running time of your algorithm.

Part B

Problem 1-4. [40 points] **Simple substitution ciphers**

One of the first and simplest encryption schemes in cryptography is attributed to Julius Caesar. The Roman historian Suetonius in "Life of Julius Caesar" writes

> "If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others."

We see that a Caesar cipher uses a key $K \in \{0, \ldots, 25\}$ and shifts every letter of the alphabet down by K positions (and wraps around if necessary).

For example, a shift of $K = 3$ would replace 'a' with 'd', 'b' with 'e', and 'z' with 'c', etc. Applied on a whole sentence, the phrase

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

with the same key transforms into

```
WKH TXLFN EURZQ IRA MXPSV RYHU WKH ODCB GRJ.
```

(note the convention that spaces and punctuation marks are left unchanged, and upper/lower case state of each letter is preserved).

While Caesar cipher is useful for certain practical situations, for example in order to hide spoilers and puzzle solutions (with the commonly used key of $K = 13$ which makes the Cipher its own inverse), it is not a safe solution for encrypting information. The reason is that an attacker may simply try all possible 26 keys and see whether a meaningful text is obtained in each case. In fact, for long English texts it is not even necessary to manually verify each alleged decryption; an automated procedure can in most cases recover the correct key by using the natural distribution of alphabet symbols in English text. The goal of this exercise is to implement one such automated decoder.

As a reference point to determine the natural frequence of letters in the English language, in this problem you are given a file that contains the complete works of William Shakespeare. To get you started, we have provided a `solution_template.py` file.

(a) [10 points] Implement the `get_frequency_distribution` function. Specifically, the function is given a text and will first count the number of occurrences of each alphabet letter in order to compute the frequency distribution of the symbols. Suppose $N_a, N_b, \ldots, N_z$ denote the number of occurrences of the letters a through z in the text

(only consider alphabetic characters and do *not* differentiate between upper and lower case).

The vector $N := (N_a, N_b, \ldots, N_z)$ describes the relative frequency of various alphabet letters in the input text. We normalize this vector to have Euclidean length 1; remember that the Euclidean length of the vector N, denoted by $\|N\|$, is defined as

$$\|N\| := \sqrt{N_a^2 + N_b^2 + \cdots + N_z^2}.$$

Let

$$F_a := \frac{N_a}{\|N\|}, F_b := \frac{N_b}{\|N\|}, \ldots, F_z := \frac{N_z}{\|N\|}.$$

Your algorithm should return the normalized vector N; that is, the vector

$$F := (F_a, F_b, \ldots, F_z).$$

The function `get_frequency_distribution` should return the resulting vector as a Python list of values in the form

```
[0.13, 0.0, 0.004, ... ],
```

where the first entry is the value of F_a, the second entry is the value of F_b and so forth. Values should be correct to as many decimal places as possible, we will test to 3 decimal places.

(b) [15 points] Implement the `get_shift` function. The input to the `get_shift` function is a ciphertext (given as a string) and a file (e.g., Shakespeare's works) from which to determine the natural letter frequency. The function is expected to output the best guess on the corresponding shift K used (an integer between 0 and 25).

The algorithm that we will use as the reference point for grading consists of the following steps:

1. Calculate the vector $(F_a, F_b, \ldots, F_z)$, as defined in the previous part, corresponding to the natural frequency of the alphabet letters in English text (computed according to the second argument given to the Python function).
2. For each shift $K \in \{0, \ldots, 25\}$, obtain the frequency vector $(F'_a, F'_b, \ldots, F'_z)$ that corresponds to the ciphertext shifted by K positions.
3. Pick the "best fitting" shift K that minimizes the distance between the frequency vectors $(F_a, F_b, \ldots, F_z)$ and $(F'_a, F'_b, \ldots, F'_z)$. Similar to what you saw in the lecture, the distance between the two vectors is defined as one minus their inner product (which defines the "angle" between the two vectors); that is, find K that minimizes
$$1 - (F_a F'_a + F_b F'_b + \cdots + F_z F'_z).$$

Note: We have provided you the complete works of William Shakespeare with which you can test your code. We will change the file used for natural frequencies during testing so please do not rely on the frequency distribution obtained by this file.

(c) [15 points] Now that we know Caesar cipher can be easily broken, a natural idea is to think of more complicated substitutions to make the attacker's life more difficult. In fact as early as the 15th century, an extension of Caesar cipher was published by Blaise de Vigenère. In Vigenère cipher, the key consists of multiple integers $K_1, \ldots, K_k \in \{0, \ldots, 25\}$. The first character of the plain text is then shifted by K_1 just as in the original Caesar cipher, the second character is shifted by K_2 and so on (and again the $(k+1)$st character is shifted by K_1, and as usual, the ciphers have no effect on white spaces and punctuation).

This makes Caesar cipher the special case with $k = 1$. With $k = 2$ there are already 26^2 possibilities for the key, i.e., the amount by which we shift the odd places and even places. In this part, we extend the attack on Caesar cipher to Vigenère cipher. The idea is very simple. For $k = 2$, we use the same attack as Caesar cipher on the text obtained from the odd positions to retrieve K_1 and independently extract K_2 from the even positions.

The extension to $k > 2$ is then similar: The sub-sequence of the cipher text obtained from the 1st, $(k+1)$st, $(2k+1)$st symbol of the plain text, and so on, is used to retrieve K_1. Similarly, K_2 is obtained from letters whose position is a multiple of k plus two, and so forth. Using this method, the search time grows linearly as a function of k.

Using this idea, implement the `get_multi` function, which is similar to the previous part but now also takes the value of k (the number of keys) as a parameter. You should return a Python list of each shift. For example, if $k = 3$ the result might be

```
[18, 2, 14],
```

which corresponds to $K_1 = 18, K_2 = 2, K_3 = 14$.