ES215: Semester I [2024-2025]

Assignment 1

Name: Aditya Kumar Roll No : 22110015

Assumptions:

Compiler Consistency: The compiler is assumed to optimize and handle all code uniformly, meaning that performance differences are attributed to the efficiency of the algorithms rather than compiler behavior or optimizations.
No External Factors: The online compiler runs in a controlled, stable environment with no external influences (e.g., background processes, system load), ensuring that the benchmarks reflect the algorithm's performance without interference from external variables.
Same Hardware and Configuration: The code is executed on the same virtual hardware with consistent resources, removing hardware variability and ensuring that performance results are comparable across different methods.
Algorithm-Focused Comparison: The comparison is focused on the relative efficiency of the algorithms (recursion, iteration, memoization), assuming a consistent environment that highlights how each approach scales with increasing input sizes, free from external influences.

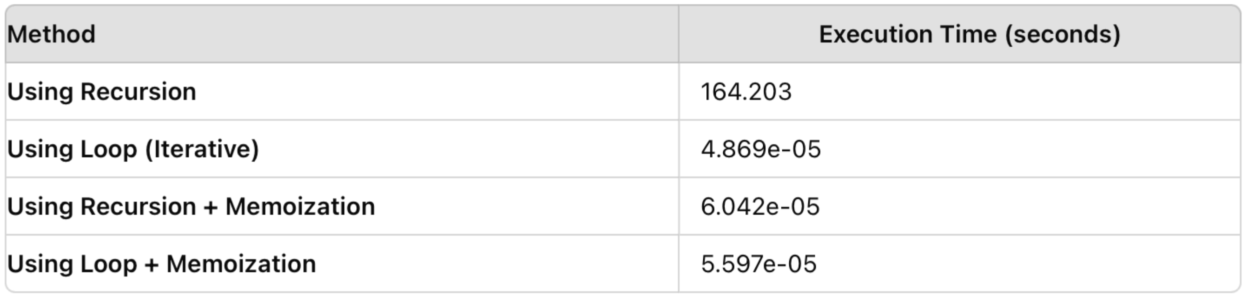
Q1:

Performance Comparison: Fibonacci Sequence Calculation

Fibonacci Sequence Output

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 4807526976 7778742049

Execution Time Comparison



Method	Execution Time (seconds)
Using Recursion	164.203
Using Loop (Iterative)	4.869e-05
Using Recursion + Memoization	6.042e-05
Using Loop + Memoization	5.597e-05

Time comparison of recursion with different Methods

	Method	Execution Time (seconds)	Speedup Compared to Recursive Method

1	Recursive	164.203	1x
2	Loop	4.869e-05	3,372,000x
3	Recursive with Memoization	6.042e-05	2,717,000x
4	Loop with Memoization	5.597e-05	2,933,000x`

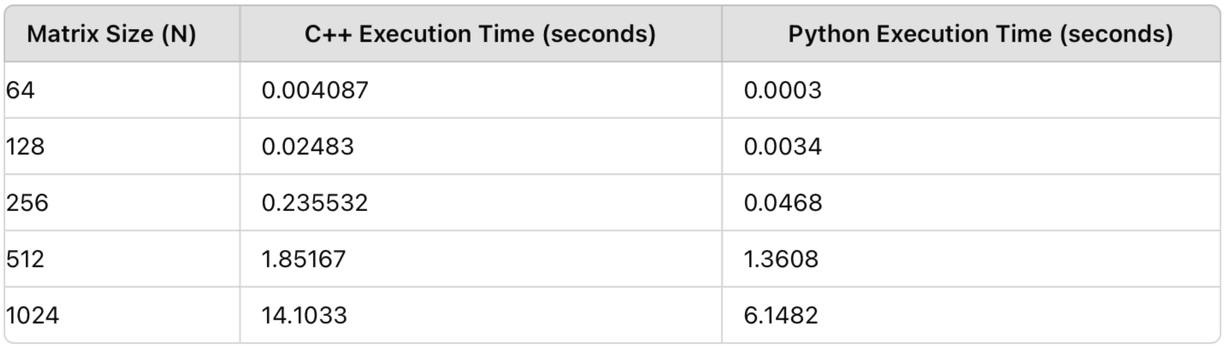
Summary of Fibonacci Calculation Methods:

1. **Recursion:** This is the slowest method, with an execution time of 164.203 seconds. The exponential growth in recursive calls makes it inefficient for larger Fibonacci numbers.
2. **Loop (Iterative):** The fastest method, with an execution time of 4.869e-05 seconds. This method uses a simple iteration, making it highly efficient.
3. **Recursion with Memoization:** Significantly faster than plain recursion, with an execution time of 6.042e-05 seconds. Memoization helps avoid redundant calculations by storing previously computed values.
4. **Loop with Memoization:** Slightly faster than recursion with memoization, with an execution time of 5.597e-05 seconds. This method combines the efficiency of iteration with the benefits of memoization.

Q2:

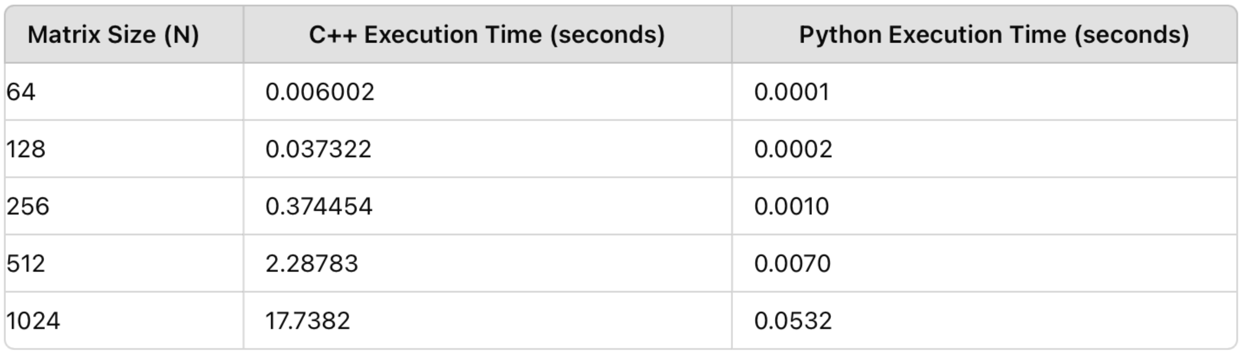
Performance Comparison: C++ vs Python

Matrix Multiplication Performance for Integer Values



Matrix Size (N)	C++ Execution Time (seconds)	Python Execution Time (seconds)
64	0.004087	0.0003
128	0.02483	0.0034
256	0.235532	0.0468
512	1.85167	1.3608
1024	14.1033	6.1482

Matrix Multiplication Performance for Double Values



Matrix Size (N)	C++ Execution Time (seconds)	Python Execution Time (seconds)
64	0.006002	0.0001
128	0.037322	0.0002
256	0.374454	0.0010
512	2.28783	0.0070
1024	17.7382	0.0532

Performance Comparison: C++ vs. Python Matrix Multiplication

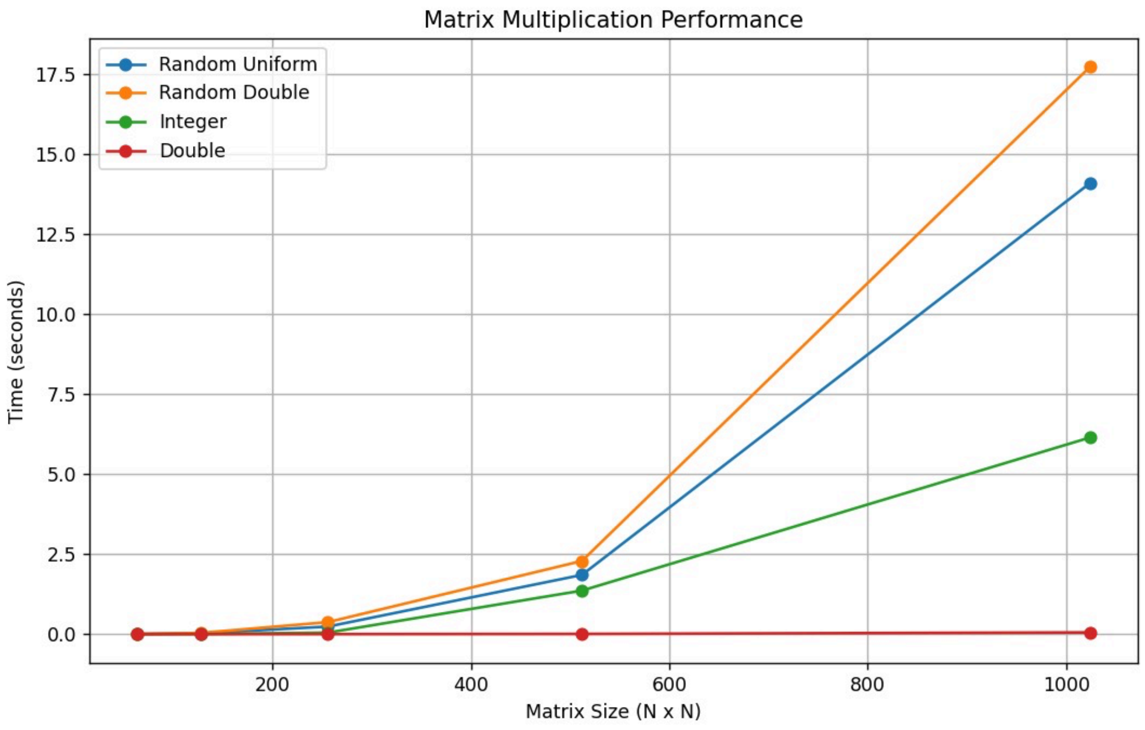
Observations

1. **Python vs. C++ Integer**
   - Python integers exhibit significantly faster performance compared to C++ integers across all matrix sizes.
2. **Python vs. C++ Double**
   - Python doubles also show a notable performance advantage over C++ doubles for all matrix sizes.
3. **C++ Integer vs. C++ Double**
   - In C++, operations involving double values consistently take more time compared to integer values.
4. **Python Integer vs. Python Double**
   - Conversely, in Python, double values consistently perform faster than integer values, which is contrary to the behavior observed in C++.

Conclusion

- Python's matrix multiplication, likely utilizing optimized libraries such as NumPy, demonstrates a significantly higher performance compared to C++'s Eigen library for the tested matrix sizes.
- The performance difference between integer and double values is notably greater in C++ than in Python.
- Python's efficiency in handling double values is particularly impressive, with much lower computation times compared to integer values. This could be attributed to highly optimized floating-point operations in Python's underlying libraries.

Graphical representation of Matrix Multiplication Performance


Matrix Multiplication Performance
Random Uniform
Random Double
Integer
Double
Time (seconds)
Matrix Size (N x N)