STAT 243 Problem Set 4

Treves Li

2024-10-10

Collaboration Statement

I did not collaborate with anyone.

Question 1

Memoization of a function involves storing (caching) the values produced by a given input and then returning the cached result instead of rerunning the function when the same input is provided in the future. It can be a good approach to improve efficiency when a calculation is expensive (and presuming that sometimes the function will be run with inputs on whic it has already been used). It's particularly useful for for recursive calculuations that involve solving a subproblem in order to solve a given (larger) problem. If you've already solved the subproblem you don't need to solve it again. For example if one is working with graphs or networks, one often ends up writing recursive algorithms. (Suppose, e.g., you have a geneology giving relationships of parents and children. If you wanted to find all the descendants of a person, and you had already found all the descendants of one of the person's children, you could make use of that without finding the descendants of the child again.)

Write a decorator that implements memoization. It should handle functions with either one or two arguments (write one decorator, not two!). You can assume these arguments are simple objects like numbers or strings. As part of your solution, explain whether you need to use `nonlocal` or not in this case.

Test your code on basic cases, such as applying the log-gamma function to a number and multiplying together two numbers. These may not be realistic cases, because the lookup process could well take more time than the actual calculation. To assess that, time the memoization approach compared to directly doing the calculation. Be careful that you are getting an accurate timing of such quick calculations (see Unit 5 notes)!

I wrote a decorator function called `memoization_fun` that takes as input some other function `func`, and does the caching in a dict called `cache`.

The nested `wrapper` function takes one or two arguments–if more than two arguments are provided, then only the first two are stored as keys in the cache. The `wrapper` function then checks if the arguments are already in the cache's keys. If so, then it simply returns the cached value(s) for efficiency's sake. If not, it stores the value(s) returned by the original function in the dictionary, before also returning the cached value(s).

I used `timeit` to get accurate timings of quick calculations.

There is no need to specify `nonlocal`, since there's no modification being done on variables outside the local scope. Instead, the dictionary is just storing values.

```
def memoization_func(func):
  # Initialise dictionary to store previously computed value(s)
  cache = {}

  def wrapper(*args):
    # Check how many arguments are given
    # We only want the first two arguments to be stored as keys in the cache
    key = args if len(args) <= 2 else args[:2]

    # If the key is already in the cache, simply return it
    if key in cache:
        return cache[key]

    # Otherwise, store the result of the function in the cache
    cache[key] = func(*args)
    return cache[key]

  # Add a method to clear the cache
  wrapper.clear_cache = cache.clear

  return wrapper
```

To test my decorator, I run it on the log-gamma function and a function to multiply two numbers together. Note that I clear the cache when timing the case WITHOUT memoization, otherwise subsequent executions during the timing process would already be using the cached result from the first execution, thus making the comparison pointless.

```
import timeit
import math


# Define functions
@memoization_func
def log_gamma(x):
  return math.lgamma(x)

@memoization_func
def multiply(a, b):
  return a * b


# Run log-gamma test cases
log_gamma_input = 42

print("Running log-gamma WITHOUT memoization ...")
log_gamma_time_no_cache = timeit.timeit(
  lambda: (log_gamma.clear_cache(), log_gamma(log_gamma_input)), number=500
)
print(f"{log_gamma_time_no_cache:.20f} seconds.\n")

# Populate the cache
log_gamma(log_gamma_input)

print("Running log-gamma WITH memoization ...")
log_gamma_time_yes_cache = timeit.timeit(
  lambda: log_gamma(log_gamma_input), number=500
)
print(f"{log_gamma_time_yes_cache:.20f} seconds.\n")


# Run multiplication test cases
a = 23
b = 18

print("Running multiplication WITHOUT memoization ...")
multiply_no_cache = timeit.timeit(
  lambda: (multiply.clear_cache(), multiply(a, b)), number=500
)
print(f"{multiply_no_cache:.20f} seconds.\n")
```

```python
# Populate the cache
multiply(a, b)

print("Running multiplication WITH memoization ...")
multiply_yes_cache = timeit.timeit(
  lambda: multiply(a, b), number=500
)
print(f"{multiply_yes_cache:.20f} seconds.\n")


# Compare results
print(
  f"Log-gamma speedup: \t\t"
  f"{round(log_gamma_time_no_cache/log_gamma_time_yes_cache, 2)}"
)

print(
  f"Multiplication speedup: "
  f"{round(multiply_no_cache/multiply_yes_cache, 2)}"
)
```

```
Running log-gamma WITHOUT memoization ...
0.00027583100018091500 seconds.

Running log-gamma WITH memoization ...
0.00006887000017741229 seconds.

Running multiplication WITHOUT memoization ...
0.00014020200069353450 seconds.

Running multiplication WITH memoization ...
0.00007402800110867247 seconds.

Log-gamma speedup:      4.01
Multiplication speedup: 1.89
```

The results show that memoization/caching speeds up the original functions by at least 2 times. (The speedup also depends on how fast my Jupyter session runs the code, versus how fast Quarto runs it.)

Question 2

Let's work more on how Python stores and copies objects, in particular lists.

Question 2a

Experiment with creating lists of real-valued numbers of different lengths. Try to determine how much memory is used for each reference (each element of the list), not counting the memory used for the actual values. Also consider lists of more complicated objects to see if the behavior is the same or similar.

First I generate six lists of different lengths with random, real-valued numbers. I then use `sys.getsizeof()` to get each list's size, which is comprised of references to the values stored in the list but not the actual values themselves. The amount of memory used on average for each reference can be obtained by dividing the list size by the number of list elements.

```
import random
import sys

# Define some arbitrary list lengths
lengths = [1, 10, 100, 1000, 10000, 100000]

# Create lists with arbitrary lengths
for length in lengths:
  real_list = [random.uniform(0, 100) for _ in range     (length)]

  # Get size of the list object
  list_size = sys.getsizeof(real_list)

  # Get memory for references
  element_size = list_size / length

  print(
    f"\nFor a list of length {length}:\n"
    f"{element_size:.2f} bytes is used for each element."
  )
```

```
For a list of length 1:
88.00 bytes is used for each element.

For a list of length 10:
18.40 bytes is used for each element.

For a list of length 100:
9.20 bytes is used for each element.

For a list of length 1000:
8.86 bytes is used for each element.

For a list of length 10000:
8.52 bytes is used for each element.

For a list of length 100000:
8.01 bytes is used for each element.
```

The results show that the allocated memory per reference in the list decreases as the list length increases. This is mainly due to the fixed overhead associated with the list object being spread across more elements. When a list is created, it incurs a constant memory overhead for metadata, which is more significant relative to smaller lists. As the list grows, this overhead becomes less significant per element.

We can try the code for lists of more complicated objects to see if the behaviour is similar. In this example, I use a list of tuples with three elements (a float, a string, and an int):

```
for length in lengths:
  # Create a list of dictionaries to represent complex objects
  complex_list = [
    (random.uniform(0, 1), f"string_{i}", i)
    for i in range(length)
  ]

  # Get size of the list object
  list_size = sys.getsizeof(complex_list)

  # Get memory for references
  element_size = list_size / length

  print(
    f"\nFor a list of length {length}:\n"
```

```
    f"{element_size:.2f} bytes is used for each tuple element."
  )
```

```
For a list of length 1:
88.00 bytes is used for each tuple element.

For a list of length 10:
18.40 bytes is used for each tuple element.

For a list of length 100:
9.20 bytes is used for each tuple element.

For a list of length 1000:
8.86 bytes is used for each tuple element.

For a list of length 10000:
8.52 bytes is used for each tuple element.

For a list of length 100000:
8.01 bytes is used for each tuple element.
```

From the above, it appears that the memory used for referencing list elements is the same regardless of whether the elements are "simple" or more "complex" objects. (This is despite the more complex "tuple" objects intuitively using up more memory for the actual element values, compared to the "simple" float objects.)

Question 2b

> Use the list `.copy` method with a list of numbers and with a list containing more complicated objects (including another list or some dictionary). Compare the behavior in terms of what happens when you modify elements at different levels of nestedness. Relate the behavior to the help information in `help(list.copy)` and to what we know about how the structure of lists.

First I create a `simple_list` with numbers, and another `complex_list` whose first element is an arbitrary dictionary and whose second list is an inner list comprised of `simple_list`. I then make copies of both lists.

We expect that for the `simple_list`, creating a copy creates a separate object in memory. So modifying an element in the `simple_list_copy` should not affect the original `simple_list`.

For the `complex_list`, creating a copy will likewise make a new outer list object, but the elements inside the copied list would still be pointing to the original objects.

The above understanding of list structures is in line with what is shown by `help(list.copy)`, which is that `.copy` will "return a *shallow copy* of the list" (emphasis mine). To quote directly from the Python documentation, "a shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original."

We can test the above ideas by running the `id()` function after modifying arbitrary elements in each of the simple and complex lists, and comparing whether the underlying object reference has changed or not.

```
# Create a simple list
simple_list = [random.uniform(0, 100) for _ in range(100)]

# Create a complex list
complex_dict = [
  {'key1': random.uniform(0, 100), 'key2': f'object_{i}'}
  for i in range(100)
]
complex_list = [complex_dict, simple_list]

# Create copies of both the simple and complex list
simple_list_copy = simple_list.copy()
complex_list_copy = complex_list.copy()

# Modify an arbitrary element in the simple list
```

```
simple_list_copy[42] = 'change me'
print("\nFor the \"simple\" list:")
print(
  "ID of original element: \t\t\t\t\t"
  f"{id(simple_list[42])}"
)
print(
  "ID of copied element after modification: \t"
  f"{id(simple_list_copy[42])}"
)
print(
  "Are the two IDs the same? \t\t\t\t   ",
  id(simple_list[42]) == id(simple_list_copy[42])
)

# Modify an arbitrary element in the complex list
complex_list_copy[0][16]['key1'] = 'change me'
print("\nFor the \"complex\" list (first example using dict object):")
print(
  "ID of original element: \t\t\t\t\t"
  f"{id(complex_list[0][16]['key1'])}"
)
print(
  "ID of copied element after modification: \t"
  f"{id(complex_list_copy[0][16]['key1'])}"
)
print(
  "Are the two IDs the same? \t\t\t\t   ",
  id(complex_list[0][16]['key1']) == id(complex_list_copy[0][16]['key1'])
)

# Modify another arbitrary element in the complex list
complex_list_copy[1][80] = 'change me'
print("\nFor the \"complex\" list (second example using inner list):")
print(
  "ID of original element: \t\t\t\t\t"
  f"{id(complex_list[1][80])}"
)
print(
  "ID of copied element after modification: \t"
  f"{id(complex_list_copy[1][80])}"
)
```

```
print(
  "Are the two IDs the same? \t\t\t\t   ",
  id(complex_list[1][80]) == id(complex_list_copy[1][80])
)
```

```
For the "simple" list:
ID of original element:                     140427580378224
ID of copied element after modification:    140427582391088
Are the two IDs the same?                   False

For the "complex" list (first example using dict object):
ID of original element:                     140428009885040
ID of copied element after modification:    140428009885040
Are the two IDs the same?                   True

For the "complex" list (second example using inner list):
ID of original element:                     140428009885296
ID of copied element after modification:    140428009885296
Are the two IDs the same?                   True
```

Question 2c

Consider creating these lists.

```
my_int = 1
my_real = 1.0
my_string = 'hat'

xi = [1, my_int, my_int, 2]
yi = [1, my_int, my_int, 2]
xr = [1.0, my_real, my_real, 2.0]
yr = [1.0, my_real, my_real, 2.0]

xc = ['hat', my_string, my_string, 'dog']
yc = ['hat', my_string, my_string, 'dog']
```

What is different about how the elements in these lists are stored?

We can run an ID-check to see where/how each variable and each item in each list is stored in memory.

```
print(f"my_int has id: \t\t {id(my_int)}")
print(f"my_real has id: \t {id(my_real)}")
print(f"my_string has id: \t {id(my_string)}")

lists = {
  'xi': xi,
  'yi': yi,
  'xr': xr,
  'yr': yr,
  'xc': xc,
  'yc': yc,
}

for list_name, list_item in lists.items():
  print(f"\nList '{list_name}':")
  for j, element in enumerate(list_item):
    print(f"id of Element {j}: {id(element)}")
```

```
my_int has id:       11642984
my_real has id:      140425906753968
my_string has id:    140428082727504

List 'xi':
id of Element 0: 11642984
id of Element 1: 11642984
id of Element 2: 11642984
id of Element 3: 11643016

List 'yi':
id of Element 0: 11642984
id of Element 1: 11642984
id of Element 2: 11642984
id of Element 3: 11643016

List 'xr':
id of Element 0: 140425906753648
id of Element 1: 140425906753968
id of Element 2: 140425906753968
id of Element 3: 140425906753776

List 'yr':
id of Element 0: 140425906754224
id of Element 1: 140425906753968
id of Element 2: 140425906753968
id of Element 3: 140425906754352

List 'xc':
id of Element 0: 140428082727504
id of Element 1: 140428082727504
id of Element 2: 140428082727504
id of Element 3: 140428039863776

List 'yc':
id of Element 0: 140428082727504
id of Element 1: 140428082727504
id of Element 2: 140428082727504
id of Element 3: 140428039863776
```

The output shows that Python re-uses references to integers and strings in lists, instead of creating new objects that have the same value. We see this reflected in `xi`, `yi`, `xc`, and `yc`,

where elements with the same value share the same memory address. This resource explains that Python uses **object interning**, which is a technique used "to optimize memory usage … by reusing immutable objects instead of creating new instances. It is particularly useful for strings, integers and user-defined objects". Specifically, this is applicable for integers in the range of -5 to 256. For strings, this is helpful for speeding up operations by allowing comparisons by memory address, instead of comparisons on a character-by-character basis.

Interestingly, in `xr` and `yr`, the same observation of "object interning" doesn't appear to hold for floats. Instead, a new object in memory is created every time a float with the same value is initialised in a different list. This reflects how Python handles mutable/larger objects differently (floats) from smaller, immutable types (integers and strings) in order to optimse memory usage.

Question 3

Suppose I want to compute the trace of a matrix, A, where $A = XY$. The trace is $\sum_{i=1}^{n} A_{ii}$. Assume both X and Y are $n \times n$. A naive implementation is `np.sum(np.diag(X@Y))`.

Question 3a

What is the computational complexity of that naive implementation: $O(n)$, $O(n^2)$, or $O(n^3)$? You can just count up the number of multiplications and ignore the additions. Why is that naive implementation inefficient?

The computational complexity is $O(n^3)$: 1. n matrix multiplication to get one element of A 2. Mutiplying over n rows 3. Multipling over n columns

This is inefficient because it has to do the entire matrix multiplication across X and Y just to get the diagonals for the trace. We can achieve the same result with n^2 operations–I show this in the subsequent question.

Question 3b

> Write Python code that (much) more efficiently computes the trace using vectorized/matrix operations on the matrices. You will not be able to use map or list comprehension to achieve this speedup. What is the computational complexity of your solution?

If we take advantage of vectorisation and use the dot product instead, we can get the same result more efficiently by doing `np.sum(X * Y.T)`. The computational complexity is now $O(n^2)$, since element-wise multiplication results in only $n \times n$ operations.

We can test that this gets the same result as the naive implementation, but around twice the speed (depending on Jupyter vs Quarto rendering):

```
import numpy as np
import timeit

# Generate two random n x n matrices
n = 8
X = np.random.rand(n, n)
Y = np.random.rand(n, n)

# Function to compute the naive solution
def trace_naive():
  return np.sum(np.diag(X @ Y))

# Function to compute the vectorized solution
def trace_vectorised():
  return np.sum(X * Y.T)

# Time both methods and compute solutions
naive_time = timeit.timeit(trace_naive, number=100)
naive_soln = trace_naive()

vector_time = timeit.timeit(trace_vectorised, number=100)
vector_soln = trace_vectorised()

print(
  f"Trace using naive implementation: \t\t"
  f"{round(naive_soln, 4)} ({round(naive_time, 8)} s)"
```

```
)
print(
  f"Trace using vectorised implementation: \t"
  f"{round(vector_soln, 4)} ({round(vector_time, 8)} s)"
)
```

```
Trace using naive implementation:       18.6967 (0.00049741 s)
Trace using vectorised implementation:  18.6967 (0.00039777 s)
```

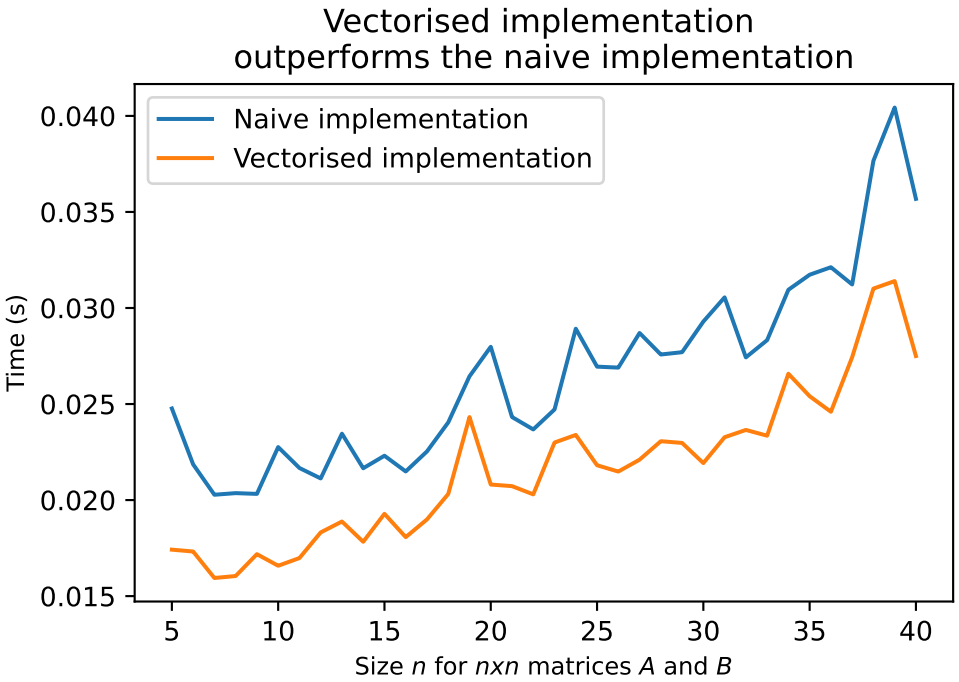
Question 3c

> Create a plot, as a function of n, to demonstrate the scaling of the original implementation compared to your improved implementation.

I interpret this question to compare the performance of the naive versus vectorised implementations as the size of the $n \times n$ matrices grow.

```
import matplotlib.pyplot as plt

# First generate linear values of n
sizes = np.arange(5, 41)

# Initialise empty lists to store results
naive_times = []
vector_times = []

# Make computations
for n in sizes:
  # Create matrixes
  X = np.random.rand(n, n)
  Y = np.random.rand(n, n)

  # Calculate times
  naive_time = timeit.timeit(trace_naive, number=5000)
  vector_time = timeit.timeit(trace_vectorised, number=5000)

  # Append times to lists
  naive_times.append(naive_time)
  vector_times.append(vector_time)

plt.plot(sizes, naive_times, label="Naive implementation")
plt.plot(sizes, vector_times, label="Vectorised implementation")
plt.title(
  "Vectorised implementation \n"
  "outperforms the naive implementation",
  size=12
)
plt.xlabel("Size $n$ for $n x n$ matrices $A$ and $B$", size=9)
plt.ylabel("Time (s)", size=9)
```

```
plt.legend()
plt.show()
```



The sudden spikes in the graph could be attributed to the computation at a given size n exceeding the cache's capacity, and so the timing takes a hit as Python has to fetch data from main memory.

Question 3d

> (Extra credit) Implement your more efficient version in using Jax (see Section 9 of Unit 5) and compare the timing to the numpy implementation, both with and without using `jax.jit()`.

For this question, I tried to follow the example in Unit 5 of the course notes.

Note that since the first call of `jax.jit()` will take a longer time for the compilation, I've called it once before the timing to make a fair comparison.

```
import jax
import jax.numpy as jnp

# Jax function to compute the vectorized solution
def trace_jnp(X_jax, Y_jax):
    return jnp.sum(X_jax * Y_jax.T)

# Evaluate performance for two sizes of matrixes, one small and another larger
n_list = [50, 1000]

for size in n_list:

  # Generate 2 random matrices
  X = np.random.rand(size, size)
  Y = np.random.rand(size, size)

  # Convert numpy matrices to JAX
  X_jax = jnp.array(X)
  Y_jax = jnp.array(Y)

  # Create a jitted version trace_jnp
  trace_jnp_jit = jax.jit(lambda: trace_jnp(X_jax, Y_jax))

  # Call jitted version once to compile
  trace_jnp_jit()

  # Time the implementations
  naive_time = timeit.timeit(trace_naive, number=500)
  vector_time = timeit.timeit(trace_vectorised, number=500)
```

```python
  jax_time = timeit.timeit(lambda: trace_jnp(X_jax, Y_jax), number=500)
  jax_jit_time = timeit.timeit(trace_jnp_jit, number=500)

  # Time the speedups
  vector_speedup = naive_time / vector_time
  jax_speedup = naive_time / jax_time
  jax_jit_speedup = naive_time / jax_jit_time

  # Print times
  print(f"\nFor n = {size}:")
  print(
    f"Naive implementation time: \t\t "
    f"{naive_time:.8f}"
  )
  print(
    f"Vectorised implementation time:  "
    f"{vector_time:.8f} ({vector_speedup:.2f} speedup)"
  )
  print(
    f"JAX implementation time: \t\t "
    f"{jax_time:.8f} ({jax_speedup:.2f} speedup)"
  )
  print(
    f"Jitted JAX implementation time:  "
    f"{jax_jit_time:.8f} ({jax_jit_speedup:.2f} speedup)"
  )
```

```
For n = 50:
Naive implementation time:       0.00485586
Vectorised implementation time:  0.00338481 (1.43 speedup)
JAX implementation time:         0.04894679 (0.10 speedup)
Jitted JAX implementation time:  0.00152496 (3.18 speedup)

For n = 1000:
Naive implementation time:       9.82675243
Vectorised implementation time:  1.92878377 (5.09 speedup)
JAX implementation time:         0.84652627 (11.61 speedup)
Jitted JAX implementation time:  0.01836071 (535.21 speedup)
```

Note that at smaller matrix sizes, we see the jitted JAX implementation being far faster than the naive or vectorised implementations. However, the simple JAX implementation is slower

than either the naive or vectorised, likely because the initial compilation overhead of setting up a small matrix in JAX outweighs the `numpy` overhead for an equivalently small-sized matrix.

However, for a larger 1000 x 1000 matrix, we clearly see the expected performance gains from JAX, and certainly with the jitted JAX implementation (with a speedup of 3 times the order of magnitude).

Question 3

Suppose we have a matrix in which each column is a vector of probabilities that add to one, and we want to generate a sample from the categorical distribution represented by the probabilities in each column. For example, the first column might be (0.9, 0.05, 0.05) and the second column might be (0.1, 0.85, .0.5). When we generate the first sample, it is very likely to be a 1 (a 0 in Python) and the second sample is very likely to be a 2 (a 1 in Python). We could do this using a for loop over the columns of the matrix, and `np.random.choice()`, but that is a lot slower than some other ways we might do it because it is a loop executing in Python over many elements (columns).

```
import numpy as np
import time

n = 100000
p = 5

# Generate a random matrix and calculate probabilities.
np.random.seed(1)
tmp = np.exp(np.random.randn(p, n))
probs = tmp / tmp.sum(axis=0)

smp = np.zeros(n, dtype=int)

# Generate sample by column.
np.random.seed(1)
start = time.time()
for i in range(n):
  smp[i] = np.random.choice(p, size=1, p=probs[:,i])[0]
print(f"Loop by column time: {round(time.time() - start, 2)} seconds.")
```

```
Loop by column time: 1.65 seconds.
```

Question 3a

> Consider transposing the matrix and looping over rows. Why might I hypothesize that this could be faster? Is it faster? Why does it make sense that you got the timing result that you got? Does using numpy's `apply` functionality help at all?

To transpose the matrix and loop over rows, we can do the below:

```
tmp_transpose = np.transpose(tmp)
probs_transpose = tmp_transpose / tmp_transpose.sum(axis=1, keepdims=True)

# Generate sample by row.
np.random.seed(1)
start = time.time()
for i in range(n):
  smp[i] = np.random.choice(p, size=1, p=probs_transpose[i,:])[0]
print(f"Loop by row time: {round(time.time() - start, 2)} seconds.")
```

```
Loop by row time: 1.69 seconds.
```

The results above verify that looping by row is faster than by column (timing results might compute differently on Quarto). This is because matrices in numpy are stored in **row-major** order. This means that row values are adjacent to each other, which increases the likelihood of cache hits and reduces the need to fetch data from main memory.

Despite the speedup, the execution times are still in the same order of magnitude. This could be because of the overheads associated with the large number of iterations in the loop, or even the overheads associated with calling the `np.random.choice` function itself in each iteration.

To see if numpy's `apply_along_axis` function would help, we can also time it:

```
start = time.time()
smp_col = np.apply_along_axis(
    lambda prob_col: np.random.choice(p, size=1, p=prob_col)[0],
    axis=0,
    arr=probs,
)
print(
  f"numpy.apply_along_axis (column) time: "
```

```
  f"{round(time.time() - start, 2)} seconds."
)

start = time.time()
smp_row = np.apply_along_axis(
    lambda prob_row: np.random.choice(p, size=1, p=prob_row)[0],
    axis=1,
    arr=probs_transpose,
)
print(
  f"numpy.apply_along_axis (row) time: "
  f"{round(time.time() - start, 2)} seconds."
)
```

```
numpy.apply_along_axis (column) time: 2.41 seconds.
numpy.apply_along_axis (row) time: 2.32 seconds.
```

In this case, using `numpy.apply_along_axis` does not seem to save time, regardless if applied by row or column. It's likely because the number of iterations is substantial enough that the overhead associated with calling the `lambda` function each time becomes significant. In fact, it is slower compared to doing straightforward looping through rows.

Question 4b

> How can we do it **much** faster (multiple orders of magnitude), exploiting vectorization? (Hint: This might involve some looping or not, but not in the ways described above. Think about how one can use random uniform numbers to generate from a categorical distribution.)

Vectorisation is the answer. The aim is to somehow operate on the entire array of probabilities all at once. The hint guides us to use `np.random.uniform`.

I'm not familiar with this function and the process, so the steps below came with the help of ChatGPT: * Calculate the cumulative distribution along column. * Use `np.random.uniform` to generate uniform random numbers. * Compare in a vectorised fashion where the cumulative probability is greater than the uniform random numbers along the columns. The first instance of this being `True` will be the selected category (retrieved using `argmax`).

```
# Get cumulative probabilities along columns
cumulative_probs = np.cumsum(probs, axis=0)

# Create uniform random numbers between 0 and 1
random_nums = np.random.uniform(0, 1, size=n)

# Vectorise comparison to find the category for each sample
start = time.time()
smp = np.argmax(
  random_nums[np.newaxis, :] < cumulative_probs, axis=0
)
print(f"Vectorisation time: {round(time.time() - start, 6   )} seconds.")
```

```
Vectorisation time: 0.003646 seconds.
```

And indeed, we see a speedup of nearly 3 orders of magnitude.