INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

EEN-300 INDUSTRY ORIENTED PROBLEM

MID Term Evaluation – WriteUp

C++ Implementation of various Matrix Operations as a Template Library

Guided by:
Prof. Dheeraj K Khatod, Deptartment of Electrical Engineering

Prepared by:
Aayush A Chaturvedi 16115002
Adarsh Kumar 16115009
Aviroop Pal 16115033
Samarth Joshi 161151099

ACCOMPLISHMENTS

Three algorithms for matrix multiplication and three algorithms for matrix inversion have been completely implemented and their run times have been analysed.
All algorithms have been written in C++, and are standalone, meaning that they can be ran individually.

1. MATRIX MULTIPLICATION

- Brute Force Multiplication.
- Strassen's Multiplication.
- Matrix Exponentiation (positive powers)
- Algorithm for positive fractional powers.

Brute Force Multiplication

The brute force multiplication of a matrix is the algorithm that follows the definition of matrix multiplication.

Following is the implementation of the brute force approach.

```
#include <bits/stdc++.h>

using namespace std;

#define ll long long
#define N 10004

ll a[N][N], b[N][N], c[N][N] = {0};

int main(){
     int n, m1, m2, m, p;
     cout<<"Enter dimensions\n";
     cin>>n>>m1>>m2>>p;
     if(m1 != m2){
          cout<<"Matrix multiplication not possible!\n";
          return 0;
     }
     m = m1;
     cout<<"Enter Matrices in order, row first\n";
     int i, j, k;
     for(i=1; i<=n; i++){
          for(j=1; j<=m; j++){
```

```
                cin>>a[i][j];
            }
        }
        for(i=1; i<=m; i++){
            for(j=1; j<=p; j++){
                cin>>b[i][j];
            }
        }
        for(i=1; i<=n; i++){
            for(j=1; j<=m; j++){
                for(k=1; k<=p; k++){
                    c[i][k] = c[i][k] + a[i][j]*b[j][k];
                }
            }
        }
        cout<<"Multiplication result\n";
        for(i=1; i<=n; i++){
            for(j=1; j<=p; j++){
                cout<<c[i][j]<<" ";
            }
            cout<<"\n";
        }
        return 0;
}
```

The implementation contains taking input from user, and using the following code, run times can be clocked.

```
int seeds[] = {41661, 82263, 31243, 70828, 46412, 25622, 67921,
74947, 78627, 84862, 66061, 70145, 95453, 43355, 37710, 99532,
59418, 11642, 36212, 82328}; // random seeds for rand() function.

clock_t clk;
srand(seeds[t-1]);

clk = clock();
// operation
clk = clock() - clk;
cout<<fixed<<setprecision(6)<<n<<""<<((double)clk)/CLOCKS_PER_SEC<
<"\n";
```

Strassen's Algorithm for Matrix Multiplication

We now turn toward Strassen's algorithm, such that we will be able to reduce the number of sub-calls to matrix-multiplies to 7, using just a bit of algebra.
In this way, we bring the work down to $\tilde{O}(n^{log7})$.
How do we do this? We use the following factoring scheme.
We write down *Cij* 's in terms of block matrices *Mk*'s. Each *Mk* may be calculated simply from products and sums of sub-blocks of *A* and *B*.
That is, we let

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})\,(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

And thus, we calculate C as

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M3 + M_6
\end{aligned}
$$

The implementation of the algorithm is given below.
(PTO)

```
#include <bits/stdc++.h>

using namespace std;

int seeds[] = {41661, 82263, 31243, 70828, 46412, 25622, 67921,
74947, 78627, 84862, 66061, 70145, 95453, 43355, 37710, 99532,
59418, 11642, 36212, 82328};

vector < vector <int> > add(vector < vector <int> > x, vector <
vector <int> > y, int s, int f){
     vector < vector <int> > ans(f-s+1);
     for(int i=s; i<=f; i++){
          for(int j=s; j<=f; j++){
               ans[i-s].push_back(x[i][j] + y[i][j]);
          }
     }
     return ans;
}

vector < vector <int> > sub(vector < vector <int> > x, vector <
vector <int> > y, int s, int f){
     vector < vector <int> > ans(f-s+1);
     for(int i=s; i<=f; i++){
          for(int j=s; j<=f; j++){
               ans[i-s].push_back(x[i][j] - y[i][j]);
          }
     }
     return ans;
}

vector < vector <int> > strassens_multiply(vector < vector <int> >
x, vector < vector <int> > y, int s, int f){
     if(s + 1 == f){
          vector < vector <int> > c(2);
          c[0].push_back(x[s][s]*y[s][s] + x[s][f]*y[f][s]);
          c[0].push_back(x[s][s]*y[s][f] + x[s][f]*y[f][f]);
          c[1].push_back(x[f][s]*y[s][s] + x[f][f]*y[f][s]);
          c[1].push_back(x[f][s]*y[s][f] + x[f][f]*y[f][f]);
          return c;
     }
     int sz = (s + f)>>1, i, j;
     vector < vector <int> > a11(sz+1), a12(sz+1), a21(sz+1),
a22(sz+1), b11(sz+1), b12(sz+1), b21(sz+1), b22(sz+1);
     for(i=s; i<=sz; i++){
          for(j=s; j<=sz; j++){
               a11[i-s].push_back(x[i][j]);
               b11[i-s].push_back(y[i][j]);
          }
     }
```

```
     for(i=s; i<=sz; i++){
          for(j=sz+1; j<=f; j++){
               a12[i-s].push_back(x[i][j]);
               b12[i-s].push_back(y[i][j]);
          }
     }
     for(i=sz+1; i<=f; i++){
          for(j=s; j<=sz; j++){
               a21[i-sz-1].push_back(x[i][j]);
               b21[i-sz-1].push_back(y[i][j]);
          }
     }
     for(i=sz+1; i<=f; i++){
          for(j=sz+1; j<=f; j++){
               a22[i-sz-1].push_back(x[i][j]);
               b22[i-sz-1].push_back(y[i][j]);
          }
     }
     vector < vector <int> > m1(sz+1), m2(sz+1), m3(sz+1),
m4(sz+1), m5(sz+1), m6(sz+1), m7(sz+1);
     m1 = strassens_multiply(add(a11, a22, 0, sz), add(b11, b22, 0,
sz), 0, sz);
     m2 = strassens_multiply(add(a21, a22, 0, sz), b11, 0, sz);
     m3 = strassens_multiply(a11, sub(b12, b22, 0, sz), 0, sz);
     m4 = strassens_multiply(a22, sub(b21, b11, 0, sz), 0, sz);
     m5 = strassens_multiply(add(a11, a12, 0, sz), b22, 0, sz);
     m6 = strassens_multiply(sub(a21, a11, 0, sz), add(b11, b12, 0,
sz), 0, sz);
     m7 = strassens_multiply(sub(a12, a22, 0, sz), add(b21, b22, 0,
sz), 0, sz);
     vector < vector <int> > c11(sz+1), c12(sz+1), c21(sz+1),
c22(sz+1);
     c11 = add(m1, add(m4, sub(m7, m5, 0, sz), 0, sz), 0, sz);
     c12 = add(m3, m5, 0, sz);
     c21 = add(m2, m4, 0, sz);
     c22 = add(m1, add(m3, sub(m6, m2, 0, sz), 0, sz), 0, sz);
     vector < vector <int> > c(f - s + 1);
     for(i=s; i<=sz; i++){
          for(j=s; j<=sz; j++){
               c[i-s].push_back(c11[i-s][j-s]);
          }
     }
     for(i=s; i<=sz; i++){
          for(j=sz+1; j<=f; j++){
               c[i-s].push_back(c12[i-s][j-sz-1]);
          }
     }
     for(i=sz+1; i<=f; i++){
          for(j=s; j<=sz; j++){
               c[i-sz+((f-s)>>1)].push_back(c21[i-sz-1][j-s]);
```

```
            }
        }
        for(i=sz+1; i<=f; i++){
            for(j=sz+1; j<=f; j++){
                c[i-sz+((f-s)>>1)].push_back(c22[i-sz-1][j-sz-1]);
            }
        }
        return c;
}

int main(){
        freopen("Strassen's2_times.txt", "w", stdout);
        clock_t clk;
        int n, i, j;
        for(int t=1; t<=9; t++){
            n = (1<<t);
            srand(seeds[t-1]);
            vector < vector <int> > a(n), b(n);
            for(i=0; i<n; i++){
                for(j=0; j<n; j++){
                    a[i].push_back(rand()%1000);
                    b[i].push_back(rand()%1000);
                }
            }
            clk = clock();
            vector < vector <int> > c = strassens_multiply(a, b, 0,
n-1);
            clk = clock() - clk;
            cout<<fixed<<setprecision(6)<<n<<"
"<<((double)clk)/CLOCKS_PER_SEC<<"\n";
        }
        return 0;
}
```

The runtime analysis is done using the in-code analysis as written.

Matrix Exponentiation Algorithm for finding poitive integral powers of a matrix

Suppose we have a matrix A of order N, and we want to find the matrix A^P, where P is a non-negative integer number.

Naïve method:
Initialise answer matrix as the Identity matrix. Now, we multiply A to it P times. The method takes N^3 time for multiplication and we multiply P times. Thus, the overall time complexity of the algorithm is $O(N^3P)$.

Logarithmic exponentiation
As the name suggests the method does matrix multiplication for order of $log_2 P$ times instead of the linear time in naïve method. The working can be explained as follows:

If P is even:

$$A^P = (A^{\frac{P}{2}})^2 = (A^2)^{\frac{P}{2}}$$

If P is odd:

$$A^P = (A)(A^{P-1})$$

Here, in every two iterations the power to be raised is reduced to at least half of the initial power. Thus, the overall time complexity of the algorithm is $O(N^3 log_2 P)$.

The C++ code:

```
#include <bits/stdc++.h>
using namespace std;

#define ld long double

const int mod = 1e9 + 7;

int sz;
const int NN = 101;

class matrix{
public:
    ld mat[NN][NN];
    matrix(){
        for(int i = 0; i < NN; i++)
            for(int j = 0; j < NN; j++)
                mat[i][j] = 0;
    }
    inline matrix operator * (const matrix &a){
        matrix temp;
        for(int i = 0; i < sz; i++)
            for(int j = 0; j < sz; j++){
                for(int k = 0; k < sz; k++){
                temp.mat[i][j] += (mat[i][k] * a.mat[k][j]);
```

```
            }
        }
        return temp;
    }
    inline matrix operator + (const matrix &a){
        matrix temp;
        for(int i = 0; i < sz; i++)
            for(int j = 0; j < sz; j++)
                temp.mat[i][j] = mat[i][j] + a.mat[i][j];
        return temp;
    }
    inline matrix operator - (const matrix &a){
        matrix temp;
        for(int i = 0; i < sz; i++)
            for(int j = 0; j < sz; j++)
                temp.mat[i][j] = mat[i][j] - a.mat[i][j];
        return temp;
    }
    inline void operator = (const matrix &b){
        for(int i = 0; i < sz; i++)
            for(int j = 0; j < sz; j++)
                mat[i][j] = b.mat[i][j];
    }
    inline void print(){
        for(int i = 0; i < sz; i++){
            for(int j = 0; j < sz; j++){
                cout << setprecision(15) << mat[i][j] << " ";
            }
            cout << endl;
        }
    }
};

matrix pow(matrix a, int k){
    matrix ans;
    for(int i = 0; i < sz; i++)
        ans.mat[i][i] = 1;
    while(k){
        if(k & 1)
            ans = ans * a;
        a = a * a;
        k >>= 1;
    }
    return ans;
}

signed main() {
     matrix a;
     int p;
     cout << "Enter size of matrix:\n";
```

```
	cin >> sz;
	cout << "Enter elements of matrix:\n";
	for(int i = 0; i < sz; i++)
		for(int j = 0; j < sz; j++)
			cin >> a.mat[i][j];
	cout << "Enter power to be raised:\n";
	cin >> p;
	a = pow(a, p);
	cout << "Resultant matrix:\n";
	a.print();
	return 0;
}
```

The run time analysis is done later.

Algorithm for finding poitive fractional power of a matrix

Diagonalization:
In linear algebra, a square matrix *A* is called diagonalizable or nondefective if it is similar to a diagonal matrix, i.e., if there exists an invertible matrix *P* such that $P^{-1}AP$ is a diagonal matrix. Diagonalization is the process of finding a corresponding diagonal matrix for a diagonalizable matrix. Diagonalizable matrices and maps are of interest because diagonal matrices are especially easy to handle; once their eigenvalues and eigenvectors are known, one can raise a diagonal matrix to a power by simply raising the diagonal entries to that same power, and the determinant of a diagonal matrix is simply the product of all diagonal entries.

For a matrix *A* which is a *N***N* matrix, to be diagonalizable, it should have *n* distinct eigenvalues.

These eigenvalues are the values that will appear in the diagonalized form of matrix *A*, so by finding the eigenvalues of *A* we have diagonalized it. The matrix formed with column as the eigenvectors of the given matrix is the invertible matrix *P*.
Diagonalization can be used to compute the not only integral power but also fractional power.

We have,

$$A = PDP^{-1}$$

Let,

$$d^m = D$$

Where d can be computed by taking the *mth* root of each of the diagonal elements of D.

Then,

$$A = Pd^mP^{-1}$$

Thus,

$$A = \underbrace{(PdP^{-1})(PdP^{-1}).....(PdP^{-1})}_{m\ times}$$

Let,

$$a = PdP^{-1}$$

Then,

$$a^m = A$$

And thus a is the *mth* root of matrix A.

Matrix Inversion using LU Decomposition

Working of LU Decomposition can be explained in the following way.

Consider the system of linear equations $Ax = B$. If we are able to write the A matrix as a product of lower triangular matrix and an upper triangular matrix, then we may do forward substitution and back substitution to get the solution. Also, forward substitution and back substitution takes $O(n^2)$ to compute, where n is the size of the matrix. So if given a set of linear equations, $Ax = B1$, $Ax = B2$, ... $Ax = Bk$, then we can first do the LU decomposition of the matrix A in $O(n^3)$, then solve for each of the linear equation systems in $O(n^2)$. So the total complexity of would be $O(n^3 + k*n^2)$.

Furthermore, traditional method of computation for inverse of matrix A is very costly in terms of number of operation.

Types of LU Decomposition:

1. Doolittle Decomposition

The diagonal entries of L are taken to be one.
For eg:

$$A = \begin{bmatrix} A11 & A12 & A13 \\ A21 & A22 & A23 \\ A31 & A32 & A33 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ L21 & 1 & 0 \\ L31 & L32 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} U11 & U12 & U13 \\ 0 & U22 & U23 \\ 0 & 0 & U33 \end{bmatrix}$$

Multiplying L and U, we get

$$\begin{bmatrix} U11 & U12 & U13 \\ L21 * U11 & L21 * U12 + U22 & L21 * U13 + U23 \\ L31 * U11 & L31 * U12 + L32 * U22 & L31 * U13 + L32 * U23 + U33 \end{bmatrix}$$

Comparing this matrix with matrix A, the general expression for solution of each value of L and U,

$$L(i,k) = (A(i,k) - \sum_{j=0}^{i} L(i,j) * U(j,k)) / U(k,k)$$

and,

$$U(i,k) = A(i,k) - \sum_{j=0}^{i} L(i,j) * U(j,k)$$

.

2. Crout Decomposition

In this type of decomposition, we assume that the diagonal entries of U matrix is 1. The computation of L and U is similiar to Doolittle Decomposition.

3. Cholesky Decompostion

In this type of decomposition, we have $U = L^T$. For cholesky decomposition, we must have A as a symmetric matrix and positive definite.

Using these, we can compute the inverse of the matrix.
Consider the equation $Ax = B$. The equation $Ax = I$ would give inverse of A as the solution. After replacing $A = LU$, we get $LUx = I$. Now we can treat Ux as a separate matrix Y. Then we get $LY = I$. Y can be calculated in $O(n^2)$ using forward substitution. After solving for Y, we can equate it to $Ux = Y$ and solve for x using back substitution.

The C++ implemenatation of the above methods is discussed below.

```
// Performs LU decomposition - Dolittle, Crout, Cholesky and
computes inverse of a matrix using these decompositions.
#include<bits/stdc++.h>
using namespace std;
typedef long double f80;
void print(vector<vector<f80>> A) {
    int n = A.size();
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            cout << A[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}
vector<vector<f80>> operator *(vector<vector<f80>> A,
vector<vector<f80>> B){
    int n = A.size();
    vector<vector<f80>> temp(n);
    for(int i = 0; i < n; i++) {
        temp[i].resize(n, 0);
    }
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
```

```
            for(int k = 0; k < n; k++) {
                temp[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return temp;
}
class Matrix{
public:
    vector<vector<f80>> A, L, U, IA;
    bool decomposed;
    int n;
    Matrix(int n) : n(n) {
        decomposed = 0;
        A.resize(n);
        for(int i = 0; i < n; i++){
            A[i].resize(n, 0);
        }
    }
    void reset(){
        L.clear();
        U.clear();
        L.resize(n);
        U.resize(n);
        for(int i = 0; i < n; i++){
            L[i].resize(n, 0);
            U[i].resize(n, 0);
        }
    }
    void LU_Crout() { // U[i][j] = 1 for 0 <= i < n
        reset();
        decomposed = 1;
       for (int j = 0; j < n; j++) {
        U[j][j] = 1;
            for (int i = j; i < n; i++) {
                L[i][j] = A[i][j];
                for (int k = 0; k < j; k++) {
                    L[i][j] -= L[i][k] * U[k][j];
                }
            }
            for (int i = j; i < n; i++) {
                U[j][i] = A[j][i];
                for(int k = 0; k < j; k++) {
                    U[j][i] -= L[j][k] * U[k][i];
                }
                U[j][i] /= L[j][j];
            }
        }
    }
    void LU_Dolittle() { // L[i][i] = 1 for 0 <= i < n
```

```
        reset();
        decomposed = 1;
        for(int i = 0; i < n; i++) {
            L[i][i] = 1;
            for(int j = i; j < n; j++) {
                U[i][j] = A[i][j];
                for(int k = 0; k < i; k++) {
                    U[i][j] -= L[i][k] * U[k][j];
                }
            }
            for(int j = i + 1; j < n; j++) {
                L[j][i] = A[j][i];
                for(int k = 0; k < i; k++) {
                    L[j][i] -= L[j][k] * U[k][i];
                }
                L[j][i] /= U[i][i];
            }
        }
    }
    void LU_Cholesky() { // U = L^T, possible only if A is
symmetric
        reset();
        decomposed = 1;
        for(int i = 0; i < n; i++) {
            L[i][i] = A[i][i];
            for(int j = 0; j < i; j++) {
                L[i][i] -= L[i][j] * L[i][j];
            }
            assert(L[i][i] >= 0);
            L[i][i] = sqrtl(L[i][i]);
            for(int j = i + 1; j < n; j++) {
                L[j][i] = A[j][i];
                for(int k = 0; k < i; k++) {
                    L[j][i] -= L[j][k] * L[i][k];
                }
                L[j][i] /= L[i][i];
            }
        }
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                U[i][j] = L[j][i];
            }
        }
    }
    void operator = (const Matrix &M){
        A = M.A, L = M.L, U = M.U, IA = M.IA, n = M.n;
    }
    void sub(vector<vector<f80>> &a, int r2, int r1, f80 k){ // r2
= r2 - k * r1
        for(int i = 0; i < n; i++) {
```

```
            a[r2][i] -= k * a[r1][i];
        }
    }
    void rowmul(vector<vector<f80>> &a, int r, f80 k){ // r = r *
k;
        for(int i = 0; i < n; i++) {
            a[r][i] *= k;
        }
    }
    void inverse() { // Call any LU - decomposition method first,
otherwise DoLittle by default
        if(!decomposed) LU_Dolittle(); // by default
        IA.clear();
        IA.resize(n);
        for(int i = 0; i < n; i++) {
            IA[i].resize(n, 0);
            IA[i][i] = 1;
        }
        auto T = L;
        for(int i = 0; i < n; i++){ // forward substitution
            f80 val = 1.0 / T[i][i];
            rowmul(T, i, val);
            rowmul(IA, i, val);
            for(int j = i + 1; j < n; j++) {
                f80 val = T[j][i];
                sub(T, j, i, val);
                sub(IA, j, i, val);
            }
        }
        T = U;
        for(int i = n - 1; i >= 0; i--) { // back substitution
            f80 val = 1.0 / T[i][i];
            rowmul(T, i, val);
            rowmul(IA, i, val);
            for(int j = i - 1; j >= 0; j--) {
                f80 val = T[j][i];
                sub(T, j, i, val);
                sub(IA, j, i, val);
            }
        }
    }
};

int main() {
    Matrix M(3);
    M.A = {{25, 15, -5}, {15, 18, 0}, {-5, 0, 11}};
    M.LU_Cholesky();
    cout << "L - Matrix: " << endl;
    print(M.L);
    cout << "U - Matrix " << endl;
```

```
    print(M.U);
    M.inverse();
    cout << "Inverse of Matrix : " << endl;
    print(M.IA);
    return 0;
}
```

Algorithm for finding inverse of a matrix using adjoint determinant method

The C++ implementation of this famous algorithm is done below.

```
#include <bits/stdc++.h>

using namespace std;

const long double eps=1e-9;
typedef long double ld;


class Matrix
{
     public:

          int N,M;
          vector<vector<ld>>mat;
          bool inverse_exists;
          ld det;

          Matrix(int n,int m)
          {
               N=n;
               M=m;
               mat.clear();
               mat.resize(N,vector<ld>(M,0));
               if(N!=M)
                    inverse_exists=0;
          }

          void reset()
          {
               mat.clear();
               N=0;
          }

          void Print()
          {
               cout<<"Matrix:"<<endl;
```

```
		for(int i=0;i<N;i++)
		{
			for(int j=0;j<M;j++)
			{
				cout<<fixed<<setprecision(5)<<mat[i][j]<<"
";
			}
			cout<<endl;
		}
	}

	ld Determinant(Matrix X)
	{
		ld res=1;
		for(int i=0;i<X.N;i++)
		{
			int select=i;
			while(select<X.N&&fabs(X.mat[select][i])<eps)
			{
				select++;
			}
			if(select==X.N)
				return 0;
			if(select!=i)
			{
				res*=-1;
				for(int j=0;j<X.N;j++)
				{
					swap(X.mat[i][j],X.mat[select][j]);
				}
			}
			res*=X.mat[i][i];
			for(int j=i+1;j<X.N;j++)
			{
				ld tmp=X.mat[j][i]/X.mat[i][i];
				for(int k=0;k<X.N;k++)
				{
					X.mat[j][k]-=X.mat[i][k]*tmp;
				}
			}
		}
		return res;
	}

	Matrix GetCofactor(Matrix X,int I,int J)
	{
		Matrix res(X.N-1,X.N-1);
		int curi=0;
		for(int i=0;i<X.N;i++)
		{
```

```
                    if(i==I)
                        i++;
                    if(i>=X.N)
                        break;
                    int curj=0;
                    for(int j=0;j<X.N;j++)
                    {
                        if(j==J)
                            j++;
                        if(j>=X.N)
                            break;
                        res.mat[curi][curj]=X.mat[i][j];
                        curj++;
                    }
                    curi++;
                }
                return res;
            }

            Matrix Adjoint(Matrix X)
            {
                Matrix res(X.N,X.N);
                for(int i=0;i<X.N;i++)
                {
                    for(int j=0;j<X.N;j++)
                    {
                        res.mat[i]
[j]=Determinant(GetCofactor(X,i,j));
                        if((i+j)%2==1)
                            res.mat[i][j]*=-1;
                    }
                }
                return res;
            }

            Matrix Inverse()
            {
                Matrix res(N,N);
                res.inverse_exists=0;
                if(N!=M)
                    return res;
                Matrix B(N,N);
                B.mat=mat;
                ld det=Determinant(B);
                if(fabs(det)<eps)
                    return res;
                res=Adjoint(B);
                res.inverse_exists=1;
                for(int i=0;i<N;i++)
                {
```

```
                    for(int j=0;j<N;j++)
                    {
                         res.mat[i][j]/=det;
                    }
               }
               for(int i=0;i<N;i++)
               {
                    for(int j=i+1;j<N;j++)
                    {
                         swap(res.mat[i][j],res.mat[j][i]);
                    }
               }
               return res;
          }


};

int main()
{
     Matrix A(3,3);
     A.mat={{5,7,9},{4,3,8},{7,5,6}};
     Matrix B=A.Inverse();
     if(B.inverse_exists==0)
     {
          cout<<"Inverse Does Not Exist!!"<<endl;
     }
     else
     {
          B.Print();
     }
     return 0;
}
```

Algorithm for finding inverse of a matrix using adjoint determinant method

The C++ implementation of this famous algorithm is done below.

```
#include <bits/stdc++.h>

using namespace std;

const long double eps=1e-9;
typedef long double ld;
```

```
class Matrix
{
     public:

          int N,M;
          vector<vector<ld>>mat;
          bool inverse_exists;

          Matrix(int n,int m)
          {
               N=n;
               M=m;
               mat.clear();
               mat.resize(N,vector<ld>(M,0));
               if(N!=M)
                    inverse_exists=0;
          }

          void reset()
          {
               mat.clear();
               N=0;
          }

          void Print()
          {
               cout<<"Matrix:"<<endl;
               for(int i=0;i<N;i++)
               {
                    for(int j=0;j<M;j++)
                    {
                         cout<<fixed<<setprecision(5)<<mat[i][j]<<"
";
                    }
                    cout<<endl;
               }
          }

          Matrix Gauss_Jordan_Elimination(Matrix X)
          {
               X.inverse_exists=0;
               for(int i=0;i<N;i++)
               {
                    int select=i;
                    for(int j=i;j<N;j++)
                    {
                         if(fabs(X.mat[j][i])>fabs(X.mat[select]
[i]))
                         {
                              select=j;
```

```
                }
            }
            if(fabs(X.mat[select][i])<eps)
                return X;
            for(int j=i;j<2*N;j++)
                swap(X.mat[i][j],X.mat[select][j]);
            for(int j=0;j<N;j++)
            {
                if(j!=i)
                {
                    ld tmp=X.mat[j][i]/X.mat[i][i];
                    for(int k=0;k<2*N;k++)
                    {
                        X.mat[j][k]-=tmp*X.mat[i][k];
                    }
                }
            }
            ld tmp=X.mat[i][i];
            for(int j=0;j<2*N;j++)
            {
                X.mat[i][j]/=tmp;
            }
        }
        X.inverse_exists=1;
        return X;
}

Matrix Inverse()
{
        Matrix res(N,N);
        res.inverse_exists=0;
        if(N!=M)
            return res;
        Matrix B(N,N+N);
        for(int i=0;i<N;i++)
        {
            for(int j=0;j<N;j++)
            {
                B.mat[i][j]=mat[i][j];
            }
            B.mat[i][N+i]=1;
        }
        B=Gauss_Jordan_Elimination(B);
        if(B.inverse_exists==0)
            return res;
        res.inverse_exists=1;
        for(int i=0;i<N;i++)
            for(int j=0;j<N;j++)
                res.mat[i][j]=B.mat[i][j+N];
        return res;
```

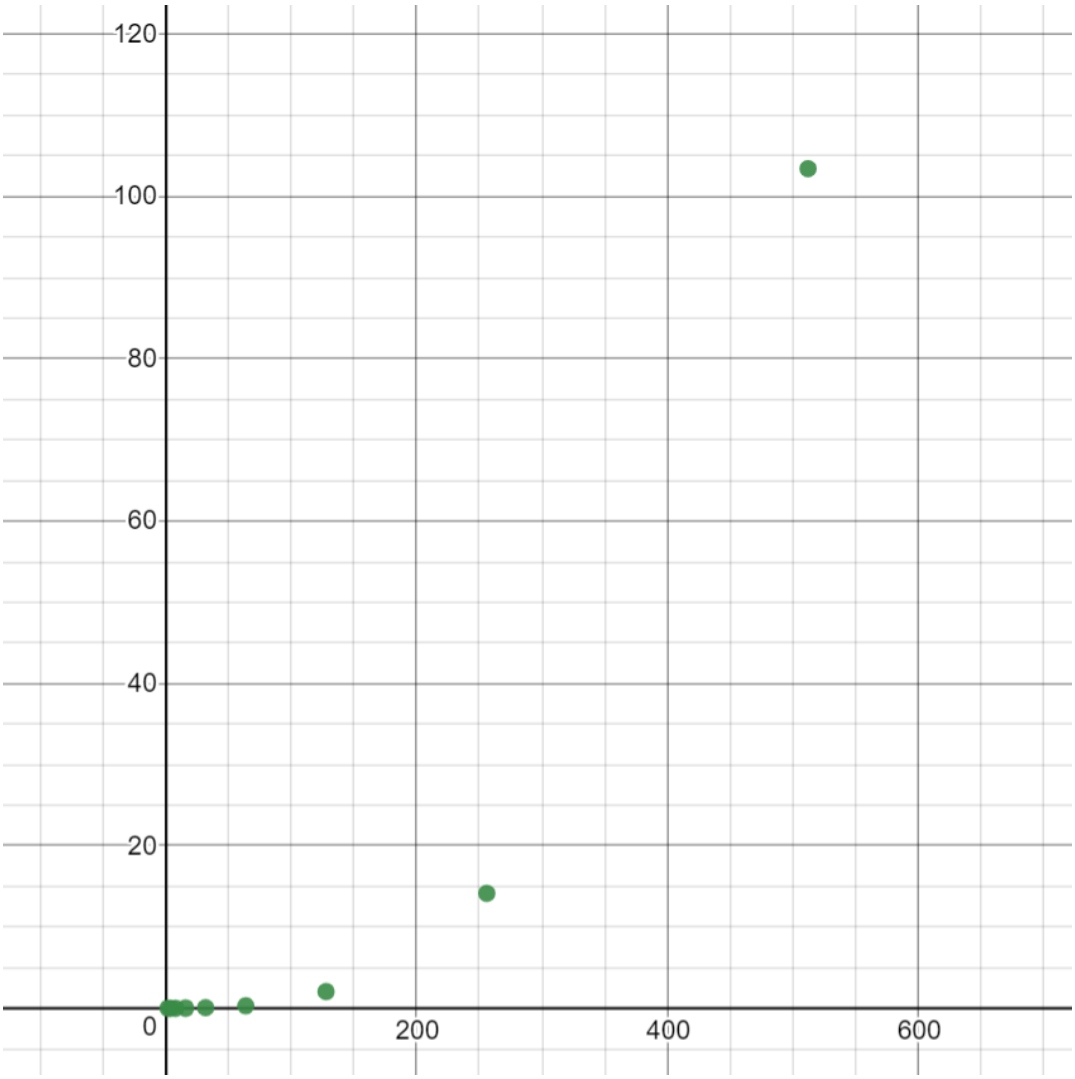
```
            }
};

int main()
{
     Matrix A(3,3);
     A.mat={{5,7,9},{4,3,8},{7,5,6}};
     Matrix B=A.Inverse();
     if(B.inverse_exists==0)
     {
          cout<<"Inverse Does Not Exist!!"<<endl;
     }
     else
     {
          B.Print();
     }
     return 0;
}
```

RunTime Ananlysis of various Algorithms

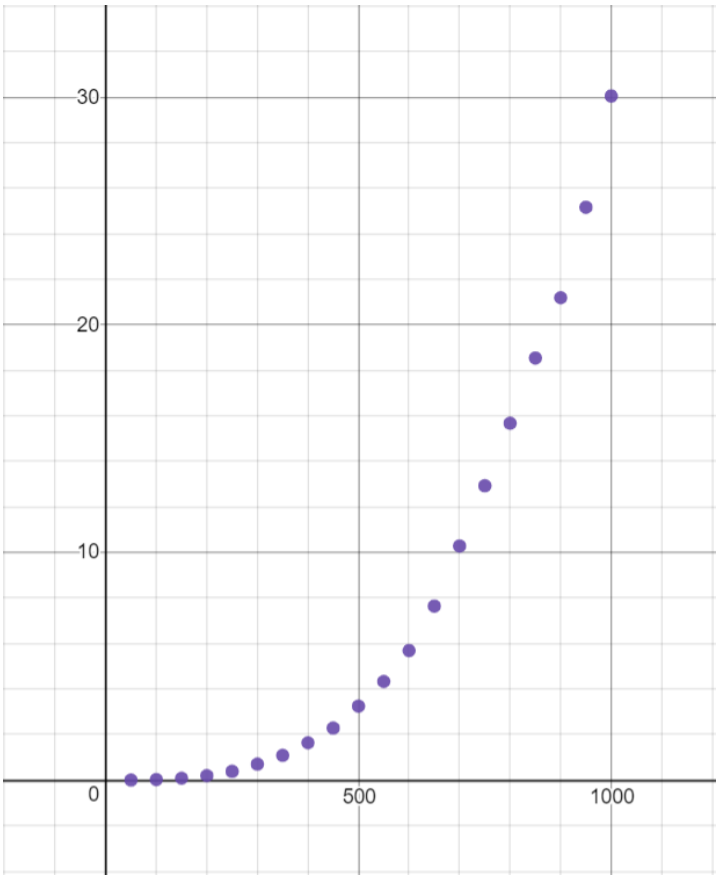
The runtime analysis of each code has been done using **input size *n* on the horizontal axis** and **time taken in seconds in the vertical axis**. This is same for all the graphs below.

All these run times have been recorded using Intel(R) Core™ i7-8550U CPU at 1.8GHz.

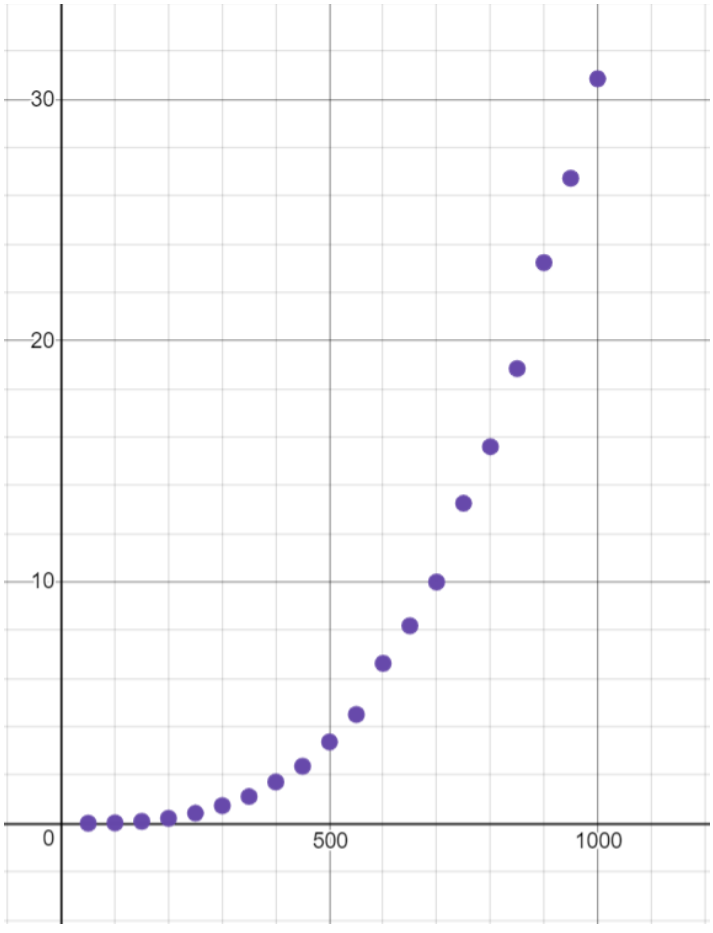
Strassen's Multiplication:



Dolittle LU Decomposition:

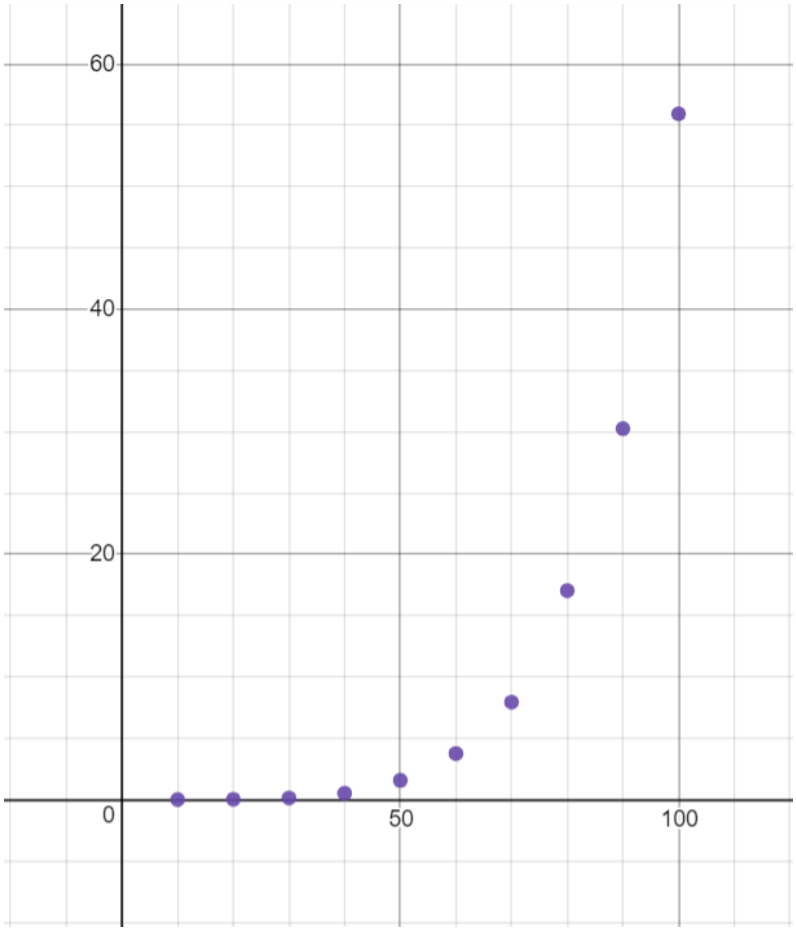


Crout LU Decomposition:

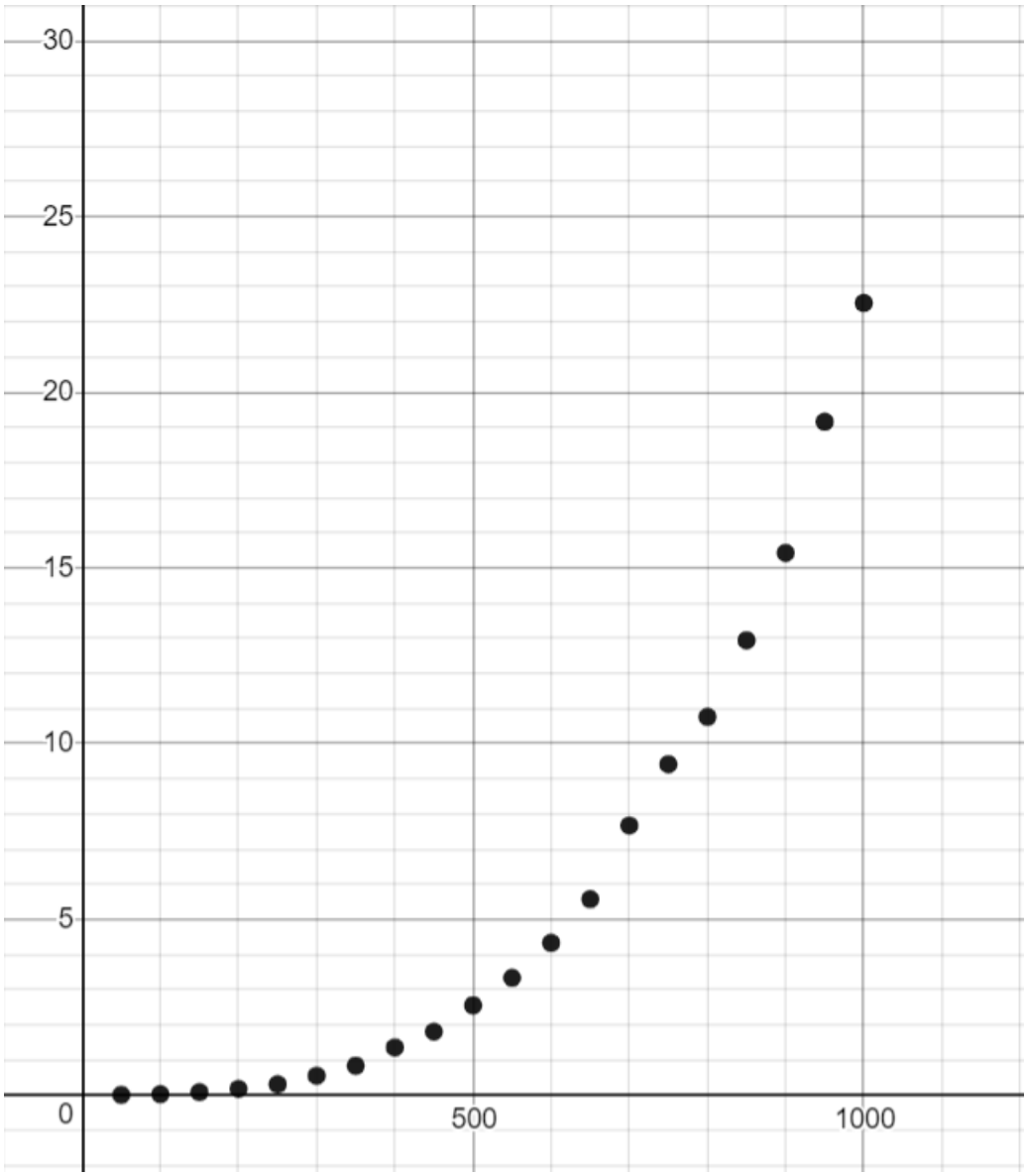


Cholesky LU Decomposition:

Inversion using Adjoint Determinant Method:



Inverse using Guass Elimination Method



Remaining Tasks

Future Goals and targets are listed below:

- Develop implementation for calculating eigenvalues and eigenvectors.
- Implement the algorithm for fractional power of matrix.
- Implement the algorithm for Normalization of a square matrix.
- Using Cayley Hamilton Theorem to find Inverse of matrix.

All codes and implementations can be found at
https://github.com/AayushChaturvedi/IOP2019