# DSC 190 - Homework 07

Due: Wednesday, February 23

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope at 11:59 PM.

## Problem 1.

In this problem, consider an alphabet of the four characters that make up DNA: $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. We'll use an encoding that maps `A` $\to 0$, `C` $\to 1$, `G` $\to 2$, and `T` $\to 3$.

**a)** Suppose we hash the first four letters of `"GATTACA"` by interpreting `"GATT"` as a number represented in base-4 using the above encoding. What is this number in base-10 (decimal)? Do not use modular arithmetic in computing the hash (that is, *do not* generate a random prime number $q$).

**Solution:** We have, reading right to left:

$$3 \times 4^0 + 3 \times 4^1 + 0 \times 4^2 + 2 \times 4^3 = 143$$

**b)** Now suppose we wish to hash letters `1:5` in `"GATTACA"` by interpreting `"ATTA"` as a number represented in base-4 using the above encoding. What is this number in base-10 (decimal)? Compute it directly by hand, without using the hash you found in the previous part.

**Solution:** We have, reading right to left:

$$0 \times 4^0 + 3 \times 4^1 + 3 \times 4^2 + 0 \times 4^3 = 60$$

**c)** Hash the letters `1:5` in `"GATTACA"` again, this time applying a *rolling* hash. That is, starting with the hash you found in part (a), apply corrections to arrive at the hash you found in part (b). Show your work.

**Solution:** We have, reading right to left:

$$(143 - 2 \times 4^3) \times 4 + 0 = 15 \times 4 = 60$$

## Programming Problem 1.

In a file named `multiple_rabin_karp.py`, create a function named `multiple_rabin_karp(s, patterns)` which accepts a string `s` and a *list* of $k$ equal-length strings `patterns` which contains one or more patterns to search for. It should return a list-of-lists, where list 0 contains the indices where pattern 0 can be found in `s`, list 1 contains the indices where pattern 1 can be found, and so forth. Your function should be a modified version of Rabin-Karp. You can assume all strings contain only ASCII characters. Your implementation should use the rolling hash functions from lecture slide 50 without modifying them.

*Hint*: Rabin-Karp performs a single pass over the string `s`. A simple-but-inefficient way to solve this problem is to run Rabin-Karp over `s` $k$ times, each time looking for another pattern. Don't do this! Instead, think of how to perform a single pass over the string `s` while simultaneously looking for all $k$ patterns at once. How does the fact that all of the patterns have the same length help?

Example:

```
>>> multiple_rabin_karp("this is a test. testing.", ["this", "test"])
[[0], [10, 16]]
```

**Solution:** In a standard Rabin-Karp, we hash the pattern and hash the window, and check for equality of these hashes at every iteration. In the modified version, we start by hashing all of the patterns and placing them in a `dict` mapping the hash to the pattern(s) that have that hash. On each iteration, we hash the current window and check to see if the hash is in our dictionary. If it is, we do a full string equality check of the window versus the pattern(s) with the same hash.

The solution (generously provided by Anamika Gupta) looks like this:

```
def multiple_rabin_karp(s, patterns):
    """
    Performs a search for multiple patterns in `s` using Rabin-Karp.

    Parameters
    ----------
    s : str
        The "haystack" to search through.
    patterns : List[str]
        The "needles" to search for.

    Returns
    -------
    List[List[int]]
        A list-of-lists, where list i contains the indices where patterns[i] was found
        in s.

    Example
    -------
    >>> multiple_rabin_karp("this is a test. testing.", ["this", "test"])
    [[0], [10, 16]]

    """
    # BEGIN PROMPT
    q = 149
    hashes = {}
    match_locations = {}
    if len(s) < len(patterns[0]):
        for i in patterns:
            match_locations[i] = []
    else:
        hashed_window = base_128_hash(s, 0, len(patterns[0]), q)

        for i in patterns:
            hashed_pattern = base_128_hash(i, 0, len(i), q)
            if hashed_pattern in hashes:
                hashes[hashed_pattern].append(i)
            else:
                hashes[hashed_pattern] = [i]

            match_locations[i] = []
            if s[0 : len(i)] == i:
                match_locations[i].append(0)
        for i in range(1, len(s) - len(patterns[0]) + 1):
            # update the hash
            hashed_window = update_base_128_hash(
                s, i, i + len(patterns[0]), hashed_window, q
            )

            # hashes are unique; no collisions
            if hashed_window in hashes:
                matches = hashes[hashed_window]
                start = i
                stop = i + len(patterns[0])
                if s[start:stop] in matches:
                    match_locations[s[start:stop]].append(i)

    return list(match_locations.values())
    # END PROMPT

# these are the hash functions from the lecture slides, provided here for your
# convenience.

def base_128_hash(s, start, stop, q):
    """Hash s[start:stop] by interpreting as ASCII base 128"""
    p = 0
    total = 0
    while stop > start:
        total = (total + ord(s[stop - 1]) * 128 ** p) % q
        p += 1
        stop -= 1
    return total


def update_base_128_hash(s, start, stop, old, q):
    # assumes ASCII encoding, base 128
    length = stop - start

    removed_char = ord(s[start - 1]) * 128 ** (length - 1)
    added_char = ord(s[stop - 1])

    return ((old - removed_char) * 128 + added_char) % q
# BEGIN REMOVE
# the solution code above is thanks to Anamika Gupta
# END REMOVE
```