

Programación II

Arreglos

Tomás Capretto

FCEyE · Universidad Nacional de Rosario

Introducción

¿Cuál es mejor?

Queremos imprimir todos los números pares entre 2 y 100...

¿Cuál es mejor?

Queremos imprimir todos los números pares entre 2 y 100...

```
numero = 2
while numero ≤ 100:
    if numero % 2 == 0:
        print(numero)
    numero += 1
```

```
numero = 2
while numero ≤ 100:
    print(numero)
    numero += 2
```

¿Cuál es mejor?

Queremos imprimir todos los números pares entre 2 y 100...

```
numero = 2
while numero ≤ 100:
    if numero % 2 == 0:
        print(numero)
    numero += 1
```

```
numero = 2
while numero ≤ 100:
    print(numero)
    numero += 2
```

¿Qué hace que un algoritmo sea mejor que otro?

¿Cuál es mejor?

Queremos imprimir todos los números pares entre 2 y 100...

```
numero = 2
while numero ≤ 100:
    if numero % 2 == 0:
        print(numero)
    numero += 1
```

99 iteraciones

```
numero = 2
while numero ≤ 100:
    print(numero)
    numero += 2
```

50 iteraciones

¿Qué hace que un algoritmo sea mejor que otro?

Algunos criterios...

- Mantenibilidad
- Legibilidad
- Organización
- Eficiencia
 - Velocidad
 - Memoria

Estructuras de datos

Data, data, data

```
x = "¡Hola! "  
y = "¿Cómo estás? "  
z = "¿Todo bien?"  
  
print(x + y + z)
```

Data, data, data

```
x = "¡Hola! "  
y = "¿Cómo estás? "  
z = "¿Todo bien?"  
  
print(x + y + z)
```

```
arreglo = [  
    "¡Hola! ", "¿Cómo estás? ", "¿Todo bien?"  
]  
  
print(arreglo[0] + arreglo[1] + arreglo[2])
```

Data, data, data

```
x = "¡Hola! "  
y = "¿Cómo estás? "  
z = "¿Todo bien?"  
  
print(x + y + z)
```

```
arreglo = [  
    "¡Hola! ", "¿Cómo estás? ", "¿Todo bien?"  
]  
  
print(arreglo[0] + arreglo[1] + arreglo[2])
```

¿Cuál es mejor?

Data, data, data

```
x = "¡Hola! "  
y = "¿Cómo estás? "  
z = "¿Todo bien?"  
  
print(x + y + z)
```

```
arreglo = [  
    "¡Hola! ", "¿Cómo estás? ", "¿Todo bien?"  
]  
  
print(arreglo[0] + arreglo[1] + arreglo[2])
```

¿Cuál es mejor?

Trade-offs...

No existe una única forma de organizar los datos de un programa.

No solo afecta cómo se ve, también puede afectar su velocidad.

Una estructura familiar: Arreglos

Colección de datos en un orden determinado.

Una estructura familiar: Arreglos

Colección de datos en un orden determinado.

En Python, listas:

```
array = ["nueces", "almendras", "pistachos", "maníes", "castañas"]
```

Una estructura familiar: Arreglos

Colección de datos en un orden determinado.

En Python, listas:

```
array = ["nueces", "almendras", "pistachos", "maníes", "castañas"]
```

- Tamaño (*size*): cuántos elementos contiene.
- Índice (*index*): identifica la posición de cada elemento en él.

Una estructura familiar: Arreglos

Colección de datos en un orden determinado.

En Python, listas:

```
array = ["nueces", "almendras", "pistachos", "maníes", "castañas"]
```

- Tamaño (*size*): cuántos elementos contiene.
- Índice (*index*): identifica la posición de cada elemento en él.

index →	0	1	2	3	4
	"nueces"	"almendras"	"pistachos"	"maníes"	"castañas"

¿Qué tan buena es una estructura?

Medimos su desempeño para realizar diferentes operaciones:

- **Lectura** (*read*): obtener el valor de una posición.
- **Búsqueda** (*search*): encontrar la posición de un valor.
- **Inserción** (*insert*): agregar un elemento.
- **Eliminación** (*delete*): quitar un elemento.

¿Qué tan buena es una estructura?

Medimos su desempeño para realizar diferentes operaciones:

- **Lectura** (*read*): obtener el valor de una posición.
- **Búsqueda** (*search*): encontrar la posición de un valor.
- **Inserción** (*insert*): agregar un elemento.
- **Eliminación** (*delete*): quitar un elemento.

Algo ya conocemos...

```
array[indice]           # Lectura
array.index(valor)       # Búsqueda
array.insert(indice, valor) # Inserción
array.pop(indice)        # Eliminación por índice
array.remove(valor)      # Eliminación por valor
```

¿Cómo se mide la velocidad de una operación?

- No se mide el tiempo con un reloj.
 - Puede variar de computadora a computadora.

¿Cómo se mide la velocidad de una operación?

- No se mide el tiempo con un reloj.
 - Puede variar de computadora a computadora.
- Se cuenta la **cantidad de pasos**.
 - Medimos costo independientemente de donde se ejecutan.

¿Cómo se mide la velocidad de una operación?

- No se mide el tiempo con un reloj.
 - Puede variar de computadora a computadora.
- Se cuenta la **cantidad de pasos**.
 - Medimos costo independientemente de donde se ejecutan.
- El conteo se hace en relación a la cantidad de elementos N .

¿Cómo se mide la velocidad de una operación?

- No se mide el tiempo con un reloj.
 - Puede variar de computadora a computadora.
- Se cuenta la **cantidad de pasos**.
 - Medimos costo independientemente de donde se ejecutan.
- El conteo se hace en relación a la cantidad de elementos N .

La pregunta

¿Cómo cambia la cantidad de pasos a realizar a medida que crece la cantidad de elementos N ?

Lectura

Acceder al valor almacenado en una posición específica del arreglo.

```
array[indice]
```

Lectura

Acceder al valor almacenado en una posición específica del arreglo.

```
array[indice]
```

Por ejemplo:

```
array = [30, 10, 15, 25, 85]  
array[4] # Devuelve 85
```

Lectura

Acceder al valor almacenado en una posición específica del arreglo.

```
array[indice]
```

Por ejemplo:

```
array = [30, 10, 15, 25, 85]  
array[4] # Devuelve 85
```

La memoria RAM permite acceder directamente a cualquier posición sin recorrer las anteriores.

¿Por qué esto es posible?

					"zzz"		
	25						
				124			
	999				"hola"		
543							

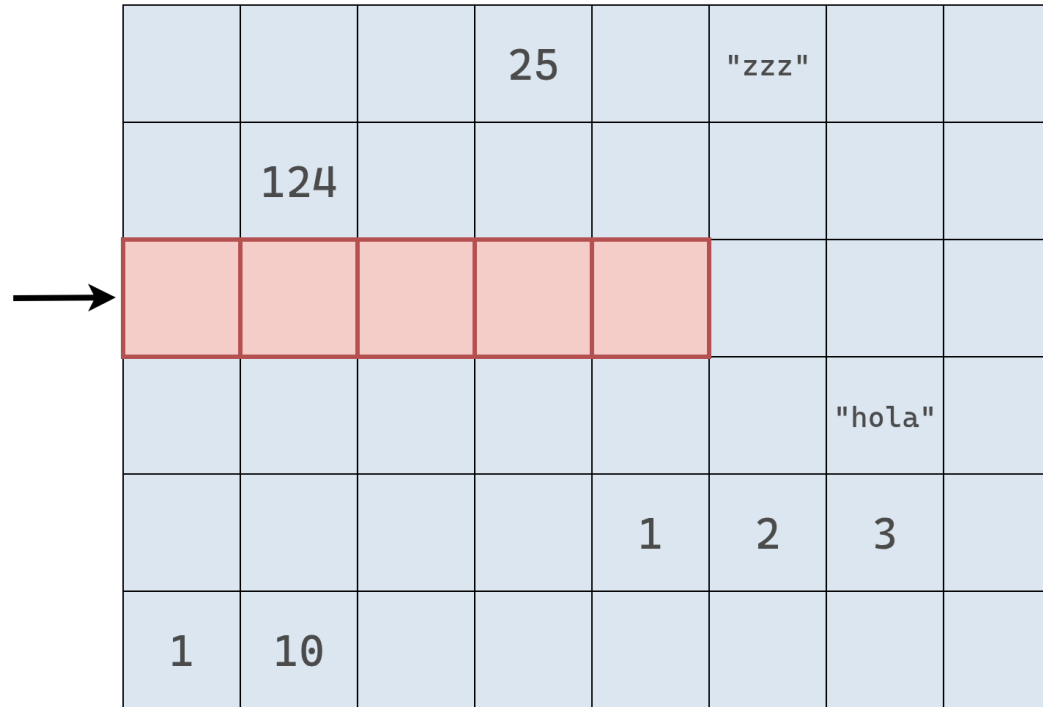
Podemos imaginar la memoria como una colección de celdas numeradas.

¿Por qué esto es posible?

1010	1011	1012	1013	1014	1015	1016	1017
1018	1019	1020	1021	1022	1023	1024	1025
1026	1027	1028	1029	1030	1031	1032	1033
1034	1035	1036	1037	1038	1039	1040	1041
1042	1043	1044	1045	1046	1047	1048	1049
1050	1051	1052	1053	1054	1055	1056	1057

Cada celda tiene una dirección única.

¿Por qué esto es posible?



A 6x8 grid representing memory cells. The third row from the top is highlighted in red, and a black arrow points to its first cell. The grid contains the following data:

			25		"zzz"		
	124						
→							
						"hola"	
				1	2	3	
1	10						

La computadora reserva un bloque contiguo de celdas para el arreglo.

¿Por qué esto es posible?

1010	1011	1012	1013	1014	1015	1016	1017
1018	1019	1020	1021	1022	1023	1024	1025
1026	1027	1028	1029	1030	1031	1032	1033
1034	1035	1036	1037	1038	1039	1040	1041
1042	1043	1044	1045	1046	1047	1048	1049
1050	1051	1052	1053	1054	1055	1056	1057

Cinco posiciones del arreglo ocupan cinco celdas consecutivas.

¿Por qué esto es posible?

En este modelo simplificado, la dirección se calcula con una operación aritmética simple:

```
dirección_base + i
```

¿Por qué esto es posible?

En este modelo simplificado, la dirección se calcula con una operación aritmética simple:

```
dirección_base + i
```

- Cada celda puede leerse o modificarse en el mismo tiempo.
- Ese tiempo es independiente de su posición.
- De ahí el nombre **memoria de acceso aleatorio**.

Lectura

index →	1010	1011	1012	1013	1014	← dirección de memoria
	0	1	2	3	4	
	"nueces"	"almendras"	"pistachos"	"maníes"	"castañas"	

El índice indica la posición del elemento y la dirección, su ubicación en memoria.

Lectura

Queremos leer el valor en el índice 3 de nuestro arreglo de frutos secos.

- El índice 0 está en la dirección 1010.

Lectura

Queremos leer el valor en el índice 3 de nuestro arreglo de frutos secos.

- El índice 0 está en la dirección 1010.
- El índice 3 está tres posiciones después.

Lectura

Queremos leer el valor en el índice 3 de nuestro arreglo de frutos secos.

- El índice 0 está en la dirección 1010.
- El índice 3 está tres posiciones después.
- Su dirección es 1013 ($1010 + 3$).

Lectura

Queremos leer el valor en el índice 3 de nuestro arreglo de frutos secos.

- El índice 0 está en la dirección 1010.
- El índice 3 está tres posiciones después.
- Su dirección es 1013 ($1010 + 3$).
- Allí podemos acceder al valor "maníes".

Lectura

Queremos leer el valor en el índice 3 de nuestro arreglo de frutos secos.

- El índice 0 está en la dirección 1010.
- El índice 3 está tres posiciones después.
- Su dirección es 1013 ($1010 + 3$).
- Allí podemos acceder al valor "maníes".

Leer desde un arreglo permite acceder a cualquier elemento en un solo paso.

El tamaño de cada elemento

En la memoria real, las direcciones se miden en bytes.

- Cada elemento puede ocupar varios bytes.
- Para calcular su dirección, tenemos en cuenta ese tamaño.

```
dirección_base + i * tamaño_del_elemento
```

El tamaño de cada elemento

En la memoria real, las direcciones se miden en bytes.

- Cada elemento puede ocupar varios bytes.
- Para calcular su dirección, tenemos en cuenta ese tamaño.

```
dirección_base + i * tamaño_del_elemento
```

Por ejemplo, con una dirección inicial de 1000 y elementos de 8 bytes:

- Las posiciones comienzan en 1000, 1008, 1016, 1024...
- El índice 3 está en $1000 + 3 \times 8 = 1024$.

Búsqueda

Comprobar si un valor existe y, de ser así, obtener su índice.

Búsqueda

Comprobar si un valor existe y, de ser así, obtener su índice.

- Leer: dar un índice y obtener un valor.
- Buscar: dar un valor y obtener su índice.

Búsqueda

Comprobar si un valor existe y, de ser así, obtener su índice.

- Leer: dar un índice y obtener un valor.
- Buscar: dar un valor y obtener su índice.

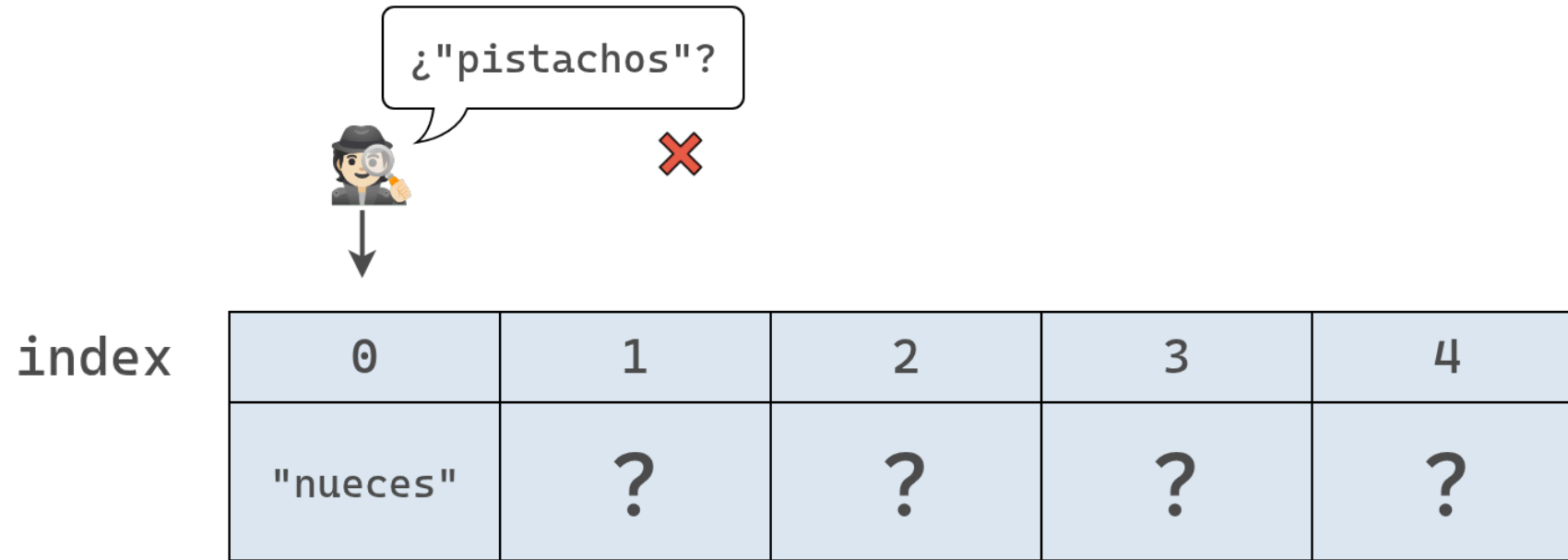
La computadora debe revisar las celdas una por una hasta encontrar el valor.

Búsqueda

index	0	1	2	3	4
	?	?	?	?	?

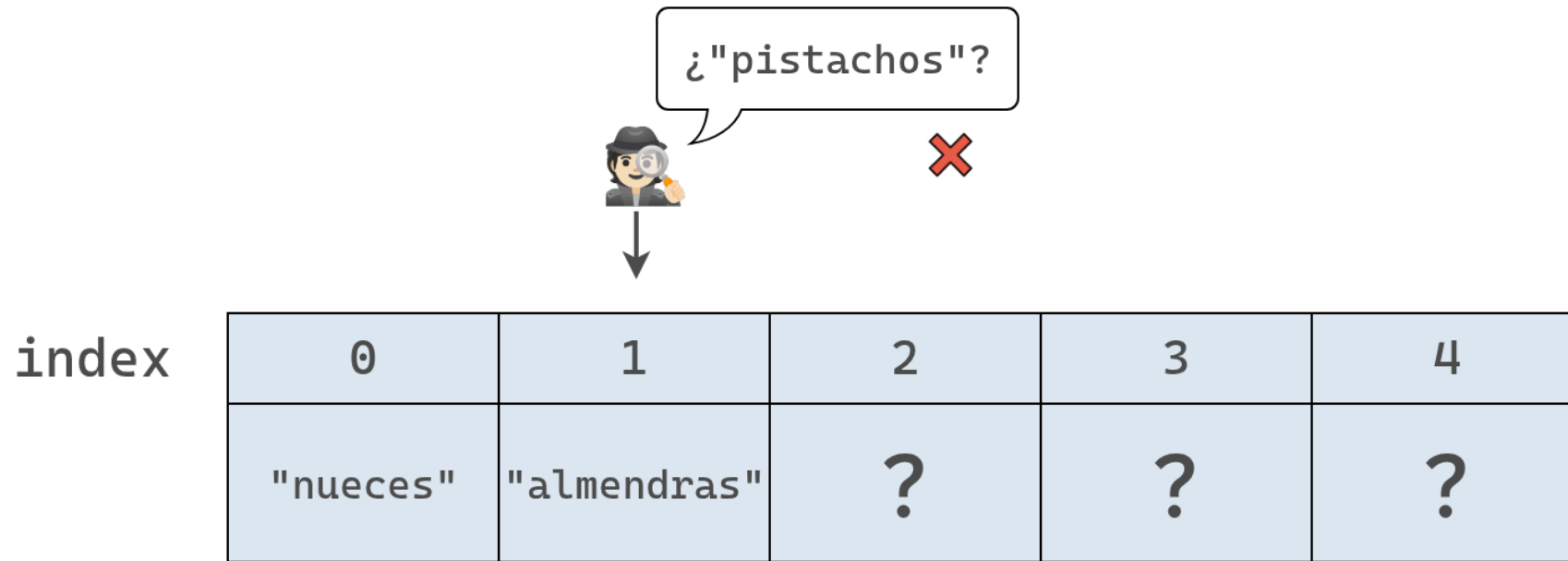
Buscamos "pistachos" en nuestro arreglo de frutos secos.

Búsqueda



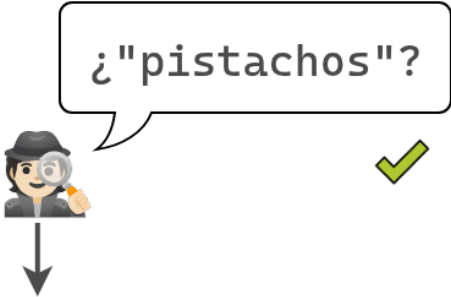
Primero revisamos el índice 0.

Búsqueda



Encontramos "nueces". Avanzamos al siguiente índice.

Búsqueda



index	0	1	2	3	4
	"nueces"	"almendras"	"pistachos"	?	?

Encontramos "pistachos" en el índice 2. La búsqueda tomó tres pasos.

Búsqueda

A esta forma de revisar las celdas una por una la llamamos **búsqueda lineal**.

- Si encontramos el valor, la búsqueda termina.

Búsqueda

A esta forma de revisar las celdas una por una la llamamos **búsqueda lineal**.

- Si encontramos el valor, la búsqueda termina.
- Si está al final o no aparece, recorremos todo el arreglo.

Búsqueda

A esta forma de revisar las celdas una por una la llamamos **búsqueda lineal**.

- Si encontramos el valor, la búsqueda termina.
- Si está al final o no aparece, recorremos todo el arreglo.
- Con 5 elementos, podemos necesitar hasta 5 pasos.

Búsqueda

A esta forma de revisar las celdas una por una la llamamos **búsqueda lineal**.

- Si encontramos el valor, la búsqueda termina.
- Si está al final o no aparece, recorremos todo el arreglo.
- Con 5 elementos, podemos necesitar hasta 5 pasos.
- Con N elementos, podemos necesitar hasta N pasos.

Inserción

La eficiencia depende del lugar donde se realice la inserción.

Inserción

La eficiencia depende del lugar donde se realice la inserción.

Queremos agregar "pasas" al final de nuestra lista de frutos secos.

- Conocemos la dirección inicial y el tamaño del arreglo.
- Si comienza en 1010 y tiene 5 elementos, el último está en 1014.
- El nuevo elemento ocupa la posición 1015, si hay capacidad disponible.

Inserción

index →	1010	1011	1012	1013	1014	1015	← dirección de memoria
	0	1	2	3	4	5	
	"nueces"	"almendras"	"pistachos"	"maníes"	"castañas"	"pasas"	

Una vez calculada la posición, la inserción al final requiere un paso.

Inserción

Si el bloque reservado solo tiene cinco celdas, agregar un sexto elemento requiere ampliar su capacidad.

Inserción

Si el bloque reservado solo tiene cinco celdas, agregar un sexto elemento requiere ampliar su capacidad.

Insertar al inicio o en el medio requiere desplazar elementos para dejar espacio.

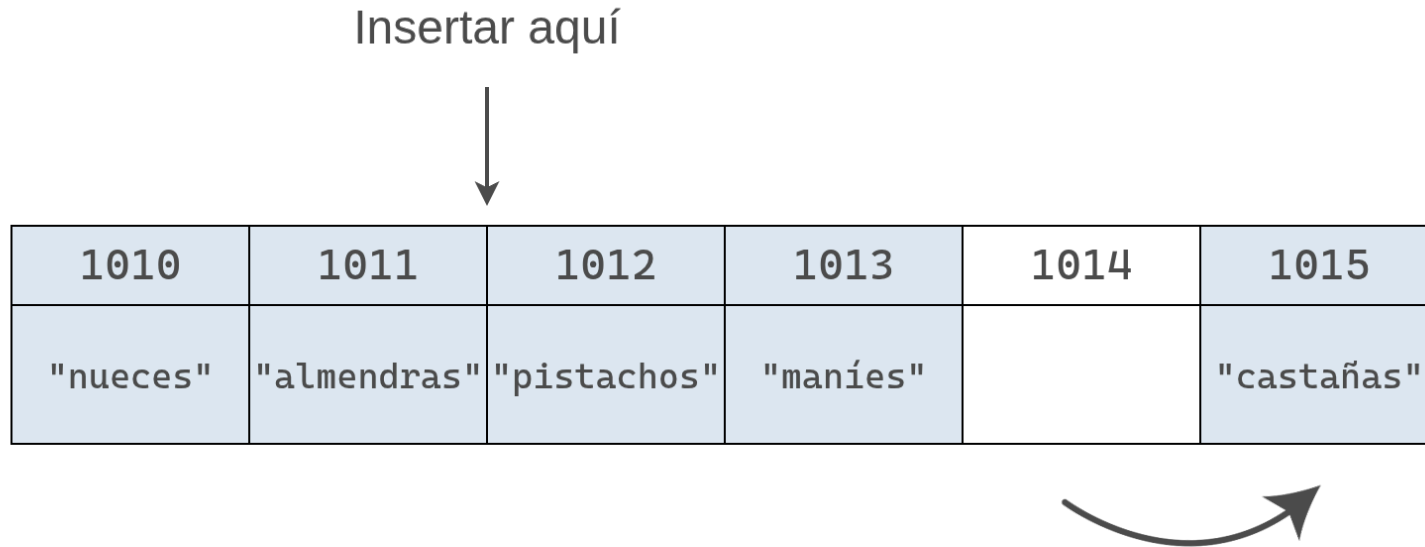
Por ejemplo, queremos insertar "pasas" en el índice 2.

Inserción

Insertar aquí			Siguiente celda libre		
↓			↓		
1010	1011	1012	1013	1014	1015
"nueces"	"almendras"	"pistachos"	"maníes"	"castañas"	

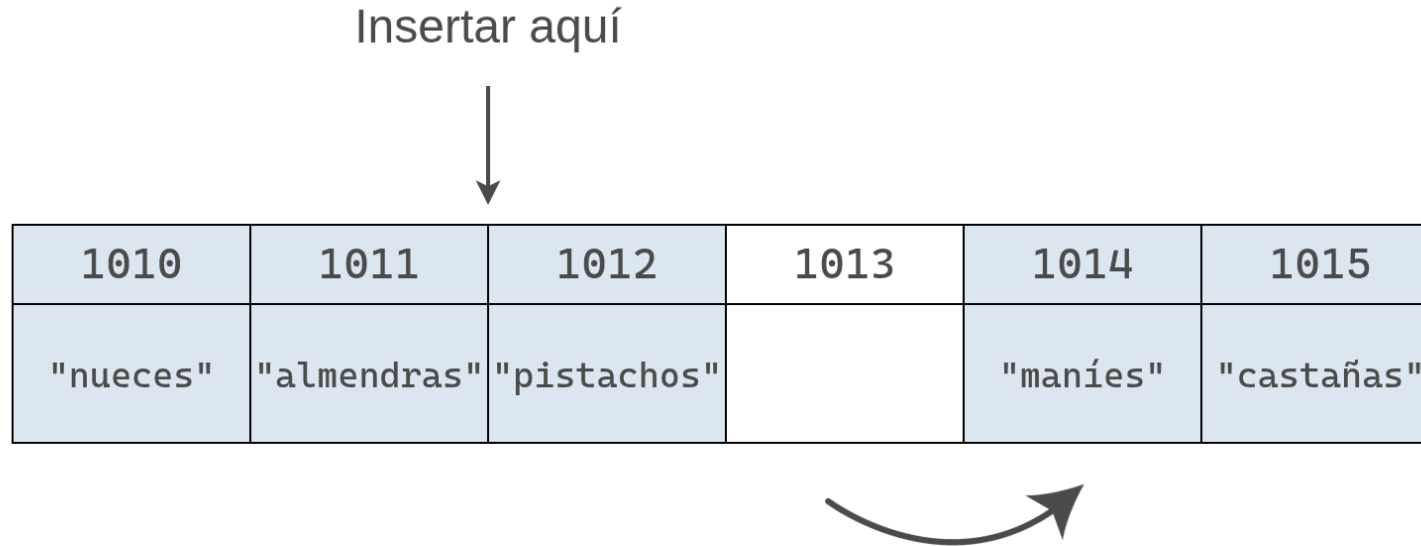
Debemos mover "pistachos", "maníes" y "castañas" una posición a la derecha.

Inserción



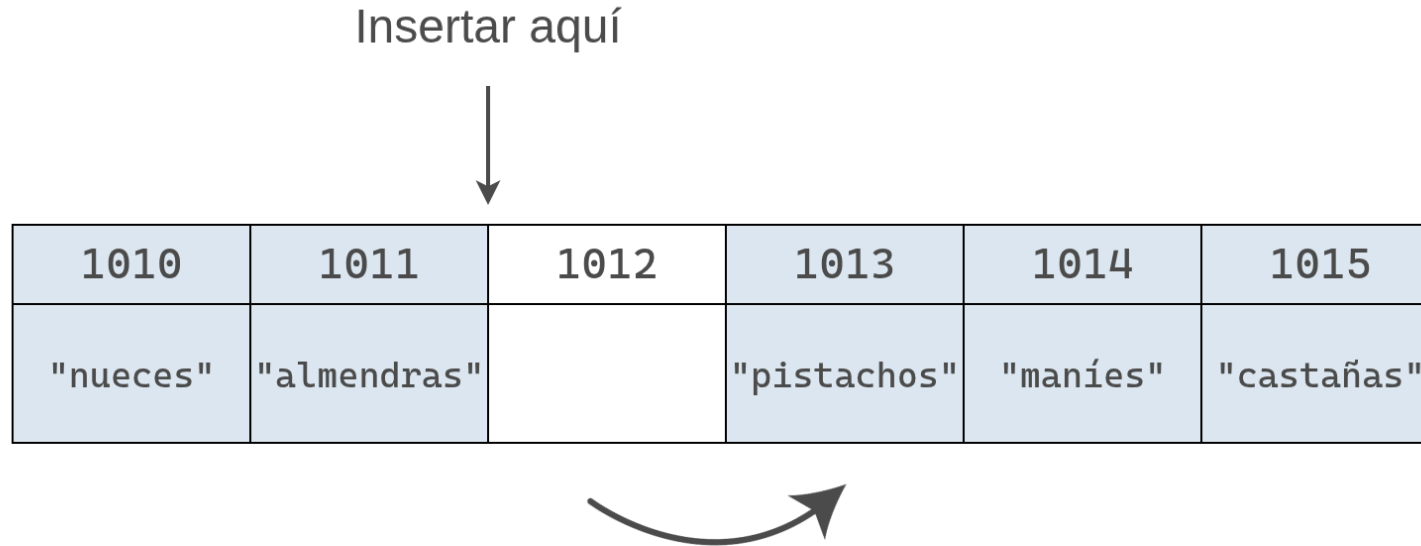
Primero movemos "castañas" una celda a la derecha.

Inserción



Luego movemos "maníes" a la derecha.

Inserción



Por último, movemos "pistachos" y liberamos el índice 2.

Inserción

1010	1011	1012	1013	1014	1015
"nueces"	"almendras"	"pasas"	"pistachos"	"maníes"	"castañas"



Insertamos "pasas" en el espacio disponible.

Inserción

index →	1010	1011	1012	1013	1014	1015	← dirección de memoria
	0	1	2	3	4	5	
	"nueces"	"almendras"	"pasas"	"pistachos"	"maníes"	"castañas"	

El arreglo tiene longitud 6 y "pasas" ocupa el índice 2.

Inserción

En este ejemplo usamos cuatro pasos:

- Tres para mover datos.
- Uno para agregar el nuevo valor.

Inserción

En este ejemplo usamos cuatro pasos:

- Tres para mover datos.
- Uno para agregar el nuevo valor.

El peor caso

El peor caso ocurre al insertar al inicio: desplazamos los N elementos y luego insertamos.

En total, $N + 1$ pasos.

Eliminación

Quitar el valor ubicado en un índice determinado.

Eliminación

Quitar el valor ubicado en un índice determinado.

Eliminemos el índice 1, que corresponde a "almendras".

- La eliminación deja una celda vacía.
- Debemos mover los elementos posteriores una posición a la izquierda.

Eliminación

Eliminar



1010	1011	1012	1013	1014
"nueces"	"almendras"	"pistachos"	"maníes"	"castañas"

Eliminamos el valor del índice 1.

Eliminación

1010	1011	1012	1013	1014
"nueces"		"pistachos"	"maníes"	"castañas"



Queda una celda vacía en el medio del arreglo.

Eliminación

1010	1011	1012	1013	1014
"nueces"	"pistachos"		"maníes"	"castañas"



Movemos "pistachos" una posición a la izquierda.

Eliminación

1010	1011	1012	1013	1014
"nueces"	"pistachos"	"maníes"		"castañas"



Continuamos moviendo los elementos posteriores.

Eliminación

1010	1011	1012	1013	1014
"nueces"	"pistachos"	"maníes"	"castañas"	



1010	1011	1012	1013
"nueces"	"pistachos"	"maníes"	"castañas"

El arreglo pasa de cinco elementos a cuatro.

Eliminación

En este ejemplo usamos cuatro pasos:

- Uno para borrar el valor.
- Tres para desplazar los datos.

Eliminación

En este ejemplo usamos cuatro pasos:

- Uno para borrar el valor.
- Tres para desplazar los datos.

El peor caso

El peor caso ocurre al eliminar el primer elemento: borramos un valor y desplazamos los $N - 1$ restantes.

En total, hasta N pasos.

Conjuntos

Un conjunto puede implementarse de diferentes maneras.

Conjuntos

Un conjunto puede implementarse de diferentes maneras.

Consideremos un **conjunto basado en arreglos**: un arreglo que no permite valores duplicados.

- Lectura, búsqueda y eliminación funcionan como en los arreglos.
- La diferencia aparece en la inserción.

Conjuntos

Antes de agregar un valor, verificamos que no exista en el conjunto.

Conjuntos

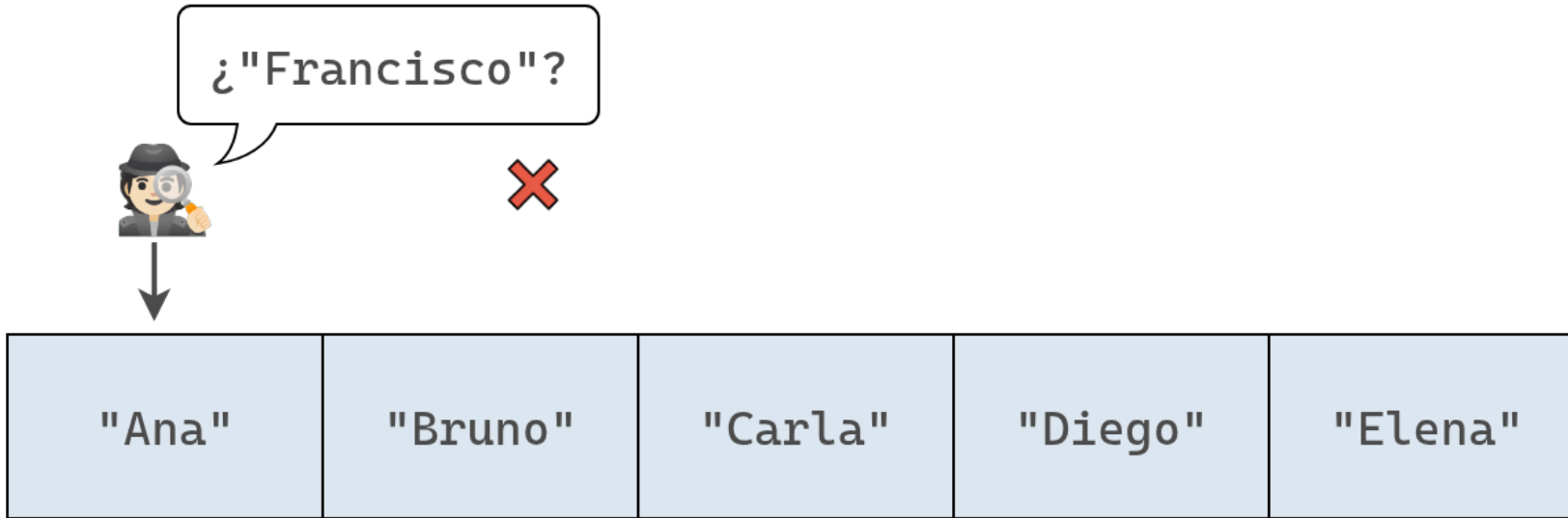
Antes de agregar un valor, verificamos que no exista en el conjunto.

```
["Ana", "Bruno", "Carla", "Diego", "Elena"]
```

Queremos agregar a "Francisco".

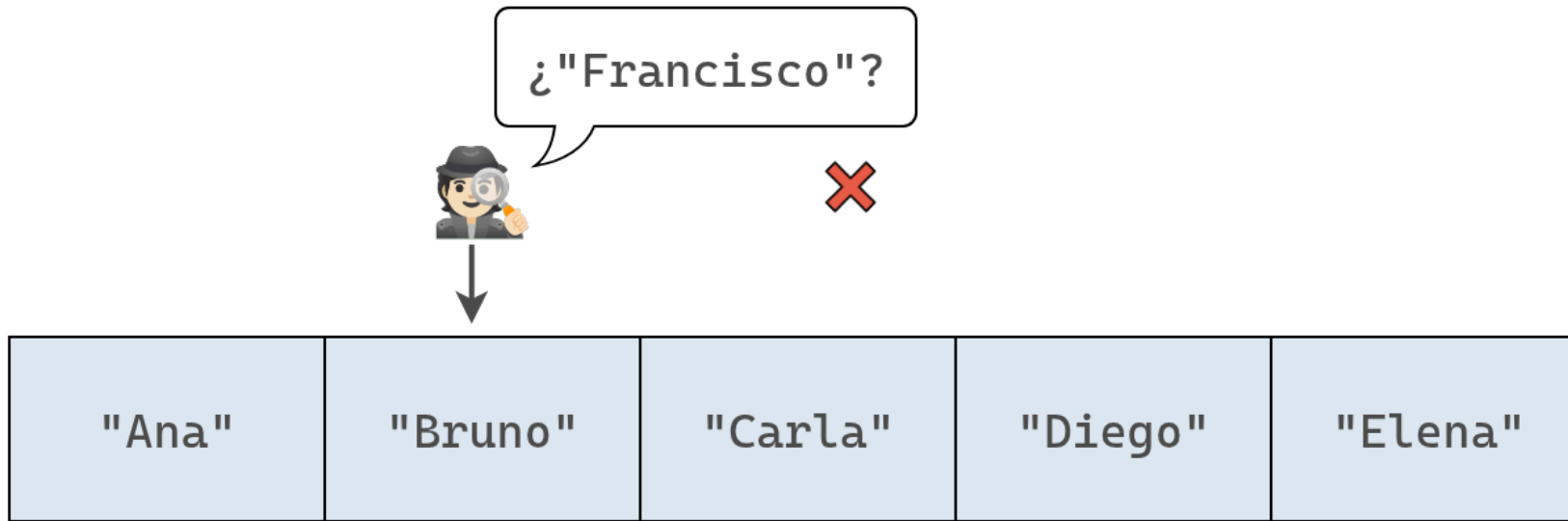
Primero tenemos que comprobar que no esté ya inscripto.

Conjuntos



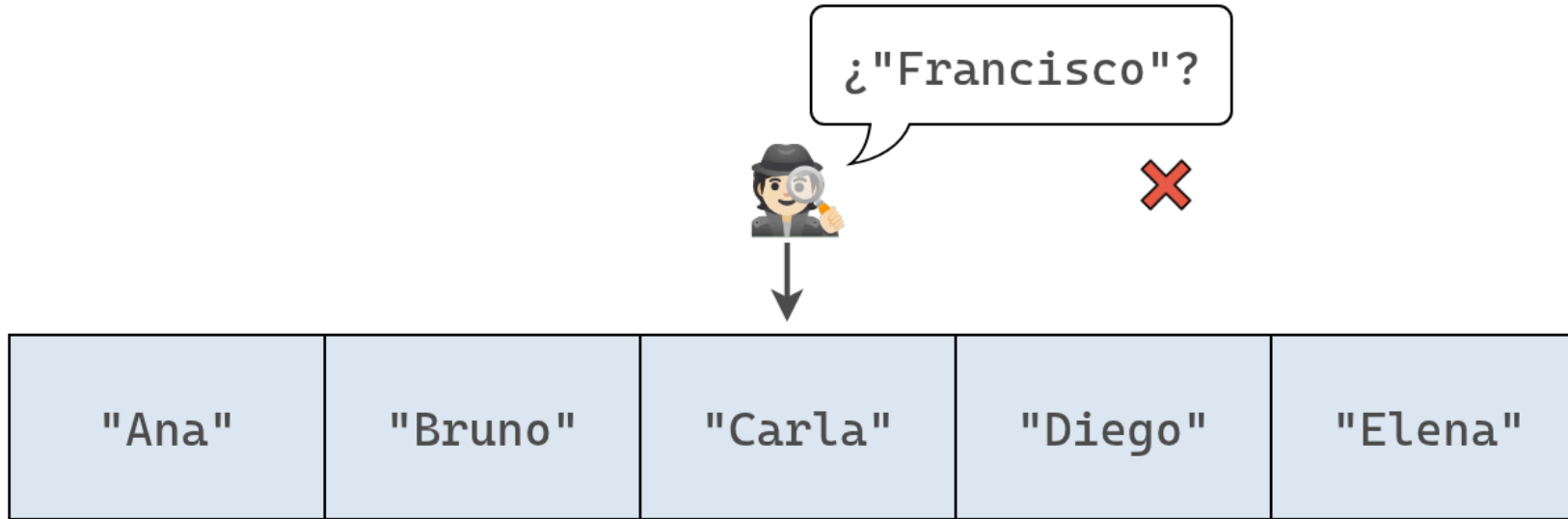
Inspeccionamos los elementos uno a uno.

Conjuntos



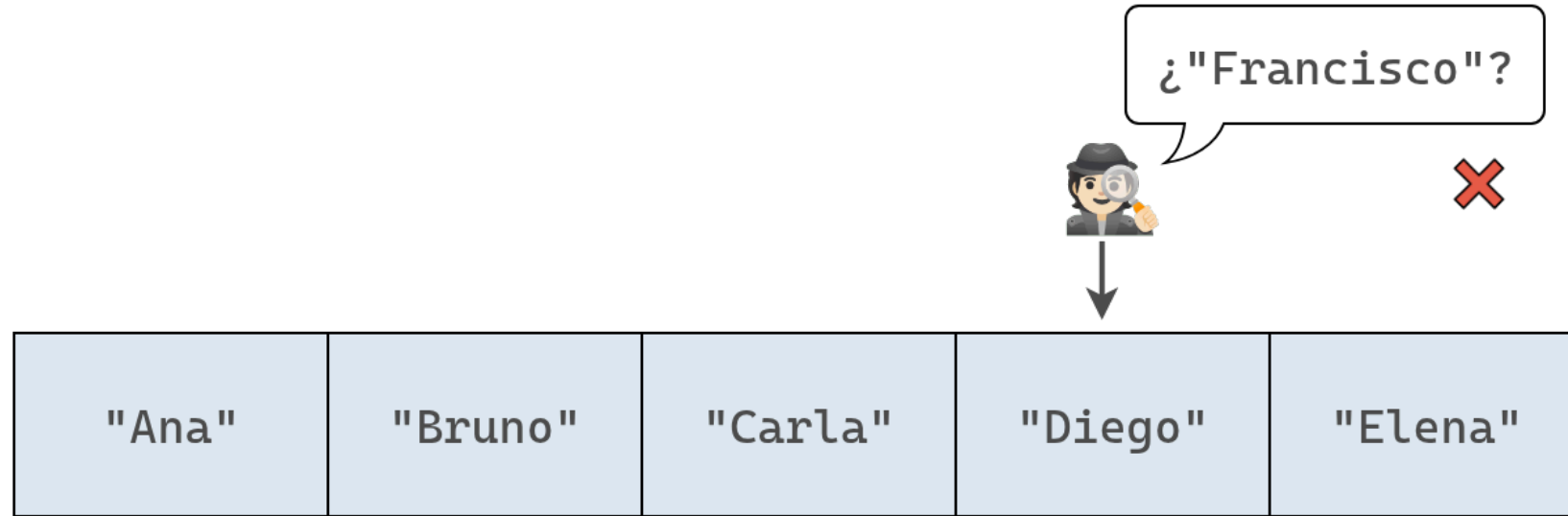
Buscamos una coincidencia con "Francisco".

Conjuntos



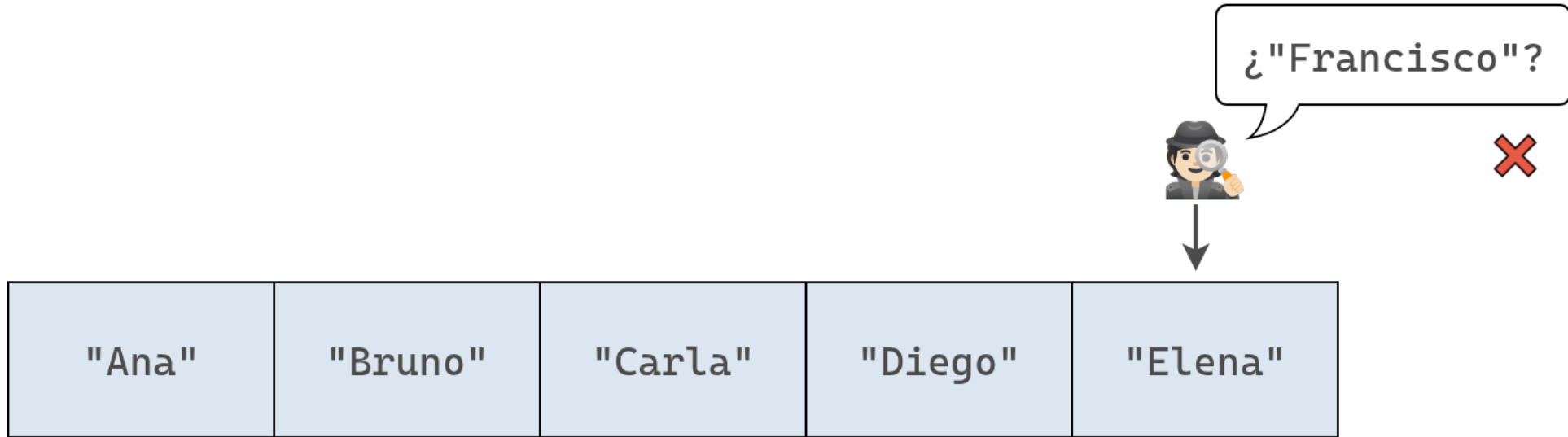
Si no encontramos el valor, seguimos avanzando.

Conjuntos



La búsqueda continúa por el arreglo.

Conjuntos



Revisamos todo el conjunto y sabemos que "Francisco" está ausente.

Conjuntos

"Ana"	"Bruno"	"Carla"	"Diego"	"Elena"	"Francisco"
-------	---------	---------	---------	---------	-------------

Ahora podemos insertar el nuevo valor.

Conjuntos

- Insertar al final requiere N pasos para buscar y uno para insertar.
- Insertar al inicio también requiere desplazar los N elementos.

Conjuntos

- Insertar al final requiere N pasos para buscar y uno para insertar.
- Insertar al inicio también requiere desplazar los N elementos.

Contando pasos

Al final: $N + 1$ pasos.

Al inicio: $2N + 1$ pasos.

Algoritmos

Algoritmos

Un conjunto de pasos definidos para realizar una tarea.

Por ejemplo, preparar un mate:

1. Calentar el agua.

Algoritmos

Un conjunto de pasos definidos para realizar una tarea.

Por ejemplo, preparar un mate:

1. Calentar el agua.
2. Colocar la yerba en el mate.

Algoritmos

Un conjunto de pasos definidos para realizar una tarea.

Por ejemplo, preparar un mate:

1. Calentar el agua.
2. Colocar la yerba en el mate.
3. Insertar la bombilla.

Algoritmos

Un conjunto de pasos definidos para realizar una tarea.

Por ejemplo, preparar un mate:

1. Calentar el agua.
2. Colocar la yerba en el mate.
3. Insertar la bombilla.
4. Servir el agua con cuidado.

Algoritmos

- La estructura de datos afecta el rendimiento del código.

Algoritmos

- La estructura de datos afecta el rendimiento del código.
- El algoritmo que usamos también influye en su eficiencia.

Algoritmos

- La estructura de datos afecta el rendimiento del código.
- El algoritmo que usamos también influye en su eficiencia.
- Distintos algoritmos pueden resolver la misma tarea con eficiencias distintas.

Algoritmos

- La estructura de datos afecta el rendimiento del código.
- El algoritmo que usamos también influye en su eficiencia.
- Distintos algoritmos pueden resolver la misma tarea con eficiencias distintas.

Ya vimos un ejemplo al imprimir números pares.

Ahora vamos a trabajar con arreglos ordenados.

Arreglos ordenados

Los valores deben mantenerse siempre en orden.

Arreglos ordenados

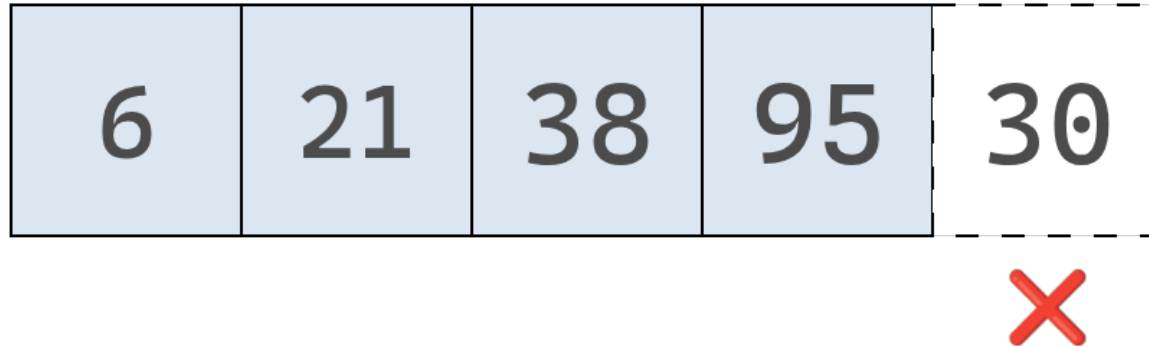
Los valores deben mantenerse siempre en orden.

```
[6, 21, 38, 95]
```

Queremos insertar el valor 30.

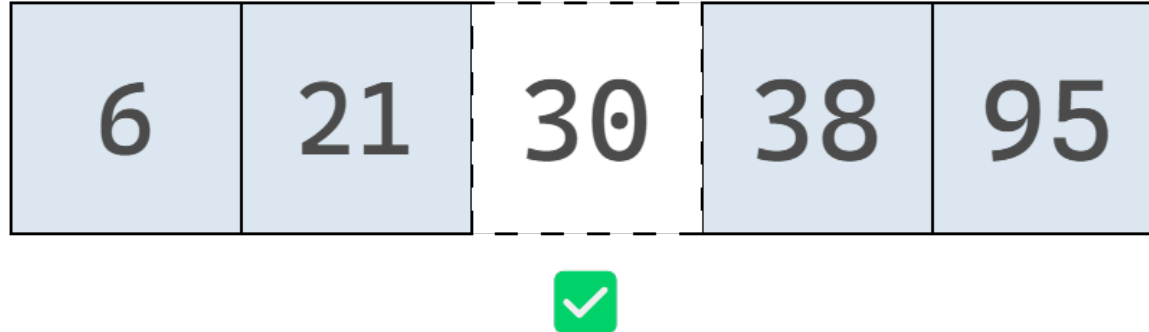
- En un arreglo común podríamos agregarlo al final.
- En uno ordenado debemos ubicarlo en la posición correcta.

Arreglos ordenados



Agregar el 30 al final rompe el orden.

Arreglos ordenados



El 30 debe quedar entre el 21 y el 38.

Arreglos ordenados

La computadora debe:

1. Encontrar dónde insertar el nuevo valor.

Arreglos ordenados

La computadora debe:

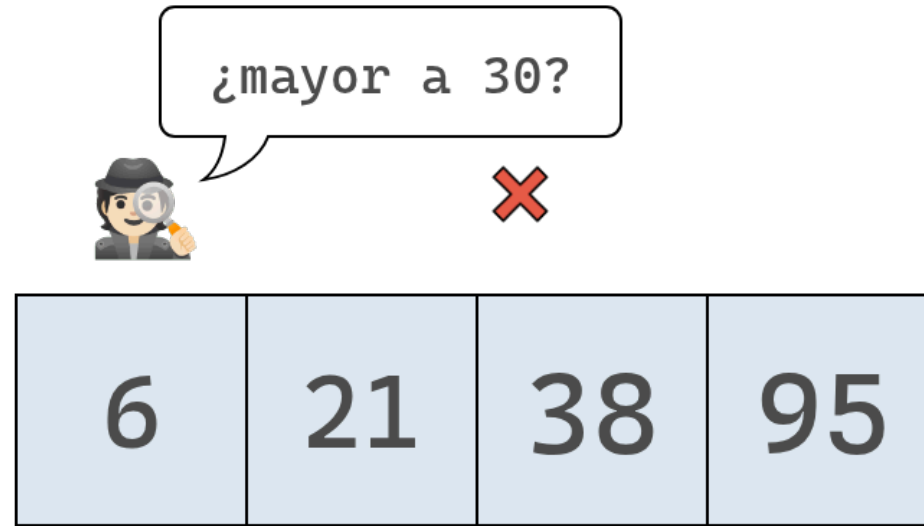
1. Encontrar dónde insertar el nuevo valor.
2. Desplazar los demás valores para crear espacio.

Arreglos ordenados

6	21	38	95
---	----	----	----

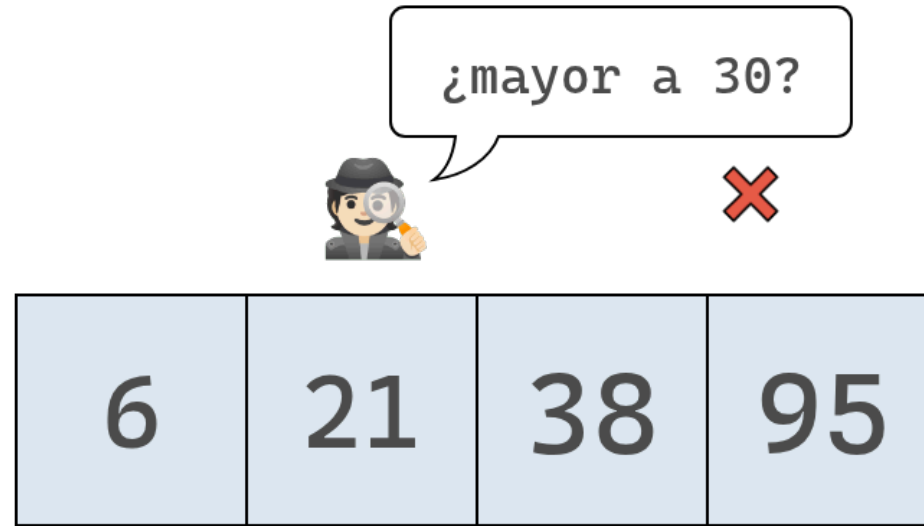
Partimos del arreglo ordenado original.

Arreglos ordenados



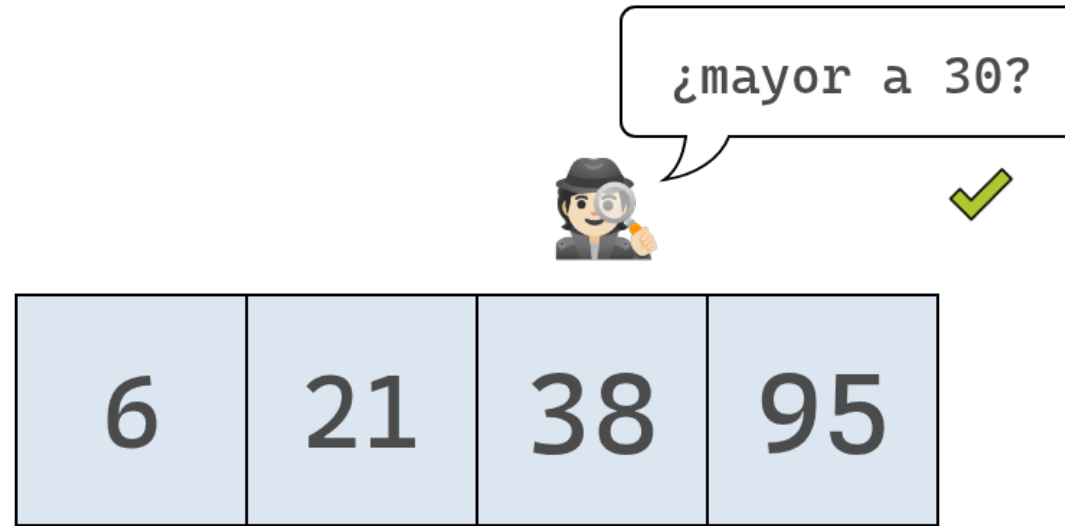
Comparamos el 30 con el valor del índice 0.

Arreglos ordenados



El 30 es mayor que el 6 y el 21. Seguimos avanzando.

Arreglos ordenados



El 38 es mayor que el 30: debemos insertar a su izquierda.

Arreglos ordenados

Insertar aquí



6	21	38	95
---	----	----	----

Necesitamos desplazar los datos y dejar espacio.

Arreglos ordenados



Movemos el 95 una posición a la derecha.

Arreglos ordenados



Queda espacio a la derecha del 38.

Arreglos ordenados



Movemos el 38 y liberamos la posición para el 30.

Arreglos ordenados

6	21	30	38	95
---	----	----	----	----

Insertamos el 30 en la posición correspondiente.

Arreglos ordenados

En nuestro ejemplo, la inserción tomó 6 pasos.

- Cerca del inicio: menos comparaciones y más desplazamientos.
- Cerca del final: más comparaciones y menos desplazamientos.

Arreglos ordenados

En nuestro ejemplo, la inserción tomó 6 pasos.

- Cerca del inicio: menos comparaciones y más desplazamientos.
- Cerca del final: más comparaciones y menos desplazamientos.

La inserción requiere buscar primero la posición correcta y luego crear el espacio para el nuevo valor.

Búsqueda en arreglos ordenados

Buscamos el valor 21.

En un arreglo clásico, puede estar en cualquier posición:

```
[67, 25, 13, 6, 132]
```

Búsqueda en arreglos ordenados

Buscamos el valor 21.

En un arreglo clásico, puede estar en cualquier posición:

```
[67, 25, 13, 6, 132]
```

En un arreglo ordenado podemos detenernos al llegar al 25:

```
[6, 13, 25, 67, 132]
```

Búsqueda en arreglos ordenados

```
def busqueda_lineal(array, valor):  
    for indice, elemento in enumerate(array):  
        if elemento == valor:  
            return indice  
        elif elemento > valor:  
            break  
    return None
```

Búsqueda en arreglos ordenados

```
def busqueda_lineal(array, valor):  
    for indice, elemento in enumerate(array):  
        if elemento == valor:  
            return indice  
        elif elemento > valor:  
            break  
    return None
```

```
print(busqueda_lineal([6, 13, 25, 67, 132], 21))  
# None
```

Búsqueda en arreglos ordenados

- La búsqueda termina cuando el elemento actual es mayor que el buscado.
- En ciertos casos, revisamos menos posiciones que en un arreglo clásico.
- En el peor caso, podemos necesitar hasta N pasos.

Búsqueda en arreglos ordenados

- La búsqueda termina cuando el elemento actual es mayor que el buscado.
- En ciertos casos, revisamos menos posiciones que en un arreglo clásico.
- En el peor caso, podemos necesitar hasta N pasos.

La gran ventaja del orden es que permite aplicar otro algoritmo: la **búsqueda binaria**.

Búsqueda binaria

Alguien piensa un número entre 1 y 100 y vos tenés que adivinarlo.

Después de cada intento, te dicen si es mayor o menor.

- Probamos con la mitad: el 50.

Búsqueda binaria

Alguien piensa un número entre 1 y 100 y vos tenés que adivinarlo.

Después de cada intento, te dicen si es mayor o menor.

- Probamos con la mitad: el 50.
- Si es mayor, podemos probar con 75. Si es menor, con 25.

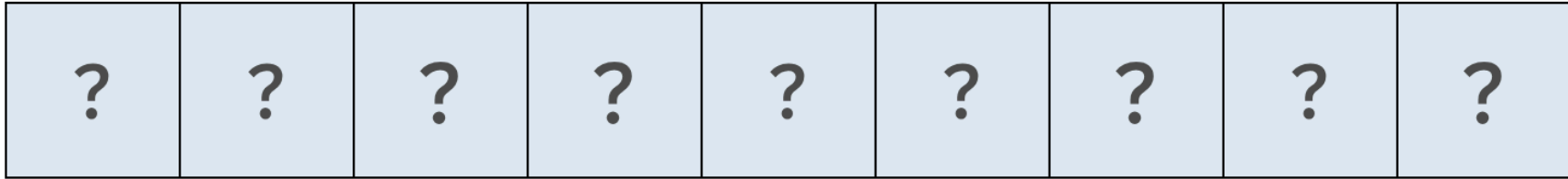
Búsqueda binaria

Alguien piensa un número entre 1 y 100 y vos tenés que adivinarlo.

Después de cada intento, te dicen si es mayor o menor.

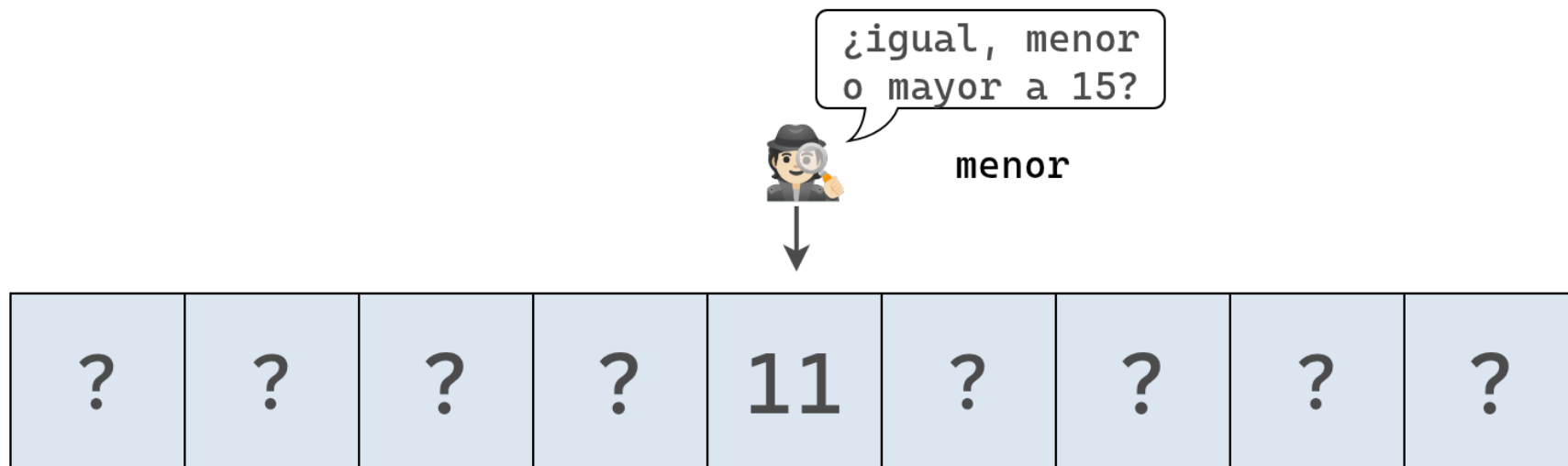
- Probamos con la mitad: el 50.
- Si es mayor, podemos probar con 75. Si es menor, con 25.
- En cada paso elegimos el punto medio y reducimos las posibilidades a la mitad.

Búsqueda binaria



Tenemos nueve elementos ordenados y buscamos el valor 15.

Búsqueda binaria



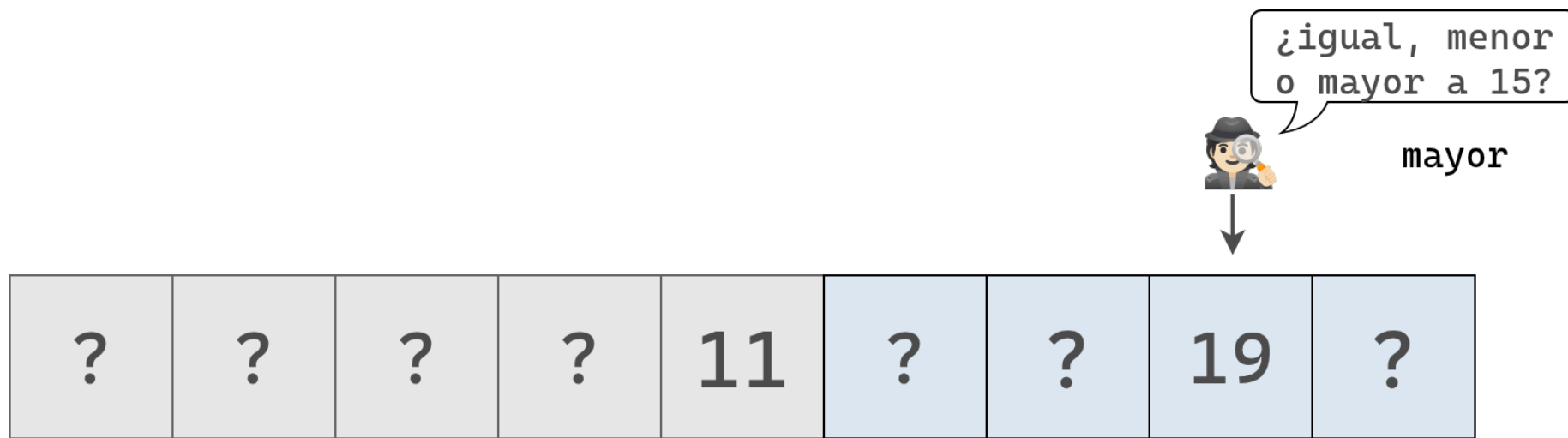
Inspeccionamos el centro y encontramos el 11.

Búsqueda binaria



El 15 está a su derecha. Descartamos la mitad izquierda, incluido el 11.

Búsqueda binaria



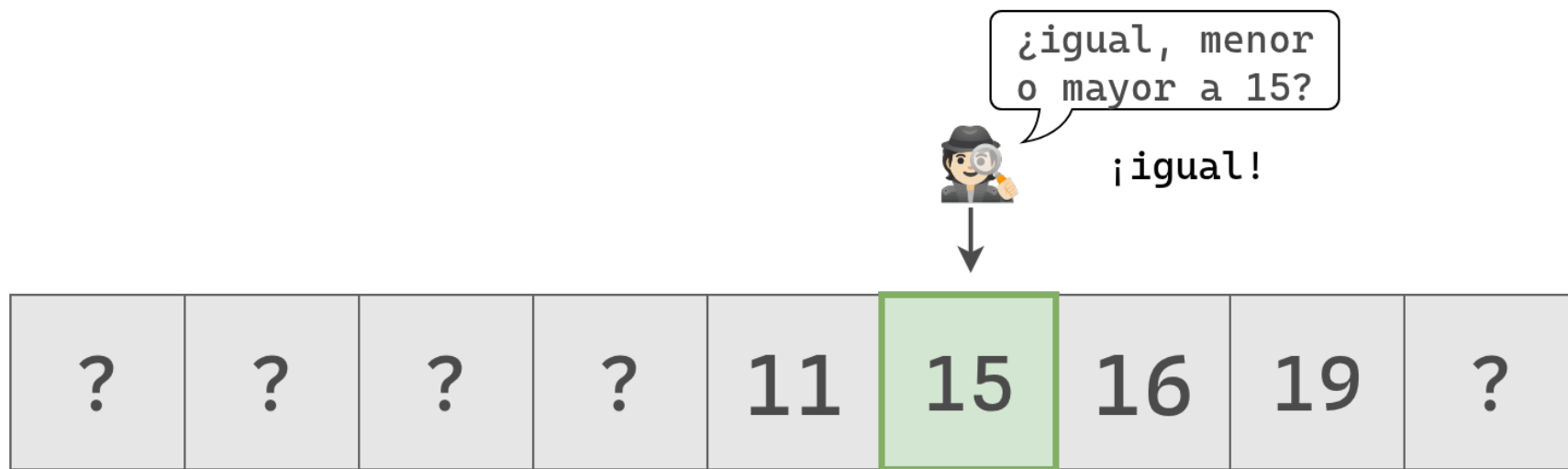
Inspeccionamos el centro de la mitad restante. El valor es mayor que 15.

Búsqueda binaria



Quedan dos valores posibles y elegimos el de la derecha.

Búsqueda binaria



Encontramos el 15 en la última celda posible.

Búsqueda binaria

- Encontramos el 15 en cuatro pasos.
- Con búsqueda lineal habríamos necesitado seis.
- La búsqueda binaria funciona sobre arreglos ordenados.

Búsqueda binaria

- Encontramos el 15 en cuatro pasos.
- Con búsqueda lineal habríamos necesitado seis.
- La búsqueda binaria funciona sobre arreglos ordenados.

En un arreglo desordenado no sabríamos hacia qué lado seguir buscando.

Búsqueda binaria

```
def busqueda_binaria(array, valor):  
    limite_inferior = 0  
    limite_superior = len(array) - 1  
    while limite_inferior ≤ limite_superior:  
        punto_medio = (limite_inferior + limite_superior) // 2  
        valor_en_punto_medio = array[punto_medio]  
  
        if valor == valor_en_punto_medio:  
            return punto_medio  
        elif valor < valor_en_punto_medio:  
            limite_superior = punto_medio - 1  
        elif valor > valor_en_punto_medio:  
            limite_inferior = punto_medio + 1  
    return None  
  
print(busqueda_binaria([3, 17, 75, 80, 202], 22))  
# None
```

Búsqueda lineal vs. búsqueda binaria

Con arreglos pequeños, la búsqueda binaria no tiene mucha ventaja.

Al aumentar el tamaño, la diferencia se vuelve notable.

Búsqueda lineal vs. búsqueda binaria

Con arreglos pequeños, la búsqueda binaria no tiene mucha ventaja.

Al aumentar el tamaño, la diferencia se vuelve notable.

- Con 100 elementos: hasta 100 pasos frente a 7.

Búsqueda lineal vs. búsqueda binaria

Con arreglos pequeños, la búsqueda binaria no tiene mucha ventaja.

Al aumentar el tamaño, la diferencia se vuelve notable.

- Con 100 elementos: hasta 100 pasos frente a 7.
- La búsqueda lineal duplica el trabajo al duplicar el tamaño.

Búsqueda lineal vs. búsqueda binaria

Con arreglos pequeños, la búsqueda binaria no tiene mucha ventaja.

Al aumentar el tamaño, la diferencia se vuelve notable.

- Con 100 elementos: hasta 100 pasos frente a 7.
- La búsqueda lineal duplica el trabajo al duplicar el tamaño.
- La binaria agrega un paso más.

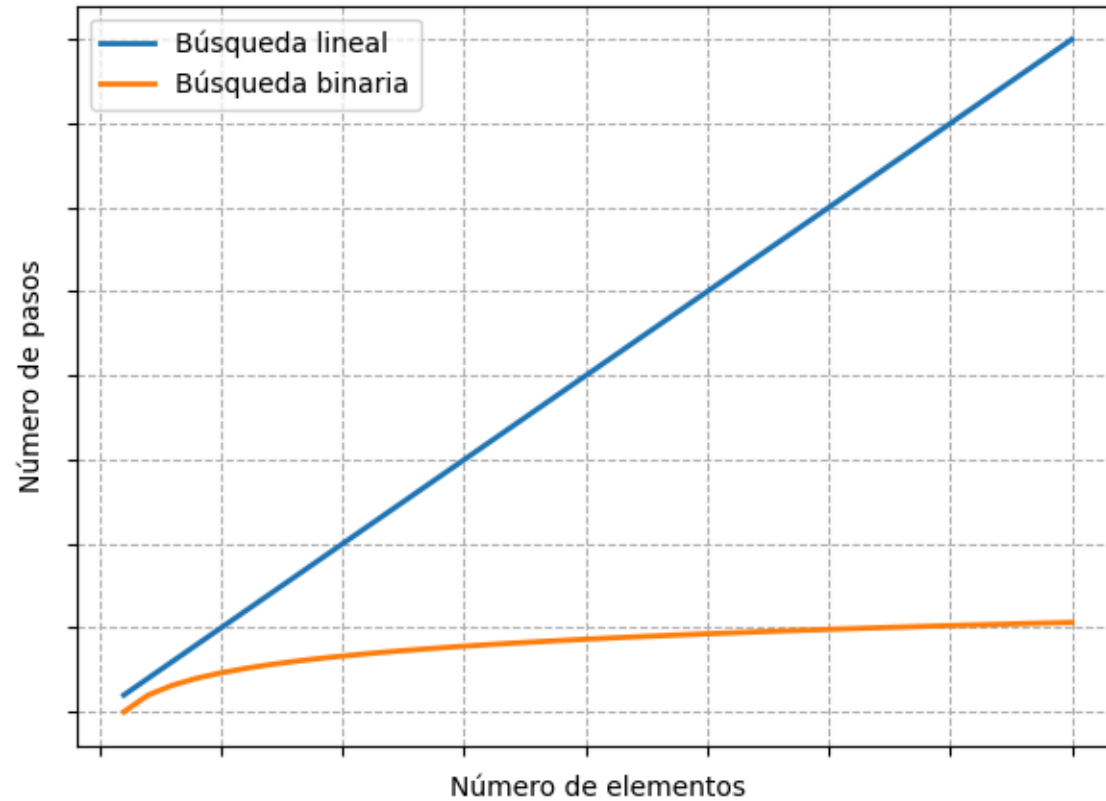
Búsqueda lineal vs. búsqueda binaria

Con arreglos pequeños, la búsqueda binaria no tiene mucha ventaja.

Al aumentar el tamaño, la diferencia se vuelve notable.

- Con 100 elementos: hasta 100 pasos frente a 7.
- La búsqueda lineal duplica el trabajo al duplicar el tamaño.
- La binaria agrega un paso más.
- Con 10.000 elementos, la binaria necesita hasta 14 pasos. Con un millón, 20.

Búsqueda lineal vs. búsqueda binaria



Nada es gratis

Cuando es posible, la búsqueda binaria es mucho más eficiente que la búsqueda lineal.

Nada es gratis

Cuando es posible, la búsqueda binaria es mucho más eficiente que la búsqueda lineal.

¿Cuál es el costo asociado?

Nada es gratis

Cuando es posible, la búsqueda binaria es mucho más eficiente que la búsqueda lineal.

¿Cuál es el costo asociado?

En un arreglo ordenado, la búsqueda es barata, pero la inserción es cara.

Nada es gratis

Cuando es posible, la búsqueda binaria es mucho más eficiente que la búsqueda lineal.

¿Cuál es el costo asociado?

En un arreglo ordenado, la búsqueda es barata, pero la inserción es cara.

¿Harás muchas búsquedas o muchas inserciones?

***Big* O**

Big O

La cantidad de pasos depende del tamaño del problema.

- Con 35 elementos, una búsqueda lineal puede requerir 35 pasos.
- Con 200 elementos, puede requerir 200.

Big O

La cantidad de pasos depende del tamaño del problema.

- Con 35 elementos, una búsqueda lineal puede requerir 35 pasos.
- Con 200 elementos, puede requerir 200.

En lugar de un número fijo, hablamos de N pasos para N elementos.

La notación *Big O* permite describir y comparar la eficiencia de los algoritmos.

***Big O*: es relativo**

Comparamos cómo crece la cantidad de pasos al aumentar la cantidad de elementos N .

***Big O*: es relativo**

Comparamos cómo crece la cantidad de pasos al aumentar la cantidad de elementos N .

- Búsqueda lineal: hasta N pasos $\rightarrow O(N)$, tiempo lineal.
- Lectura por índice: un paso para cualquier tamaño $\rightarrow O(1)$, tiempo constante.

***Big O*: es relativo**

Comparamos cómo crece la cantidad de pasos al aumentar la cantidad de elementos N .

- Búsqueda lineal: hasta N pasos $\rightarrow O(N)$, tiempo lineal.
- Lectura por índice: un paso para cualquier tamaño $\rightarrow O(1)$, tiempo constante.

En $O(1)$, el costo es independiente de la cantidad de elementos N .

***Big O*: lo que realmente importa**

La pregunta

¿Cómo crece la cantidad de pasos del algoritmo a medida que aumenta la cantidad de elementos N ?

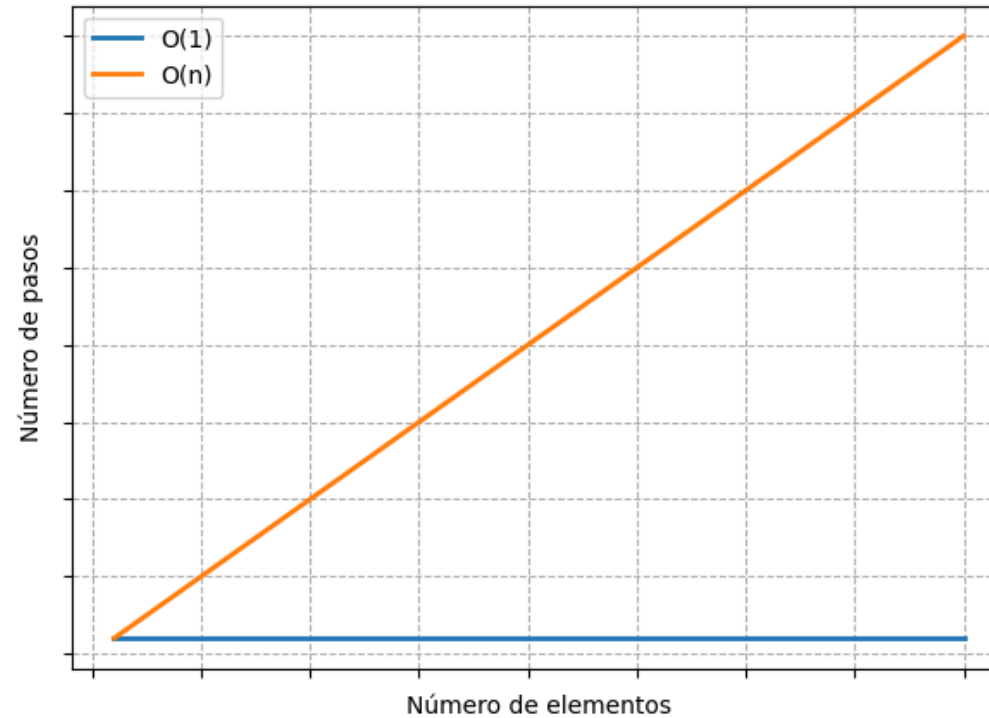
***Big O*: lo que realmente importa**

La pregunta

¿Cómo crece la cantidad de pasos del algoritmo a medida que aumenta la cantidad de elementos N ?

- Un algoritmo que siempre realiza tres pasos tiene costo constante.
- Lo expresamos como $O(1)$, igual que si siempre realizara un solo paso.
- Nos interesa cómo cambia el trabajo al aumentar el tamaño del problema.

Big O: lo que realmente importa



En este ejemplo, N pasos crecen con los datos y un paso se mantiene constante.

***Big O*: lo que realmente importa**

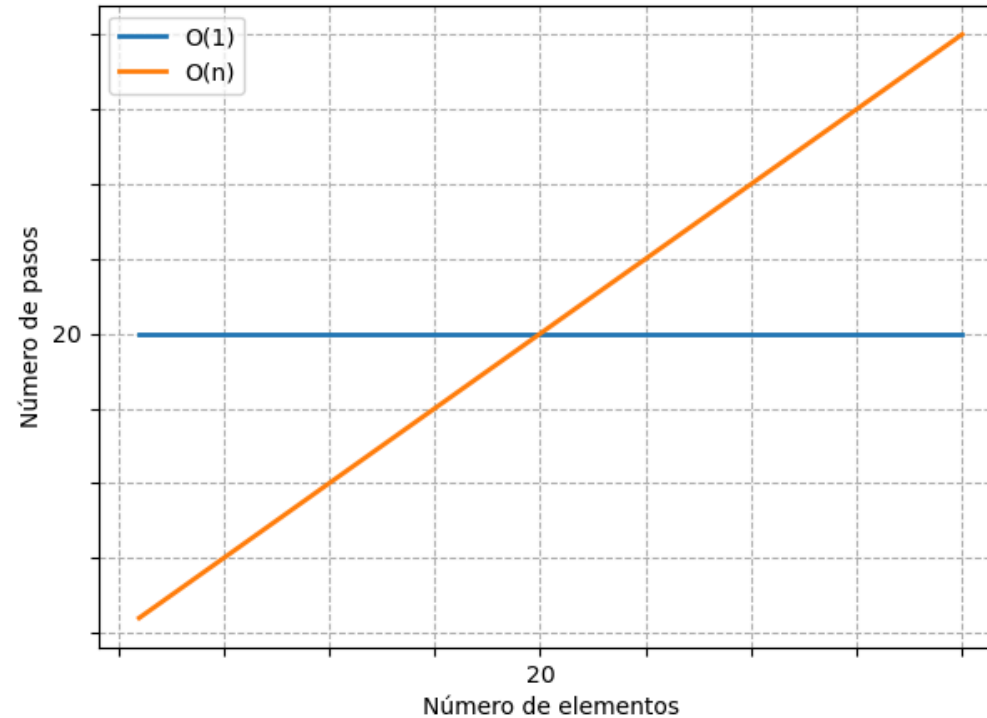
Imaginemos un algoritmo que siempre toma 20 pasos.

***Big O*: lo que realmente importa**

Imaginemos un algoritmo que siempre toma 20 pasos.

¿Es más o menos eficiente que uno que realiza N pasos?

Big O: lo que realmente importa



Comparamos un algoritmo de 20 pasos constantes con otro de N pasos.

***Big O*: lo que realmente importa**

- Con menos de 20 elementos, el algoritmo de N pasos hace menos trabajo.

***Big O*: lo que realmente importa**

- Con menos de 20 elementos, el algoritmo de N pasos hace menos trabajo.
- Con 20 elementos, ambos realizan la misma cantidad de pasos.

***Big O*: lo que realmente importa**

- Con menos de 20 elementos, el algoritmo de N pasos hace menos trabajo.
- Con 20 elementos, ambos realizan la misma cantidad de pasos.
- A partir de ahí, el de N pasos requiere cada vez más trabajo.

***Big O* pesimista**

El costo de la búsqueda lineal depende de dónde se encuentra el valor buscado.

- Mejor caso: primera posición $\rightarrow O(1)$.
- Peor caso: última posición o valor ausente $\rightarrow O(N)$.

***Big O* pesimista**

El costo de la búsqueda lineal depende de dónde se encuentra el valor buscado.

- Mejor caso: primera posición $\rightarrow O(1)$.
- Peor caso: última posición o valor ausente $\rightarrow O(N)$.

Big O puede aplicarse al mejor caso, al promedio o al peor caso.

Hablaremos del peor caso, salvo que se indique lo contrario.

Un tercer tipo de complejidad

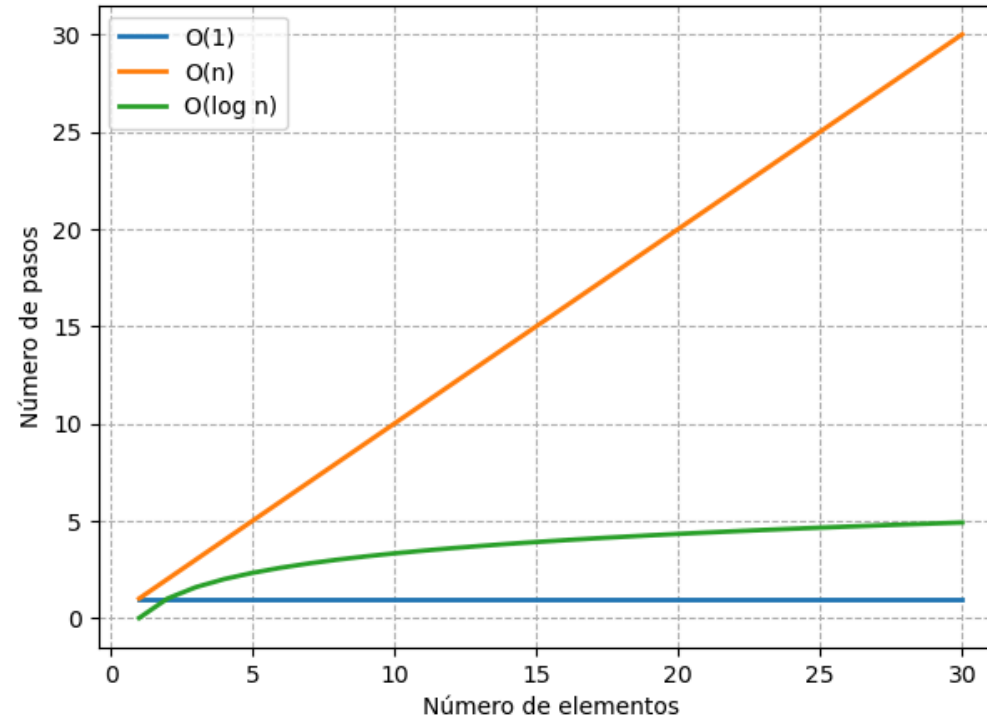
En la búsqueda binaria, el número de pasos aumenta con los datos, pero más lentamente que en la búsqueda lineal.

Un tercer tipo de complejidad

En la búsqueda binaria, el número de pasos aumenta con los datos, pero más lentamente que en la búsqueda lineal.

- Con 100 elementos, necesita como máximo 7 intentos.
- Una cota ajustada es $O(\log N)$.
- Se lee “O de log N” y corresponde al tiempo logarítmico.
- Duplicar los datos agrega como máximo un intento al peor caso.

Un tercer tipo



Comparamos el crecimiento constante, lineal y logarítmico.

Intuición detrás de $O(\log N)$

Reducimos el problema a la mitad en cada paso:

$$8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

- Con 8 elementos, hacen falta tres divisiones para llegar a uno.

Intuición detrás de $O(\log N)$

Reducimos el problema a la mitad en cada paso:

$$8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

- Con 8 elementos, hacen falta tres divisiones para llegar a uno.
- Con 16 elementos, una división más.

Intuición detrás de $O(\log N)$

Reducimos el problema a la mitad en cada paso:

$$8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

- Con 8 elementos, hacen falta tres divisiones para llegar a uno.
- Con 16 elementos, una división más.
- Con 32 elementos, otra más.

Intuición detrás de $O(\log N)$

Reducimos el problema a la mitad en cada paso:

$$8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

- Con 8 elementos, hacen falta tres divisiones para llegar a uno.
- Con 16 elementos, una división más.
- Con 32 elementos, otra más.
- El logaritmo en base 2 cuenta esas divisiones.

Intuición detrás de $O(\log N)$

Elementos	N	$\log_2 N$
8	8	3
16	16	4
128	128	7
1024	1024	10

Intuición detrás de $O(\log N)$

Elementos	N	$\log_2 N$
8	8	3
16	16	4
128	128	7
1024	1024	10

Cada vez que se duplica la cantidad de datos, agregamos una etapa de reducción.

Complejidad cuadrática

Queremos verificar si un arreglo de enteros contiene números repetidos.

```
[1, 5, 3, 9, 1, 4]
```

Complejidad cuadrática

Queremos verificar si un arreglo de enteros contiene números repetidos.

```
[1, 5, 3, 9, 1, 4]
```

Una primera idea: usar bucles anidados.

```
def contiene_duplicados(array):  
    for i in range(len(array)):  
        for j in range(len(array)):  
            if (i != j) and (array[i] == array[j]):  
                return True  
    return False
```

```
contiene_duplicados([1, 5, 3, 9, 4, 6])    # False  
contiene_duplicados([1, 5, 3, 9, 1, 4, 6]) # True
```

Complejidad cuadrática

¿Cuántos pasos realiza la función en el peor escenario?

Complejidad cuadrática

¿Cuántos pasos realiza la función en el peor escenario?

- El peor caso ocurre cuando todos los valores son distintos.

Complejidad cuadrática

¿Cuántos pasos realiza la función en el peor escenario?

- El peor caso ocurre cuando todos los valores son distintos.
- El bucle externo se ejecuta N veces.

Complejidad cuadrática

¿Cuántos pasos realiza la función en el peor escenario?

- El peor caso ocurre cuando todos los valores son distintos.
- El bucle externo se ejecuta N veces.
- Dentro de cada iteración, el segundo bucle se ejecuta N veces.

Complejidad cuadrática

¿Cuántos pasos realiza la función en el peor escenario?

- El peor caso ocurre cuando todos los valores son distintos.
- El bucle externo se ejecuta N veces.
- Dentro de cada iteración, el segundo bucle se ejecuta N veces.
- En total: $N \times N = N^2$ iteraciones.

Complejidad $O(N^2)$, también conocida como tiempo cuadrático.

Bucles anidados

Con frecuencia, aunque no siempre, un bucle dentro de otro tiene complejidad $O(N^2)$.

Solución lineal

¿Existe una alternativa más eficiente?

Solución lineal

¿Existe una alternativa más eficiente?

Consideremos calificaciones enteras de productos, de 0 a 10.

- Conocemos de antemano los valores posibles.
- Podemos usar un arreglo auxiliar con una posición para cada calificación.
- Realizamos como máximo un recorrido de la entrada.

Solución lineal

```
def contiene_duplicados_lineal(array):  
    valores_existentes = [0] * 11  
  
    for i in range(len(array)):  
        if valores_existentes[array[i]] == 1:  
            return True  
        else:  
            valores_existentes[array[i]] = 1  
  
    return False
```

Solución lineal

```
def contiene_duplicados_lineal(array):  
    valores_existentes = [0] * 11  
  
    for i in range(len(array)):  
        if valores_existentes[array[i]] == 1:  
            return True  
        else:  
            valores_existentes[array[i]] = 1  
  
    return False
```

```
contiene_duplicados_lineal([1, 5, 3, 9, 4, 6])    # False  
contiene_duplicados_lineal([1, 5, 3, 9, 1, 4, 6]) # True
```

Solución lineal

El arreglo `valores_existentes` comienza con once ceros.

Por ejemplo, si la entrada es `[3, 5, 8]`:

- Al encontrar el 3: `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`.

Solución lineal

El arreglo `valores_existentes` comienza con once ceros.

Por ejemplo, si la entrada es `[3, 5, 8]`:

- Al encontrar el 3: `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`.
- Al encontrar el 5: `[0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0]`.

Solución lineal

El arreglo `valores_existentes` comienza con once ceros.

Por ejemplo, si la entrada es `[3, 5, 8]`:

- Al encontrar el 3: `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`.
- Al encontrar el 5: `[0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0]`.
- Al encontrar el 8: `[0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0]`.

Solución lineal

- Si la posición ya contiene un 1, encontramos un duplicado y devolvemos `True`.

Solución lineal

- Si la posición ya contiene un 1, encontramos un duplicado y devolvemos `True`.
- Si completamos el recorrido, devolvemos `False`.

Solución lineal

- Si la posición ya contiene un 1, encontramos un duplicado y devolvemos `True`.
- Si completamos el recorrido, devolvemos `False`.
- Cada iteración realiza una cantidad constante de operaciones.

Solución lineal

- Si la posición ya contiene un 1, encontramos un duplicado y devolvemos `True`.
- Si completamos el recorrido, devolvemos `False`.
- Cada iteración realiza una cantidad constante de operaciones.
- Para N elementos, realizamos **hasta N iteraciones**.

Solución lineal

- Si la posición ya contiene un 1, encontramos un duplicado y devolvemos True.
- Si completamos el recorrido, devolvemos False.
- Cada iteración realiza una cantidad constante de operaciones.
- Para N elementos, realizamos **hasta N iteraciones**.

$O(N)$ es una cota superior de su complejidad temporal.

Letra chica

- Esta implementación supone valores enteros entre 0 y 10.
- Con más de once elementos, necesariamente aparecen duplicados.
- Se revisan como máximo doce elementos: $O(1)$ es una cota más ajustada.
- El arreglo auxiliar consume memoria adicional.

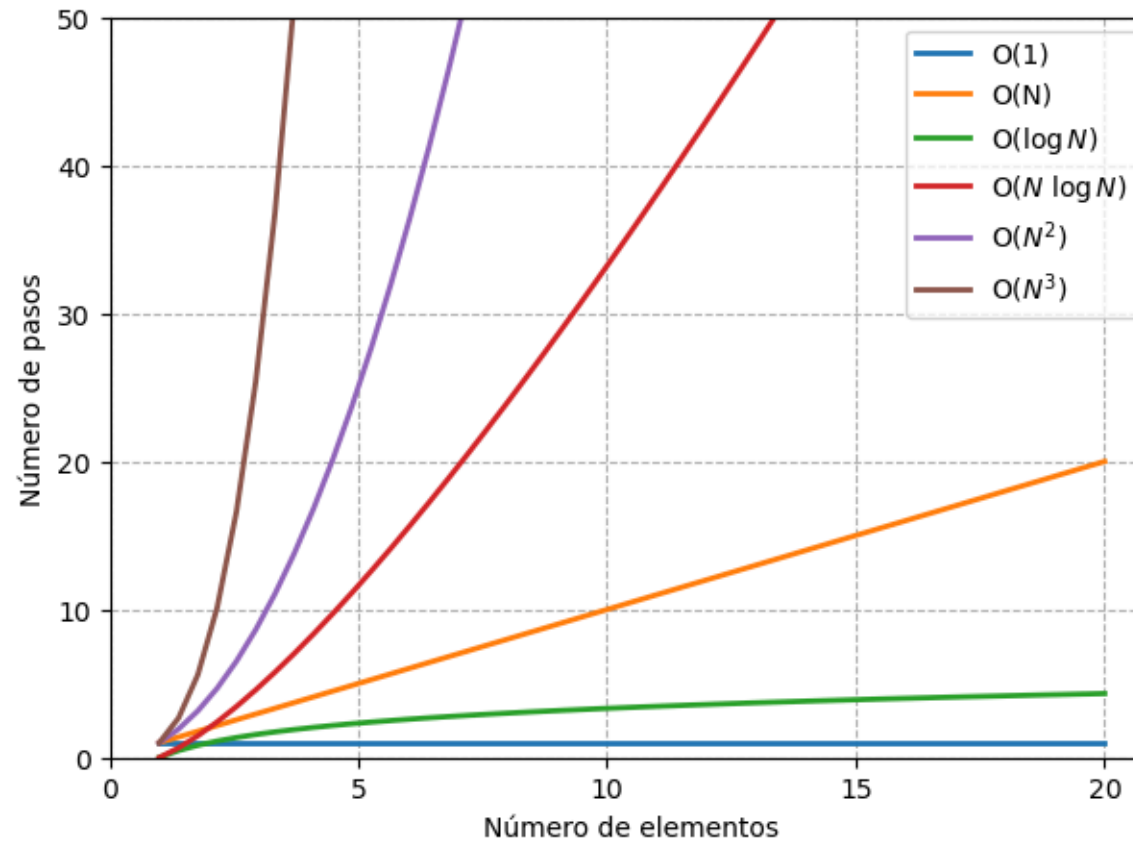
Resumen de complejidades

Vimos ejemplos de complejidad constante, lineal, logarítmica y cuadrática.

Otros ejemplos:

- Lineal-logarítmica: $O(N \log N)$.
- Cúbica: $O(N^3)$.
- Exponencial: $O(2^N)$.
- Factorial: $O(N!)$.

Resumen de complejidades



Gracias