

Agenda



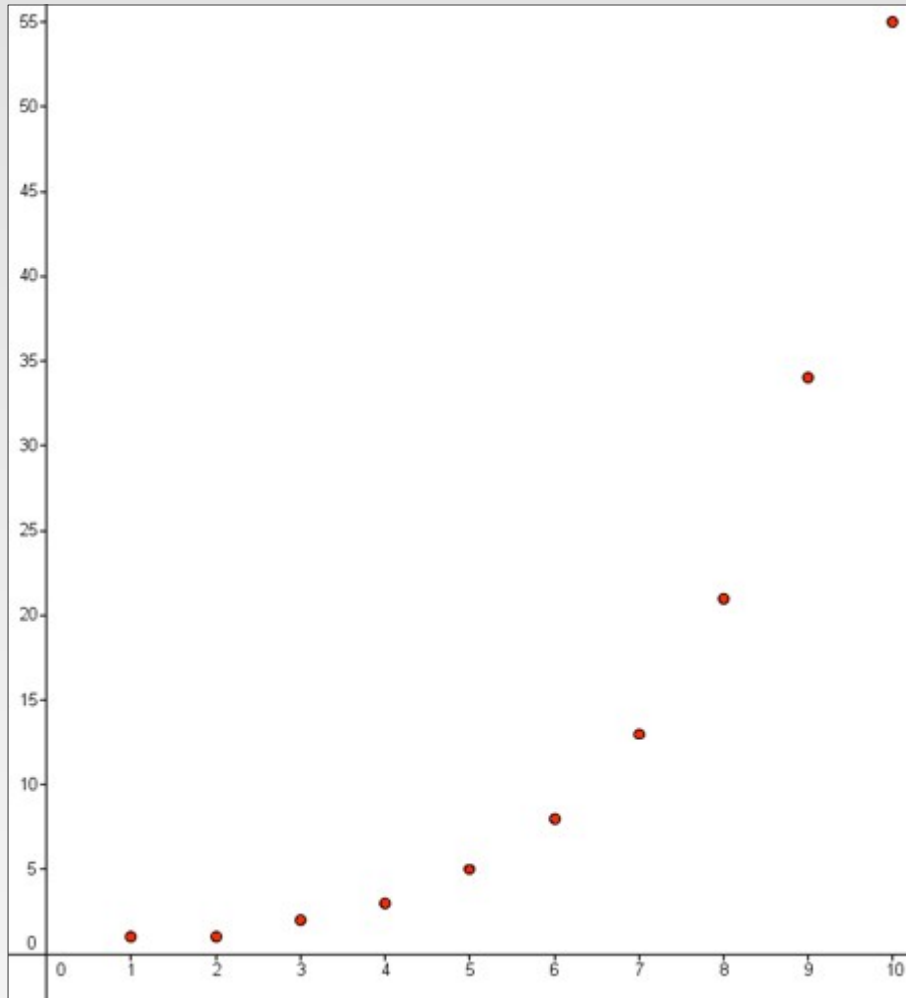
- 1) Entrega de Quiz 2
- 2) Dudas de Quiz 2
- 3) Exposición de Tarea
- 4) Hashing y peso de las funciones
- 5) Realización de Quiz 3 (3:20)
- 6) Recordatorio:
 - 1) Mañana Examen final.

Recordando

Fibonacci

```
int fibonacci(int n) {  
    if (n <= 1) { return n; }  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```

Gráfica de resultado Fibonacci



Crecimiento Exponencial

Función ESTÁNDAR

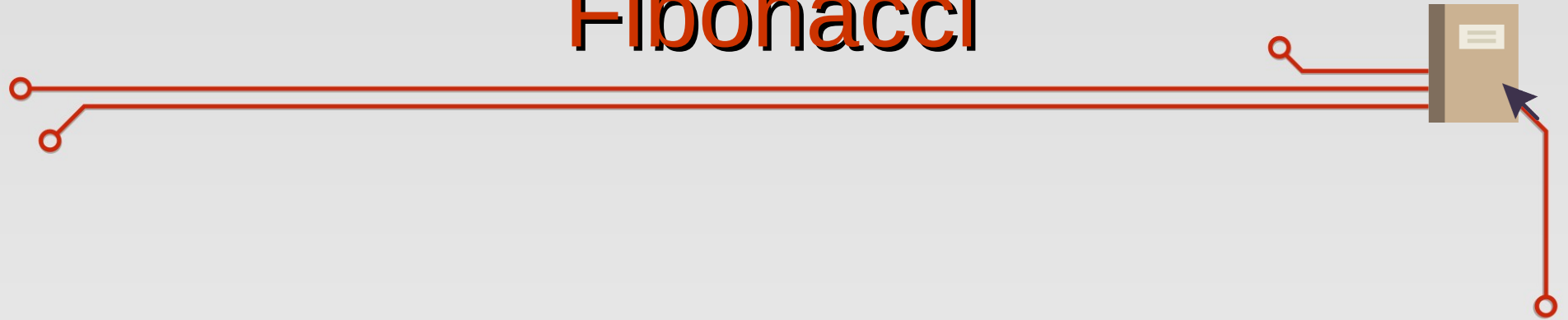
entrada = 0, salida = 0

entrada = 3, salida = 2

entrada = 4, salida = 3

entrada = 6, salida = 8

Fibonacci



¿Cómo se podría modificar el código para mejorar el rendimiento de esta función?

Para mejorar el rendimiento



Mediante un diccionario hash que
guarde valores ya calculados o
conocidos

Para mejorar el rendimiento

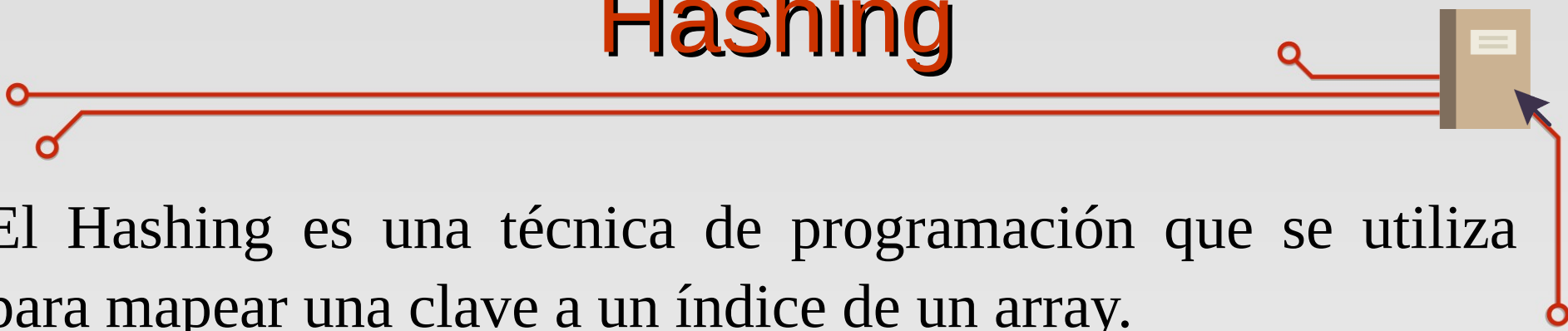


```
#include <unordered_map>
// Se asume que la clase tiene un arreglo, diccionario o hashmap
// llamado fibonacciCalcsSaved con los valores precalculados
std::unordered_map<int, int> fibonacciCalcsSaved;

int fibonacciWithMemorization(int n) {
    if (n <= 1) {
        return n;
    }
    //.count comprueba si la clave se guardó en el hashmap
    if (fibonacciCalcsSaved.count(n)) {
        return fibonacciCalcsSaved[n];
    }
    int result = fibonacciWithMemorization(n - 1) +
                  fibonacciWithMemorization(n - 2);
    //Se aprovecha que se calculó y se guarda
    fibonacciCalcsSaved[n] = result;

    return result;
}
```

Hashing



El Hashing es una técnica de programación que se utiliza para mapear una clave a un índice de un array.

The diagram shows a horizontal red line with several small circles at the ends and along its length. On the right side, a vertical brown rectangle represents a hash table. A red line connects one of the circles on the horizontal line to the hash table, and a blue arrow points from the hash table back to the horizontal line, illustrating the mapping process.

Es una forma eficiente de buscar y almacenar datos en una estructura de datos, como una tabla hash o un mapa en los lenguajes de programación.

La idea detrás del Hashing es transformar la clave en un índice numérico, que luego se utiliza para acceder al valor almacenado en esa posición en la tabla.

Ejemplos de Hashing



Tablas Hash: Es una estructura de datos que utiliza una función de hash para mapear claves a índices de un array. Se pueden usar para implementar diccionarios, mapas u otras estructuras de datos que requieren una búsqueda rápida de elementos.

Hash de contraseñas: En la seguridad informática, el Hashing se utiliza para almacenar contraseñas de forma segura. La función de hash (SHA-#) genera una representación de la contraseña que es segura para almacenar y se puede comparar con la contraseña ingresada por el usuario.

Ejemplos de Hashing



Indexación de documentos: Se utiliza en sistemas de indexación de documentos para representar palabras clave en documentos y para almacenar información sobre la frecuencia de estas palabras en los documentos.

Verificación de integridad de archivos: Se utiliza para verificar la integridad de archivos grandes descargados desde Internet. La función de hash se utiliza para calcular un valor hash para el archivo descargado y luego se compara con un valor hash conocido para verificar si el archivo se ha corrompido durante la descarga.

Ejemplos de Hashing



```
{  
  "marcadores": [  
    {  
      "latitude": 40.416875,  
      "longitude": -3.703308,  
      "city": "Madrid",  
      "description": "Puerta del Sol"  
    },  
    {  
      "latitude": 40.417438,  
      "longitude": -3.693363,  
      "city": "Madrid",  
      "description": "Paseo del Prado"  
    },  
    {  
      "latitude": 40.407015,  
      "longitude": -3.691163,  
      "city": "Madrid",  
      "description": "Estación de Atocha"  
    }  
  ]  
}
```

Diccionario

Nota: Aunque JSON no es técnicamente un diccionario, su estructura de pares clave-valor lo hace similar en muchos aspectos.

Hola Mundo

SHA-256

C3A4A2E49D91F217711
3A9ADFCB9EF9AF9679D
C4557A0A3A4602E1BD3
9A6F481

Peso de las funciones



- El tiempo complejidad o "peso" de una función se refiere a la cantidad de tiempo que toma para ejecutarse en función del tamaño de la entrada. La complejidad temporal suele expresarse utilizando la notación de big O, que indica el límite superior del tiempo de ejecución a medida que el tamaño de la entrada se acerca a infinito.
- Por ejemplo, una función que tiene una complejidad temporal de $O(1)$ es una función constante que siempre toma el mismo tiempo para ejecutarse, independientemente del tamaño de la entrada. Por otro lado, una función con una complejidad temporal de $O(n)$ es una función lineal que toma un tiempo proporcional al tamaño de la entrada.
- Es importante tener en cuenta que la complejidad temporal es una estimación y no es una medida exacta del tiempo de ejecución. Sin embargo, puede ser útil para comparar diferentes algoritmos y para tener una idea de cuánto tiempo se requerirá para ejecutar un algoritmo con diferentes tamaños de entrada.

Ejercicio 1



Implemente el factorial, tanto en su versión iterativa como en recursiva.

De acuerdo a lo anterior:

1) Abra una hoja de cálculo y realice la siguientes pruebas:

- 1) Para ambos algoritmos pruebe con números pequeños, medianos y grandes.
- 2) Registre en la hoja de cálculo los tiempos que tardan en completar la tarea.
- 3) Mediante un gráfico lineal, grafique los resultados de los tiempos.
- 4) Analice las gráficas y los datos. ¿Cómo es el crecimiento según la cantidad de datos?

Factorial (código)



```
int factorial(int n) {  
    if (n == 0) return 1;  
    return n * factorial(n - 1);  
}
```

```
int factorial(int n) {  
    int result = 1;  
    for (int i = n; i > 0; i--) {  
        result *= i;  
    }  
    return result;  
}
```

```
int main() {  
    int num;  
    scanf_s("%d", &num);  
    printf("%d\n", factorial(num));  
    return 0;  
}
```

Ejercicio 2



De igual manera implemente el algoritmo MergeSort.

De acuerdo a lo anterior:

1) Abra una hoja de cálculo y realice la siguientes pruebas:

- 1) Pídale a ChatGPT que le genere un vector 500 datos, luego 1000, 5000, 1 millón, etc.
- 2) Corra el algoritmo para cada cantidad de datos y registre en la hoja de cálculo los tiempos que tardan en completar la tarea.
- 3) Mediante un gráfico lineal, grafique los resultados de los tiempos.
- 4) Analice la gráfica y los datos. ¿Cómo es el crecimiento según la cantidad de datos?

Merge Sort (Código)

```
void mergeSort(int arr[], int start, int end) {  
    if (start < end) {  
        int middle = start + (end - start) / 2;  
        mergeSort(arr, start, middle);  
        mergeSort(arr, middle + 1, end);  
        merge(arr, start, middle, end);  
    }  
}
```

```
int main() {  
    int array[MAX] = { 5, 2, 4, 7, 1, 3, 2, 6 };  
  
    mergeSort(array, 0, MAX - 1);  
  
    for (int i = 0; i < MAX; i++) {  
        printf("%d ", array[i]);  
    }  
    printf("\n");  
    return 0;  
}
```

```
#include <iostream>  
using namespace std;  
const int MAX = 8;
```

Merge Sort (Código)

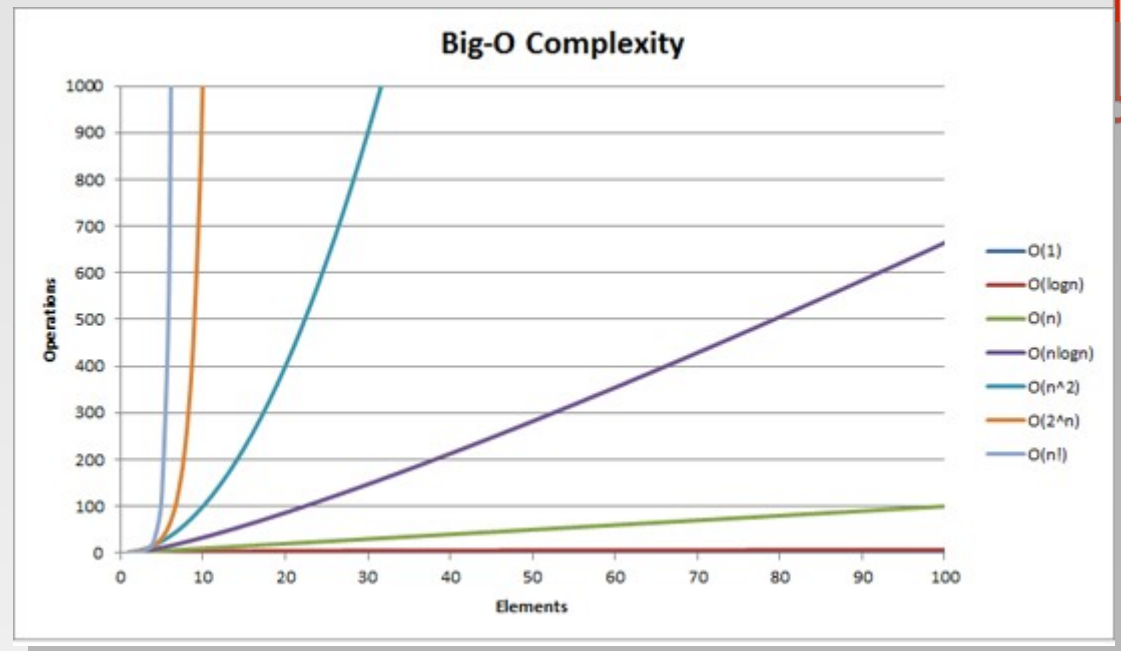


```
void merge(int arr[], int start, int middle, int end) {  
    int leftSize = middle - start + 1;  
    int rightSize = end - middle;  
    int* leftArray = new int[leftSize];  
    int* rightArray = new int[rightSize];  
  
    for (int i = 0; i < leftSize; i++) {  
        leftArray[i] = arr[start + i];  
    }  
    for (int i = 0; i < rightSize; i++) {  
        rightArray[i] = arr[middle + 1 + i];  
    }  
  
    int i = 0, j = 0, k = start;  
    while (i < leftSize && j < rightSize) {  
        if (leftArray[i] <= rightArray[j]) {  
            arr[k] = leftArray[i];  
            i++;  
        }  
        else {  
            arr[k] = rightArray[j];  
            j++;  
        }  
        k++;  
    }  
    while (i < leftSize) {  
        arr[k] = leftArray[i];  
        i++;  
        k++;  
    }  
    while (j < rightSize) {  
        arr[k] = rightArray[j];  
        j++;  
        k++;  
    }  
    delete[] leftArray;  
    delete[] rightArray;  
}
```


Los tipos más comunes de Big-O

Están ordenadas de la más rápida a la más lenta
N se refiere al número de operaciones

- $O(1)$ - Constant Runtime
- $O(\log N)$ - Logarithmic Runtime
- $O(N)$ - Linear Runtime
- $O(N \log N)$ - Linearithmic Runtime
- $O(N^2)$ - Quadratic Runtime
- $O(2^N)$ - Exponential Runtime
- $O(N!)$ - Factorial Runtime



- 1) Las funciones básicas (solo hacen una cosa, no importa si hacen otra) son de categoría 1.
- 2) Las funciones de un ciclo generalmente son de categoría n.
- 3) Las funciones de dos ciclos generalmente anidados son de n^2 .
- 4) Las funciones de tres ciclos generalmente anidados son de n^3 .
- 5) Las funciones recursivas generalmente son de notación $n!$.
- 6) Es importante notar que la eficiencia de un algoritmo no se mide en segundos. Se mide en el incremento en cantidad de operaciones necesarias.