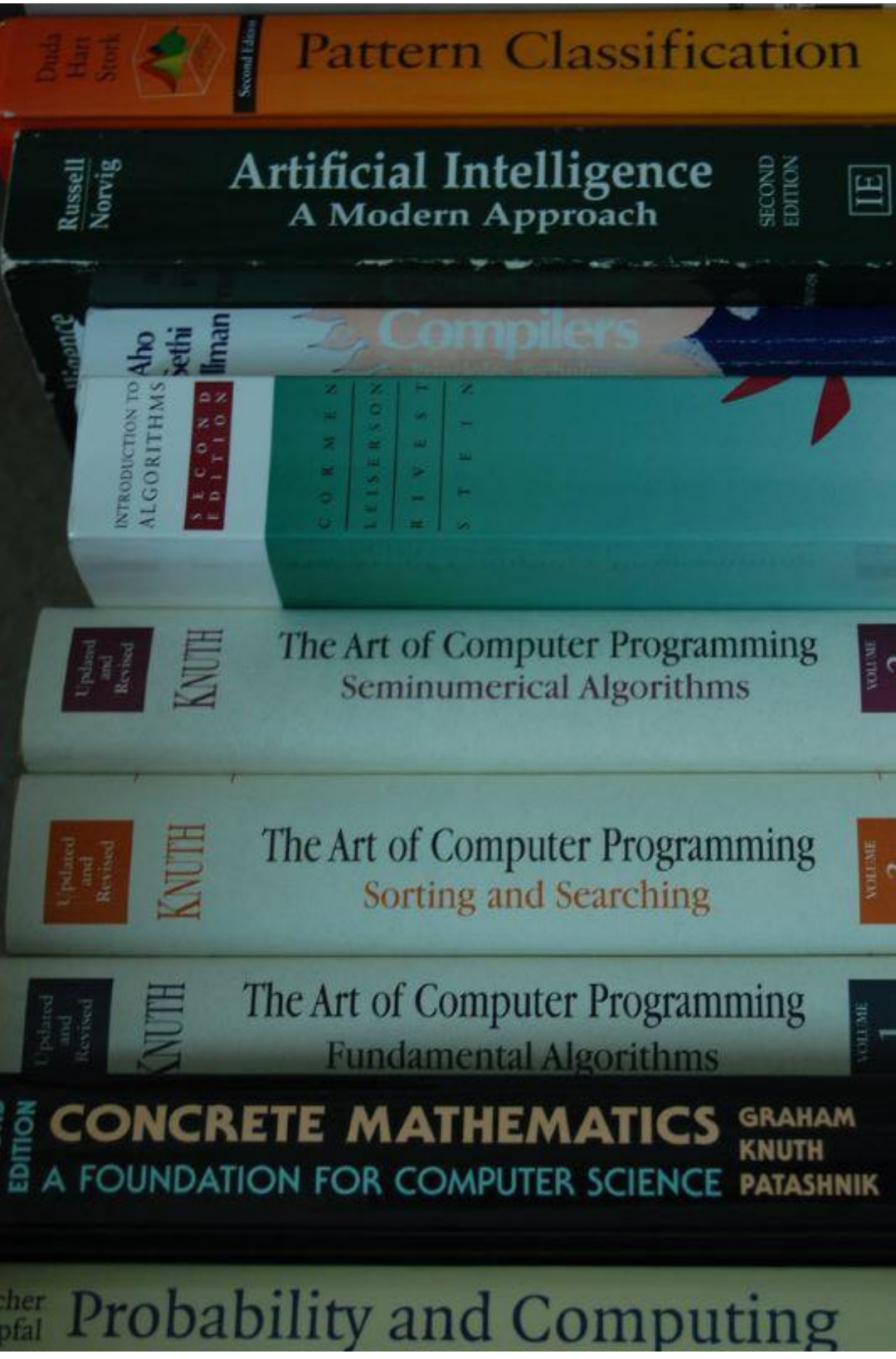
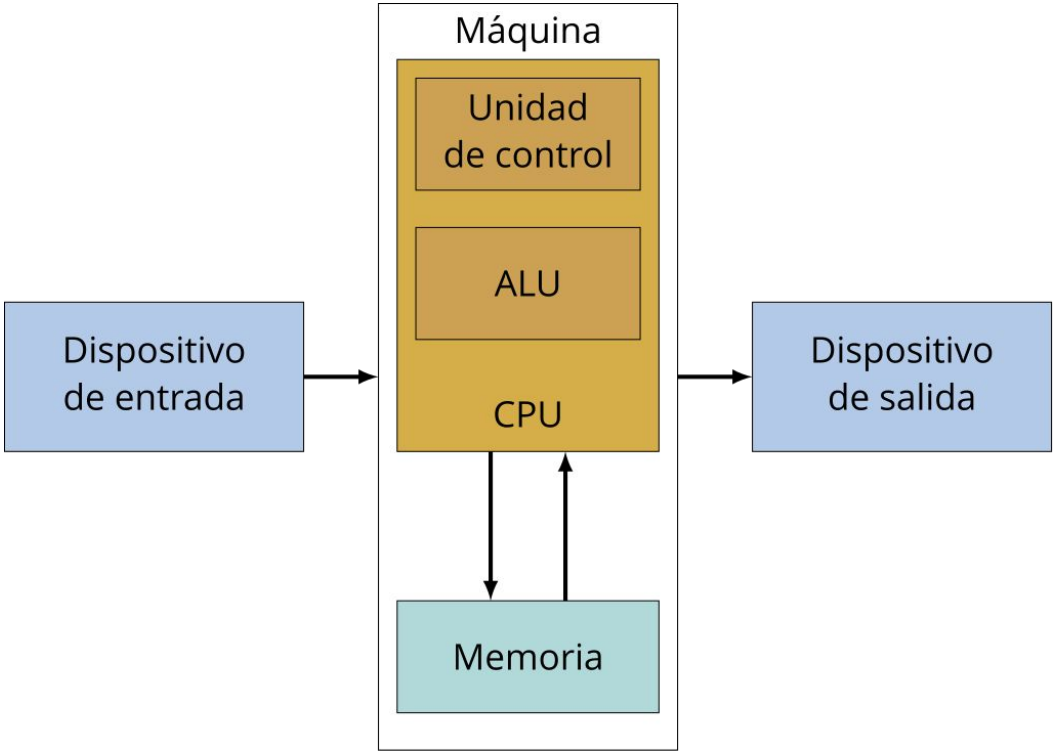
Algoritmos de Ordenamiento


Duda
Hart
Stork
Second Edition
Pattern Classification
Russell
Norvig
Artificial Intelligence
A Modern Approach
SECOND EDITION
IE
Compilers
Aho
ethi
llman
INTRODUCTION TO ALGORITHMS
SECOND EDITION
CORMEN
LEISERSON
RIVEST
STEIN
Updated and Revised
KNUTH
The Art of Computer Programming
Seminumerical Algorithms
VOLUME
Updated and Revised
KNUTH
The Art of Computer Programming
Sorting and Searching
VOLUME
Updated and Revised
KNUTH
The Art of Computer Programming
Fundamental Algorithms
VOLUME
EDITION
CONCRETE MATHEMATICS
A FOUNDATION FOR COMPUTER SCIENCE
GRAHAM
KNUTH
PATASHNIK
cher
pfal
Probability and Computing

Ordenamiento y búsqueda están relacionados

Me interesa ordenar, para buscar más rápido.

La CPU busca permanentemente datos en memoria a través del bus, un camino lento y angosto



Tiempos de latencia

1 nanosegundo (ns) = 10^{-9} segundos

- Operación básica computadora: 1 ns
- Acceso a memoria principal (RAM): 100 ns
- Acceso a disco: 10 000 000 ns

si fuese 1 segundo
casi 2 minutos
4 meses

Reglas del juego en el ordenamiento

- Los algoritmos de ordenamiento reordenan arreglos de elementos donde cada elemento tiene una clave.
- El objetivo de los algoritmos de ordenamiento es reordenar los elementos de tal forma que las claves estén ordenadas de acuerdo a alguna regla de ordenamiento (usualmente numérica o alfabética).

Input: Arreglo X[] de n elementos

Output: Una permutación de X tal que X[0] <= X[1] <= X[2] … <= X[n-1]

Algoritmos de Ordenamiento

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort (no lo vemos en Programación 1)
- QuickSort (no lo vemos en Programación 1)

Bubble Sort: visualización

6 5 3 1 8 7 2 4

Bubble Sort: idea

Si vemos los elementos en un arreglo ordenado: el elemento más grande está en la posición (n-1), el segundo más grande en la posición (n-2), y así.

La idea básica es recorrer el arreglo de 0 a n-1, y colocar el elemento más grande en la posición (n-1). Ahora, recorremos otra vez el arreglo desde el 0 al (n-2) y colocamos el segundo elemento más grande en la posición (n-2) y así sucesivamente.

En general, en la i-ésima iteración, colocamos el i-ésimo máximo elemento en la posición (n-i).

El proceso continúa hasta que todo el arreglo esté ordenado.

Bubble Sort: pseudocódigo

```
BUBBLESORT(A)

n=length(A)
for i=0 to n-2
     for j=0 to n-i-2
          if A[j] > A[j+1]
               Intercambiar A[j] con A[j+1]
```

Bubble Sort: análisis

Corremos 2 ciclos anidados donde las comparaciones y los intercambios son las operaciones claves. Independientemente de la entrada, las operaciones de comparación se ejecutarán siempre. Los intercambios suceden solamente si X[j]>X[j+1]. En cada *i* iteración del ciclo externo, el ciclo interno hace (n-i-1) iteraciones. Entonces, la cantidad total de iteraciones es la sumatoria de n-i-1 desde i=0 hasta n-2 = (n-1)+(n-2)+...+2+1=n(n-1)/2=$O(n^2)$

Peor caso: Array ordenado en orden decreciente. Operaciones de comparación: $O(n^2)$. Operaciones de intercambio: $O(n^2)$. Complejidad temporal: $2O(n^2)=O(n^2)$

Mejor caso: Array ordenado en orden creciente. Operaciones de comparación: $O(n^2)$. Operaciones de intercambio: 0. Complejidad temporal: $O(n^2)$

Selection Sort: visualización

El naranja es la sublista ordenada
El verde es el ítem actual
El rojo es el mínimo actual

Selection Sort: idea

Recorremos todo el array para encontrar el elemento mínimo, e intercambiamos el elemento mínimo con el primer elemento del array. Entonces buscamos el elemento mínimo en el subarray resultante de excluir el primer elemento, y lo intercambiamos con el segundo elemento del array. Continuamos hasta que todo el array esté ordenado.

En general, en el paso i-ésimo del algoritmo, se mantienen 2 subarrays:

- El subarray ordenado X[0… i-1]
- El subarray desordenado X[i… n-1]

Construimos el subarray ordenado en forma incremental, encontrando el mínimo del subarray desordenado y agregándolo al final del subarray ordenado. En cada iteración, el subarray ordenado crece en 1, y el subarray desordenado se decrementa en 1.

Selection Sort: pseudocódigo

```
SELECTION-SORT(A)

for i = 0 to length(A)-2
//min es el índice del mínimo
     min = i
     for j=i+1 to length(A)-1
          if A[j] < A[min]
               min = j
     auxiliar = A[i]
     A[i] = A[min]
     A[min] = auxiliar
```

Selection Sort: análisis

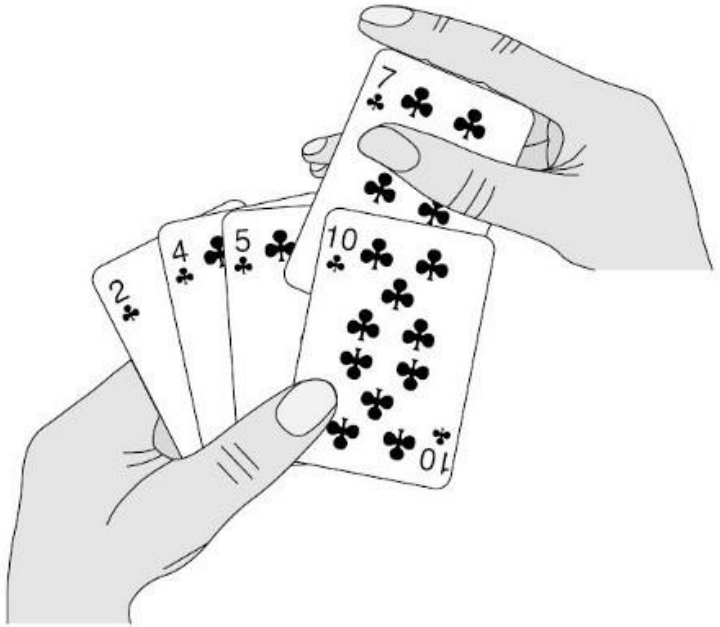
Corremos 2 ciclos anidados donde las comparaciones, los intercambios y la actualización del mínimo son las operaciones claves. Independientemente de la entrada, las operaciones de comparación e intercambio se ejecutarán siempre. La actualización del mínimo sucede solamente si X[j]<X[min]. En cada *i* iteración del ciclo externo, el ciclo interno hace (n-i) iteraciones. Entonces, la cantidad total de iteraciones es la sumatoria de n-i desde i=0 hasta n-2 = $(n)+(n-1)+(n-2)+...+2+1=n(n+1)/2=O(n^2)$

Peor caso: Array ordenado en orden decreciente. $O(n^2)$
Mejor caso: Array ordenado en orden creciente (ahorra actualizar el mínimo). $O(n^2)$

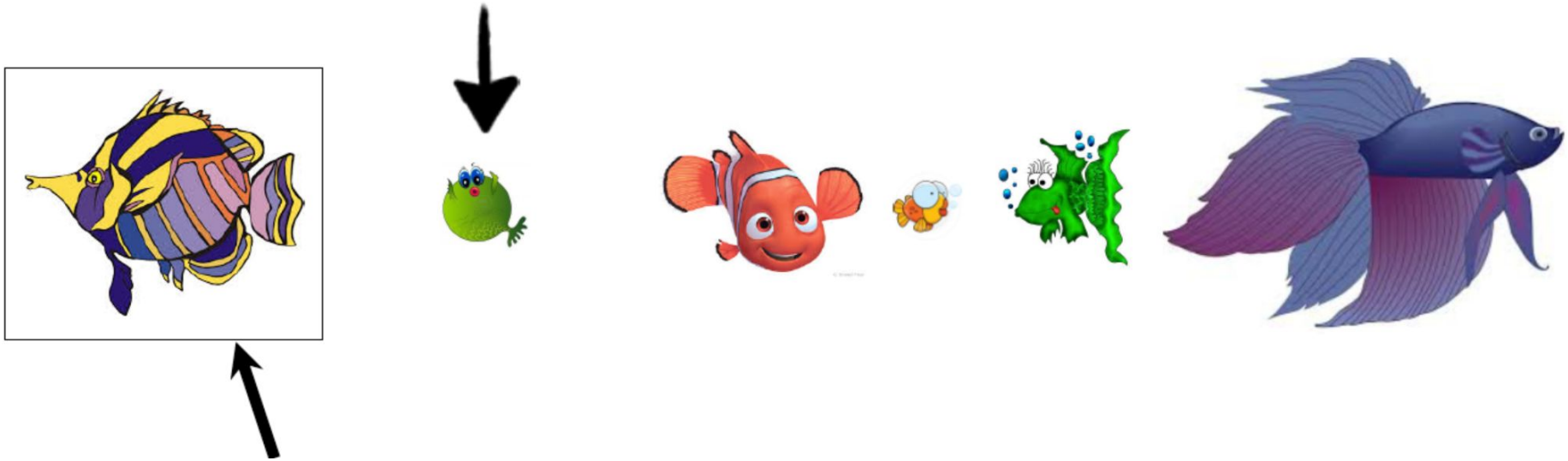
Insertion Sort: idea

El mismo método que cuando ordenamos una mano de cartas:

- Empezamos con la mano izquierda vacía y las cartas en la mesa.
- Agarramos una carta [clave] a la vez de la mesa [array desordenado], y la insertamos en la posición correcta en la mano izquierda [array ordenado].
- Encontramos la posición correcta, comparándola con las cartas en la mano, de derecha a izquierda.

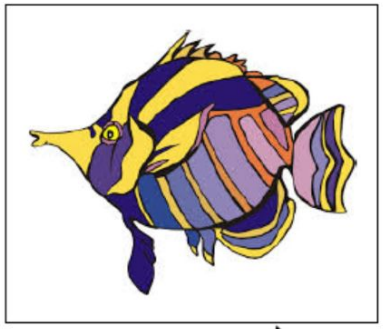


Insertion Sort: visualización



Parte ordenada

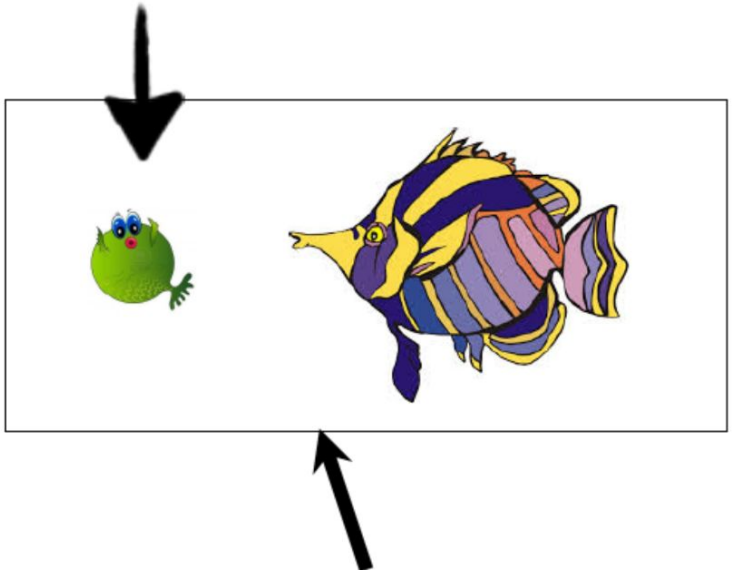
Insertion Sort: visualización



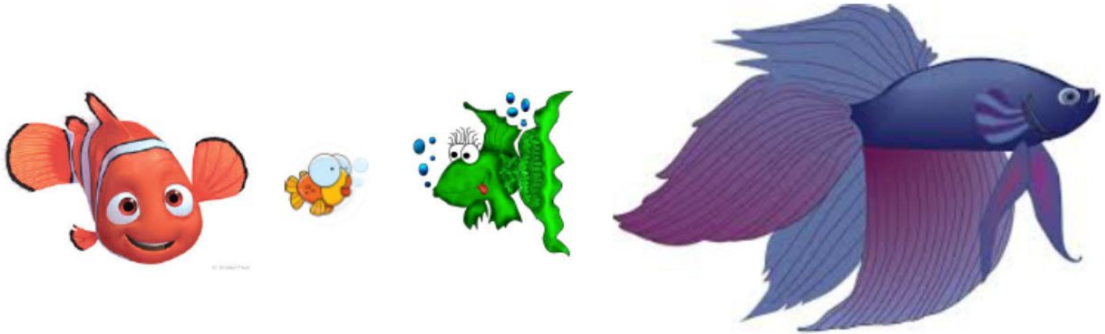
Parte ordenada

Insertion Sort: visualización

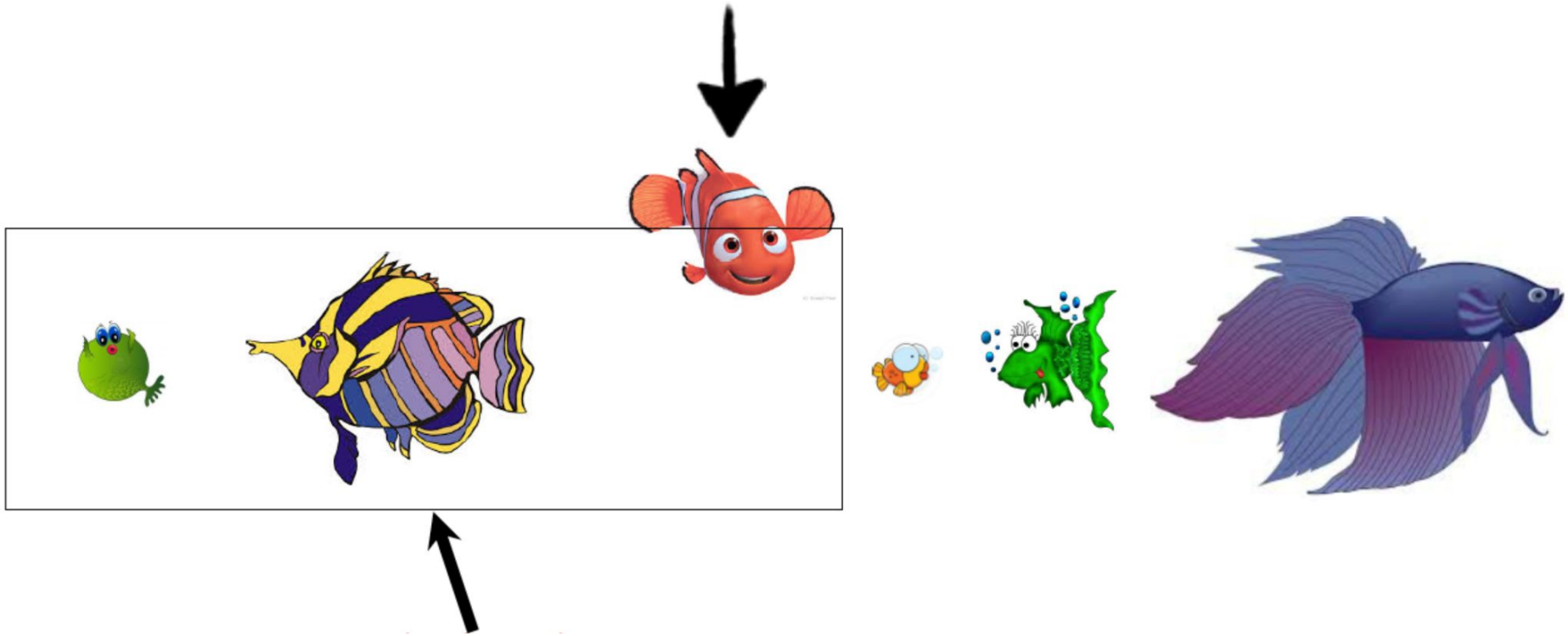
Insertion Sort: visualización



Parte ordenada

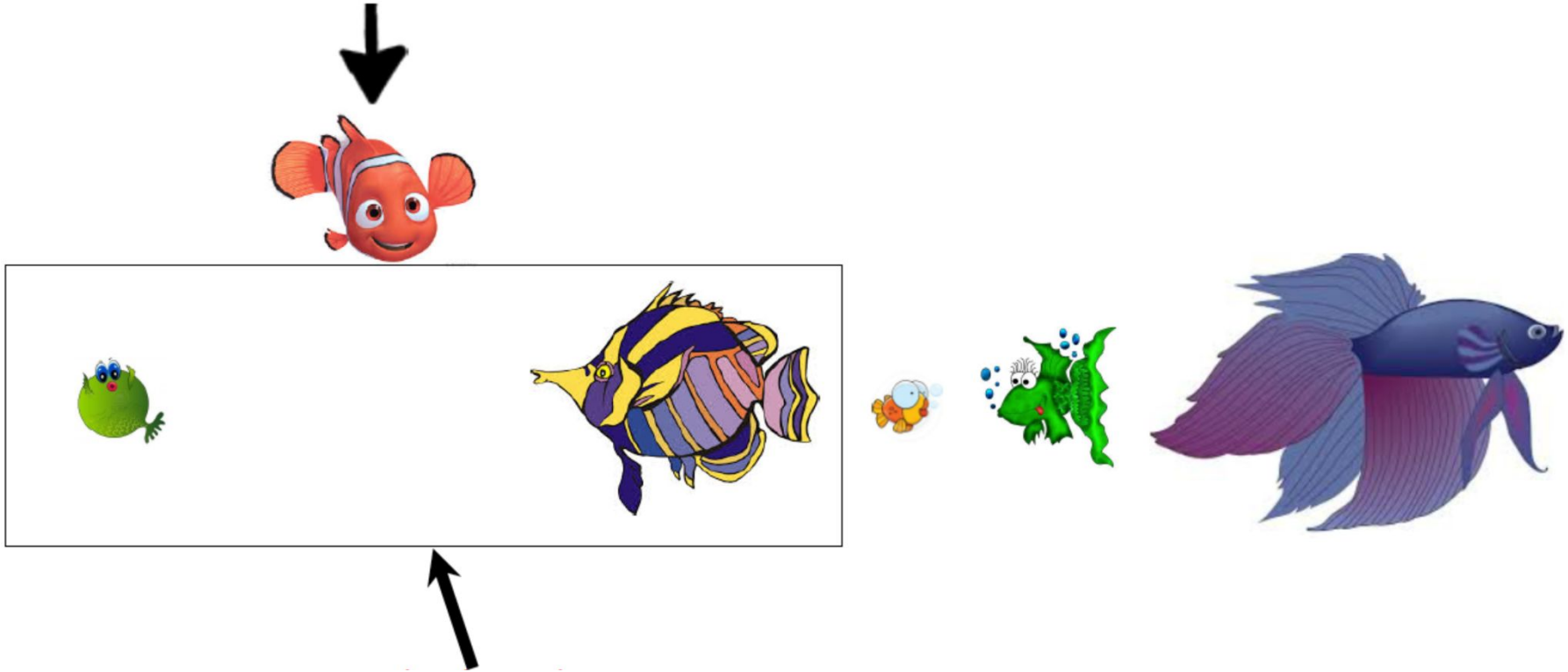


Insertion Sort: visualización



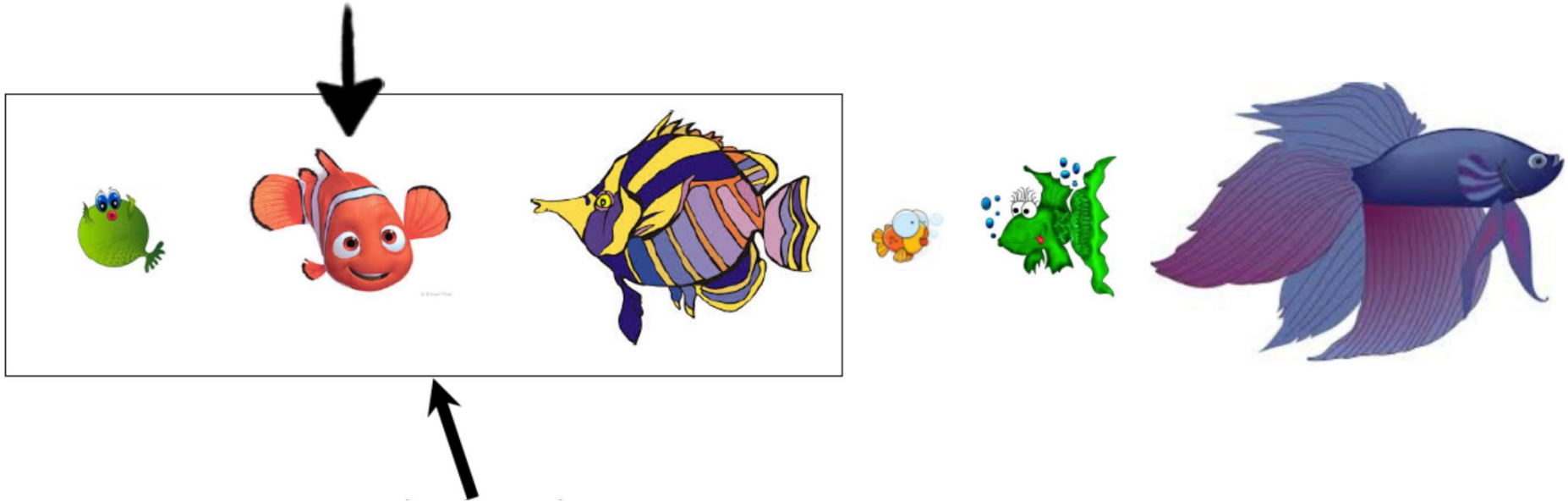
Parte ordenada

Insertion Sort: visualización



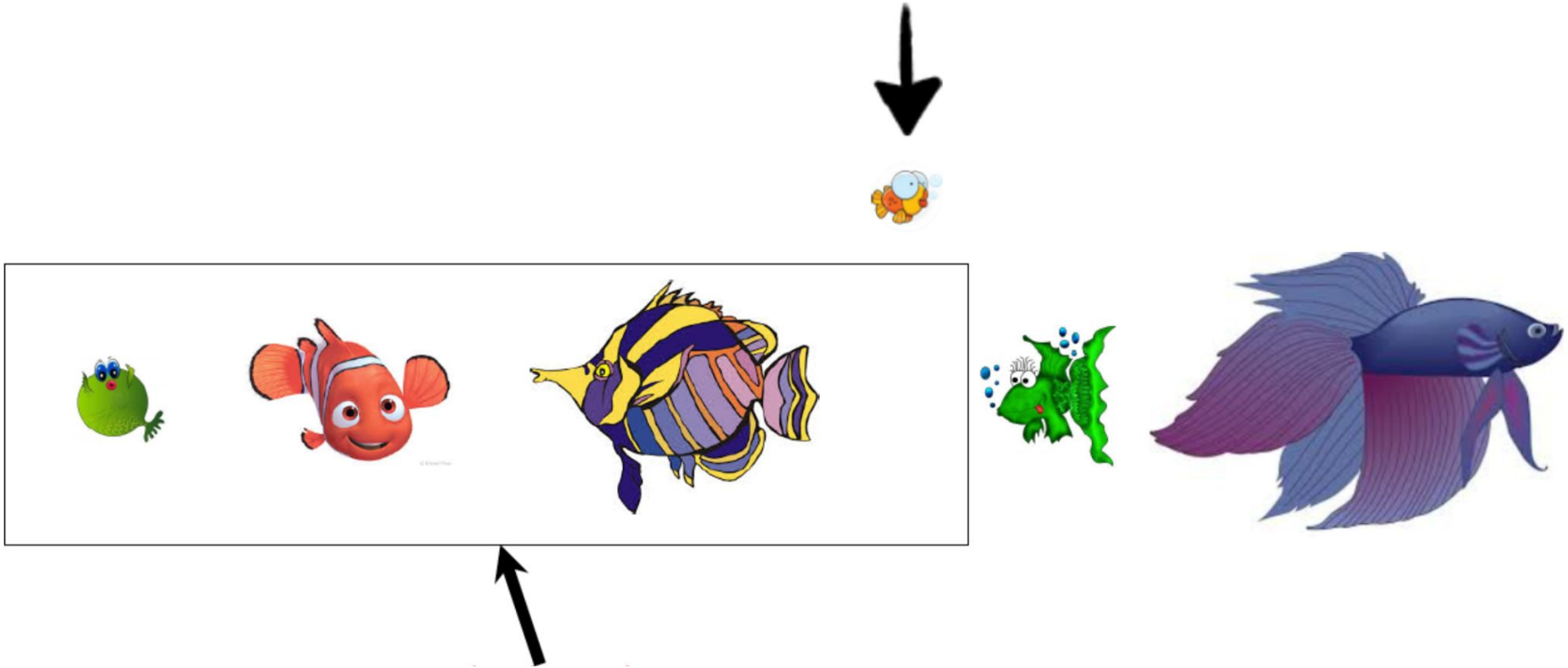
Parte ordenada

Insertion Sort: visualización



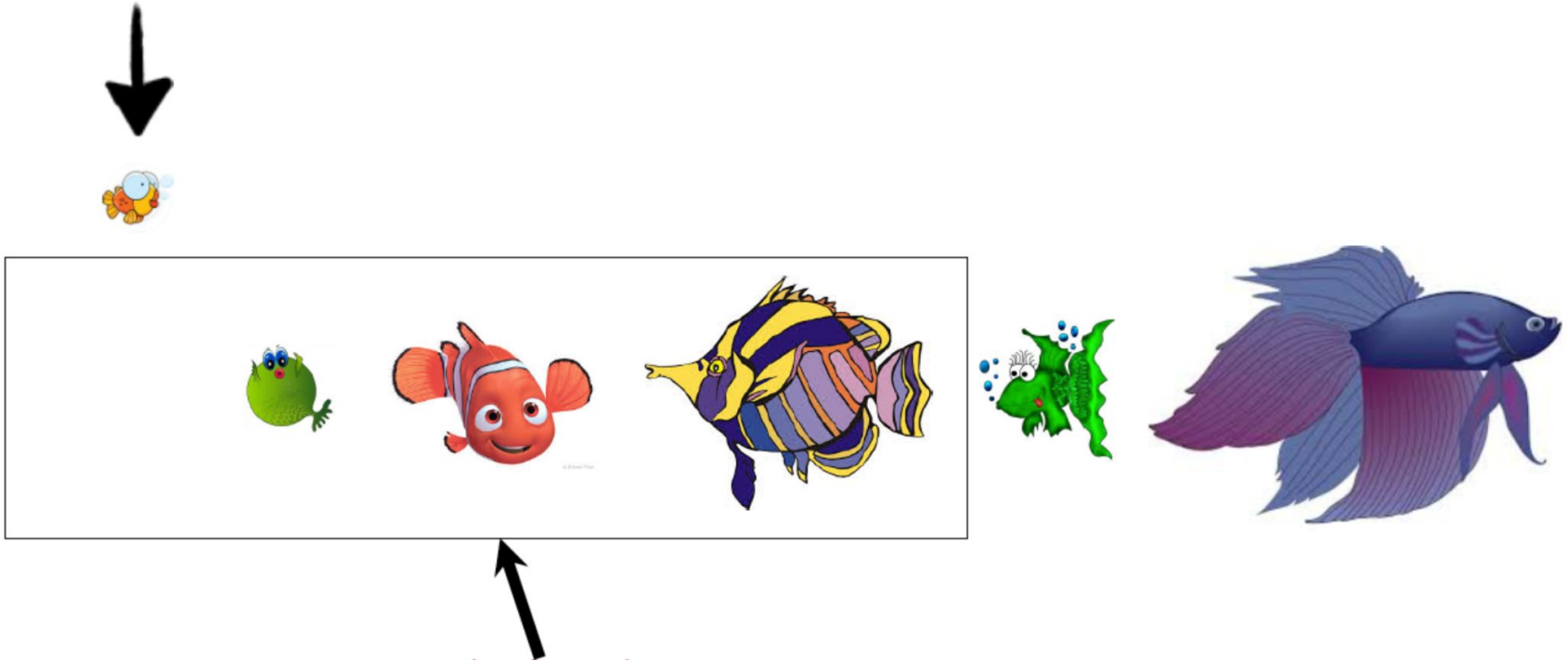
Parte ordenada

Insertion Sort: visualización



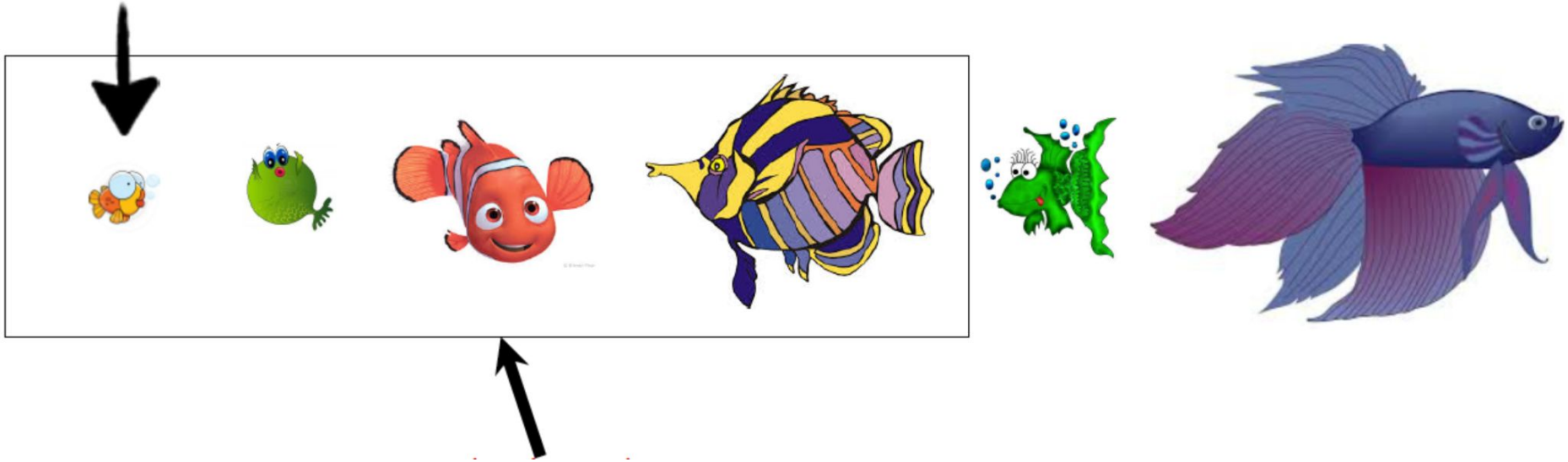
Parte ordenada

Insertion Sort: visualización



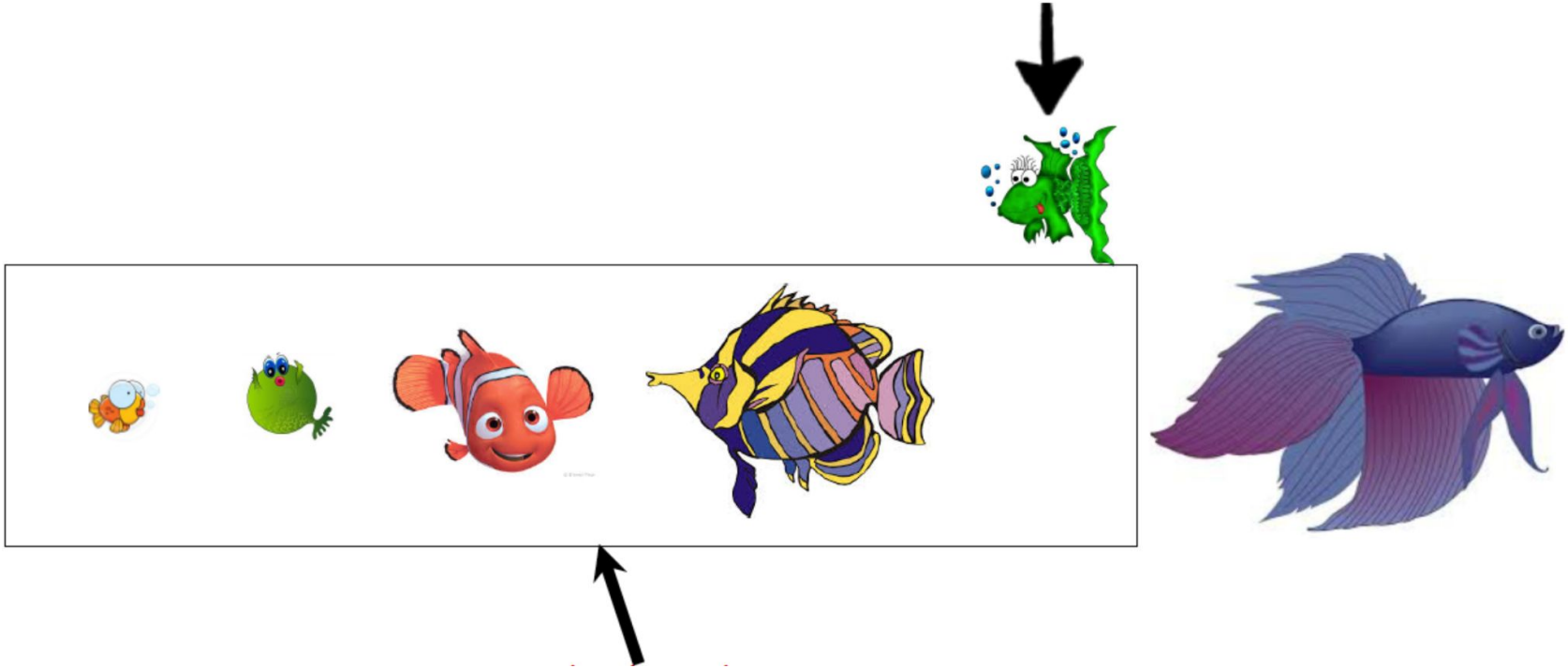
Parte ordenada

Insertion Sort: visualización



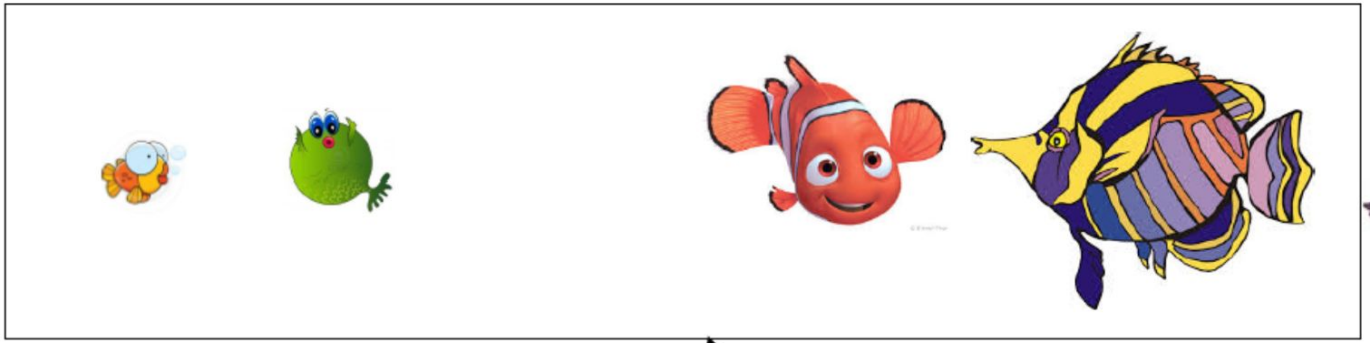
Parte ordenada

Insertion Sort: visualización



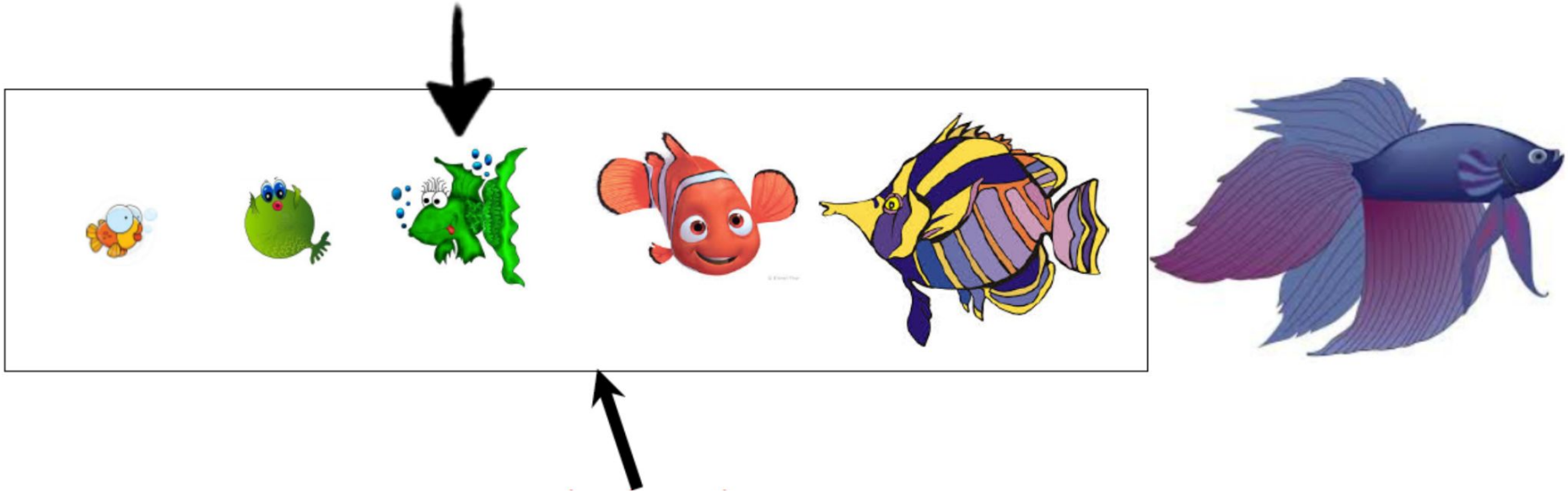
Parte ordenada

Insertion Sort: visualización



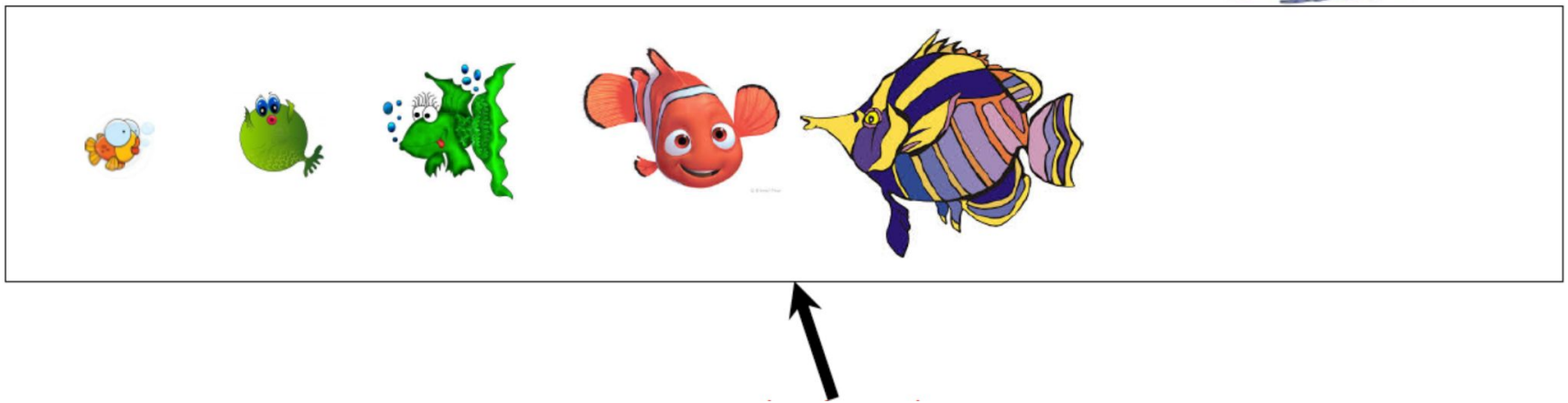
Parte ordenada

Insertion Sort: visualización



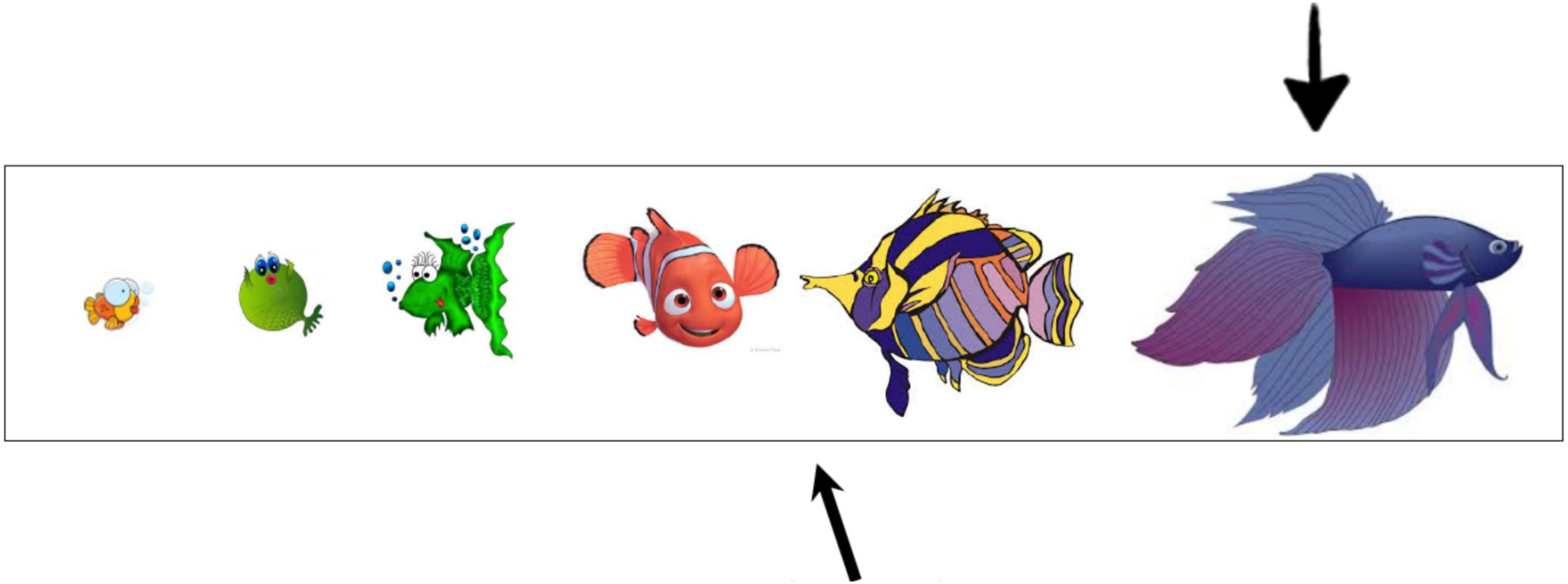
Parte ordenada

Insertion Sort: visualización



Parte ordenada

Insertion Sort: visualización



Parte ordenada

Insertion Sort: idea

Si los primeros elementos están ordenados, podemos fácilmente insertar un elemento en el lugar correspondiente en la parte ordenada. En cada iteración i, hay dos subarrays:

- El subarray ordenado X[0…i-1]
- El subarray desordenado X[i…n-1]

Construimos incrementalmente la parte ordenada, eligiendo el primer valor de la parte desordenada, X[i], e insertándolo en la parte ordenada X[0…i-1]. En cada iteración, el tamaño de la parte ordenada crece en 1, y la parte desordenada se decrementa en 1.

Insertion Sort: pseudocódigo

INSERTION-SORT(A)

```
for j=1 to length(A)-1
        key = A[j]
        // Insertar A[j] en la secuencia ordenada A[0.. j-1]
        i = j-1
        while i >= 0 and A[i] > key
             A[i+1] = A[i]
             i = i-1
        A[i+1] = key
```

Insertion Sort: análisis

Corremos 2 ciclos anidados, pero el ciclo interno se corta cuando el A[i] es menor que el elemento a insertar.

Mejor caso: Array ordenado en orden creciente. El ciclo externo itera (n-1) veces, y el ciclo interno sólo 1 vez, ya que cada vez que entramos al ciclo interno, A[i] es menor o igual que el elemento a insertar. Complejidad temporal: O(n)

Peor caso: Array ordenado en orden decreciente. El ciclo externo iterará n-1 veces, y en cada iteración del ciclo externo, el ciclo interno itera i veces. Complejidad temporal:$O(n^2)$

Performance de los algoritmos de ordenamiento

Algoritmo	Complejidad temporal	Complejidad espacial	Notas
Bubble Sort	N^2	1	
Selection Sort	N^2	1	
Insertion Sort	entre N y N^2	1	Depende del orden de los elementos
Mergesort	N log N	N	
Quicksort	N log N	lg N	
…	…	…	…