TEMA 2: ALGORITMOS SOBRE MATRICES

ALGORITMOS Y ESTRUCTURAS DE DATOS

M. Colebrook Santamaría
J. Riera Ledesma

Objetivos

- Comparación de números enteros y de números reales
- Recorrido de un vector. Búsqueda secuencial de un elemento Contabilización del número de ocurrencias de un elemento dentro de un vector. Búsqueda del menor y mayor elemento
- Suma de los elementos (todos, pares, impares) de un vector
- Producto escalar de vectores
- Implementación de una matriz sobre un vector
- Recorrido de una matriz por filas y columnas
- Suma de los elementos de la matriz. Suma de los elementos tal que la suma índices es par
- Recorrido de la diagonal principal y secundaria. Recorrido de la submatriz triangular inferior y superior
- Suma y producto de matrices
- Obtención de submatrices
- Representación y multiplicación de matrices escasas

Comparación de números enteros

- La comparación de números enteros (`short`, `int`, `long`) en C++ es directa usando los siguientes operadores:

```
==   !=   <   >   <=   >=
```

Comparación de números reales

- La comparación de números reales en C++ no es tan directa como la de números enteros, debido a la precisión. Por ejemplo:

```
1/3.0 != 0.33333333
```

- Por ello, se debe realizar la siguiente operación:

```cpp
#include <cmath>

const double EPS = 1e-6;

bool igual(const double a, const double b,
           const double epsilon = EPS)
{ // devuelve verdadero si a y b son iguales
  return fabs(a - b) < epsilon; // |a - b| < ϵ
}
```

Recorrido de un vector

- Dado un vector estático o dinámico, se puede recorrer fácilmente usando un bucle **for**:

```
const int n = 10;
int A[n]; // int *A = new int[n];

for (int i = 0; i < n; i++)
  A[i] = i;

cout << "A: ";
for (int i = 0; i < n; i++)
  cout << A[i] << " ";
cout << endl;
//delete[] A;
```

Búsqueda secuencial de un elemento

- Para buscar un elemento dentro de un vector, deberemos recorrer todos los elementos hasta encontrarlo. Si tomamos el vector A definido en la diapositiva anterior:

```
int v = 7;  // valor a buscar
bool encontrado = false;

for (int i = 0; i < n && !encontrado; i++)
  if (A[i] == v)
    encontrado = true;

cout << "Valor " << v << (encontrado ? " " : " no ")
     << "encontrado." << endl;
```

Búsqueda secuencial de un elemento en un vector ordenado

- En este caso, deberemos recorrer todos los elementos hasta encontrar un elemento mayor o alcanzar el final del vector.

```
int ele = 6;  // valor a buscar
bool encontrado;
int v[]={1,3,5,34,67,82,332};
int tam=7;


for(i=0;i<tam && V[i]<ele;i++);
encontrado=(V[i]==ele) ?  true :  false;


   cout << "Valor " << v << (encontrado ? " " : " no
")<< "encontrado." << endl;
```

Contabilización de las ocurrencias de un elemento dentro de un vector

- Para contar el número de ocurrencias de un elemento en un vector debemos declarar un **contador**.

```
const int n = 10;
int A[n] = { 5, 3, 2, 1, 5, 1, 4, 2, 1, 6 };
int v = 1, contador = 0;

for (int i = 0; i < n; i++)
  if (A[i] == v)
    contador++;

cout << "Valor " << v << " encontrado " << contador
     << " veces." << endl;
```

Búsqueda del menor/mayor elemento

- Para esta búsqueda en un vector desordenado, necesitamos una variable **min/max** que nos permita comparar los valores de las celdas.

```
const int n = 10;
int A[n] = { 5, 3, 2, 1, 5, 1, 4, 2, 1, 6 };
// INFINITY definida en cmath
int min = INFINITY , max = -INFINITY;

for (int i = 0; i < n; i++) {
  if (A[i] < min) min = A[i];
  if (A[i] > max) max = A[i];
}

cout << "min = " << min << endl;
cout << "max = " << max << endl;
```

Suma de los elementos de un vector

- Para realizar la suma de todos los elementos del vector, necesitamos una variable **suma** que vaya acumulando todos y cada uno de los valores del vector.

```
const int n = 10;
int A[n] = { 5, 3, 2, 1, 5, 1, 4, 2, 1, 6 };
int suma = 0;

for (int i = 0; i < n; i++)
  suma += A[i];

cout << "suma = " << suma << endl;
```

Suma de los elementos en posiciones pares/impares de un vector

- En la suma de los elementos en posiciones **pares/impares**, necesitamos comprobar si el **índice del elemento** es par/impar.

```
const int n = 10;
int A[n] = { 5, 3, 2, 1, 5, 1, 4, 2, 1, 6 };
int suma_pares = 0, suma_impares = 0;

for (int i = 0; i < n; i++)
  if (i % 2 == 0) suma_pares += A[i];
  else            suma_impares += A[i];

cout << "Suma pares = " << suma_pares << endl;
cout << "Suma impares = " << suma_impares << endl;
```

Producto escalar de dos vectores

- El producto escalar de dos vectores (ver Wikipedia) se define como:

$$\mathbf{A} \cdot \mathbf{B} = (a_1, a_2, a_3, ..., a_n) \cdot (b_1, b_2, b_3, ..., b_n) = a_1 b_1 + a_2 b_2 + ... a_n b_n = \sum a_i \cdot b_i$$

```
const int n = 3;
int A[n] = { 5, 3, 2 }, B[n] = { 1, 5, 1 };
int producto_escalar = 0;

for (int i = 0; i < n; i++)
  producto_escalar += A[i] * B[i];

cout << "Producto escalar = " << producto_escalar
     << endl;
```

Implementación de una matriz sobre un vector (1)

- La implementación de una matriz de ***m* filas** y ***n* columnas** sobre un vector supone reservar la memoria total (***m* * *n***) y acceder usando una fórmula en base a la fila y la columna:

```
// m: filas, n: columnas
const int m = 2, n = 3;

int *M = new int [m * n];
for (int i = 0; i < m * n; i++)
  M[i] = (i + 1) * 10;

//         0   1   2   3   4   5
// M = { 10, 20, 30, 40, 50, 60 }
// pero realmente queremos tratar a M como:
//             1   2   3
// M = { 1: { 10, 20, 30 },
//       2: { 40, 50, 60 } }
// ...
```

Implementación de una matriz sobre un vector (2)

Run C++

```
//         0   1   2   3   4   5
// M = { 10, 20, 30, 40, 50, 60 }
// pero realmente queremos tratar a M como:
//             1   2   3
// M = { 1: { 10, 20, 30 },
//       2: { 40, 50, 60 } }

// M se puede indexar con fila i y columna j usando:
// pos = (i - 1) * n + (j - 1);

int i = 1, j = 2, pos = (i - 1) * n + (j - 1);
cout << "M[" << i << "," << j << "] = M[" << pos << "] = " << M[pos]
     << endl; // M[1,2] = M[1] = 20

i = 2, j = 3, pos = (i - 1) * n + (j - 1);
cout << "M[" << i << "," << j << "] = M[" << pos << "] = " << M[pos]
     << endl; // M[2,3] = M[5] = 60


delete[] M;
```

Recorrido de una matriz por filas/columnas

- El recorrido de una matriz por filas y columnas necesita dos iteradores.

```
// filas m, columnas n
const int m = 3, n = 4;
int A[m][n] = { { 4, 5, 2, 9 },
                { 1, 7, 4, 2 },
                { 8, 6, 2, 0 } };

for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++)
    cout << A[i][j] << " ";
  cout << endl;
}
```

```
4 5 2 9
1 7 4 2
8 6 2 0
```

Suma de los elementos de una matriz

- Con el recorrido anterior, es muy sencillo obtener la suma total de todos los elementos de la matriz:

```
int suma = 0;
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++)
    suma += A[i][j];
}
// suma= 50
cout << "suma = " << suma << endl;
```

Suma de los elementos tal que la suma de índices es par

- En este caso, habría que aplicar un filtro de forma que la suma total sería solo para aquellos índices **fila y columna** cuya **suma sea par**.

```
suma = 0;
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++)
    if ((i + j) % 2 == 0)
      suma += A[i][j];
}
// suma= 25
cout << "suma = " << suma << endl;
```

Recorrido de la diagonal principal/secundaria (1)

- La diagonal principal de una matriz de una **matriz cuadrada** son todos los elementos cuya **índice de fila *i*** es igual al **índice de columna *j***, mientras que la **diagonal secundaria** es la secuencia de elementos (*i*, *j*) de la matriz tal que $j = m - 1 - i$, con $0 \leq i < m$.

```
const int m = 3;
int B[m][m] = { { 4, 5, 9 },
                { 1, 7, 3 },
                { 8, 6, 2 } };

cout << "Elementos de la diagonal principal: ";
for (int i = 0; i < m; i++)
  cout << B[i][i] << " ";
cout << endl; // 4 7 2

cout << "Elementos de la diagonal secundaria: ";
for (int i = 0; i < m; i++)
  cout << B[i][m - 1 - i] << " ";
cout << endl; // 9 7 8
```

Recorrido de la diagonal principal/secundaria (2)

- Otras dos formas de obtener la **diagonal secundaria**:

```
cout << "Elementos de la diagonal secundaria: ";
for (int j = m - 1; j >= 0; j--)
  cout << B[m - 1 - j][j] << " ";
cout << endl; // 9 7 8

cout << "Elementos de la diagonal secundaria: ";
for (int i = 0, j = m - 1; i < m; i++, j--)
  cout << B[i][j] << " ";
cout << endl; // 9 7 8
```

Recorrido de la submatriz triangular inferior/superior

- Dada un matriz cuadrada, la **submatriz triangular inferior** son todos aquellos elementos (*i*, *j*) de la matriz que cumplan 0 <= *j* <= *i*.
- La **matriz triangular superior** serán todos los elementos que cumplan *i* <= *j* < *m*.

```
cout << "Elementos de la submatriz triangular inferior: " << endl;
for (int i = 0; i < m; i++) {
  for (int j = 0; j <= i; j++)
    cout << B[i][j] << " ";
  cout << endl;
}
```

```
4
1 7
8 6 2
```

```
cout << "Elementos de la submatriz triangular superior: " << endl;
for (int i = 0; i < m; i++) {
  for (int j = i; j < m; j++)
    cout << B[i][j] << " ";
  cout << endl;
}
```

```
4 5 9
  7 3
    2
```

- **Ejercicio**: hacer recorrido de submatrices triangulares con respecto a la diagonal secundaria.

Obtención de submatrices

- Una vez vistas las submatrices triangulares inferiores y superiores, se pueden realizar las siguientes operaciones sobre ellas:
  - Recuento del número de elementos.
  - Suma de todos los elementos.
  - Búsqueda de un valor.
  - Búsqueda del valor mínimo o máximo.

Suma de matrices

- La única condición para la suma de dos matrices es que las dimensiones de ambas deben ser iguales.

```
const int m = 2, n = 3;
int A[m][n] = { { 4, 5, 9 }, { 1, 7, 3 } },
    B[m][n] = { { 8, 6, 2 }, { 3, 6, 1 } },
    C[m][n];

for (int i = 0; i < m; i++)
  for (int j = 0; j < n; j++)
    C[i][j] = A[i][j] + B[i][j];

for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++)
    cout << C[i][j] << " ";
  cout << endl;
}
```

```
12 11 11
4  13 4
```

Producto de matrices

- Dos matrices **A[m][n]** y **B[n][p]** se pueden multiplicar al tener la matriz **A** el nº de columnas igual al nº de filas de **B**. El resultado es una matriz **C[m][p]**.

```
const int m = 2, n = 3, p = 2;
int A[m][n] = { { 4, 5, 9 }, { 1, 7, 3 } },
    B[n][p] = { { 8, 6 }, { 2, 3 }, { 6, 1 } },
    C[m][p];

for (int i = 0; i < m; i++)
  for (int j = 0; j < p; j++) {
    C[i][j] = 0;
    for (int k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
  }

for (int i = 0; i < m; i++) {
  for (int j = 0; j < p; j++)
    cout << C[i][j] << " ";
  cout << endl;
}
```

```
|4 5 9|   |8 6|
|1 7 3|   |2 3|
          |6 1|
```

```
|96 48|
|40 30|
```

Representación de vectores escasos (1)

- Un **vector escaso (*sparse vector*)** es una representación eficiente de un vector en el que la mayoría de valores son nulos, por lo que solo se almacenan los elementos distintos de cero. Por ejemplo:

```
// vector normal con 6 elementos
n: 6
0.0000   0.0000   5.0000   6.0000   0.0000   4.0000
     0        1        2        3        4        5

// vector escaso (sparse vector)
n: 6
nz: 3
5.0000   6.0000   4.0000
     2        3        5
```

Representación de vectores escasos (2)

- Por tanto, para desarrollar una clase que almacene un vector escaso necesitaremos los siguientes atributos:

```
double* val_;
int*    inx_;
int     nz_;
int     n_;
```

- Los métodos a desarrollar serían:
  - Constructor
  - Constructor de copia
  - Destructor
  - Entrada/Salida
  - Operaciones

Producto escalar entre un vector denso y un vector disperso

- El algoritmo del producto escalar entre un vector normal (denso) **v** y un vector disperso **sv**, se puede implementar de la siguiente forma:

```
double producto = 0;
for (int i = 0; i < nz_; i++)
  producto += val_[i] * v[inx_[i]];
```

Plantillas (`template`) (1)

- Las plantillas (**`template`**) permiten que una clase, función o variable trabaje con diferentes tipos de datos sin tener que reescribir el código para cada tipo. Por ejemplo:

```
template<class T>
void intercambio(T& a, T& b)
{ T tmp = a;
  a = b;
  b = tmp;
}

int a = 1, b = 2;
intercambio<int>(a, b);
cout << "a= " << a << ", b= " << b << endl;
// a= 2, b= 1
```

Plantillas (`template`) (2)

- Las plantillas de clases (***class templates***) permiten que los miembros puedan usar los parámetros de la template como tipos.

```
template<class T>
class punto_t
{ T x, y;
 public:
  punto_t(const T a, const T b): x(a), y(b) {}
  ostream& write(ostream& os)
  { return os << "x= " << x << ", y= " << y << endl; }
};

punto_t<int> i(3, 4);
i.write(cout); // x= 3, y= 4

punto_t<float> f(1.23, 5.67);
f.write(cout); // x= 1.23, y= 5.67

punto_t<double> d(2.345, 6.789);
d.write(cout); // x= 2.345, y= 6.789
```

Referencias

- Olsson, M. (2015), “C++ 14 Quick Syntax Reference”, Apress. Disponible en PDF en la BBTK-ULL: absysnetweb.bbtk.ull.es/cgi-bin/abnetopac01?TITN=533049
- Stroustrup, B. (2002), “El Lenguaje de Programación C++”, Addison Wesley.
- C++ Syntax Highlighting (código en colores): tohtml.com/cpp