# Compito 9.1

Il compito riguarda l'implementazione di MCMatrixMultiplicationVerifier, algoritmo preso come rappresentate della categoria di riduzione casuale della dimensionalità, in cui si sfrutta la probabilità per ridurre al dimensione degli input mediante funzioni casuali.
In particolare, questo algoritmo riduce la complessità temporale della verifica di una moltiplicazione tra matrici dal costo di $O(n^3)$ a $O(n^2)$, grazie al fatto che quando il numero di run è sufficientemente alto -come da "tradizione" per gli algoritmi Montecarlo- la probabilità di non ottenere un risultato giusto è sufficientemente bassa da rendere l'algoritmo affidabile.
Nel caso preso in esame, l'uguaglianza di cui dobbiamo verificare la veridicità sarà sempre falsa, in modo da poter studiare il numero di volte in cui l'algoritmo fornisce un esito errato.

# Il codice:

Per la codifica dell'algoritmo ho scelto di usare il linguaggio C++ con nomi di variabili e funzioni in lingua inglese e questa volta commenti in inglese, come per i compiti precedente la casualità (in questo caso legata alla creazione delle matrici e sopratutto del vettore r) è stata affidata alla funzione rand() della stdlib di C++.
Eviterò di presentare le funzioni che effettuano calcoli tra matrici, considerando la loro implementazione poco rilevante e poco creativa, oltre che i loro nomi chiari ed esplicativi.

Inizializzazioni:

```
const int N = 100;
const int IndexToChange = 5;
const int Runs = 100;

const int matrixNumbers[5] = {-2, -1, 0, 1, 2};
```

Riporto i valori sopratutto per averli presenti anche nella relazione e non solo nel testo.
Come da testo, la dimensione delle matrici, così come il numero di run è stata impostata a 100, mentre la scelta del punto da incrementare di 1 nella matrice C, non essendo rilevante per l'algoritmo l'ho invece imposta arbitrariamente a (5,5).
Il sacchetto dal quale venivano estratti i numeri contenuti nelle matrici l'ho riempito come indicato dal testo.
I valori di K (5, 10 e 20) per comodità non li ho dichiarati tramite costante, ma

direttamente inseriti all'interno dei cicli for, pagando la semplicità con la modificabilità del codice.

Singola run:

```
vector<bool> singleRun(vector<vector<int>> A){

    //Inizializing matrixes
    vector<vector<int>> B = matrixBuilder(N);
    vector<vector<int>> C = matrixMultiplier(A, B);

    vector<bool>returnVector = vector<bool>(3, false);

    C[IndexToChange][IndexToChange]++;

    for(int k = 5; k <= 20; k*=2){   //Running the test with k = 5, 10, 20
        bool result = MCMatrixMultiplicationVerifier(A, B, C, k);

        //Writing the result on a file
        ofstream file;
        file.open("MCMatrixMultiplicationVerifier.txt", ios::app);
        file << '\t' << "k = " << k << ": " << result << endl;
        if(k == 20) file << endl;   //Adding a new line at the end of the run
        file.close();

        switch(k){
            case 5:
                returnVector[0] = result;
                break;
            case 10:
                returnVector[1] = result;
                break;
            case 20:
                returnVector[2] = result;
                break;
        }
    }
    return returnVector;
}
```

La funzione per la singola run (che dal main verrà chiamata cento volte) ri inizializza B e calcola C a partire da B ed A, la quale è invece generata un unica volta e quindi passata come argomento alla funzione.
A questo punto, rompe l'uguaglianza AB = C e itera tre volte, uno per ogni valore di K considerato, chiamando la funzione protagonista: MCMatrixMultiplicationverifier.
Evito di descrivere il resto della funzione, che serve solamente a conservare su file le informazioni riguardanti i risultati di questa run.

```
bool MCMatrixMultiplicationVerifier(vector<vector<int>> A, vector<vector<int>> B,
vector<vector<int>> C, int k){
    for(  int i=0; i < k; i++){
        //Inizilazing vectors r, s, t, u
        vector<int> r = randomVector(N);
        vector<int> s = matrixVectorMultiplier(B, r);
        vector<int> t = matrixVectorMultiplier(A, s);
        vector<int> u = matrixVectorMultiplier(C, r);

       if (t != u) return false;    //If t != u, it is certanly false,
                       //Otherwise could be true so we check again for another r
    }
    return true;
}
```

Ho implementato la funzione centrale seguendo lo pseudo algoritmo delle note con l'unica variazione dell'inserimento dell'iterazione guidata dal parametro K, che ha lo scopo di rendere esplicita l'importanza di aumentare il numero di verifiche probabilistiche per assicurarsi la validità del risultato.
Indipendemente dal valore di K, nel caso l'algoritmo riveli che l'uguaglianza è falsa non è necessario iterare oltrre: il risultato è certamente quello.
Altrimenti il campionamento del vettore r potrebbe aver invalidato il risultato ed è quindi necessario progredire quanto previsto con i tentativi basati su campionamenti differenti.

# Analisi risultati:

Result distribution:
r000: 95 r001: 0 r010: 0
r011: 0 r100: 5 r101: 0
r110: 0 r111: 0

Ho raccolto i risultati per tipologia, indicando nel nome di oguna il risultato ottenuto con k = 5 (0 se la verifica ci comunica sbagliata l'uguaglianza, 1 se probabilmente giusta), poi k= 10 e k= 20 (forse potevo trovare una nomenclatura migliore per questi dati).
Essenzialmente quindi, senza bisogno di esporre grafici ciò che emerge è che su un numero di run contenuto, la maggior parte rivelano già con 5 iterazioni correttamente l'uguaglianza come errata, mentre una porzione ben ridotta (in questo caso 5%) necessità 10 iterazioni, mentre in nessun caso è necessario effettuarne di più.
Questi risultati, che a primo impatto potrebbero far domandare se sia effettivamente tutto giusto, data la alta percentuale di verifiche corrette già con valori bassi di K, è in realtà in linea con quanto appreso durante il corso:
Infatti abbiamo studiato che la probabilità che eseguendo MCMatrixMultiplicationVerifier k volte, la probabilità di una corretta verifica dell'uguaglianza è $Pr(k) = 1 - 1/2^k$ .
Quindi per il valore minore di $k = 1 - 1/2^5 = 1 - 0{,}03125 = 96{,}875\%$, che dista di poco dal nostro 95%.
La potenza di questo algoritmo risiede quindi nella probabilità di insuccesso basata sul reciproco di un'esponenziale, la quale decresce molto rapidamente e richiede quindi di moltiplicare costo dell'algoritmo per una costante moltiplicativa contenuta.