# ORIE 5270: Big Data Technologies
# Homework 3 – Due: Friday, 04/09/2021

**Instructions.** The deadline to submit is **Friday, April 9th** at **11:59pm (midnight) US EST)**. Submit your answers on **Gradescope**. Please submit a **single file per problem**.

**Note**: You may submit a Jupyter notebook for Problem 1.

---

**Problem 1** (Solving sparse linear systems)**.** Suppose you have a set of measurements

$$y_i = a_i^\mathsf{T} x_\sharp, \quad i = 1, \dots, m,$$

where $a_i \in \mathbb{R}^n$ are **known** design vectors and $x_\sharp \in \mathbb{R}^n$ is an unknown signal you want to recover. In matrix-vector notation, this is equivalent to

$$y = Ax_\sharp, \quad A = \begin{bmatrix} a_1^\mathsf{T} \\ \vdots \\ a_m^\mathsf{T} \end{bmatrix}.$$

You are given than $m \ll n$, which means that the system is underdetermined. Without any assumptions, it is statistically impossible to recover $x_\sharp$, since there is an infinite number of solutions. One assumption that enables unique recovery is that that $x_\sharp$ is **sparse**; this means that most of its elements are zero, i.e.,

$$\left|\left\{i \mid (x_\sharp)_i \neq 0\right\}\right| = k \ll n.$$

An algorithm to recover $x_\sharp$ is that of **iterative hard thresholding**, presented below. *The operation $\mathscr{P}_k(\tilde{x}_t)$ sets all but $k$ largest elements (in magnitude) of $\tilde{x}_t$ to zero.*

**Algorithm 1** Iterative hard thresholding

**Input**: matrix $A$, measurements $y$, initial guess $x_0$, iterations $T$, step $\eta > 0$, sparsity $k$.
**for** $t = 1, \dots, T$ **do**
  $\tilde{x}_t := x_{t-1} - \eta A^\mathsf{T}(Ax_{t-1} - y)$
  $x_t := \mathscr{P}_k(\tilde{x}_t)$ ▷ Projection to set of sparse vectors
**end for**
**return** $x_T$

1. Implement Algorithm 1 in Python (using `NumPy`). The function signature should be `iht_solve(A, y, x_0, T, eta, k)` and your function should return a single vector (the output $x_T$ of Algorithm 1). If you wish, you can follow the outline given in the Assignments Page.

2. Try your algorithm on a few random instances with $m = 100, n = 500,\ T = 500$ and varying sparsity level $k \in \{2^h \mid h = 0, 1, \ldots, 5\}$. You can use the function '`genInstance`' from the Assignments Page to generate the instances; for example:

```
y, A, x_true = genInstance(100, 500, 10)      # k = 10
```

Write a Python script that generates an error plot with $k$ in the $x$-axis and the approximation error $\|x_T - x_\sharp\|_2$ in the $y$-axis (use `matplotlib` for the plots).

---

**Problem 2** (All-pairs distances)**.** Suppose you are given a set of vectors $x_1, \ldots, x_N$ in $\mathbb{R}^n$ and you want to compute

$$d_{ij} = \|x_i - x_j\|_2^2, \quad \forall i, j \in \{1, \ldots, N\}.$$

Write a Python function `all_pairs_dist(X)` that accepts a NumPy array $X \in \mathbb{R}^{N \times n}$ with each row being a vector $x_i$ and computes a $N \times N$ matrix $D$ with $D_{ij} = \|x_i - x_j\|_2^2$. **Your function cannot use loops, just Numpy operations and broadcasting.**

*Hint: use the fact that* $D_{ij} = \|x_i - x_j\|_2^2 = \|x_i\|_2^2 + \|x_j\|_2^2 - 2x_i^\mathsf{T} x_j$ *and NumPy broadcasting.*

---

**Problem 3** (Hashing and bloom filters)**.** In class, we mentioned that constructing an *ideal* hash function that maps from $\{0, \ldots, m-1\}$ to $\{0, \ldots, k-1\}$ (i.e., a hash function $h$ such that $h(i)$ is drawn uniformly at random from $\{0, \ldots k-1\}$) is impossible. However, since we usually only care about minimizing collision probabilities, we can build something called a **universal** hash function using the Algorithm 2 below.

**Algorithm 2** Universal hash function

**Input**: input universe size $m$, output universe size $k$
1. Pick a prime number $p > m$.
2. Draw an integer $a \in \{1, \ldots, p-1\}$ uniformly at random.
3. Draw an integer $b \in \{0, \ldots, p-1\}$ uniformly at random.

**return** the function $h(x) := \big((ax + b) \mod p\big) \mod k$.

**Part I:** Write a Python function `genHash(m, k)` that implements Algorithm 2. Your function should return a callable that is the function $h(x)$ described in the algorithm. For example, the output `hash` below should itself be a function that can be used to map elements from the input universe to $\{0, \ldots, k-1\}$.

```
>>> hash = genHash(100, 10)
>>> hash(97)    # should return something in {0, 1, .., 9}.
>>> hash(97)    # should return the same number
```

You may use the following Theorem to narrow your search for the prime number $p$ described in the Algorithm.

**Theorem 0.1** (Bertrand-Chebyshev)**.** *If $m > 3$ is an integer, there exists at least one prime number $p$ such that $m < p < 2m - 2$.*

**Note**: you can use code from the Internet to implement a function that check if a number is prime, as long as you acknowledge / cite your source.

**Part II:** Use the above function to implement a Bloom filter. Bloom filters are efficient data structures for checking membership in a set. The cost of efficiency is a (small) probability of false positives. Here, we assume that our input universe is $\{0, \ldots, m-1\}$. A Bloom filter works as follows:

**Bloom filter**

1. Initialize a $k$-dimensional bit array (i.e. with values in 0 or 1), with all elements initially set to 0. Call this array $A$.
2. Generate $p$ universal hash functions mapping from $\{0, \ldots, m-1\}$ to $\{0, \ldots, k-1\}$ using Algorithm 2. Call these functions $h_1, \ldots, h_p$.
3. **Insertion**: to "insert" some $x \in \{0, \ldots, m-1\}$ to the set, modify $A$ as follows:

$$A[h_i(x)] = 1, \quad \forall i = 1, \ldots, p.$$

4. **Lookup**: to check if some $y \in \{0, \ldots, m-1\}$ belongs to the set, return:
   - `TRUE` if $A[h_i(y)] = 1$ *for all* $i = 1, \ldots, p$.
   - `FALSE` otherwise.

Create a Python class called `BloomFilter` with a constructor that accepts the size of the input universe $m$, the number of elements $n$ to insert to the filter, the number of bits $k$, and the number of hash functions $p$. If the user omits $p$, you should choose

$$p = \left\lceil \frac{k}{n} \ln 2 \right\rceil.$$

Your class should support the following instance methods:

- `empty()`: clear the Bloom filter by setting the bit array $A$ to zero.
- `insert(x)`: insert an element $x$ into the filter. It should update the state of the filter and return `True` if the element was added and `False` if it was already present.
- `lookup(x)`: lookup an element $x$. It should return `True` if the element was found and `False` otherwise.

**Problem 4** (Streams)**.** In this problem, you will implement a couple of algorithms operating on data streams.

1. Implement a Python function `randomStream(m, n)` that returns a generator containing up to $n$ random numbers from the set $\{0,\ldots,m-1\}$. You will use this function later to emulate a stream; note that a generator takes up way less space than e.g., calling `np.random.randint`, which would allocate the entire $n$-element list.

2. Write a Python function `sample(stream, k)` that implements the reservoir sampling algorithm for choosing $k$ elements at random from a random stream. Your function should use $O(k)$ memory; you can assume that `stream` is a generator like the one you implemented in Part (1).

3. Suppose we now want to (approximately) find the $f$ **most frequent** elements of a stream. An algorithm for doing that is the CountMinSketch algorithm, which approximates the frequency of each distinct element seen in the stream. The algorithm relies on the concept of a universal hash function from Problem 3 and operates as follows:

   **CountMinSketch**

   Below, we assume that the input "universe" is the set $\{0,\ldots,m-1\}$.

   - **Initialize**: create a matrix $C$ with $t$ rows and $k$ columns, and generate $t$ universal hash functions $h_1,\ldots,h_t : \{0,\ldots,m-1\} \to \{0,\ldots,k-1\}$.
   - **Insert**: if $x$ is the new element of the stream, do the following:

     $$C[i,h_i(x)] \leftarrow C[i,h_i(x)]+1, \quad \text{for all rows } i.$$

   - **Lookup**: to find the approximate frequency of an element $y \in \{0,\ldots,m-1\}$, return

     $$\hat{c}_y := \min_{1\le i\le t} C[i,h_i(y)]$$

   Write a Python function `countMinSketch(stream, m, t, k)` that implements the above algorithm and uses it to find the $f$ most frequent elements of an input stream. You can assume that `stream` is a Python generator like the one you wrote in Part (1), and you may use the universal hash function implementation from Problem 3.