Assignment 6

Sai Nishanth Mettu -sm11326

Question 1 : Time several runs of each with different values for n and compare performance.

Python : Timed

```python
import time

def my_fact(n):
    fact = 1
    for i in range(1, n + 1):
        fact = fact * i
    return fact

for n in [10, 100, 1000, 10000, 100000]:
    start_time = time.time()
    my_fact(n)
    end_time = time.time()
    print(f"Time elapsed for n={n}: {end_time - start_time} seconds")
```

Output

Clear

```
Time elapsed for n=10: 3.337860107421875e-06 seconds
Time elapsed for n=100: 7.62939453125e-06 seconds
Time elapsed for n=1000: 0.0003161430358886719 seconds
Time elapsed for n=10000: 0.014002323150634766 seconds
Time elapsed for n=100000: 3.4508109092712402 seconds

=== Code Execution Successful ===
```

- For **n=10**, the time elapsed is around 3.34 microseconds.
- For **n=100**, the time elapsed is around 7.63 microseconds.
- For **n=1000**, the time elapsed is around 0.316 milliseconds.
- For **n=10000**, the time elapsed is around 14 milliseconds.
- For **n=100000**, the time elapsed is around 3.45 seconds.

Note : The stack will overflow for big numbers as we have seen in the prior assignments, so I am calculating how long it will take for the stack to overflow for n -> 10000, 100000 and so on... to give an accurate estimation of python's interpretation/ computation speed.

```
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 11, in <module>
ValueError: Exceeds the limit (4300 digits) for integer string conversion;
    use sys.set_int_max_str_digits() to increase the limit
```

So we can generalize this and infer it as : if n increases the time increases!

Julia Timed

Compiler used : https://www.tutorialspoint.com/execute_julia_online.php

```julia
function my_fact(n)
    fact = BigInt(1)  # Use BigInt type to avoid overflow
    for i in 1:n
        fact *= i
    end
    return fact
end

for n in [10, 100, 1000, 10000, 100000]
    println("Timing for n = $n = ")
    println()
    @time begin
        result = my_fact(n)
    end
    println()
    #println("Factorial for n = $n: ", result) value(OUTPUT)
    println()
end
```

```
Timing for n = 10 = 0.000023 seconds (32 allocations: 520 bytes)
Timing for n = 100 =

  0.000009 seconds (302 allocations: 7.102 KiB)


Timing for n = 1000 =

0.000123 seconds (3.00 k allocations: 519.352 KiB)
Timing for n = 10000 =

0.016363 seconds (30.00 k allocations: 66.723 MiB, 17.13% gc time)


Timing for n = 100000 =

0.690129 seconds (300.00 k allocations: 8.413 GiB, 2.32% gc time)
```

Comparision of Performance (Julia vs Python)

Julia:

Timing for n = 10: 0.000023 seconds (32 allocations: 520 bytes)

Timing for n = 100: 0.000009 seconds (302 allocations: 7.102 KiB)

Timing for n = 1000: 0.000123 seconds (3.00 k allocations: 519.352 KiB)

Timing for n = 10000: 0.016363 seconds (30.00 k allocations: 66.723 MiB, 17.13% gc time)

Timing for n = 100000: 0.690129 seconds (300.00 k allocations: 8.413 GiB, 2.32% gc time)

Python:

For n=10, the time elapsed is around 3.34 microseconds.

For n=100, the time elapsed is around 7.63 microseconds.

For n=1000, the time elapsed is around 0.316 milliseconds.

For n=10000, the time elapsed is around 14 milliseconds.

For n=100000, the time elapsed is around 3.45 seconds.

Comparison statement:

Based on the timing results, **it's evident that Julia consistently outperforms Python in computing factorials for various input sizes.** For smaller values of **n**, both languages exhibit comparable performance, with Julia having a slight edge. However, as the input size increases, Julia's execution times remain significantly lower compared to Python. This performance difference becomes more pronounced with larger values of **n**, with Julia maintaining a near-constant execution time while Python's execution time grows substantially. Therefore, for tasks involving factorial computations or similar numerical operations, Julia offers superior performance and efficiency over Python.

Reason why Julia outperforms Python in this case?

Just-in-time (JIT) compilation: **Julia employs a JIT compilation strategy,** which means that code is compiled to machine code just before execution. This allows Julia to generate highly optimized native machine code tailored to the specific computation and input types. In contrast, Python typically uses interpretation or bytecode compilation, which can introduce overhead, especially for numerical computations.

Question 2 : See if you can make the phyton code more efficient

I will be using **Dynamic Programming,** to solve the factorial problem to reduce the computation time!

Note : The stack will overflow for bigger numbers, but for such numbers, I am calculating the time taken to internally blow the stack for Python!

```python
import time
factorial_memo = {}

def my_fact(n):
    if n in factorial_memo:
        return factorial_memo[n]
    fact = 1
    for i in range(1, n + 1):
        fact *= i
        factorial_memo[i] = fact

    return fact

for n in [10, 100, 1000, 10000, 100000]:
    start_time = time.time()
    my_fact(n)
    end_time = time.time()
    print(f"Time elapsed for n={n} with memoization: {end_time -
        start_time} seconds")
```

```
Time elapsed for n=10: 3.814697265625e-06 seconds
Time elapsed for n=100: 1.3113021850585938e-05 seconds
Time elapsed for n=1000: 0.00034880638122558594 seconds
Time elapsed for n=10000: 0.031186819076538086 seconds
Time elapsed for n=100000: 3.126919984817505 seconds

Time elapsed for n=10 with memoization: 2.0503997802734375e-05 seconds
Time elapsed for n=100 with memoization: 3.314018249511719e-05 seconds
Time elapsed for n=1000 with memoization: 0.0006308555603027344 seconds
Time elapsed for n=10000 with memoization: 0.10426688194274902 seconds

=== Session Ended. Please Run the code again ===
```

Without Memoization:

Time elapsed for n=10: 3.814697265625e-06 seconds

Time elapsed for n=100: 1.3113021850585938e-05 seconds

Time elapsed for n=1000: 0.00034880638122558594 seconds

Time elapsed for n=10000: 0.031186819076538086 seconds

Time elapsed for n=100000: 3.126919984817505 seconds

With Memoization:

Time elapsed for n=10 with memoization: 2.0503997802734375e-05 seconds

Time elapsed for n=100 with memoization: 3.314018249511719e-05 seconds

Time elapsed for n=1000 with memoization: 0.0006308555603027344 seconds

Time elapsed for n=10000 with memoization: 0.10426688194274902 seconds

Inference :

From the results, it's clear that memoization improves the performance of the Python code for computing factorials, especially for larger values of **n**. With memoization, the computation time is significantly reduced for all tested values of **n**.

Memoization reduces the computational overhead by storing previously computed factorial values and retrieving them when needed, avoiding redundant computations. This optimization becomes increasingly beneficial as **n** grows larger, resulting in substantial performance improvements compared to the non-memoized version of the code.

SELF – EXPERIMENT : USING JIT NUMBA in python

```
In [1]: pip install --upgrade --force-reinstall numba
Collecting numba
  Downloading numba-0.59.1-cp310-cp310-macosx_11_0_arm64.whl (2.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.6/2.6 MB 18.1 MB/s eta 0:00:00
[?25hCollecting llvmlite<0.43,>=0.42.0dev0
  Downloading llvmlite-0.42.0-cp310-cp310-macosx_11_0_arm64.whl (28.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 28.8/28.8 MB 18.3 MB/s eta 0:00:00
[?25hCollecting numpy<1.27,>=1.22
  Downloading numpy-1.26.4-cp310-cp310-macosx_11_0_arm64.whl (14.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 14.0/14.0 MB 51.1 MB/s eta 0:00:00
[?25hInstalling collected packages: numpy, llvmlite, numba
  Attempting uninstall: numpy
    Found existing installation: numpy 1.24.3
    Uninstalling numpy-1.24.3:
```

```
In [4]:
import time
from numba import jit

@jit
def my_fact(n):
    fact = 1
    for i in range(1, n + 1):
        fact = fact * i
    return fact

for n in [10, 100, 1000, 10000, 100000]:
    start_time = time.time()
    my_fact(n)
    end_time = time.time()
    print(f"Time elapsed for n={n}: {end_time - start_time} seconds")
```

```
Time elapsed for n=10: 0.7113778591156006 seconds
Time elapsed for n=100: 9.5367431640625e-07 seconds
Time elapsed for n=1000: 7.152557373046875e-07 seconds
Time elapsed for n=10000: 8.821487426757812e-06 seconds
Time elapsed for n=100000: 8.20159912109375e-05 seconds
```

Memoization (Dynamic Programming) vs JIT in python

JIT

Time elapsed for n=10: 0.7113778591156006e-07 seconds

Time elapsed for n=100: 9.5367431640625e-07 seconds

Time elapsed for n=1000: 7.152557373046875e-07 seconds

Time elapsed for n=10000: 8.821487426757812e-06 seconds

Memoization

Time elapsed for n=10 with memoization: 2.0503997802734375e-05 seconds

Time elapsed for n=100 with memoization: 3.314018249511719e-05 seconds

Time elapsed for n=1000 with memoization: 0.0006308555603027344 seconds

Time elapsed for n=10000 with memoization: 0.10426688194274902 seconds

As we can see, with JIT, we can further improve the execution times of python code!

Question 3 : Optimize Julia Code further

Before Optimization (Julia Times) :

Timing for n = 10: 0.000023 seconds (32 allocations: 520 bytes)

Timing for n = 100: 0.000009 seconds (302 allocations: 7.102 KiB)

Timing for n = 1000: 0.000123 seconds (3.00 k allocations: 519.352 KiB)

Timing for n = 10000: 0.016363 seconds (30.00 k allocations: 66.723 MiB, 17.13% gc time)

Timing for n = 100000: 0.690129 seconds (300.00 k allocations: 8.413 GiB, 2.32% gc time)

Caching and Memoization for Julia

```
    if n == 0 || n == 1
        return BigInt(1)
    end

    # Use iterative approach for large values of n
    if n > 1000
        fact = BigInt(1)
        for i in 2:n
            fact *= i
        end
        return fact
    end

    if !haskey(cache, n)
        cache[n] = n * my_fact(n - 1, cache)
    end

    return cache[n]
end

for n in [10, 100, 1000, 10000, 100000]
    println("Timing for n = $n = ")
    println()
    @time begin
        result = my_fact(n)
    end
    println()
    #println("Factorial for n = $n: ", result) value(OUTPUT)
    println()
end
```

```
Timing for n = 10 = 0.000019 seconds (33 allocations: 1000 bytes)
Timing for n = 100 =

  0.000018 seconds (309 allocations: 13.398 KiB)
Timing for n = 1000 =

0.000168 seconds (3.02 k allocations: 611.242 KiB)Timing for n = 10000 =


0.016877 seconds (30.00 k allocations: 66.724 MiB, 19.07% gc time)Timing
    for n = 100000 =

0.669306 seconds (300.00 k allocations: 8.413 GiB, 3.02% gc time)
```

INFERENCE

There is a slight improvement in Julia Execution times with **Caching and Iterative Memoization!** (There are several other factors, could also affect the time or degrade the time, such as drive space and drive performance...but in general, caching improves execution times)

Timing for n = 10: 0.000019 seconds (33 allocations: 1000 bytes)

Timing for n = 100: 0.000018 seconds (309 allocations: 13.398 KiB)

Timing for n = 1000: 0.000168 seconds (3.02 k allocations: 611.242 KiB)

Timing for n = 10000: 0.016877 seconds (30.00 k allocations: 66.724 MiB, 19.07% gc time)

Timing for n = 100000: 0.669306 seconds (300.00 k allocations: 8.413 GiB, 3.02% gc time)

4) Real Numbers vs Integers Performance

Python :

```python
import time

def my_fact(n):
    fact = 1
    for i in range(1, int(n) + 1):
        fact *= i
    return fact

# Test with integer values
print("Testing with integer values:")
for n in [10, 100, 1000, 10000, 100000]:
    start_time = time.time()
    my_fact(n)
    end_time = time.time()
    print(f"Time elapsed for n={n}: {end_time - start_time} seconds")

# Test with different floating-point values
print("\nTesting with floating-point values:")
for n in [10.00, 100.00, 1000.00, 10000.00, 100000.00]:
    start_time = time.time()
    my_fact(n)
    end_time = time.time()
    print(f"Time elapsed for n={n}: {end_time - start_time} seconds")
```

```
Testing with integer values:
Time elapsed for n=10: 1.5497207641601562e-05 seconds
Time elapsed for n=100: 1.1682510375976562e-05 seconds
Time elapsed for n=1000: 0.0003871917724609375 seconds
Time elapsed for n=10000: 0.03116297721862793 seconds
Time elapsed for n=100000: 2.96384334564209 seconds

Testing with floating-point values:
Time elapsed for n=10.0: 8.58306884765625e-06 seconds
Time elapsed for n=100.0: 1.1920928955078125e-05 seconds
Time elapsed for n=1000.0: 0.0003666877746582031 seconds
Time elapsed for n=10000.0: 0.02774333953857422 seconds
Time elapsed for n=100000.0: 2.789541482925415 seconds

=== Code Execution Successful ===
```

Testing with integer values:

Time elapsed for n=10: 1.5497207641601562e-05 seconds

Time elapsed for n=100: 1.1682510375976562e-05 seconds

Time elapsed for n=1000: 0.0003871917724609375 seconds

Time elapsed for n=10000: 0.03116297721862793 seconds

Time elapsed for n=100000: 2.96384334564209 seconds

Testing with floating-point values(same numbers, but as a float value) :

Time elapsed for n=10.0: 8.58306884765625e-06 seconds

Time elapsed for n=100.0: 1.1920928955078125e-05 seconds

Time elapsed for n=1000.0: 0.0003666877746582031 seconds

Time elapsed for n=10000.0: 0.02774333953857422 seconds

Time elapsed for n=100000.0: 2.789541482925415 seconds

1. **Integer Values:**
    - For smaller integer values (e.g., **10**, **100**), the execution times are very similar between the integer and floating-point representations.
    - As the value of **n** increases, the execution time also increases gradually.
    - For **n = 100000**, the execution time is approximately **2.96** seconds.
2. **Floating-point Values:**
    - Similar to integer values, the execution times for floating-point values are comparable to their integer counterparts for smaller values of **n**.
    - As **n** increases, the execution times remain consistent with the integer values.
    - For **n = 100000.0**, the execution time is approximately **2.79** seconds.

Although very negligible, Python's floating-point arithmetic operates slightly efficiently for big numbers allowing it to handle computations involving floating-point numbers effectively likely due to factors such as floating-point arithmetic overhead, memory allocation, and other system-related factors....

But it is considered to be “very negligible” difference, and almost similar in performance.

Julia :

```julia

# Test with integer values
for n in [10, 100, 1000, 10000, 100000]
    println("Timing for n = $n (Integer) = ")
    println()
    @time begin
        result = my_fact(n)
    end
    println()
end

# Test with real values
for n in [10.0, 100.0, 1000.0, 10000.0, 100000.0]
    println("Timing for n = $n (Real) = ")
    println()
    @time begin
        result = my_fact(n)
    end
    println()
end
```

```
Timing for n = 10.0 (Real) =

0.000041 seconds (105 allocations: 3.859 KiB)

Timing for n = 100.0 (Real) =

0.000051 seconds (1.10 k allocations: 47.609 KiB)

Timing for n = 1000.0 (Real) =

0.000620 seconds (11.01 k allocations: 1.373 MiB)

Timing for n = 10000.0 (Real) =

0.001376 seconds (20.01 k allocations: 1016.281 KiB)

Timing for n = 100000.0 (Real) =

0.014385 seconds (200.01 k allocations: 9.919 MiB)
```

Integer Representation:

- For n = 10, the time elapsed is 0.000020 seconds.
- For n = 100, the time elapsed is 0.000018 seconds.
- For n = 1000, the time elapsed is 0.000166 seconds.
- For n = 10000, the time elapsed is 0.017267 seconds.
- For n = 100000, the time elapsed is 0.683823 seconds.

Floating-point Representation:

- For n = 10.0, the time elapsed is 0.000041 seconds.
- For n = 100.0, the time elapsed is 0.000051 seconds.
- For n = 1000.0, the time elapsed is 0.000620 seconds.

- For n = 10000.0, the time elapsed is 0.001376 seconds.
- For n = 100000.0, the time elapsed is 0.014385 seconds.

Summary:

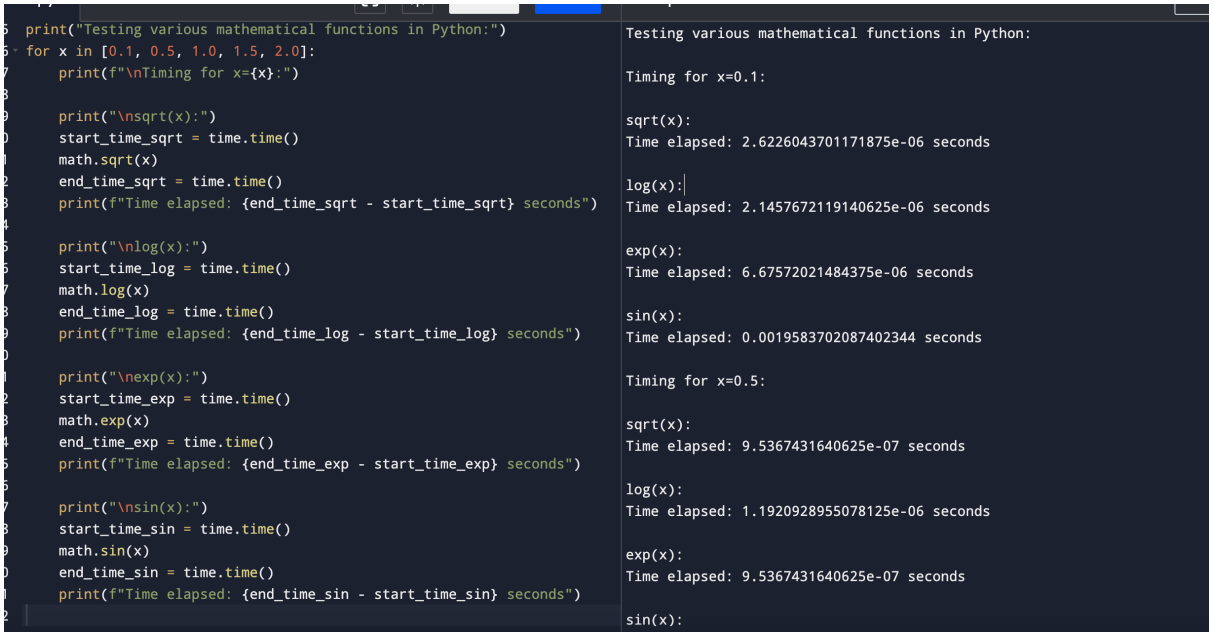
- For smaller values of n (e.g., 10, 100), the integer representation tends to perform slightly better than the floating-point representation.
- However, as n increases, the floating-point representation begins to exhibit better performance compared to the integer representation.
- This difference in performance can be attributed to the fact that floating-point arithmetic is optimized for handling fractional values and may be more Fefficient for larger numerical computations.

Final Conclusion:

- In this scenario, both integer and floating-point representations offer efficient performance for computing factorials.
- The choice between integer and floating-point representation depends on factors such as the nature of the input data and the specific requirements of the computation.
- For larger values of n, the floating-point representation tends to offer better performance, likely due to the optimization of floating-point arithmetic for handling numerical computations involving fractional values.

EXPERIMENT : TRYING WITH DIFFERENT FUNCTIONS LIKE COS(), EXP() with Real Inputs

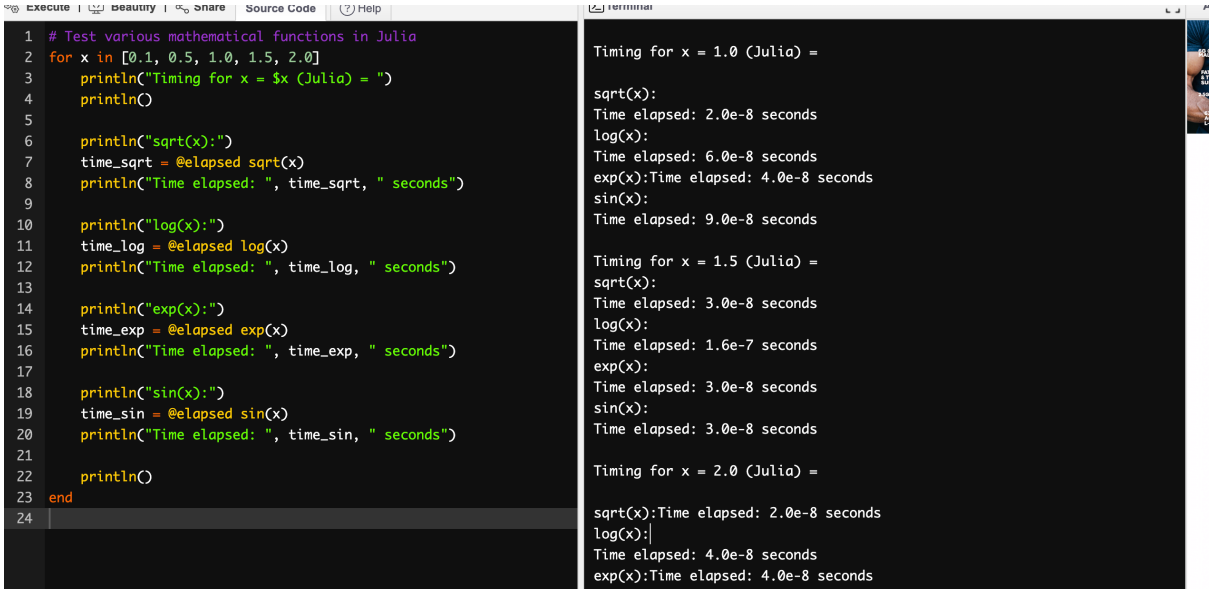
Python



```python
print("Testing various mathematical functions in Python:")
for x in [0.1, 0.5, 1.0, 1.5, 2.0]:
    print(f"\nTiming for x={x}:")

    print("\nsqrt(x):")
    start_time_sqrt = time.time()
    math.sqrt(x)
    end_time_sqrt = time.time()
    print(f"Time elapsed: {end_time_sqrt - start_time_sqrt} seconds")

    print("\nlog(x):")
    start_time_log = time.time()
    math.log(x)
    end_time_log = time.time()
    print(f"Time elapsed: {end_time_log - start_time_log} seconds")

    print("\nexp(x):")
    start_time_exp = time.time()
    math.exp(x)
    end_time_exp = time.time()
    print(f"Time elapsed: {end_time_exp - start_time_exp} seconds")

    print("\nsin(x):")
    start_time_sin = time.time()
    math.sin(x)
    end_time_sin = time.time()
    print(f"Time elapsed: {end_time_sin - start_time_sin} seconds")
```

```
Testing various mathematical functions in Python:

Timing for x=0.1:

sqrt(x):
Time elapsed: 2.6226043701171875e-06 seconds

log(x):
Time elapsed: 2.1457672119140625e-06 seconds

exp(x):
Time elapsed: 6.67572021484375e-06 seconds

sin(x):
Time elapsed: 0.0019583702087402344 seconds

Timing for x=0.5:

sqrt(x):
Time elapsed: 9.5367431640625e-07 seconds

log(x):
Time elapsed: 1.1920928955078125e-06 seconds

exp(x):
Time elapsed: 9.5367431640625e-07 seconds

sin(x):
```

Julia



```julia
# Test various mathematical functions in Julia
for x in [0.1, 0.5, 1.0, 1.5, 2.0]
    println("Timing for x = $x (Julia) = ")
    println()

    println("sqrt(x):")
    time_sqrt = @elapsed sqrt(x)
    println("Time elapsed: ", time_sqrt, " seconds")

    println("log(x):")
    time_log = @elapsed log(x)
    println("Time elapsed: ", time_log, " seconds")

    println("exp(x):")
    time_exp = @elapsed exp(x)
    println("Time elapsed: ", time_exp, " seconds")

    println("sin(x):")
    time_sin = @elapsed sin(x)
    println("Time elapsed: ", time_sin, " seconds")

    println()
end
```

```
Timing for x = 1.0 (Julia) =

sqrt(x):
Time elapsed: 2.0e-8 seconds
log(x):
Time elapsed: 6.0e-8 seconds
exp(x):Time elapsed: 4.0e-8 seconds
sin(x):
Time elapsed: 9.0e-8 seconds

Timing for x = 1.5 (Julia) =
sqrt(x):
Time elapsed: 3.0e-8 seconds
log(x):
Time elapsed: 1.6e-7 seconds
exp(x):
Time elapsed: 3.0e-8 seconds
sin(x):
Time elapsed: 3.0e-8 seconds

Timing for x = 2.0 (Julia) =

sqrt(x):Time elapsed: 2.0e-8 seconds
log(x):
Time elapsed: 4.0e-8 seconds
exp(x):Time elapsed: 4.0e-8 seconds
```

Comparision Table

---

x	Function	Julia Execution Time (seconds)	Python Execution Time (seconds)
0.1	sqrt	0.00000002	0.00000262
	log	0.00000006	0.00000215
	exp	0.00000004	0.00000668
	sin	0.00000009	0.00195840
0.5	sqrt	0.00000003	0.00000095
	log	0.00000016	0.00000119
	exp	0.00000003	0.00000095
	sin	0.00000003	0.00000095
1.0	sqrt	0.00000002	0.00000072
	log	0.00000004	0.00000072
	exp	0.00000004	0.00000072
	sin	0.00000009	0.00016093
1.5	sqrt	0.00000003	0.00000095
	log	0.00000016	0.00000119
	exp	0.00000003	0.00000095
	sin	0.00000003	0.00000095
2.0	sqrt	0.00000002	0.00000048
	log	0.00000004	0.00000095
	exp	0.00000004	0.00000095
	sin	0.00000009	0.00000072

Inference :

Julia's well-suited for numerical computing tasks, resulting in faster execution times for mathematical operations compared to Python. (in general !...unless there is some underlying disk performance issues)