**Engineering design**

**Phase 1. Identification of the problem**

Venus and Mars have been at war for the last 4 years. In the current year, Venus wants to win the war decisively, to do so they want to predict the positions of Mars’ fleet, having in mind that they archive the positions of past battles.

Identification of necessities and symptoms

* It’s required to multiply matrices.
* Venus’ troops don’t have a way of predicting the exact location of the enemy’s fleet.
* The found location has to be expressed in terms of x and y coordinates.
* The solution to the problem has to be efficient so that Venus can make decisions on time.

Definition of the problem

Venus requires the development of a software that allows them to visualize in a matrix the locations of the ships of the enemy.

The software will have the following functionalities:

1. A graphical interface that allows the user to input the values of the number of rows and columns of the past battle board.
2. A graphical interface that allows the user to input whether the values generated randomly will have repeats or not.
3. A graphical interface that shows the positions of the Martians in the current battle.
4. A graphical interface that allows to generate n matrices that are to be multiplied.

**Phase 2. Recompilation of information**

Matrix: In mathematics, a matrix (plural: matrices) is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns.

Matrix Multiplication: Multiplication of two matrices is defined if and only if the number of columns of the left matrix is the same as the number of rows of the right matrix. If A is an m-by-n matrix and B is an n-by-p matrix, then their matrix product AB is the m-by-p matrix whose entries are given by the dot product of the corresponding row of A and the corresponding column of B.

Formally, the definition of matrix multiplication is that if *C* = *AB* for an *n* × *m* matrix *A* and an *m* × *p* matrix *B*, then *C* is an *n* × *p* matrix with entries



Dot product: In mathematics, the dot product or scalar product is an algebraic operation that takes two equal-length sequences of numbers (usually coordinate vectors) and returns a single number. Algebraically, the dot product is the sum of the products of the corresponding entries of the two sequences of numbers.

Prime number: A prime number (or a prime) is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers.

**Phase 3. Looking for creative solutions**

In order to multiply matrices we are considering different algorithms that have proven to do so with matrices of different dimensions. The algorithms to consider are the following:

1. Iterative algorithm: This is the brute force approach which applies the definition of matrix multiplication explained above, applying each dot product and adding the result to each position of the resulting matrix.

Assuming the two input matrices can be multiplied, this algorithm works both for square matrices (matrices with the same number of columns and rows) and for matrices that are not square.

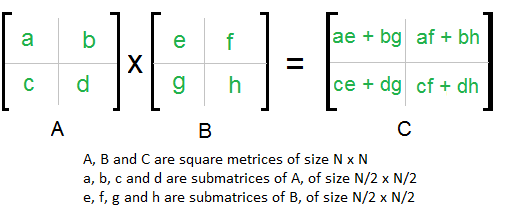
1. Divide and conquer algorithm: This algorithm uses the divide and conquer paradigm which is basically breaking a problem down into two or more subproblems of the same type, until these become simple enough to be solved directly.

All the solutions to the subproblems are then combined to give solution to the original problem.

For the matrix multiplication problem, we can use a divide and conquer approach. Let A and B be two matrices of size N x N we want to multiply and C the result of their multiplication



The algorithm divides A and B in 4 submatrices of size N/2 x N/2 and then calculates the following values recursively: ae + bg, af + bh, ce + dg and cf + dh.

¹

This algorithm works for all square matrices whose dimensions are powers of two, that is, the shapes are 2*n* × 2*n* for some *n*. Matrices that have dimensions that don’t meet the criteria mentioned above can be filled , or padded with zeros until they do.

Using the block matrix identity

²

Here A,B are n×n matrices, and the big matrices are N×N for some N>n. In other words, we add N−n rows and N−n columns of zeroes.

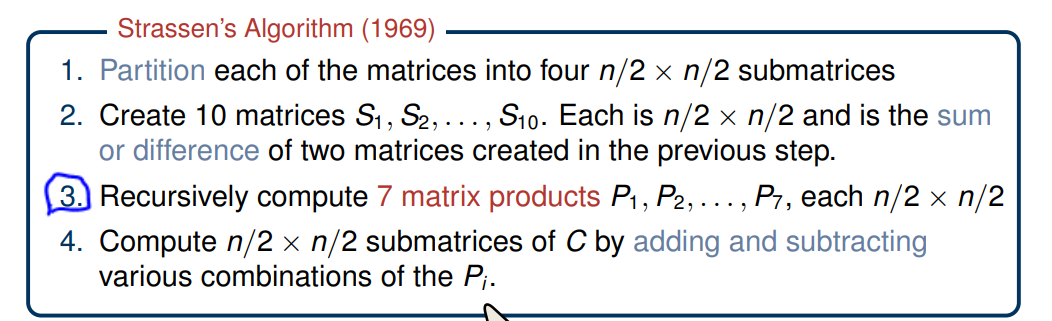
This can be done in two ways:

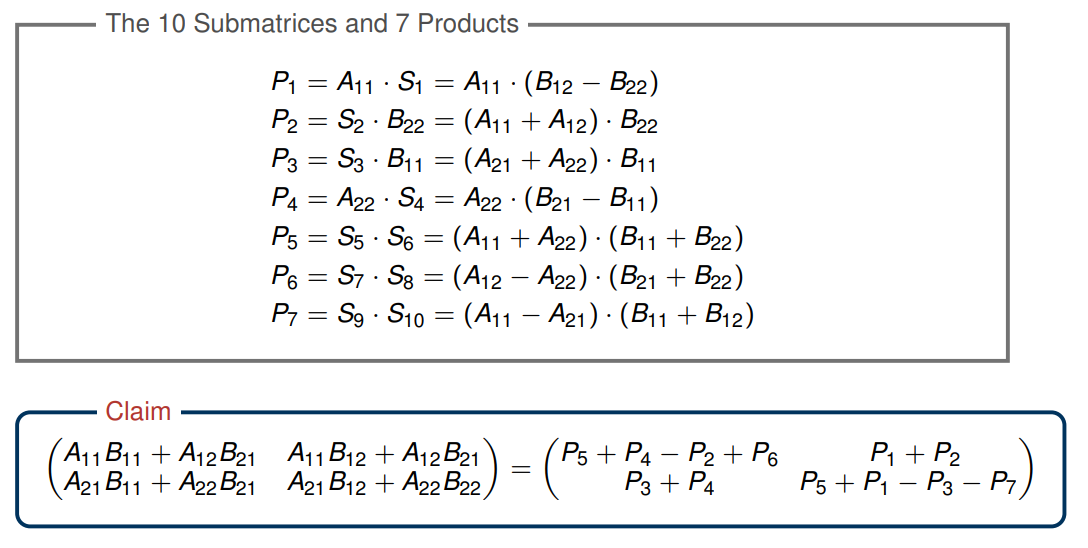
* Pad the original n×n matrices to N×N matrices, where N is the closest power of 2. Note that N<2n, so this doesn't affect the asymptotic complexity.
* Pad the matrices recursively. Whenever a recursive call gets m×m matrices with m>1 odd, we pad them to (m+1)×(m+1) matrices by adding a single row and a single column of zeroes. This also doesn't affect the asymptotic complexity.

Formally, the divide and conquer algorithm consists of eight multiplications of pairs of submatrices, followed by an addition step. The divide and conquer algorithm computes the smaller multiplications recursively, using the scalar multiplication *c*11 = *a*11*b*11 as it’s base case.

A variant of this algorithm that works for matrices of arbitrary shapes and is faster in practice splits matrices in two instead of four submatrices, as follows. Splitting a matrix now means dividing it into two parts of equal size, or as close to equal sizes as possible in the case of odd dimensions.

1. Strassen algorithm: The goal of Strassen’s algorithm is to reduce the number of multiplications in the divide and conquer algorithm. It does so by performing only 7 recursive multiplications of N/2 x N/2 matrices.

³



We iterate this division process *n* times (recursively) until the submatrices degenerate into numbers.

1. Coppersmith-Winograd algorithm: The current *O*(*nk*) algorithm with the lowest known exponent *k* is a generalization of the Coppersmith–Winograd algorithm that has an asymptotic complexity of *O*(*n*2.3728639), by François Le Gall. The Le Gall algorithm, and the Coppersmith–Winograd algorithm on which it is based, are similar to Strassen's algorithm: a way is devised for multiplying two *k* × *k*-matrices with fewer than *k*3 multiplications, and this technique is applied recursively. However, the constant coefficient hidden by the Big O notation is so large that these algorithms are only worthwhile for matrices that are too large to handle on present-day computers.

Another problem we are dealing with is to find prime numbers, some useful algorithms to find prime numbers are the following:

1. Sieve of Eratosthenes: In mathematics, the **Sieve of Eratosthenes** is a simple, ancient algorithm for finding all prime numbers up to any given limit.

It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2. The multiples of a given prime are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime. This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime.

1. Sieve of Atkins: It’s a modern algorithm for finding all prime numbers up to a specified integer. Compared with the ancient sieve of Eratosthenes, which marks off multiples of primes, the sieve of Atkin does some preliminary work and then marks off multiples of squares of primes, thus achieving a better theoretical asymptotic complexity.

**Phase 4. Transition from ideas to preliminary designs**

Firstly, we’ll mention that we won’t be using the Coppersmith-Winograd algorithm due to its constant factor being to big for small matrices , this renders the algorithm unsuitable for this project.

Revising the other alternatives for the matrix multiplication problem we get:

Iterative algorithm

* Works both for square matrices and not square matrices providing an accurate result every time.
* For large enough matrices, the time it takes grows too fast making this algorithm less efficient.
* This algorithm suggests that arithmetic operations are necessary putting this algorithm at an Θ(*n*3) time complexity.

Divide and conquer algorithm:

* Has one variant for square matrices and another that works in matrices of arbitrary shape that is faster in practice.
* Although the complexity of this algorithm as a function of n is

T(n) = 8T(n/2) + Θ(*n*2), Application of the master theorem for divide-and-conquer recurrences shows this recursion to have the solution Θ(*n*3), the same as the iterative algorithm, also making it less efficient.

Strassen algorithm

* Has reduced execution time in comparison to the other algorithms because it uses a clever mathematical deduction.
* Works for square matrices whose dimensions are powers of two, for matrices that don’t meet this criteria the solution is to pad the matrices with zeros, using the block matrix identity.
* Practical implementations of Strassen's algorithm switch to standard methods of matrix multiplication for small enough submatrices, for which those algorithms are more efficient.
* The asymptotic complexity for multiplying matrices of size N = 2*n* using the Strassen algorithm is O(N*2.8074*)

For the prime finding problem, we revise the next alternative:

Sieve of Eratosthenes

* A number is prime, if none of the smaller prime numbers divides it. Since we iterate over the prime numbers in order, we already marked all numbers, who are divisible by at least one of the prime numbers, as divisible. Hence if we reach a cell and it is not marked, then it isn't divisible by any smaller prime number and therefore has to be prime.
* Sieve of Eratosthenes is an algorithm for finding all the prime numbers in a segment [1;n] using O(nlog⁡log⁡n) operations which is incredibly fast making us able to find primes up to 10 to the 8th power very quickly.
* The biggest weakness of the algorithm is that it “walks” along the memory multiple times, only manipulating single elements. This is not very cache friendly. And because of that, the constant which is concealed in O(nlog⁡log⁡n) is comparably big.
* Despite this, there are plenty of optimizations to reduce the memory problem.

Sieve of Atkins

* This algorithm uses different results of applying modulo of certain numbers, equations and the concept of squarefree to determine whether a number is prime or not.

<https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html>

**Phase 5. Evaluation and selection of solutions**

**Criteria**

Considering the options that remain after going through phase three and four, the following criteria has been proposed in order to choose the best solution. The greater the total score, the higher rated the algorithm will be.

* Criteria A. Efficiency, a fundamental requirement is that the matrix multiplication has to be executed quickly. The complexity of the solution is:
* [3] logarithmic complexity
* [2] less than cubic complexity
* [1] cubic complexity
* Criteria B. Completeness, the solution gives a correct output for any two matrices that can be multiplied. The solution gives a correct result for:
* [2] any pair of matrices
* [1]some specific cases
* Criteria C. Ease of implementation
* [2] the solution can be easily implemented in the programming language
* [1] the implementation of the solution may require more advanced techniques

**Evaluation**

Taking into account the criteria established above, we will carry out the evaluation of the solutions

*For matrix multiplication:*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Solution | Criteria A | Criteria B | Criteria C | Total |
| Solution 1. Iterative algorithm | 1 | 1 | 2 | 4 |
| Solution 2. .Divide and conquer algorithm | 1 | 1 | 2 | 4 |
| Solution 3. Strassen algorithm | 1 | 2 | 2 | 5 |

*For finding prime numbers:*

For the evaluation of these solutions we will only consider criteria A and C.

|  |  |  |  |
| --- | --- | --- | --- |
| Solution | Criteria A | Criteria C | Total |
| Solution A. Sieve of Eratosthenes | 3 | 2 | 5 |
| Solution B. Sieve of Atkins | 3 | 1 | 4 |

**Selection**

The requirements that were assigned indicate that three algorithms should be used to solve the problem of matrix multiplication, given this condition, the three remaining solutions will be implemented. Although it should be mentioned that the third solution (Strassen algorithm) is the best among all of them.

As for the problem of finding prime numbers, according to the evaluation, the best solution is the solution A(Sieve of Eratosthenes). This was chosen mainly because its implementation is already known and it is not too complicated.

**Phase 6. Preparation of information and specifications**

***class diagram :***

**https://drive.google.com/file/d/1TSfvmKzIEOb8iEtqAZ5PiHxJrb-ccJ86/view?usp=sharing**

Problem specification (Input/Output)

* Problem: Matrix multiplication

Input: Integers N, M which represent the number of rows and columns of the first matrix respectively, P, Q, which represent the number of rows and columns of the second matrix and finally and boolean value that indicates whether or not the matrix will have repeats.

Output: A matrix C, the solution of the multiplication between two randomly generated matrices A and B.

Constraints:

* The solution validates that the matrices that are given can be multiplied, that is, they have valid dimensions. If not, the operation can not be executed.
* The maximum possible size of these matrices should be 1000x1000 given that it is very hard to fit a bigger matrix in memory.

**Flow chart of the algorithm:** [FlowMatrix](https://drive.google.com/file/d/1N9JrX6sPEL4l3pWcpj0LFGQhpeqBwNFS/view?usp=sharing)

Following is pseudocode of the three algorithms that are to be implemented to solve the matrix multiplication problem

Iterative approach

* Input: matrices *A* and *B*
* Let *C* be a new matrix of the appropriate size
* For *i* from 1 to *n*:
  + For *j* from 1 to *p*:
    - For *k* from 1 to *m*:
      * ← + *Aik* × *Bkj*
* return *C*

Divide and conquer approach for square matrices

* Input: matrices *A*, B and integer n
* Let *C* be a new n x n matrix
* if n == 1

←

else partition A, B into 4 submatrices of size n/2 x n/2 and recurse

← DivideAndConquer(, ) + DivideAndConquer(, )

← DivideAndConquer(, ) + DivideAndConquer(, )

← DivideAndConquer(, ) + DivideAndConquer(, )

← DivideAndConquer(, ) + DivideAndConquer(, )

* return *C*

Strassen’s algorithm for square matrices

* Input: matrices *A*, B and integer n
* Let *C* be a new n x n matrix
* if n == 1

←

else calculate the 7 submatrices recursively

← Strassen(, )  
 ← Strassen(+ )

← Strassen(+ )

← Strassen(, )

← Strassen(, )

← Strassen(, )

← Strassen(, )

←

←

←

←

* return *C*

Problem specification (Input/Output)

* Problem: Prime finding

Input: Given a number N, determine if this number is prime or if it’s definite composite.

Output: True if the number given is prime, False otherwise.

Constraints:

* The given number N is a positive integer, 1 <= N <= 1x10^6.

**Flow chart of the algorithm:** [FlowPrimes](https://drive.google.com/file/d/1gHze58KHMSB1iYYjlC4yD3AizB3-hT7i/view?usp=sharing)

Sieve of Eratosthenes pseudocode

* Let primes be an integer array indexed by integers 2 to n, initially all set to 0
* for i from 2 to

if primes[i] == 0

for j from to an

primes[j] = 1

* return primes

Space complexity Iterative approach

Assuming A, B are square matrices of equal size

|  |  |  |
| --- | --- | --- |
|  | **Variables** | **Amount of atomic values** |
| **Input** | A,B | , |
| **Output** | C |  |
| **Auxiliary** | i, j, k | 1, 1, 1 |
|  |  |  |

So the total space complexity is: O()

Time complexity Iterative Approach

|  |  |
| --- | --- |
| Input: matrices *A* and *B* | Amount of repeats: |
| Let *C* be a  new matrix of the appropriate size | 1 |
| For *i* from 1  to *n*: | N+1 |
| For *j* from 1 to *p*: | N(P+1) = NP + N |
| For *k* from 1 to *m* | MNP + NP + MN + N – M -1 |
| Cij ← Cij+ *Aik* × *Bkj* | MNP + NP + MN + N – M -2 |
| return C | 1 |

T(n) =

T(n) =

T(n) =

So the total time complexity is: O(MNP)

And in case the matrices are square with the same size n, we get a time complexity of:

O(

**Note:** The complexity of the recursive algorithms is not to be calculated due to the lack of knowledge so we will just assume it’s correctness.

Time complexity divide and conquer algorithm: O(

Time complexity Strassen’s algorithm: O(

**Phase 7. Implementation of the design**

Implementation in a programming language, we decided to use Java in this case.

1. Validate matrix sizes

|  |  |
| --- | --- |
| Name: | validateSizes |
| Description: | Tells whether or not two matrices can be multiplied |
| Input: | Number of columns from the first matrix and number of rows from the second matrix |
| Output: | True if the matrices can be multiplied, false otherwise |

public boolean validateSizes(int a, int b){

boolean success = false;

if(a == b)

success = true;

return success;

}

1. Calculate prime numbers

|  |  |
| --- | --- |
| Name: | generatePrimes |
| Description: | generates all prime numbers up to 1000000 |
| Input: | None |
| Output: | An array of integers which tells what numbers are prime |

public void generatePrimes() {

primeNumbers[0] = primeNumbers[1] = 1;

for(int i = 2; i < 1000; i++){

if(primeNumbers[i] == 0){

int mul = 2;

while(mul \* i < 1000001)

primeNumbers[mul++ \* i] = 1;

}

}

}

c. Iterative matrix multiplication

|  |  |
| --- | --- |
| Name: | multiplyMatricesFirstOption |
| Description: | Multiplies all matrices in the list of matrices iteratively. |
| Input: | None |
| Output: | A matrix C, the answer to the matrix multiplication. |

public int[][] multiplyMatricesFirstOption() {

int[][] A = matrices.get(0).getMatrix();

for (int y = 1; y < matrices.size(); y++) {

int[][] B = matrices.get(y).getMatrix();

int n = A.length;

int m = B[0].length;

int p = A[0].length;

int[][] C = new int[n][m];

for(int i = 0; i < n; i++){

for(int j = 0; j < m; j++){

for(int k = 0; k < p; k++){

C[i][j] += A[i][k] \* B[k][j];

}

}

}

A = C;

}

return A;

}

d. Divide and Conquer matrix multiplication

|  |  |
| --- | --- |
| Name: | multiplyMatricesSecondOption |
| Description: | Multiplies all matrices in the list of matrices using the divide and conquer approach. |
| Input: | None |
| Output: | A matrix C, the answer to the matrix multiplication. |

private int[][] multiplyMatricesSecondOptionAux(int[][] A, int[][] B, int n) {

int[][] C = new int[n][n];

if (n == 1) {

C[0][0] = A[0][0] \* B[0][0];

return C;

} else {

int[][] A11 = new int[n / 2][n / 2];

int[][] A12 = new int[n / 2][n / 2];

int[][] A21 = new int[n / 2][n / 2];

int[][] A22 = new int[n / 2][n / 2];

int[][] B11 = new int[n / 2][n / 2];

int[][] B12 = new int[n / 2][n / 2];

int[][] B21 = new int[n / 2][n / 2];

int[][] B22 = new int[n / 2][n / 2];

deconstructMatrix(A, A11, 0, 0);

deconstructMatrix(A, A12, 0, n / 2);

deconstructMatrix(A, A21, n / 2, 0);

deconstructMatrix(A, A22, n / 2, n / 2);

deconstructMatrix(B, B11, 0, 0);

deconstructMatrix(B, B12, 0, n / 2);

deconstructMatrix(B, B21, n / 2, 0);

deconstructMatrix(B, B22, n / 2, n / 2);

int[][] C11 = addMatrix(divideAndConquerMM(A11, B11, n / 2), divideAndConquerMM(A12, B21, n / 2), n / 2);

int[][] C12 = addMatrix(divideAndConquerMM(A11, B12, n / 2), divideAndConquerMM(A12, B22, n / 2), n / 2);

int[][] C21 = addMatrix(divideAndConquerMM(A21, B11, n / 2), divideAndConquerMM(A22, B21, n / 2), n / 2);

int[][] C22 = addMatrix(divideAndConquerMM(A21, B12, n / 2), divideAndConquerMM(A22, B22, n / 2), n / 2);

constructMatrix(C11, C, 0, 0);

constructMatrix(C12, C, 0, n / 2);

constructMatrix(C21, C, n / 2, 0);

constructMatrix(C22, C, n / 2, n / 2);

}

return C;

e. Strassen matrix multiplication algorithm

|  |  |
| --- | --- |
| Name: | multiplyMatricesThirdOption |
| Description: | Multiplies all matrices in the list of matrices using Strassen’s algorithm. |
| Input: | None |
| Output: | A matrix C, the answer to the matrix multiplication. |

public static void multiplyMatricesThirdOption(int[][] A, int[][] B, int[][] C, int n) {

if (n == 2) {

C[0][0] = (A[0][0] \* B[0][0]) + (A[0][1] \* B[1][0]);

C[0][1] = (A[0][0] \* B[0][1]) + (A[0][1] \* B[1][1]);

C[1][0] = (A[1][0] \* B[0][0]) + (A[1][1] \* B[1][0]);

C[1][1] = (A[1][0] \* B[0][1]) + (A[1][1] \* B[1][1]);

} else {

int[][] A11 = new int[n / 2][n / 2];

int[][] A12 = new int[n / 2][n / 2];

int[][] A21 = new int[n / 2][n / 2];

int[][] A22 = new int[n / 2][n / 2];

int[][] B11 = new int[n / 2][n / 2];

int[][] B12 = new int[n / 2][n / 2];

int[][] B21 = new int[n / 2][n / 2];

int[][] B22 = new int[n / 2][n / 2];

int[][] P = new int[n / 2][n / 2];

int[][] Q = new int[n / 2][n / 2];

int[][] R = new int[n / 2][n / 2];

int[][] S = new int[n / 2][n / 2];

int[][] T = new int[n / 2][n / 2];

int[][] U = new int[n / 2][n / 2];

int[][] V = new int[n / 2][n / 2];

deconstructMatrix(A, A11, 0, 0);

deconstructMatrix(A, A12, 0, n / 2);

deconstructMatrix(A, A21, n / 2, 0);

deconstructMatrix(A, A22, n / 2, n / 2);

deconstructMatrix(B, B11, 0, 0);

deconstructMatrix(B, B12, 0, n / 2);

deconstructMatrix(B, B21, n / 2, 0);

deconstructMatrix(B, B22, n / 2, n / 2);

strassenMMHelper(addMatrix(A11, A22, n / 2),

addMatrix(B11, B22, n / 2), P, n / 2);

strassenMMHelper(addMatrix(A21, A22, n / 2), B11, Q, n / 2);

strassenMMHelper(A11, subtractMatrix(B12, B22, n / 2), R, n / 2);

strassenMMHelper(A22, subtractMatrix(B21, B11, n / 2), S, n / 2);

strassenMMHelper(addMatrix(A11, A12, n / 2), B22, T, n / 2);

strassenMMHelper(subtractMatrix(A21, A11, n / 2),

addMatrix(B11, B12, n / 2), U, n / 2);

strassenMMHelper(subtractMatrix(A12, A22, n / 2),

addMatrix(B21, B22, n / 2), V, n / 2);

int[][] C11 = addMatrix(

subtractMatrix(addMatrix(P, S, P.length), T, T.length), V, V.length);

int[][] C12 = addMatrix(R, T, R.length);

int[][] C21 = addMatrix(Q, S, Q.length);

int[][] C22 = addMatrix(

subtractMatrix(addMatrix(P, R, P.length), Q, Q.length), U, U.length);

constructMatrix(C11, C, 0, 0);

constructMatrix(C12, C, 0, n / 2);

constructMatrix(C21, C, n / 2, 0);

constructMatrix(C22, C, n / 2, n / 2);

}

}

f. Generate random matrix with repetitions

|  |  |
| --- | --- |
| Name: | fillMatrix |
| Description: | Fills a matrix with random values that can repeat. |
| Input: | None |
| Output: | The filled matrix. |

private void fillMatrix() {

int max =100;

if(type.equals(*COEFFICIENTS\_MATRIX*))

max = 5;

for (int i = 0; i < rows; i++) {

for (int j = 0; j < columns; j++) {

int num = (int) (Math.*random*()\*max)+1;

matrix[i][j] = num;

}

}

}

UNIT TEST DESIGN

Stages configuration.

|  |  |  |
| --- | --- | --- |
| **NAME** | **CLASS** | **STAGE** |
| stage1 | BoardTest |  |

Test cases design.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** check the correct multiplication of matrices (standard case) using the iterative algorithm. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| Board | +multiplyMatricesFirstOption () | Stage1 | None. | The method multiplies correctly the matrices in the arraylist of matrices. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** check the correct multiplication of big matrices (limit case) using the iterative algorithm. In this test, the identity matrix is used. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| Board | +multiplyMatricesFirstOption () | Stage1 | None. | The method multiplies correctly the matrices in the arraylist. Although it takes some time because of the size of the matrices. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check the correct multiplication of triangular matrices (interesting case) using the iterative algorithm. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| Board | +multiplyMatricesFirstOption () | Stage1 | None. | The method returns, as expected, a triangular matrix as result. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check the correct multiplication of regular matrices (standard case) using the Divide and Conquer algorithm. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| Board | +multiplyMatricesSecondOption () | Stage1 | None. | The method multiplies successfully square matrices, that is, matrices of N x N size. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check the correct multiplication of big square matrices (limit case) using the Divide and Conquer algorithm. In this test, the matrix of identity is used. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| Board | +multiplyMatricesSecondOption () | Stage1 | None. | The method returns the correct result for big square matrices. However, it takes an important amount of time to do that (about 50 seconds). |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check the correct multiplication of triangular matrices (interesting case) using the Divide and Conquer algorithm. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| Board | +multiplyMatricesSecondOption () | Stage1 | None. | The method returns a triangular matrix as a result of the multiplication. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check the correct multiplication of regular square matrices (standard case) using the Strassen algorithm. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| Board | +multiplyMatricesThirdOption () | Stage1 | None. | The method multiplies correctly the square matrices in the arraylist of matrices. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check the correct multiplication of big square matrices (limit case) using the Strassen algorithm. In this test, the identity matrix is used. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| Board | +multiplyMatricesThirdOption () | Stage1 | None. | The method returns the correct result of multiplying big square matrices. However, it takes an important amount of time (about 30 seconds) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Check the correct multiplication of triangular matrices (interesting case) using the Strassen algorithm. | | | | |
| **Class** | **Method** | **Stage** | **Input** | **Output** |
| Board | +multiplyMatricesThirdOption () | Stage1 | None. | The method returns, as expected, a triangular matrix as result. |