UNIT 4A
Iteration: Searching

Last Course

- Algorithms
- Sieve of Eratosthenes

Function calls and parameters

```
def getSumOf (numList, start, end):
    sum = 0
    for pos in ranges(start, end):
        sum = sum + numList[pos]
    return sum
```

```
def sumOf (a, b, c, d):
    sum = a + b + c + d
    return sum
```

```
>>> start = "This is a program to ...  "
>>> numbers = [3, 6, 8, 2, 5, 7]
>>> sum = getSumOf(numbers, 1, 5)
>>> sum = sumOf (3, 6, 8, 2)
```

Are these sum variables same?

What Does Your Code Say About You?

```
def linsearch(items, key):
    # search for key in items
    ln = len(items)
    i = 0
    while i < ln:
        if items[i] == key:
            return i
        i = i + 1
    return None
```

What Does Your Code Say About You?

```
def linsearch  (fred   ,x):
    ln =len(fred)

    v= 0
    while  v<ln:
      if fred[v  ]==x:
      return v
     v= v+   1
    return None
```

Grading on Code Formatting

- Grading on the appearance of your code will happen.
- Meaningful variable names, consistent spacing, explanatory comments (#), no gratuitous blank lines. (But in long functions, blank lines can be a good way to group code into sections.)
- Why are we doing this?
  – Because we're mean.
  – Because you cannot find the bugs in your code if you cannot read it properly.

Goals of this Unit

- Understand simple mechanical **searching** and **sorting** procedures

- Analyze how the **time consumed** scales as the **amount of data** grows

- Understand how **characters** (letters, digits, etc.) are encoded **as numbers**

Specifically

- Algorithm: linear (sequential) search
- Algorithm: insertion sort
- Coding: Unicode

Outline for today

- Unicode
- Sequential (linear) searching
- Thinking about efficiency
  - analyzing
  - measuring runtime

Strings and Unicode

- You can use relational operators to compare strings: <, <=, >, >=, ==, !=
- How can that be?
  Characters are coded as numbers.
- Strings of characters are coded as sequences of numbers
- Sequences are compared using rules of alphabetical order ("lexicographical order")

String comparisons

```
>>> 'A' < 'a'

>>> '1' < 'A'

>>> '1' < '2'

>>> '11' < '2'

>>>
```

```
>>> '12' < '112'

>>> 'abc' < 'b'

>>> 'alpha' < 'alphabet'

>>> 'awkward' < 'able'

>>>
```

String comparisons

```
>>> 'A' < 'a’
True

>>> '1' < 'A'
True

>>> '1' < '2'
True

>>> '11' < '2'
True

>>>
```

```
>>> '12' < '112'
False

>>> 'abc' < 'b'
True

>>> 'alpha' < 'alphabet'
True

>>> 'awkward' < 'able'
False

>>>
```

Unicode

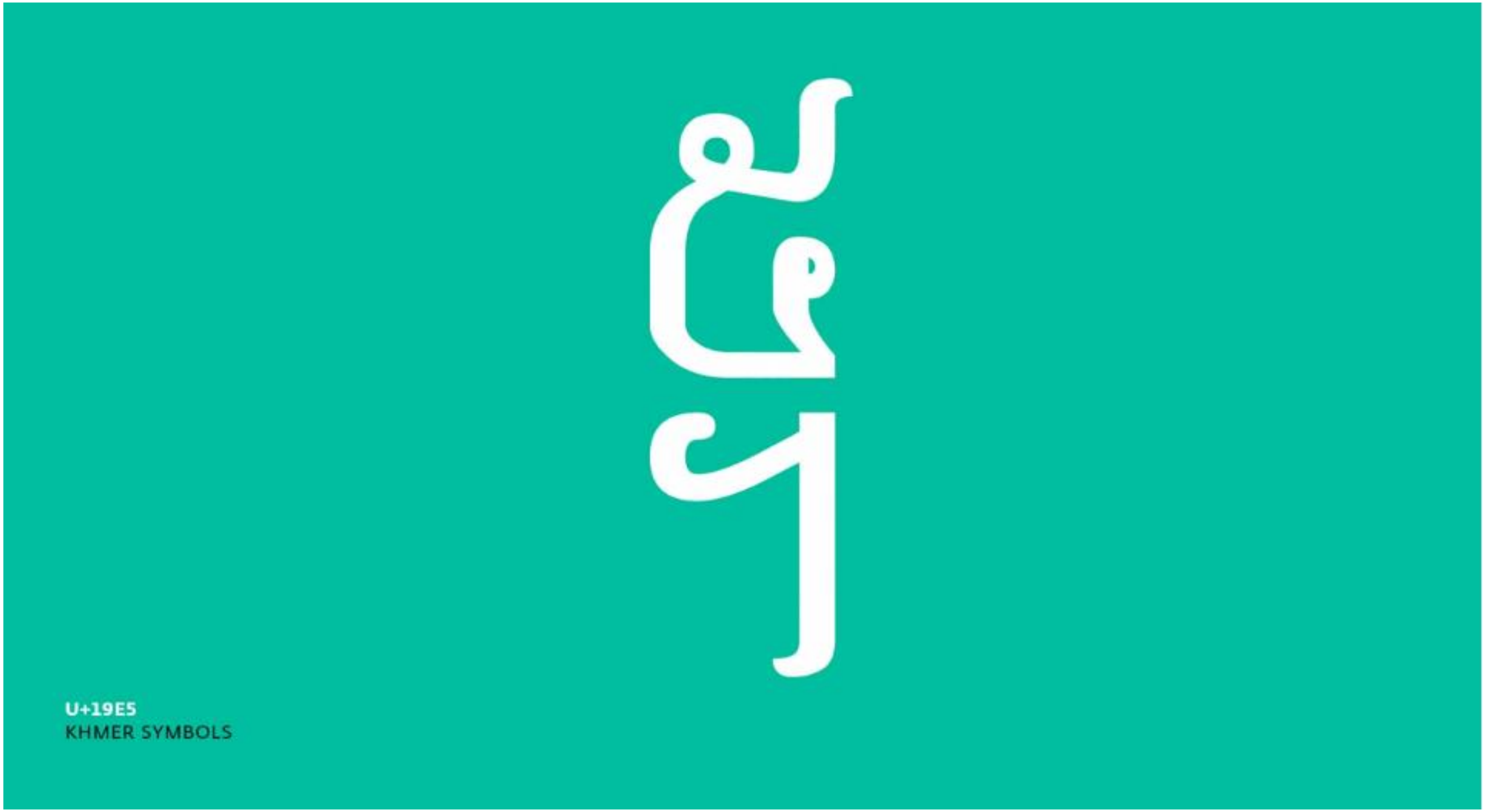
- Codes 48…57: digits 0 through 9
- Codes 65…91: A through Z
- Codes 97…122: a through z
- Other numbers: various special characters

Roman alphabet

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20	Space	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	DEL

…but many others!

	1F0	1F1	1F2	1F3	1F4	1F5	1F6	1F7	1F8	1F9	1FA	1FB	1FC	1FD	1FE	1FF
0	ἀ 1F00	ἐ 1F10	ἠ 1F20	ἰ 1F30	ὀ 1F40	ὐ 1F50	ὠ 1F60	ὰ 1F70	ᾀ 1F80	ᾐ 1F90	ᾠ 1FA0	ᾰ 1FB0	῀ 1FC0	ῐ 1FD0	ῠ 1FE0	
1	ἁ 1F01	ἑ 1F11	ἡ 1F21	ἱ 1F31	ὁ 1F41	ὑ 1F51	ὡ 1F61	ά 1F71	ᾁ 1F81	ᾑ 1F91	ᾡ 1FA1	ᾱ 1FB1	῁ 1FC1	ῑ 1FD1	ῡ 1FE1	
2	ἂ 1F02	ἒ 1F12	ἢ 1F22	ἲ 1F32	ὂ 1F42	ὒ 1F52	ὢ 1F62	ὲ 1F72	ᾂ 1F82	ᾒ 1F92	ᾢ 1FA2	ᾲ 1FB2	ῂ 1FC2	ῒ 1FD2	ῢ 1FE2	ῲ 1FF2
3	ἃ 1F03	ἓ 1F13	ἣ 1F23	ἳ 1F33	ὃ 1F43	ὓ 1F53	ὣ 1F63	έ 1F73	ᾃ 1F83	ᾓ 1F93	ᾣ 1FA3	ᾳ 1FB3	ῃ 1FC3	ΐ 1FD3	ΰ 1FE3	ῳ 1FF3
4	ἄ 1F04	ἔ 1F14	ἤ 1F24	ἴ 1F34	ὄ 1F44	ὔ 1F54	ὤ 1F64	ὴ 1F74	ᾄ 1F84	ᾔ 1F94	ᾤ 1FA4	ᾴ 1FB4	ῄ 1FC4		ῤ 1FE4	ῴ 1FF4
5	ἅ 1F05	ἕ 1F15	ἥ 1F25	ἵ 1F35	ὅ 1F45	ὕ 1F55	ὥ 1F65	ή 1F75	ᾅ 1F85	ᾕ 1F95	ᾥ 1FA5				ῥ 1FE5	
6	ἆ 1F06		ἦ 1F26	ἶ 1F36		ὖ 1F56	ὦ 1F66	ὶ 1F76	ᾆ 1F86	ᾖ 1F96	ᾦ 1FA6	ᾶ 1FB6	ῆ 1FC6	ῖ 1FD6	ῦ 1FE6	ῶ 1FF6
7	ἇ 1F07		ἧ 1F27	ἷ 1F37		ὗ 1F57	ὧ 1F67	ί 1F77	ᾇ 1F87	ᾗ 1F97	ᾧ 1FA7	ᾷ 1FB7	ῇ 1FC7	ῗ 1FD7	ῧ 1FE7	ῷ 1FF7
8	Ἀ 1F08	Ἐ 1F18	Ἠ 1F28	Ἰ 1F38	Ὀ 1F48		Ὠ 1F68	ὸ 1F78	ᾈ 1F88	ᾘ 1F98	ᾨ 1FA8	Ᾰ 1FB8	Ὲ 1FC8	Ῐ 1FD8	Ῠ 1FE8	Ὸ 1FF8
9	Ἁ 1F09	Ἑ 1F19	Ἡ 1F29	Ἱ 1F39	Ὁ 1F49	Ὑ 1F59	Ὡ 1F69	ό 1F79	ᾉ 1F89	ᾙ 1F99	ᾩ 1FA9	Ᾱ 1FB9	Έ 1FC9	Ῑ 1FD9	Ῡ 1FE9	Ό 1FF9
A	Ἂ 1F0A	Ἒ 1F1A	Ἢ 1F2A	Ἲ 1F3A	Ὂ 1F4A		Ὢ 1F6A	ὺ 1F7A	ᾊ 1F8A	ᾚ 1F9A	ᾪ 1FAA	Ὰ 1FBA	Ὴ 1FCA	Ὶ 1FDA	Ὺ 1FEA	Ὼ 1FFA
B	Ἃ 1F0B	Ἓ 1F1B	Ἣ 1F2B	Ἳ 1F3B	Ὃ 1F4B	Ὓ 1F5B	Ὣ 1F6B	ύ 1F7B	ᾋ 1F8B	ᾛ 1F9B	ᾫ 1FAB	Ά 1FBB	Ή 1FCB	Ί 1FDB	Ύ 1FEB	Ώ 1FFB
C	Ἄ 1F0C	Ἔ 1F1C	Ἤ 1F2C	Ἴ 1F3C	Ὄ 1F4C		Ὤ 1F6C	ὼ 1F7C	ᾌ 1F8C	ᾜ 1F9C	ᾬ 1FAC	ᾼ 1FBC	ῌ 1FCC		Ῥ 1FEC	ῼ 1FFC
D	Ἅ 1F0D	Ἕ 1F1D	Ἥ 1F2D	Ἵ 1F3D	Ὅ 1F4D	Ὕ 1F5D	Ὥ 1F6D	ώ 1F7D	ᾍ 1F8D	ᾝ 1F9D	ᾭ 1FAD	᾽ 1FBD	῍ 1FCD	῝ 1FDD	῭ 1FED	´ 1FFD
E	Ἆ 1F0E		Ἦ 1F2E	Ἶ 1F3E			Ὦ 1F6E		ᾎ 1F8E	ᾞ 1F9E	ᾮ 1FAE	ι 1FBE	῎ 1FCE	῞ 1FDE	΅ 1FEE	῾ 1FFE
F	Ἇ 1F0F		Ἧ 1F2F	Ἷ 1F3F		Ὗ 1F5F	Ὧ 1F6F		ᾏ 1F8F	ᾟ 1F9F	ᾯ 1FAF	᾿ 1FBF	῏ 1FCF	῟ 1FDF	` 1FEF	



a unicode video: http://vimeo.com/48858289 109, 242 characters/codes in 2 hours, 31 mintes, and 25 seconds
Amazingly, everything after around 14:00 seems to be (Chinese) ideographs!

more later on encodings, now

ONWARD TO SEARCH

Searching, we use it

Built-in Search in Python

```
movies = ["The Wolf of Wall Street", "American Hustle",
         "Frozen", "Her",  "Lone Survivor", "12 Years a
         Slave",  "Nosferatu”, "Arnacoeur", "Sullivan's
         Travels"]
```

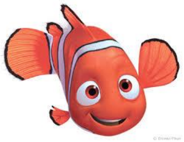
```
"American Hustle" in movies        →     True

"American" in movies               →     False

movies.index("Frozen")             →     2

movies.index("Lone")               →     ValueError: 'Lone'
                                         is not in list
```

Let’s Write Our Own Search

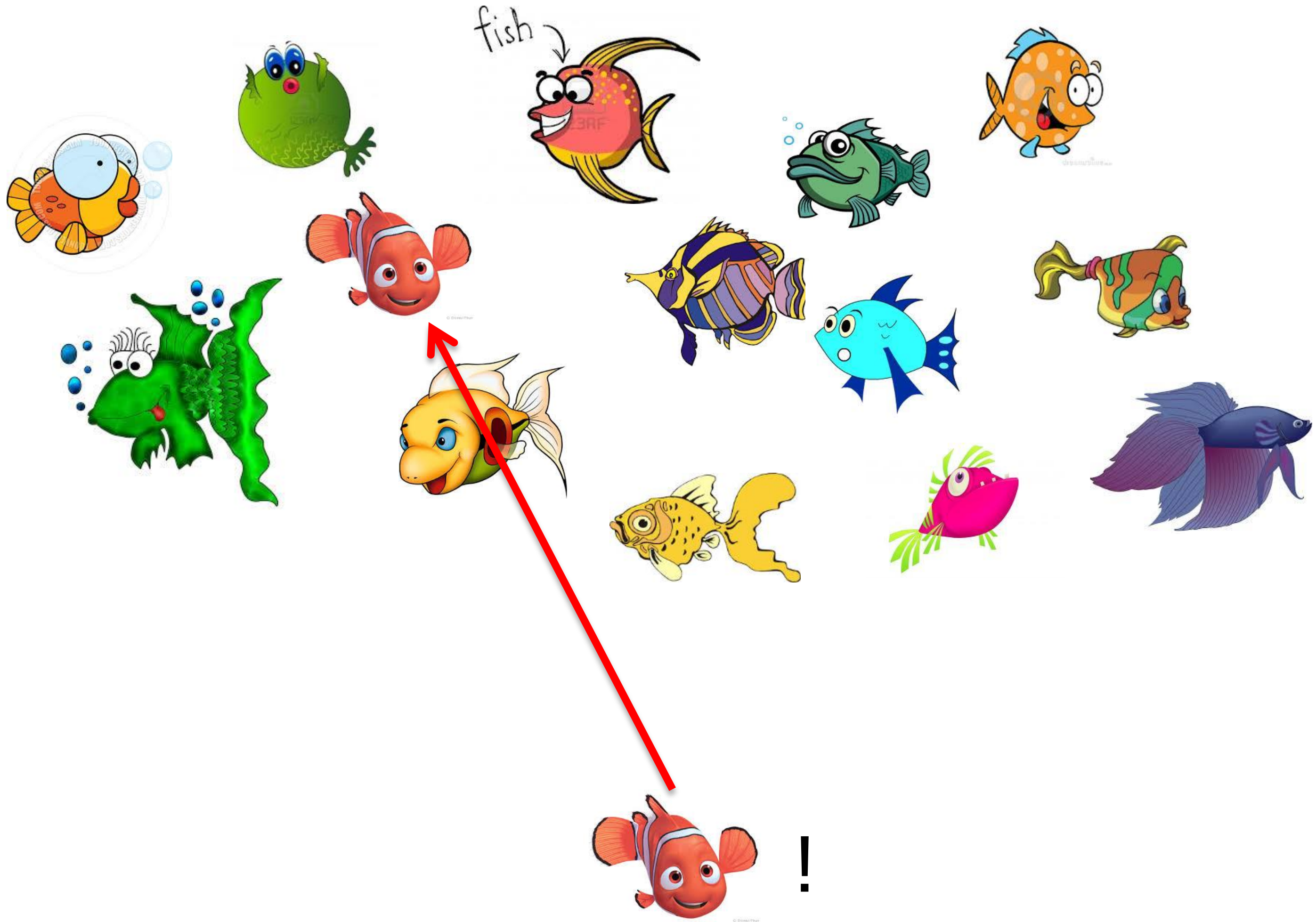
- Method: contains(items, key)
- Input: items to be searched
  (could be strings or numbers or …)
- Input: key to search for
- Output: True or False

- Approach: think linearly

Not thinking linearly…

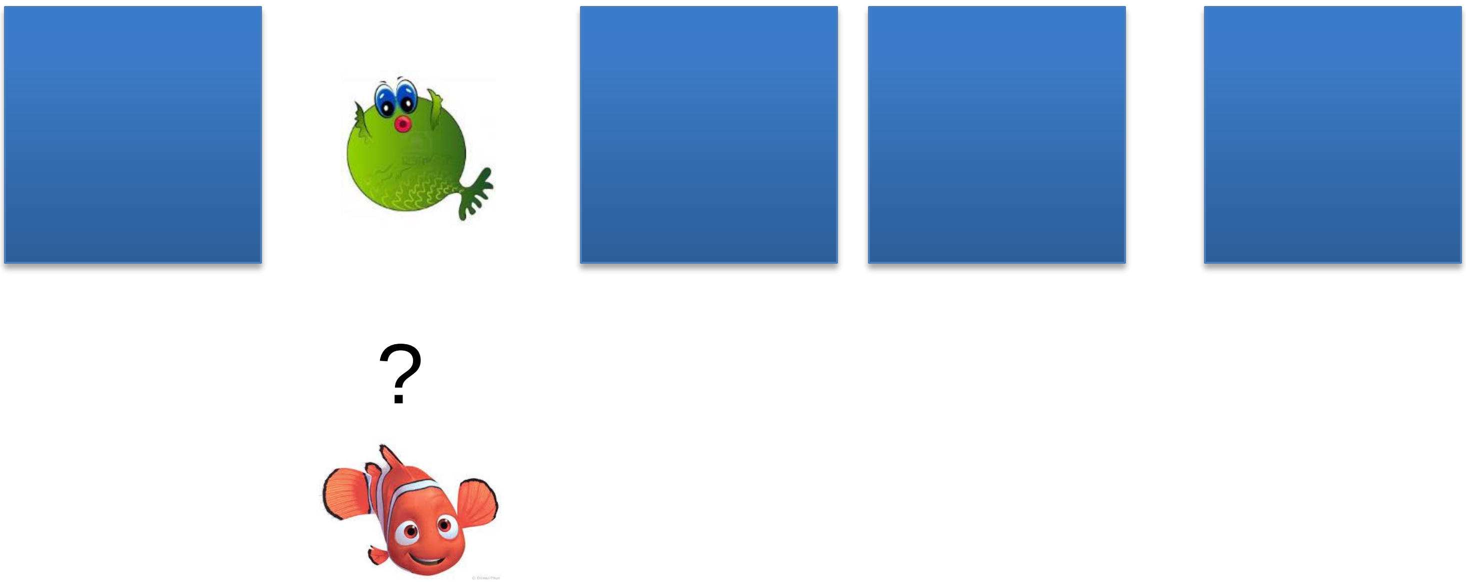
 ?

Not thinking linearly…

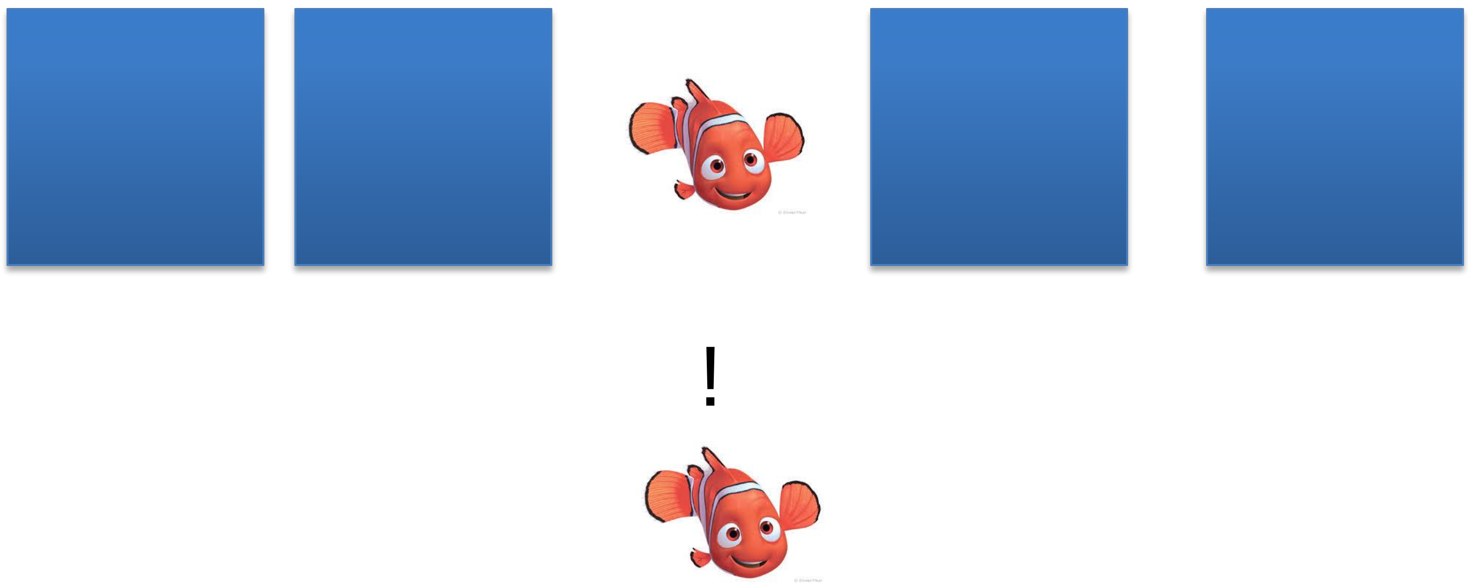


Thinking linearly…

Thinking linearly…



Thinking linearly…



A contains() method

```
def contains(items, key):
    for index in range(len(items)):
        if items[index] == key:
            return True
    return False
```

Another contains() method

```
def contains(items, key):
    for item in items:
        if item == key:
            return True
    return False
```

Getting More Information

- Method: **search(items, key)**
- Input: list to be searched

  (could be strings or numbers or …)
- Input: key to search for
- Output: **index of the first member** of the list that matches the key, or **None** if the key isn't in the list (instead of True or False)

Search using a for-loop

```
def search(items, key):
    for index in range(len(items)):
        if items[index] == key:
            return index
    return None
```

Alternatively?

```
def search(items, key):
    for item in items:
        if item == key:
            return index
    return None
```

Why can't we do this?

Ok, but...

```
def search(items, key):
    for item in items:
        if item == key:
            return items.index(key)
    return None
```

What's undesirable about this?

Be aware of the cost of the things Python does for you "behind the scenes"!

Problems, Algorithms and Programs

- One problem : potentially many algorithms
- One algorithm : potentially many programs
- We can compare how efficient different programs are both analytically and empirically

Analytically: Which One is Faster?

```
def contains1(items, key):

  index = 0
  while index < len(items):
    if items[index] == key:
      return True
    index = index + 1
  return False
```

```
def contains2(items, key):
  ln = len(items)
  index = 0
  while index < ln:
    if items[index] == key:
      return True
    index = index + 1
  return False
```

`len(items)` is executed each time loop condition is checked

`len(items)` is executed only once and its value is stored in `ln`

Is a for-loop faster than a while-loop?

Add the following function to our collection of contains functions from the previous page:

```
def contains3(items, key):
    for index in range(len(items)):
        if items[index] == key:
            return True
    return False
```

Empirical Measurement

- Three programs for the same algorithm; let's measure which is faster:

```
import time
def time1(items, key) :
    start = time.time()
    contains1(items, key)
    runtime = time.time() - start
    print("contains1:", runtime)
```

- Define `time2` and `time3` similarly to call `contains2` and `contains3`

Doing the Measurement

```
>>> items = [None]*1000000
>>> time1(items, 0)
contains1: 0.34292006492614746
>>> time2(items, 0)
contains2: 0.28722596168518066
>>> time3(items, 0)
contains3: 0.15350890159606934
>>>
```

while loop

while loop with saved length

for loop

Conclusion: using `for` and `range`() is faster than using `while` and addition, when doing an unsuccessful search. Whyyyyyyy?

A Different Measurement

- What if we want to know how the different loops perform when the key matches the first element?

```
>>> time1(items1, None)
contains1: 1.0013580322265625e-05
>>> time2(items1, None)
contains2: 1.0967254638671875e-05
>>> time3(items1, None)
contains3: 1.9073486328125e-05
```

while loop

while loop with saved length

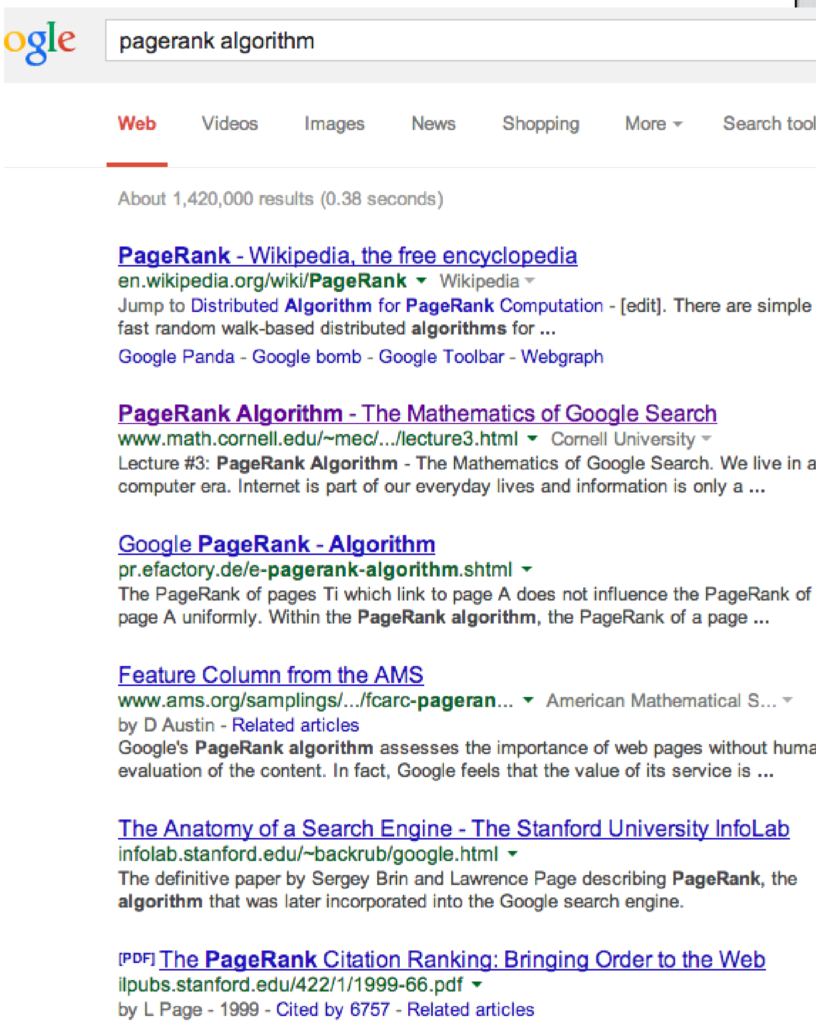
for loop

Now the relationship is reversed; contains1 is fastest! Whyyyyyyyyyyyy?

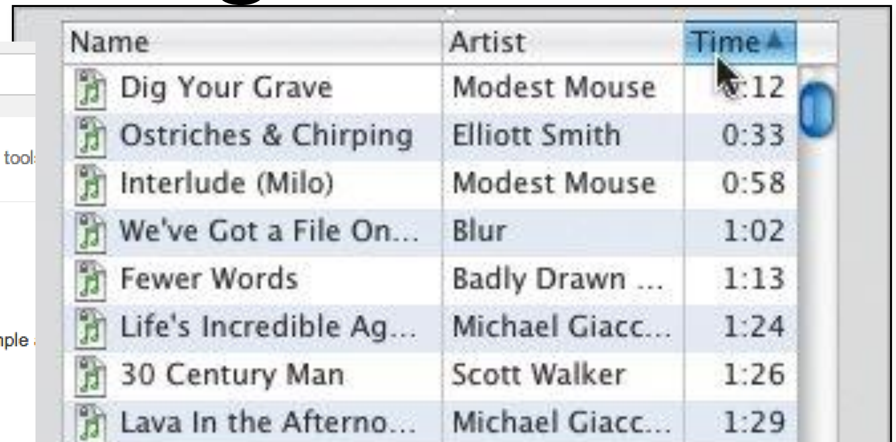
Now

SORTING

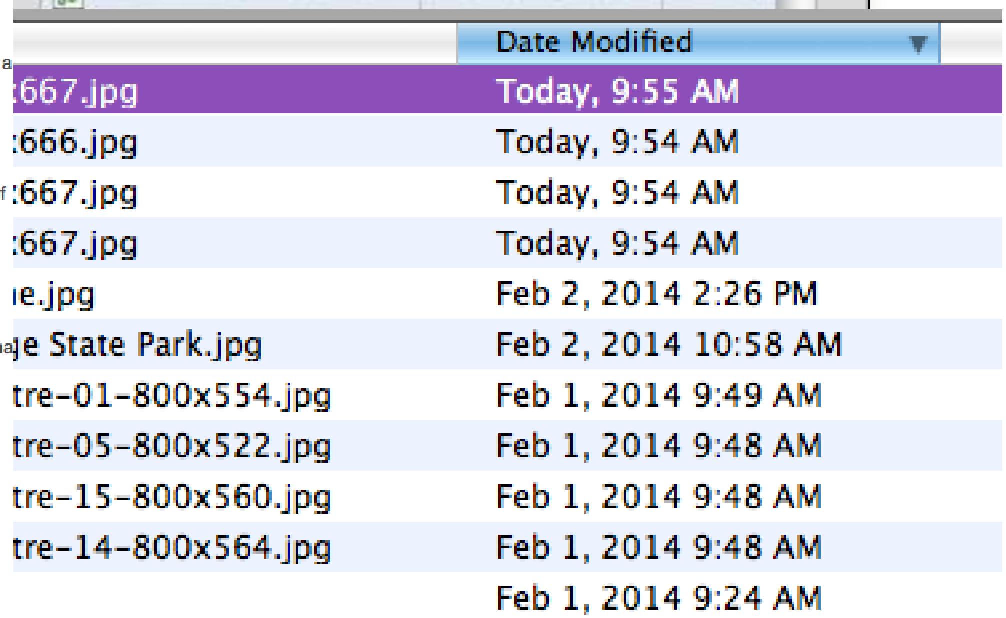
Sorting



ogle
pagerank algorithm
Web
Videos
Images
News
Shopping
More
Search tool
About 1,420,000 results (0.38 seconds)
PageRank - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/PageRank Wikipedia
Jump to Distributed Algorithm for PageRank Computation - [edit]. There are simple
fast random walk-based distributed algorithms for ...
Google Panda - Google bomb - Google Toolbar - Webgraph
PageRank Algorithm - The Mathematics of Google Search
www.math.cornell.edu/~mec/.../lecture3.html Cornell University
Lecture #3: PageRank Algorithm - The Mathematics of Google Search. We live in a
computer era. Internet is part of our everyday lives and information is only a ...
Google PageRank - Algorithm
pr.efactory.de/e-pagerank-algorithm.shtml
The PageRank of pages Ti which link to page A does not influence the PageRank of
page A uniformly. Within the PageRank algorithm, the PageRank of a page ...
Feature Column from the AMS
www.ams.org/samplings/.../fcarc-pageran... American Mathematical S...
by D Austin - Related articles
Google's PageRank algorithm assesses the importance of web pages without huma
evaluation of the content. In fact, Google feels that the value of its service is ...
The Anatomy of a Search Engine - The Stanford University InfoLab
infolab.stanford.edu/~backrub/google.html
The definitive paper by Sergey Brin and Lawrence Page describing PageRank, the
algorithm that was later incorporated into the Google search engine.
[PDF] The PageRank Citation Ranking: Bringing Order to the Web
ilpubs.stanford.edu/422/1/1999-66.pdf
by L Page - 1999 - Cited by 6757 - Related articles

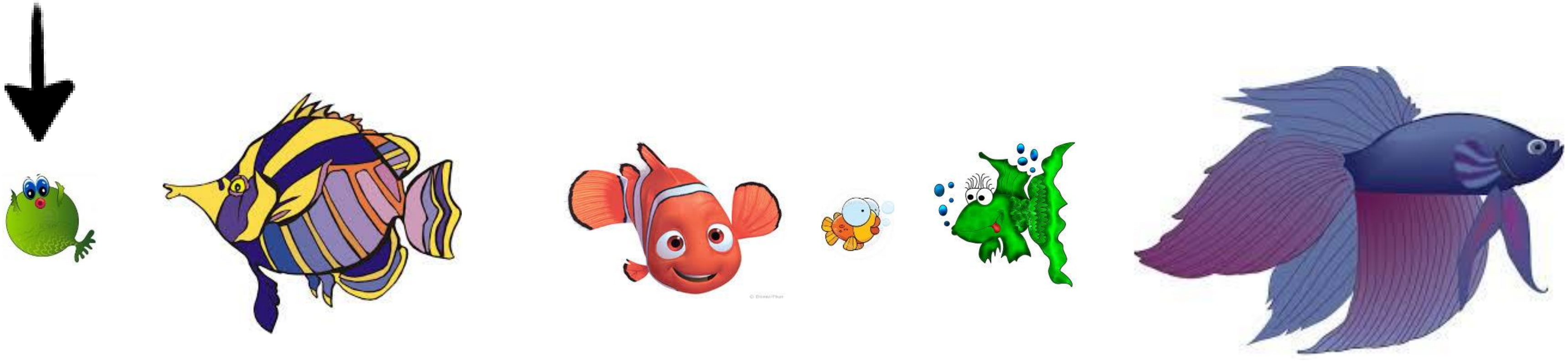


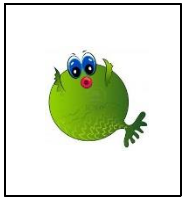
Name
Artist
Time
Dig Your Grave
Modest Mouse
Ostriches & Chirping
Elliott Smith
0:33
Interlude (Milo)
Modest Mouse
0:58
We've Got a File On...
Blur
1:02
Fewer Words
Badly Drawn ...
1:13
Life's Incredible Ag...
Michael Giacc...
1:24
30 Century Man
Scott Walker
1:26
Lava In the Afterno...
Michael Giacc...
1:29



Date Modified
667.jpg
Today, 9:55 AM
666.jpg
Today, 9:54 AM
667.jpg
Today, 9:54 AM
667.jpg
Today, 9:54 AM
e.jpg
Feb 2, 2014 2:26 PM
e State Park.jpg
Feb 2, 2014 10:58 AM
tre-01-800x554.jpg
Feb 1, 2014 9:49 AM
tre-05-800x522.jpg
Feb 1, 2014 9:48 AM
tre-15-800x560.jpg
Feb 1, 2014 9:48 AM
tre-14-800x564.jpg
Feb 1, 2014 9:48 AM
Feb 1, 2014 9:24 AM

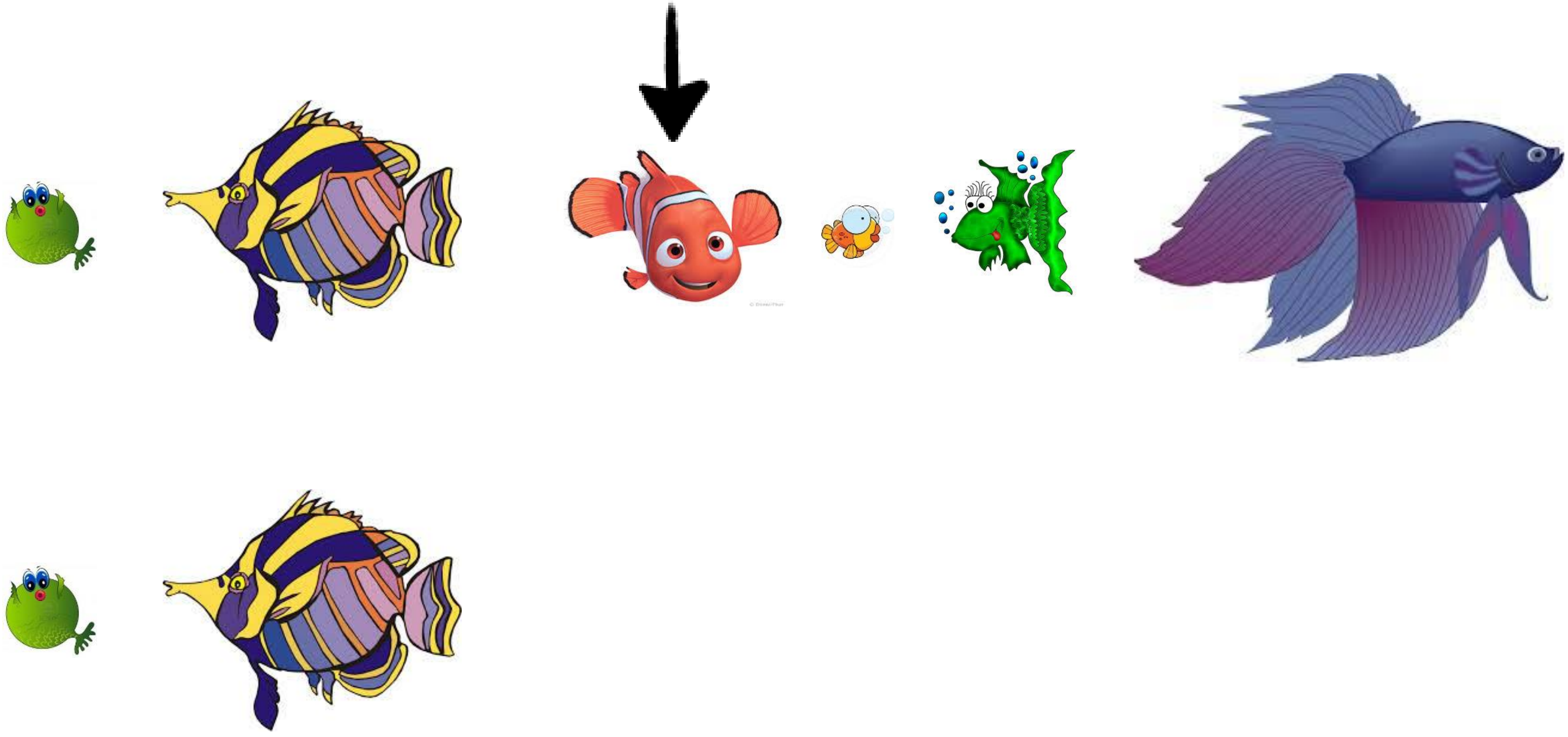
Insertion Sort



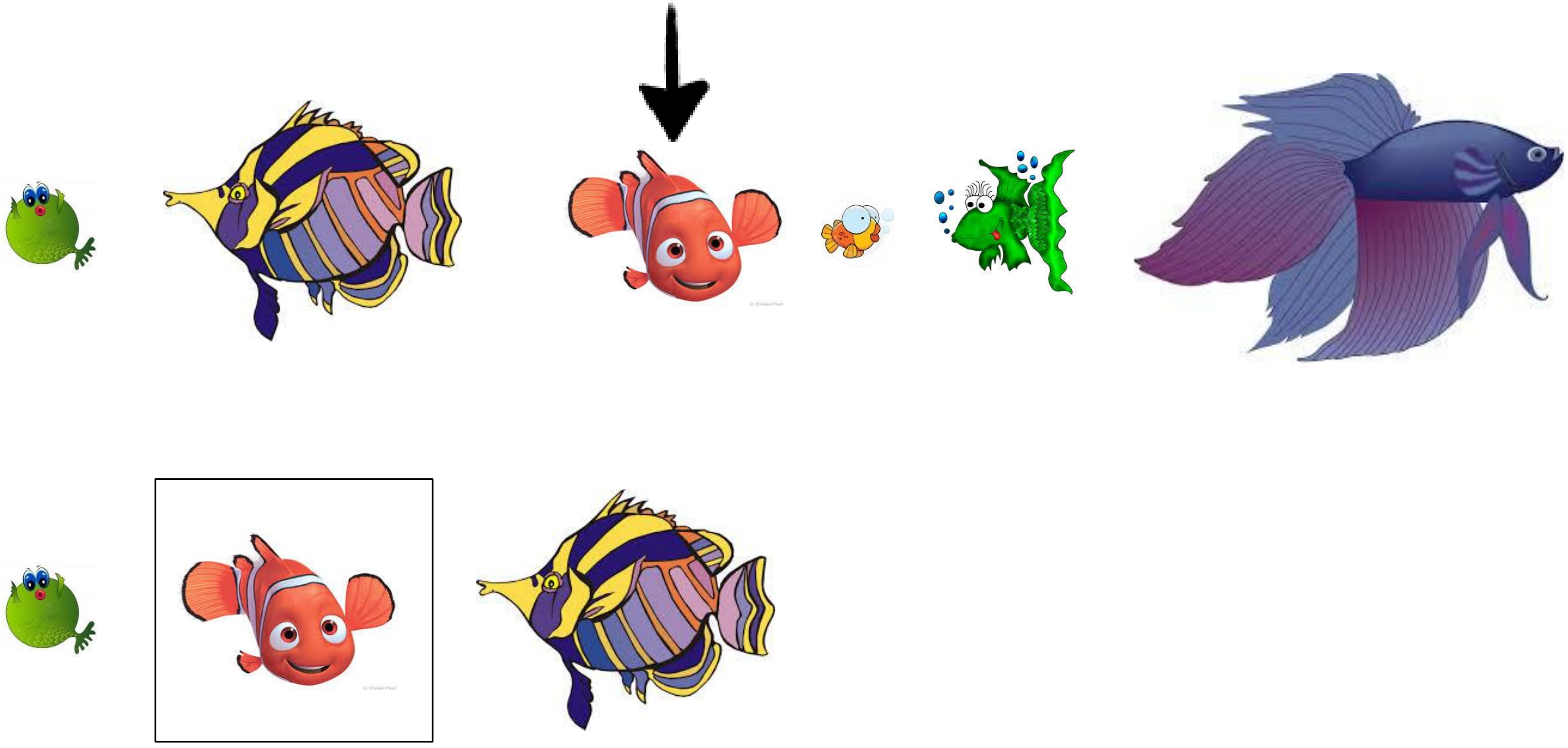


Insertion Sort

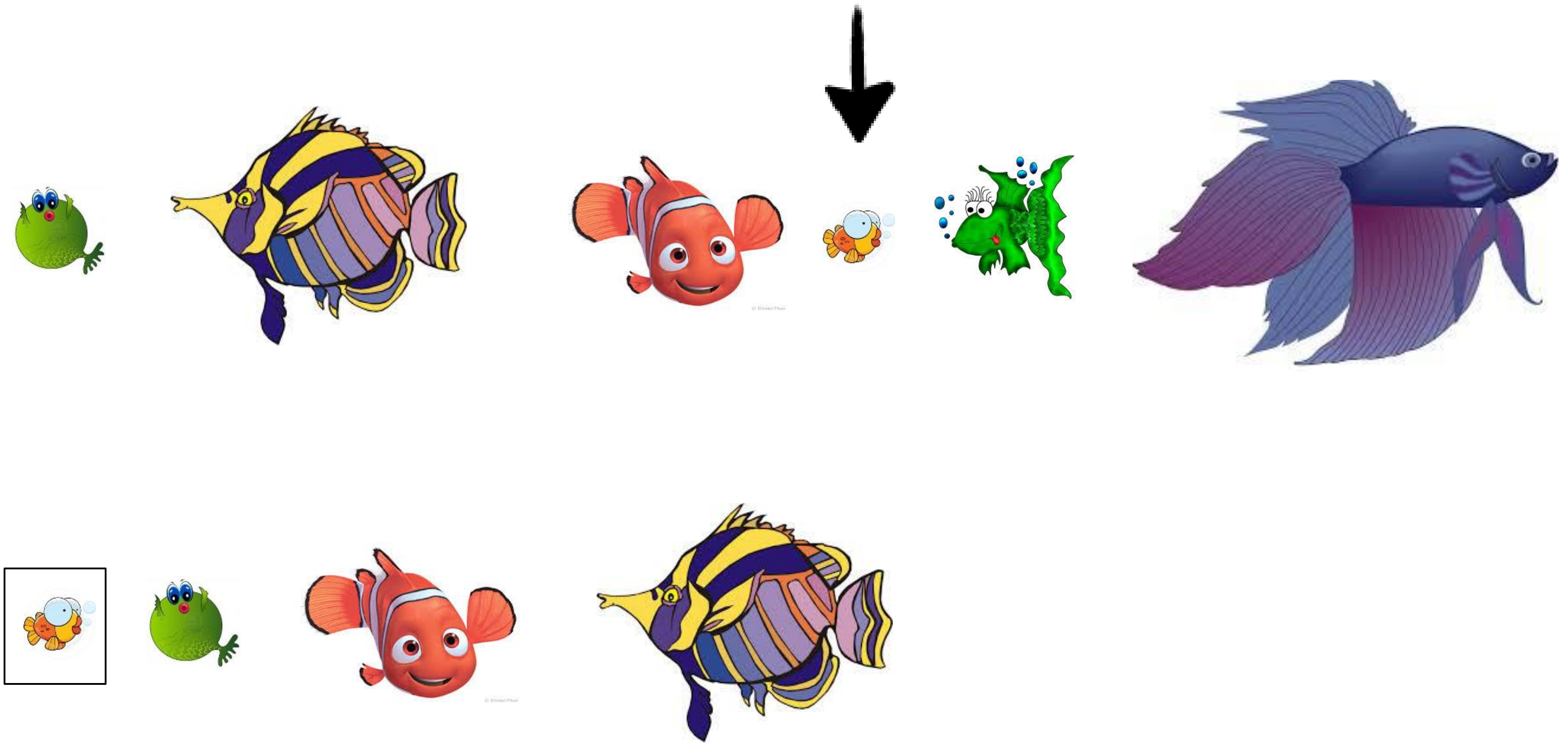
Insertion Sort



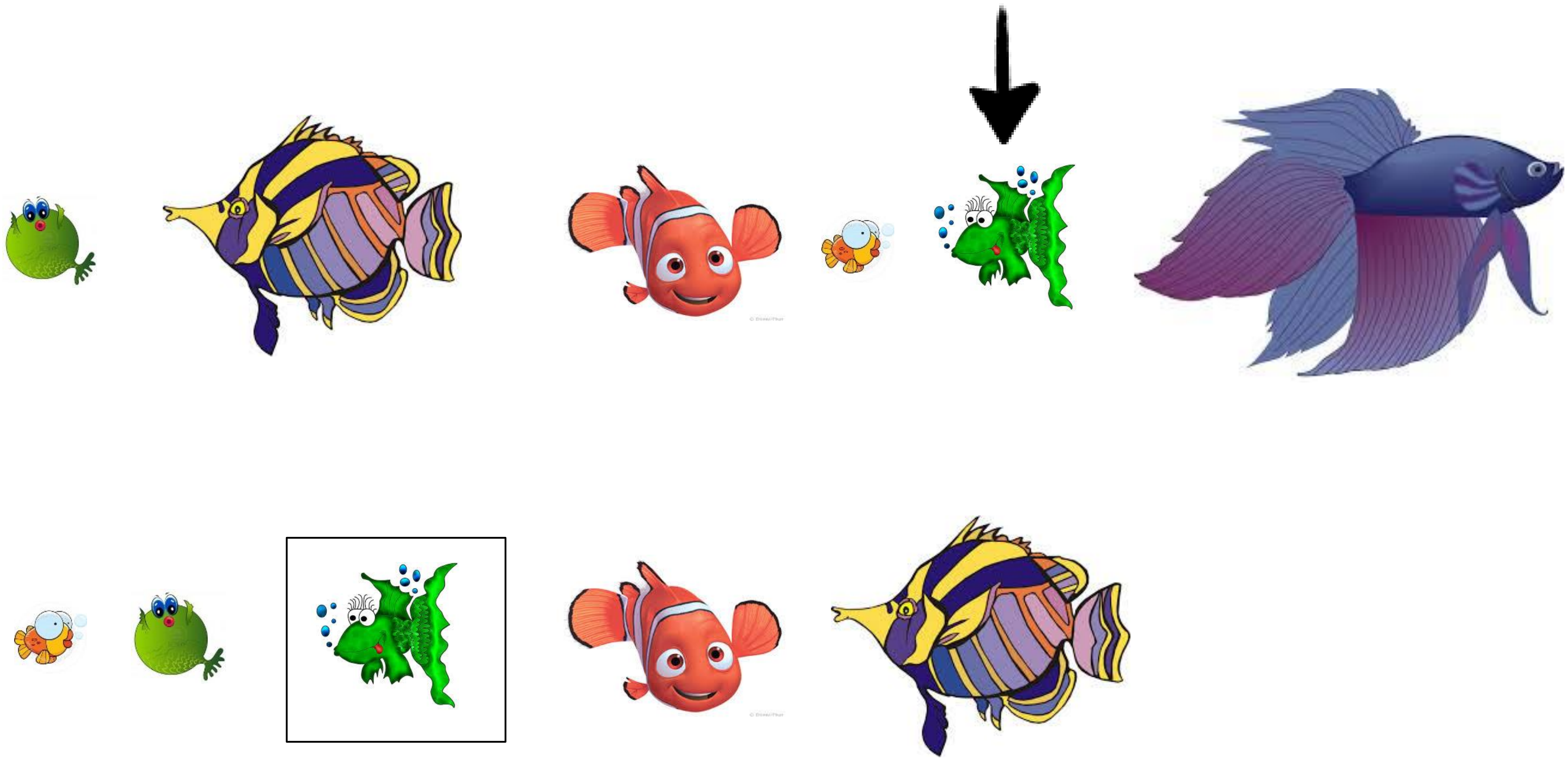
Insertion Sort



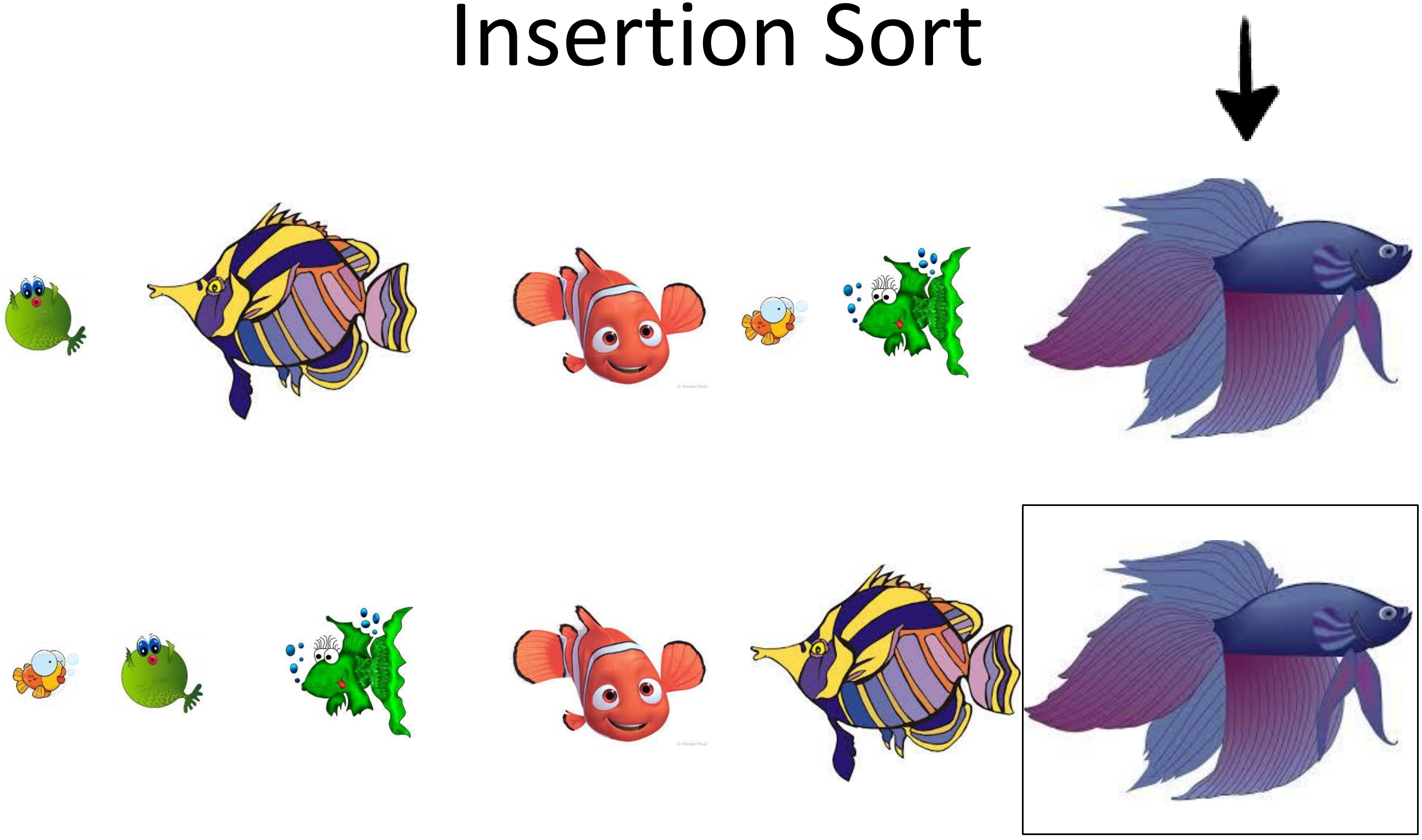
Insertion Sort



Insertion Sort



Insertion Sort

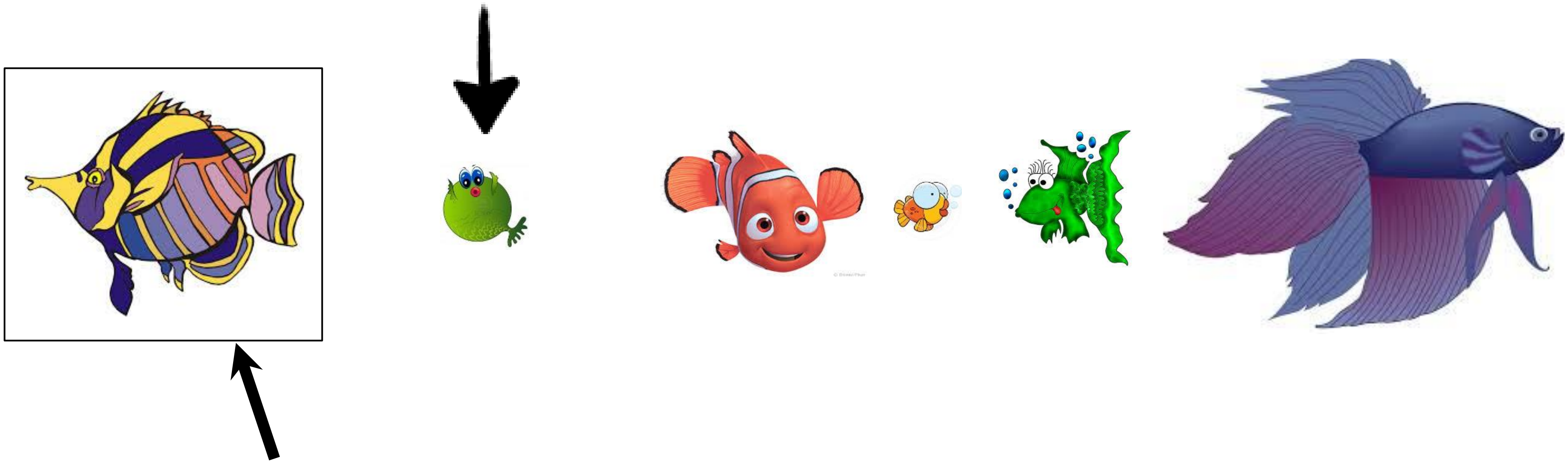


In-place Insertion Sort

- **Idea:** during sorting, a **prefix** of the list is *already sorted.* (This prefix might contain one, two, or more elements.)
- Each element that we process is inserted into the correct place in the sorted prefix of the list.
- Result: sorted part of the list gets bigger until the whole thing is sorted.

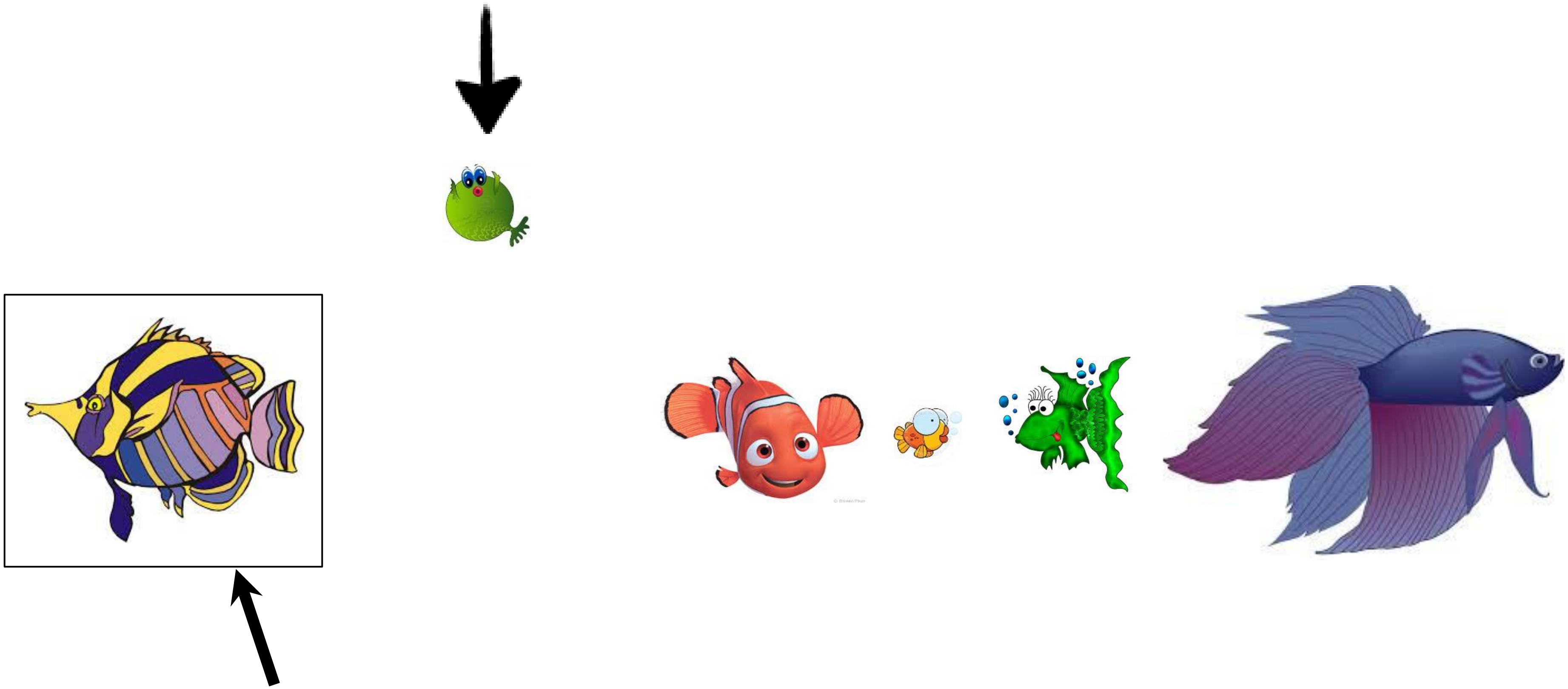
67

In-place Insertion Sort



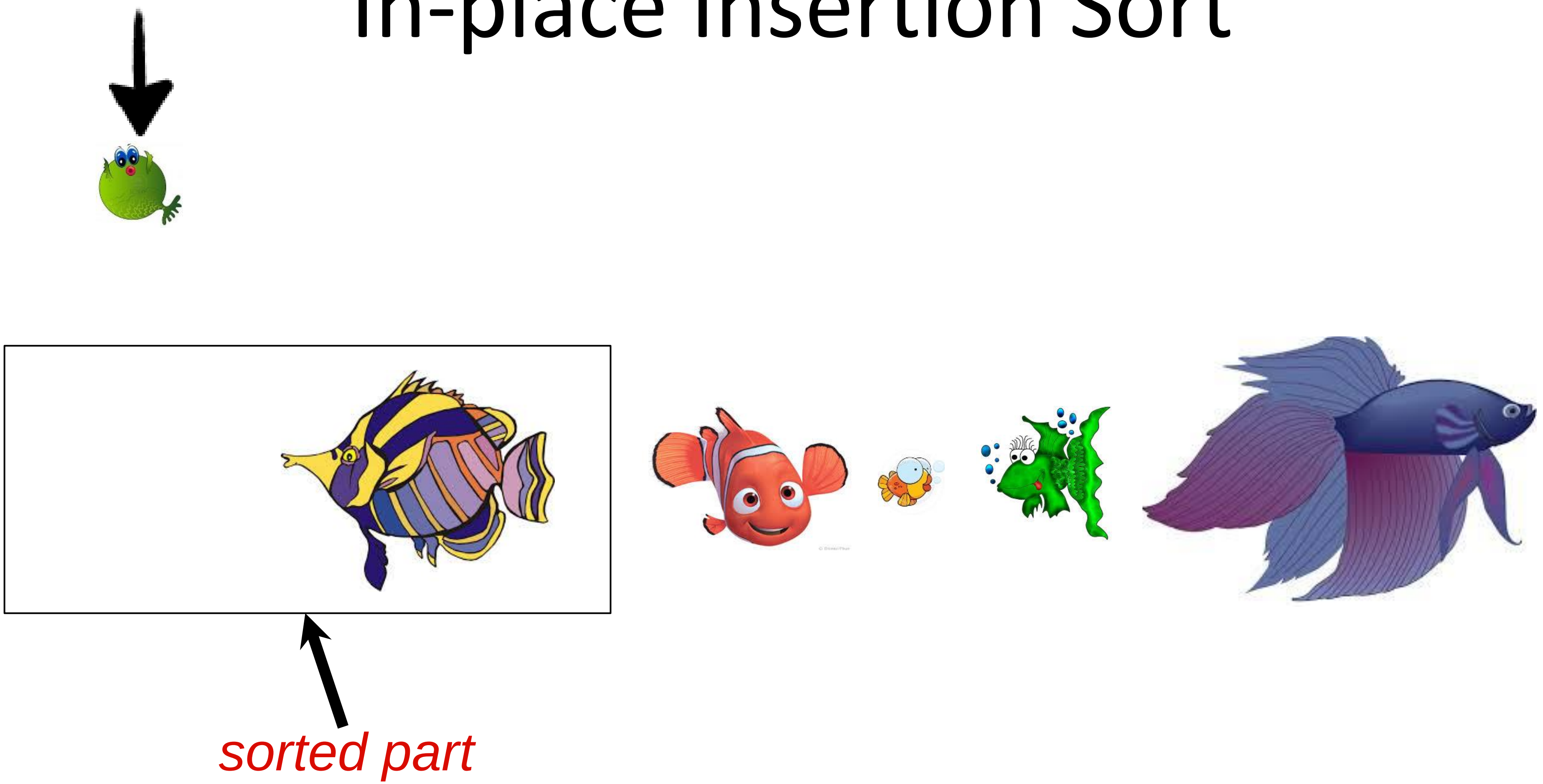
sorted part

In-place Insertion Sort

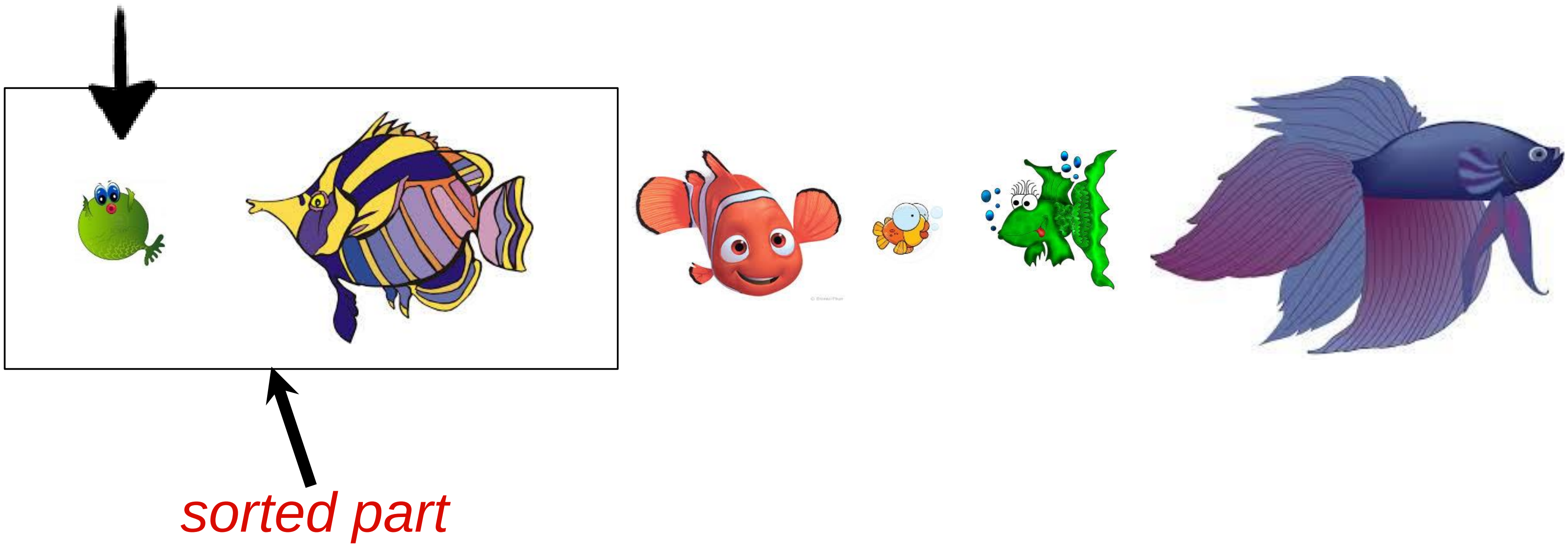


sorted part

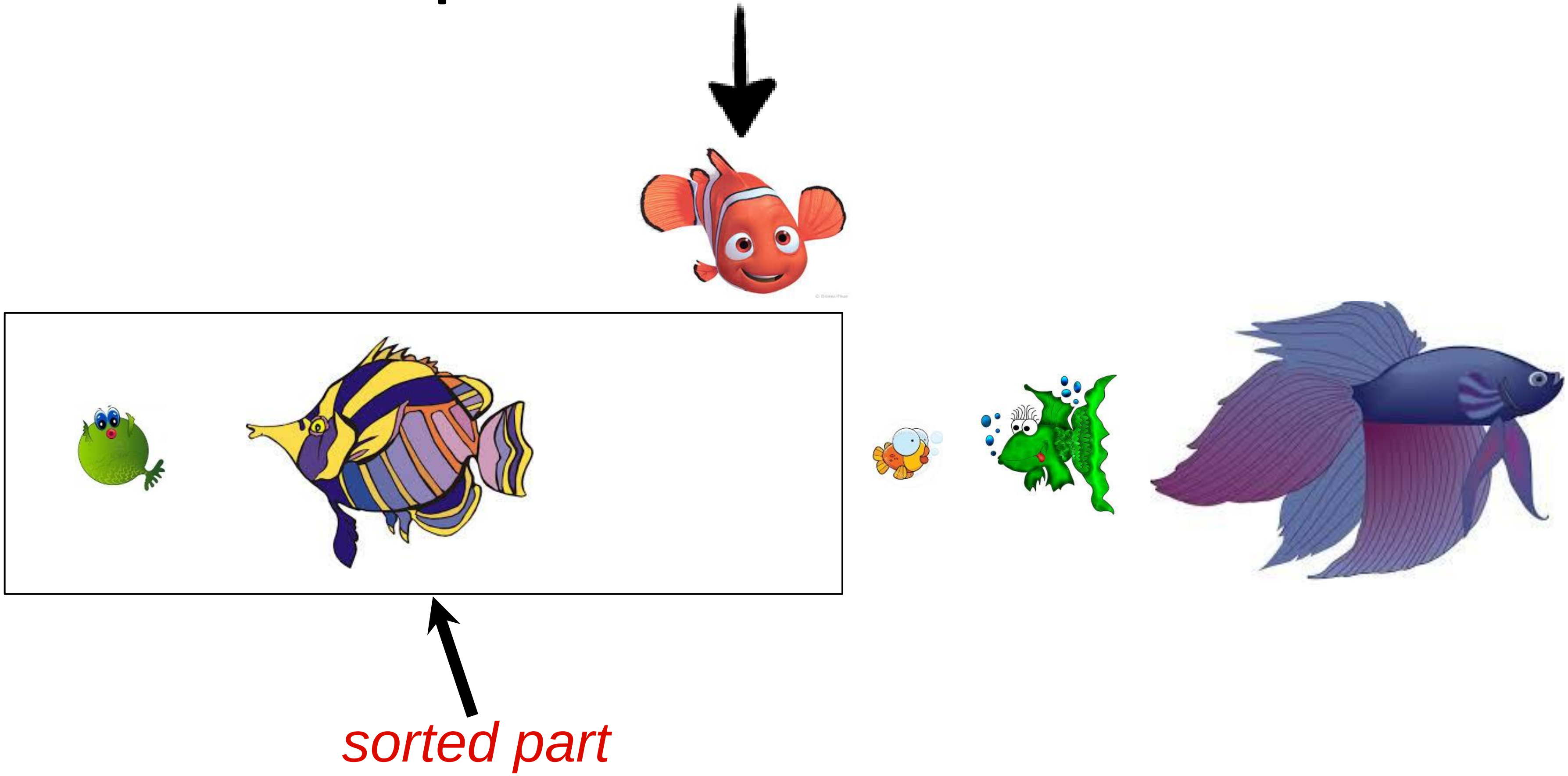
In-place Insertion Sort



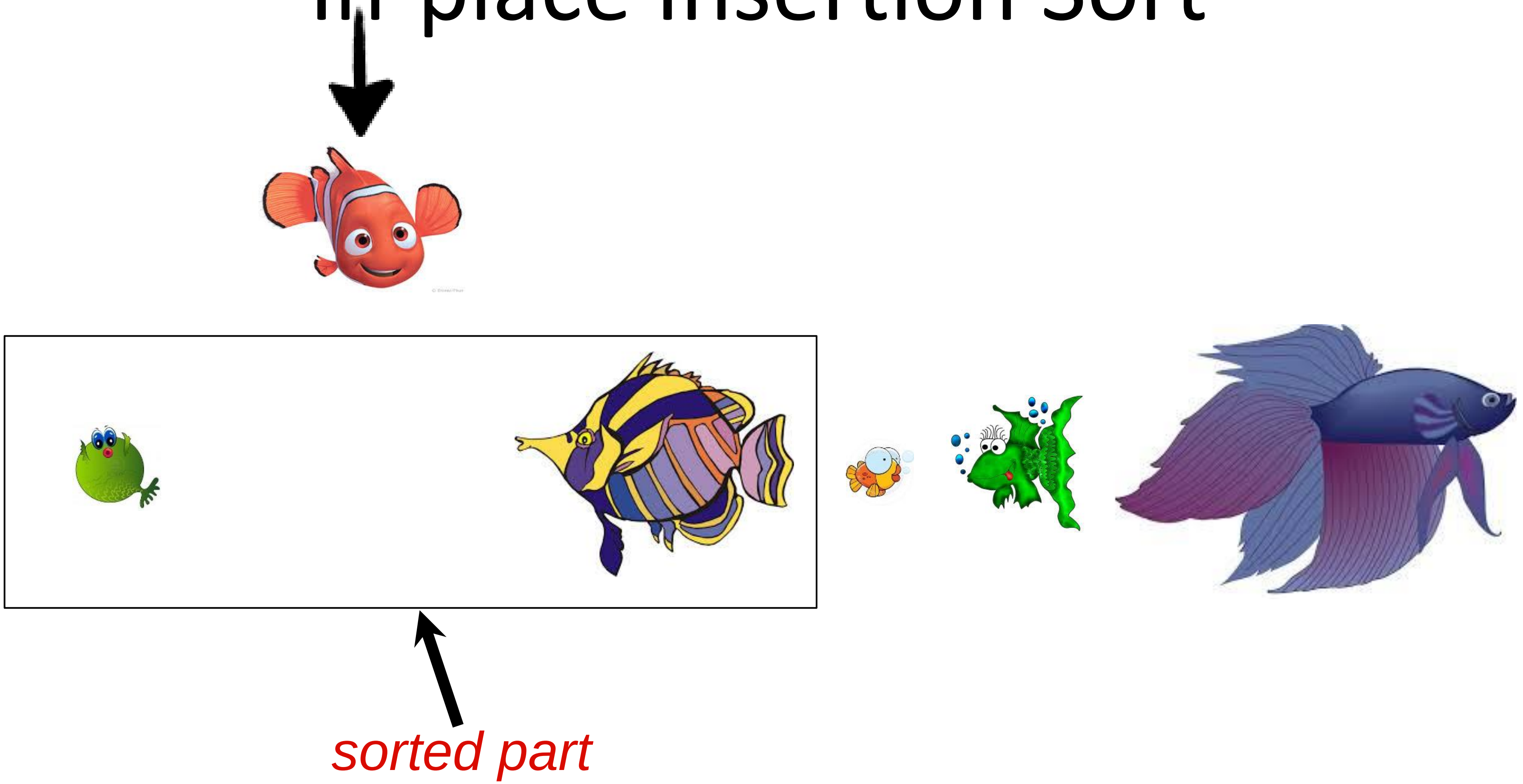
In-place Insertion Sort



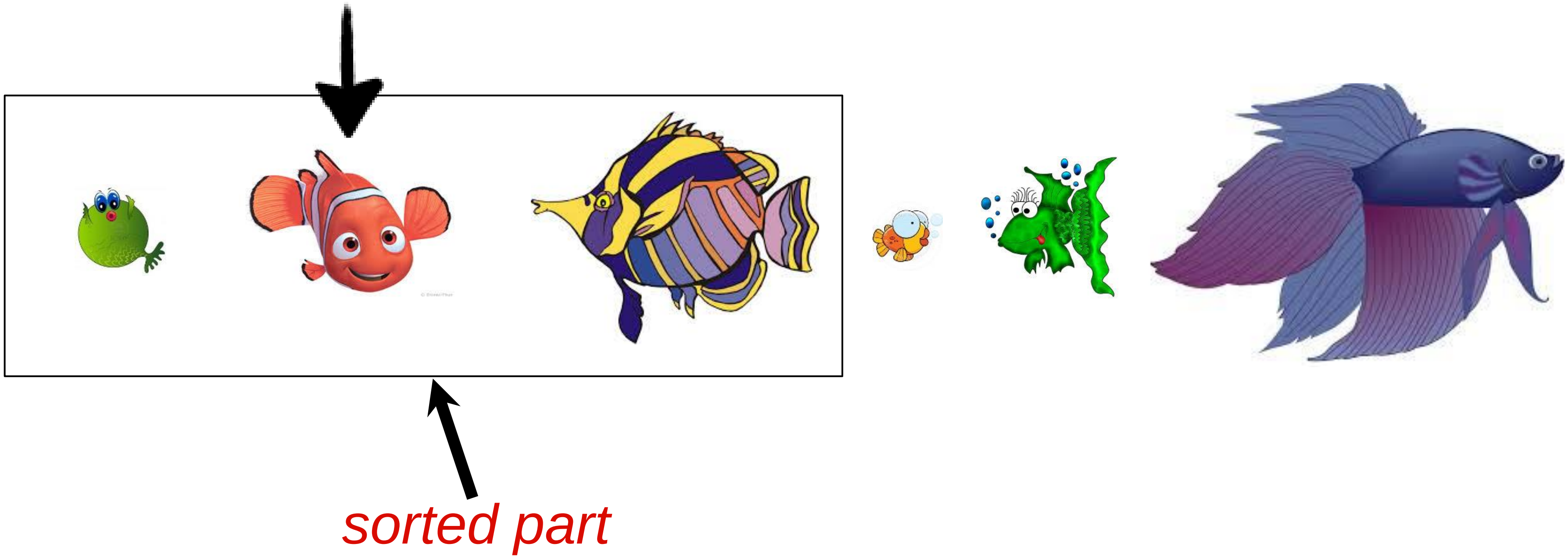
In-place Insertion Sort



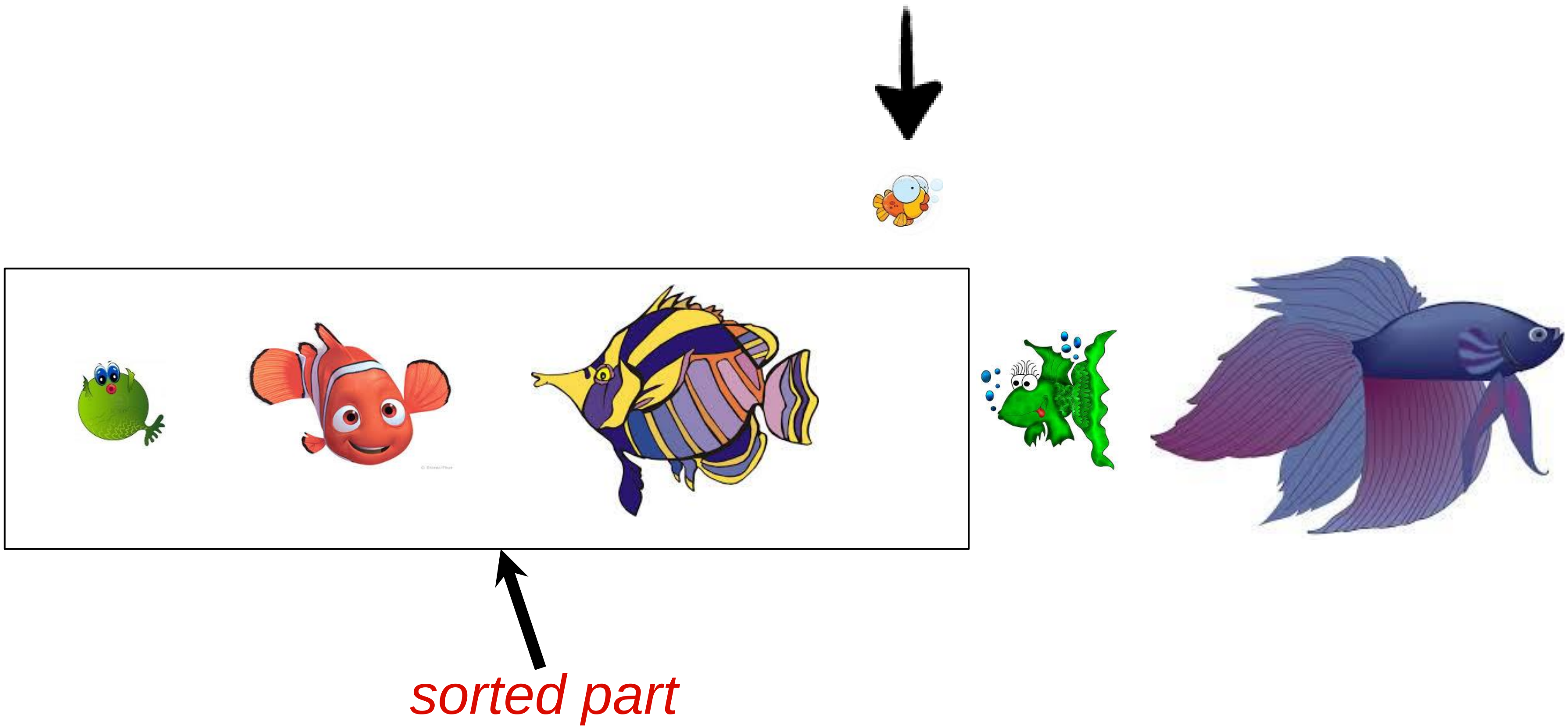
In-place Insertion Sort



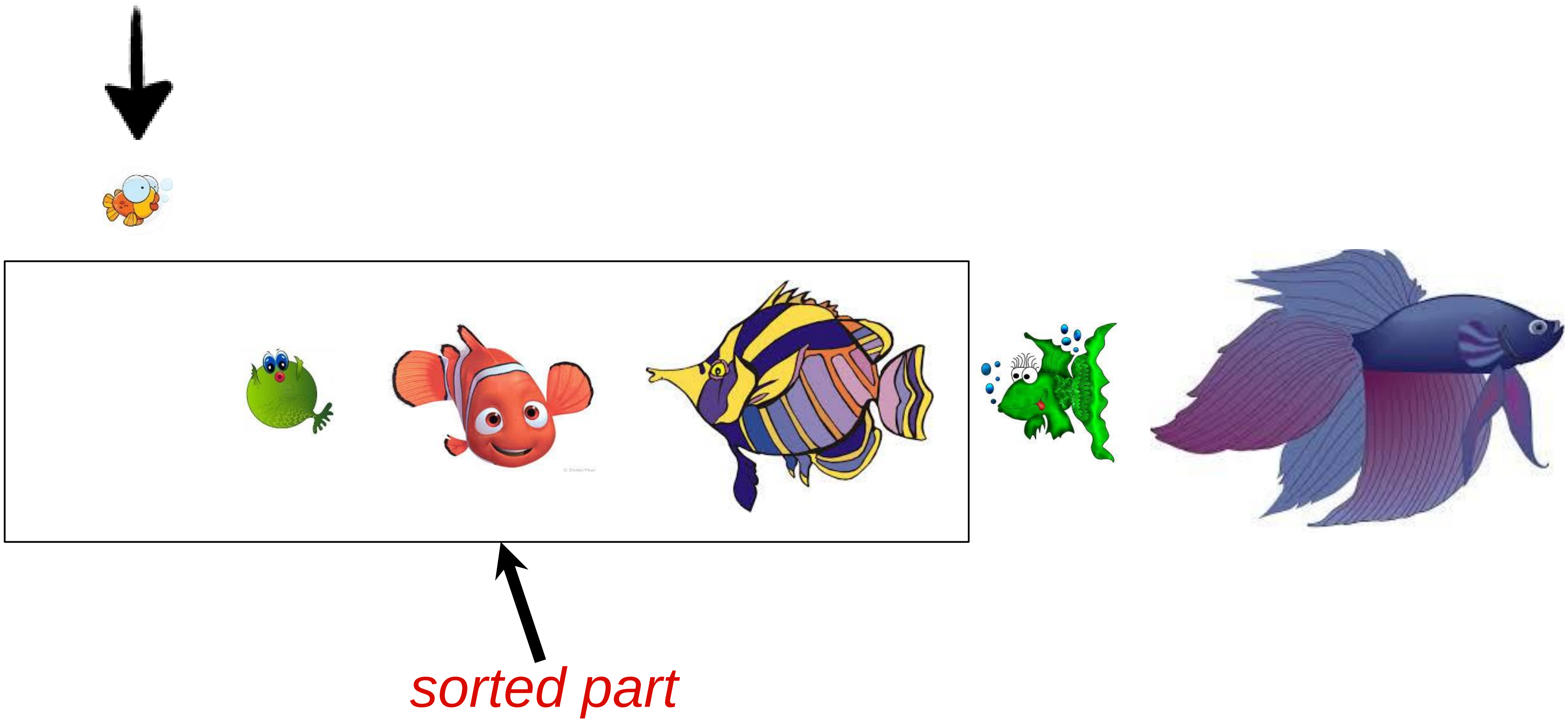
In-place Insertion Sort



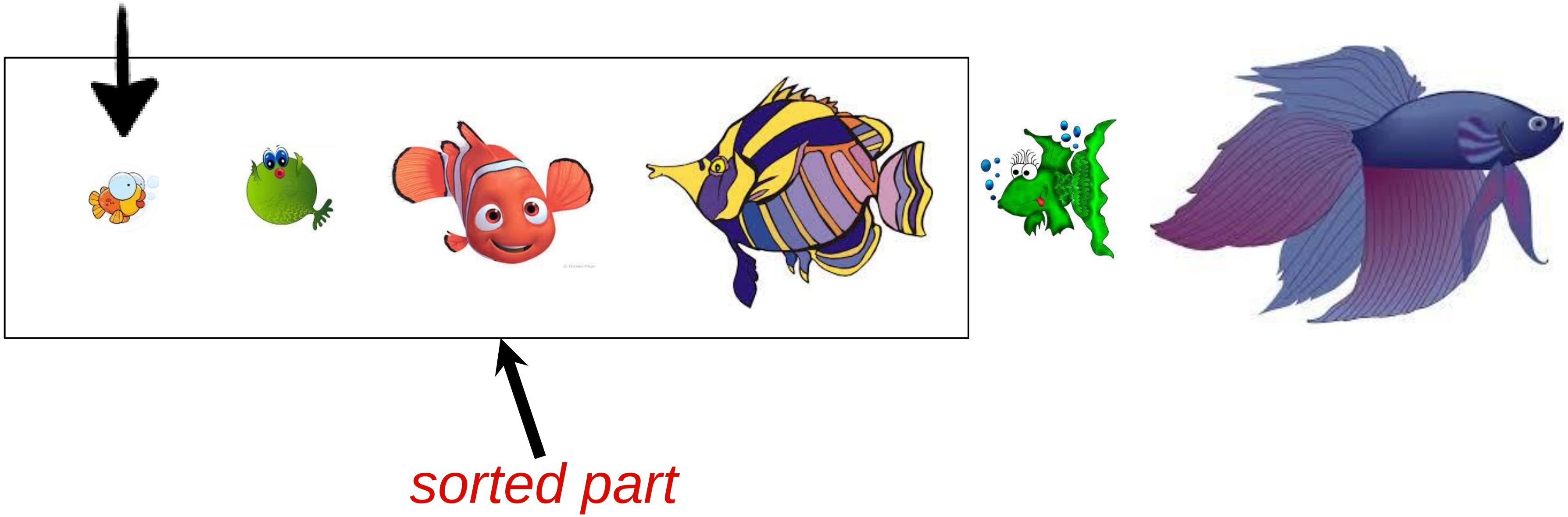
In-place Insertion Sort



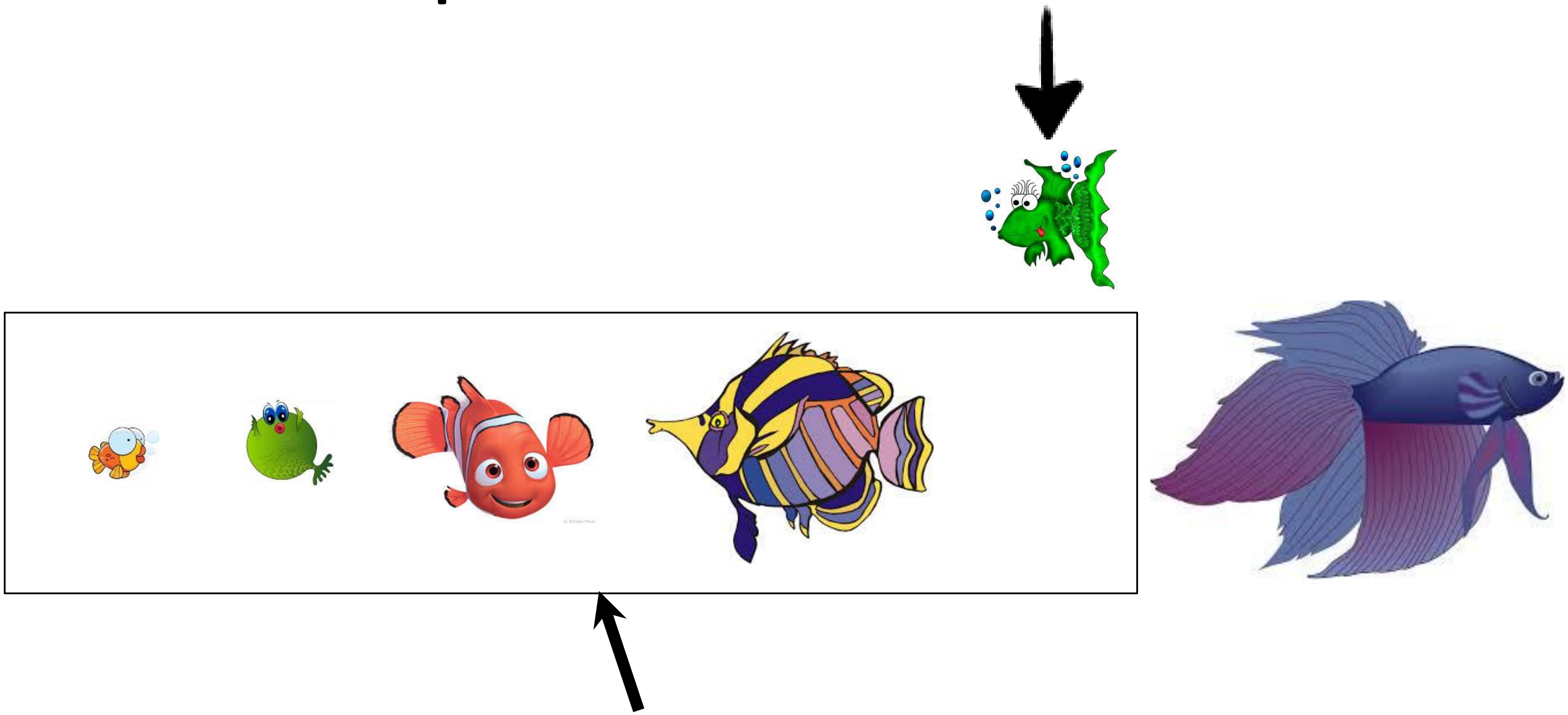
In-place Insertion Sort



In-place Insertion Sort

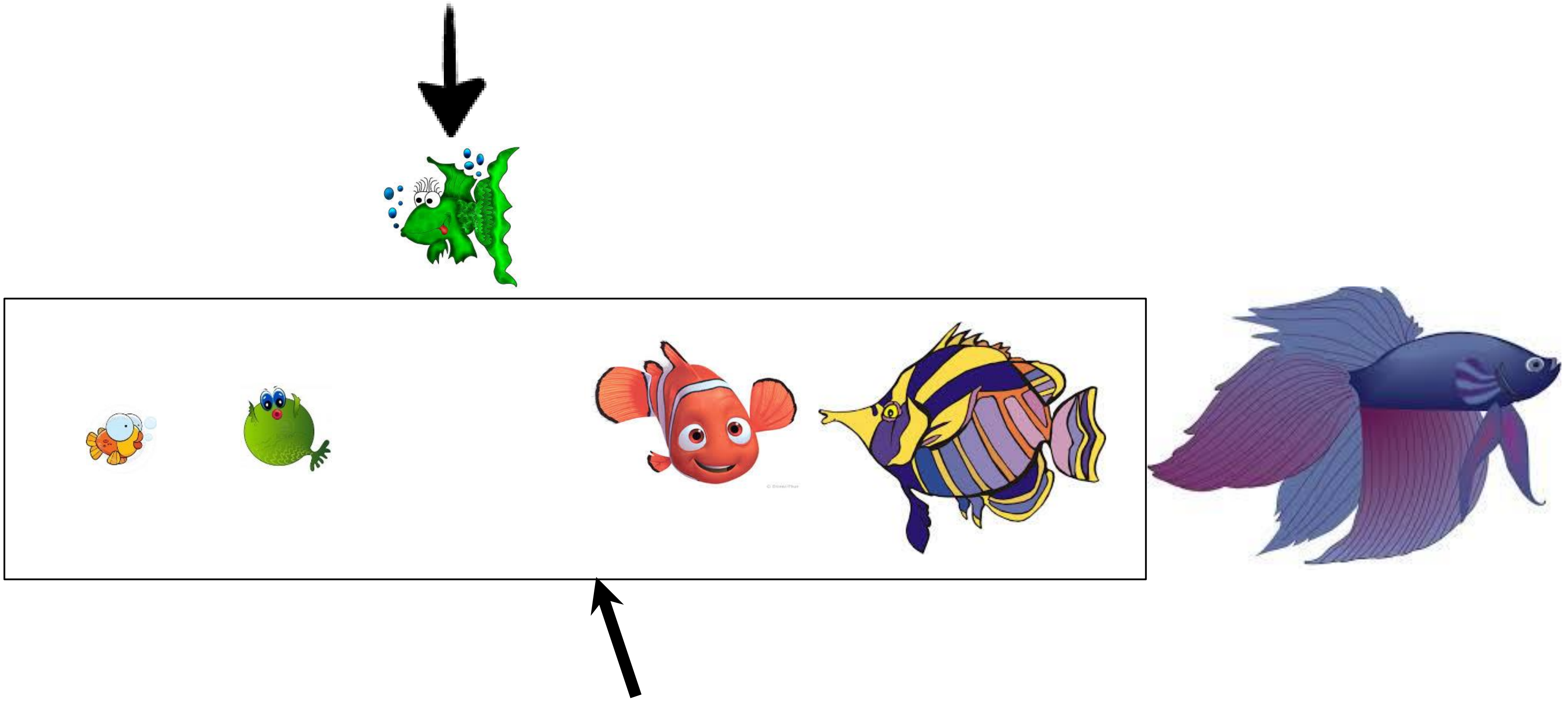


In-place Insertion Sort



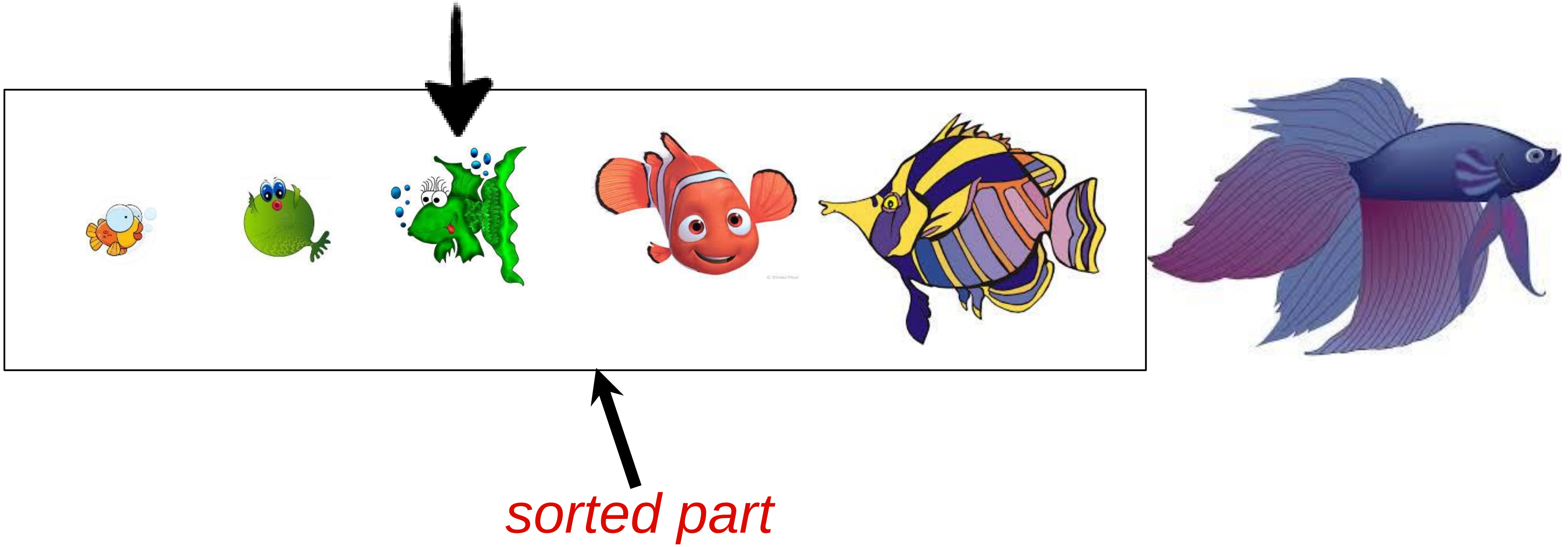
sorted part

In-place Insertion Sort

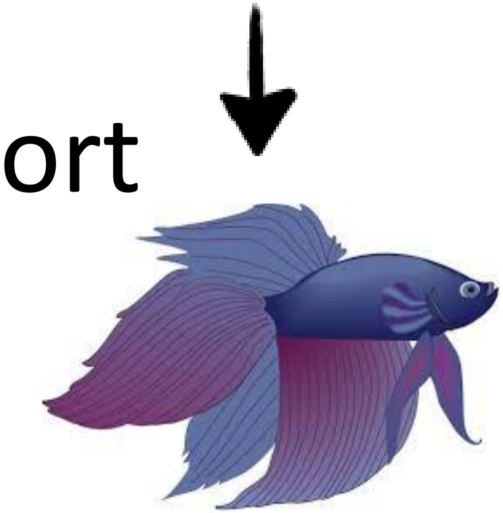


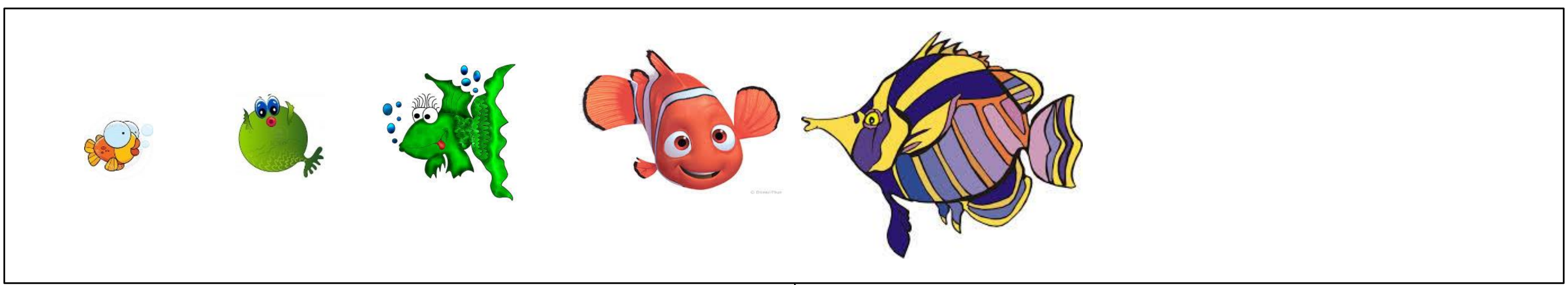
sorted part

In-place Insertion Sort



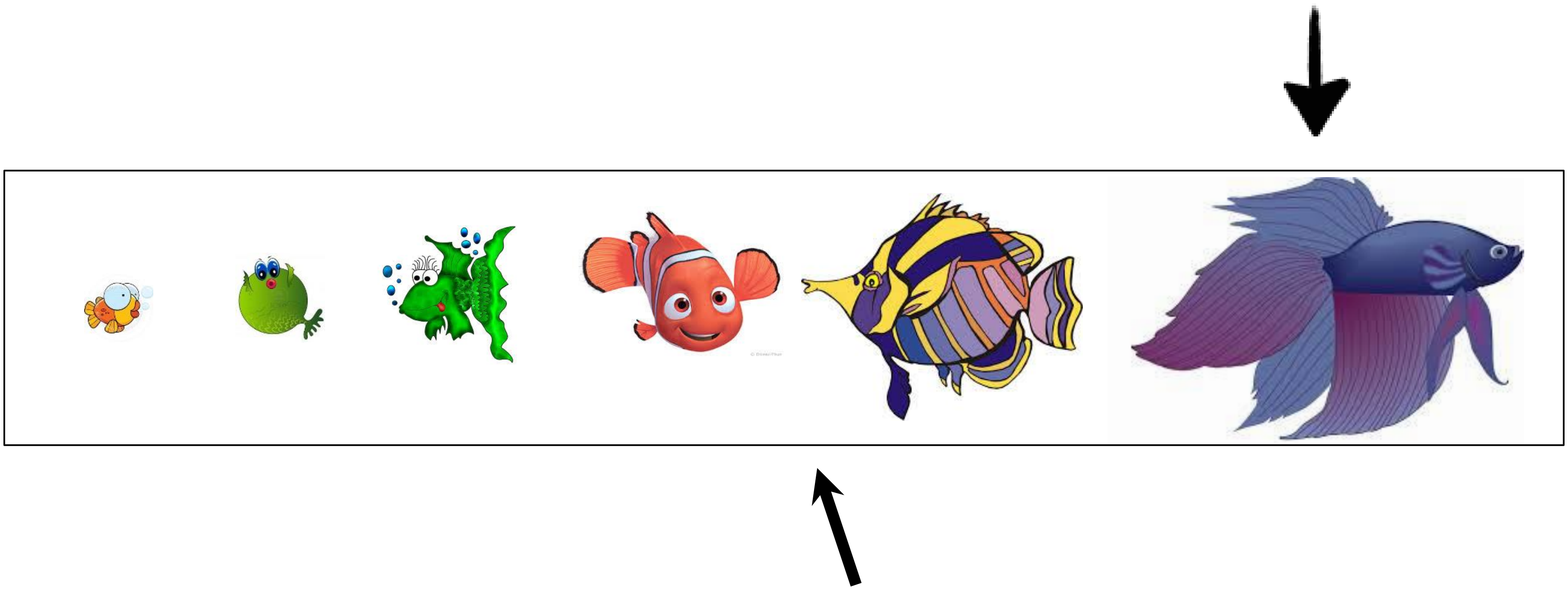
In-place Insertion Sort





sorted part

In-place Insertion Sort



sorted part

In-place Insertion Sort Algorithm

Given a list *a* of length *n*, *n* > 0.

1. Set *i* = 1.
2. While *i* is not equal to *n*, do the following:
   a. Insert *a*[*i*] into its correct position in *a*[0] to *a*[*i*] (inclusive).
   b. Add 1 to *i*.
3. Return the list *a* (which is now sorted).

Example

a = [53, 26, 76, 30, 14, 91, 68, 42]

i = 1

Insert a[1] into its correct position in a[0..1] and then add 1 to i:

53 moves to the right,

26 is inserted into the list at position 0

a = [26, 53, 76, 30, 14, 91, 68, 42]

i = 2

Writing the Python code

```
def isort(items):
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

insert a[i] into a[0..i]
in its correct sorted
position

Great, we've written isort….

BUT WE HAVE TO WRITE THE MOVE_LEFT FUNCTION!

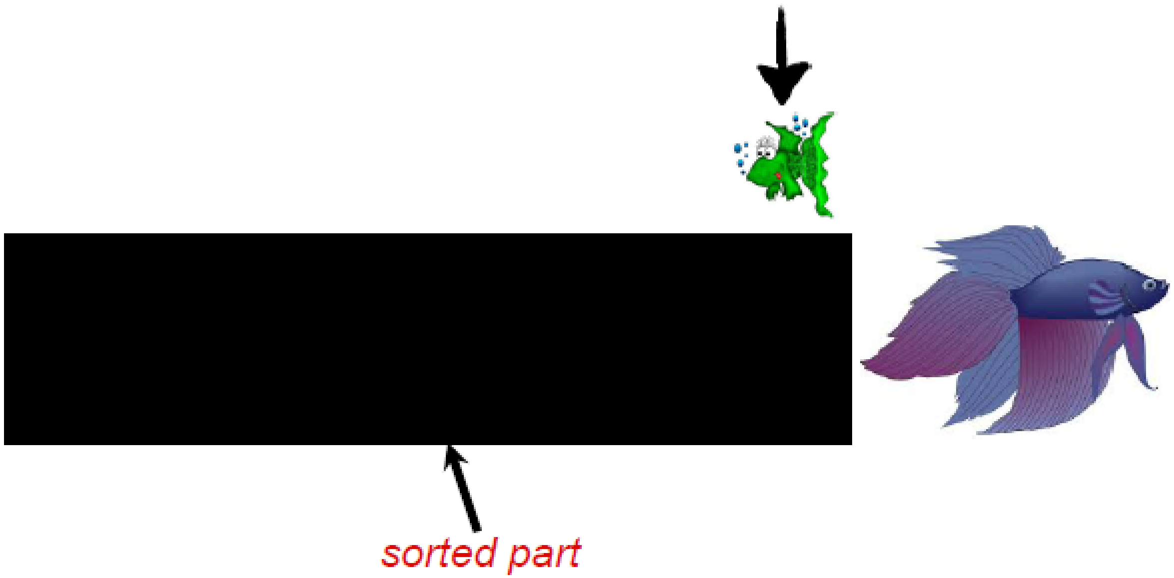
Moving left using search

To move the element x at index i "left" to its correct position, start at position i-1, and **search from right to left** until we find the first element that is less than or equal to x.

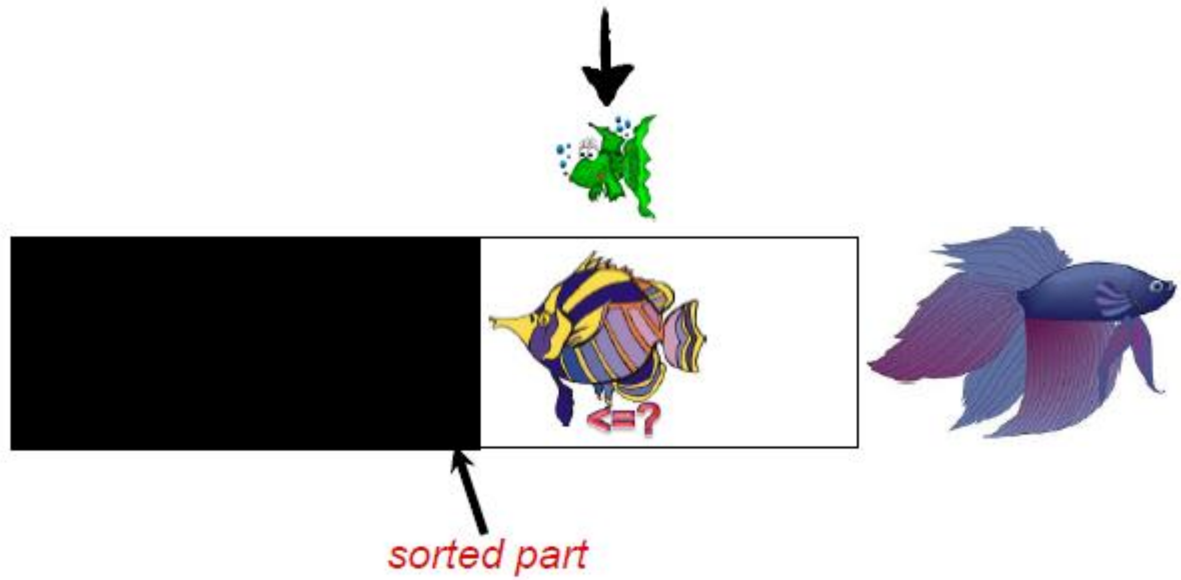
Then insert x back into the list to the right of that element.

(The Python insert operation does not overwrite. Think of it as "squeezing into the list".)

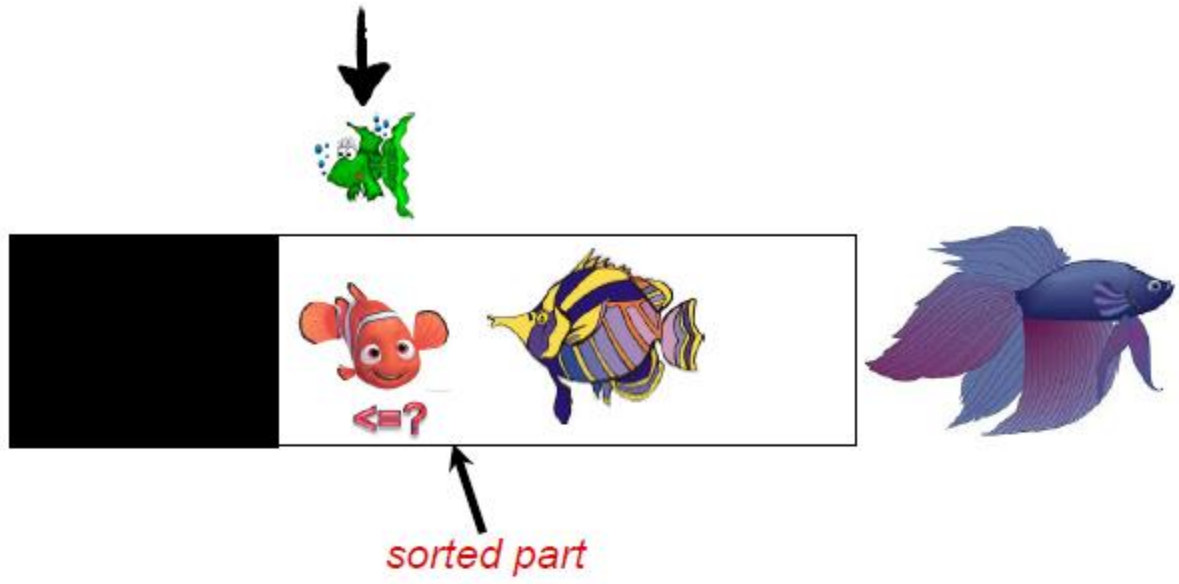
Move_left via linear search



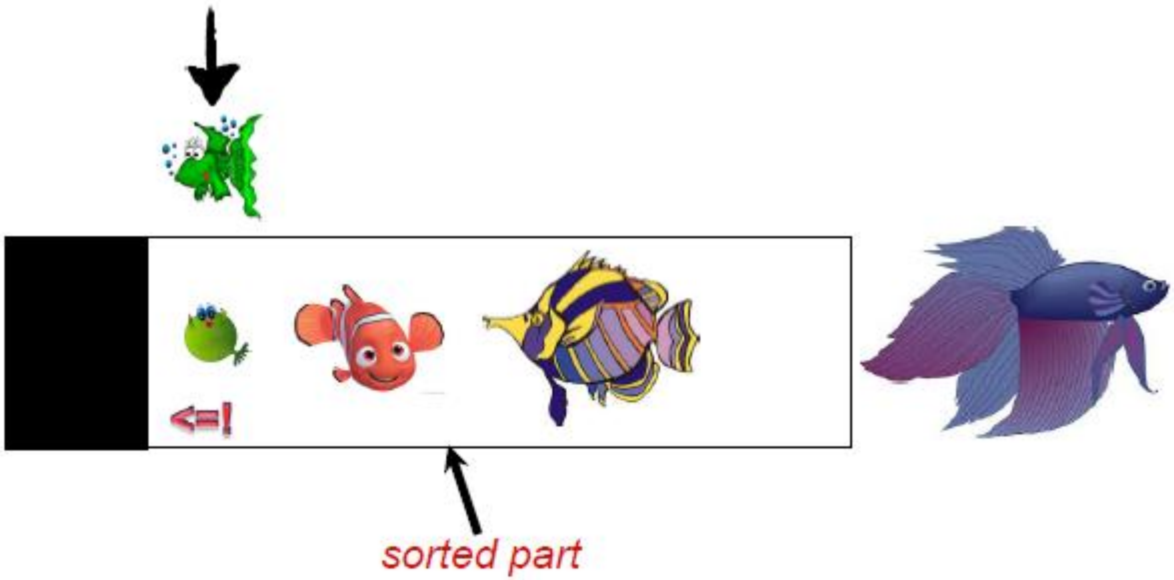
Move_left via linear search



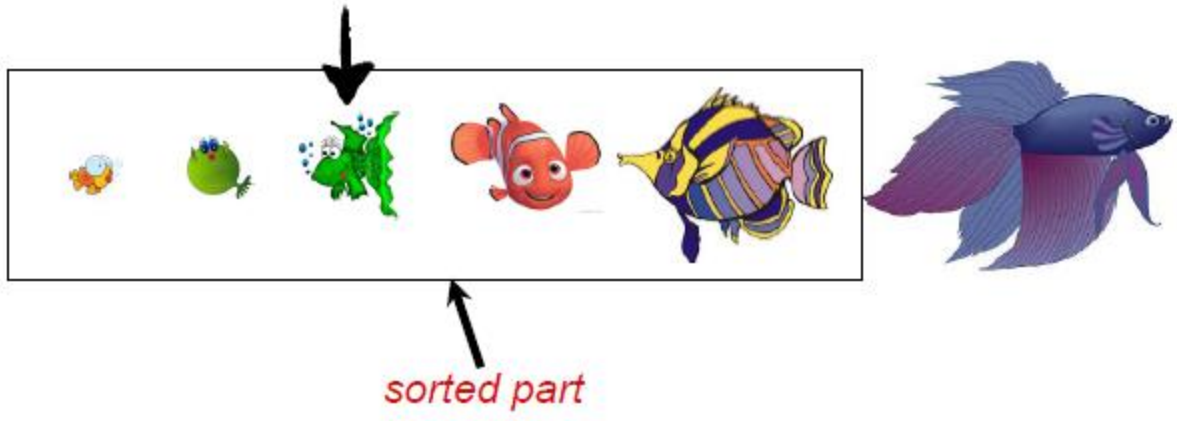
Move_left via linear search



Move_left via linear search



Move_left via linear search



Moving left: examples

76:

```
a = [26, 53, 76, 30, 14, 91, 68, 42]
```

Searching from right to left starting with 53, the first element less than 76 is 53. Insert 76 to the right of 53 (where it was before).

14:

```
a = [26, 30, 53, 76, 14, 91, 68, 42]
```

Searching from right to left starting with 76, all elements left of 14 are greater than 14. Insert 14 into position 0.

68:

```
a = [14, 26, 30, 53, 76, 91, 68, 42]
```

Searching from right to left starting with 91, the first element less than 68 is 53. Insert 68 to the right of 53.

The `move_left` algorithm

Given a list *a* of length *n*, *n* > 0 and a value at index *i* to be "moved left" in the list.

1. Remove *a*[*i*] from the list and store in *x*.
2. Set *j* = *i-1.*
3. While *j* >= 0 and *a*[*j*] > *x*, do the following:

    a. Subtract 1 from *j*.

4. (**At this point, what do we know? Either *j* is …, or *a*[*j*] is …**) Reinsert *x* into position *a*[*j*+1].

From algorithm to code

- Our algorithm says to ***remove*** and ***insert*** elements of a list.

How do we do that?

- There are built –in Python operations for that

Removing a list element: pop

```
>>> a = ["Wednesday", "Monday", "Tuesday"]

>>> day = a.pop(1)
>>> a
['Wednesday', 'Tuesday']
>>> day
'Monday'

>>> day = a.pop(0)
>>> day
'Wednesday‘
>>> a
['Tuesday']
```

Inserting an element: insert

```
a = [10, 20, 30]  → [10, 20, 30]

a.insert(0,"foo") → ["foo", 10, 20, 30]

a.insert(2,"bar") → ["foo", 10, "bar", 20, 30]

a.insert(5,"baz") → ["foo", 10, "bar", 20, 30, "baz"]
```

move_left in Python

remove the item at position i in list a and store it in x

logical operator AND: both conditions must be true for the loop to continue

```
def move_left(items, i):
    x = items.pop(i)
    j = i - 1
    while j >= 0 AND items[j] > x:
        j = j – 1
    items.insert(j + 1, x)
```

insert x at position j+1 of list a, shifting all elements from j+1 and beyond over one position

Next Time

1. Debugging
2. Thinking about algorithm efficiency