Iteration: Searching

Announcements

- Questions?
  - Lab 3
  - PA 3
  - OLI
  - PS 3
- Tonight
  - Lab 4
- Autograding…

Today

- Sieve of Erotosthenes (lists) review?
- Coding: Unicode
- Algorithm: linear (sequential) search
- Thinking about efficiency
- *Algorithm: insertion sort*

Lists and Loops

```
myList =[‘pizza’, ‘tacos’, ‘burgers’, ‘kale’, ‘lentils’]
```

```
i= 1

while i < len(myList):
       print(i)
       print(myList[i])
        i = i + 1
```

```
i= 1

for i in
   range(len(myList)):
      print(i)
      print(myList[i])
```

```
i= 1

for i in myList:
     print(i)
     print(myList[i])
```

Algorithmic Thinking: Sieve of Erathosthenes

Do we need to review?

Prime Numbers

- An integer is "prime" if it is not divisible by any smaller integers except 1.

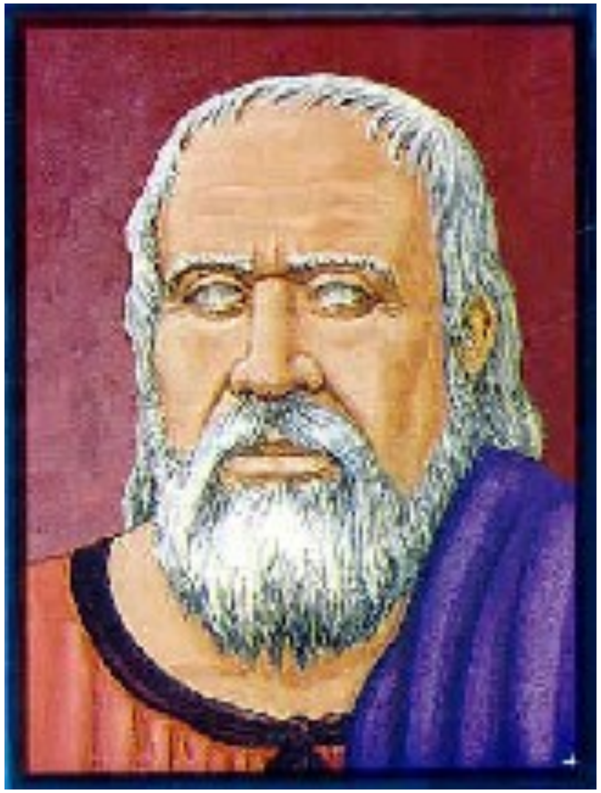
- 10 is **not** prime because 10 = 2 × 5
- 11 **is** prime
- 12 is **not** prime because 12 = 2 × 6 = 2 × 2 × 3
- 13 **is** prime
- 15 is **not** prime because 15 = 3 × 5

The Sieve of Eratosthenes

Start with a table of integers from 2 to N.

Cross out all the entries that are divisible by the primes known so far.

The first value remaining is the *next* prime.



What’s the point?

- From algorithm to code…!
- Problem Solving

Finding Primes Between 2 and 50

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

2 is the first prime

Finding Primes Between 2 and 50

	2	**3**	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Filter out everything divisible by 2.

Now we see that 3 is the next prime.

Finding Primes Between 2 and 50

	2	**3**	4	**5**	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Filter out everything divisible by 3.

Now we see that 5 is the next prime.

Finding Primes Between 2 and 50

	2	**3**	4	**5**	6	**7**	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Filter out everything divisible by 5.

Now we see that 7 is the next prime.

Finding Primes Between 2 and 50

	2	**3**	4	**5**	6	**7**	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Filter out everything divisible by 7.

Now we see that11 is the next prime.

Finding Primes Between 2 and 50

	2	**3**	4	**5**	6	**7**	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Since 11 x 11 > 50, all remaining numbers must be primes. Why?

An Algorithm for Sieve of Eratosthenes

Input: A number *n*:

1. Create a list *numlist* with every integer from 2 to n, in order. (Assume *n* > 1.)

2. Create an empty list *primes*.

3. For each element in *numlist*

    a. If element is not marked, copy it to the end of *primes*.

    b. Mark every number that is a multiple of the most recently discovered prime number.

Output: The list of all prime numbers less than or equal to *n*

Automating the Sieve

numlist

2 3 4 5
6 7 8 9
10 11 12 13
…

primes

Use *two* lists: candidates, and confirmed primes.

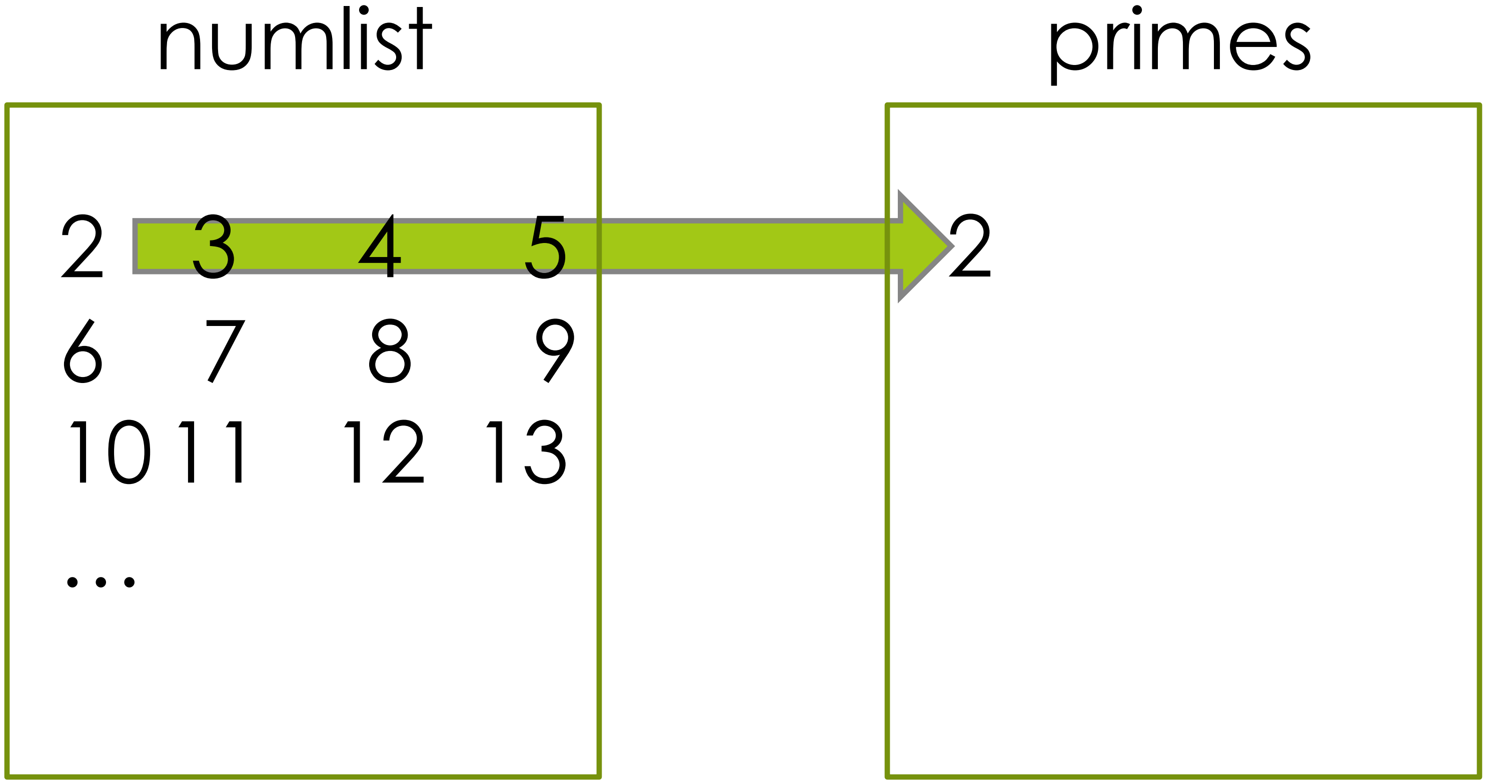
Steps 1 and 2

numlist

2 3 4 5
6 7 8 9
10 11 12 13
…

primes

Step 3a



Append the current number in numlist to the end of primes.

Step 3b

numlist

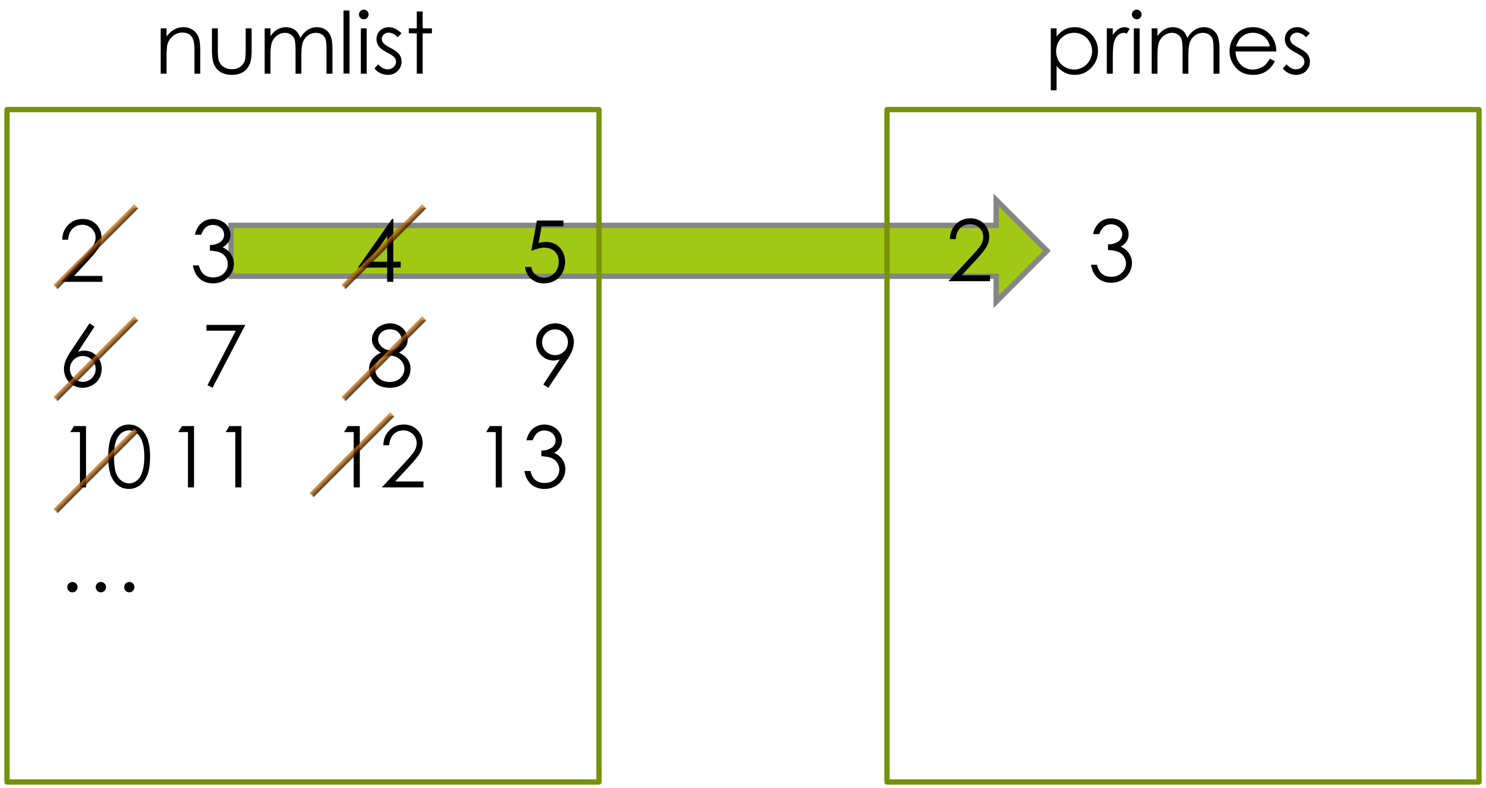
~~2~~ 3 ~~4~~ 5
~~6~~ 7 ~~8~~ 9
~~10~~ 11 ~~12~~ 13
…

primes

2

Cross out all the multiples of the last number in primes.

Iterations



Append the current number in numlist to the end of primes.

Iterations

numlist

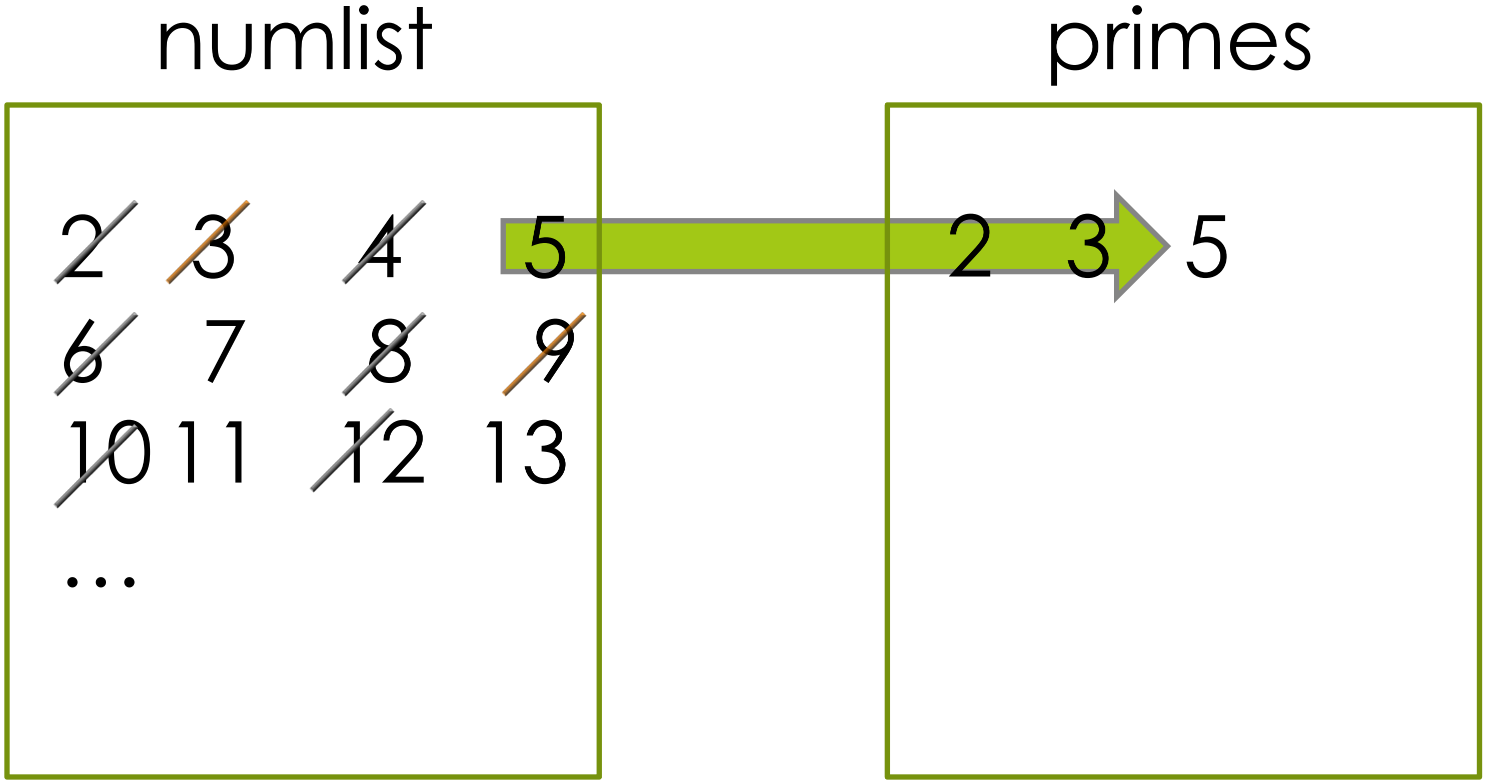
~~2~~ ~~3~~ ~~4~~ 5
~~6~~ 7 ~~8~~ ~~9~~
~~10~~ 11 ~~12~~ 13
…

primes

2 3

Cross out all the multiples of the last number in primes.

Iterations



Append the current number in numlist to the end of primes.

Iterations

numlist

~~2~~ ~~3~~ ~~4~~ ~~5~~
~~6~~ 7 ~~8~~ ~~9~~
~~10~~ 11 ~~12~~ 13
…

primes

2 3 5

Cross out all the multiples of the <u>last</u> number in primes.

An Algorithm for Sieve of Eratosthenes

Input: A number *n*:

1. Create a list *numlist* with every integer from 2 to n, in order. (Assume *n* > 1.)

2. Create an empty list *primes*.

3. For each element in *numlist*

   a. If element is not marked, copy it to the end of *primes*.

   b. Mark every number that is a multiple of the most recently discovered prime number.

Output: The list of all prime numbers less than or equal to *n*

Implementation Decisions

- How to implement *numlist* and *primes?*
    - *For numlist* we will use a list in which crossed out elements are marked with the special value `None`. For example,

```
[None, 3, None, 5, None, 7, None]
```

- Use a helper function to mark the multiples, step 3.b. We will call it sift.

Relational Operators

- If we want to compare two integers to determine their relationship, we can use these **relational operators**:

<	less than	<=	less than or equal to
>	greater than	>=	greater than or equal to
==	equal to	!=	not equal to

- We can also write compound expressions using the **Boolean operators** and and or.

x >= 1 and x <= 1

Sifting: Removing Multiples of a Number

```
def sift(lst,k):
    # marks multiples of k with None
    i = 0
    while i < len(lst):
        if lst[i] != None and lst[i] % k == 0:
            lst[i] = None
        i = i + 1
    return lst
```

Filters out the multiples of the number k from list by marking them with the special value None (greyed out ones).

Sifting: Removing Multiples of a Number (Alternative version)

```
def sift2(lst,k):
    i = 0
    while i < len(lst):
        if lst[i] % k == 0:
            lst.remove(lst[i])
        else:
            i = i + 1
    return lst
```

Filters out the multiples of the number k from list by modifying the list. **Be careful** in handling indices.

A Working Sieve

Use the first version of sift in this function, which does the filtering using Nones.

```
def sieve(n):
    numlist = list(range(2, n+1))
    primes = []
    for i in range(0, len(numlist)):
        if numlist[i] != None:
            primes.append(numlist[i])
            sift(numlist, numlist[i])
    return primes
```

We could have used `primes[len(primes)-1]` instead.

Helper function that we defined before

Observation for a Better Sieve

We stopped at 11 because all the remaining entries must be prime since $11 \times 11 > 50$.

2	**3**	4	**5**	6	**7**	8	9	10	
11	12	**13**	14	15	16	**17**	18	**19**	20
21	22	**23**	24	25	26	27	28	**29**	30
31	32	33	34	35	36	**37**	38	39	40
41	42	**43**	44	45	46	**47**	48	49	50

A Better Sieve

```
def sieve(n):
    numlist = list(range(2, n + 1))
    primes = []
    i = 0 # index 0 contains number 2
    while (i+2) <= math.sqrt(n):
        if numlist[i] != None:
            primes.append(numlist[i])
            sift(numlist, numlist[i])
        i = i + 1
    return primes + numlist
```

Strings and Unicode

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")

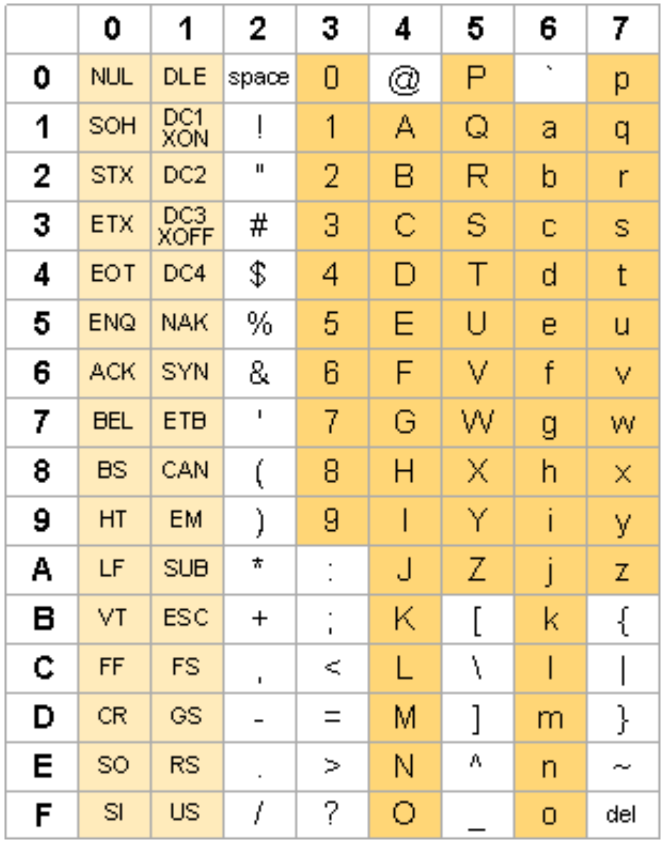
String comparisons

```
>>> 'A' < 'a’

>>> '1' < 'A'

>>> '1' < '2'

>>> '11' < '2'

>>> '12' < '112'

>>> 'abc' < 'b'

>>> 'alpha' < 'alphabet'

>>> 'awkward' < 'able'

>>>
```

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

Unicode in hexadecimal: 00 – 7F$_{16}$



	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Some non-printing characters:

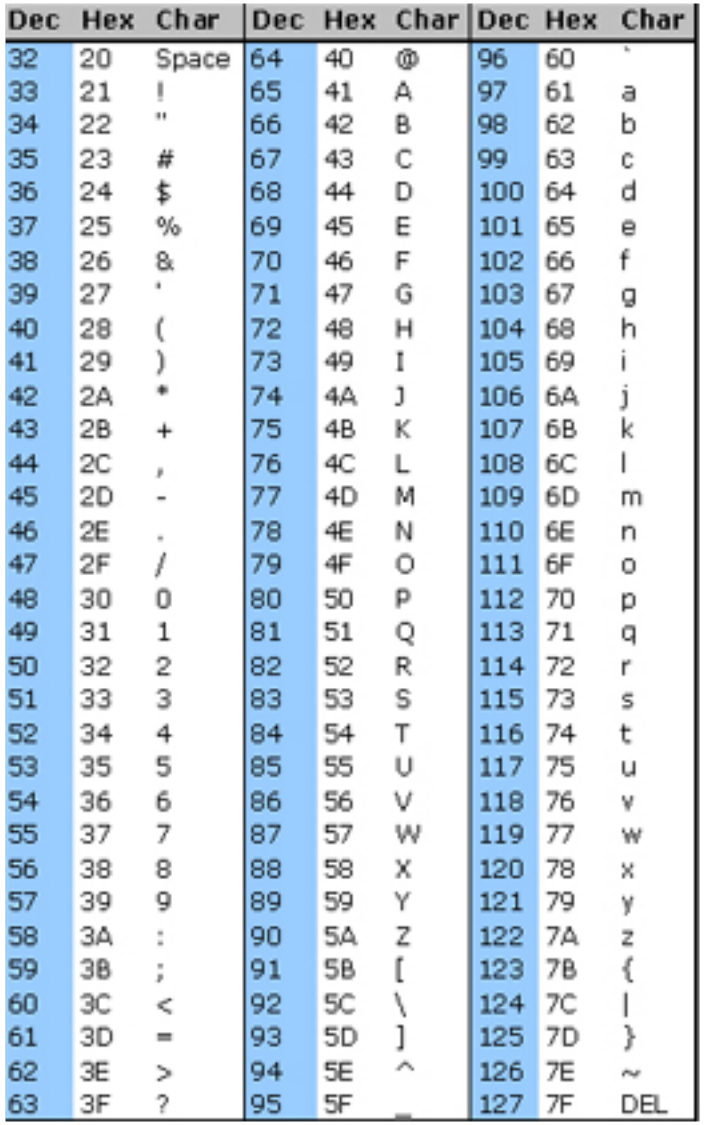
08 – back space
09 – horizontal tab
0A – newline character (in Python)

These are only the first 128 codes in the Unicode standard.

Chosen to correspond to the *entire* set of codes in the older ASCII standard.

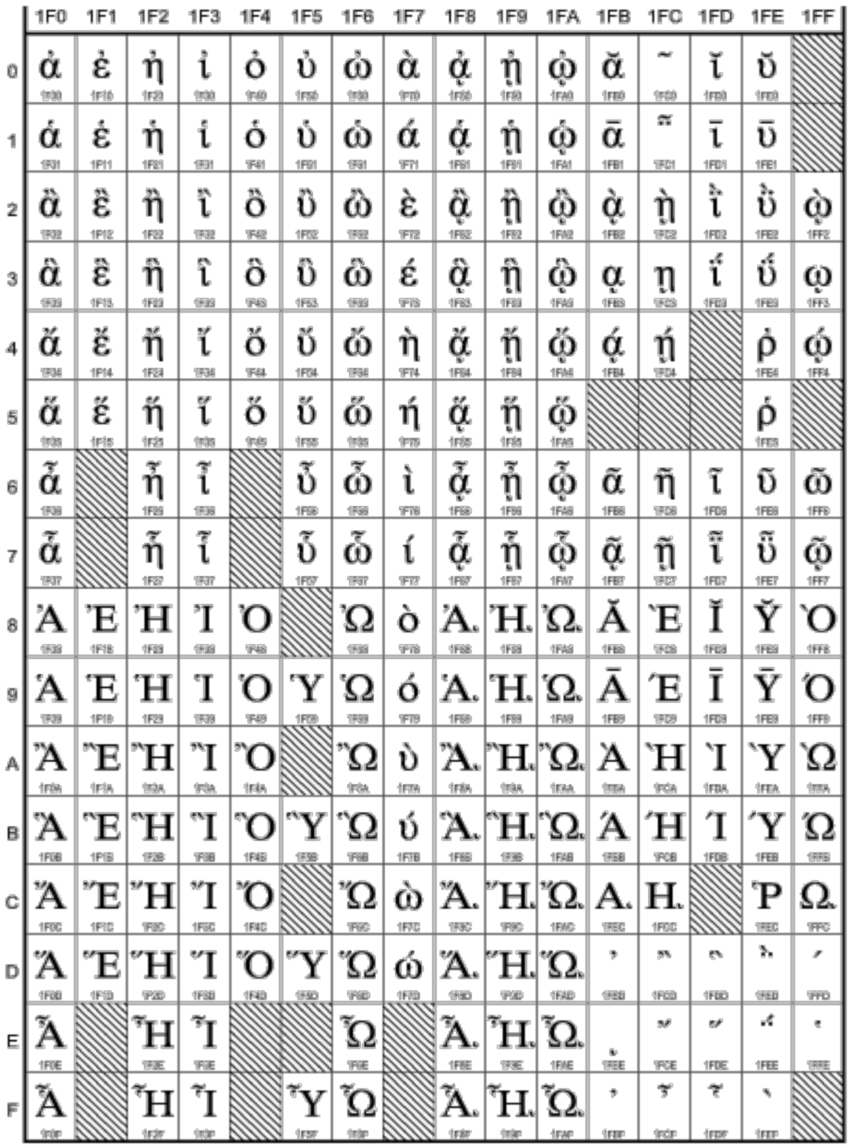
from ascii-table.com

Roman alphabet

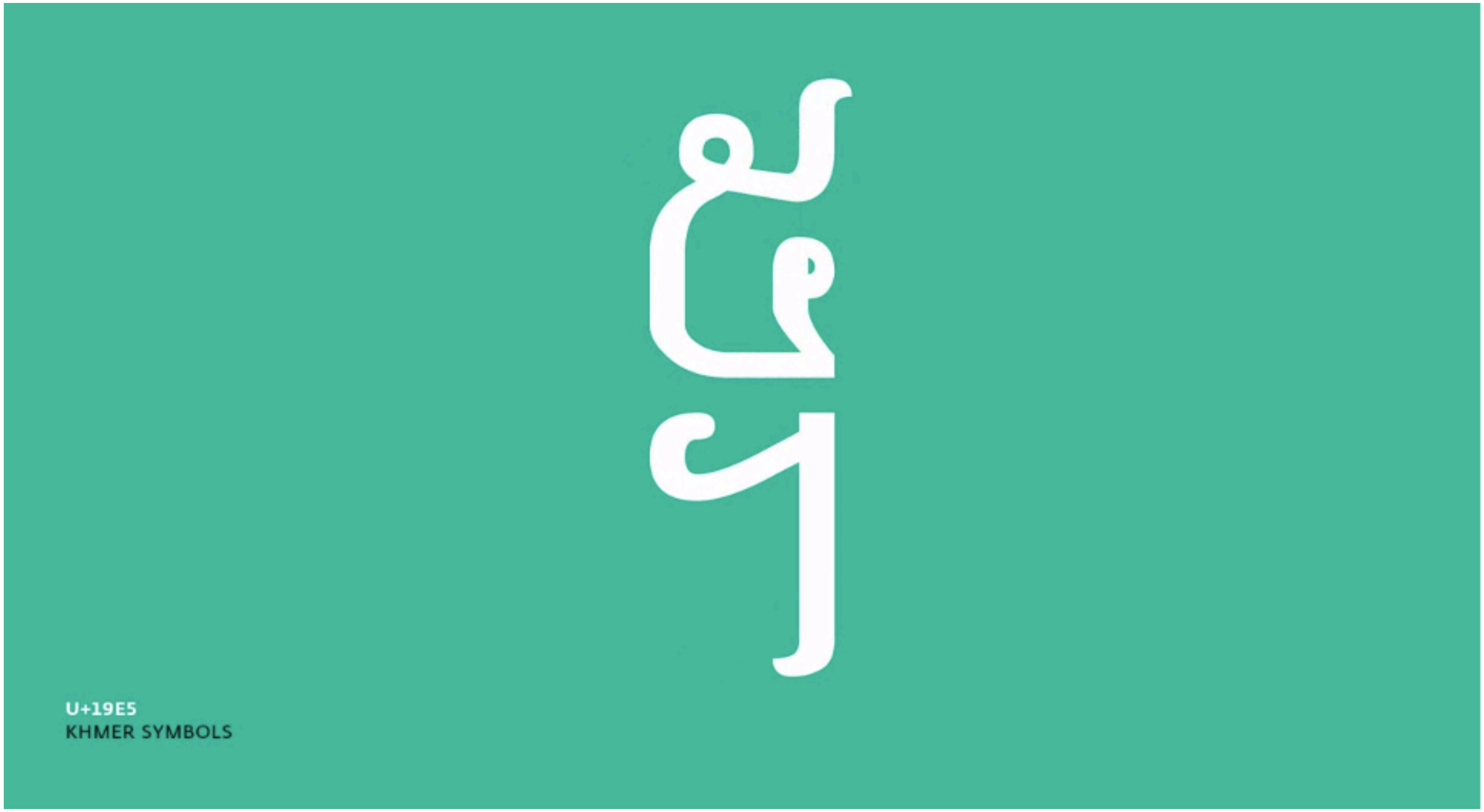


Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20	Space	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	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	▨

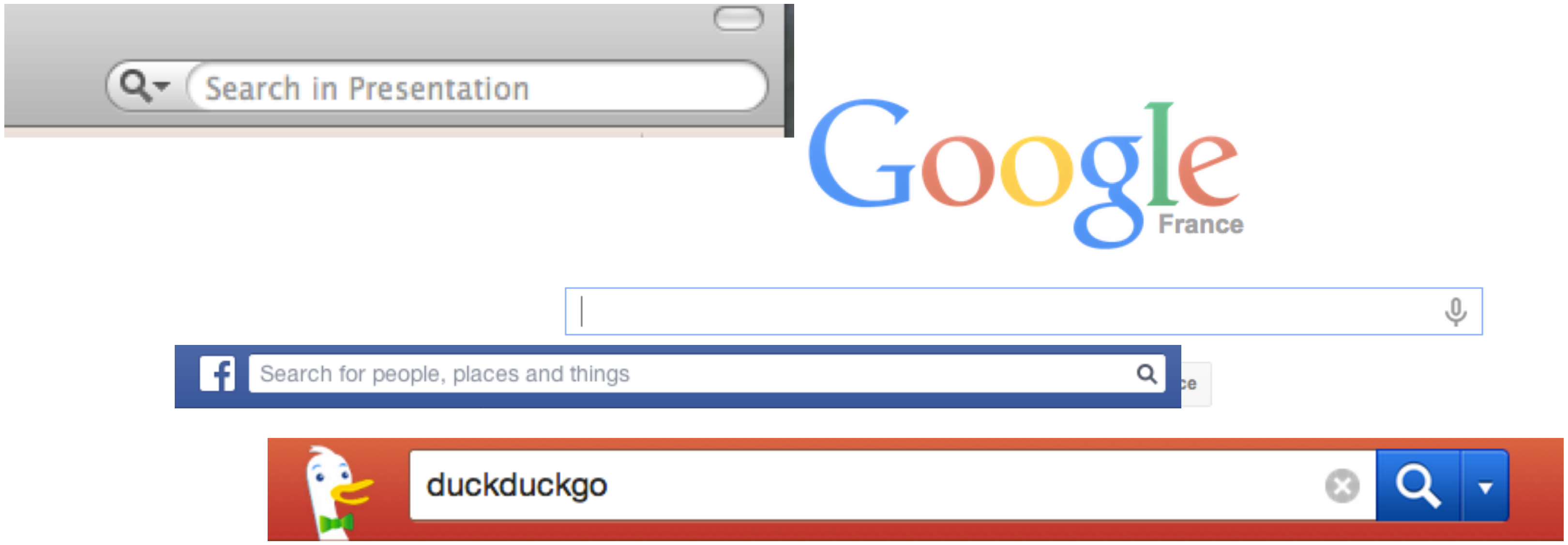


a unicode video: http://vimeo.com/48858289 109, 242 characters/codes in 2 hours, 31 mintes, and 25 seconds Amazingly, everything after around 14:00 seems to be (Chinese) ideographs!

Onward to search

more later on encodings, now

Searching, we use it



Built-in Search in Python

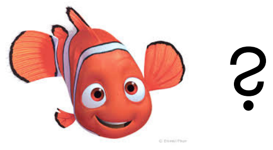
```
>>> movies = ["The Wolf of Wall Street", "American Hustle",
   "Frozen", "Her",  "Lone Survivor", "12 Years a Slave",
   "Nosferatu”, "Arnacoeur", "Sullivan's Travels", "Last Jedi"]

>>> "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

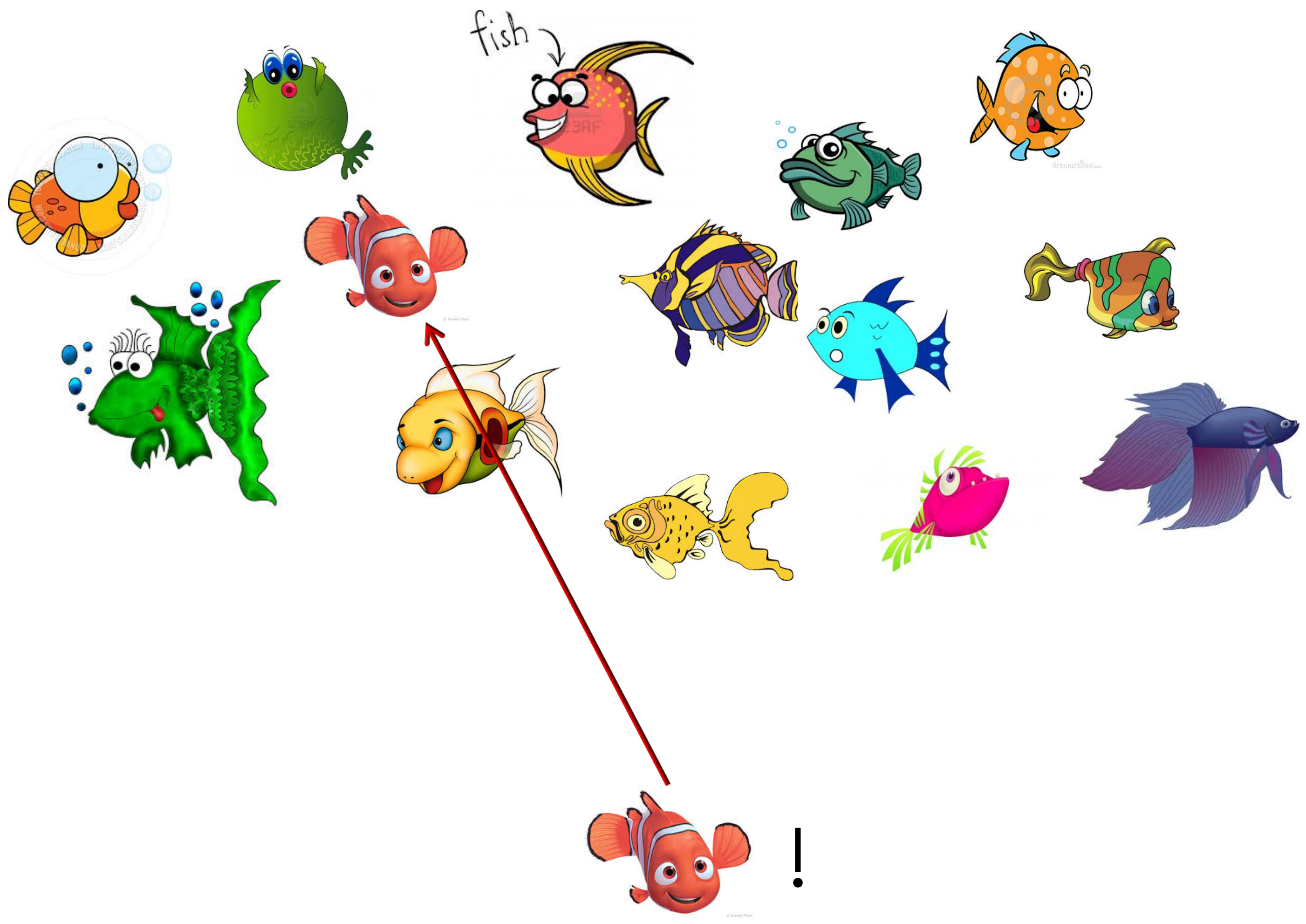
- Method `contains(items, key)`
- Input: `items` to be searched (could be strings or numbers or ...)
- Input: `key` to search for
- Output: `True` or `False`

- Approach: think linearly

Not thinking linearly…



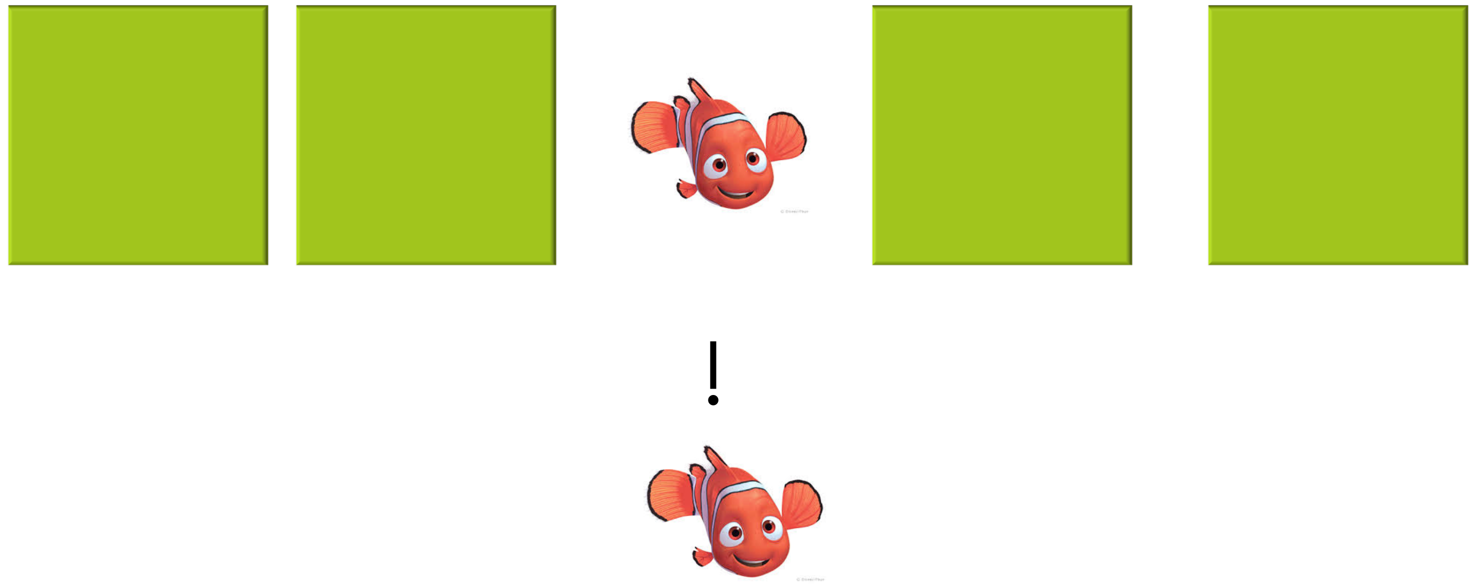
Not thinking linearly…



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 undesirabl
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
```

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

```
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 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:
- Define `time2` and `time3` similarly to call `contains2` and `contains`

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

Doing the measurement

```
>>> items = [None] * 1000000

>>> time1(items1, 1)
```

while loop

```
contains1: 0.1731700897216797

>>> time2(items1, 1)
```

while loop with saved length

```
contains2: 0.1145467758178711

>>> time3(items1, 1)
```

for loop

```
contains3: 0.07184195518493652
```

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

A Different Measurement

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

```
>>> time1(items1, None)
```
while loop

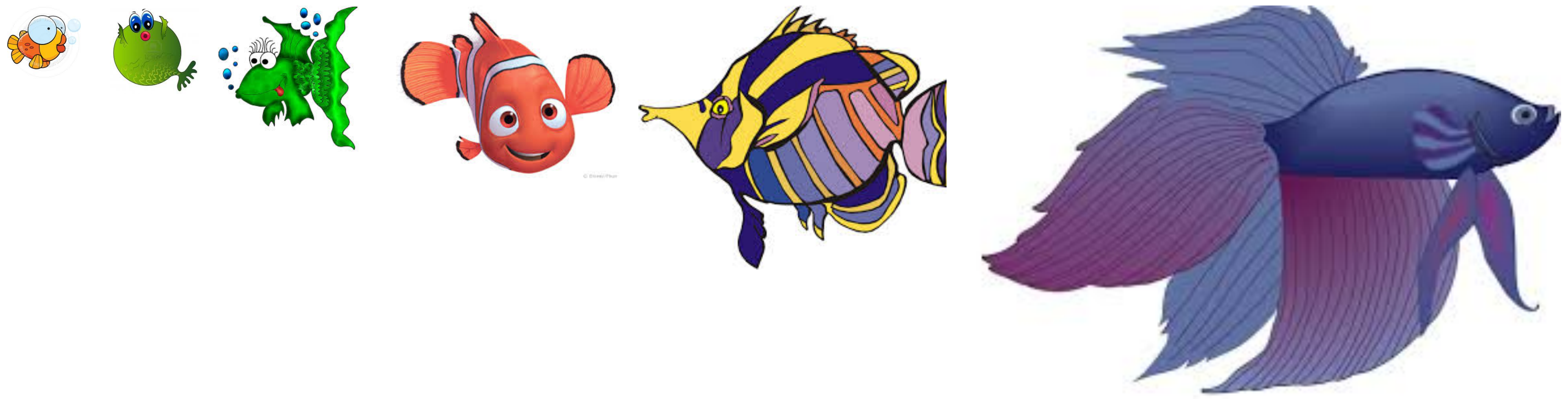
```
contains1: 4.0531158447265625e-06
>>> time2(items1, None)
```
while loop with saved length

```
contains2: 4.291534423828125e-06
>>> time3(items1, None)
```
for loop

```
contains3: 1.0013580322265625e-05
```

Now the relationship is different; `contains3` is slowest! Why?

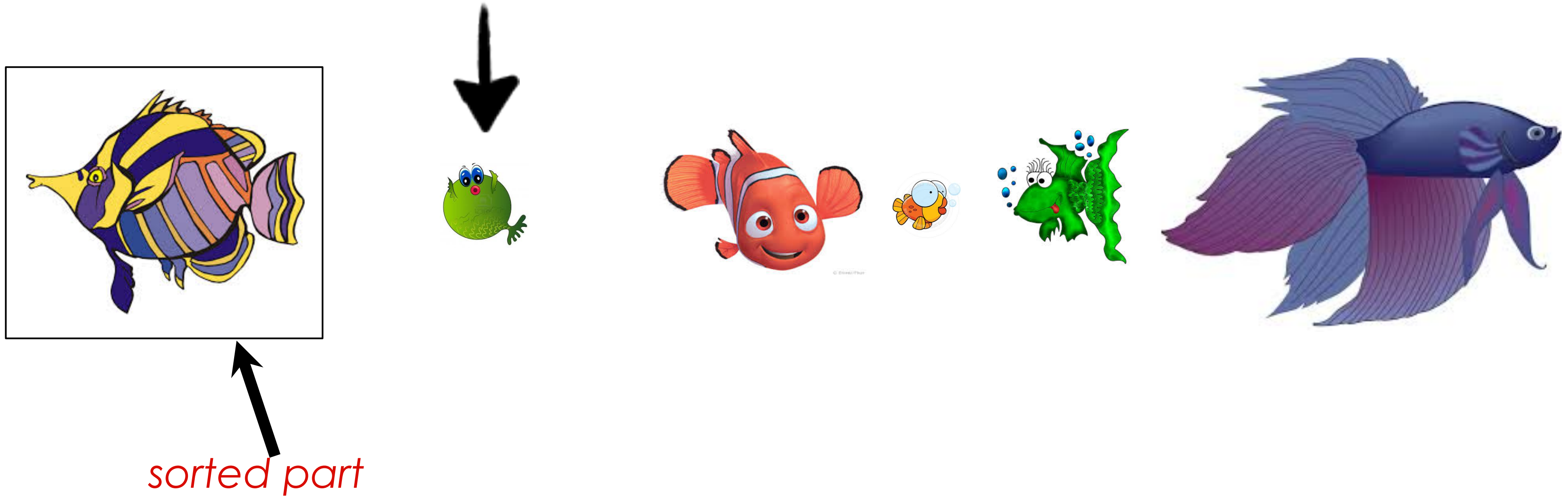
Sorting



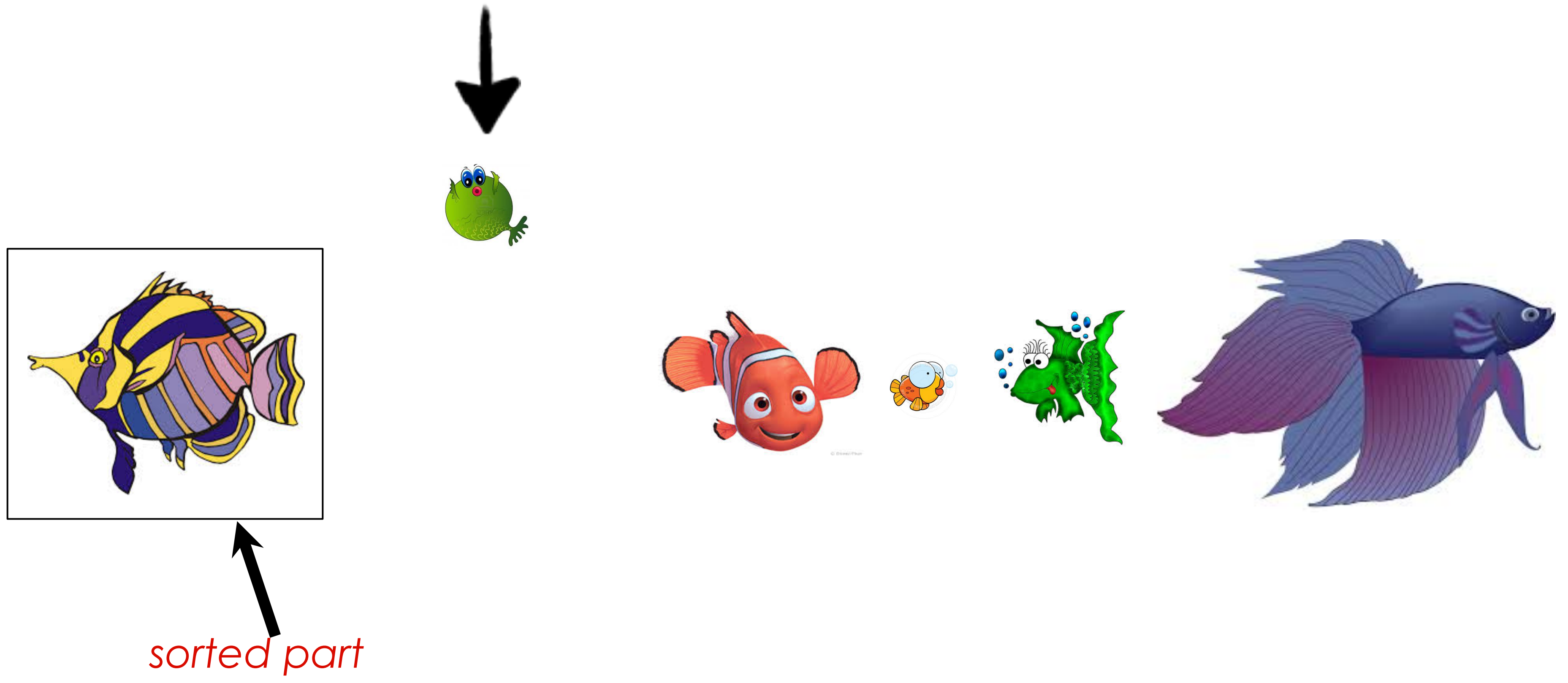
In-place Insertion Sort

- 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.

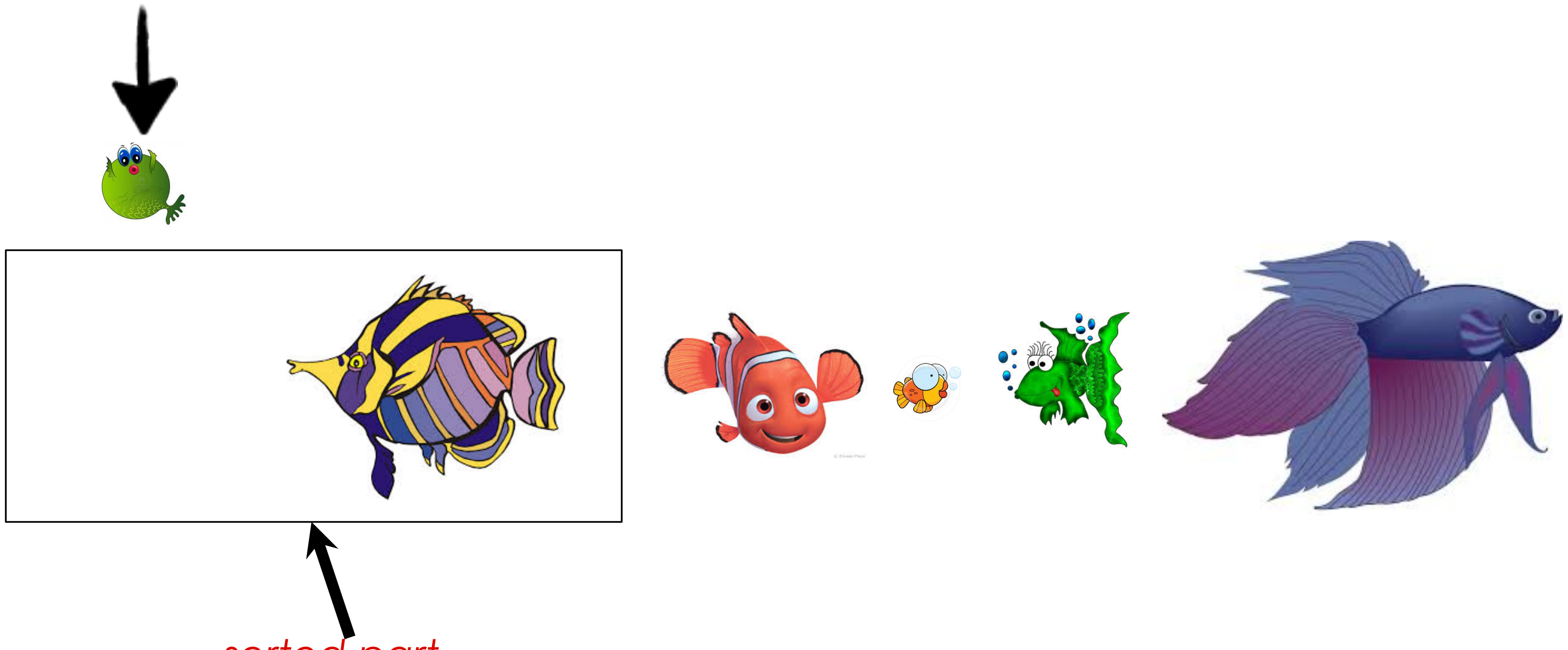
In-place Insertion Sort



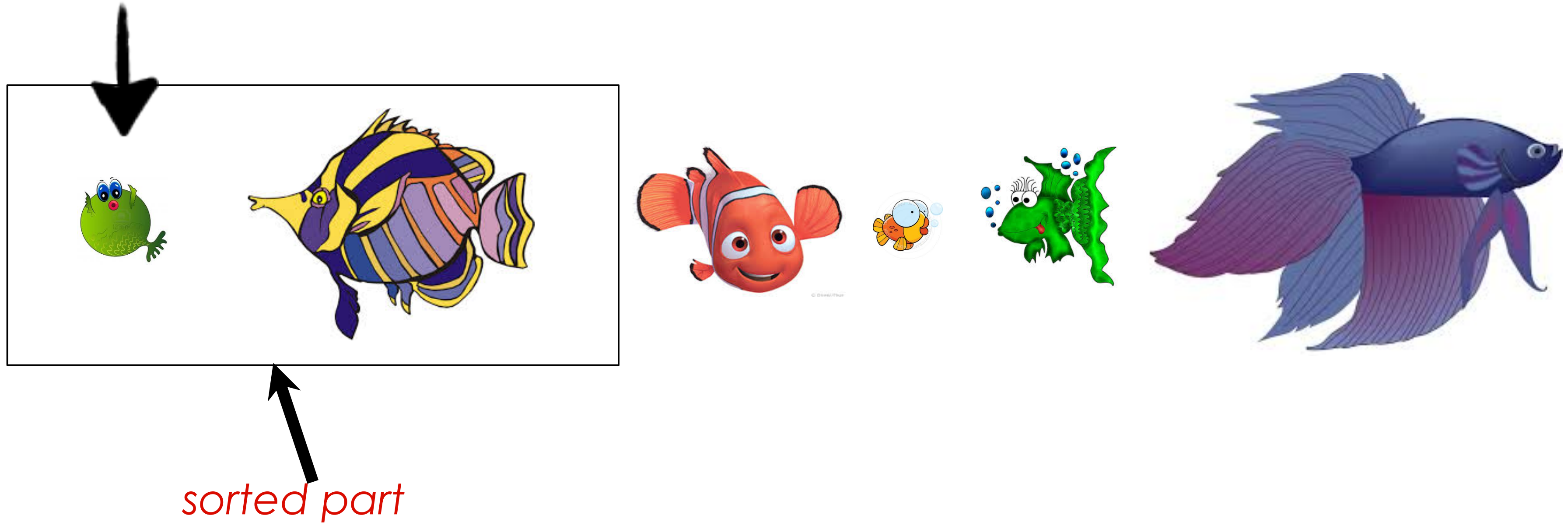
In-place Insertion Sort



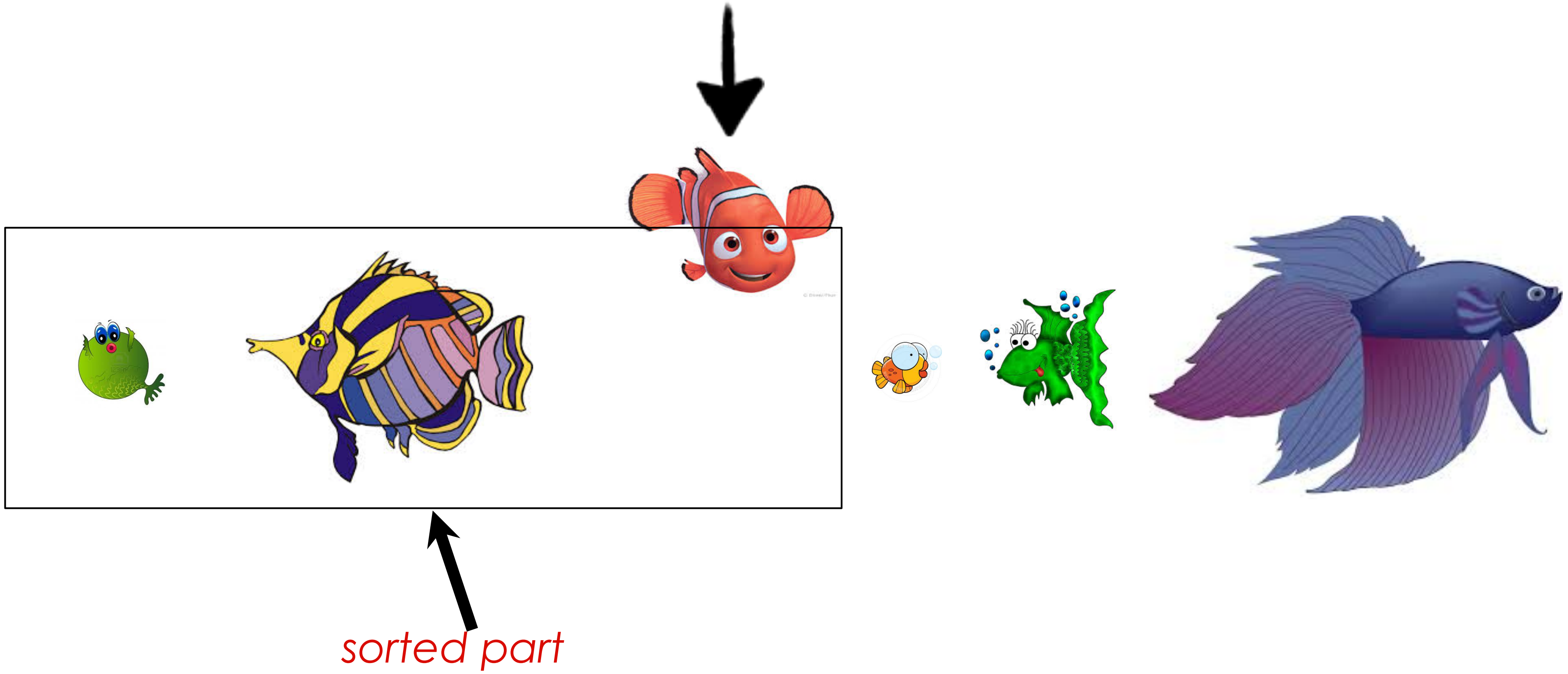
In-place Insertion Sort



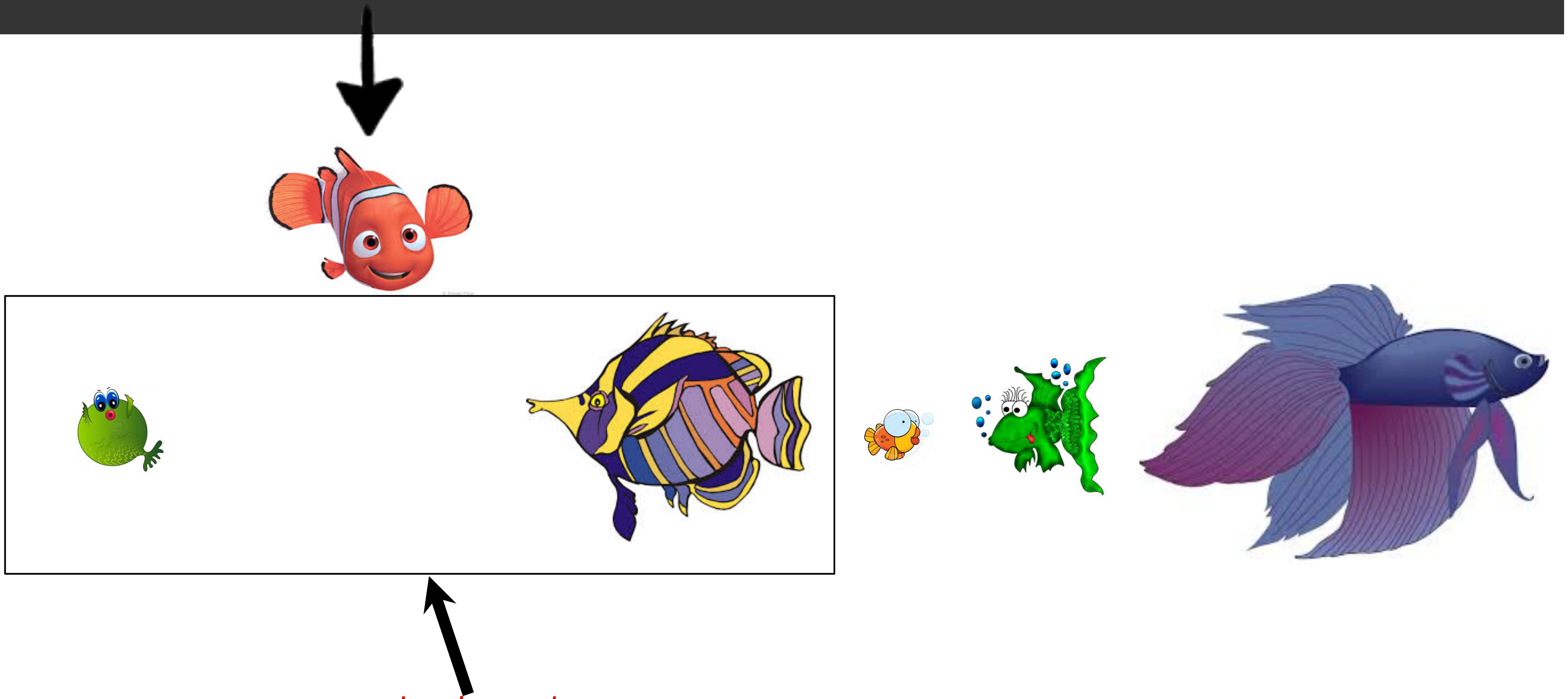
In-place Insertion Sort



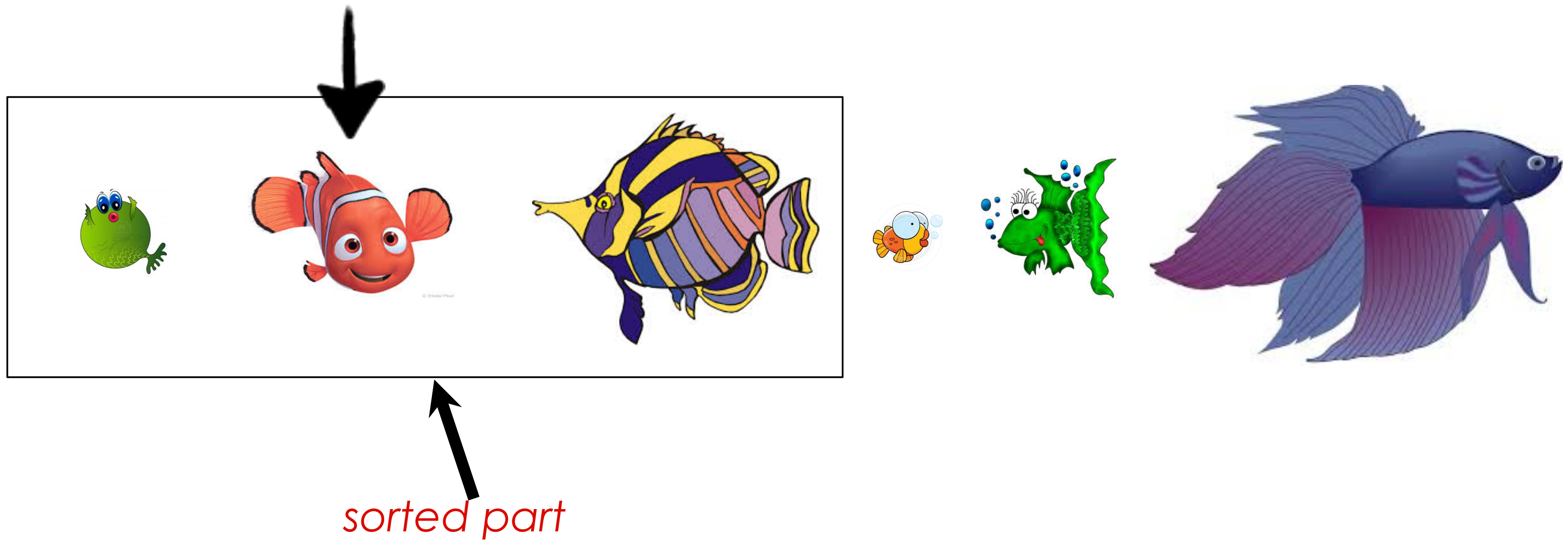
In-place Insertion Sort



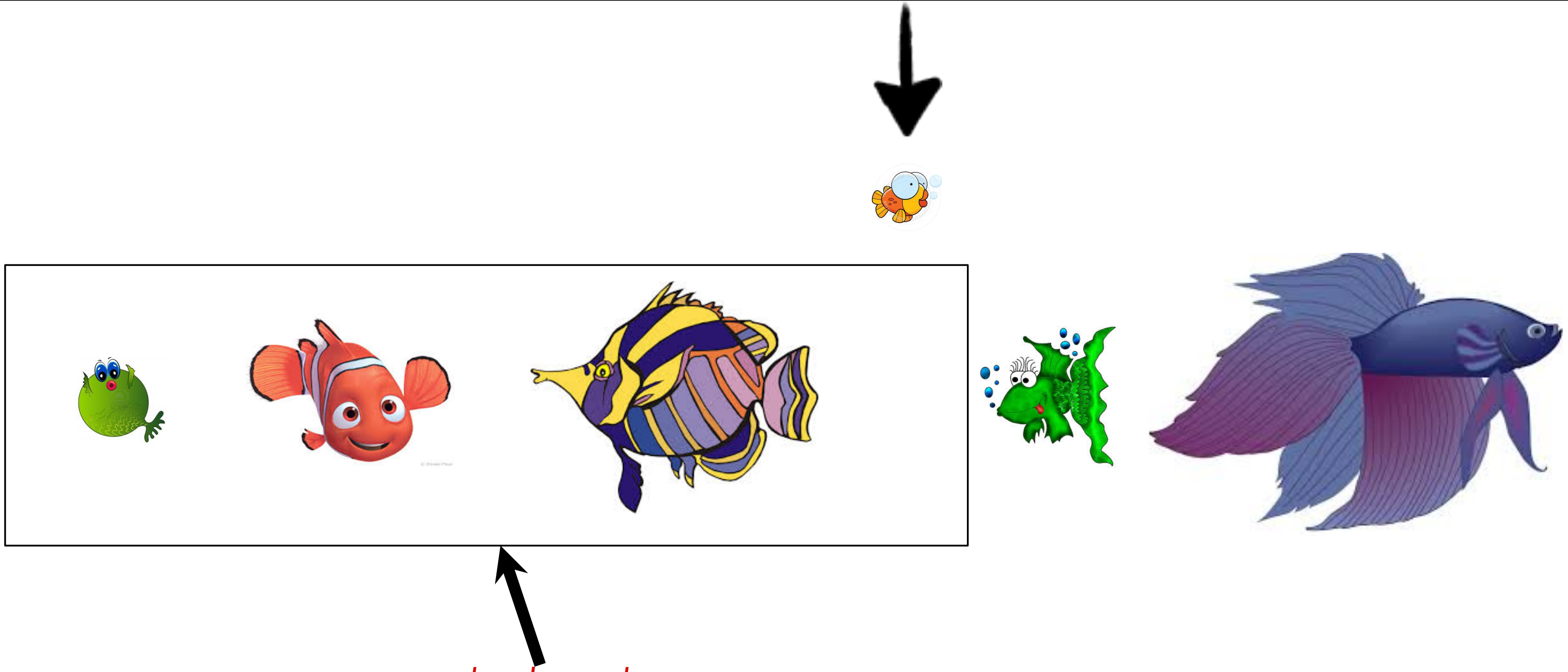
In-place Insertion Sort



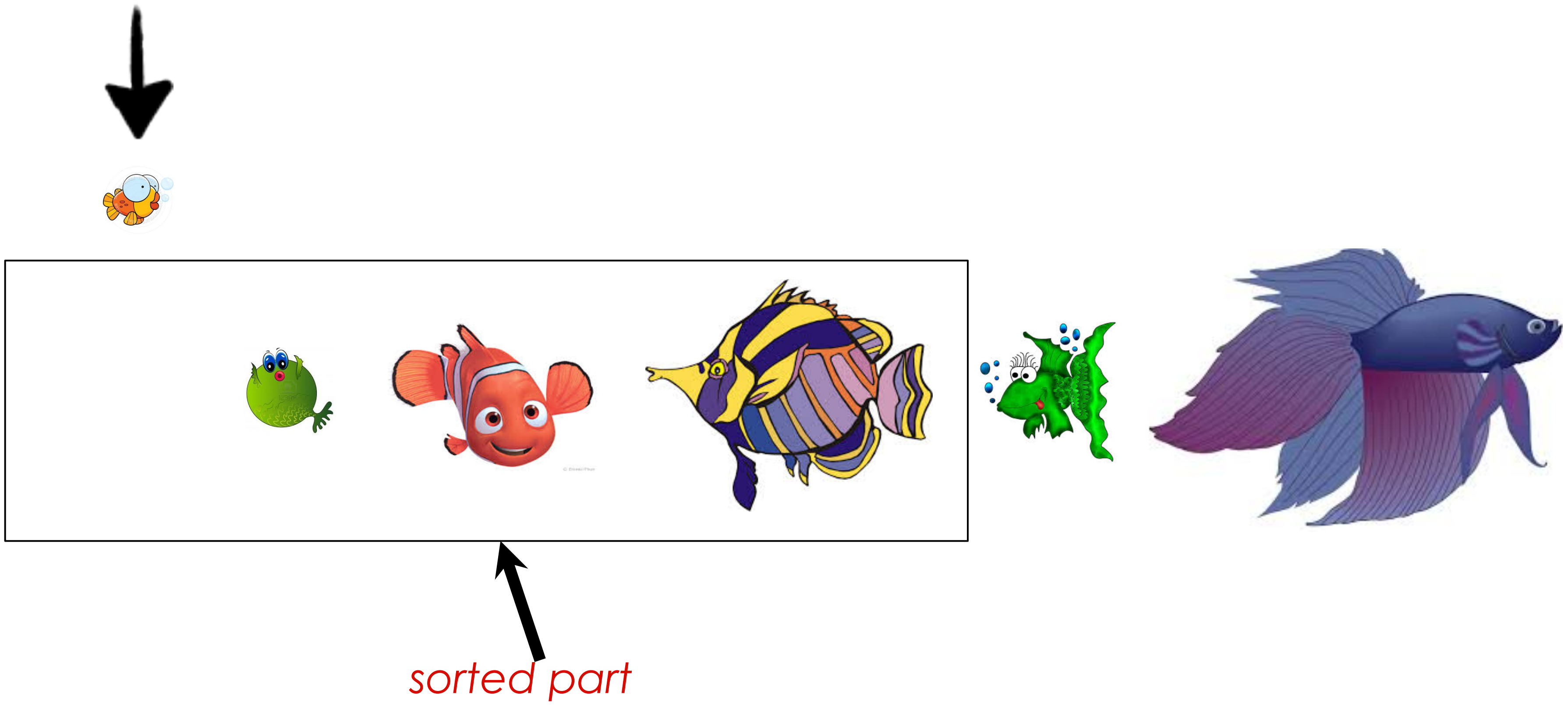
In-place Insertion Sort



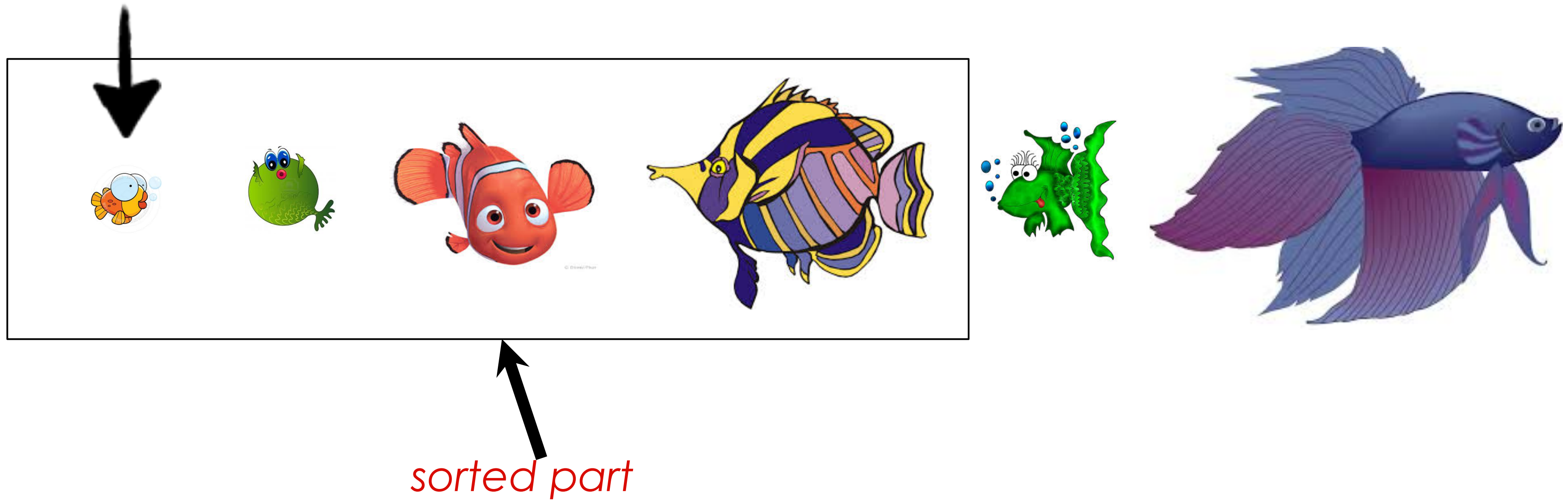
In-place Insertion Sort



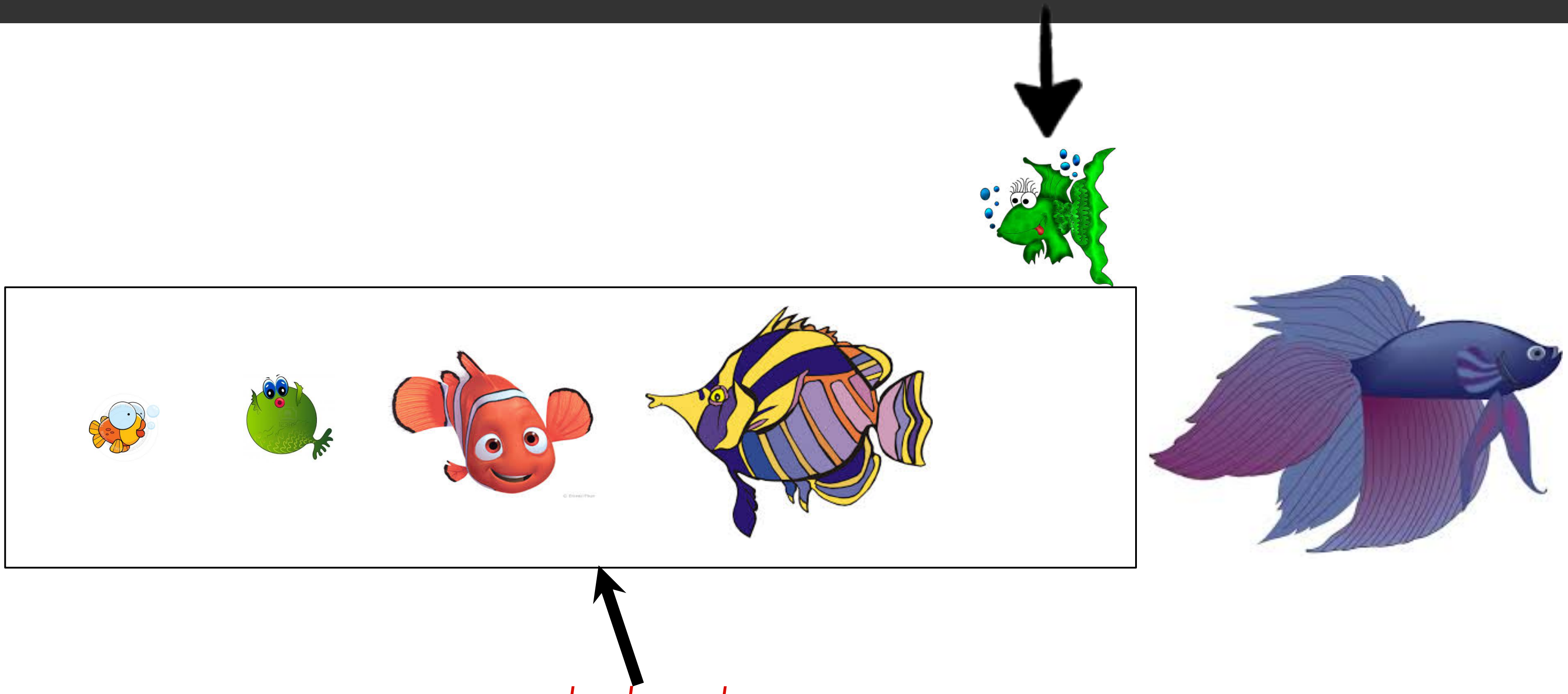
In-place Insertion Sort



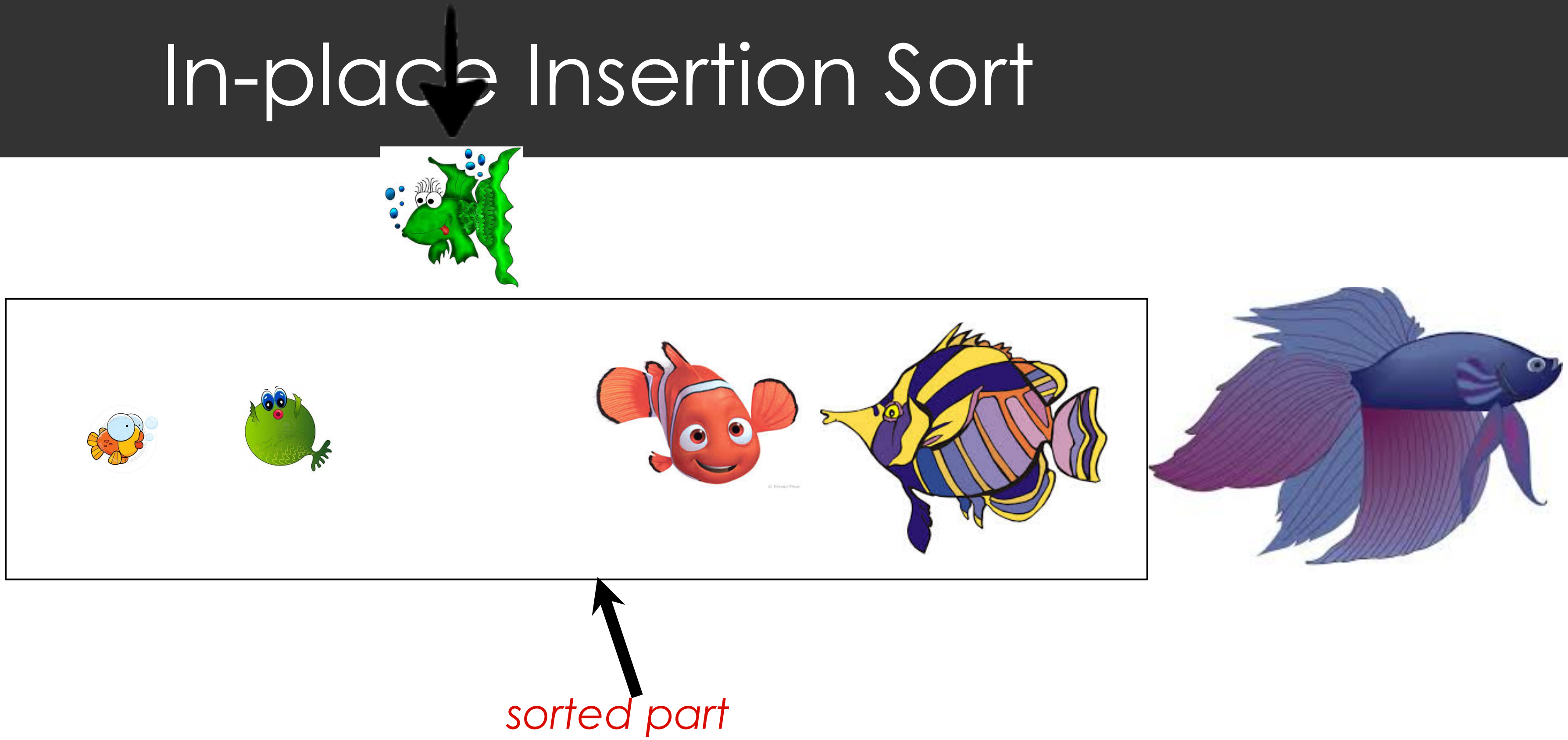
In-place Insertion Sort



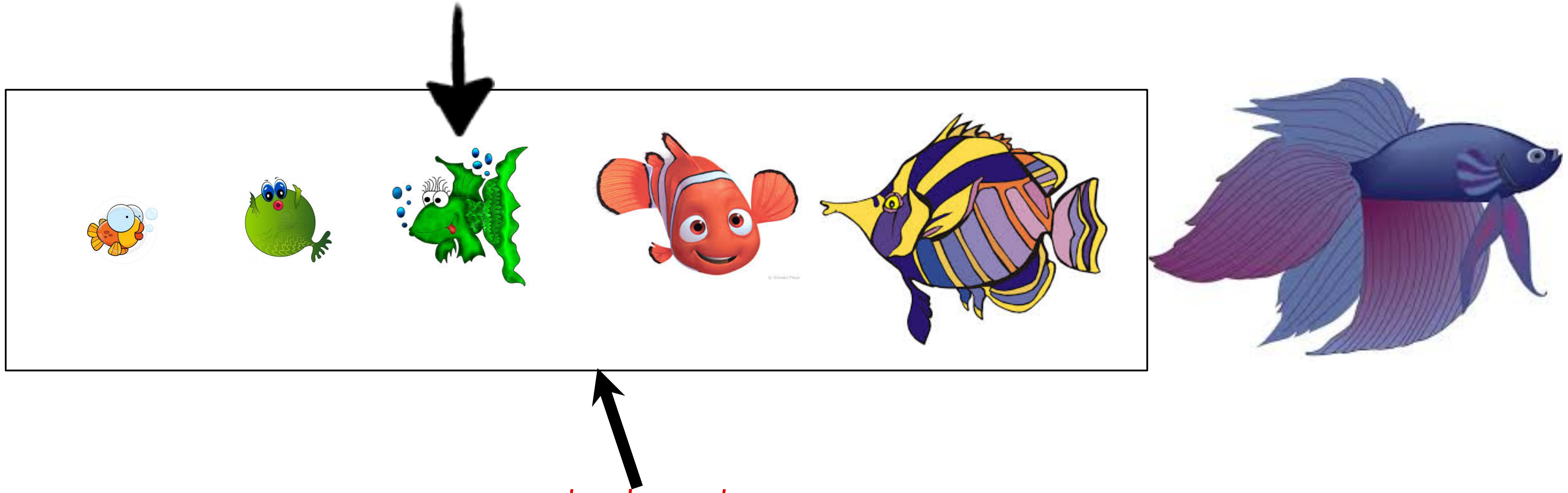
In-place Insertion Sort



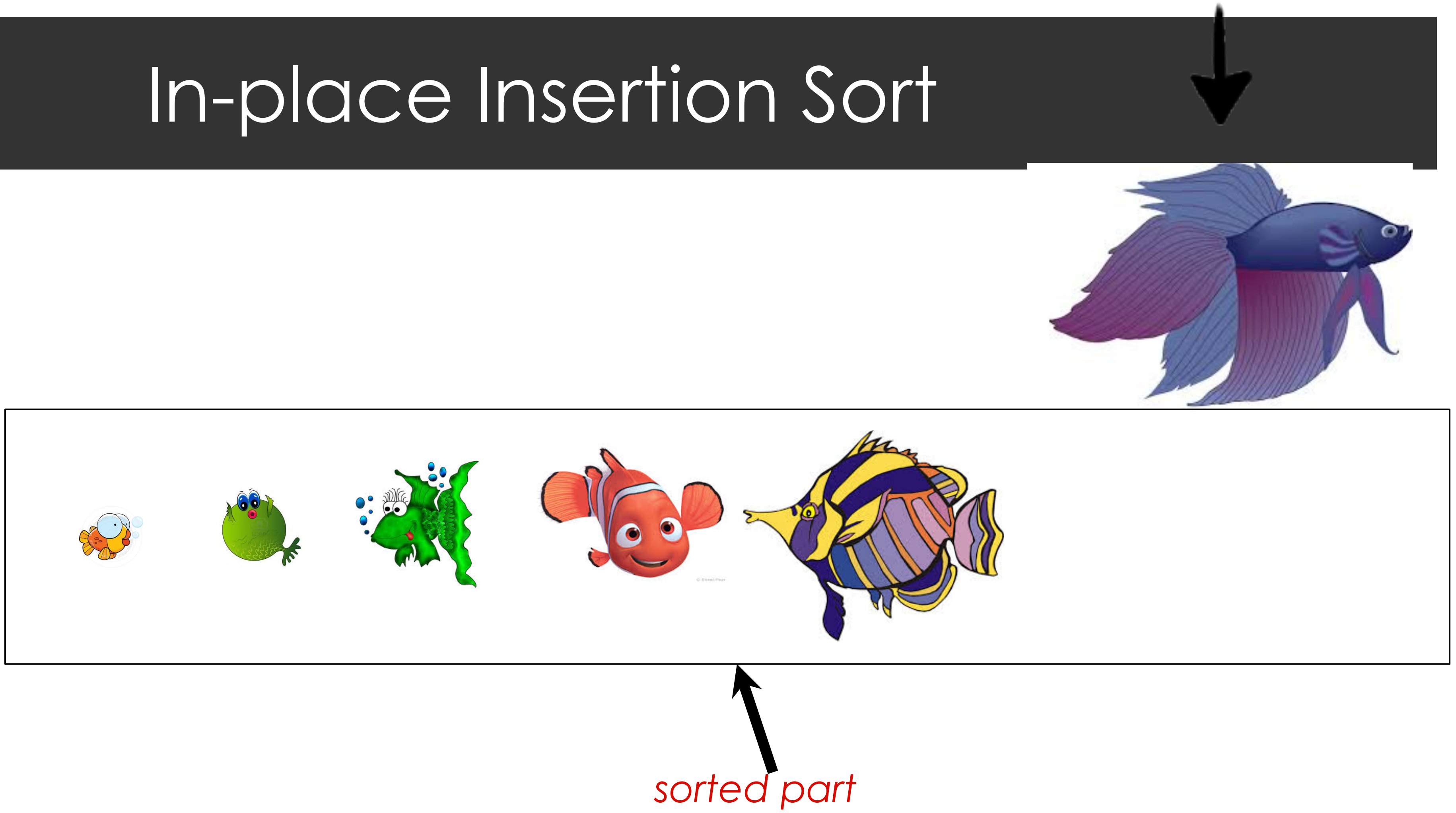
In-place Insertion Sort



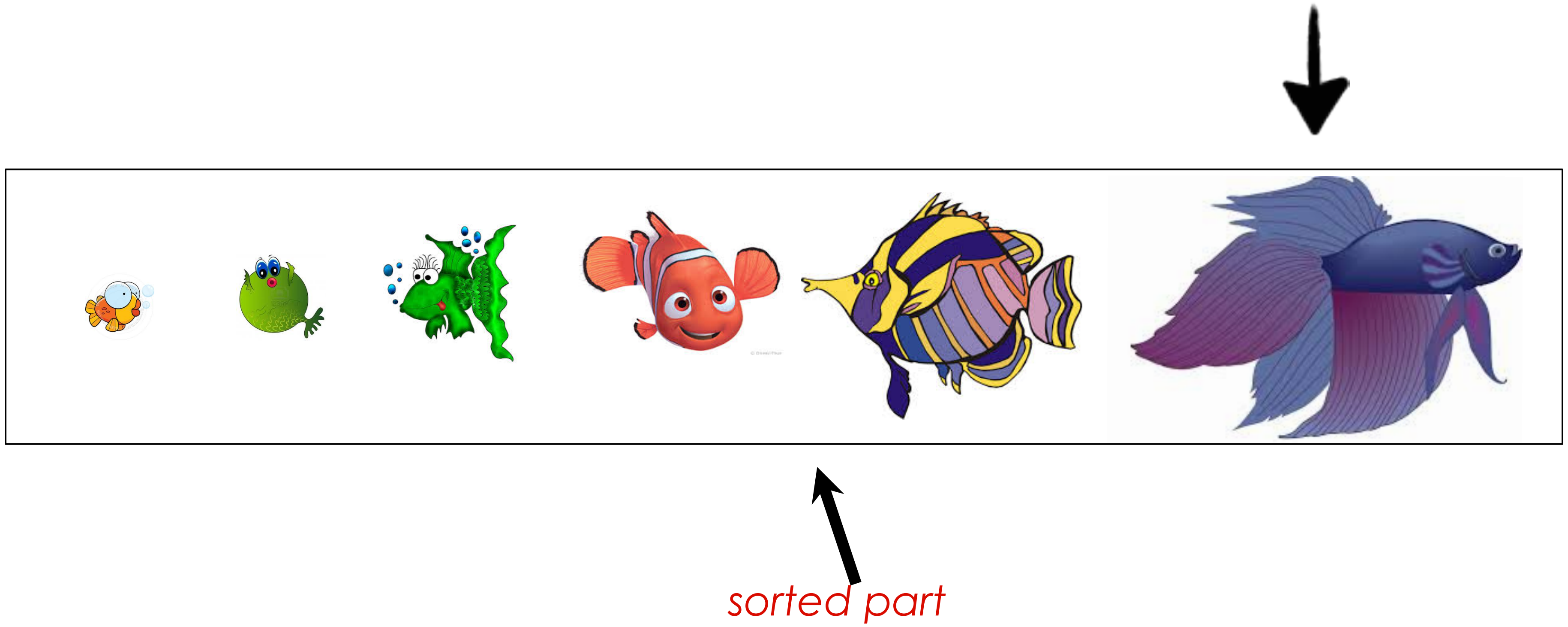
In-place Insertion Sort



In-place Insertion Sort



In-place Insertion Sort



In-place Insertion Sort 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
```

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

But now we have to write the move_left function!

-

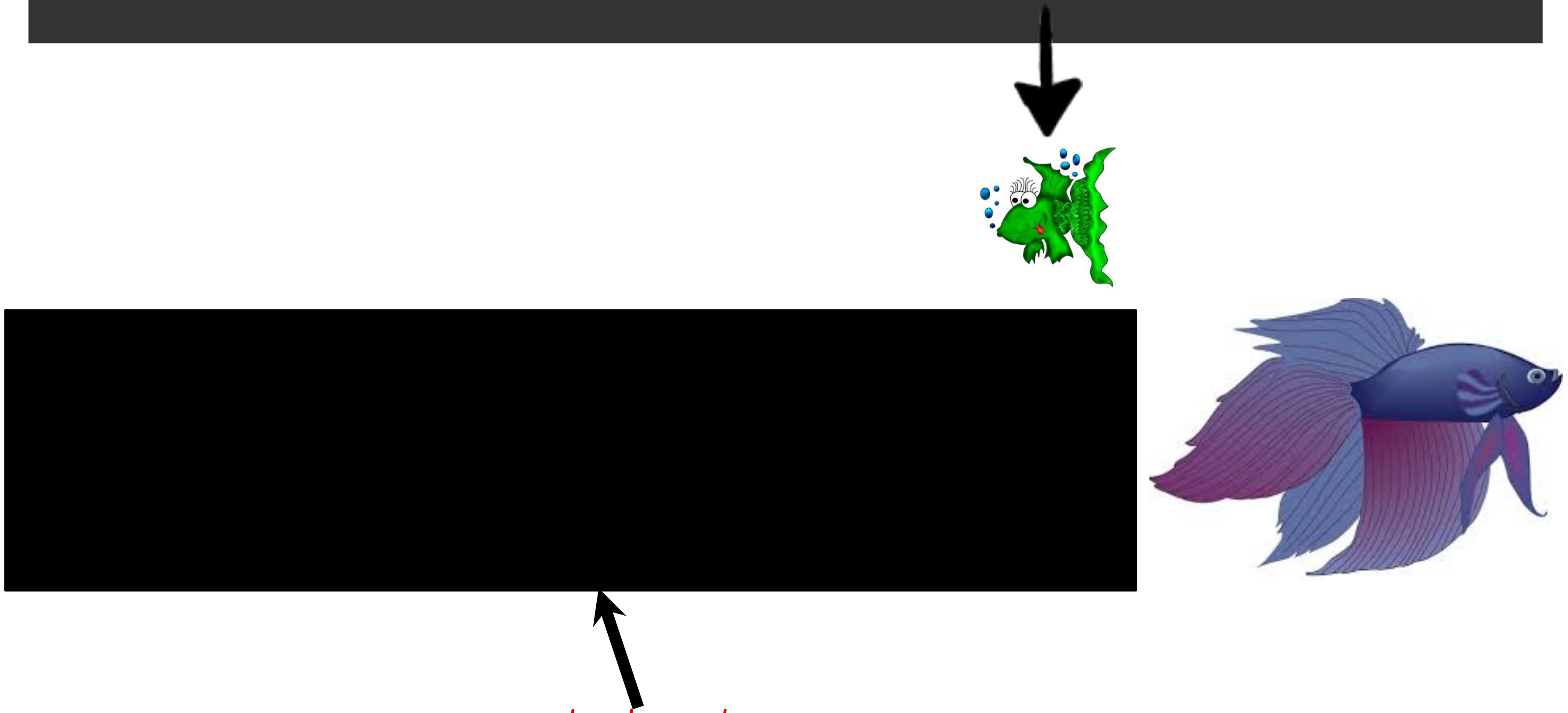
Moving left using search

To move the element x at index i “left” to its correct position, remove it, 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.

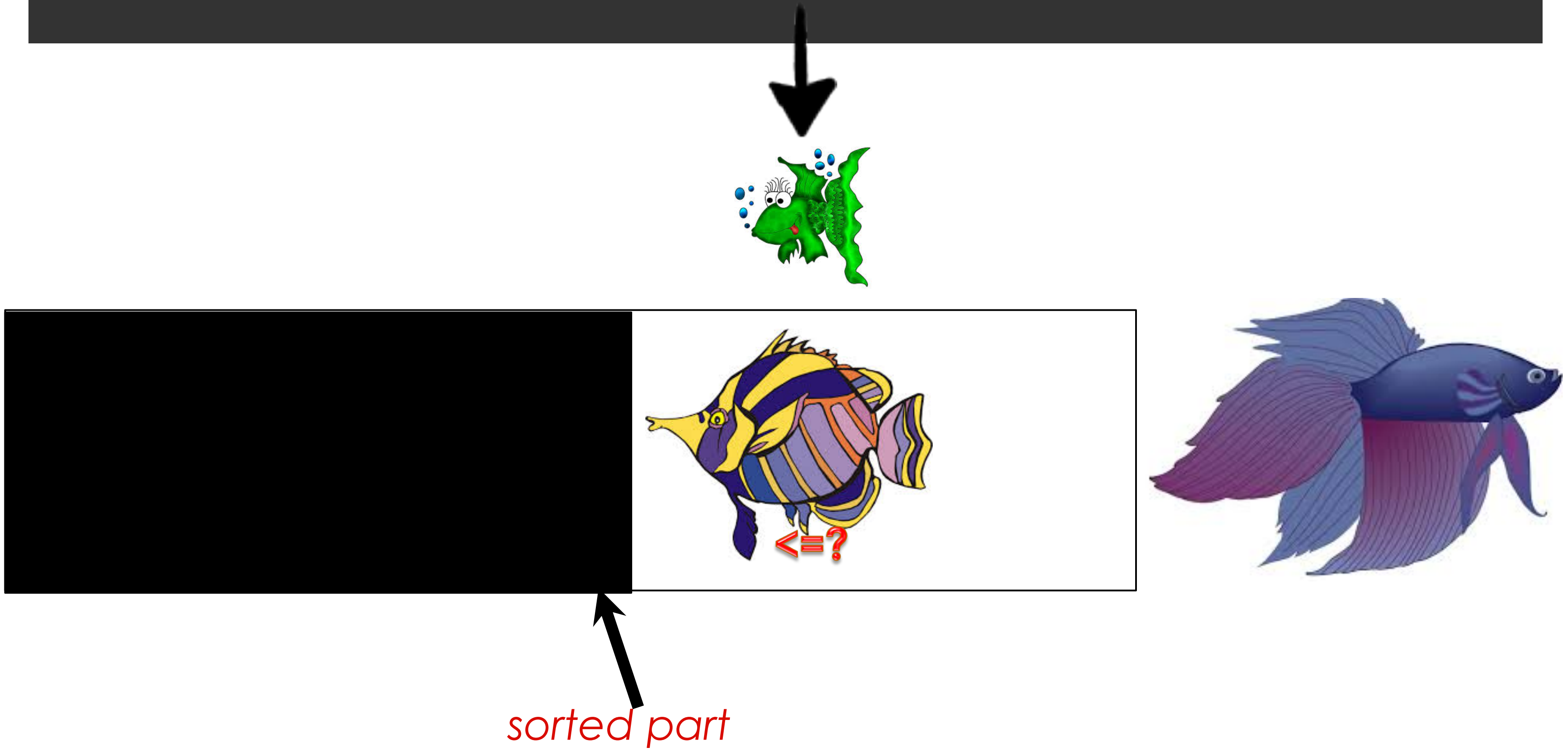
Then insert x back into the list to the right of that element.

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

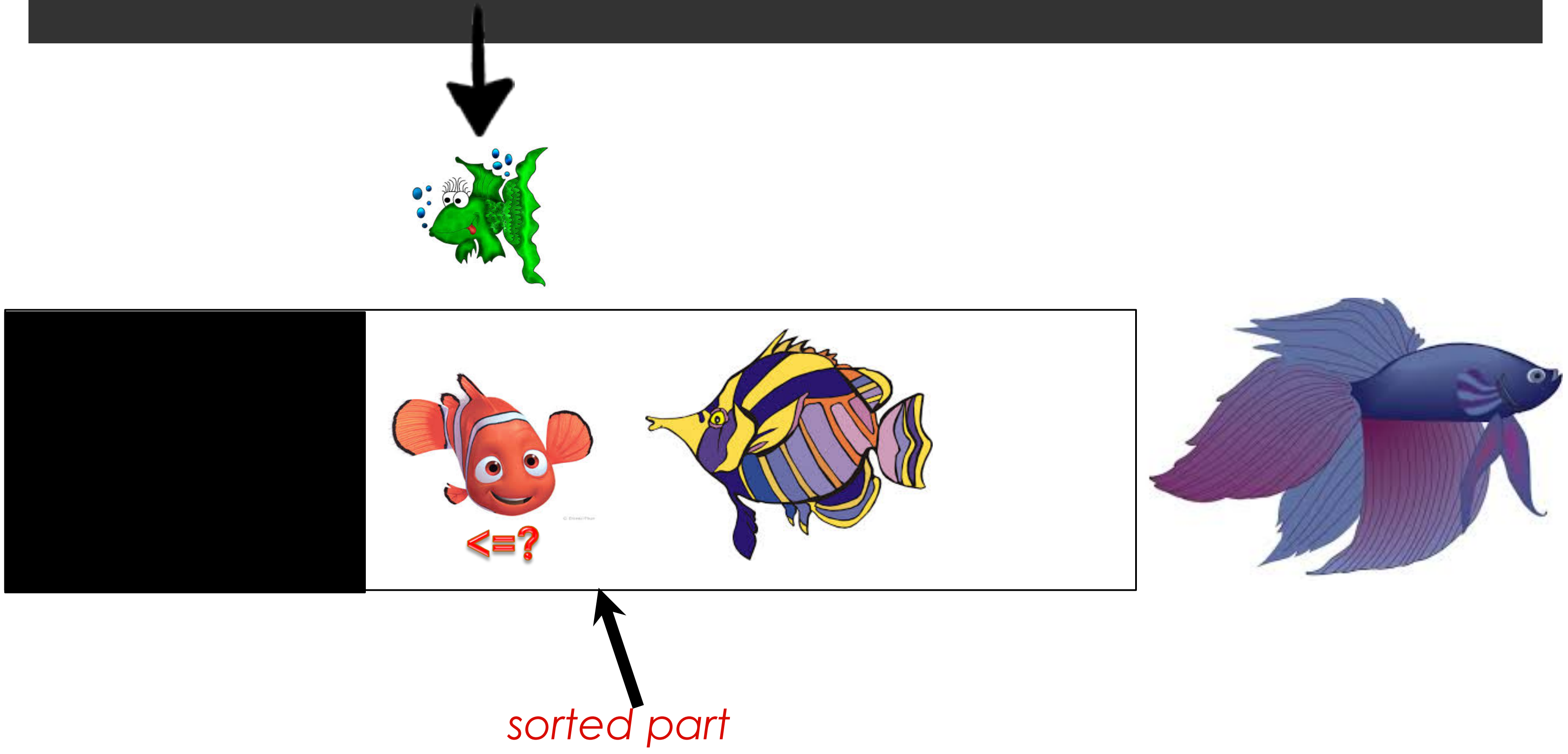
move_left via linear search



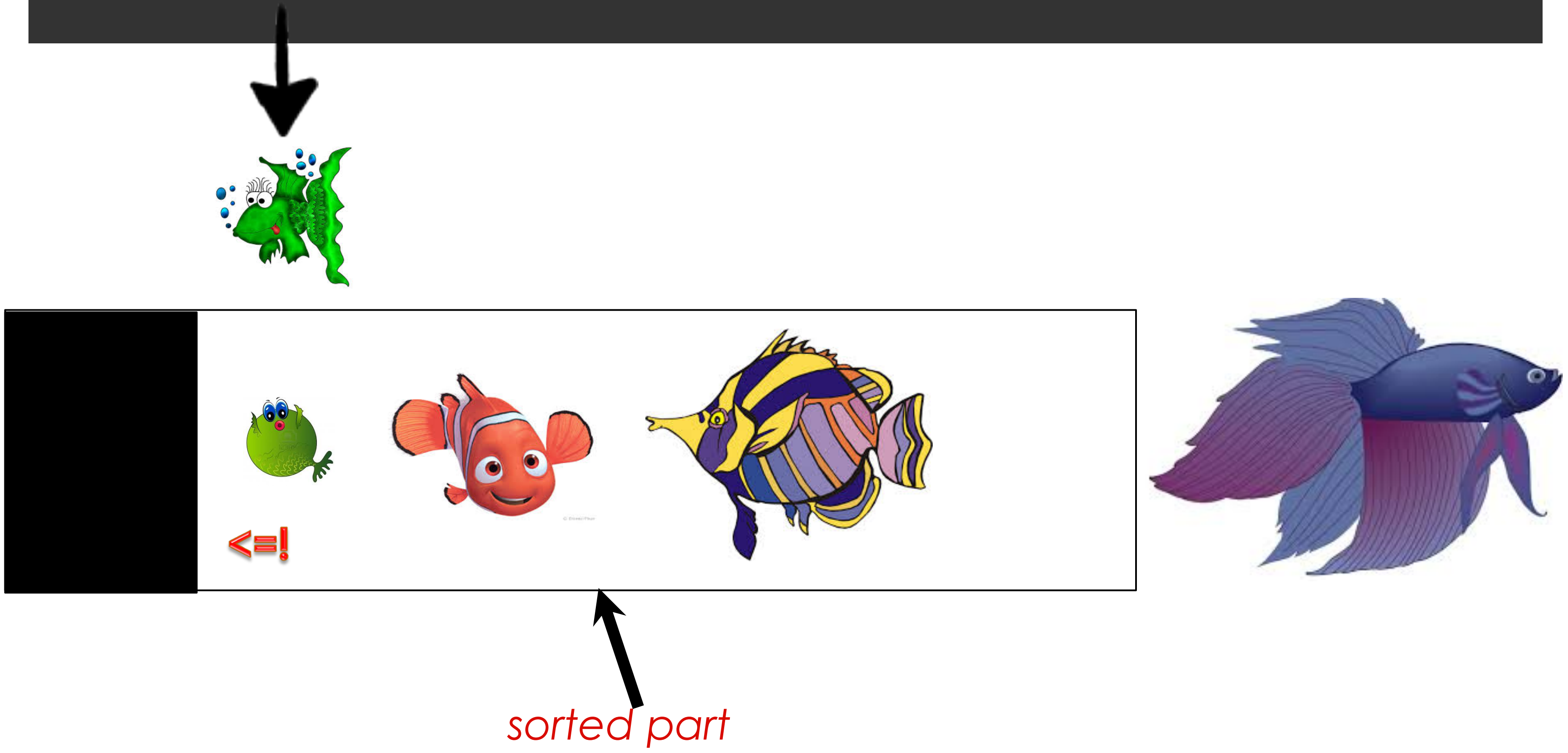
move_left via linear search



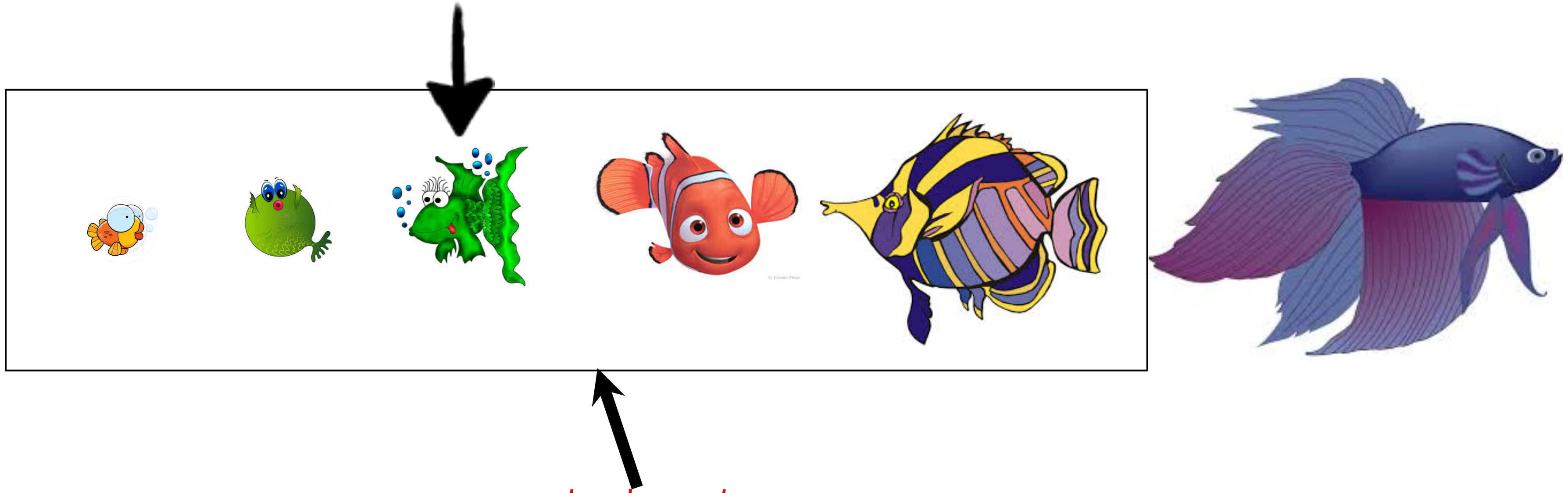
move_left via linear search



move_left via linear search



In-place Insertion Sort



Moving left (numbers)

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

From algorithm to code

- Our algorithm says to "remove" and "insert" elements of a list.
- But how do we do that?
- Fortunately 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

```
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)
```

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

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

insert x at position j+1 of list, shifting elements j+1 and beyond

Insertion sort with a bug

```
def move_left(items, i):
    # Insert the element at items[i] into its 
place
    x = items.pop(i)
    j = i - 1
    while j > 0 and items[j] > x:
        j = j - 1
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

Why should we believe our code works?

- We can test it:

```
>>> data = [13, 78, 18, 25, 100, 89, 12]
>>> isort(data)
[13, 12, 18, 25, 78, 89, 100]
>>>
```

- Hmmmm. What went wrong?

Using `assert` to debug

- What do we know has to be true for move_left to do the right thing?
- We have a loop that decreases j and checks for an element at index j smaller than or equal to x. **When should it stop looping?**
  - When the value of j is -1,
  - or when the item at index j is <= x
  - `j == -1 or items[j] <= x`

So add an assertion to the code

```
def move_left(items, i):
    # Insert the element at items[i] into its
place
    x = items.pop(i)
    j = i - 1
    while j > 0 and items[j] > x:
        j = j – 1
    assert(j == -1 or items[j] <= x)
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

Run the same test again

```
>>> data = [13, 78, 18, 25, 100, 89, 12]
>>> isort(data)
[13,  12, 18, 25, 78, 89, 100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "isort.py", line 16, in isort
    move_left(items, i)
  File "isort.py", line 7, in move_left
    assert(j == -1 or items[j] <= x)
AssertionError
```

This tells us we did something wrong with the loop!

Where's the bug?

```
def move_left(items, i):
    # Insert the element at items[i] into its
place
    x = items.pop(i)
    j = i - 1
    while j > 0 and items[j] > x:
        j = j – 1
    assert(j == -1 or items[j] <= x)
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

FALSE!
Why?????

The fix

```
def move_left(items, i):
    # Insert the element at items[i] into its place
    x = items.pop(i)
    j = i - 1
    while j >= 0 and items[j] > x:
        j = j – 1
    assert(j == -1 or items[j] <= x)
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

Run the same test again

```
>>> data = [13, 78, 18, 25, 100, 89, 12]
>>> isort(data)
[12, 13, 18, 25, 78, 89, 100]
```

Hurray!

Do we know for sure that the program will always do the right thing now?