Por: Diego Alejandro Angarita Arboleda, Diego Andres Aza

Ejercicio 1 Multiplicación de matrices

Para este ejercicio se realizó un programa que realiza la multiplicación de 2 matrices, usando la forma tradicional (i-j-k), la forma optimizada para row major (i-k-j) y la división de las matrices en submatrices (solo para matrices cuadradas).

Decisiones de diseño

- Ya que el algoritmo tiene como objetivo, realizar multiplicaciones de matrices AxB, sería innecesario crear variables para filas de A, columnas de A, filas de B y columnas de B ya que para realizar la multiplicación entre matrices, el número columnas de A deben ser igual al número de filas de B, por lo tanto, los 2 valores pueden ser asignados a una sola variable.
- Con el fin de hacer la multiplicación de matrices usando el método de Strassen con un código más fácil de entender, se usará la siguiente forma:

$$
\begin{aligned}
P_1 &= A_{11} \cdot (B_{12} - B_{22}) \\
P_2 &= (A_{11} + A_{12}) \cdot B_{22} \\
P_3 &= (A_{21} + A_{22}) \cdot B_{11} \\
P_4 &= A_{22} \cdot (B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
P_6 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\
P_7 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12})
\end{aligned}
$$

$$
\begin{aligned}
P_5 + P_4 - P_2 + P_6 &= C_{11} \\
P_1 + P_2 &= C_{12} \\
P_3 + P_4 &= C_{21} \\
P_5 + P_1 - P_3 - P_7 &= C_{22}
\end{aligned}
$$

- Se crean algunos parámetros al empezar como:
  -Umbral (para Strassen).
  -Las matrices A y B que se multiplicaran.
  -Las matrices CBase, C_ikj y C_strassen (esta será una matriz dinámica con el fin de evitar problemas de espacio).

```
const int maxSize = 1024;
const int umbral = 64;

int A[maxSize][maxSize], B[maxSize][maxSize], CBase[maxSize][maxSize], C_ikj[maxSize][maxSize];
int** C_strassen = nullptr;
```

- Funcion para generar números al azar
  -Se crea una función para generar números entre un intervalo escogido, usando siempre la misma semilla.

- También se hizo un algoritmo que evita que el programa deje de funcionar si el usuario ingresa datos incorrectos

-

```cpp
int m, p, n;
while (true) {
    try {
        cout << "ingrese el numero de filas de A (m): ";
        cin >> m;
        if (cin.fail()) throw runtime_error("debe ingresar un numero entero.");
        cout << "ingrese el numero de columnas de A / filas de B (p): ";
        cin >> p;
        if (cin.fail()) throw runtime_error("debe ingresar un numero entero.");
        cout << "ingrese el numero de columnas de B (n): ";
        cin >> n;
        if (cin.fail()) throw runtime_error("debe ingresar un numero entero.");

        if (m <= 0 || m > maxSize || p <= 0 || p > maxSize || n <= 0 || n > maxSize) {
            cout << "error: El numero de filas y columnas debe estar entre 1 y " << maxSize << "." << end
            continue;
        }
        break;
    } catch (const exception& e) {
        cout << "Error: " << e.what() << endl;
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
    }
}
```

-Multiplicación tradicional (i-j-k)

```cpp
void matmul_base(int A[maxSize][maxSize], int B[maxSize][maxSize], int C[maxSize][maxSize], int m, int p, int n) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][0] * B[0][j];
            for (int k = 1; k < p; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Esta función recorre las filas de A y las columnas de B para calcular cada elemento de de la matriz C resultante.

-Multiplicacion I-K-J

```cpp
void matmul_ikj(int A[maxSize][maxSize], int B[maxSize][maxSize], int C[maxSize][maxSize], int m, int p, int n) {
    // Inicializar la primera iteración sin suma
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][0] * B[0][j];
        }
    }
    // Continuar con las iteraciones restantes
    for (int i = 0; i < m; i++) {
        for (int k = 1; k < p; k++) {
            for (int j = 0; j < n; j++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Esta función nos permite calcular el producto de 2 matrices de forma más eficiente debido a que en c++ la información de las matrices se guardan por filas, así que al acceder a la matriz A con con los iteradores i y k de forma consecutiva permite acceder a cada elemento de las filas y columnas de forma contigua en la memoria. Aunque por esta misma razón el rendimiento para recorrer la matriz B empeora pero la mejora en ;la matriz A es suficiente para compensar esto.

-Multiplicacion por bloques

Primero definimos un Umbral para definir cuándo usar la multiplicación base o dividir las matrices en submatrices, ya que si el tamaño de las matrices no supera el umbral, la multiplicación por bloques tendrá peor rendimiento.
También es más fácil definir las matrices como matrices dinámicas usando punteros dobles para evitar problemas al separar y volver a unir matrices.
Para realizar la multiplicación por bloques necesitar
emos algunas funciones auxiliares:

-Una función para crear matrices dinámicas (no densas):

```
int** mat_din(int size) {
    int** mat = new int*[size];
    for (int i = 0; i < size; i++) {
        mat[i] = new int[size];
    }
    return mat;
}
```

-funciones para sumar y restar matrices dinámicas:

```
int** sum_mat(int** A, int** B, int size) {
    int** C = new int*[size];
    for (int i = 0; i < size; i++) {
        C[i] = new int[size];
        for (int j = 0; j < size; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    return C;
}

int** sub_mat(int** A, int** B, int size) {
    int** C = new int*[size];
    for (int i = 0; i < size; i++) {
        C[i] = new int[size];
        for (int j = 0; j < size; j++) {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
    return C;
}
```

-Una función para copiar una matriz dinámica:

```
int** sum_mat(int** A, int** B, int size) {
    int** C = new int*[size];
    for (int i = 0; i < size; i++) {
        C[i] = new int[size];
        for (int j = 0; j < size; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    return C;
}

int** sub_mat(int** A, int** B, int size) {
    int** C = new int*[size];
    for (int i = 0; i < size; i++) {
        C[i] = new int[size];
        for (int j = 0; j < size; j++) {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
    return C;
}
```

Esto nos permitirá operar sobre cada submatriz.

-Una función para reintegrar submatrices:

```
int** reintegrar_mat(int** submat, int** mat, int row_start, int col_start, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            mat[row_start + i][col_start + j] = submat[i][j];
        }
    }
    return mat;      You, yesterday • Blas innescesarias eliminadas, y trabajo parcia…
}
```

Esta función nos permitirá unir las 4 submatrices resultantes en una sola.
-Una función para eliminar matrices dinámicas:

```
void eliminar_mat(int** mat, int size) {
    for (int i = 0; i < size; i++) {
        delete[] mat[i];
    }
    delete[] mat;
}
```

ya que estas matrices se guardarán en el Heap, es necesario borrarlas manualmente.

- Pruebas y resultados:
Vamos a probar con matrices cuadradas de tamaños 256, 512, 768 y 1024

promedios de resultados con 256:
Multiplicación ijk = (0,187522 + 0,109762 + 0,109626)/3 = 0,1356366
Multiplicación ikj = (0,0941537 + 0,0980294 + 0,0930938)/3 = 0,0950923

Multiplicación por Bloques = (0,0973329 + 0,114761 + 0,0977904)/3 = 0,103294766

promedios de resultados con 512:
Multiplicación ijk = (1,21618 + 1,02289 + 0,917119)/3 = 1,052063
Multiplicación ikj = (0,727246 + 0,711946 + 0,714534)/3 = 0,71790866
Multiplicación por Bloques = (0,682012 + 0,674536 + 0,680701)/3 = 0,679083

promedios de resultados con 768:
Multiplicación ijk = (4,94427 + 4,80409 + 4,78049)/3 = 4,84295
Multiplicación ikj = (2,78900 + 2,37016 + 2,40037)/3 = 2,5198433
Multiplicación por Bloques = (2,87308 + 2,15176 + 2,137009)/3 = 2,387283

promedios de resultados con 1024:
Multiplicación ijk = (11,8144 + 11,8865 + 11,9043)/3 = 11,8684
Multiplicación ikj = (5,57419 + 5,58323 + 5,63223)/3 = 5,59655
Multiplicación por Bloques = (6,3975 + 6,83602 + 5,21186)/3 = 6,14846

Con todas las pruebas, podemos ver que la multiplicación por bloques no es la más efectiva con matrices muy pequeñas o muy grandes.

Ejercicio 2: Análisis de BST

Para este ejercicio se realizó un programa con un menú que permite a usuario crear u n BST con los números que ingrese, buscar un elemento en ese BST, hacer recorridos preorden, inorden, postorden, tambien podran saber la altura máxima y el grado del BST, también podrán crear Rápidamente un BST con inserciones o aleatorias y comparar estas.

Decisiones de diseño:

- Para realizar este ejercicio se optó por usar el paradigma de programación orientada a objetos (POO), esto con el fin de guardar todas las funciones de os BST en una estructura (clase), para luego poder crear BST'S de forma más sencilla.

- Se usó la siguiente estructura para el BST incluyendo un contador para los números repetidos que se inserten, ya que estos no serán tomados en el BST.

```
You, 3 days ago | 1 author (You)
struct Nodo{
    int key;
    Nodo* left;
    Nodo* right;
    int contador;

    Nodo(int k) : key(k), left(nullptr), right(nullptr), contador(1) {};
};
```

Funciones:

- Insertar elemento(insert):
  Esta función se usa para insertar elementos en un BST, sin importar si había elementos en el BST (se vuelve raíz), el número ya existe en el BST (solo aumenta el contador) o si debe insertar el número a la derecha o a la izquierda para que el BST no pierda su condición de ser AVL.

```
Nodo* insert(Nodo* raiz, int key) {
    if (raiz == nullptr) {
        return new Nodo(key);
    }

    if (key == raiz->key) {
        raiz->contador++;
        return raiz;
    }

    if(key < raiz->key) {
        raiz->left = insert(raiz->left, key);
    } else {
        raiz->right = insert(raiz->right, key);
    }

    return raiz;
}
```

- Buscar elemento(find):
  Esta función usará la búsqueda binaria para buscar un elemento en un BST, y usará un vector para guardar los elementos que recorre hasta llegar al elemento que se busca, solo si este existe en el BST.

```
bool find(Nodo* raiz, int key, vector<int>& camino) {
    if (raiz == nullptr) {
        return false;
    }

    camino.push_back(raiz->key);

    if(key == raiz->key) {
      return true;
    }

    if(key < raiz->key) {
        return find(raiz->left, key, camino);
    } else {
        return find(raiz->right, key, camino);
    }
}
```

- Funciones para recorrer el BST:
  -Recorrido preorden:
  Este recorrido empezará con el nodo raíz, luego todo el subárbol izquierdo y por último el subárbol derecho.

```
void recorridoPreOrder(Nodo* raiz){
    if(raiz != nullptr){
        cout << raiz->key << " ";
        recorridoPreOrder(raiz->left);
        recorridoPreOrder(raiz->right);
    }
}       You, 3 days ago • Blas innescesarias
```

  -Recorrido inorden:
  Este recorrido empezará con el subárbol izquierdo, luego el nodo raíz y por último el subárbol izquierdo.

```
void recorridoInOrder(Nodo* raiz){
    if(raiz != nullptr){
        recorridoInOrder(raiz->left);
        cout << raiz->key << " ";
        recorridoInOrder(raiz->right);
    }
}
```

-Recorrido postorden:
Este recorrido empieza con el subárbol izquierdo, luego el subárbol derecho y por último, el nodo raíz.

```
void recorridoPostOrder(Nodo* raiz){
    if(raiz != nullptr){
        recorridoPostOrder(raiz->left);
        recorridoPostOrder(raiz->right);
        cout << raiz->key << " ";
    }
}
```

- Obtener la altura y el grado máximo del árbol:
  Para obtener la altura se creó un algoritmo que recorre todos los los subárboles izquierdos y derechos que se guardan en una variable entera que aumenta en uno por cada nodo que se recorre, luego devuelve el valor más grande.

```
int getAltura(Nodo* raiz) {
    if (raiz == nullptr) {
        return 0;
    }
    int alturaIzq = getAltura(raiz->left);
    int alturaDer = getAltura(raiz->right);
    return 1 + max(alturaIzq, alturaDer);
}
```

  Para obtener el grado máximo del árbol, se creó una función que recorre todos los subárboles izquierdos y derechos, por cada nodo que se visite, se verificará si el nodo tiene hijos, luego retornara la mayor cantidad la mayor cantidad de nodos hijos que tenga un nodo (Para un BST nunca será más de 2).

```
int getGradoMax(Nodo* raiz) {
    if (raiz == nullptr) {
        return 0;
    }
    int grado = 0;
    if (raiz->left != nullptr) grado++;
    if (raiz->right != nullptr) grado++;
    int gradoIzq = getGradoMax(raiz->left);
    int gradoDer = getGradoMax(raiz->right);
    return max(grado, max(gradoIzq, gradoDer));
}
```

- Crear un caso degenerado(ordenado):
  Se creó una función que elimina un BST si es que existe alguno, luego crea un BST con número consecutivos desde 1 hasta n (n es dado por el usuario).

```
void crearCasoDegenerado(int n) {
    limpiar();
    cout << "Creando BST degenerado con " << n << " elementos ordenados..." << endl;
    for (int i = 1; i <= n; i++) {
        insert(i);
    }
    cout << "BST degenerado creado. Altura: " << height() << endl;
}
```

- Crear un caso aleatorio:
  se creo una funcion que elimina un BST si es que existe alguno, luego se crea un vector que almacena números producidos aleatoriamente para luego insertarlos en un BST.

```
    void crearCasoAleatorio(int n) {
        limpiar();
        cout << "Creando BST aleatorio con " << n << " elementos..." << endl;

        // Crear vector con números del 1 al n
        vector<int> numeros;
        for (int i = 1; i <= n; i++) {
            numeros.push_back(i);
        }

        // Mezclar aleatoriamente
        random_device rd;
        mt19937 gen(rd());
        shuffle(numeros.begin(), numeros.end(), gen);

        // Insertar en orden aleatorio
        for (int num : numeros) {
            insert(num);
        }
        cout << "BST aleatorio creado. Altura: " << height() << endl;
        cout << "Grado maximo del BST>" << gradoMax() << endl;
    }
};
```

- Comparar alturas(degenerado vs aleatorio)

  Se creó una función que crea 2 BST's (uno para el caso aleatorio y otro para el caso degenerado), para luego compararlos como se ve en la imagen.

```
void compararAlturas() {
    cout << "\nComparacion de alturas" << endl;
    cout << "Ingrese el numero de elementos para la comparacion: ";
    int n;
    cin >> n;

    if (n <= 0) {
        cout << "El numero debe ser positivo." << endl;
        return;
    }

    BST arbolDegenerado, arbolAleatorio;

    // Crear caso degenerado
    cout << "\nCreando caso degenerado..." << endl;
    arbolDegenerado.crearCasoDegenerado(n);      You, 16 minutes ago • Uncommitted changes
    int alturaDegenerada = arbolDegenerado.height();

    // Crear caso aleatorio
    cout << "Creando caso aleatorio..." << endl;
    arbolAleatorio.crearCasoAleatorio(n);
    int alturaAleatoria = arbolAleatorio.height();

    // Mostrar comparacion
    cout << "\nResultados de comparacion" << endl;
    cout << "Numero de elementos: " << n << endl;
    cout << "Altura del BST degenerado (ordenado): " << alturaDegenerada << endl;
    cout << "Altura del BST aleatorio: " << alturaAleatoria << endl;
    cout << "Diferencia de altura: " << (alturaDegenerada - alturaAleatoria) << endl;
    cout << "Altura teorica minima (log2n): " << (int)ceil(log2(n + 1)) << endl;
    cout << "Altura teorica maxima: " << n << endl;
}
```

Funciones auxiliares

- borrar Árbol:
  esta función elimina todos los nodos de un BST ,esto se debe hacer, ya que los BST se almacenarán en la memoria.

```
void borrarArbol(Nodo* raiz){
    if(raiz ≠ nullptr){
        borrarArbol(raiz→left);
        borrarArbol(raiz→right);
        delete raiz;
    }
}
```

- estaVacio:
  esta función verifica si un BSt está vacío.

```
bool estaVacio() {
    return raiz == nullptr;
}
```

- borrarBST:
  Esta función nos permitirá Borrar el BST creado en el menú y en la función para comparar el caso degenerado y el caso aleatorio.

- imprimirVisitados:
  esta función nos permite imprimir los elemento de un vector, se usa para ver el camino -> raíz de un BST.

Usos de IA

1. mediante el prompt “haz un try catch en el case 1 para que el programa no termine al ingresar un dato que no sea int ” en claude 4.
2. mediante el prompt “como puedo realizar la operación log2(n +1)” en claude 4.