Memory and Memoization

Matthew Seguin

Preliminary Importing of Packages

```
import sys
import math
import time
import timeit
import inspect
import numpy as np
import pandas as pd
from plotnine import *
```

1.

```
def memoizer(fcn):
  '''
  Function to save evaluations to memory
  so they can be referenced rather than
  re-evaluating
  '''
  # Create object to save ouputs in memory
  memory = {}

  # Inner function to "evaluate" the provided
  # function and add it to memory
  def memory_checker(*args, **kwargs):
    # Get a string containing all the functional arguments
    arguments = str(args) + str(kwargs)

    # Check to see if the string is in memory's keys
    # (and add it if needed)
    if arguments not in memory.keys():
      memory[arguments] = fcn(*args, **kwargs)

    # Return the value held at the indicated argument location
    # (which must be the function value for the given arguments)
    return(memory[arguments])


  # Return the function that checks if the value is in
  # memory and gets the function value
  return(memory_checker)
```

First we need to define our decorator function. We know it takes in another function, now we want to keep a memory variable so that we are able to check if we have already ran this function on a given input, I use a dictionary for this so that I can reference both the key (for the inputs used) and the item (for the function evaluation output). Now we define a function that checks if the function has already been evaluated at the given inputs. First we make a string of the inputs provided and then we check if that string is already a key in our memory dictionary. If it is then we have already evaluated the function at that point and we can just return the value in our memory dictionary there. Otherwise we add that value to our dictionary with the key being the arguments provided. Finally we return the inner function that will give us the function value, whether it was evaluated or already saved in memory. We do not need to use nonlocal here since what happens is memory is created in the memoizer scope and so is memory_checker so memoizer is the enclosing scope of memory_checker and hence memory is locally available inside memory_checker. This is because Python uses lexical scoping so a function's scope is defined based on the block of code where the function is defined, and so the function has access to all variables in it's enclosing scope.

Here is some testing:

```
mem_source = inspect.getsource(memoizer)

# First test function
@memoizer
def sum_memoized(lst):
  return(sum(lst))
```

```
# Get the setup code to add to timeit
sum_mem_source = mem_source + "\n@memoizer\ndef sum_memoized(lst):\n    return(sum(lst))"

# Testing simple inputs
timeit.timeit(
  "sum_memoized([1,2,3])",
  setup = sum_mem_source,
  number = 100
  )
```

4.250000347383332e-05

```
timeit.timeit(
  "sum([1,2,3])",
  number = 100
  )
```

7.499998901039362e-06

```
# Testing more complicated inputs

# Get the setup code to add to timeit
base_source = "import numpy as np\nx = np.random.normal(loc=50,scale=2,size=10**6)\n"
new_source = base_source + "\n" + sum_mem_source

timeit.timeit(
  "sum_memoized(x)",
  setup = new_source,
  number = 100
  )
```

0.040596500009996817

```
timeit.timeit(
  "sum(x)",
  setup = base_source,
  number = 100
  )
```

3.652564899995923

```
# Second test function
@memoizer
def lgamma_memoized(x):
  return(math.lgamma(x))

# Get the setup code to add to timeit
lgam_mem_source = "import math\n" + mem_source + "\n@memoizer\n"
lgam_mem_source += "def lgamma_memoized(x):\n    return(math.lgamma(x))\n"

# Testing inputs
```

```
timeit.timeit(
  "lgamma_memoized(10)",
  setup = lgam_mem_source,
  number = 100
  )
```

3.369999467395246e-05

```
timeit.timeit(
  "math.lgamma(10)",
  setup = "import math\n",
  number = 100
  )
```

1.2800010154023767e-05

```
# A silly third test function
#
# note: this is silly because
# when we just repeatedly use
# the value stored in memory
# we are just going to get the
# same sample repeatedly

@memoizer
def unif_sample_memoized(size):
  return(np.random.uniform(size=size))

# Get the setup code to add to timeit
unif_mem_source = mem_source + "\n@memoizer\n"
unif_mem_source += "def unif_sample_memoized(size):\n    return(np.random.uniform(size=size))\n"
base_source = "import numpy as np\n"
new_source = base_source + unif_mem_source

# Testing simple inputs
timeit.timeit(
  "unif_sample_memoized(10**6)",
  setup = new_source,
  number = 100
  )
```

0.004901000007521361

```
timeit.timeit(
  "np.random.uniform(size=10**6)",
  setup = base_source,
  number = 100
  )
```

0.4937443000089843

For inputs that take very little time to evaluate we don't see a major difference in computational efficiency but often times it is the case that just running the normal function works faster. For larger inputs (for example summing the large normal vector) there is a huge increase in performance using memoization.

As a side note I tested the timeit before including a print statement if the object was saved in memory and it worked, so when we are using our memoization functions they are indeed just using the values stored in the memory dictionary. Here is an example using a slightly modified version (I didn't add print statements because they appeared every time the function was run with timeit).

```
# Got rid of comments to save space, function is essentially
# the same as before with an added print statement
def memoizer_alt(fcn):
  memory = {}
  def memory_checker(*args, **kwargs):
    arguments = str(args) + str(kwargs)
    if arguments not in memory.keys():
      memory[arguments] = fcn(*args, **kwargs)
    else:
      print("Function value already stored in memory")
    return(memory[arguments])
  return(memory_checker)

mem_alt_source = inspect.getsource(memoizer_alt)

@memoizer_alt
def sum_memoized_alt(lst):
  return(sum(lst))

sum_mem_alt_source = mem_alt_source + "\n@memoizer_alt\n"
sum_mem_alt_source += "def sum_memoized_alt(lst):\n    return(sum(lst))\n"

timeit.timeit(
  "sum_memoized_alt([1,2,3])",
  setup = sum_mem_alt_source,
  number = 2
  )
```

```
Function value already stored in memory
```

```
3.960001049563289e-05
```

2.

a.

Recall that lists store values that reference the location in memory of the objects "contained" in the list not the actual objects themselves.

First we will look at lists of real valued numbers (floats as they are called in Python):

```
m = 10000
lst_sizes1 = [sys.getsizeof(list(np.random.normal(loc=0,scale=1,size=n))) for n in range(m)]
lst_sizes2 = [sys.getsizeof([1.0]*n) for n in range(m)]
lst_sizes3 = [sys.getsizeof([float(i) for i in range(n)]) for n in range(m)]

lst_sizes1[0:10]
```

```
[56, 72, 72, 88, 88, 104, 104, 120, 120, 136]
```

```
lst_sizes2[0:10]
```

```
[56, 64, 72, 80, 88, 96, 104, 112, 120, 128]
```

```
lst_sizes3[0:10]
```

```
[56, 88, 88, 88, 88, 120, 120, 120, 120, 184]
```

```
abs_errors1 = [abs(56 + 8*n - lst_sizes1[n]) for n in range(m)]
abs_errors2 = [abs(56 + 8*n - lst_sizes2[n]) for n in range(m)]
abs_errors3 = [abs(56 + 8*n - lst_sizes3[n]) for n in range(m)]

rel_errors1 = [abs_errors1[n]/lst_sizes1[n] for n in range(m)]
rel_errors2 = [abs_errors2[n]/lst_sizes2[n] for n in range(m)]
rel_errors3 = [abs_errors3[n]/lst_sizes3[n] for n in range(m)]

max(abs_errors1)
```

```
8
```

```
max(abs_errors2)
```

```
0
```

```
max(abs_errors3)
```

```
9496
```

```
rel_errors1[-10:]
```

```
[0.0,
 0.00010001000100010001,
 0.0,
 9.999000099990002e-05,
 0.0,
 9.997000899730081e-05,
 0.0,
 9.995002498750625e-05,
 0.0,
 9.9930048965724e-05]
```

```
rel_errors2[-10:]
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
rel_errors3[-10:]
```

```
[0.06105006105006105,
 0.060956137879214806,
 0.060862214708368556,
 0.060768291537522307,
 0.06067436836667606,
 0.060580445195829814,
 0.060486522024983565,
 0.060392598854137315,
 0.060298675683291066,
 0.06020475251244482]
```

We see that given a method of generating lists of real numbers there is a pattern in the amount of memory based on the size of the list, however across methods of generating lists of real numbers there is not a consistency in the exact pattern. However, the memory generally increases linearly in the size of the list (with there being about 8 bytes per element) as shown by the relative errors of the projected size $(56 + 8 \times n)$ and the actual size. Note that the second method follows this linear relation exactly, the first method follows it very closely, and the final method definitely has larger deviation but this only happens as n gets very large so the relative error is actually quite low since the error is a fraction of the total size of memory.

Now we will examine behavior with more complicated objects:

```
k = 100
lst_sizes4 = [
  sys.getsizeof([
    list(np.random.normal(loc=0,scale=1,size=n)) for n in range(l)
    ]) for l in range(k)
  ]
lst_sizes5 = [sys.getsizeof([[1.0]*n for n in range(l)]) for l in range(k)]
lst_sizes6 = [sys.getsizeof([[float(i) for i in range(n)] for n in range(l)]) for l in range(k)]

lst_sizes4 == lst_sizes5 == lst_sizes6
```

```
True
```

```
lst_sizes4[0:10]
```

```
[56, 88, 88, 88, 88, 120, 120, 120, 120, 184]
```

```
abs_errors4 = [abs(56 + 8*n - lst_sizes4[n]) for n in range(k)]

rel_errors4 = [abs_errors4[n]/lst_sizes4[n] for n in range(k)]

max(abs_errors4)
```

```
120
```

```
rel_errors4[-10:]
```

```
[0.020202020202020204,
 0.010101010101010102,
 0.0,
 0.13043478260869565,
 0.12173913043478261,
 0.11304347826086956,
 0.10434782608695652,
 0.09565217391304348,
 0.08695652173913043,
 0.0782608695652174]
```

```
lst1 = [
    [],
    ["a"],
    [1,2],
    [{}],
    [{"A": "a", "B": "b"}, 4, 0.45253],
    [17]
  ]

lst2 = [
    [1, 2, 3, 4],
    [1, 2, 5, 9, 3],
    [6, 0, 3, 5, 2, 5]
  ]

size1 = sys.getsizeof(lst1)
size2 = sys.getsizeof(lst2)

lst_sizes4[len(lst1)] == size1
```

```
False
```

```
lst_sizes4[len(lst2)] == size2
```

```
False
```

At first the different methods of generating lists seem consistent as we have a number of ways to generate a list of lists of real numbers we get the same sizes for the same lengths. However we see when we manually create lists that this is not consistent. So once again within a method of generating the lists there seems to be a pattern in the memory usage but accross methods there is no clear exact pattern. However once again they are approximately linear in list length. First of all manually creating lists of lists uses exacly a linear amount of memory (Memory $= 56 + 8 \times n$) and the other methods I used are approximately linear as shown in the relative difference vector.

b.

```
# Real numbers
lst = list(np.random.uniform(low=0,high=1,size=10))
lst2 = lst.copy()
lst1 == lst2
lst2
```

```
[0.5030757651337522,
 0.1909817114488992,
 0.7136668940386087,
 0.31550892895664384,
 0.5052086163762411,
 0.7007890135490041,
 0.8463053413541366,
 0.3379934389567617,
 0.84255827940746,
 0.6934023403087862]
```

```
lst[1] = -1
lst
```

```
[0.5030757651337522,
 -1,
 0.7136668940386087,
 0.31550892895664384,
 0.5052086163762411,
 0.7007890135490041,
 0.8463053413541366,
 0.3379934389567617,
 0.84255827940746,
 0.6934023403087862]
```

```
lst2
```

```
[0.5030757651337522,
 0.1909817114488992,
 0.7136668940386087,
 0.31550892895664384,
 0.5052086163762411,
 0.7007890135490041,
 0.8463053413541366,
 0.3379934389567617,
 0.84255827940746,
 0.6934023403087862]
```

```
lst2[0] = -2
lst
```

```
[0.5030757651337522,
 -1,
 0.7136668940386087,
 0.31550892895664384,
 0.5052086163762411,
 0.7007890135490041,
 0.8463053413541366,
 0.3379934389567617,
 0.84255827940746,
 0.6934023403087862]
```

```
lst2
```

```
[-2,
 0.1909817114488992,
 0.7136668940386087,
 0.31550892895664384,
 0.5052086163762411,
 0.7007890135490041,
 0.8463053413541366,
 0.3379934389567617,
 0.84255827940746,
 0.6934023403087862]
```

```
# More complicated objects
lst3 = [[430384.30418, 40224.24], {"A": "a", "B": "b"}, 4, 9.10, 435.249]
lst4 = lst3.copy()
lst3 == lst4
```

```
True
```

```
lst4
```

```
[[430384.30418, 40224.24], {'A': 'a', 'B': 'b'}, 4, 9.1, 435.249]
```

```
lst3[0][1] = 3
lst3
```

```
[[430384.30418, 3], {'A': 'a', 'B': 'b'}, 4, 9.1, 435.249]
```

```
lst4
```

```
[[430384.30418, 3], {'A': 'a', 'B': 'b'}, 4, 9.1, 435.249]
```

```
lst3[1] = {}
lst3
```

```
[[430384.30418, 3], {}, 4, 9.1, 435.249]
```

```
lst4
```

```
[[430384.30418, 3], {'A': 'a', 'B': 'b'}, 4, 9.1, 435.249]
```

```
lst4[0][0] = 4
lst3
```

```
[[4, 3], {}, 4, 9.1, 435.249]
```

```
lst4
```

```
[[4, 3], {'A': 'a', 'B': 'b'}, 4, 9.1, 435.249]
```

```
lst4[0] = []
lst3
```

```
[[4, 3], {}, 4, 9.1, 435.249]
```

```
lst4
```

```
[[], {'A': 'a', 'B': 'b'}, 4, 9.1, 435.249]
```

```
lst3[2] = 0
lst3
```

```
[[4, 3], {}, 0, 9.1, 435.249]
```

```
lst4
```

```
[[], {'A': 'a', 'B': 'b'}, 4, 9.1, 435.249]
```

```
lst4[3] = 3.1415
lst3
```

```
[[4, 3], {}, 0, 9.1, 435.249]
```

```
lst4
```

```
[[], {'A': 'a', 'B': 'b'}, 4, 3.1415, 435.249]
```

```
help(list.copy)
```

```
Help on method_descriptor:

copy(self, /)
    Return a shallow copy of the list.
```

Whether or not there are nested elements in the list changing a non nested element of the copied or the original list does not change the corresponding element in the other this is because the element is simply changed, no reference to the original object in memory is needed. This is different for nested elements. Since copy only creates a shallow copy (which means it just uses a reference to the elements in the copied list) so when you modify an element in one of the nested elements in either list (original or copied) the other list sees the same change since it modifies the object in memory that the list has the reference to.

c.

```
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']
```

First trivially we know that since 1 is an int type and 1.0 is a float type that "1.0" and "1" will reference different objects in memory (similarly for "2.0" vs "2" and of course none of the strings will be the same reference to any of the numbers).

Then we know that the storage of my_int, my_real, and my_string will just be references to those objects in memory. I will show this with the id and is functions later. Similarly, all of the stored integers and strings that are equal (ex: 1 and my_int) are references to the same object in memory. The floats will be different, each individual float will be a reference to a different object in memory. This is according to the Unit 5 notes. I'm not sure why the python developers made this decision for floats and not for ints and strings, maybe because ints and strings are in some way less complicated objects?

But to summarize: Each "1" and "my_int" element in xi and yi will be a reference to the same object in memory, the "2" in xi and the "2" in yi will be references to the same object in memory. Each "hat" and "my_string" element in xc and yc will be a reference to the same object in memory, the "dog" in xc and the "dog" in yc will be references to the same object in memory. Then each "my_real" in xr and yr is a reference to the same object in memory. Then every other individual float will be a reference to its own standalone object in memory. This is because of how python stores ints and strings themselves (getting rid of copies) as opposed to floats (keeping possible copies).

```
[id(x) for x in xi]
```

```
[140708640973608, 140708640973608, 140708640973608, 140708640973640]
```

```
[id(y) for y in yi]
```

```
[140708640973608, 140708640973608, 140708640973608, 140708640973640]
```

```
# These are the same
[id(x) for x in xi] == [id(y) for y in yi]
```

```
True
```

```
# We only need to do this for one since they are the same
[[xi[i] is xi[j] for i in range(4)] for j in range(4)]
```

```
[[True, True, True, False],
 [True, True, True, False],
 [True, True, True, False],
 [False, False, False, True]]
```

```
[id(x) for x in xr]
```

```
[1842163974768, 1840464334672, 1840464334672, 1840466878768]
```

```
[id(y) for y in yr]
```

```
[1840466882928, 1840464334672, 1840464334672, 1840466882640]
```

```
[[xr[i] is xr[j] for i in range(4)] for j in range(4)]
```

```
[[True, False, False, False],
 [False, True, True, False],
 [False, True, True, False],
 [False, False, False, True]]
```

```
[[yr[i] is yr[j] for i in range(4)] for j in range(4)]
```

```
[[True, False, False, False],
 [False, True, True, False],
 [False, True, True, False],
 [False, False, False, True]]
```

```
# We only need one of these since by doing the above and
# this we have checked every element in each list with
# every element in each list
[[xr[i] is yr[j] for i in range(4)] for j in range(4)]
```

```
[[False, False, False, False],
 [False, True, True, False],
 [False, True, True, False],
 [False, False, False, False]]
```

```
[id(x) for x in xc]
```

```
[1842539925808, 1842539925808, 1842539925808, 1840466911408]
```

```
[id(y) for y in yc]
```

```
[1842539925808, 1842539925808, 1842539925808, 1840466911408]
```

```
# These are the same
[id(x) for x in xc] == [id(y) for y in yc]
```

```
True
```

```
# We only need to do this for one since they are the same
[[xc[i] is xc[j] for i in range(4)] for j in range(4)]
```

```
[[True, True, True, False],
 [True, True, True, False],
 [True, True, True, False],
 [False, False, False, True]]
```

This is consistent with what we said before.

3.

a.

In order to find the i, j entry from a matrix multiplication of two $n \times n$ matrices X and Y you need to take the dot product of the ith row of X with the jth column of Y. This involves n multiplications followed by $n-1$ additions so the time complexity of this step is $O(n) + O(n-1) = O(n+n-1) = O(2n-1) = O(n)$. Then there are n^2 total such computations since the resulting matrix $A = XY$ will be $n \times n$. Therefore the time complexity of the matrix multiplication is $O(n^2)O(n) = O(n^3)$. The time complexity of getting the diagonal is $O(n)$ so the total time complexity up to now is $O(n^3) + O(n) = O(n^3 + n) = O(n^3)$. Finally taking the trace just amounts to $n-1$ additions so the total time complexity would be $O(n^3) + O(n-1) = O(n^3 + n - 1) = O(n^3)$. We were told we could just find the time complexity of the multiplication but the final step was simple so I just left it.

This approach is naive because we don't need to compute the product of the matrices, we can simply calculate the resulting entries that will be on the diagonal. Again for each diagonal entry you need to take the dot product of the ith row of X with the ith column of Y which we saw before has time complexity $O(n)$. Then there are only n such computations since there are n diagonal elements. So the total time complexity up to this point now is $O(n)O(n) = O(n^2)$. Then now we only need to sum these values which takes $n-1$ additions so the total time complexity here would be $O(n^2) + O(n-1) = O(n^2 + n - 1) = O(n^2)$.

b.

Theoretically we could transpose B then take the elementwise product of the rows of A with the rows B^T we will get a matrix where the sum of the ith row of that matrix is the dot product of the ith row of A with the ith column of B. Therefore if we sum over the sums of all the rows of such matrix (i.e. just sum all of the elements) we will get the trace. We can use vectorized code to implement the exact procedure described above.

Here is quickly an example that A.T is faster than np.transpose(A), I saw a question about it in the Unit 5 notes.

```
# Get setup for timeit
base_source = "import numpy as np\nX = np.random.normal(loc=0,scale=1,size=(1000,1000))\n"

# Try .T approach
timeit.timeit(
  "X.T",
  setup = base_source,
  number = 100
  )
```

```
8.400005754083395e-06
```

```
# Try np.transpose approach
timeit.timeit(
  "np.transpose(X)",
  setup = base_source,
  number = 100
  )
```

```
3.070000093430281e-05
```

So A.T is clearly faster, I am not sure why. Now writing and testing our function.

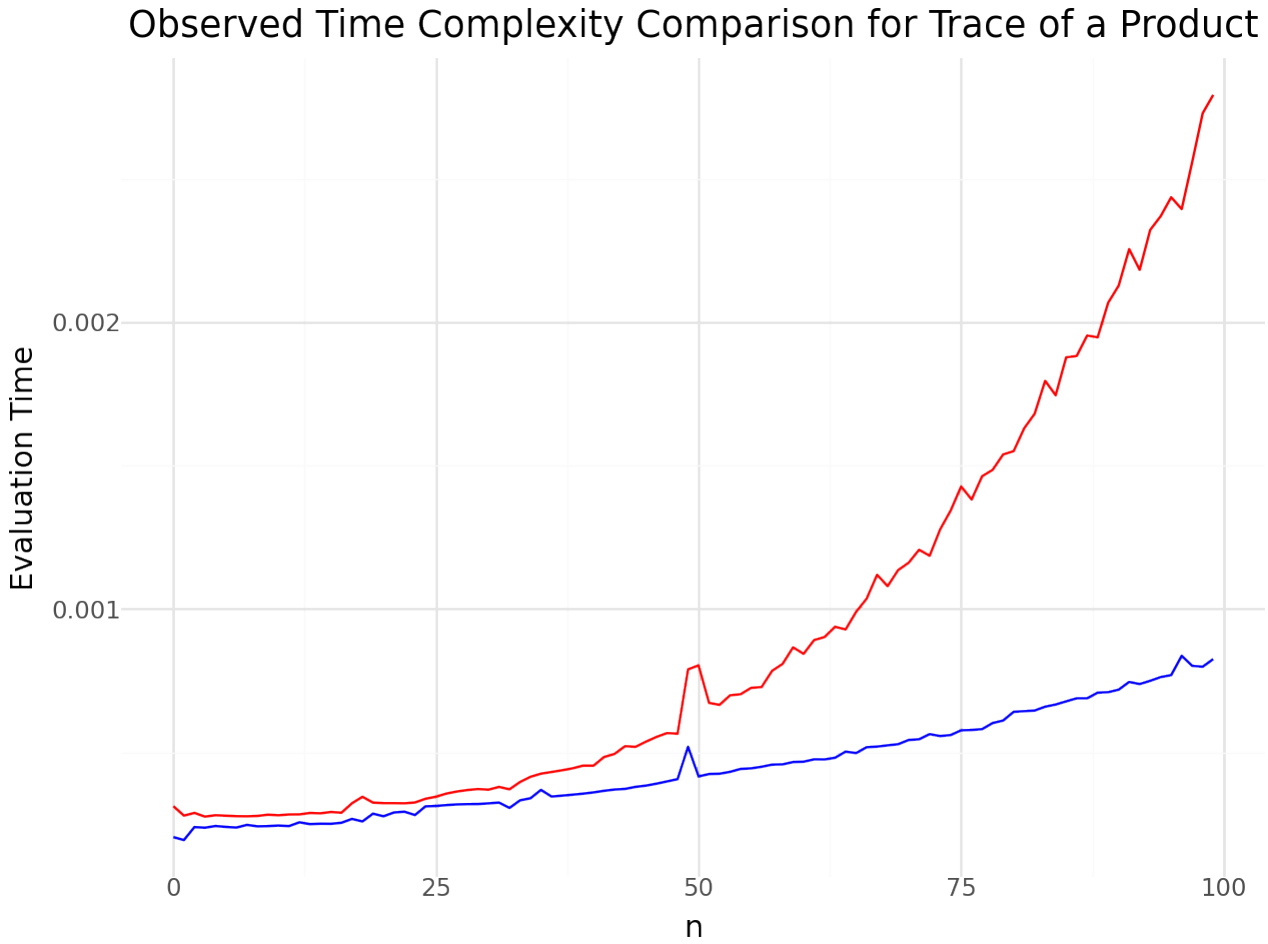
```
def prod_trace(X,Y):
  return(np.sum(X*Y.T))
```

When you take the transpose of a matrix you are changing the i,jth entry into the j,ith entry then since there n^2 entries this is an $O(n^2)$ operation at most (I read somewhere that it is constant but this doesn't make a difference since the sum is $O(n^2)$ anyways) then after that we do a vectorized multiplication of Y^T with X which involved n^2 multiplications so this step is $O(n^2)$ but in practice will actually be faster because it is being performed with vectorization (which according to Chris will use C a staticly typed language) and each row might be run on a separate process (parallelization) to speed time up. So the total process so far is $O(n^2) + O(1) = O(n^2 + 1) = O(n^2)$ which again in practice is faster. Then the final step involved $n^2 - 1$ additions so the total time complexity is $O(n^2) + O(n^2 - 1) = O(2n^2 - 1) = O(n^2)$ which again in practice is actually faster.

This compares the performance of the naive implementation to my implementation. First here are comparisons of actual data run:

```
# Get the source of my function
prod_trace_source = inspect.getsource(prod_trace)

# Initialize lists
naive_times = []
times = []
```

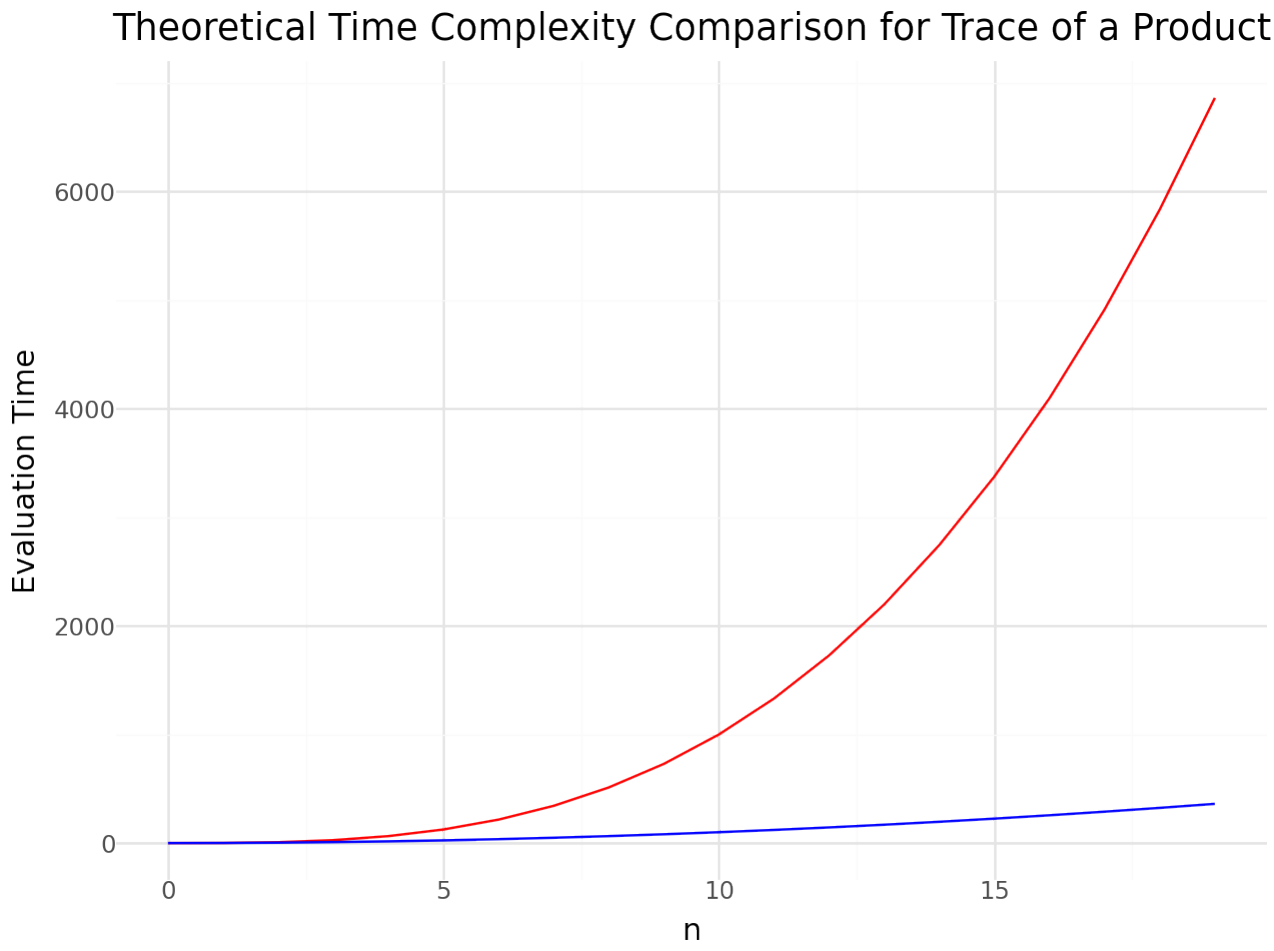
```
for j in range(100):
  # Make the setup for timeit
  base_source = "import numpy as np\n"
  base_source += f"X = np.random.normal(loc=0,scale=1,size=({j},{j}))\n"
  base_source += f"Y = np.random.normal(loc=0,scale=1,size=({j},{j}))\n"

  # Run naive approach and add to the times
  naive_times.append(timeit.timeit(
    "np.sum(np.diag(X@Y))",
    setup = base_source,
    number = 100
  ))

  # Run improved approach and add to the times
  times.append(timeit.timeit(
    "prod_trace(X,Y)",
    setup = base_source + prod_trace_source,
    number = 100
  ))

# Combine into dictionary and make a dataframe
time_comp = {"n": list(range(100)), "Naive Times": naive_times, "Improved Times": times}
time_comp = pd.DataFrame(time_comp)

# Graph results
(
ggplot(time_comp,
       aes(x = "n")
      ) +
      geom_line(aes(y = "Naive Times"),
                color = "red",
               ) +
      geom_line(aes(y = "Improved Times"),
                color = "blue",
               ) +
      # Relabel graph
      labs(title = "Observed Time Complexity Comparison for Trace of a Product",
           y = "Evaluation Time") +
      # Use a simplistic theme
      theme_minimal()
).show()
```



Now here are the theoretical time complexity comparisons:

```
# Make x variable for n
n = list(range(20))

# Make the improved theoretical time complexity
n_squared = [j**2 for j in n]

# Make the naive theoretical time complexity
n_cubed = [j**3 for j in n]

# Combine into dictionary and make a dataframe
theoretical_time_comp = {"n": n, "Theoretical Naive Times": n_cubed, "Theoretical Improved Times": n_s
theoretical_time_comp = pd.DataFrame(theoretical_time_comp)

# Graph results
(
ggplot(theoretical_time_comp,
       aes(x = "n")
      ) +
      geom_line(aes(y = "Theoretical Naive Times"),
                color = "red",
               ) +
      geom_line(aes(y = "Theoretical Improved Times"),
                color = "blue",
               ) +
      # Relabel graph
```

```
      labs(title = "Theoretical Time Complexity Comparison for Trace of a Product",
           y = "Evaluation Time") +
      # Use a simplistic theme
      theme_minimal()
).show()
```



We can see this doesn't quite look like the observed comparison graph if they were on the same scale. This is probably because there are things going on behind the scenes that make the naive implementation perform a little better than $O(n^3)$. Nonetheless, in either case we see the improved function does significantly better.

4.

```
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, 5)} seconds.")
```

```
Loop by column time: 0.92112 seconds.
```

a.

We might think this would be faster because the probabilities in the matrix are stored by row so it would be faster to access the probabilities to make the sample if we transpose and loop by row. Let's test it:

```
tmp = np.exp(np.random.randn(p, n))
probs = tmp / tmp.sum(axis=0)
probs = probs.T

smp = np.zeros(n, dtype=int)
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, 5)} seconds.")
```

```
Loop by column time: 0.92225 seconds.
```

We can see there really isn't a significant difference. The reason there isn't really a big increase in efficiency is that the probability matrix is not large so the cache can actually just store the whole matrix all at once and there is not a significant difference in the time to loop by column or by row because the cache does not need to be repeatedly freed up and the matrix reloaded to get the columns.

```
sys.getsizeof(probs)
```

```
128
```

Now we will test using the apply function:

```
tmp = np.exp(np.random.randn(p, n))
probs = tmp / tmp.sum(axis=0)

smp = np.zeros(n, dtype=int)
# Make function to sample
sampler = lambda prob: np.random.choice(p,size=1,p=prob)[0]

# Generate sample by column.
start = time.time()
smp = np.apply_along_axis(sampler, axis=0, arr=probs)
print(f"Apply function time: {round(time.time() - start, 5)} seconds.")
```

```
Apply function time: 0.99969 seconds.
```

The apply function is not any more efficient, in fact it takes longer. This is because numpy does not use vectorization with its apply function so this code is essentially acting as a loop anyways.

b.

The way you sample from a distribution X with uniform random variables is you take a uniform random variables $U_1, U_2, ..., U_n \overset{\text{iid}}{\sim} \text{Uniform}(0,1)$ then you take the CDF $F_X(x)$ of your random variable X and then your sample $X_1, X_2, ..., X_n$ is defined such that X_i is the smallest value of x_i such that $F_X(x_i) \geq U_i$ or formally $X_i = min\{x_i : F_X(x_i) \geq U_i\}$.

In order to do this in our example first we need to find the CDF for each of the columns in our probability matrix (which amounts to finding the cumulative sum vector for each column). Then we get a sample of uniform random variables and for each of those columns we take the index of the smallest value such that the cumulative probability is greater than or equal to our uniform random variable. We know that the minimum over all x_i must be one of the values in the cumulative vector because for discrete random variables this is where the CDF jumps. We only see an increase in the CDF at values that have nonzero probability (i.e. the values that are possible to observe).

We can do this by first utilizing the vectorized code cumul_probs >= unif which for each column makes a vector of booleans indicating whether or not each cumulative probability is greater than the corresponding uniform value found. Then argmax using the axis=0 argument gets the index of first the max value of each column, since these are all booleans the first index for a True value will be returned for each column which is precisely the category we would get as our sample.

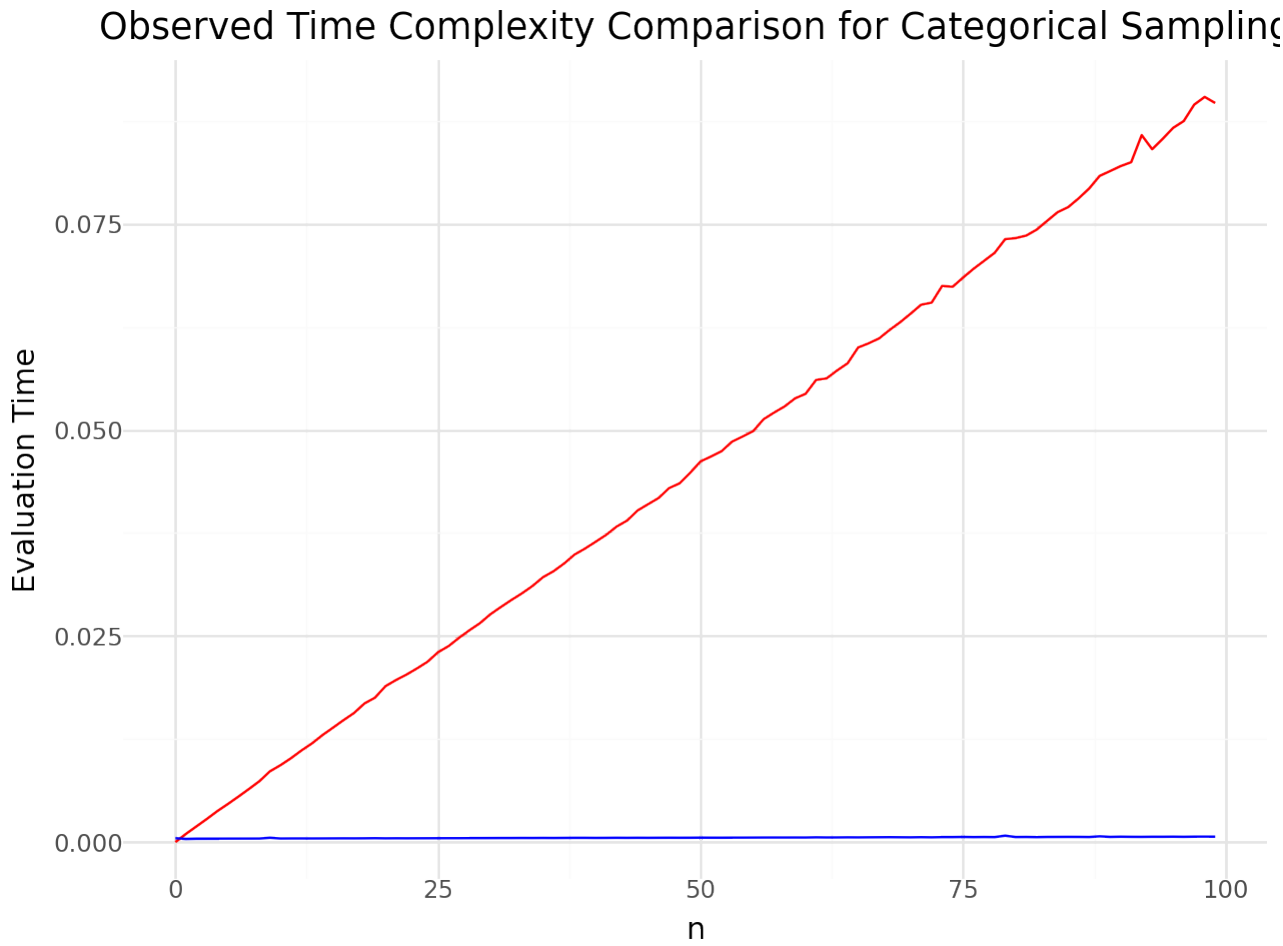
```
tmp = np.exp(np.random.randn(p, n))
probs = tmp / tmp.sum(axis=0)

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

# Generate sample by column.
start = time.time()
unif = np.random.uniform(low=0,high=1,size=n)
cumul_probs = np.cumsum(probs, axis=0)
smp = np.argmax(cumul_probs >= unif, axis = 0)

print(f"Improved approach time: {round(time.time() - start, 5)} seconds.")
```

```
Improved approach time: 0.00401 seconds.
```

We can see this is significantly faster. Here is some testing below to determine how this approach and the original approach perform as n increases:

```
def initial_approach(n, p, probs):
  # Generate sample by column.
  start = time.time()
  for i in range(n):
    smp[i] = np.random.choice(p, size=1, p=probs[:,i])[0]
  return(time.time() - start)

def improved_approach(n, p, probs):
  # Generate sample by column.
  start = time.time()
  unif = np.random.uniform(low=0,high=1,size=n)
  cumul_probs = np.cumsum(probs, axis=0)
  smp = np.argmax(cumul_probs >= unif, axis = 0)
  return(time.time() - start)

initial_source = inspect.getsource(initial_approach)
improved_source = inspect.getsource(improved_approach)
base_source = "import numpy as np\nimport time\n"

# Initialize lists
initial_times = []
improved_times = []

for j in range(100):
  # Make the setup for timeit
  base_source = f"import numpy as np\nimport time\np=5\nn={j}\n"
  base_source += f"tmp = np.exp(np.random.randn({p}, n))\n"
  base_source += "probs = tmp / tmp.sum(axis=0)\n"
  base_source += "smp = np.zeros(n, dtype=int)\n"

  # Run naive approach and add to the times
  initial_times.append(timeit.timeit(
    "initial_approach(n, p, probs)",
    setup = base_source + initial_source,
    number = 100
  ))

  # Run improved approach and add to the times
  improved_times.append(timeit.timeit(
    "improved_approach(n, p, probs)",
    setup = base_source + improved_source,
    number = 100
  ))

# Combine into dictionary and make a dataframe
time_comp = {"n": list(range(100)), "Initial Times": initial_times, "Improved Times": improved_times}
time_comp = pd.DataFrame(time_comp)

# Graph results
(
ggplot(time_comp,
```

```
       aes(x = "n")
      ) +
      geom_line(aes(y = "Initial Times"),
                color = "red",
               ) +
      geom_line(aes(y = "Improved Times"),
                color = "blue",
               ) +
      # Relabel graph
      labs(title = "Observed Time Complexity Comparison for Categorical Sampling",
           y = "Evaluation Time") +
      # Use a simplistic theme
      theme_minimal()
).show()
```



As expected the original approach seems to have linear time complexity and the improved approach seems to actually have constant time complexity