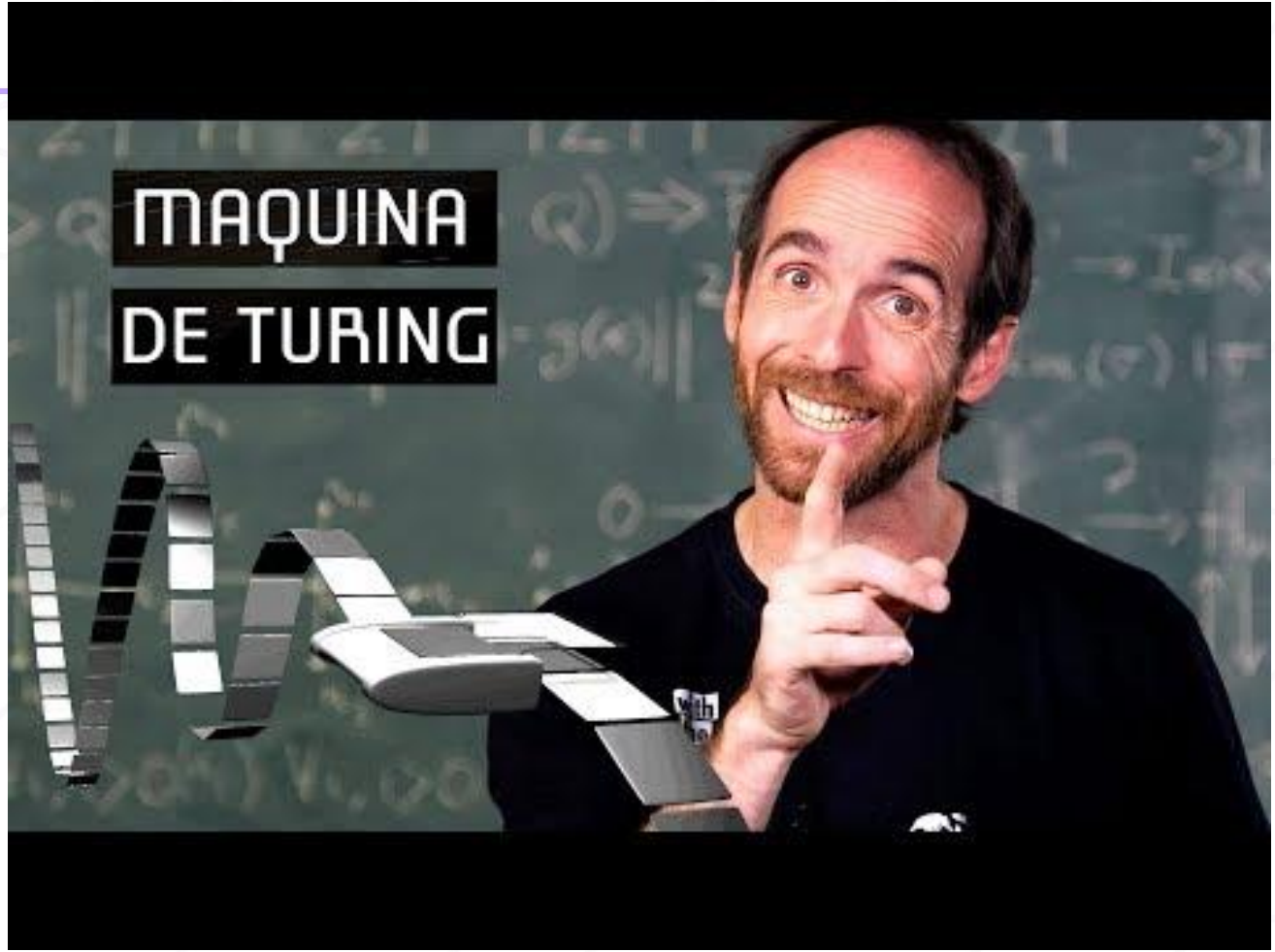
Complejidad Computacional

DEV.F::
DESARROLLAMOS(PERSONAS);


MAQUINA
DE TURING

Tipos de problemas

Optimización

Son aquellos en los cuales se busca minimizar o maximizar el valor de una solución en un grupo de soluciones generadas para una entrada específica.

Decisión

Son aquellos donde se busca una respuesta de ‘‘si’’ o ‘‘no’’.

Cualquier problema de optimización puede ser manejado como un problema de decisión definiendo un valor objetivo y preguntando si existe o no una solución factible en el conjunto de soluciones con un valor de la solución máximo o mínimo según sea el caso. En ese sentido es más conveniente tratar con problemas de decisión que con problemas de optimización.

Clasificación de los problemas de decisión

Desde la computabilidad

Decidibles o resolubles mediante algoritmos: Si existe un procedimiento, este se detiene ante cualquier entrada.

Parcialmente Decidible o reconocible: Si existe un procedimiento este se detiene para las entradas que son solución.

No decidible

Clasificación de los problemas de decisión

Desde la complejidad computacional

- **Clase L (D Log Space):** Son aquellos problemas que pueden ser resueltos por una MT determinista utilizando una cantidad de espacio o memoria logarítmicamente proporcional al tamaño de la entrada: log(n), donde n es el tamaño de la entrada; en estos problemas si existe una solución es única.

- **Clase NL (Nondeterministic Logarithmic space):** Son los problemas de decisión que pueden ser resueltos en espacio log(n), por una MT no determinista tal que la solución si existe es única.

Clasificación de los problemas de decisión

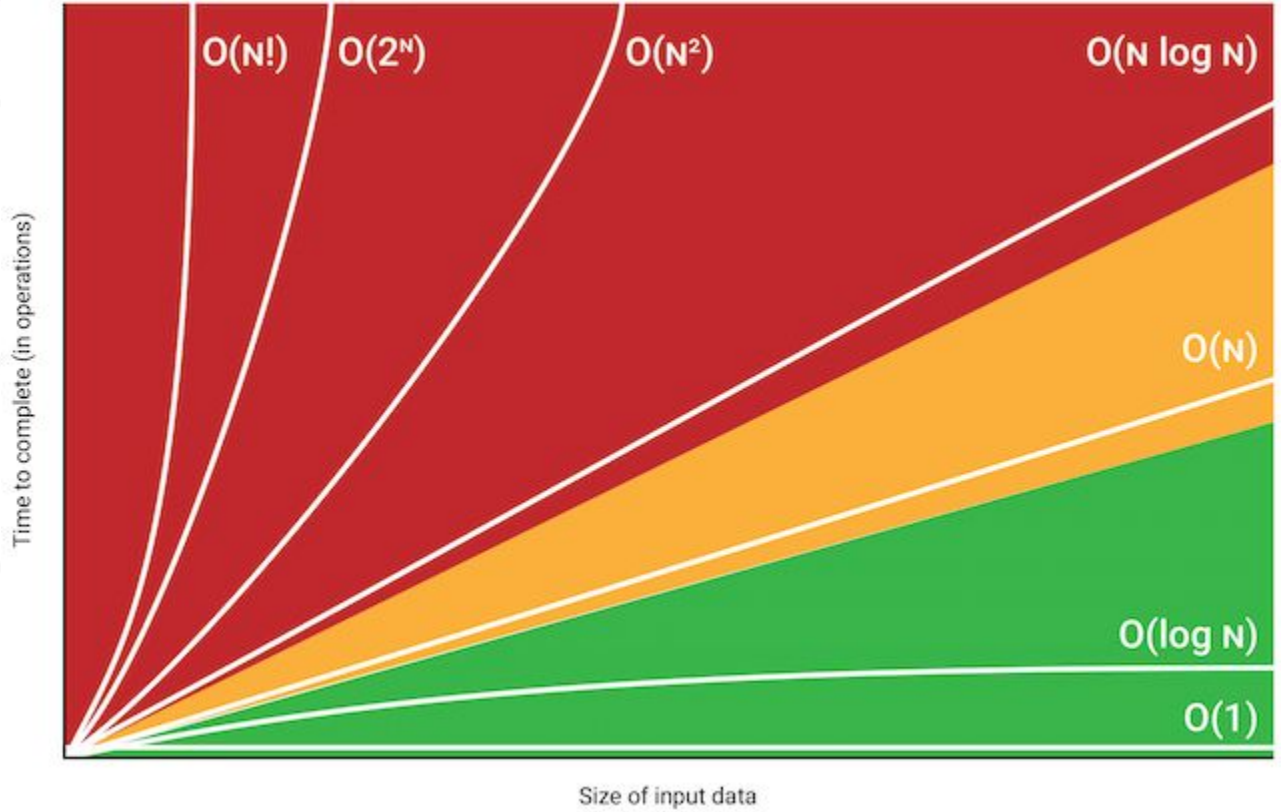
Desde la complejidad computacional

- **Clase P (Polynomial Time):** Son todos aquellos problemas de decisión que pueden ser resueltos por una MT determinista en un período de tiempo polinómico. Estos problemas son tratables, es decir se pueden resolver en tiempos razonables; buena parte de los problemas de ordenamiento, priorización y búsqueda caben dentro de esta clase.

- **Clase NP (Non Deterministic Polynomial Time):** Formalmente, es el conjunto de problemas que pueden ser resueltos en tiempo polinómico por una MT no determinista. Una definición más intuitiva es la siguiente: Es el conjunto de los problemas de decisión para los cuales las instancias donde la respuesta es “si” tienen demostraciones verificables por cálculos determinísticos en tiempo polinómico.

Big O notation

DEV.F::
DESARROLLAMOS(PERSONAS);


O(N!)
O(2^N)
O(N^2)
O(N log N)
O(N)
O(log N)
O(1)
Time to complete (in operations)
Size of input data

Ejemplos de Big O

O(1) - notación constante

Esta expresión indica **tiempo constante**, lo que significa que el algoritmo se ejecutará con el **mismo rendimiento sin importar el tamaño del input**. Esto no quiere decir que sólo tendrá un input, más bien no se verá afectado por la cantidad de datos que estemos manejando.

```
function findByIndex(food, index) {
  return food[index];
}

var food = ['🍿', '🍔', '🍩', '🍗'];

console.log(findByIndex(food, 2));   🍩
```

Ejemplos de Big O

O(n) - notación lineal

Esta es la expresión de crecimiento lineal, la complejidad del algoritmo **aumenta de manera proporcional al tamaño** del input.

```
function selectedFood(food) {
  food.forEach(objectFood =>
       console.log(objectFood, objectFood)
  );
}

const food = ['🍿', '🍔', '🍩', '🍗'];

selectedFood(food); // 🍿🍿 🍔🍔 🍩🍩 🍗🍗
```

DEV.F

Ejemplos de Big O

O(n^2) - notación cuadrática

Indica que el crecimiento en complejidad es **proporcional al cuadrado del tamaño del input**. Estos algoritmos suelen ser los menos eficientes, **no se recomiendan para datos grandes** y normalmente se dan cuando **usamos ciclos for o iteraciones anidadas**; es decir, si dentro de cada iteración de un arreglo lo vuelves a iterar, tendrás un algoritmo de complejidad cuadrada. Estos pueden llegar a complejidades cúbicas o factoriales.

```
function bubbleSort(array){
  array = array.slice();
  for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length - 1; j++) {
      if (array[j] > array[j + 1]) {
        [array[j], array[j + 1]] = [array[j + 1], array[j]];
      }
    }
  }
  return array;
}

var array = [ 4, 3, 2, 1];

console.log(bubbleSort(array)); // [ 1, 2, 3, 4]
```

DEV.F

Ejemplos de Big O

O(logn) - notación logaritmica

Indica que **el tiempo aumenta linealmente**, mientras que **n sube exponencialmente**. Entonces, si se tarda 1 segundo en calcular 10 elementos, se necesitarán 2 para 100, 3 para 1000 y así sucesivamente.

```
function binarySearch(array, element, start = 0, end = (array.length -
1)) {
  if (end < start) return -1;
  var middle = Math.floor((start + end) / 2);
  return element === array[middle]
    ? middle
    : element < array[middle]
      ? binarySearch(array, element, start, middle - 1)
      : binarySearch(array, element, middle + 1, end);
}


var unsortedArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];


console.log("Index of 2: ", binarySearch(unsortedArray, 2));    //
Index of 2: 1
console.log("22 not found: ", binarySearch(unsortedArray, 22)); // 22
not found: -1
```

FAST

SLOW

# OF BOXES	O(Log n)	O(n)	O(n Log n)	O(n²)	O(n!)
16	0.4 sec	1.6 sec	6.4 sec	25.6 sec	6630 years
256	0.8 sec	25.6 sec	3.4 min	1.8 hrs	8.6×10^{506} years
1024	1.0 sec	1.7 min	17 min	1.2 days	5.4×10^{2659} years

Calculando la Big O

DEV.F:
DESARROLLAMOS(PERSONAS);

Elimina las constantes

```
const numbers = [1, 2, 3, 4, 5];

function logEverythingTwice(items) {
    for (let i = 0; i < items.length; i++) { // O(n)
        console.log(items[i]); // O(1)
    }

    for (let i = 0; i < items.length; i++) { // O(n)
        console.log(items[i]); // O(1)
    }
}
```

Elimina las constantes

```
const numbers = [1, 2, 3, 4, 5];

function logEverythingFiveTimes(items) {
  for (let i = 0; i < items.length; i++) { // O(n)
    for (let j = 0; j < items.length; j++) { // O(n)
      console.log(items[i]) // O(1)
    }
  }
}
```

Elimina los términos no dominantes

```
const numbers = [1, 2, 3, 4, 5];

function printMultiplesThenSum(items) {
  for (let i = 0; i < items.length; i++) { // O(n)
    for (let j = 0; j < items.length; j++) { // O(n)
      console.log(items[i]); // O(1)
    }
  }

  const sum = items.reduce((acc, item) => { // O(n)
    return acc += item; // O(1)
  }, 0);

  return sum; // O(1)
}
```

Complejidad algorítmica: ejercicio (resuelto)

Indicar, con notación "Big-O", la complejidad algorítmica (respecto al tiempo) de cada fragmento de código mostrado a continuación.

```
for i = 0 to N:
    if i % 2 == 0:
        print i
```

O(N)

```
for i = 0 to N:
    for j = i to N:
        print i + "," + j
```

$O(N^2)$

```
for i = 0 to length(A):
    for j = 0 to length(B):
        print A[i] + "," + B[j]
```

O(A*B)

```
for i = 0 to N:
    j = 1
    while j < N:
        print j
        j = j * 2
```

O(N*log N)

```
for i = 0 to N:
    if i % 2 == 0:
        print i
for i = 0 to N:
    if i % 2 != 0:
        print i
```

O(N)

```
i = 1
while (i < N):
    print i
    i = i * 2
```

O(log N)

```
N = 1000
if N % 2 == 0:
    print "par"
else:
    print "impar"
```

O(1)