El Misterio de Intro. a la Progra.

- Al profesor Nico le llega un correo de la universidad preguntando si el alumno Misterio Zallen está incluido en el curso iic1103, que tiene aprox. mil alumnos
- Pero la universidad aún no termina el nombramiento de Nico, por lo que Nico no puede ver su curso en Canvas
- Nico acude al profesor Yadran…

La lista de alumnos está ordenada según la hora de inscripción efectiva en el curso

¿ Zallen Misterio ∈

Alen Misterio
Misterio Misterio
Gonzalópez D
Zalen B
Gonzalópez J
Gazali Misterio
Misterio Yadran
Allen Javier
Zeta Hache
Ararán Jota
Alenn Cristóbal
… pág. 1/376

?

El problema se simplifica si la lista es ordenada alfabéticamente

¿ Zallen Misterio ∈

Aaa Yadran
Aab Cristóbal
Aac Yadran
Aca Javier
Acb Javier
Acb Yadran
Acb Cristóbal
Acc Yadran
Acd Javier
Ace Yadran
Ace Yadran
…
pág. 1/376

?

Similarmente, si los datos que hay buscar son números

¿ 245 ∈

1
2
3
56
57
64
68
99
124
125
126
… pág. 1/376

?

¿Qué es una secuencia de números ordenada?

Una secuencia de números $x_1, \cdots, x_n$ se dice **ordenada** (no decrecientemente) si cumple que

$$x_1 \leq \cdots \leq x_n$$

¿Qué es entonces **ordenar** una secuencia de números?

El algoritmo de ordenación del profesor Yadran

1. En la lista original, encontrar el menor valor
2. Borrarlo
3. Escribirlo al final (en el primer espacio disponible) de la lista nueva
4. Si quedan valores en la lista original, entonces volver al paso 1

¿Es correcto el algoritmo del profesor Yadran?

¿Qué quiere decir que un algoritmo sea **correcto**?

Para nosotros (en este curso) dos propiedades:

- termina en una cantidad finita de pasos
- cumple su propósito, es decir (en este caso), **ordena** los datos

Recordemos demostraciones por inducción

Demostración **por inducción**:

1.- **Caso base**. Se cumple para n=1.

2.- **Paso inductivo**. Si se cumple para n=k (*hipótesis inductiva*), entonces se cumple para n=k+1.

Ahora … a trabajar ustedes

Demuestra que el algoritmo anterior es correcto:

- termina en una cantidad finita de pasos
- cumple su propósito, es decir, **ordena** los datos

Termina en una cantidad finita de pasos

- La cantidad de valores a ordenar es finita, digamos *n.*

- En cada vuelta, borramos un valor de la lista original y lo escribimos en la lista nueva.

- Después de *n* vueltas, todos los valores en la lista original fueron borrados. Debido al paso 4, el algoritmo termina.

Cumple su propósito: ordena los datos

Demostración **por inducción**:

- **Caso base**. El primer valor borrado en la lista original y escrito en la lista nueva es el menor de todos (criterio de selección) y está ordenado (único valor en la lista nueva): ✓

- **Hipótesis inductiva**. Los *k* primeros valores borrados en la lista original y escritos en la lista nueva son los *k* valores más chicos y están ordenados

- **Por demostrar** (usando la hipótesis inductiva): *k*+1 **...**

Por demostrar, usando la hipótesis inductiva

Los *k*+1 primeros valores borrados en la lista original y escritos en la lista nueva son los *k*+1 valores más chicos y están ordenados:

- los primeros *k* valores en la lista nueva son los *k* más chicos (por hipótesis inductiva) y están borrados de la lista original (por paso 2); el siguiente valor que pasa a la lista nueva es el menor de los restantes (por criterio de selección) → el *k*+1 más chico

- los primeros *k* números en la hoja nueva están ordenados (por hipótesis inductiva); el siguiente número que se escribe al final de la hoja nueva no es menor que ninguno de los *k* números que ya están en la hoja nueva (por criterio de selección) → queda ordenado

¡Cumple su propósito!

Demostración **por inducción**:

1.- **Caso base**. Se cumple para n=1.

2.- **Paso inductivo**. Si se cumple para n=k (*hipótesis inductiva*), entonces se cumple para n=k+1.

El algoritmo *selection sort*

Para la secuencia inicial de datos, A:

1. Definir una secuencia ordenada, B, inicialmente vacía
2. Buscar el menor dato x en A
3. Sacar x de A e insertarlo al final de B
4. Si quedan elementos en A, volver a 2.

Raciocinio para determinar la complejidad de *selection sort*

Buscar el menor dato en A significa revisar A entero: $O(n)$

Este proceso se hace una vez por cada dato: n veces

La complejidad es entonces $n \cdot O(n) = O(n^2)$

Otra forma de calcular la complejidad de *selection sort*

También se puede hacer de manera explícita:

Buscar el mínimo cuesta $n - 1$, y el siguiente $n - 2$, y así:

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n^2 - n}{2}$$

$$T(n) \in O(n^2)$$

Complejidad de memoria de *selection sort*

selection sort se puede hacer en un solo **arreglo**, ya que $|A| + |B| = n$

Eso significa que no necesita memoria adicional … o, más precisamente, necesita $O(1)$ memoria adicional

Los algoritmos que tienen esta propiedad se conocen como *in place*

Los profesores tienen ahora otro problema

El profesor Yadran ya ordenó la lista de estudiantes de iic1103

La universidad solicita agregar 5 estudiantes a iic1103

El profesor Nico necesita actualizar el cambio en la lista

¿Cómo lo hace para no tener que volver a ordenar todo?

En realidad, este nuevo problema no es tan difícil

Dada una lista ya ordenada,

... insertar pocos elementos ordenadamente es (relativamente) barato

¿Cómo podemos usar este hecho para ordenar?

El algoritmo *insertion sort*

Para la secuencia inicial de datos, A:

1. Definir una secuencia ordenada, B, inicialmente vacía
2. Tomar el primer dato x de A y sacarlo de A
3. Insertar x en B de manera que B quede ordenado
4. Si quedan datos en A, volver a 2.

lista *B* de datos
ordenados:
I N O R S T

último dato
insertado
ordenadamente
en *B*: N

S O R T I N G E X A M P L E
O S R T I N G E X A M P L E
O **R** S T I N G E X A M P L E
O R S **T** I N G E X A M P L E
I O R S T N G E X A M P L E
I **N** O R S T G E X A M P L E
G I N O R S T E X A M P L E
E G I N O R S T X A M P L E
E G I N O R S T **X** A M P L E
A E G I N O R S T X M P L E
A E G I **M** N O R S T X P L E
A E G I M N O **P** R S T X L E
A E G I **L** M N O P R S T X E
A E **E** G I L M N O P R S T X
A E E G I L M N O P R S T X

próximo dato
a ser insertado
ordenadamente: G

lista *A* de datos
desordenados:
G E X A M P L E

¿Cómo se hace una inserción?

Depende de la estructura de datos usada para almacenar la lista

Se suele usar **arreglos**, pero también se puede usar **listas ligadas**

(las próximas diapos. describen la memoria del computador)

En cualquier caso, **el algoritmo no necesita memoria adicional**

La memoria (RAM) del computador: un experimento

Abre una consola de Python en tu computador

Ejecuta el siguiente código:

```
a = object()
print(a)
```

¿Qué significa lo que aparece en consola?

Bits y direcciones de memoria

Bit: unidad indivisible de información computacional que sólo puede valer 0 o 1 **;** celda física indivisible de almacenamiento

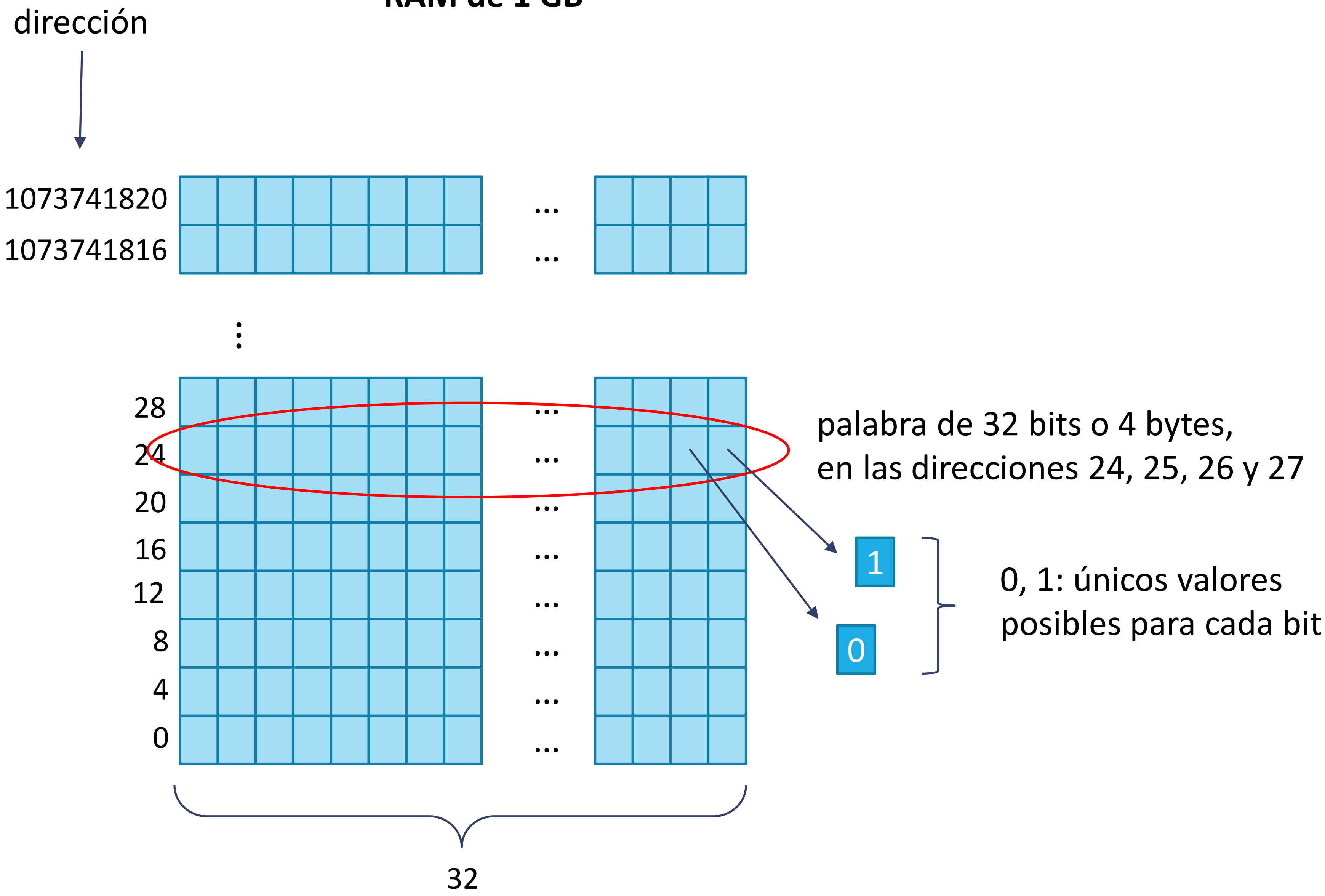
La memoria principal (RAM) de un computador puede ser imaginada como una gran tabla o matriz de bits:

- 32 columnas
- algunos miles de millones de filas

Cada fila —o **palabra** (*word*)— tiene una **dirección** única:

- la posición relativa de la fila dentro de la tabla
- un número natural que parte en 0 (la dirección de la primera fila) y aumenta de 4 en 4

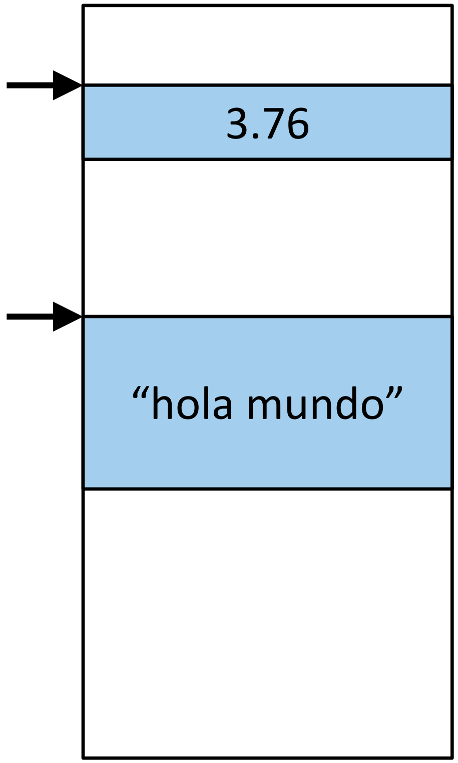
RAM de 1 GB


dirección
1073741820
1073741816
28
24
20
16
12
8
4
0
palabra de 32 bits o 4 bytes,
en las direcciones 24, 25, 26 y 27
1
0
0, 1: únicos valores
posibles para cada bit
32

Las variables de un programa, en C o Python, “viven” en la memoria del computador

Cada variable de un programa tiene:

- dirección (ubicación) en memoria
- tamaño, en número de bytes o en número de palabras
- valor



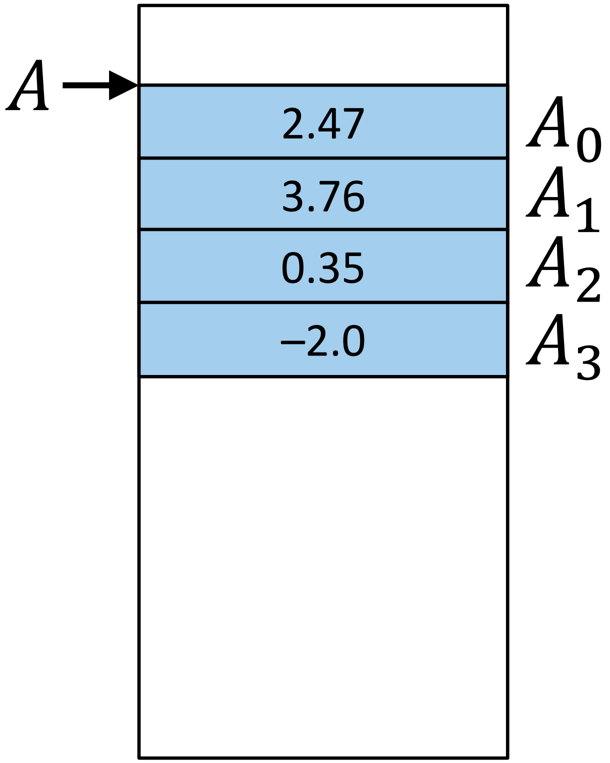
Un **arreglo** en un programa en C

Es una secuencia de **largo fijo** de "celdas" de memoria del mismo tamaño (en número de bytes o de palabras)

… y que almacenan valores del mismo tipo (todos *integer*, o todos *strings*)

Se almacena de manera **contigua** en memoria

Permite acceso por índice en tiempo O(1)



Arreglo: ejemplo abstracto

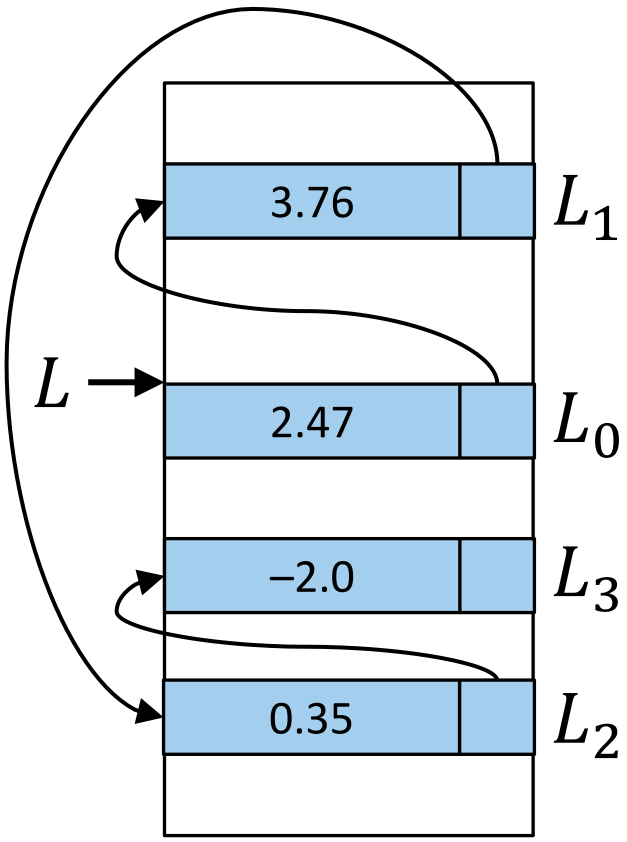
	2.47	**3.76**	**0.35**	**−2.0**	

Listas ligadas en un programa en C

Secuencia de **largo variable** de celdas del mismo tamaño

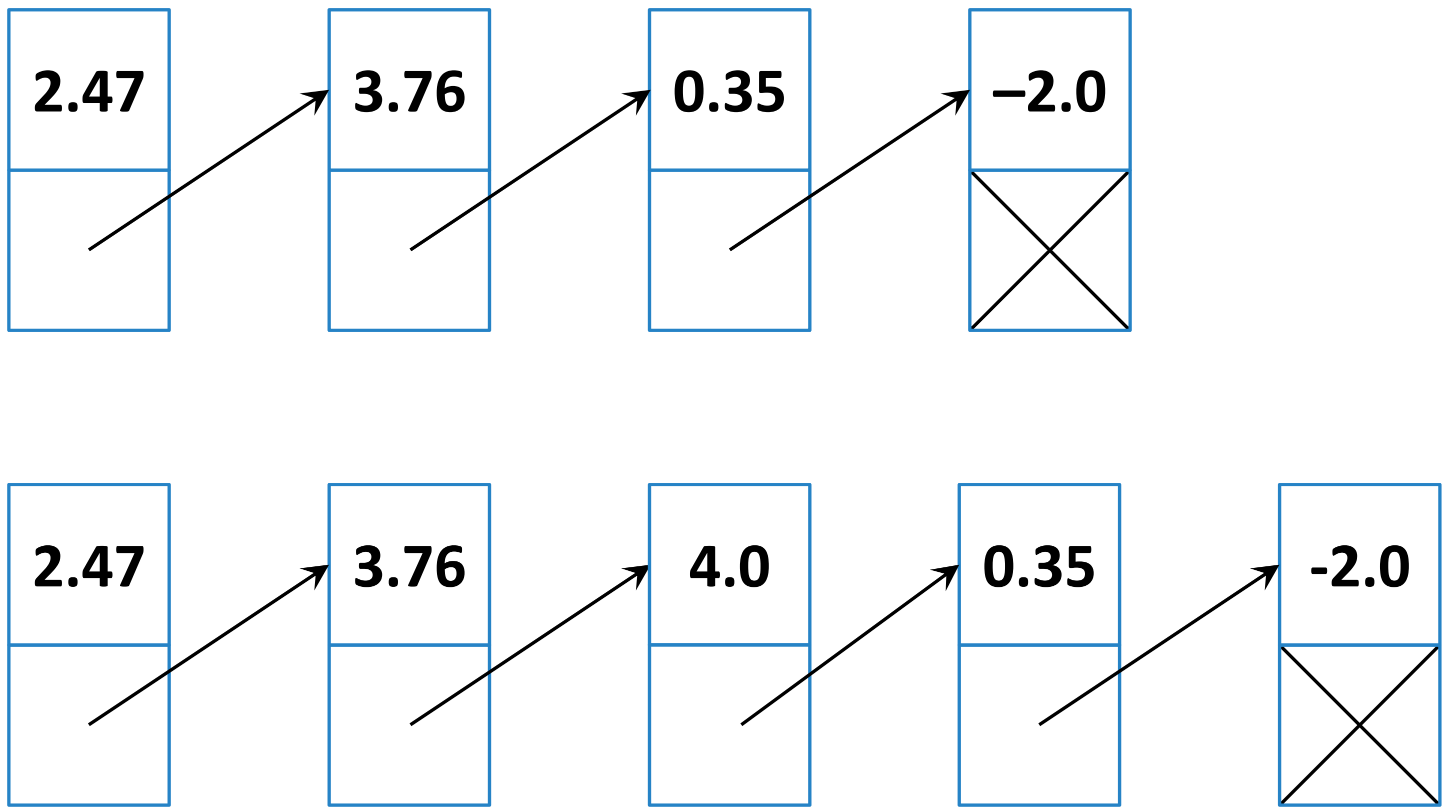
Las celdas ocupan posiciones **cualquiera** en la memoria, pero están conectadas entre ellas mediante **punteros**:

- palabras que contienen la dirección de la próxima celda



No permite acceso eficiente por índice

Lista ligada: ejemplo abstracto



Los dos pasos de la inserción

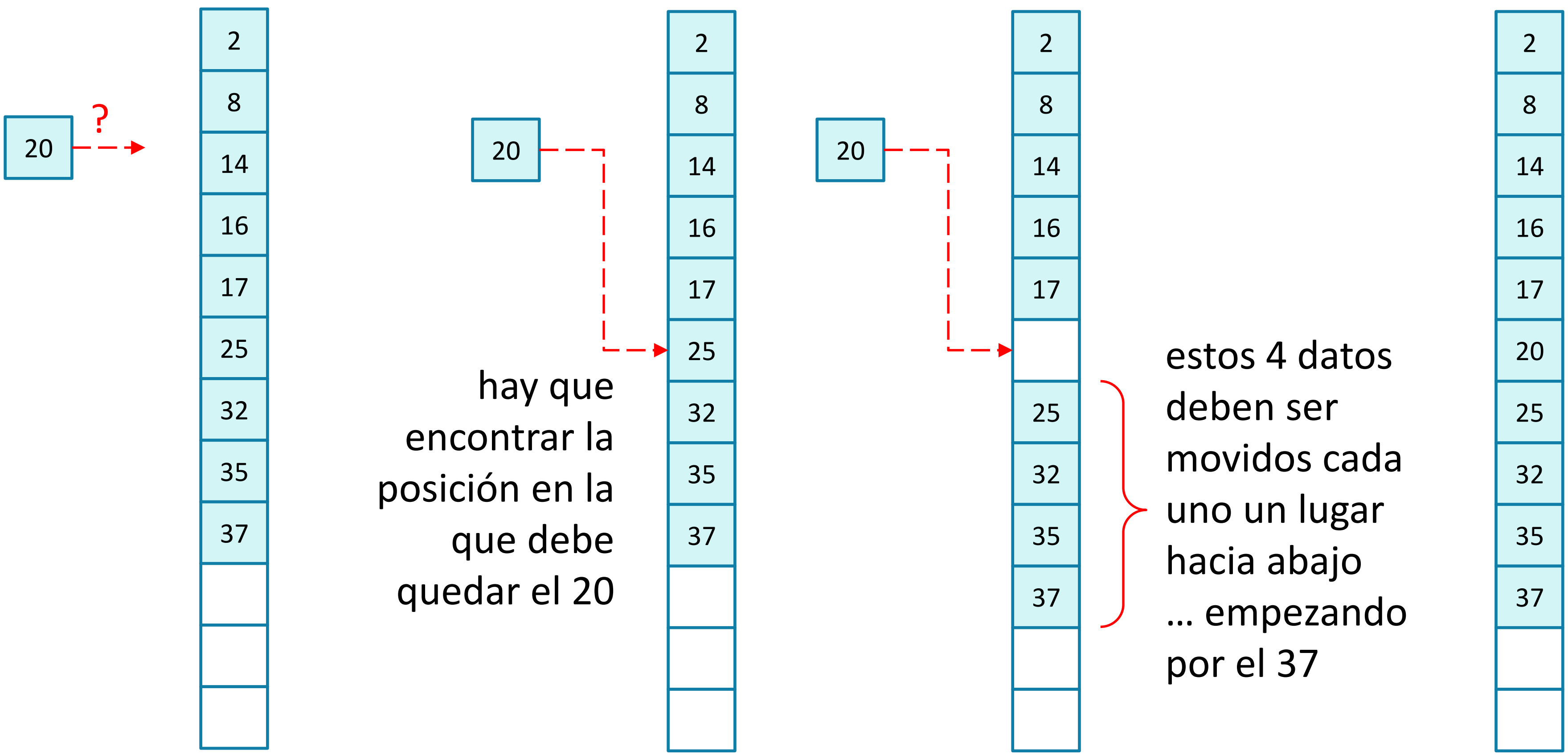
1. Primero, hay que buscar dónde corresponde insertar el dato

2. Luego, hay que llevar a cabo la inserción propiamente tal

¿Cuál es la complejidad usando **arreglos**?

¿Y con **listas ligadas**?

Insertando en un arreglo ordenado



Insertar en un arreglo

El primer paso —la búsqueda— podemos hacerlo en $O(\log n)$ con búsqueda binaria

Pero … para insertar x hay que desplazar todos los datos $> x$ un lugar hacia la derecha (o hacia abajo) → $O(n)$

Por lo tanto, en **arreglos**, **insertar** es $O(n)$

Insertar en una lista (doblemente*) ligada

Para el primer paso es necesario revisar toda la lista: $O(n)$

Teniendo el nodo donde corresponde insertar, hacerlo es $O(1)$

Por lo tanto, en **listas ligadas**, **insertar** es $O(n)$

*Es decir, cada celda tiene dos punteros: uno a la siguiente celda en la lista (como en las diapos. 29 y 30); y otro a la celda anterior en la lista

¿Es correcto *insertion sort*?

Para la secuencia inicial de datos, A:

1. Definir una secuencia ordenada, B, inicialmente vacía
2. Tomar el primer dato x de A y sacarlo de A
3. Insertar x en B de manera que B quede ordenado
4. Si quedan datos en A, volver a 2.

Demostración de finitud

En cada paso se saca un dato de A y se inserta en B

Cuando no quedan datos en A, el algoritmo termina

La inserción requiere como máximo recorrer todo B

Como A y B son finitos, el algoritmo termina en tiempo finito

Demostración, por inducción, de que cumple con su propósito

PD: Al terminar la n-ésima iteración, B se encuentra ordenada

Caso Base: Después de la primera iteración, B tiene un solo dato

$\rightarrow B$ está ordenada

Hipótesis Inductiva: Después de la i-ésima iteración**,** B está ordenada

Demostraremos que después de la iteración $i + 1$, B está ordenada

Extraemos el primer dato de A, y lo insertamos ordenadamente en B.
Termina el paso $i + 1$ y B tiene $i\ +\ 1$ datos ordenados.

Por inducción, al terminar el algoritmo después del paso n, B está ordenada.

Entonces *insertion sort* es $O(?)$

¿Qué tiempo toma si los datos vienen ordenados?

A E E G I L M N O P R S T X

$$\boldsymbol{insertionSort(A, n)}:$$

$$\boldsymbol{for\ i = 1 \,..\, n - 1}:$$

$$\boldsymbol{j = i}$$

$$\boldsymbol{while}\ (\boldsymbol{j} > \mathbf{0}) \wedge (\boldsymbol{A}[\boldsymbol{j}] < \boldsymbol{A}[\boldsymbol{j} - \mathbf{1}]):$$

$$\text{Intercambiar } \boldsymbol{A}[\boldsymbol{j}] \text{ con } \boldsymbol{A}[\boldsymbol{j} - \mathbf{1}]$$

$$\boldsymbol{j = j - 1}$$

Complejidad de *insertion sort*

Parecería que la complejidad de *insertion sort* depende de qué tan ordenados vienen los datos

¿Cómo podemos medir "qué tan ordenados vienen los datos"?

Inversiones

Sea A un arreglo con n números distintos de 1 a n

Si $i < j$ pero $A[i] \; > \; A[j]$, entonces se dice que el par ordenado (i, j) es una **inversión**

El número de inversiones es una medida de **desorden**

Inversiones: ejemplo

P.ej., el arreglo

$$A = [\ 34\ \ 8\ \ 64\ \ 51\ \ 32\ \ 21\ ]$$

tiene 9 inversiones:

(34, 8), (34, 32), (34, 21), (64, 51), (64, 32), (64, 21),

(51, 32), (51, 21), (32, 21)

¿Cómo depende *insertion sort* del número de inversiones?

Tenemos un arreglo A de largo n que tiene k **inversiones**

¿Cuánto tiempo toma *insertion sort* en ordenar A?

¿Cuántas inversiones se arreglan con un intercambio?

$$\boldsymbol{insertionSort(A, n)}:$$

$$\boldsymbol{for\ i = 1 \ldots n - 1}:$$

$$\boldsymbol{j = i}$$

$$\boldsymbol{while}\ (\boldsymbol{j > 0}) \wedge (\boldsymbol{A[j] < A[j - 1]}):$$

$$\text{Intercambiar } \boldsymbol{A[j]} \text{ con } \boldsymbol{A[j - 1]}$$

$$\boldsymbol{j = j - 1}$$

Antes de cada intercambio se hace una comparación entre los datos con índices j y j–1

Los datos se intercambian sólo si el par de índices $(j-1, j)$ es una inversión

- es decir, si $A[j–1] > A[j]$

Por lo tanto, **cada intercambio de datos (adyacentes) en el arreglo corrije exactamente una inversión**

Además, cada dato se compara al menos una vez

Complejidad de *insertion sort*

La complejidad es entonces $O(n+k)$

cada dato se compara
al menos una vez

número de inversiones

Complejidad de *insertion sort*

La complejidad es entonces $O(n + k)$

cada dato se compara
al menos una vez

número de inversiones

¿Qué valor tiene k en el mejor caso?

Complejidad de *insertion sort*

La complejidad es entonces $O(n + k)$

cada dato se compara
al menos una vez

número de inversiones

¿Qué valor tiene k en el mejor caso? ¿Y el en peor?

- mejor caso: 0 (no hay inversiones)

Complejidad de *insertion sort*

La complejidad es entonces $O(n + k)$

cada dato se compara
al menos una vez

número de inversiones

¿Qué valor tiene k en el mejor caso? ¿Y el en peor?

- mejor caso: 0 (no hay inversiones)
- peor caso: $(n^2 - n)/2$ (todos los pares posibles* son inversiones
  → el arreglo está ordenado de mayor a menor)

* pares posibles = $\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2!} = \frac{n^2-n}{2}$

Complejidad de *insertion sort*

La complejidad es entonces $O(n + k)$

cada dato se compara
al menos una vez

número de inversiones

¿Qué valor tiene k en el mejor caso? ¿Y el en peor?

- mejor caso: 0 (no hay inversiones)
- peor caso: $(n^2 - n)/2$ (todos los pares posibles* son inversiones
  → el arreglo está ordenado de mayor a menor)

¿Qué hay del *caso promedio*?

El caso promedio

¿Cuál es el número promedio de inversiones en un arreglo con *n* datos?

- lo vamos a definir como **el promedio aritmético del número de inversiones en cada una de las *n*! permutaciones de los *n* datos**

Suponemos que

- no hay datos repetidos*
- todas las permutaciones de los *n* datos son igualmente probables (de que aparezcan como input)

(* → podemos suponer que los *n* datos son los números 0, 1, 2, …, *n*–1)

¿Cómo calculamos el promedio aritmético del número de inversiones de las n! permutaciones?

- podríamos contar el número de inversiones en cada permutación, luego sumar y finalmente dividir por n!

En vez de contar, sumar y dividir, observemos que

… para cualquier permutación L, consideremos la permutación inversa L_r (hay $n!/2$ parejas distintas de permutaciones definidas de esta manera):

- p.ej., si $n = 10$ y $L = \{ 8, 1, 4, 9, 0, 3, 5, 2, 7, 6 \}$

  … entonces $L_r = \{ 6, 7, 2, 5, 3, 0, 9, 4, 1, 8 \}$

Tomemos cualquier par de elementos (x, y), con $y \neq x$

L y L_r tienen la propiedad de que en **exactamente una** de ellas el par ordenado de los índices de x y y es una inversión:

- p.ej., si $(x, y) = (9, 5)$, entonces el par de los índices es (3, 6), y

  … (3, 6) **es** una inversión en L, ya que $9 > 5$, pero **no** es una inversión en L_r

- p.ej., si $(x, y) = (0, 7)$, entonces el par de los índices es (4, 8), y

  … (4, 8) **no** es una inversión en L, ya que $0 < 7$, pero **sí** es una inversión en L_r

$L = \{ 8, 1, 4, 9, 0, 3, 5, 2, 7, 6 \}$

Es decir, un par de índices (i,j) es una inversión en L o es una inversión en L_r (y no cabe otra posibilidad)

… así, el número total de inversiones en L más el número total de inversiones en L_r debe ser igual al número total de pares posibles entre n datos: $n(n–1)/2$

Es decir, cada dos permutaciones (L y su respectiva L_r) el número de inversiones es exactamente $n(n–1)/2$

… por lo tanto, una permutación promedio tiene la mitad de esta cantidad de inversiones: $n(n–1)/4 = O(n^2)$

Complejidad de *insertion sort*

La cantidad de inversiones promedio en un arreglo de *n* elementos distintos es $O(n^2)$

… por lo que *insertion sort* es $O(n^2)$ en el caso promedio

Ahora, más allá de *insertion sort* …

Si un algoritmo sólo corrige una inversión por intercambio, no puede ordenar más rápido que $O(n^2)$ en promedio y por lo tanto en el peor caso

Algoritmos de Ordenación

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
SelectionSort	?	?	?	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
QuickSort	?	?	?	?
MergeSort	?	?	?	?
HeapSort	?	?	?	?

Algoritmos de Ordenación

Algoritmo	**Mejor caso**	**Caso promedio**	**Peor caso**	**Memoria adicional**
SelectionSort	?	?	?	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
QuickSort	?	?	?	?
MergeSort	?	?	?	?
HeapSort	?	?	?	?

Ejercicio propuesto:

- Escribir pseudocódigo de SelectionSort
- Especificar casos mejor, promedio y peor.