section 02: Hashing and Randomness

For Friday section by Shuvom Sadhuka [9/18/2020], Allison Kao [9/20/2019], adapted from Daniel Eaton [9/21/2018] and William Mallard [9/15/2017]

Revisiting Kallisto

Why did kallisto make waves when the software was published? Up until kallisto, the main bottlenecks in the processing of RNA-seq data were computational power and time. Unlike previous methods, kallisto could efficiently process reads in minutes on a personal laptop. Let's explore how kallisto achieves this feat: hashing and pseudoalignment of k-mers.

This section will cover the basics of hashing, graphs, and other ideas fundamental to kallisto. For information relevant to the pset (specifically generative models), we encourage you to go to office hours.

k-mers and De Bruijn graph

At its core, kallisto is a string matching algorithm: given an input sequence and a library of possible transcripts, can we identify which transcript(s) the sequence was from?

Imagine an analogous problem in which we want to catch plagiarism by comparing a student's essay to a library of books. We know one (or more) of the books contains the essay (perhaps as a chapter in the book, for example).

One naive method to do this would be to compare the essay to each book letter by letter, but this would waste a lot of time. Instead we could group letters into words and compare the essay to each book word by word. Moreover, if the book's first word is "lake" then we don't need to scan over *all* the words in every book but only need to start our search at every occurrence of the word "lake" in the books:

Book	Chapter	Excerpt
1	1	"The lake in Canada..."
1	5	"Johnny went to the shiny lake with..."

2	7	"The lake in Russia..."

We can basically start our search for plagarism at these words and ignore the rest. This analogy captures both the basic intuitions behind kallisto: (1) grouping letters into words to speed up the search is analogous to constructing **k-mers** and (2) searching for occurrences of the word "lake" to narrow our search is analogous to **hashing**. In the table above we see multiple possible books from which "lake" was plagiarized. kallisto calls this set of possible transcripts an **equivalence class**.

To give a little more grounding to these ideas, a **k-mer** is a set of nucleotides of length k (e.g. $k = 3$: AAC). We **hash** a **k-mer** by giving it a randomly generated identifier — for example, the 3-mer AAC might get the identifier 107. Each transcript's k-mers are hashed beforehand (think of this as "prework" we do to save time later) and every instance of "lake" is given the identifier 107.

The de Bruijn graph is a way of organizing these **k-mers**. In the case of words in a book, it would be easier, for example, if we not only stored the identifier associated with each word, but also all possible words that could be next and the associated books/chapters they come from:

Next Word	Next Word ID (randomly generated)	Next Word Source
in	109	Book 1 Ch. 1
with	5	Book 1 Ch. 5
in	100	Book 2 Ch. 7

We store this information as a **graph**, where the "next word" (or next k-mer) is represented as an edge in a graph, and the source (or transcript) is represented as a **color**. If word or group of words (equivalently, k-mer or group of k-mers) could belong to more than source, we call the set of sources an *equivalence class*.

Hashing

Using its constructed De Bruijn graph, kallisto *hashes* k-mers to quickly find their corresponding k-compatibility classes. Kallisto stores this map in a *hash table*.

A hash function maps *keys* (data of arbitrary size) to *hashes* (indices or values of a fixed size). For example, consider a hash function that takes any possible string of characters as its input key, and generates a 32-bit integer as its output hash. This function would take the string's bytes, smash them together via some special combination of bit shifts and logical operators, and spit out a 32-bit integer. It does so in such a way that keys will be uniformly distributed across the range of outputs -- so if you hash a string, and then change it by a single letter and hash it again, the two hash values will be totally different.

```
def hash_function(string):
    total = 0
    for char in string:
        total = total + ord(char)
    output = total % (2**32)
    string = '{:032b}'.format(output)
    return string

print(hash_function("apple"))
print(hash_function("orange"))
print(hash_function("pineapple"))
print(hash_function("apple tree"))
print(hash_function("AGC"))
```

```
00000000000000000000001000010010
00000000000000000000001001111100
00000000000000000000001110111110
00000000000000000000001111100010
00000000000000000000000011001011
```

This is just one example of the many hash algorithms out in the world. Python also has its own method `hash()`:

```
print(hash("apple"))
print(hash("orange"))
print(hash("pineapple"))
print(hash("apple tree"))
print(hash("AGC"))
```

```
-1326216271903328685
-8008615618095799975
4247188155257407316
-4149317906861304984
7760202784351401612
```

A hash table is a data structure built on top of a normal list, with its length equal to the number of possible hashes. To add a value to the hash table, you hash the key, and add some value to the bucket at the index corresponding to the hash. To illustrate why we use hash functions and hash tables, consider the problem of accessing keyed information without the aid of a hash. You would have to access each piece of data separately and check if it is related to the key (presumably through some logical operation). In the worst case, you would comb through the entire dataset before finding your key's assocated value as the last entry.

Image taken from Wikipedia

Pseudoalignment

Kallisto doesn't use direct counts to quantify transcript abundances. Instead, kallisto determines a *k-compatibility class*, the set of compatible transcripts, for each k-mer. We then input our k-compatibility classes into a *likelihood function*, and use a fancy *expectation-maximization algorithm* to find the transcript abundance parameters that maximize likelihood. Kallisto then outputs these transcript abundances (τ_i) in units of TPM.

Likelihood function:

$$L(\nu) \propto \prod_{f \in F} \sum_{i \in T} y_{f,t} \frac{\nu_i}{\ell_i}$$

We won't go into the details of this likelihood function, but it's important to know how to interconvert between *nucleotide abundance* (ν_i, as in the likelihood function) and *transcript abundance* (τ_i, what Kallisto outputs).

The conversion is as follows:

$$\tau_i = \frac{\nu_i}{\ell_i} \left(\sum_j \frac{\nu_j}{\ell_j} \right)^{-1}$$

To make sure we understand what's happening, let's just test it out on Arc 1 from the hw2 data.

transcript	abundance (ν_i)	TPM ($10^6\tau_i$)	length (L_i)	segments_covered
Arc1	0.008	6000	4000	ABCD
Arc2	0.039	58000	2000	BC
Arc3	0.291	290000	3000	CDE
Arc4	0.112	83000	4000	DEFG
Arc5	0.127	94000	4000	EFGH
Arc6	0.008	7800	3000	FGH
Arc7	0.059	87000	2000	GH
Arc8	0.060	88000	2000	HI
Arc9	0.022	22000	3000	IJA
Arc10	0.273	270000	3000	JAB

If we did the math right, for Arc 1, everything on the right side of the conversion equation should be equal to:

$$\tau_i = \frac{TPM}{10^6} = \frac{6000}{10^6} = 0.006$$

Let's calculate the right side:

$$\frac{0.008}{4000}\left(\frac{0.008}{4000} + \frac{0.039}{2000} + \frac{0.291}{3000} + \frac{0.112}{4000} + \frac{0.127}{4000} + \frac{0.008}{3000} + \frac{0.059}{2000} + \frac{0.022}{3000} + \frac{0.273}{3000}\right)^{-1}$$

```
factor1 = 0.008/4000
factor2 = (0.008/4000 + 0.039/2000 + 0.291/3000 + 0.112/4000 +
print(round(factor1 * factor2,3)) # rounds to 3 digits
```

```
0.006
```

As expected, the math works out!

Generating Randomness

In hw2, we ask you to create synthetic data in a random fashion to check if kallisto works. This allows us to validate kallisto's results to a known answer.

Random vs. Pseudorandom

Randomness refers to the absence of any pattern. Truly random numbers only arise from physical processes (eg, radioactive decay, thermal noise, etc). Sequences of random numbers exhibit certain statistical properties that are useful for various computational applications.

Computers are deterministic, so they cannot generate truly random numbers on their own. However, there are ways to make them generate sequences of numbers with many of the statistial properties of a truly random sequence. We call these random-looking (though ultimately deterministic) sequences *pseudorandom*.

Pseudorandom Number Generators

Pseudorandom number generators (PRNGs) produce sequences of pseudorandom numbers. At their core is a recursive function combining bit shifts and bitwise logical operations. You feed the previous number into the generator to get the next number in the sequence.

RANDU (simple, but bad)

RANDU is a pseudorandom number generator developed in the 1960s that produces awful, correlated random numbers. For this reason, it is no longer in widespread use. But, it is remarkably simple and illustrates some of the principles of pseudorandom number generators.

No need to understand how RANDU works, but note that when we define RANDU as a function in python, RANDU generates numbers that look pretty random:

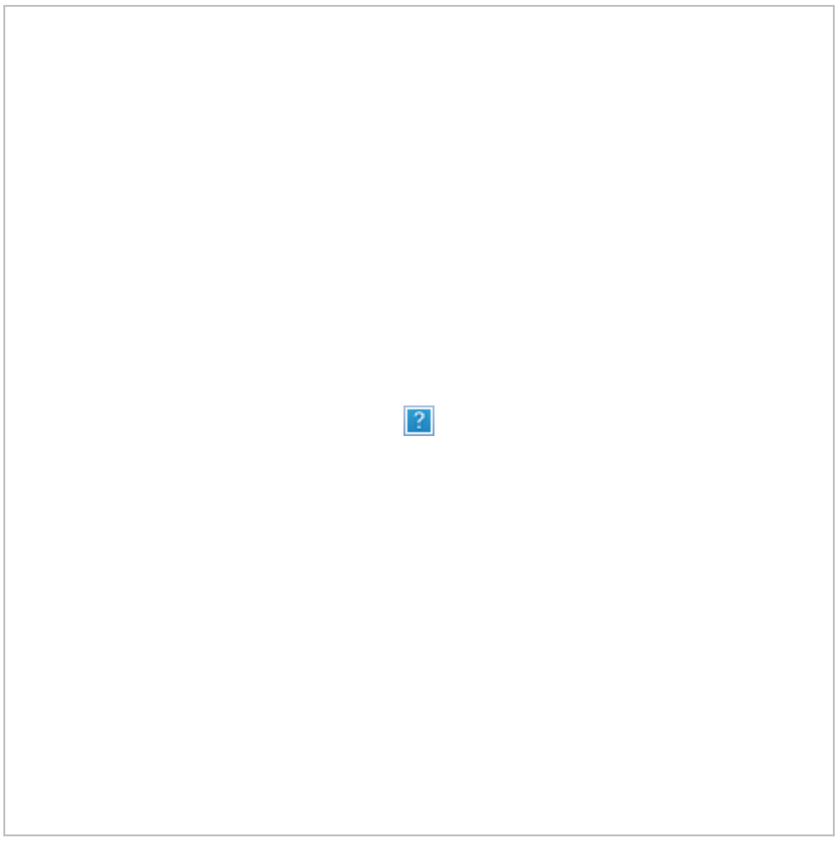
```
def randu(V_j):
    big_number = 65539 * V_j
    V_j_1 = big_number%(2**31)
    return V_j_1

V_0 = 189243
V_1 = randu(V_0)
V_2 = randu(randu(V_0))
V_3 = randu(randu(randu(V_0)))

print(V_0,V_1,V_2,V_3)
```

```
189243 1665378737 1400634643 2005333817
```

Unfortunately RANDU does not produce random numbers, they are *pseudo*-random, with an emphasis on the pseudo. RANDU actually generates highly correlated numbers. The easiest way to see this is to make a 3D plot whose points' coordinates correspond to three consecutive numbers generated by RANDU. A truely random number generator should produce a uniform "cloud" in this space. As you can see, these points are very regularly distributed into 15 two-dimensional planes.



Plot taken from Wikipedia

Luckily python uses a much less pathological random number generator than RANDU that is (usually) sufficient for generic random sampling tasks.

Seeds

Where does the PRNG get its very first number? In the case of RANDU, we clearly specified a V_0. By default, Python seeds its PRNG with whatever time it is when you ask for your first random number. But there's nothing stopping you from overriding that and giving it your favorite number!

A *seed* is a number you give a PRNG to initialize its internal state. So in a sense, the seed serves as a unique identifier for a sequence of pseudorandom numbers.

What's nice about this, is you can initialize Python's PRNG to some state at the beginning of your program, and then every subsequent run will use the same sequence of pseudorandom numbers.

Why would you want that?

1) Debugging. This is useful for comparing your results as you tweak your code. If your edits didn't alter the number or order of calls to the PRNG, then the random data you're working with should be consistent across runs.

2) Reproducibility. When you give your code to someone else, or publish it in a journal, other people can re-run your code and verify that they get the exact same output. Biology is currently plagued by irreproducible results. Biological systems are intrinsically noisy, so there's probably a limit to how reproducible we can make results from the bench. But computational analyses have no excuse for being irreproducible, as long as you provide your analysis code, and seed your random number generators!

NumPy includes a `random` module which includes a number of handy functions for generating and working with random numbers.

To access `numpy`'s functions, import numpy:

```
import numpy as np
```

To seed Python's pseudorandom number generator in the `np.random` module:

```
np.random.seed(42)
```

You can then proceed to use functions from `np.random` as usual.

```
# Seed the PRNG with 1, and randomly generate 20 integers from
np.random.seed(1)
np.random.randint(10, size=20)
```

```
array([5, 8, 9, 5, 0, 0, 1, 7, 6, 9, 2, 4, 5, 2, 4, 2, 4, 7, 7
```

```
# Seed the PRNG with 2, and randomly generate 20 integers from
np.random.seed(2)
np.random.randint(10, size=20)
# These 20 numbers differ from the first 20.
```

```
array([8, 8, 6, 2, 8, 7, 2, 1, 5, 4, 4, 5, 7, 3, 6, 4, 3, 7, 6
```

```
# Seed the PRNG with 1 again, and randomly generate 20 integer
np.random.seed(1)
np.random.randint(10, size=20)
# These 20 numbers match the first 20 exactly!
```

```
array([5, 8, 9, 5, 0, 0, 1, 7, 6, 9, 2, 4, 5, 2, 4, 2, 4, 7, 7
```

```
# Seed the PRNG with 1 again, and randomly generate 20 integer
np.random.seed(1)
print(np.random.randint(10, size=5))
print(np.random.randint(10, size=5))
print(np.random.randint(10, size=5))
print(np.random.randint(10, size=5))
# These 20 numbers still match the first 20, even though we pu
```

```
[5 8 9 5 0]
[0 1 7 6 9]
[2 4 5 2 4]
[2 4 7 7 9]
```

Other Random Tips

To generate a random number from the half-open interval [0,1):

```
x = np.random.random()
print(x)
```

```
0.7783892363365335
```

To select an item from a list according to a list of weights:

```
L = ['abc', 'def', 'ghi']
w = [.2, .5, .3]

x = np.random.choice(L, p=w)
print(x)
```

```
def
```

Note that your list of weights is supposed to be a probability distribution, so it must sum to 1. If it doesn't, `choice()` will complain.

To select a number from a normal distribution with mean `mu` and standard deviation `sigma`:

```
mu = 0.
sigma = 1.
x = np.random.normal(mu, sigma)
print(x)
```

```
0.37665663535750415
```

You can read up on the various random functions on the SciPy website.

Manipulating Strings

Translation tables

If you want to transform a string according to some specific set of character substitutions, you can efficiently do so with a *translation table*.

```
T = str.maketrans('abc', 'xyz')

print('abacab'.translate(T))
print('ccccba'.translate(T))
print('bcbcba'.translate(T))
```

```
xyxzxy
zzzzyx
yzyzyx
```

Note that we only need to build the translation table once. As long as we store it somewhere, we can reuse the same translation table over and over again.

String reversal

To reverse a string, we can use a common list and string slicing idiom.

```
S = 'abcdef'
print(S[::-1])
```

```
fedcba
```

Command-Line Tricks

Writing gzip Files

Text-based bioinformatics data is usually stored and shared in compressed form. Common compression tools are gzip (.gz files), WinZip (.zip files), and bzip2 (.bz2 files). If you have an uncompressed file called `foo.txt`, simply run `gzip foo.txt` to generate a compressed version called `foo.txt.gz`. If `foo.txt` was larger than a few kilobytes, this compressed version will take up a fraction of the space.

To generate a gzip'd text file directly from Python, you can use the `gzip` library. The `gzip` library provides an `open()` function that works just like the normal `open()` function, except it compresses the data before writing it to disk. There is one small difference in its usage: Instead of opening the file in *write* mode with 'w', we need to specify that we want to open the gzip file in *text-writing* mode with 'wt'.

```
# Note: This code uses unassigned variables, so it'll run with
import gzip

with gzip.open('foo.txt.gz', 'wt') as fd:
    for line in lines:
        print(line, file=fd)
```

Basic Numpy

Creating Arrays

Making a 1D array

```
list_a = [0,1,2]
array_a = np.array(list_a)

print(array_a)
```

```
[0 1 2]
```

Making a 2D array

```
list_b = [[0,1,2],[3,4,5],[6,7,8]]
array_b = np.array(list_b)

print(array_b)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

Checking the size of each dimension

```
print(array_a.shape)
print(array_b.shape)
```

```
(3,)
(3, 3)
```

Making a zero array of arbitrary shape

```
zero_array = np.zeros((4,3))
print(zero_array)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Simple Operations

```
A = np.array([[1,0],[1,1]])
B = np.array([[0,1],[1,1]])
print(A)
print(B)
```

```
[[1 0]
 [1 1]]
[[0 1]
 [1 1]]
```

Element-wise addition

```
C = A + B
print(C)
```

```
[[1 1]
 [2 2]]
```

Element-wise multiplication

```
C = A * B
print(C)
```

```
[[0 0]
 [1 1]]
```

Multiplicative scaling

```
C = A * 0.5
print(C)
```

```
[[0.5 0. ]
 [0.5 0.5]]
```

Adding to each element

```
C = A + 0.5
print(C)
```

```
[[1.5 0.5]
 [1.5 1.5]]
```

Matrix multiplication

```
C = A @ B
print(C)
```

```
[[0 1]
 [1 2]]
```

Adding together all elements of an array

```
C = np.sum(A)
print(C)
```

```
3
```

Summing along the rows of an array

```
C = np.sum(A, axis=1)
print(C)
```

```
[1 2]
```

Summing along the columns of an array

```
C = np.sum(A, axis=0)
print(C)
```

```
[2 1]
```