Módulo 9

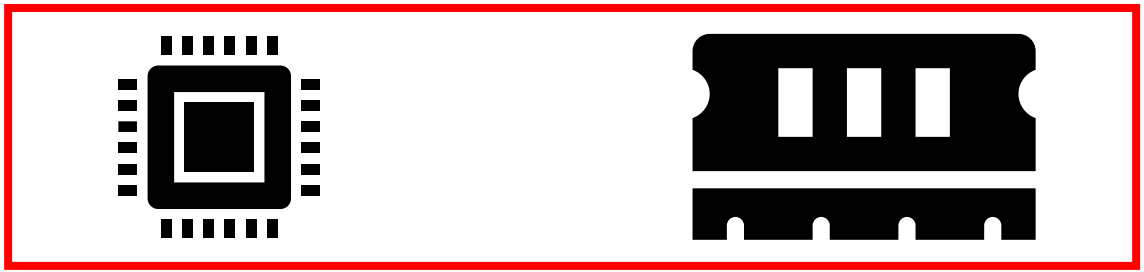
Análisis de Algoritmos

Algoritmos

El análisis de algoritmos es un procedimiento por el cual podemos **predecir los recursos** que requiere la ejecución de un algoritmo.

Los recursos evaluados generalmente son:

- Tiempo de procesamiento (Processing time)
- Memoria RAM
- Recursos externos (network, hard drive, server, database)



Big-O Notation

La **notación Big-O** es una notación matemática que describe el comportamiento de un algoritmo conforme incrementa la cantidad de elementos a procesar.

La notación Big-O te permite clasificar operaciones de acuerdo a su tasa de crecimiento.

¿Por qué es importante?

Es importante contar con un vocabulario preciso para hablar sobre cómo se comporta nuestro código.

Nos permite comparar distintos enfoques al resolver un problema.

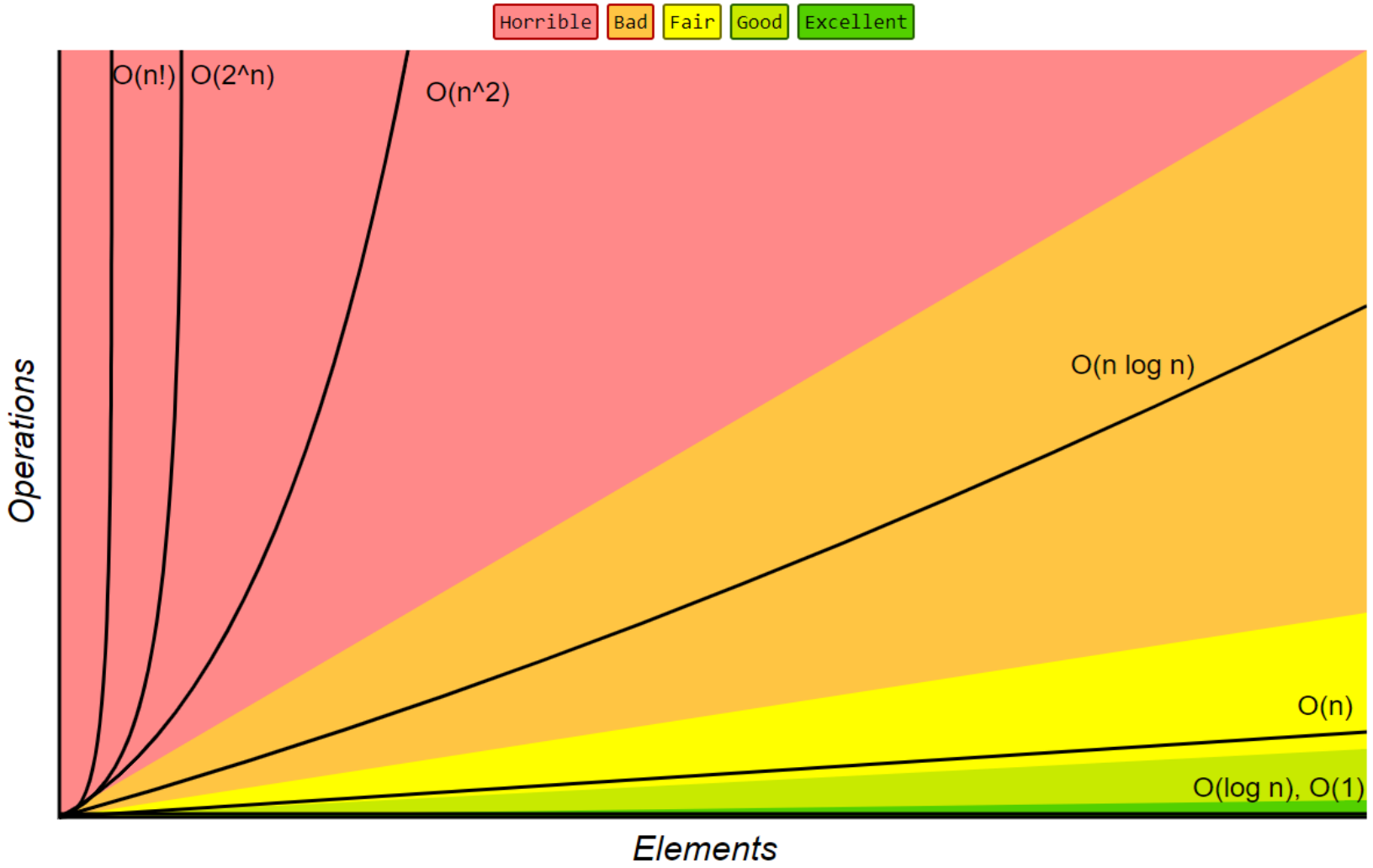
Cuando un programa se vuelve lento o falla, podemos identificar las partes menos eficientes de nuestra aplicación.

Aparece mucho en entrevistas!

Big-O Notation

Generalmente lo veremos expresado en función de n, que representa el número de elementos

- $O(1)$ = Constante
- $O(n)$ = Lineal
- $O(\log(n))$ = Logarítmico
- $O(n^2)$ = Cuadrático


Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Ejemplo

Eres el director de TI de una empresa local de noticias, implementando la funcionalidad de búsqueda en las noticias publicadas en su sitio web.

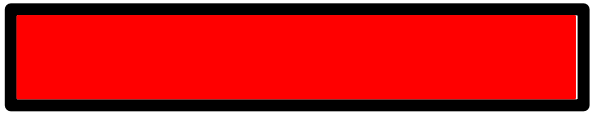
1er año
2000 noticias

2 minutos

2ndo año
4000 noticias

4 minutos

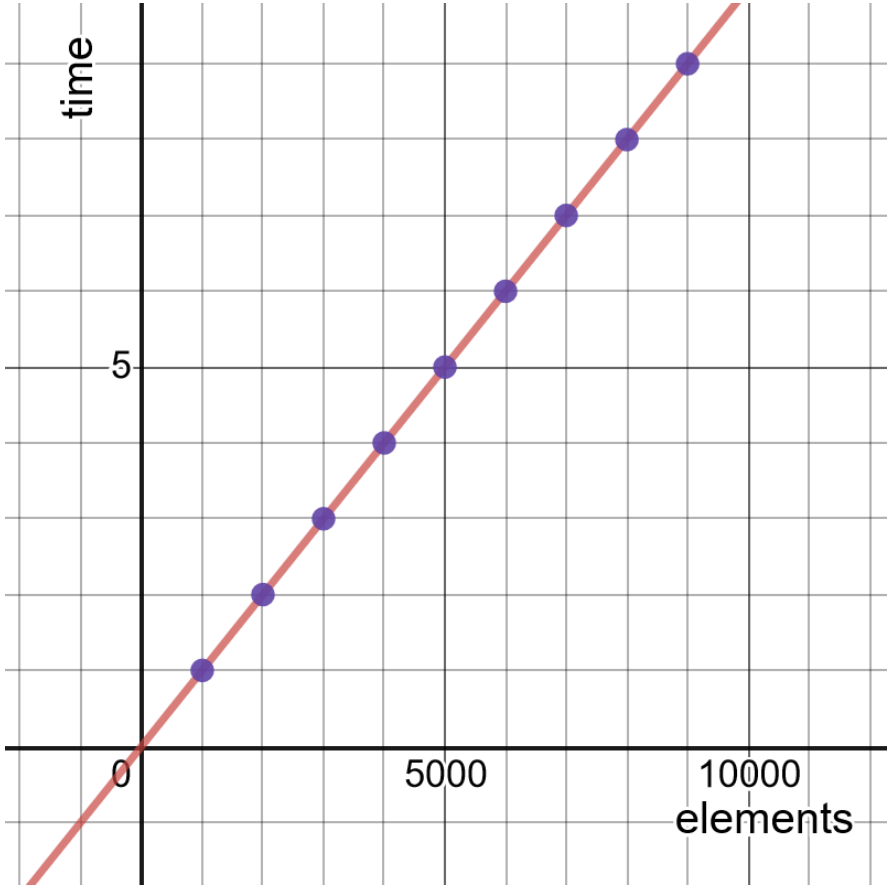
3er año
6000 noticias



6 minutos

4to año
8000 noticias

8 minutos



Crecimiento lineal

Por cada 1000 nuevas noticias publicadas, la búsqueda incrementa en 1 segundo.

La relación entre noticias y tiempo es lineal. El Big-O es O(n).

Big-O Notation

La notación Big-O se preocupa por representar el orden de la magnitud del algoritmo a ejecutar. Tomemos en cuenta las siguientes consideraciones:

1. Sólo tomaremos en cuenta el componente de mayor orden.
   - $O(6n) \rightarrow O(n)$
   - $O(4n + 3n + 10) \rightarrow O(n)$
   - $O(4n^2 + n) \rightarrow O(n^2)$
   - $O(n^3 + 10000n^2 + 9999n) \rightarrow O(n^3)$

2. Todas las instrucciones unitarias los mismos recursos (operaciones aritmeticas, comparaciones, actualizaciones, etc).
   - `int i = 0;`
   - `int i = 4+6;`
   - `System.out.println("Hola");`
   - `double var = Math.pow(3,100);`
   - `int j = ++4;`

Notación Big-O

¿Cuál es la notación Big-O del siguiente algoritmo?

```
public static void countDown(int n) {
  int i = 0;

  System.out.println("Begin");

  for(i=0; i<n; i++) {
    System.out.println(i);
  }

  System.out.println("End");
}
```

La declaración de una variable se puede realizar en tiempo constante. = 1 instrucción

Imprimir una palabra en consola una vez es constante. = 1 instrucción

- Inicializar i = 1 vez
- i<n = n+1 veces
- i++ = n veces
- Impresion de n= n veces

1 + n+1 + n + n

Imprimir una instrucción en consola una sola vez. = 1 instrucción

Al calcular la suma de cada operación obtenemos:

$= 3n + 5$

Y esto equivale a

$=O(n)$

https://rithmschool.github.io/function-timer-demo/

Fibonacci

¿Recursivo o Iterativo?

Recursivo

```
public static int fibonacci(int n) {
  if (n <= 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  }

  return fibonacci(n-1) + fibonacci(n-2);
}
```

Cuando n es menor o igual a 0

= 1 comparacion + 1 return
= O(1)

Recursivo

```
public static int fibonacci(int n) {
  if (n <= 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  }

  return fibonacci(n-1) + fibonacci(n-2);
}
```

Cuando n es igual a igual a 1

= 2 comparaciones + 1 return
= O(1)

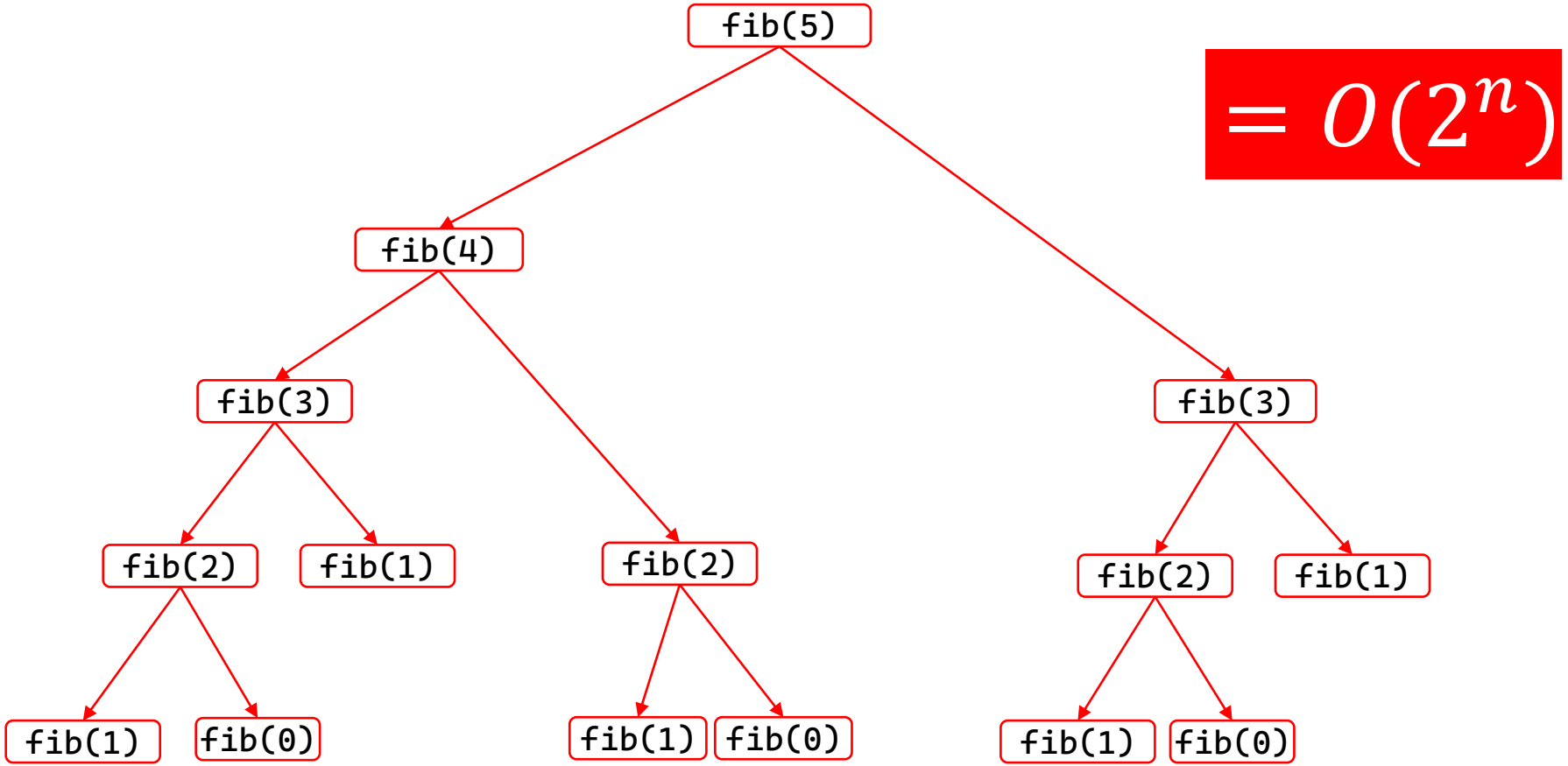
Recursivo

```
public static int fibonacci(int n) {
  if (n <= 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  }

  return fibonacci(n-1) + fibonacci(n-2);
}
```

Cuando n es mayor a 1

- 2 comparaciones
- 1 return
- 2 restas
- 1 suma
- fibonacci(n-1)
- fibonacci(n-2).

= O(1) + fibonacci(n-1) + fibonacci(n-2)


fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(1)
fib(2)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)
$= O(2^n)$

Recursivo

```
public static int fibonacci(int n) {
  if (n <= 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  }

  return fibonacci(n-1) + fibonacci(n-2);
}
```

Cuando n es mayor a 1

- 2 comparaciones
- 1 return
- 2 restas
- 1 suma
- fibonacci(n-1)
- fibonacci(n-2)

$= 4 + O(2^n) + O(2^n) \rightarrow O(2^n)$

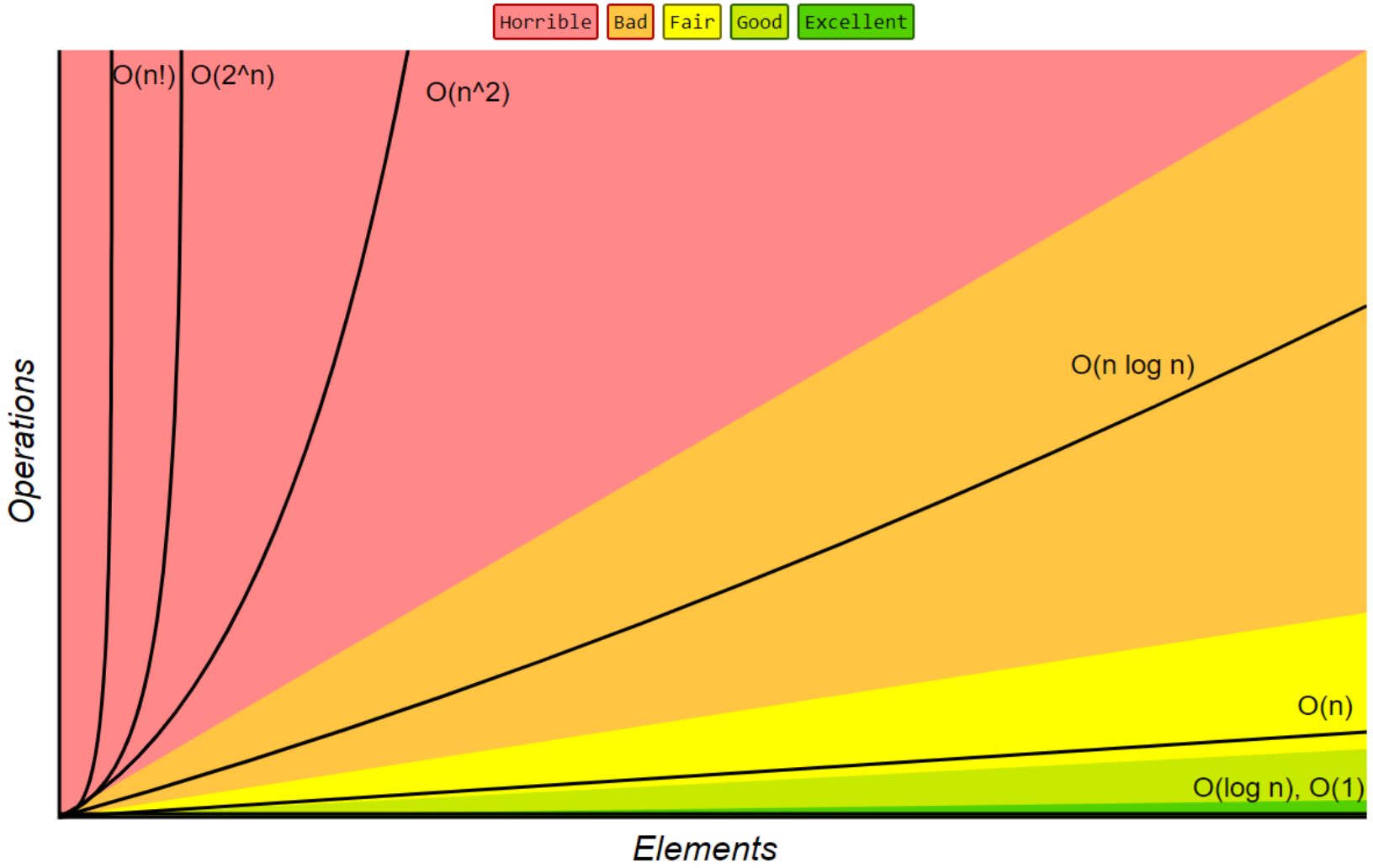
$$= O(2^n)$$

Iterativo

```
public static int fibonacciIterative(int n) {

  int[] calc = {0,0};

  if (n <= 0) {
    calc[0] = 0;
    calc[1] = 0;
  } else if (n >= 1) {
    calc[0] = 0;
    calc[1] = 1;
  }

  for (int i=2; i<=n; i++) {
    int newNumber = calc[0] + calc[1];
    calc[0] = calc[1];
    calc[1] = newNumber;
  }

  return calc[1];
}
```

- O(1)
- O(n)

= $O(n)$ + O(1) → O(n)


Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Ejemplo: Búsqueda secuencial

La búsqueda secuencial itera sobre cada uno de los elementos de un arreglo tamaño n.

¿Cuál es el mejor escenario? El elemento que estamos buscando se encuentra en la primera posición del arreglo.

Por lo tanto, la búsqueda secuencial en el mejor escenario corresponde a:

Best case complexity: O(1)

Ejemplo: Búsqueda secuencial

¿Cuál sería el escenario promedio? El elemento que buscamos se encuentra a la mitad del arreglo, por lo tanto debemos recorrer n/2 elementos. La Big-O notation sólo se interesa por representar el orden del algoritmo, por lo que podemos descargar n/2 y asumir directamente la complejidad O(n).

Average case complexity: O(n)

¿Cuál sería el peor escenario? El elemento que estamos buscando no se encuentra en el arreglo, por lo tanto, debemos recorrer los **n** elementos:

Worst case complexity: O(n)

Ejemplo: Búsqueda binaria

La búsqueda binaria divide el arreglo en mitades conforme va buscando elementos.

¿Cuál es el mejor escenario? El elemento que estamos buscando se encuentra en la mitad del arreglo, por lo que lo encontramos con una sola comparación.

Por lo tanto, la búsqueda binaria en el mejor escenario corresponde a:

Best case complexity: O(1)

Ejemplo: Búsqueda binaria

¿Cuál sería el escenario promedio y el peor escenario? Link

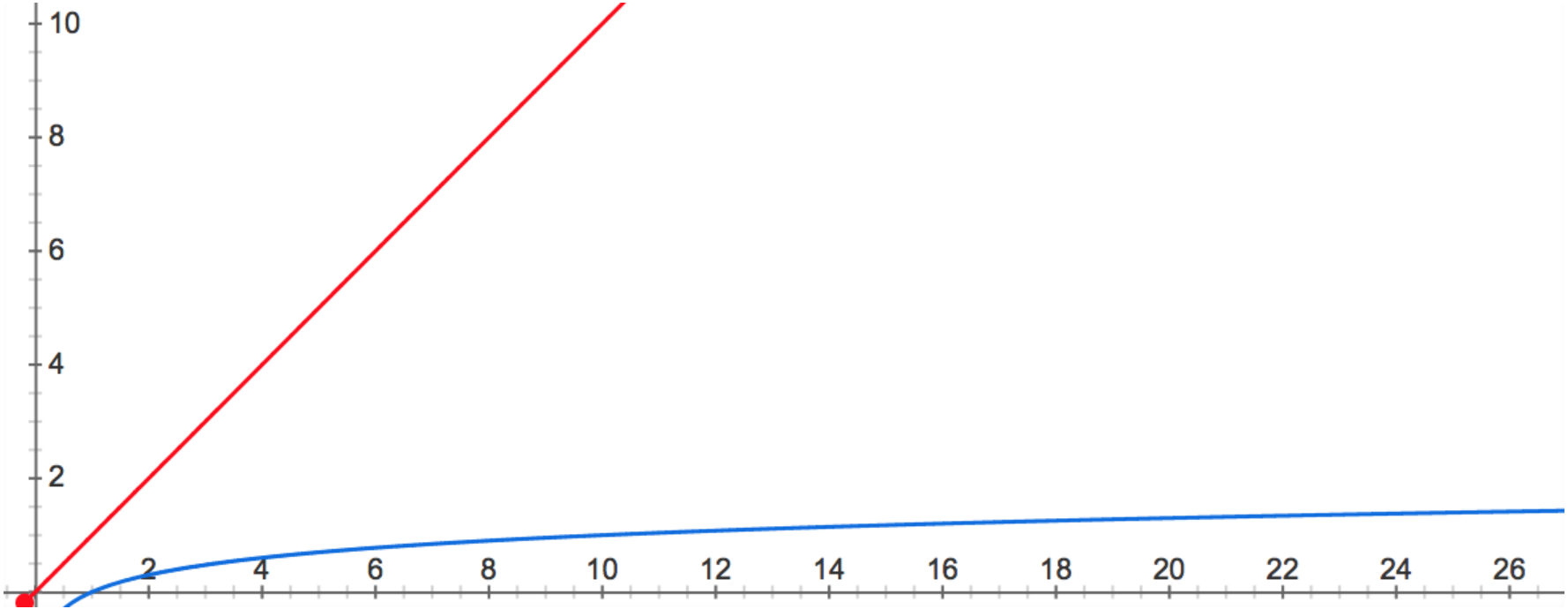
La búsqueda en un arreglo ordenada escala logarítmicamente, por lo que se aproxima a:

Average case complexity = O(log(n)).

El elemento no se encuentra en el arreglo, por lo tanto debemos recorrer todas las subdivisiones del arreglo. Esto equivale a:

Worst case complexity = O(log(n))

Lineal search vs Binary Search



Búsqueda binaria (azul)

Búsqeda secuencial (rojo)

Big-O Notation: Bubble Sort

```java
public static void bubbleSort(int[] x) {

    //indicates no swaps where done during the run
    boolean flagSwap = false;

    //iterate from 0 to length -1 to avoid outOfBounds
    //include "no swaps" exit condition
    for (int i = 0; i < x.length - 1 && !flagSwap; i++) {
        flagSwap = true;

        //On every run, we can assume everything after x.length - i
        // is already sorted
        for (int j = 0; j < x.length - 1 - i; j++) {
            if (x[j] > x[j + 1]) {
                swap(x, j, j + 1);
                flagSwap = false;
            }

        }
    }

}
```

Ciclo exterior se ejecuta (n-1) veces, lo que se aproxima a (n) veces

Ciclo interior se ejecuta (n-1-i) veces, lo que se aproxima a (n) veces

Por lo tanto, podemos concluir que el Bubble Sort es en promedio un algoritmo $O(n^2)$

Big-O Notation: Bubble Sort

¿Cuál es el mejor escenario de un Bubble Sort? El arreglo ya está ordenado, por lo tanto sólo es necesario recorrer el arreglo 1 vez.

Best case complexity = O(n)

El caso promedio y el peor caso son iguales, es necesario recorrer el arreglo e ir intercambiando los índices hasta que tengamos un arreglo ordenado. Al tener un par de ciclos anidados, la complejidad del algoritmo escala de manera cuadrática.

Average case complexity = $O(n^2)$.

Worst case complexity = $O(n^2)$

Big-O Notation

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Big-O Notation

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Sunjay liked

jwcarroll
@jwcarroll

Alternative Big O notation:

$O(1) = O(yeah)$
$O(\log n) = O(nice)$
$O(n) = O(ok)$
$O(n^2) = O(my)$
$O(2^n) = O(no)$
$O(n!) = O(mg!)$

8:10 PM · 06 Apr 19 · Twitter for Android

3,665 Retweets **9,265** Likes