Assignment -1

Github link

Problem – 1

Assumptions

-The first 2 Fibonacci numbers are defined as 0 and 1.

-Let fib(n) be the nth Fibonacci number, then for n>=2, we have-

fib(n) = fib(n-1) + fib(n-2)

-long long datatype is used to store the Fibonacci numbers.

- The timespec structure is used for precise measurement of execution time. This structure provides nanosecond precision, which is necessary for accurate time measurement of the algorithms.

-CPU time is measured using CLOCK_PROCESS_CPU_TIME_ID

Resulting first 50 fibonacci numbers-

```
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
```

1. Using recursion

The recursion is based on the recurrence relation fib(n) = fib(n-1) + fib(n-2). This recursion is used to find the ith fibonacci number from i=1 to 50. The time complexity is O(2**k) where we need the find the kth fibonacci number. The overall will be O(2**(N+1)) where we need to find the first 50 fibonacci numbers.

CODE-

```
#include <iostream>
#include <vector>
#include <ctime>

using namespace std;

long long fib(int n){
    if (n<=1) return n;
    return fib(n-1)+fib(n-2);
}

int main(){
    struct timespec start,end;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&start);
    for (int i=0;i<50;i++){
        cout<<fib(i)<<endl;
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&end);
    long long sec_time=(end.tv_sec-start.tv_sec);
    long long nsec_time=(end.tv_nsec-start.tv_nsec)/1e9;
    double time_taken=sec_time+nsec_time;
    cout<<"Time taken: "<<time_taken<<" seconds";
    return 0;
}
```

Time taken to print the first 50 fibonacci numbers using recursion: 110 seconds

2. Using loop

In this approach two variables fib1 and fib2 are iteratively updated in accordance of the relation fib(n) = fib(n-1) + fib(n-2). This is done for i=0 to 50 to get the first 50 fibonacci numbers.

The time complexity O(N**2) because each of the loop takes O(k) where k is the fibonacci number we are trying to find.

So 1+2+3… N -> O(N**2)

CODE-

```
#include <iostream>
#include <vector>
#include <ctime>
using namespace std;


long long fib(int n){
    if (n==1){
        return 0;
    }
    if (n==2){
        return 1;
    }
    long long fib1=0;
    long long fib2=1;
    for (int i=3;i<=n;i++){
        long long temp=fib2;
        fib2=fib1+fib2;
        fib1=temp;
    }
    return fib2;
}
```

```
int main(){
    struct timespec start,end;

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&start);
    for (int i=0;i<50;i++){
        cout<<fib(i+1)<<endl;
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&end);
    long long sec_time=(end.tv_sec-start.tv_sec);
    long long nsec_time=(end.tv_nsec-start.tv_nsec);
    double time_taken=sec_time+nsec_time*1e-9;
    cout<<"Time taken: "<<time_taken<<" seconds";
    return 0;
}
```

Time taken to print the first 50 fibonacci numbers using loop: 6.6e-05 seconds

3. Using recursion with memoization

Here, the idea is to use an array to store a certain fibonacci number to avoid its recalculation and then use it whenever it already exists and is required. This brings down the time complexity to O(N) where we need to find the first N fibonacci numbers.

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<long long>first_50_fibonacci(50);
    struct timespec start,end;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&start);
    first_50_fibonacci[0]=0;
    first_50_fibonacci[1]=1;
    for (int i=2;i<50;i++){
        first_50_fibonacci[i]=first_50_fibonacci[i-1]+first_50_fibonacci[i-2];
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&end);
    double time_taken=(end.tv_sec-start.tv_sec)+(end.tv_nsec-start.tv_nsec)/1e9;
    cout<<"Time taken: "<<time_taken<<" seconds";
    return 0;
}
```

Time taken to print the first 50 fibonacci numbers using recursion with memoization: 8e-06 seconds

4. Using loop with memoization

Here, we build the fibonacci numbers in a bottom up approach.

The first 2 are used to find the 3rd. The 2nd and 3rd are used to find the 4th and so on till 50th.

The time complexity is O(N) where we need to find the first N fibonacci numbers.

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<long long>first_50_fibonacci(50);
    struct timespec start,end;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&start);
    first_50_fibonacci[0]=0;
    first_50_fibonacci[1]=1;
    for (int i=2;i<50;i++){
        first_50_fibonacci[i]=first_50_fibonacci[i-1]+first_50_fibonacci[i-2];
    }
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&end);
    double time_taken=(end.tv_sec-start.tv_sec)+(end.tv_nsec-start.tv_nsec)/1e9;
    cout<<"Time taken: "<<time_taken<<" seconds";
    return 0;
}
```

Time taken to print the first 50 fibonacci numbers using loop with memoization: 2e-06 seconds

SpeedUp

Speedup = (Execution time of baseline algorithm) / (Execution time of improved algorithm)

In mathematical notation, this can be written as:

S = T_baseline / T_improved

Calculating the speedup for each of the case-

- The loop approach is 1,666,666.66 times faster than the recursion approach i.e. speedup = 1,666,666.66
- The recursion + memoization approach is 13,750,000 times faster than the recursion approach i.e. speedup = 13,750,000
- The loop +memoization approach is 55,000,000 times faster than the recursion approach i.e. speedup = 55,000,000

Discussion

1. The recursive implementation is the slowest due to repeated calculations.
2. The loop implementation shows significant improvement by avoiding redundant computations.
3. Recursion with memoization drastically reduces execution time by storing previously calculated values.
4. The loop with memoization is the fastest, combining the benefits of iteration and cached results.

The speedup calculations demonstrate the efficiency gains from using loops and memoization techniques.

Problem – 2

Assumptions and Methodology

- **Matrix Sizes:** The matrix sizes tested were N=64,128,256,512,1024.
- **Data Types:** Two data types were used in the multiplication:
    - **Integer**
    - **Double**
- **Timing Measurements:**
    - **System Time**: Measured using CLOCK_MONOTONIC in C++ and time.time() in Python, representing the total wall-clock time for the execution.
    - **CPU Time**: Measured using CLOCK_PROCESS_CPUTIME_ID in C++ and time.process_time() in Python, representing the time during which the CPU was actively executing the program.
    - **Matrix Multiplication Time**: Specifically measured for the multiplication process, indicating the efficiency of the core algorithm.
- **Loop-based Implementation:** For both C++ and Python, a simple three-loop approach was used to implement matrix multiplication, ensuring that the core algorithm was the focus of the performance evaluation.

A_part-

C++ CODE-

```
#include <iostream>
#include <vector>
#include <ctime>
#include <iomanip>

using namespace std;

void multiplyMatrices(const vector<vector<int>>& A, const vector<vector<int>>& B, vector<vector<int>>& C, int
N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void multiplyMatrices(const vector<vector<double>>& A, const vector<vector<double>>& B,
vector<vector<double>>& C, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0.0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void calculateAndPrintTime(const struct timespec& start, const struct timespec& end, const string& label) {
    double time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    cout << label << fixed << time_taken << setprecision(9) << " seconds" << endl;
}

int main() {
```

```cpp
int sizes[] = {64, 128, 256, 512, 1024};

for (int N : sizes) {
    struct timespec sys_start, sys_end, cpu_start, cpu_end;

    clock_gettime(CLOCK_MONOTONIC, &sys_start);
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &cpu_start);

    vector<vector<int>> A(N, vector<int>(N, 1));
    vector<vector<int>> B(N, vector<int>(N, 1));
    vector<vector<int>> C(N, vector<int>(N, 0));


    struct timespec mult_start, mult_end;
    clock_gettime(CLOCK_MONOTONIC, &mult_start);

    multiplyMatrices(A, B, C, N);

    clock_gettime(CLOCK_MONOTONIC, &mult_end);

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &cpu_end);
    clock_gettime(CLOCK_MONOTONIC, &sys_end);

    cout << "N=" << N << ", Integer Matrix Multiplication: " << endl;
    calculateAndPrintTime(sys_start, sys_end, "Total System Time: ");
    calculateAndPrintTime(cpu_start, cpu_end, "Total CPU Time: ");
    calculateAndPrintTime(mult_start, mult_end, "Matrix Multiplication Time: ");

    clock_gettime(CLOCK_MONOTONIC, &sys_start);
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &cpu_start);

    vector<vector<double>> A_d(N, vector<double>(N, 1.0));
    vector<vector<double>> B_d(N, vector<double>(N, 1.0));
    vector<vector<double>> C_d(N, vector<double>(N, 0.0));



    clock_gettime(CLOCK_MONOTONIC, &mult_start);

    multiplyMatrices(A_d, B_d, C_d, N);
```

```
        clock_gettime(CLOCK_MONOTONIC, &mult_end);

        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &cpu_end);
        clock_gettime(CLOCK_MONOTONIC, &sys_end);

        cout << "N=" << N << ", Double Matrix Multiplication: " << endl;
        calculateAndPrintTime(sys_start, sys_end, "Total System Time: ");
        calculateAndPrintTime(cpu_start, cpu_end, "Total CPU Time: ");
        calculateAndPrintTime(mult_start, mult_end, "Matrix Multiplication Time: ");
    }

    return 0;
}
```

PYTHON CODE-

```
import time

def multiply_matrices_int(A, B, N):
    C = [[0] * N for _ in range(N)]
    for i in range(N):
        for j in range(N):
            for k in range(N):
                C[i][j] += A[i][k] * B[k][j]
    return C

def multiply_matrices_double(A, B, N):
    C = [[0.0] * N for _ in range(N)]
    for i in range(N):
        for j in range(N):
            for k in range(N):
                C[i][j] += A[i][k] * B[k][j]
    return C

sizes = [64, 128, 256, 512, 1024]

for N in sizes:
    sys_start = time.time()
```

```
cpu_start = time.process_time()
A = [[1] * N for _ in range(N)]
B = [[1] * N for _ in range(N)]



mult_start = time.time()

C = multiply_matrices_int(A, B, N)

mult_end = time.time()

cpu_end = time.process_time()
sys_end = time.time()

total_sys_time = sys_end - sys_start
total_cpu_time = cpu_end - cpu_start
mult_time = mult_end - mult_start

print(f"N={N}, Integer Matrix Multiplication:")
print(f"Total System Time: {total_sys_time:.9f} seconds")
print(f"Total CPU Time: {total_cpu_time:.9f} seconds")
print(f"Matrix Multiplication Time: {mult_time:.9f} seconds")

sys_start = time.time()
cpu_start = time.process_time()

A_d = [[1.0] * N for _ in range(N)]
B_d = [[1.0] * N for _ in range(N)]



mult_start = time.time()

C_d = multiply_matrices_double(A_d, B_d, N)

mult_end = time.time()

cpu_end = time.process_time()
sys_end = time.time()
```

```
total_sys_time = sys_end - sys_start
total_cpu_time = cpu_end - cpu_start
mult_time = mult_end - mult_start

print(f"N={N}, Double Matrix Multiplication:")
print(f"Total System Time: {total_sys_time:.9f} seconds")
print(f"Total CPU Time: {total_cpu_time:.9f} seconds")
print(f"Matrix Multiplication Time: {mult_time:.9f} seconds")
```

Results for C++

	N	Data Type	Total System Time (s)	Total CPU Time (s)	Matrix Multiplication Time (s)
0	64	Integer	0.003494	0.003446	0.003385
1	64	Double	0.003166	0.003163	0.003059
2	128	Integer	0.021381	0.020997	0.021207
3	128	Double	0.016978	0.016825	0.016794
4	256	Integer	0.109888	0.108837	0.109612
5	256	Double	0.102274	0.100764	0.101964
6	512	Integer	0.847472	0.841313	0.846977
7	512	Double	0.813432	0.810980	0.812541
8	1024	Integer	6.509373	6.450987	6.507898
9	1024	Double	7.938114	7.904500	7.935701

Results for Python

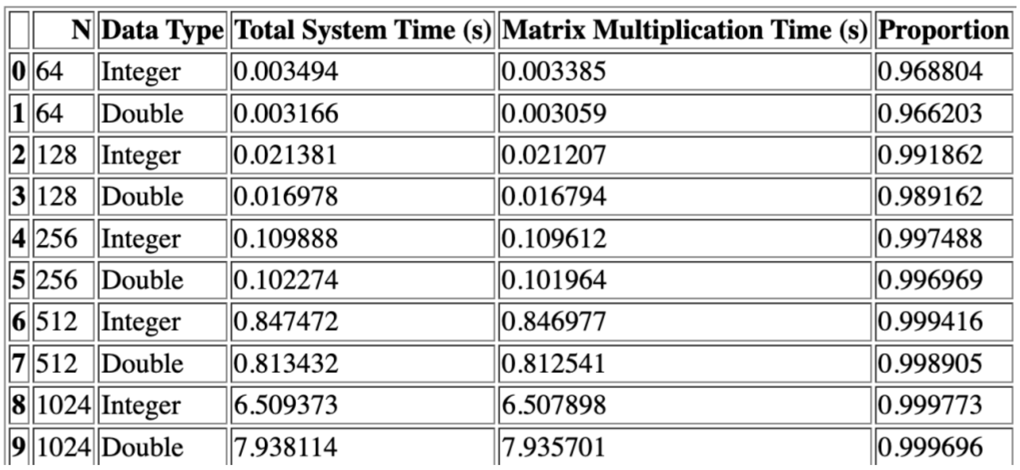
	N	Data Type	Total System Time (s)	Total CPU Time (s)	Matrix Multiplication Time (s)
0	64	Integer	0.025699	0.027030	0.025374
1	64	Double	0.022015	0.021971	0.021914
2	128	Integer	0.106392	0.106205	0.106275
3	128	Double	0.093269	0.092897	0.092976
4	256	Integer	0.806635	0.804168	0.806427
5	256	Double	0.794458	0.791945	0.792944
6	512	Integer	6.971561	6.955179	6.945327
7	512	Double	6.479693	6.459194	6.476284
8	1024	Integer	64.494542	64.005899	64.488086
9	1024	Double	59.009520	58.742013	58.999472

B_part-

Here “meat time” is the matrix multiplication time

Proportion = matrix multiplication time/ total system time

Proportion data for cpp



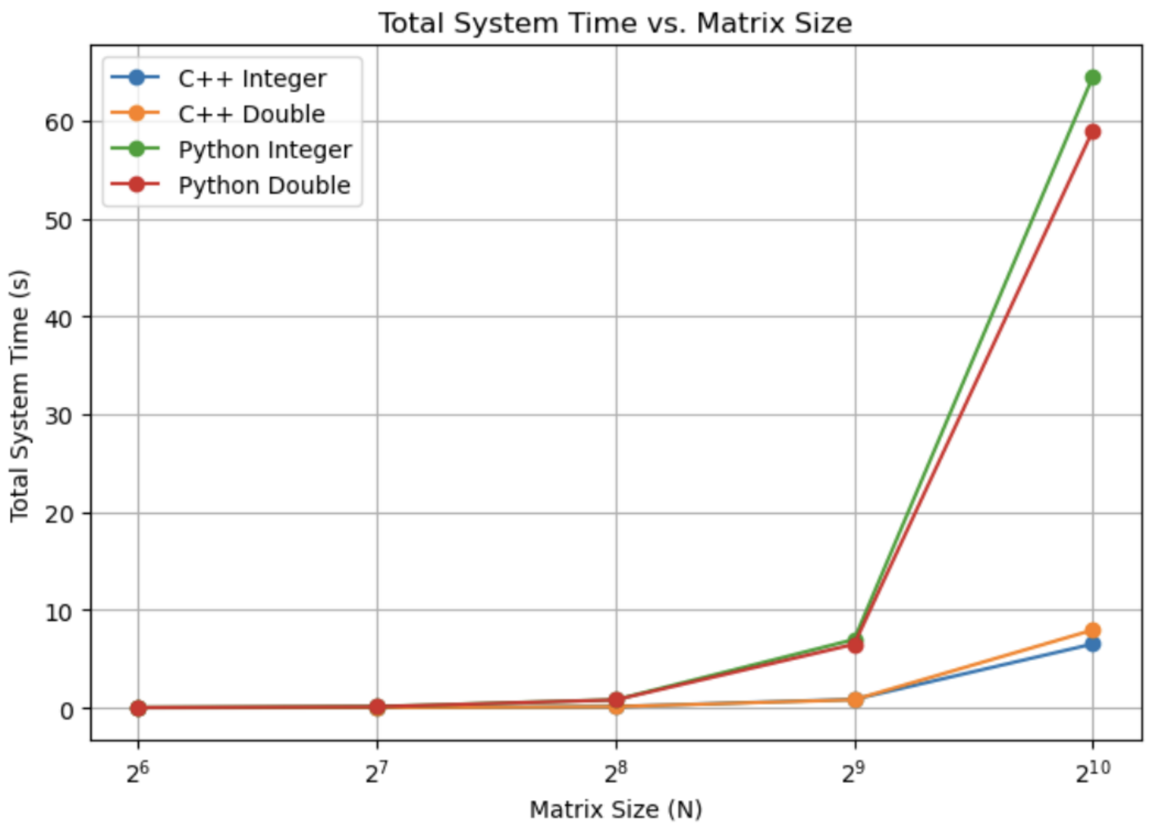
	N	Data Type	Total System Time (s)	Matrix Multiplication Time (s)	Proportion
0	64	Integer	0.003494	0.003385	0.968804
1	64	Double	0.003166	0.003059	0.966203
2	128	Integer	0.021381	0.021207	0.991862
3	128	Double	0.016978	0.016794	0.989162
4	256	Integer	0.109888	0.109612	0.997488
5	256	Double	0.102274	0.101964	0.996969
6	512	Integer	0.847472	0.846977	0.999416
7	512	Double	0.813432	0.812541	0.998905
8	1024	Integer	6.509373	6.507898	0.999773
9	1024	Double	7.938114	7.935701	0.999696

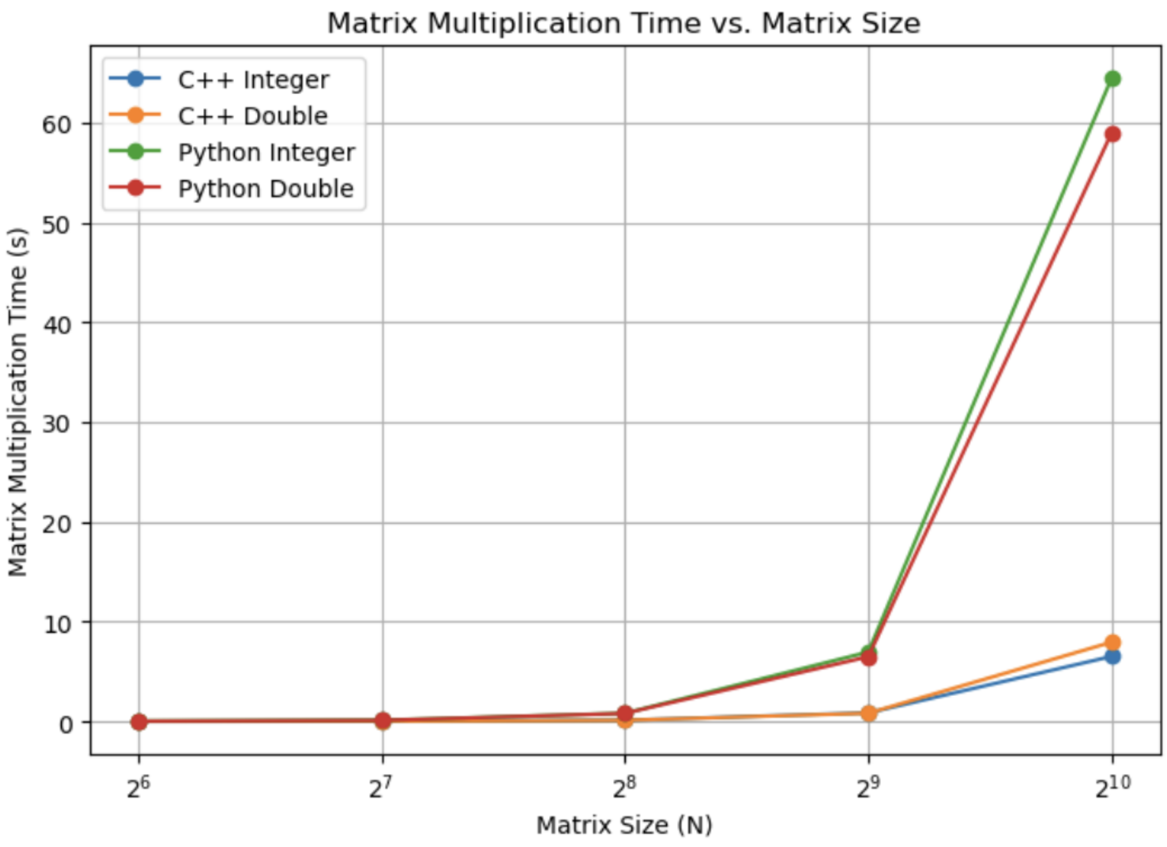
Proportion data for python

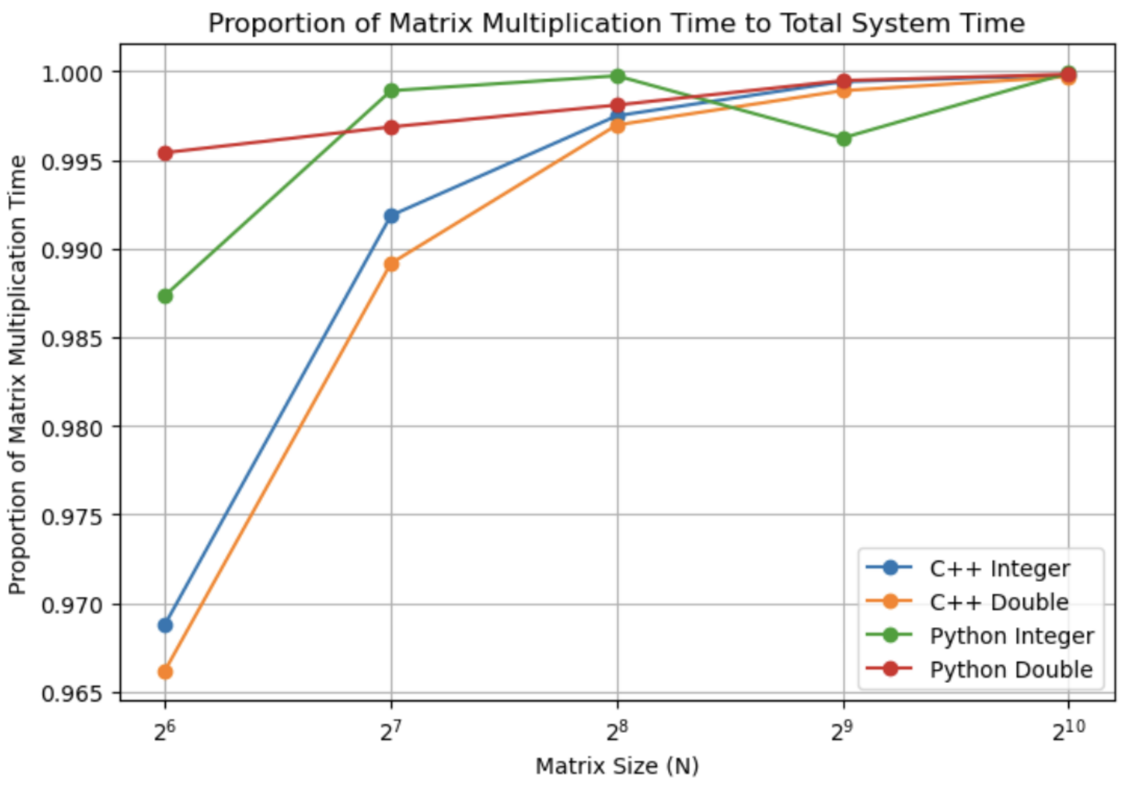
	N	Data Type	Total System Time (s)	Matrix Multiplication Time (s)	Proportion
0	64	Integer	0.025699	0.025374	0.987355
1	64	Double	0.022015	0.021914	0.995408
2	128	Integer	0.106392	0.106275	0.998902
3	128	Double	0.093269	0.092976	0.996861
4	256	Integer	0.806635	0.806427	0.999743
5	256	Double	0.794458	0.792944	0.998094
6	512	Integer	6.971561	6.945327	0.996237
7	512	Double	6.479693	6.476284	0.999474
8	1024	Integer	64.494542	64.488086	0.999900
9	1024	Double	59.009520	58.999472	0.999830

C_part-

GRAPHICAL ANALYSIS-




Matrix Multiplication Time vs. Matrix Size
C++ Integer
C++ Double
Python Integer
Python Double
Matrix Multiplication Time (s)
Matrix Size (N)
0
10
20
30
40
50
60
2^6
2^7
2^8
2^9
2^{10}


Proportion of Matrix Multiplication Time to Total System Time
Proportion of Matrix Multiplication Time
Matrix Size (N)
1.000
0.995
0.990
0.985
0.980
0.975
0.970
0.965
2^6
2^7
2^8
2^9
2^{10}
C++ Integer
C++ Double
Python Integer
Python Double

OBSERVATIONS

- **C++ Performance**: Typically shows lower system and multiplication times compared to Python due to its compiled nature and efficient memory management.
- **Python Performance**: Slower execution times are expected, especially for larger matrices, due to the interpreted nature of Python and additional overhead.
- **Proportions**: For both languages, the proportion of time spent on matrix multiplication relative to the total system time is high, indicating that the matrix multiplication operation dominates the execution time.