Programación con GPUs

Teórica 02 - Algoritmos y Estructura de Datos

Segundo Semestre, 2025

Contents

2.1 Introducción . . . 3
2.2 Números . . . 3
2.2.1 Enteros . . . 3
2.2.2 Aritmética modular . . . 4
2.2.3 Flotantes . . . 4
2.3 Macros . . . 4
2.4 Manipulación de bits . . . 5
2.5 Operaciones con bits . . . 5
2.6 Representación de Conjuntos . . . 6
2.7 Eficiencia algorítmica . . . 7
2.8 Soluciones a los algoritmos . . . 8
2.9 Teoría de Números . . . 9
2.10 Números primos . . . 9
2.10.1 Búsqueda lineal . . . 9
2.10.2 Búsqueda lineal hasta $\sqrt{n}$ . . . 9
2.10.3 Criba de Eratóstenes . . . 10
2.11 Vectores . . . 11
2.12 Matrices . . . 12
2.12.1 Acceso a los elementos de una matriz . . . 12
2.12.2 Suma de matrices . . . 13
2.12.3 Multiplicación de matrices . . . 13
2.12.4 Flip de matrices . . . 14
2.12.5 Tranposición de matrices . . . 14
2.12.6 Rotación de matrices . . . 15
2.13 Búsqueda y Ordenamiento . . . 16
2.13.1 *Bubble Sort* . . . 16
2.13.2 *Merge Sort* . . . 16
2.13.3 Bucket Sort / Counting Sort . . . 17
2.13.4 Conclusiones sobre el ordenamiento . . . 17
2.14 Problemas que se pueden resolver con ordenamiento . . . 20
2.14.1 Encontrar duplicados en un array . . . 20
2.14.2 Búsqueda binaria . . . 20
2.15 Árboles . . . 21
2.15.1 Recorridos de árboles . . . 21

2.1 Introducción

La teórica de algoritmos se enfoca a la resolución de problemas algorítmicos. La razón por la cual es importante estudiar algoritmos, es que necesitamos aprender a resolver problemas con CPUs de manera eficiente para poder luego empezar a pensar los problemas de una manera paralela y distribuida con GPUs.

Las técnicas para resolver algoritmos pueden variar dependiendo del problema, y sólo vamos a ver algunas de las más sencillas para que puedan empezar a pensar en cómo resolverlos. Otras técnicas más avanzadas las verán en materias como **Algoritmos 1** y **Algoritmos 2**, que son materias enfocadas a exclusivamente al estudio de algoritmos y cómo implementarlos.

Es por eso que esta teórica no pretende ser exhaustiva, sino que es una introducción a los conceptos básicos que necesitaremos para la materia.

Definición de algoritmo: Un algoritmo es un conjunto finito y ordenado de pasos o instrucciones que se siguen para resolver un problema o realizar una tarea específica. Cada paso debe ser claro, preciso y ejecutable, y el algoritmo debe llevar a una solución definida en un número finito de pasos.

Gran parte de esta teórica está basada en el libro "Guide to Competitive Programming" **gcc2017** y "Cracking the Coding Interview" **cracking2015** que son libros de textos para estudiantes de computación que se enfocan en la solución de problemas algorítmicos para competencias de programación y la preparación para entrevistas técnicas en empresas de tecnología. Ambos libros son excelentes recursos en el caso de que quieran profundizar en el tema de algoritmos y aprender a pasar entrevistas técnicas. [1]

2.2 Números

2.2.1 Enteros

C posee varios tipos de datos para representar números. El más utilizado es el tipo `int`, que tiene 32 bits y representa números enteros en el rango de -2^{31} a $2^{31}-1$. Cuando `int` no es suficiente, se puede utilizar el tipo `long long`, que tiene 64 bits y representa números enteros en el rango de -2^{63} a $2^{63}-1$. De todas formas siempre se puede agregar el modificador `unsigned` para representar números enteros sin signo, lo que permite representar números enteros en el rango de 0 a $2^{32}-1$ para `unsigned int` y 0 a $2^{64}-1$ para `unsigned long long`.

[1] Recuerden siempre que tener el conocimiento para resolver problemas algoritmos es el primer paso para conseguir un trabajo en empresas de tecnología, particularmente en FAANG (Facebook, Amazon, Apple, Netflix y Google).

2.2.2 Aritmética modular

A veces necesitamos trabajar con números que se van de los rangos de los tipos de datos. En estos casos, podemos utilizar la aritmética modular, que nos permite trabajar con números en un rango específico. El módulo (operación representada por `%`) es una operación que nos da el resto de la división. Por ejemplo, `5 % 3 == 2`, porque 5 dividido por 3 da 1 con un resto de 2. En general si `f(n)` es una función que devuelve el un valor entero, `f(n) % m` nos va a dar un número en el rango de 0 a $m - 1$, sin importar el valor de retorno de `f(n)`.

2.2.3 Flotantes

El tipo `float` en C representa números de punto flotante de precisión simple, mientras que el tipo `double` representa números de punto flotante de precisión doble. Estos tipos son útiles para representar números con decimales, pero es importante tener en cuenta que pueden introducir errores de redondeo debido a la forma en que se almacenan los números en la memoria [2].

Con lo cual, supongamos que tenemos el siguiente código:

```c
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

Esto se debe a que 0.3 y 0.1 no pueden ser representados exactamente en binario, lo que provoca un error de redondeo al realizar la operación. Esto no es exclusivo de C, sino que es un problema de la representación de números de punto flotante en general.

Con lo cual muchas veces es mejor trabajar con números enteros, o si es necesario trabajar con números de punto flotante es preferible utilizar umbrales de tolerancia, por ejemplo:

```c
#include <math.h> // Recordar incluir esta librería para poder usar fabs(
    double)

if(fabs(a-b) < 1e-9) {
  // a y b son considerados iguales
}
```

2.3 Macros

Las macros son una herramienta poderosa en C que nos permite definir constantes y algunas funciones cuando necesitamos repetir código. Por ejemplo, podemos definir una macro para el valor de π o una función para calcular el máximo y mínimo entre dos números. Las macros se definen utilizando la directiva `#define` y se pueden utilizar en cualquier parte del código, pero normalmente se definen al principio del archivo (para más prolijidad):

```c
#define PI 3.14159265358979323846
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
```

[2] IEEE 754

2.4 Manipulación de bits

En programación, los números enteros se representan con n-bits, donde n es el número de bits que se utilizan para almacenarlos. Por ejemplo, un número entero de 32 bits se representa con 32 bits en la memoria. Cada bit puede ser 0 o 1, lo que significa que un número entero de 32 bits puede representar 2^{32} valores diferentes. Por ejemplo el número `43` se representa en binario como `00000000000000000000000000101011`.

Genéricamente la fórmula para convertir un número decimal en binario es:

$$d = 2^{n-1} \cdot b_{n-1} + 2^{n-2} \cdot b_{n-2} + \ldots + 2^1 \cdot b_1 + 2^0 \cdot b_0$$

Entonces por ejemplo el número 43 se representa en binario como:

$$43 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

Los números negativos se representan utilizando el *complemento a dos*, que se calcula invirtiendo todos los bits del número y sumando 1 al resultado. Por ejemplo, el número -43 se representa en binario como:

$$-43 = 11111111111111111111111111010101$$

En C, el operador de complemento a dos se representa con el operador `-`.

2.5 Operaciones con bits

En C se pueden hacer operaciones sobre bits utilizando los operadores `&`, `|`, `^`, `~`, `<<` y `>>`.

Operación AND produce un 1 en el bit de salida si ambos bits de entrada son 1. Por ejemplo:

$$\begin{array}{r} 10110\ (22) \\ \texttt{\&}\ 11010\ (26) \\ \hline 10010\ (18) \end{array}$$

Operación OR produce un 1 en el bit de salida si al menos uno de los bits de entrada es 1. Por ejemplo:

$$\begin{array}{r} 10110\ (22) \\ \texttt{|}\ 11010\ (26) \\ \hline 11110\ (30) \end{array}$$

Operación XOR produce un 1 en el bit de salida si los bits de entrada son diferentes. Por ejemplo:

$$\begin{array}{r} 10110\ (22) \\ \texttt{\^{}}\ 11010\ (26) \\ \hline 01100\ (12) \end{array}$$

Operación NOT invierte los bits de entrada. Por ejemplo:

$$\texttt{\~{}}43 = 11111111111111111111111111010100 = 4294967252$$

Operación SHIFT desplaza los bits de entrada a la izquierda o a la derecha. Esto es análogo a multiplicar o hacer la división entera por 2. Por ejemplo:

$$43 << 1 = 00000000000000000000000001010110 = 86$$
$$43 >> 1 = 00000000000000000000000000001011 = 21$$

Máscaras se utilizan para seleccionar bits específicos de un número. Por ejemplo, si se quiere seleccionar los bits 4 a 7 de un número, se puede hacer lo siguiente:

$$\begin{array}{r} 43 = 00000000000000000000000000101011 \\ \text{mask} = 00000000000000000000000000001111 \\ \hline 43\&\text{mask} = 00000000000000000000000000001011 = 11 \end{array}$$

2.6 Representación de Conjuntos

Cada subconjunto del conjunto $\{0, 1, 2, \ldots, n-1\}$ se puede representar como un número entero de n bits, donde cada bit representa si el elemento está presente en el subconjunto o no. Por ejemplo:

```
int x = 0;
x |= (1 << 1); // Agregar el elemento 1 al conjunto
x |= (1 << 3); // Agregar el elemento 3 al conjunto
x |= (1 << 4); // Agregar el elemento 4 al conjunto
x |= (1 << 8); // Agregar el elemento 8 al conjunto

for(int i = 0; i < 32; i++) {
  if(x & (1 << i)) {
    printf("%d ", i); // Imprimir los elementos del conjunto
  }
}
```


2.7 Eficiencia algorítmica

Esto lo vamos a ver con más detalle en la próxima teórica, pero es importante mencionar que la eficiencia algorítmica estima cuánto tiempo va a utilizar un algoritmo para resolver un problema en función del tamaño de la entrada.

La complejidad algorítimica se nota con la notación $\mathcal{O}$ (grande O), que estima la cota superior del tiempo de ejecución de un algoritmo en función del tamaño de la entrada. La cota superior es la estimación del peor caso posible en base al tamaño de la entrada.

Ejemplos:

- $\mathcal{O}(1)$: una asignación o una cuenta como $\frac{1}{2} + \frac{1}{3}$, el tiempo de ejecución no depende del tamaño de la entrada.
- $\mathcal{O}(\log n)$: una búsqueda binaria, el tiempo de ejecución crece logarítmicamente con el tamaño de la entrada.
- $\mathcal{O}(n)$: recorrer un arreglo de n elementos.
- $\mathcal{O}(n \log n)$: un algoritmo de ordenamiento eficiente como `quicksort` o `mergesort`.
- $\mathcal{O}(n^2)$: buscar todos los pares de elementos que cumplen con una cierta condición en un arreglo de n elemento.
- $\mathcal{O}(2^n)$: Generar todos los subconjuntos de un conjunto de n elementos.
- $\mathcal{O}(n!)$: Resolver el problema de las n-reinas, que consiste en colocar n reinas en un tablero de ajedrez de $n \times n$ de tal manera que ninguna reina ataque a otra.

Estimar la eficiencia algorítmica de un algoritmo es algo que puede revisarse *antes* de implementarlo, y es una herramienta muy útil para saber si la solución que estamos pensando es viable o no.

Formalmente: que un algoritmo funcione en $O(f(n))$ significa que existe una constante c y un valor n_0 tal que para todo $n \geq n_0$ se cumple que el algoritmo va a realizar como máximo $c \cdot f(n)$ operaciones para todas las operaciones donde $n \geq n_0$. De todas formas se intenta utilizar el mejor caso posible de límite superior, es decir que si un algoritmo funciona en $O(n)$, su límite también podría ser $O(n^2)$, pero se prefiere utilizar el límite más bajo posible.

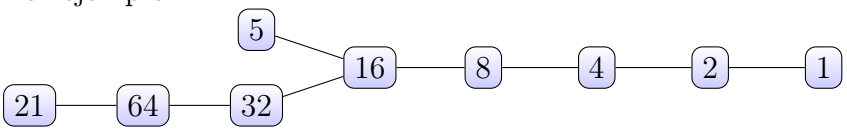
2.8 Soluciones a los algoritmos

Los algoritmos presentados en las soluciones no están escritos en un lenguaje de programación específico, sino que están escritos en un pseudocódigo que es fácil de entender e implementar en cualquier lenguaje de programación. La idea es que por un lado aprendan a leer algoritmos y por otro lado aprendan a escribir código sin copiar y pegar.

Por ejemplo imaginemos que queremos escribir un algoritmo que implemente la Conjetura de Collatz. En pocas palabras, la conjetura de Collatz determina que una función $f(n)$ que se define de la siguiente manera:

$$f(n) = \begin{cases} \frac{n}{2} & \text{si } n \text{ es par} \\ 3n+1 & \text{si } n \text{ es impar} \end{cases}$$

Por ejemplo:



Se puede pensar en el siguiente algoritmo para resolver la conjetura de Collatz:

Algorithm 1 Conjetura de Collatz

Define: $f(n)$
Input: $n \in \mathbb{N}$
Initialization: $\emptyset$
Output: Los números de la conjetura de Collatz hasta llegar a 1

```
1: function COLLATZ(n)
2:     while n ≠ 1 do
3:         print n
4:         if n is odd then
5:             n ← n/2
6:         else
7:             n ← 3n + 1
8:     print n
```


2.9 Teoría de Números

La teoría de númers es la rama de las matemáticas que estudia los números enteros. En esta sección vamos a charlar un poco sobre algunos conceptos de algoritmos y teoría de números.

2.10 Números primos

Se dice que un número entero a es un *factor* o un *divisor* de un número entero b, si a divide a b. Esto significa que existe un número entero c tal que $b = a \cdot c$. Se nota como $a \mid b$. Por ejemplo los factores de 24 son 1, 2, 3, 4, 6, 8, 12 y 24.

Un número entero $n > 1$ es *primo* si sus únicos factores son 1 y n. Por ejemplo los números 2, 3, 5, 7, 11 son primos, mientras que 93 no es primo ya que $93 = 3 \cdot 31$. Además para cada entero $n \geq 1$ existe una *única factorización* de n en números primos:

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$$

Por ejemplo:

$$84 = 2^2 \cdot 3^1 \cdot 7^1$$

Los algoritmos de búsqueda de números primos son bastante lentos e incluso las formas más complejas de encontrar si un número es primo o no, como el test de primalidad de AKS [3], son muy lentos para primos muy grandes (números de 512 bits o más).

Supongamos que queremos saber si un número n es primo o no, veamos los algoritmos más sencillos para resolver este problema.

2.10.1 Búsqueda lineal

Este es el algoritmo más simple y consiste en recorrer todos los números desde 2 hasta n y verificar si son primos o no. Si bien el algoritmo no es muy eficiente, es fácil de entender y de implementar.

```
bool es_primo(int n) {
    if (n <= 1) return false; // no es primo porque
    for (int i = 2; i <= n; i++)
        if (n % i == 0) return false; // No es primo
    return true; // Es primo
}
```

2.10.2 Búsqueda lineal hasta $\sqrt{n}$

Si pensamos un poco el algoritmo anterior nos vamos a dar cuenta de que no es necesario recorrer todos los números hasta n. Sin embargo nos podemos preguntar **¿Hasta qué número tenemos que recorrer para saber si un número es primo o no?**, la respuesta no es inmeditamente trivial.

Claramente $n-1$ nunca puede ser un divisor de n (excepto en el caso $n = 2$), pero entonces ¿es divisor $n-2, n-3, \cdots, n-k$?

[3] Test de primalidad AKS

Si n no es primo (por lo que dijimos antes) debe tener por lo menos dos factores a y b tales que $n = a \cdot b$. Se puede demostrar fácilmente que al menos uno de esos factores debe ser menor o igual a $\sqrt{n}$.

Demostración: Sea $n \in \mathbb{N}$. Si n no es primo, entonces tiene al menos dos factores positivos distintos, digamos a y b, tales que $a \cdot b = n$. Supongamos que ambos a y b son mayores que $\sqrt{n}$. Entonces se tendría que $a > \sqrt{n}$ y $b > \sqrt{n}$, lo que implica que $ab > \sqrt{n} \cdot \sqrt{n} = n$, una contradicción. Por lo tanto, al menos uno de los factores debe ser menor o igual a $\sqrt{n}$. Así, si n tiene un divisor propio (por ejemplo, un primo que lo divide), debe encontrarse entre 2 y $\sqrt{n}$.

Código:

```
bool es_primo(int n) {
    if (n <= 1) return false; // no es primo porque
    for (int i = 2; i * i <= n; i++)
        if (n % i == 0) return false; // No es primo
    return true; // Es primo
}
```

Notar que hacer que el algoritmo recorra hasta $i * i \leq n$ es lo mismo que recorrer hasta $i \leq \sqrt{n}$, con la ventaja de que no necesitamos utilizar números de punto flotante para calcular la raíz cuadrada, lo que evita errores de redondeo.

2.10.3 Criba de Eratóstenes

Una forma muy eficiente de encontrar números primos es utilizar la Criba de Eratóstenes. Pero tiene la desventaja de que necesitamos tener muchísima memoria disponible, ya que utlizaremos un vector de booleanos de tamaño n para saber si el número es primo o no. El algoritmo consiste en recorrer todos los números desde 2 hasta n y e ir marcando los números compuestos como NO primos. Cuando terminamos de recorrer todos los números vamos a tener un listado de números primos entre 2 y n.

```
#include <stdio.h>
#include <stdbool.h>

int main() {
  int n = 100; // Cambiar este valor para encontrar primos hasta n
  bool es_primo[n + 1];
  for (int i = 0; i <= n; i++) es_primo[i] = true; // Inicializar el vector

  es_primo[0] = es_primo[1] = false; // 0 y 1 no son primos

  for (int i = 2; i * i <= n; i++) {
      if (es_primo[i]) { // Si es primo
          for (int j = i * i; j <= n; j += i) { // Marcar todos los
   multiplos de i como NO primos
              es_primo[j] = false;
          }
      }
  }

  // Imprimir los numeros primos
```



```
    printf("Numeros primos hasta %d:\n", n);
    for (int i = 2; i <= n; i++) {
        if (es_primo[i]) printf("%d ", i);
    }
}
```

Hay otros métodos más avanzados para encontrar números primos como *Pollard Rho* o el *Test de Miller-Rabin*, pero exceden lárgamente el alcance de esta teórica.

2.11 Vectores

Los arrays en C son porciones de memoria contiguas que almacenan elementos del mismo tipo. Se pueden declarar de la siguiente manera:

```
int arr[10]; // Un array de 10 enteros
int arr15] = {1, 2, 3, 4, 5}; // Un array de 5 enteros inicializado
int arr2] = {1, 2, 3, 4, 5}; // Un array de 5 enteros inicializado
int *arr3= malloc(10 * sizeof(int)); // Un array de 10 enteros dinamico
```

Los arrays se pueden recorrer utilizando un bucle `for` o `while`, y se accede a sus elementos utilizando el operador de índice `[]`. Por ejemplo, para recorrer un array de enteros y sumar sus elementos:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n = 5;
    int *arr = malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1; // Inicializar el array
    }

    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i]; // Sumar los elementos del array
    }

    printf("La suma de los elementos del array es: %d\n", sum);
    free(arr); // Liberar la memoria del array
    return 0;
}
```

2.12 Matrices

Las matrices son un concepto matemático que se utiliza para representar datos en forma de tablas. En C, las matrices se pueden declarar como arrays bidimensionales; o como veremos más adelante, como arrays unidimensionales donde cada elemento de la matriz se calcula como un índice en el array unidimensional.

Por ejemplo una matriz de 3×3 se puede declarar de la siguiente manera:

```c
int matriz[3][3]; // matriz de 3x3 enteros
int matriz2[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
}; // matriz de 3x3 enteros inicializada

int *matriz3 = (int *)malloc(3 * 3 * sizeof(int)); // matriz de 3x3 enteros
    dinamica
```

Por otro lado un vector unidimensional de n elementos es como si fuera una matriz de $1 \times n$ o $n \times 1$, por lo que se puede declarar de la siguiente manera:

```c
int vector[10]; // Vector de 10 enteros
int vector2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Vector de 10 enteros
    inicializado
int *vector3 = (int *)malloc(10 * sizeof(int)); // Vector de 10 enteros
    dinamico
```

2.12.1 Acceso a los elementos de una matriz

Cuando se trabajan con índices en un vector unidimensional se suele utilizar el índice `i` para hablar del i-ésimo elemento del vector. Lamentablemente, por analogía, en una matriz bidimensional se suelen utilizar los índices `i` y `j` para hablar de las filas y las columnas de la matriz, lo que puede llevar a confusiones. En el libro Cracking The Coding Interview **cracking2015** se recomiendan la utilización de los índices `row` y `col` para hablar de las filas y las columnas de una matriz, lo que ayuda a evitar confusiones, y es lo que utilizaremos en la materia (a veces también `fila` y `col`).

Sin embargo, como veremos en la práctica de CUDA, a veces no podemos saber el tamaño de la matriz en tiempo de compilación y tenemos que utilizar un array unidimensional para representar la matriz. En este caso, el elemento `(row, col)` de la matriz se puede calcular como:

$$\text{matriz}[row][col] = \text{array}[row \cdot n + col]$$

Donde n es el número de columnas de la matriz. Por ejemplo si tenemos una matriz de 5 filas y 10 columnas y queremos accedeer al número en la posición `matrix[2][3]` (es decir, fila 2 y columna 3), podemos calcular el índice de la siguiente manera:

$$\text{matrix}[2][3] = \text{array}[2 \cdot 10 + 3] = \text{array}[20 + 3] = \text{array}[23]$$

2.12.2 Suma de matrices

La suma de dos matrices $A+B$ está definida sólo para matrices del mismo tamaño. El resultado es una matriz donde cada elemento es la suma de los elementos correspondientes de las dos matrices, por ejemplo si tenemos las matrices:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \end{bmatrix}$$

2.12.3 Multiplicación de matrices

La multiplicación de matrices $A \cdot B$ está definida sólo para matrices donde el número de columnas de la primera matriz es igual al número de filas de la segunda matriz. El resultado se calcula con la siguiente fórmula:

$$(A \cdot B)_{ij} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

(Sí, en este caso utilizamos i y j para las filas y columnas de la matriz, ya que es una convención común en álgebra, si fuera código recomendaríamos utilizar `row_a, col_a` y `row_b, col_b`.

Si tuviéramos dos matrices A y B de tamaño $n \times n$, el código tendría la siguiente forma:

```
for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= n; j++) {
    for (int k = 1; k <= n; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

2.12.4 Flip de matrices

Definimos el flip de una matriz como la operación que intercambia las filas y las columnas de la matriz. Es decir, que dada una matriz A el flip horizontal se define como $swap(A_{i,j}) = A_{n-i,j})\forall i, j$ y el flip vertical se define como $swap(A_{i,j}) = A_{i,m-j})\forall i, j$ con n el número de filas y m el número de columnas.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \xrightarrow{\text{flip vertical}} \begin{bmatrix} a_{21} & a_{22} & a_{23} \\ a_{11} & a_{12} & a_{13} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \xrightarrow{\text{flip horizontal}} \begin{bmatrix} a_{13} & a_{12} & a_{11} \\ a_{23} & a_{22} & a_{21} \end{bmatrix}$$

Algorithm 2 Flip de una matriz

Define: flipMatrix
Input: `matrix` (matriz de $n \times m$)
Output: La matriz espejada horizontalmente
for $i = 0$ **to** $n - 1$ **do**
  for $j = 0$ **to** $m/2 - 1$ **do**
    $temp \leftarrow matrix[i][j]$
    $matrix[i][j] \leftarrow matrix[i][m - j - 1]$
    $matrix[i][m - j - 1] \leftarrow temp$

2.12.5 Tranposición de matrices

La matriz tranpuesta A^T de una matriz A es una matriz que se obtiene de intercambiando las filas y las columnas de la matriz original $A^T_{i,j} = A_{j,i}$.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \xrightarrow{\text{tranposición}} \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \end{bmatrix}$$

Algorithm 3 Tranposición de una matriz

Define: transposeMatrix
Input: `matrix` (matriz de $n \times m$)
Output: `output` (matriz tranpuesta de $m \times n$)
for $i = 0$ **to** $n - 1$ **do**
  for $j = 0$ **to** $m - 1$ **do**
    $output[j][i] \leftarrow matrix[i][j]$

2.12.6 Rotación de matrices

La rotación de una matriz es la operación que rota los elementos de la matriz en sentido horario o antihorario. Es bastante sencillo de implementar utilizando una matriz de salida y recorriendo los elementos de la matriz de entrada de la siguiente manera $A_{i,j} \rightarrow B_{j,n-i}$ para la rotación en sentido horario.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \xrightarrow{\text{rotación en sentido horario}} \begin{bmatrix} a_{21} & a_{11} \\ a_{22} & a_{12} \\ a_{23} & a_{13} \end{bmatrix}$$

Es interesante señalar que la rotación de una matriz cuadrada de $n \times n$ en sentido horario se puede realizar *in-place* (sin utilizar memoria extra). Transponiendo la matriz y luego haciendo un flip horizontal de la matriz, como se muestra en el siguiente algoritmo:

Algorithm 4 Rotación de una matriz cuadrada (in-place) en sentido horario

Define: rotateMatrix
Input: `matrix` (matriz cuadrada de $n \times n$)
Output: La matriz rotada en sentido horario
for $i = 0$ **to** $n - 1$ **do**
  for $j = i + 1$ **to** $n - 1$ **do**
    $temp \leftarrow matrix[i][j]$
    $matrix[i][j] \leftarrow matrix[j][i]$
    $matrix[j][i] \leftarrow temp$
for $i = 0$ **to** $n - 1$ **do**
  for $j = 0$ **to** $n/2 - 1$ **do**
    $temp \leftarrow matrix[i][j]$
    $matrix[i][j] \leftarrow matrix[i][n - j - 1]$
    $matrix[i][n - j - 1] \leftarrow temp$

2.13 Búsqueda y Ordenamiento

Muchos algoritmos eficientes se basan en ordenar los datos de entrada antes de resolver el problema, dado que el ordenamiento a menudo facilita la resolución del problema. En esta sección discutiremos algo de teoría de búsqueda y ordenamiento de datos.

El problema básico de ordenamiento es el siguiente: Dado un *array* que contiene n elementos, ordenarlos de manera ascendente. Es bastante fácil resolver este problema en $O(n^2)$, sin embargo existen algunos algoritmos que resuelven este problema de manera más eficiente.

2.13.1 *Bubble Sort*

Este algoritmo lárgamente visto en todos los cursos de programación, es el más fácil de explicar por su simplicidad y facilidad de implementación. El algoritmo consiste en recorrer el array y comparar cada elemento con el siguiente, si el elemento actual es mayor que el siguiente, se intercambian los elementos. Este proceso se repite hasta que no se realizan más intercambios, lo que significa que el array está ordenado.

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-1; j++) {
    if(arr[j] > arr[j + 1]) {
      // Intercambiar los elementos
      int temp = arr[j];
      arr[j] = arr[j + 1];
      arr[j + 1] = temp;
    }
  }
}
```

Inversiones: Un concepto importante en el ordenamiento de datos es el concepto de *inversiones*. Una inversión de los índices `(a, b)` tales que $a < b$ y `arr[a] > arr[b]` (o lo que es lo mismo, que los elementos están en el orden incorrecto). El número de inversiones en un array es una medida de cuán desordenado está el array. Por ejemplo, si el array está totalmente ordenado, el número de inversiones es 0, y si el array está en orden inverso, el número de inversiones es:

$$1 + 2 + 3 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

Esto es porque intercambiar dos elementos del array reduce exactamente en 1 el número de inversiones. Con lo cual si un algoritmo sólo puede reducir el número de inversiones en 1 por iteración, el número de iteraciones necesarias para ordenar el array es al menos $O(n^2)$.

2.13.2 *Merge Sort*

Si queremos crear un algoritmo de ordenamiento más eficiente, podemos utilizar el algoritmo *Merge Sort*. Este algoritmo nos permite ordenar elementos que están en partes diferentes del array. El algoritmo entra en el conjunto de algoritmos que permiten ordenar un array en $O(n \log n)$ tiempo.

El algoritmo merge sort intenta ordenar el subarray `arr[a..b]` de la siguiente manera:

1. Si $a = b$, el array contiene un solo elemento por lo que se considera ordenado.

2. Calculamos la posición del elemento del medio $\lfloor \frac{a+b}{2} \rfloor$.
3. Recursivamente ordenamos el subarray `arr[a..k]`
4. Recursivamente ordenamos el subarray `arr[k + 1..b]`
5. Combinamos (*merge*) el subarray `arr[a..k]` y el subarray `arr[k + 1..b]` en un solo array ordenado.

El algoritmo *merge sort* es eficiente porque divide el array en dos mitades cada vez y el merge de las mitades se hace en tiempo lineal porque ya están ordenadas. Dado que hay $log(n)$ niveles de división y procesar cada nivel toma $O(n)$ tiempo, el tiempo total de ejecución es $O(n \log n)$.

2.13.3 Bucket Sort / Counting Sort

En vista de lo que vimos anteriormente con resepcto a las inversiones, podemos pensar si se puede ordenar un array en menos complejidad que $O(nlogn)$. Y la respuesta es sí, pero para ello no tenemos que comparar elementos entre sí, en caso contrario, no podemos superar la barrera de $O(nlogn)$.

El algoritmo *Bucket Sort* es un algoritmo de ordenamiento que utiliza un array para contar la cantidad de veces que aparece cada elemento del array. El algoritmo funciona de la siguiente manera:

1. Creamos un array de tamaño k donde k es el número de elementos distintos del array a ordenar.
2. Recorremos el array y contamos la cantidad de veces que aparece cada elemento, guardando el resultado en el array de conteo.
3. Recorremos el array de conteo y escribimos los elementos en el array original en orden.

La ventaja es que este algoritmo tiene una complejidad $O(n)$, pero tiene la desventaja de que necesita un array de tamaño k para contar los elementos.

2.13.4 Conclusiones sobre el ordenamiento

Aunque es importante entender los conceptos básicos de los algoritmos de ordenamiento, en la práctica suele ser una mala idea implementar algoritmos de ordenamiento desde cero. Todos los lenguajes de programación tienen implementaciones de ordenamiento eficientes y optimizadas. Por ejemplo, en C podemos utilizar la función `qsort` de la biblioteca estándar, que implementa el algoritmo *Quicker Sort* (un algoritmo de ordenamiento que es una variante del algoritmo *Quick Sort*). Pero tiene la desventaja de que no es estable, es decir, que no mantiene el orden de los elementos iguales.

MergeSort en C:

```c
#include <stdio.h>
#include <stdlib.h>

//
// MERGE function: Combina dos subarrays ordenados en uno solo
//
void merge(int array[], int left, int mid, int right) {
  // Calcula el size de los subarrays
  int leftSize = mid - left + 1;
  int rightSize = right - mid;

  // Crea arreglos temporales para los subarrays
  int* leftSubarray = (int*)malloc(leftSize * sizeof(int));
  int* rightSubarray = (int*)malloc(rightSize * sizeof(int));

  if (!leftSubarray || !rightSubarray) {
    fprintf(stderr, "No se pudo reservar la memoria\n");
    exit(1);
  }

  // Copiamos los datos a los subarrays temporales
  for (int i = 0; i < leftSize; i++)
    leftSubarray[i] = array[left + i];
  for (int j = 0; j < rightSize; j++)
    rightSubarray[j] = array[mid + 1 + j];

  // Combinamos los subarrays temporales en el array original
  int i = 0;    // Indice inicial del subarray izquierdo
  int j = 0;    // Indice inicial del subarray derecho
  int k = left; // Indice inicial del array combinado

  // Compar los elementos de los subarrays y combinarlos
  while (i < leftSize && j < rightSize) {
    if (leftSubarray[i] <= rightSubarray[j]) {
      array[k] = leftSubarray[i];
      i++;
    } else {
      array[k] = rightSubarray[j];
      j++;
    }
    k++;
  }

  // Copiar cualquier elemento restante del subarray izquierdo
  while (i < leftSize) {
    array[k] = leftSubarray[i];
    k++;
    i++;
  }

  // Copiar cualquier elemento restante del subarray derecho
  while (j < rightSize) {
```

```
    array[k] = rightSubarray[j];
    k++;
    j++;
  }

  // Liberar la memoria de los subarrays temporales
  free(leftSubarray);
  free(rightSubarray);
}

//
// mergeSort: implementa la estrategia divide-and-conquer para ordenar un
    array
//
void mergeSort(int array[], int left, int right) {
  if (left < right) {
    // Buscamos el punto medio del array (divide)
    int mid = left + (right - left) / 2;

    // Ordenamos recursivamente las dos mitades
    mergeSort(array, left, mid);        // Sort left half
    mergeSort(array, mid + 1, right);   // Sort right half

    // Combinamos las dos mitades ordenadas
    merge(array, left, mid, right);
  }
}

//
// Utility function to print an array
//
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++)
    printf("%d ", array[i]);
  printf("\n");
}

//
// Main function to demonstrate merge sort
//
int main() {
  int arr[] = {38, 27, 43, 3, 9, 82, 10};
  int size = sizeof(arr) / sizeof(arr[0]);

  printf("Array original:\n");
  printArray(arr, size);

  mergeSort(arr, 0, size - 1);

  printf("\nArray Ordenado:\n");
  printArray(arr, size);

  return 0;
}
```

2.14 Problemas que se pueden resolver con ordenamiento

2.14.1 Encontrar duplicados en un array

Por ejemplo supongamos que queremos saber si los elementos de un array son todos diferentes entre sí. Claramente el algoritmo por fuerza bruta es el más sencillo de implementar, sin embargo ese algoritmo tiene una complejidad de $O(n^2)$.

```
bool unicos = true;
for (int i = 0; i < n; i++) {
  for (int j = i + 1; j < n; j++) {
    if (arr[i] == arr[j]) {
      unicos = false;
      break;
    }
  }
}
```

Sin embargo, si ordenamos el array primero, el problema puede ser resuelto en $O(n \log n)$. En la práctica tendrás un ejercicio para resolver este problema.

2.14.2 Búsqueda binaria

Supongamos que tenemos un array ordenado y queremos saber si un elemento está presente en el array. El algoritmo de búsqueda binaria es un algoritmo eficiente que nos permite encontrar un elemento en un array ordenado en $O(\log n)$ tiempo. La implementación del algoritmo en C es la siguiente:

```
int busquedaBinaria(int arr[], int n, int x) {
  int left = 0, right = n - 1;
  while (left <= right) {
    int mid = (right + left) / 2; // Encontrar el punto medio
    if (arr[mid] == x) return mid; // Encontramos el elemento!!
    if (x > arr[mid]) left = mid + 1; // Buscar en la mitad derecha
    else right = mid - 1; // Buscar en la mitad izquierda
  }
  return -1; // No se encontro el elemento
}
```

2.15 Árboles

Los grafos son una estructura de datos que consiste en un conjunto de nodos conectados por aristas, se utilizan para representar relaciones entre datos. Los grafos pueden ser dirigidos o no dirigidos, y pueden tener ciclos o no. Un grafo dirigido es aquel en el que las aristas tienen una dirección, es decir, que van de un nodo a otro nodo específico, mientras que un grafo no dirigido es aquel en el que las aristas no tienen dirección, es decir, que la relación entre los nodos es bidireccional.

Un árbol es un caso especial de grafo, que es **no dirigido**, **acíclico** y **conexo**. Es decir, que un árbol es un grafo que no tiene ciclos y que está conectado (esto es, que existe un camino entre cualquier par de nodos). Cada árbol tiene un nodo raíz, que es el nodo superior del árbol, y cada nodo puede tener cero o más nodos hijos. Los árboles son una estructura de datos muy útil para representar jerarquías entre datos, como por ejemplo un sistema de archivos o una base de datos.

Uno de los árboles más comunes es el árbol binario, que es un árbol en el que cada nodo tiene como máximo dos hijos, y es una estructura de datos muy utilizada en algoritmos de búsqueda y ordenamiento.

Si bien los árboles se pueden representar de muchas maneras, la más común es utilizar una estructura de datos que contenga un valor y dos punteros a los nodos hijos izquierdo y derecho. Por ejemplo:

```
struct Nodo {
    int valor;
    struct Nodo* izquierdo;
    struct Nodo* derecho;
};
```

Esto tiene la ventaja de que podemos representar árboles de cualquier tamaño y forma, y es fácil de implementar en C; otra forma de representar un árbol es utilizando un array, donde el nodo en la posición i tiene como hijos los nodos en las posiciones $2i + 1$ y $2i + 2$. Esta representación es útil para árboles completos o casi completos, ya que si el árbol está muy desbalanceado, la representación en array puede desperdiciar mucha memoria.

2.15.1 Recorridos de árboles

Los recorridos de árboles son una forma de visitar todos los nodos de un árbol en un orden específico. Los tres recorridos más comunes son el recorrido en preorden, el recorrido en inorden y el recorrido en postorden.

- **Preorden:** En este recorrido, se visita primero el nodo raíz, luego se recorre el subárbol izquierdo y finalmente se recorre el subárbol derecho. El orden de visita es: raíz, izquierda, derecha.
- **Inorden:** En este recorrido, se recorre primero el subárbol izquierdo, luego se visita el nodo raíz y finalmente se recorre el subárbol derecho. El orden de visita es: izquierda, raíz, derecha.
- **Postorden:** En este recorrido, se recorre primero el subárbol izquierdo, luego se recorre el subárbol derecho y finalmente se visita el nodo raíz. El orden de visita es: izquierda, derecha, raíz.

Supongamos que tenemos el siguiente árbol binario:

1/ 23/ 456

El recorrido en preorden sería: 1, 2, 4, 5, 3, 6

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int value;
    struct Node* left;
    struct Node* right;
} Node;

Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        perror("Error al asignar memoria");
        exit(EXIT_FAILURE);
    }
    newNode->value = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

void preorder(Node* root) {
    if (root == NULL) return;
    printf("%d ", root->value); // Visitar nodo
    preorder(root->left);       // Recorrer subárbol izquierdo
    preorder(root->right);      // Recorrer subárbol derecho
}

int main() {
    // Construcción del árbol:
    //
    //        1
    //       / \
    //      2   3
    //     / \   \
    //    4   5   6
    //
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->right = createNode(6);

    printf("Recorrido en preorden: ");
    preorder(root);
    printf("\n");

```

```
49      return 0;
50  }
```