**Especificación de requerimientos funcionales**

|  |  |
| --- | --- |
| **Nombre** | Ingresar los valores de la cantidad de filas y columnas de las dos matrices a multiplicar |
| **Resumen** | permita ingresar los valores de la cantidad de filas y columnas de las dos matrices a multiplicar (tablero de batalla) |
| **Entradas** | |
| Valores para las filas y las columnas | |
| **Resultados** | |
| Se recibe los datos y almacenan correctamente | |

|  |  |
| --- | --- |
| **Nombre** | Generar aleatoriamente los valores de las posiciones de cada matriz |
| **Resumen** | Permite generar aleatoriamente los valores de las posiciones de cada matriz e indica si los números a generar deben ser todos diferentes o pueden haber repetidos. |
| **Entradas** | |
| Identificador que indica si pueden ser diferentes o si puede repetirse. | |
| **Resultados** | |
| Los valores aleatorios indicados son generados correctamente. | |

|  |  |
| --- | --- |
| **Nombre** | Generar una lista aleatoria de matrices |
| **Resumen** | Permita generar una lista aleatoria de matrices para posteriormente ser multiplicadas. |
| **Entradas** | |
| Número de matrices que contiene la lista. | |
| **Resultados** | |
| Se crea la lista de matrices éxitosamente. | |

|  |  |
| --- | --- |
| **Nombre** | **Mostrar las posiciones de las naves marcianas** |
| **Resumen** | Muestra el resultado de la multiplicación con las posiciones exactas de las tropas de Marte. |
| **Entradas** | |
|  | |
| **Resultados** | |
| Se muestra el resultado de la multiplicación realizada. | |

**Pseudcódigo de los algoritmos más relevantes**

**Función para computar el producto de dos matrices cuadradas de tamaño n**

Matrix matrixMultStrassen(Matrix A, Matrix B)

{

    // Encontrar la dimensión de las matrices (Todas son las mismas)

    int n = A.n;

    // Declarar la nueva matriz que almacena el producto de las matrices A y B

    Matrix C = init\_matrix(n);

    for (int i = 0; i < n; ++i)

    {

        for (int j = 0; j < n; ++j)

        {

            int sum = 0;

            for (int k = 0; k < n; ++k)

            {

                sum = sum + A.content[i][k] \* B.content[k][j];

            }

            C.content[i][j] = sum;

        }

    }

    return C;

}

**Clase matriz**

struct Matrix

{

    int n;

    int [#][#]content;

};

// Función para inicializar la matriz

Matrix init\_matrix(int n)

{

    // Declarar una nueva matriz

    Matrix matrix;

    // Inicializar el tamaño de la matriz

    matrix.n = n;

    // Establecer el tamaño del contenido de la matriz

    matrix.content = new int [#][n];

    for (int i = 0; i < n; ++i)

    {

        matrix.content[i] = new int[n];

    }

    // Inicializar todas las entradas a 0

    for (int i = 0; i < n; ++i)

    {

        for (int j = 0; j < n; ++j)

        {

            matrix.content[i][j] = 0;

        }

    }

    return matrix;

}

**Identificar números primos en una matriz**

function isStrongPseudoprime(n, a)

d := n - 1; s := 0

while d % 2 == 0

d := d / 2

s := s + 1

t := powerMod(a, d, n)

if t == 1 return ProbablyPrime

while s > 0

if t == n - 1

return ProbablyPrime

t := (t \* t) % n

s := s - 1

return Composite

function isPrime(n)

for i from 1 to k

a := randInt(2, n-1)

if isStrongPseudoprime(n, a) == Composite

return Composite

return ProbablyPrime

function powerMod(b, e, m)

x := 1

while e > 0

if e % 2 == 1

x := (b \* x) % m

b := (b \* b) % m

e := e // 2 # integer division

return x

**Diseño de casos de prueba**

|  |  |  |
| --- | --- | --- |
| **Nombre** | **Clase** | **Escenario** |
| setupScenary1 | Board | *vacío* |
| setupScenary2 | Board | C:\Users\JuanDiego\Downloads\Diagrama en blanco (10).png |