



**UNRN**

Universidad Nacional  
de Río Negro



| unrionegro

# Funciones Básicas de la Complejidad del Tiempo

- En un orden creciente de complejidad:
  - Tiempo Constante :  $O(1)$
  - Tiempo Logarítmico :  $O(\log n)$
  - Tiempo Lineal :  $O(n)$
  - Tiempo Polinomial :  $O(n^2)$
  - Tiempo Exponencial :  $O(2^n)$
- Supóngase que cada paso toma 1 microsegundo ( $10^{-6}$ ):

n	$O(1)$	$O(\log n)$	$O(n)$	$O(n^2)$	$O(2^n)$
10	1	1	10	100	1024
100	1	2	100	10000	40196936841331500 years
1000	1	3	1000	1 sec	...
10000	1	4	10000	1.67 min	

## Unidades de Tiempo para calcular

---

- 1 por la asignación.
  - 1 asignamiento,  $n+1$  pruebas, y  $n$  incrementos.
  - $n$  iteraciones de 3 unidades por un asignamiento, una suma, y una multiplicación.
  - 1 por la sentencia return.
- 

**Total:**  $1+(1+n+1+n)+3n+1$   
 $= 5n+4 = O(n)$

```
int sum (int n)
{
    int partial_sum = 0;
    for (int i = 1; i <= n; i++)
        partial_sum = partial_sum + (i * i);
    return partial_sum;
}
```

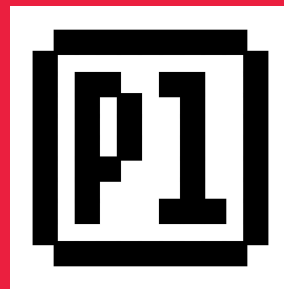
# Algoritmos de búsqueda y ordenamiento

**UNRN**

Universidad Nacional  
de Río Negro

**15**

**2023**



# Algoritmos de búsqueda:

- Secuencial
- Binaria

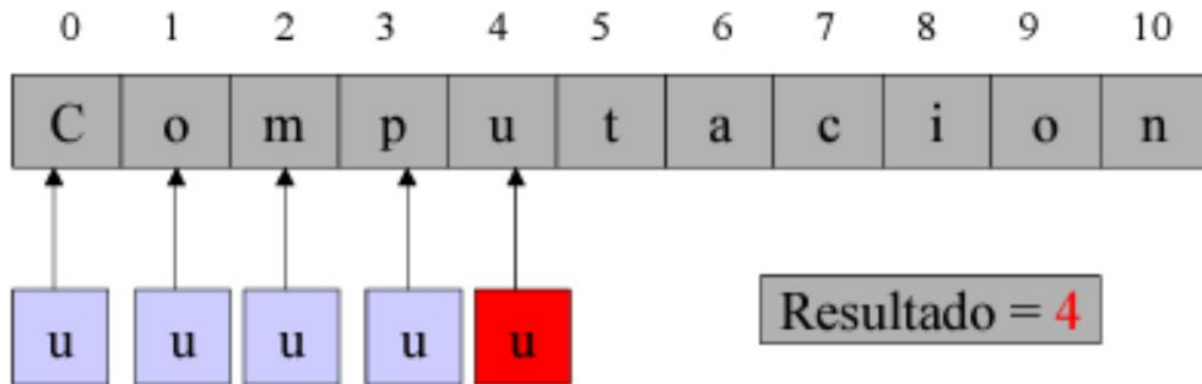
---

# Búsqueda Secuencial

# Búsqueda secuencial

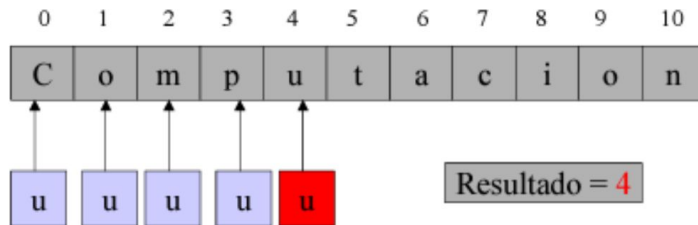
- Consiste en ir comparando el elemento que se busca con cada elemento del arreglo hasta cuando se encuentra.
- Busquemos el elementos 'u'

Busquemos el elementos **u**



# Búsqueda secuencial (Arreglo)

Busquemos el elementos **u**



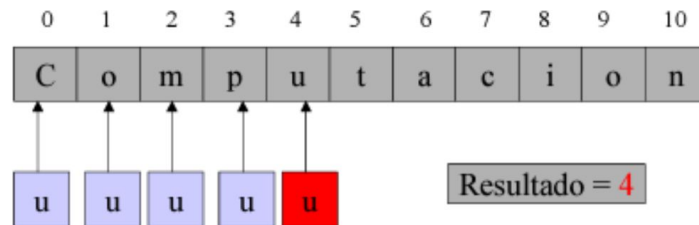
```
int busquedaSecuencial(int arreglo[], int longitud,
int valor) {
    for (int i = 0; i < longitud; i++) {
        if (arreglo[i] == valor) {
            return i;
        }
    }
    return -1;
}
```



# Búsqueda secuencial (Lista enlazada)

Busquemos el elemento **u**

```
int busquedaSecuencial(struct Nodo* cabeza, int valor) {  
    int indice = 0;  
    struct Nodo* actual = cabeza;  
    while (actual != NULL) {  
        if (actual->dato == valor) {  
            return indice;  
        }  
        actual = actual->siguiente;  
        indice++;  
    }  
    return -1;  
}
```



# Búsqueda secuencial

## Eficiencia y Complejidad

Considerando la Cantidad de Comparaciones

Mejor Caso:

- El elemento buscado está en la primera posición. Es decir, se hace una sola comparación

Peor Caso:

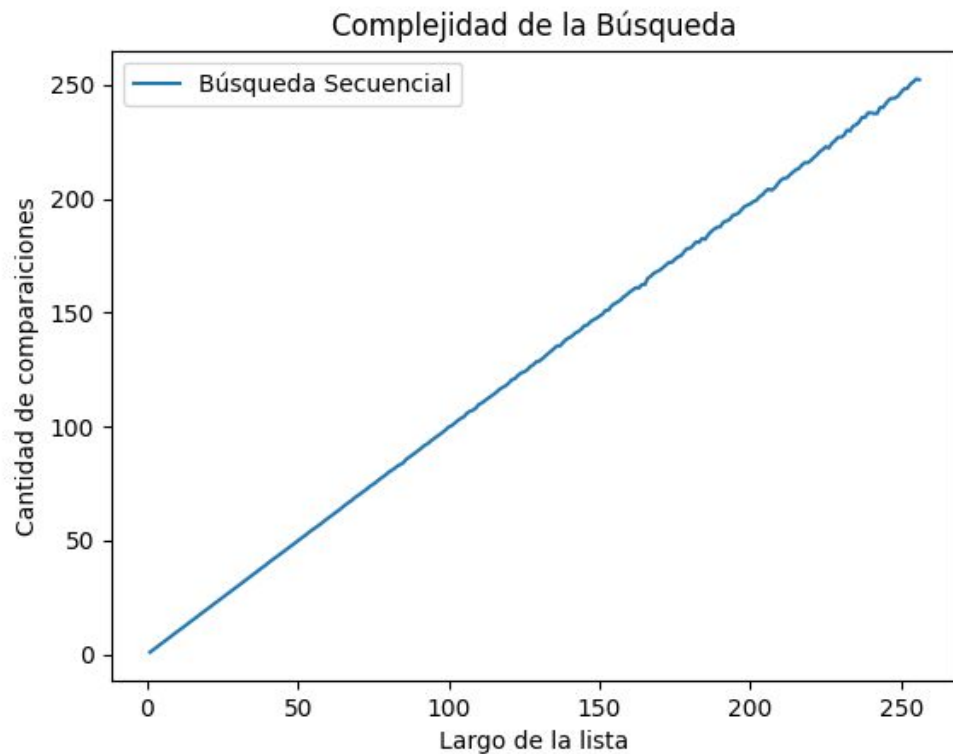
- El elemento buscado está en la última posición. Necesitando igual cantidad de comparaciones que de elementos el arreglo

En Promedio:

- El elemento buscado estará cerca de la mitad. Necesitando en promedio, la mitad de comparaciones que de elementos

Por lo tanto, la velocidad de ejecución depende linealmente del tamaño del arreglo

# Búsqueda secuencial

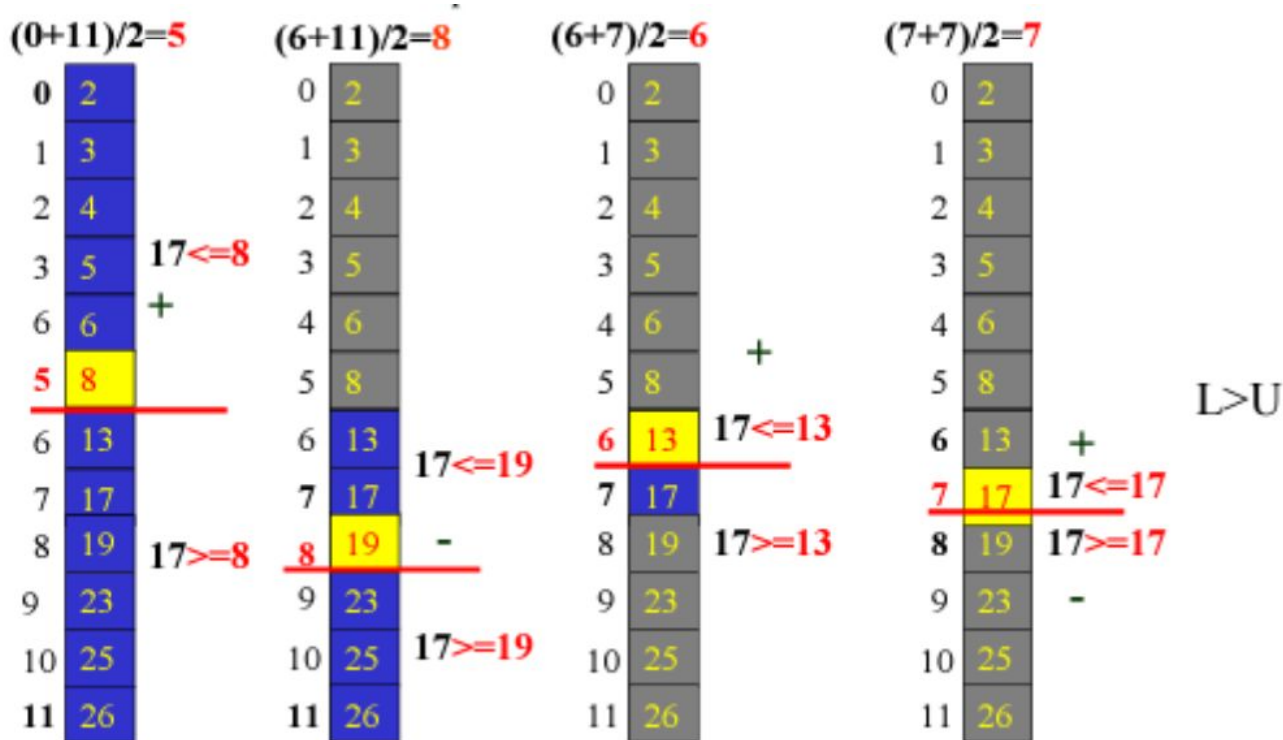


---

# Búsqueda Binaria

# Búsqueda binaria

- Siempre debe estar ordenado



# Búsqueda binaria

```
int busquedaBinaria(int arreglo[], int longitud, int valor) {  
    int izquierda = 0;  
    int derecha = longitud - 1;  
    while (izquierda <= derecha) {  
        int medio = izquierda + (derecha - izquierda) / 2;  
        if (arreglo[medio] == valor) {  
            return medio;  
        }  
        if (arreglo[medio] < valor) {  
            izquierda = medio + 1;  
        } else {  
            derecha = medio - 1;  
        }  
    }  
    return -1;  
}
```

$(0+11)/2=5$

0	2	
1	3	
2	4	
3	5	$17 \leq 8$
6	6	+
5	8	
6	13	
7	17	
8	19	$17 \geq 8$
9	23	
10	25	
11	26	

# Búsqueda binaria

## Eficiencia y Complejidad

Contando Comparaciones

Mejor Caso:

El elemento buscado está en el centro. Por lo tanto, se hace una sola comparación

Peor Caso:

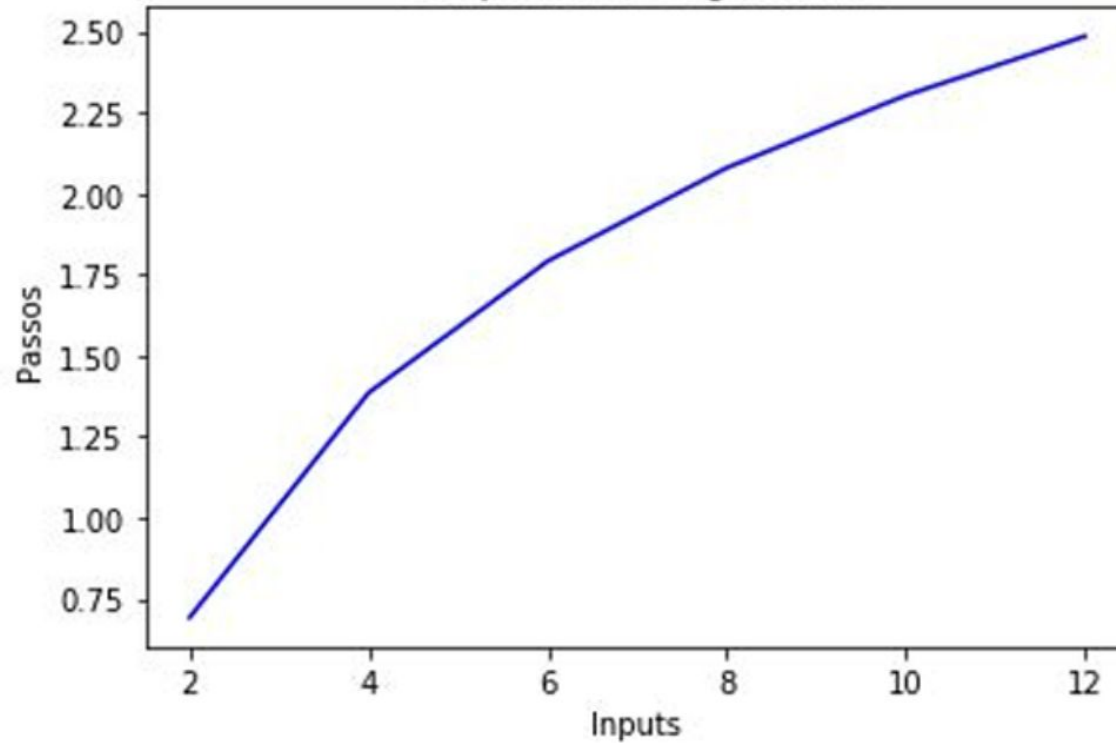
El elemento buscado está en una esquina. Necesitando  $\log_2(n)$  cantidad de comparaciones

En Promedio:

Serán algo como  $\log_2(n/2)$

Por lo tanto, la velocidad de ejecución depende logarítmicamente del tamaño del arreglo

# Búsqueda binaria







**¿Preguntas?**

# Algoritmos de Ordenamiento:

- Burbuja
- Inserción
- Selección
- Merge Sort.
- Quick Sort.
- Bogosort

---

# Burbuja

# Burbuja

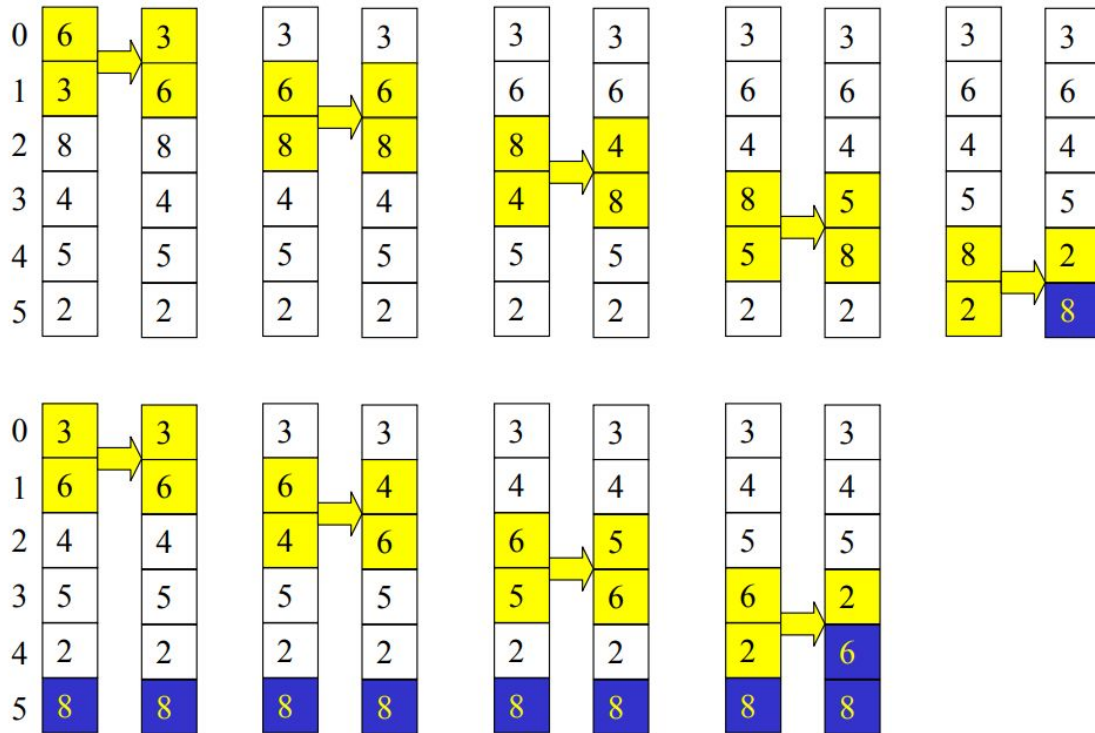
Ordenamiento Burbuja (bubblesort):

Idea: vamos comparando elementos adyacentes y empujamos los valores más livianos hacia arriba (los más pesados van quedando abajo).

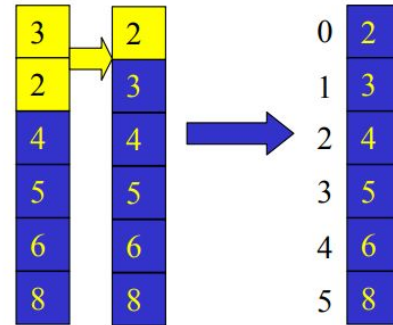
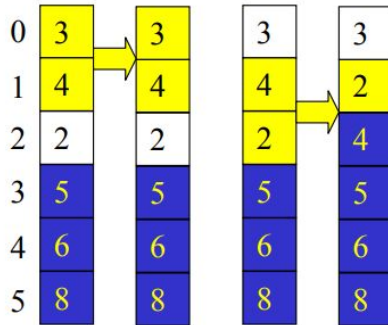
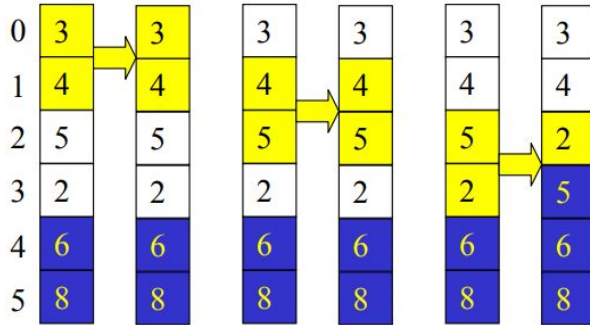
Idea de la burbuja que asciende, por lo liviana que es.

0	3
1	6
2	8
3	4
4	5
5	2

# Burbuja



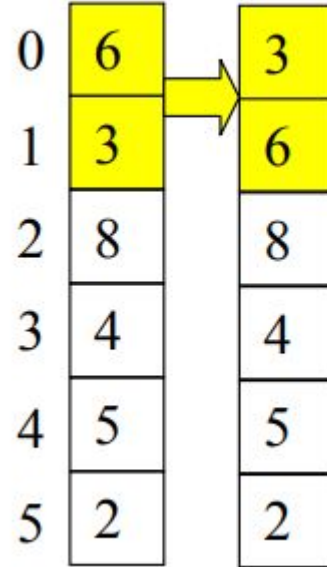
# Burbuja



# Burbuja

```
void intercambiar(int* a, int* b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
void ordenamientoBurbuja(int arreglo[], int longitud) {  
    for (int i = 0; i < longitud - 1; i++) {  
        for (int j = 0; j < longitud - 1 - i; j++) {  
            if (arreglo[j] > arreglo[j + 1]) {  
                intercambiar(&arreglo[j], &arreglo[j + 1]);  
            }  
        }  
    }  
}
```



0	6	→	3
1	3		6
2	8		8
3	4		4
4	5		5
5	2		2

# Burbuja

## Complejidad y Eficiencia

- Cantidad de Comparaciones:

- Constante:  $n*(n+1)/2$

- Cantidad de Intercambios:

- Mejor Caso:

- Arreglo ordenado. Por lo tanto, no se hace ni un solo swap

- Peor Caso:

- Arreglo ordenado inversamente. Se necesitarán  $n*(n+1)/2$  cantidad de swaps

- En Promedio:

- Serán algo como  $n*(n+1)/4$  swaps

- Por lo tanto, la velocidad de ejecución depende cuadráticamente del tamaño del arreglo



---

# Inserción

# Inserción

54	26	93	17	77	31	44	55	20
----	----	----	----	----	----	----	----	----

Se asume que 54 es una lista ordenada de 1 ítem

26	54	93	17	77	31	44	55	20
----	----	----	----	----	----	----	----	----

Se inserta 26

26	54	93	17	77	31	44	55	20
----	----	----	----	----	----	----	----	----

Se inserta 93

17	26	54	93	77	31	44	55	20
----	----	----	----	----	----	----	----	----

Se inserta 17

17	26	54	77	93	31	44	55	20
----	----	----	----	----	----	----	----	----

Se inserta 77

17	26	31	54	77	93	44	55	20
----	----	----	----	----	----	----	----	----

Se inserta 31

17	26	31	44	54	77	93	55	20
----	----	----	----	----	----	----	----	----

Se inserta 44

17	26	31	44	54	55	77	93	20
----	----	----	----	----	----	----	----	----

Se inserta 55

17	20	26	31	44	54	55	77	93
----	----	----	----	----	----	----	----	----

Se inserta 20

# Inserción

```
void Insertion_sort(int* t)
{
    int i, j;
    int actual;

    for (i = 1; i < 20; i++) {
        actual = t[i];
        for (j = i; j > 0 && t[j - 1] > actual; j--) {
            t[j] = t[j - 1];
        }
        t[j] = actual;
    }
}
```

17	26	54	77	93	31	44	55	20
----	----	----	----	----	----	----	----	----

Se necesita insertar 31 de vuelta en la lista ordenada

17	26	54	77		93	44	55	20
----	----	----	----	--	----	----	----	----

93>31, entonces debe desplazarse a la derecha

17	26	54		77	93	44	55	20
----	----	----	--	----	----	----	----	----

77>31, entonces debe desplazarse a la derecha

17	26		54	77	93	44	55	20
----	----	--	----	----	----	----	----	----

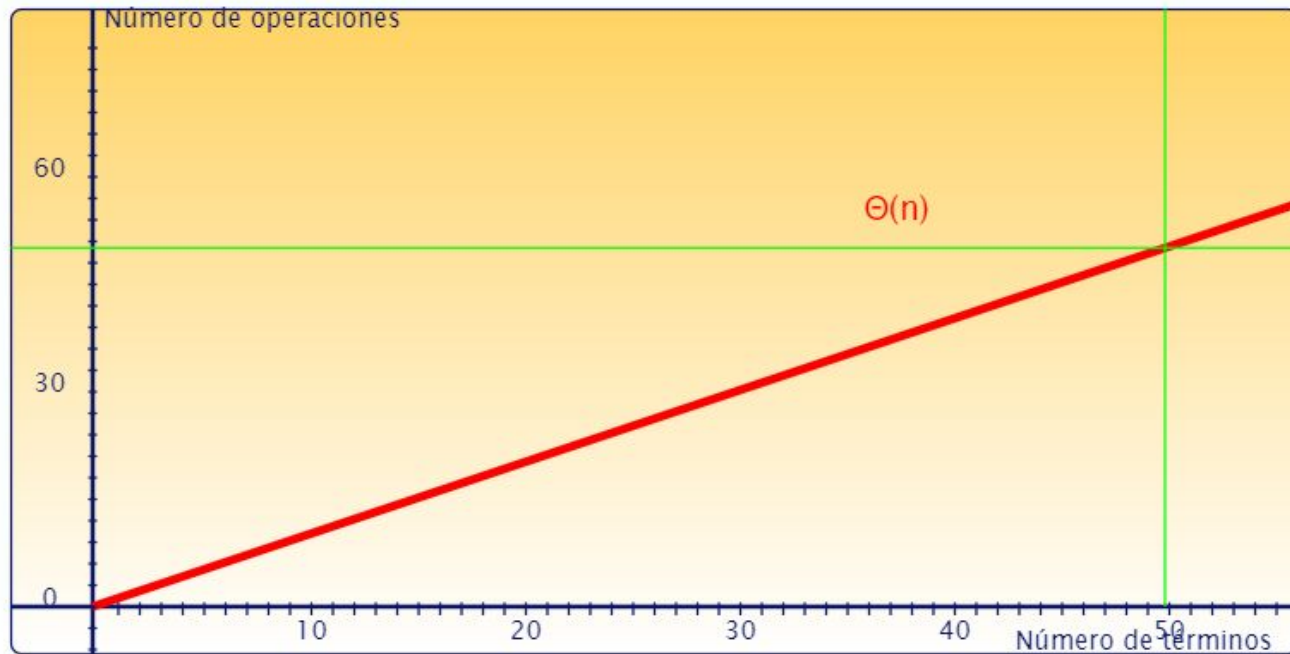
54>31, entonces debe desplazarse a la derecha

17	26	31	54	77	93	44	55	20
----	----	----	----	----	----	----	----	----

26<31, entonces insertar 31 en esta posición

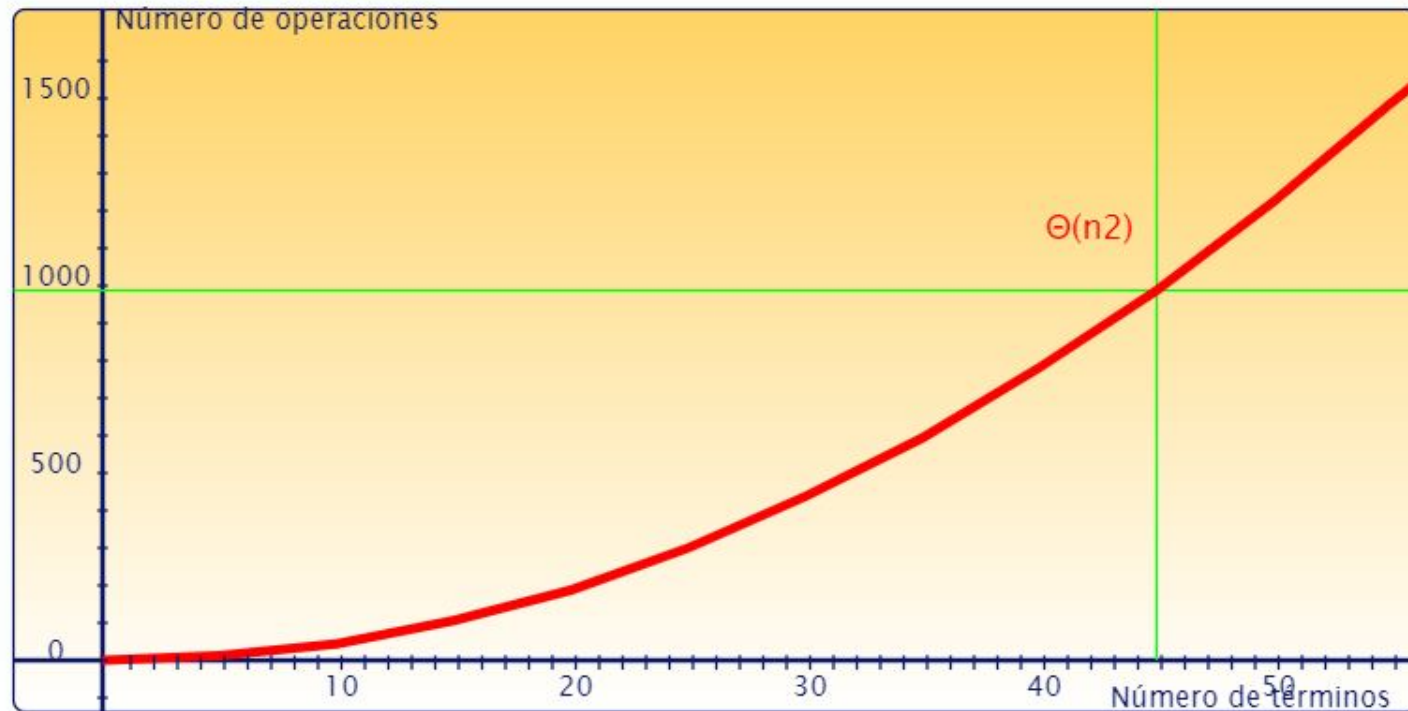
# Inserción

Rendimiento caso óptimo



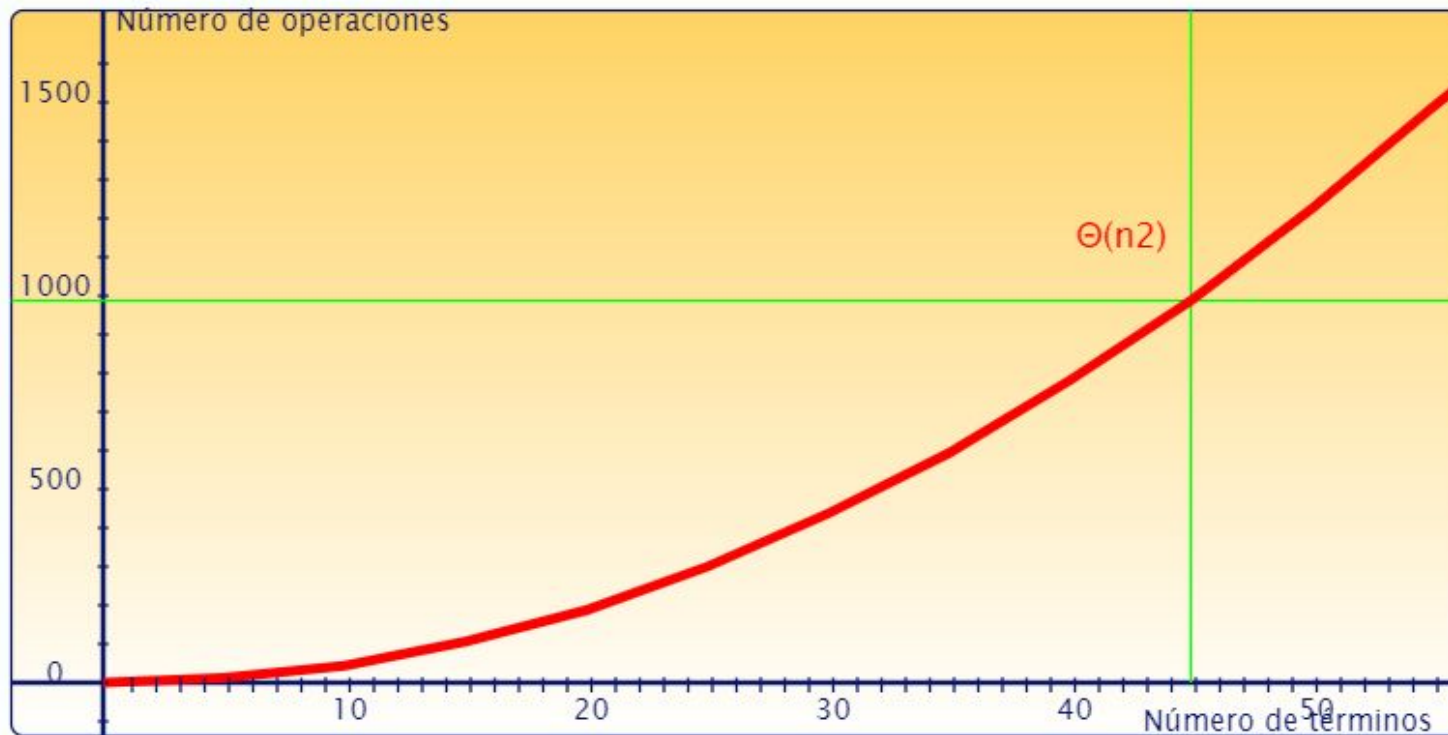
# Inserción

Rendimiento caso desfavorable



# Inserción

Rendimiento caso óptimo



---

# Selección

# Selección

## ANÁLISIS DEL ALGORITMO.

- Ø Requerimientos de Memoria: Al igual que el ordenamiento burbuja, este algoritmo sólo necesita una variable adicional para realizar los intercambios.
- Ø Tiempo de Ejecución: El ciclo externo se ejecuta  $n$  veces para una lista de  $n$  elementos. Cada búsqueda requiere comparar todos los elementos no clasificados.

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7



# Selección

Ventajas:

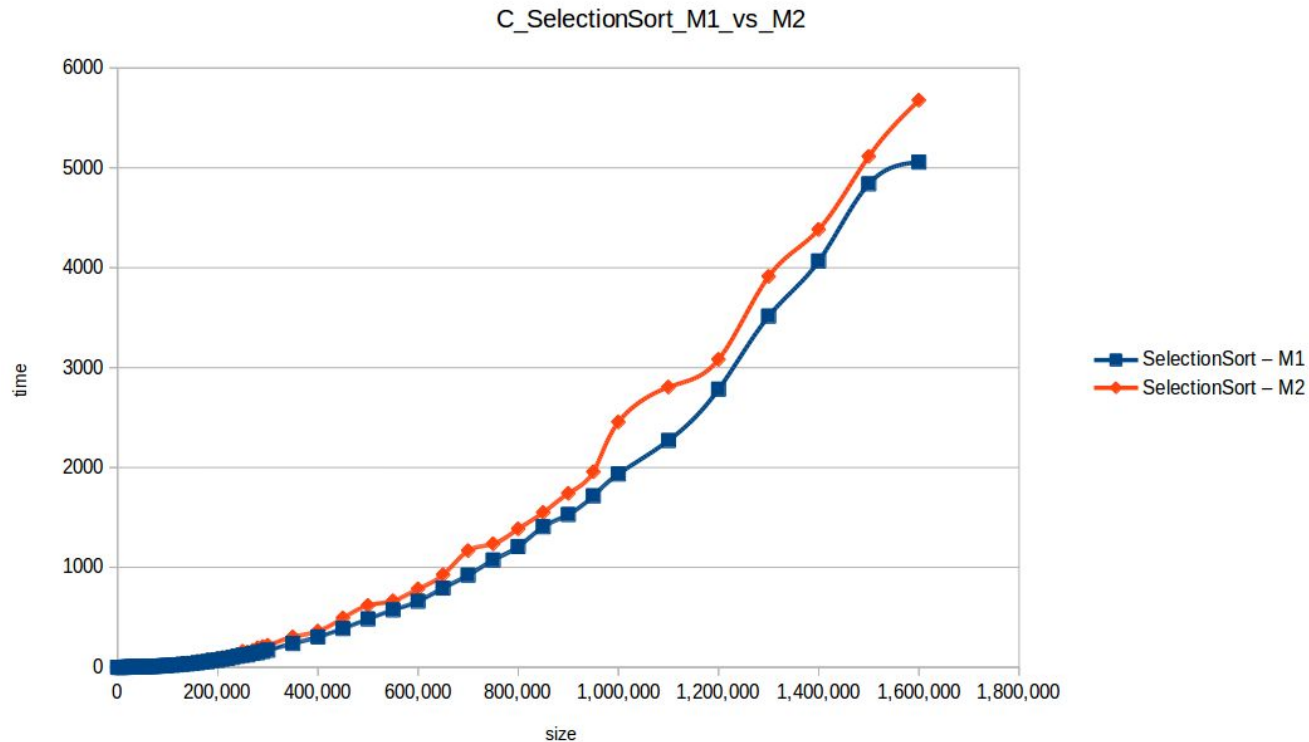
1. Fácil implementación.
2. No requiere memoria adicional.
3. Rendimiento constante: poca diferencia entre el peor y el mejor caso.

Desventajas:

1. Lento.
2. Realiza numerosas comparaciones.

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

# Selección



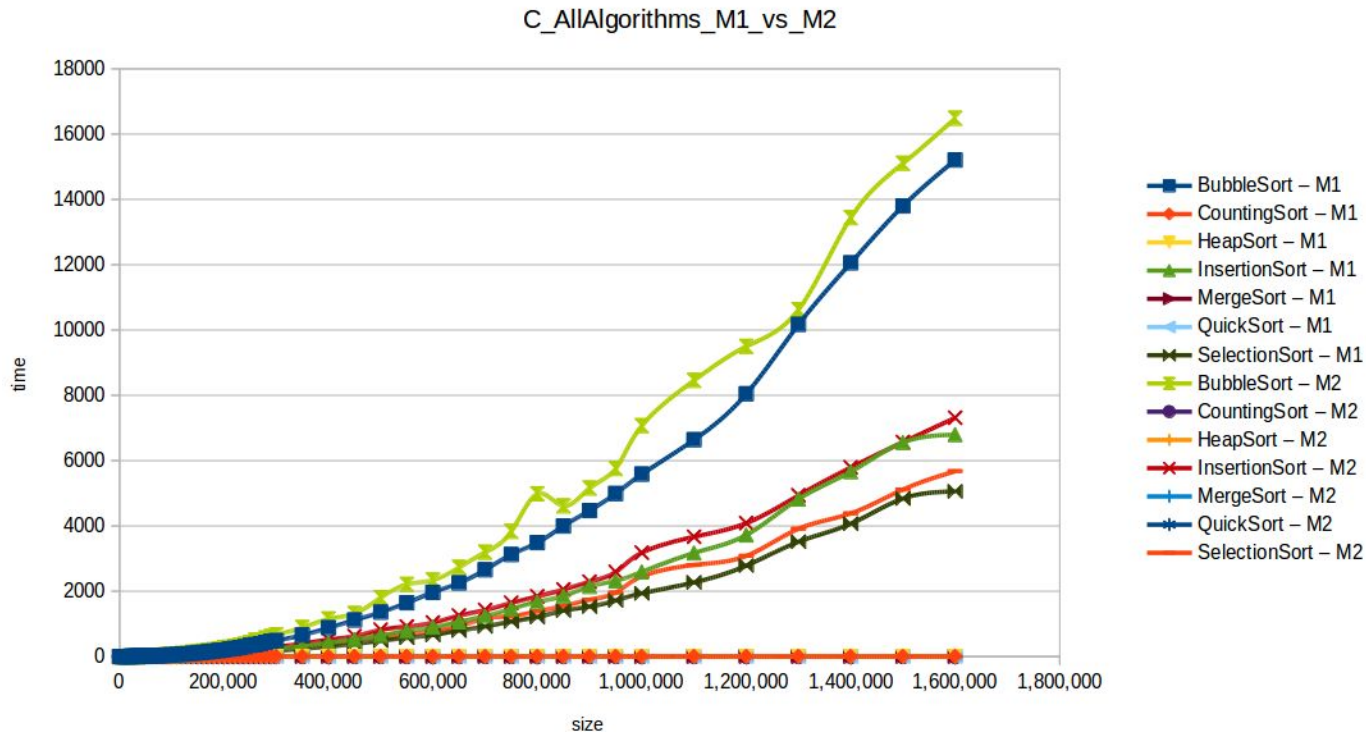
	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

# Selección

```
void intercambiar(int* a, int* b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
void ordenamientoSeleccion(int arreglo[], int longitud) {  
    for (int i = 0; i < longitud - 1; i++) {  
        int indiceMinimo = i;  
        for (int j = i + 1; j < longitud; j++) {  
            if (arreglo[j] < arreglo[indiceMinimo]) {  
                indiceMinimo = j;  
            }  
        }  
        if (indiceMinimo != i) {  
            intercambiar(&arreglo[i], &arreglo[indiceMinimo]);  
        }  
    }  
}
```

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

# Comparativa





**¿Preguntas?**

---

# **P vs NP**

## **o cómo ganar un millón de USD**



La cuestión, central en el campo de la teoría de la complejidad computacional, es uno de los siete problemas del milenio seleccionados por el Instituto Clay y su resolución está premiada con un millón de dólares

---

Daniel Graça, profesor de la Universidad del Algarve

# ¿es $P = NP$ completo?

- ¿Es posible "**verificar**" rápidamente las soluciones de un problema? (es decir, es un problema de tipo NP)
- ¿eso implica que también es posible "obtener" las respuestas con la misma rapidez? (es decir, es un problema de tipo P)

donde "rápidamente" significa "en **tiempo polinómico**".

El **tiempo**: mediante una aproximación al número de pasos de ejecución que un **algoritmo** emplea para resolver un problema.

El **espacio**: mediante una aproximación a la cantidad de memoria utilizada para resolver el problema.



# Problemas NP

A Venn diagram illustrating the relationship between complexity classes. A large light blue rounded rectangle is labeled 'Problemas NP'. Inside this rectangle, there are two smaller ovals. The first oval, on the left, is light green and labeled 'Problemas P'. The second oval, on the right and overlapping the first, is light pink and labeled 'Problemas NP-completos'. This visualizes that P is a subset of NP, and NP-complete problems are a subset of NP.

Problemas P

Problemas  
NP-completos



**¿Preguntas?**

# Problemas NP



A Venn diagram illustrating the relationship between complexity classes. A large light blue rounded rectangle is labeled 'Problemas NP'. Inside this rectangle, there are two smaller ovals. The first oval, on the left, is light green and labeled 'Problemas P'. The second oval, on the right, is light pink and labeled 'Problemas NP-completos'. The 'Problemas P' oval is entirely contained within the 'Problemas NP' rectangle, and the 'Problemas NP-completos' oval is also entirely contained within the 'Problemas NP' rectangle, but it does not overlap with the 'Problemas P' oval.

Problemas P

Problemas  
NP-completos

---

P

**unrn.edu.ar**

**UNRN**

Universidad Nacional  
de **Río Negro**



| **unrionegro**

---

# P vs NP

---

# Adivinar contraseñas

---

# Clases de complejidad



# Alcanzabilidad en grafos

# 2-SAT

**asignación de verdad satisfactoria  
en dos variables**

---

# NP

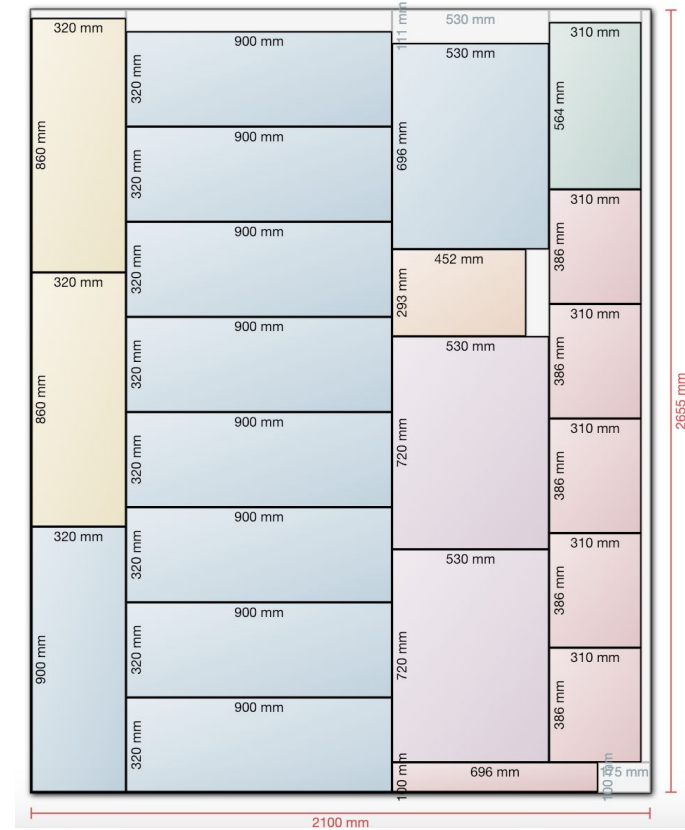
---

# NP-Hard

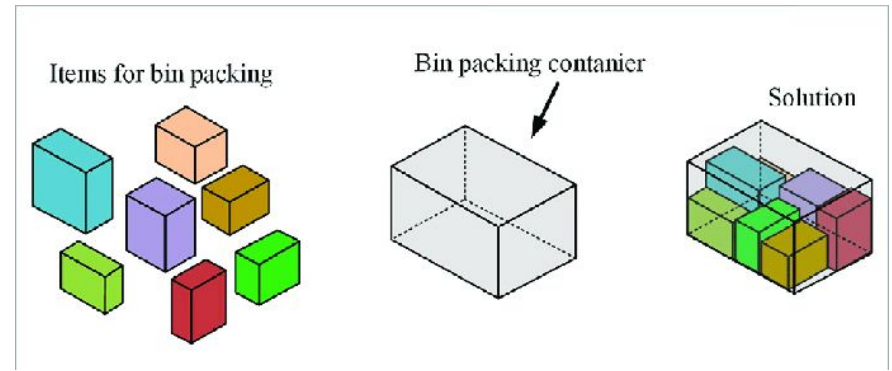
# **Empaquetado**

**[https://en.wikipedia.org/wiki/Bin\\_packing\\_problem](https://en.wikipedia.org/wiki/Bin_packing_problem)**

# Bidimensional



¿Como reducimos el desperdicio de placa cortada?



# Tridimensional

**¿Como reducimos el espacio sin utilizar?**

---

# Algoritmos de ordenamiento



---

# Algoritmos de búsqueda

---

# Sobre la compilación

---

# Optimizaciones del compilador

---

# Estrategias de ejecución

---

# Profiling



**¿Preguntas?**



**Abran  
hilo**