Iteration:
Sorting, Scalability, Big O Notation

Announcements

- Yesterday?
  - Lab 4
- Tonight
  - Lab 5
- Tomorrow
  - PS 4
  - PA 4

Yesterday

- Quick Review: Sieve of Eratosthenes
- Character Comparisons (Unicode)
- Linear Search
- Sorting

Today

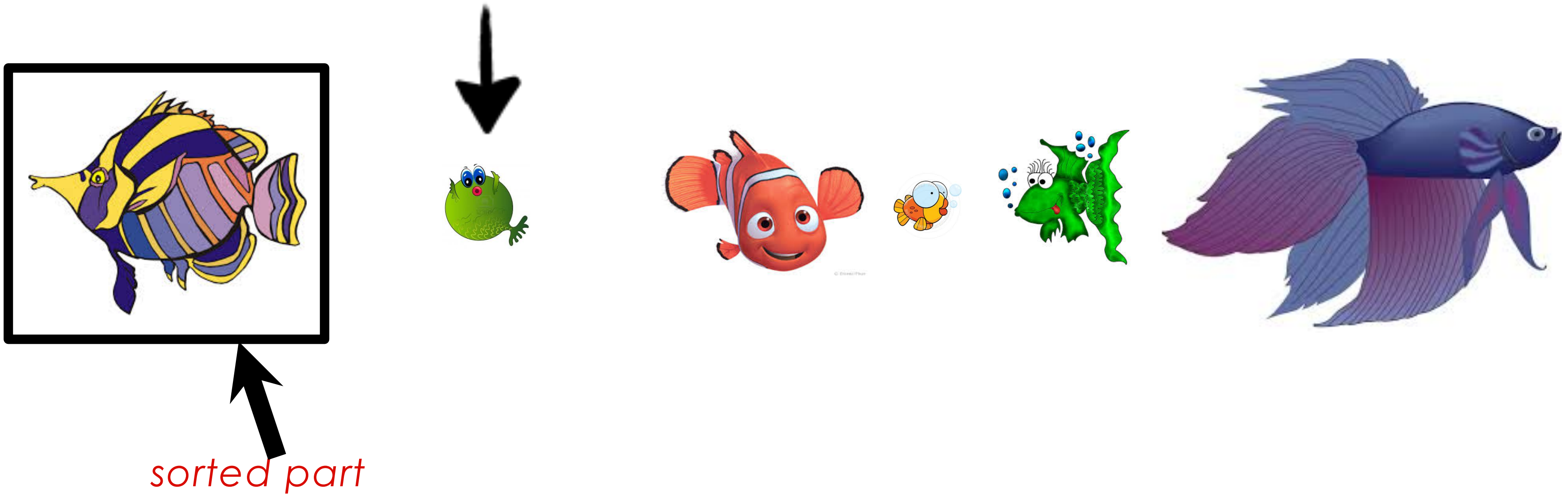
- Review: Insertion Sort
- Scalability
- Big O Notation

Sorting

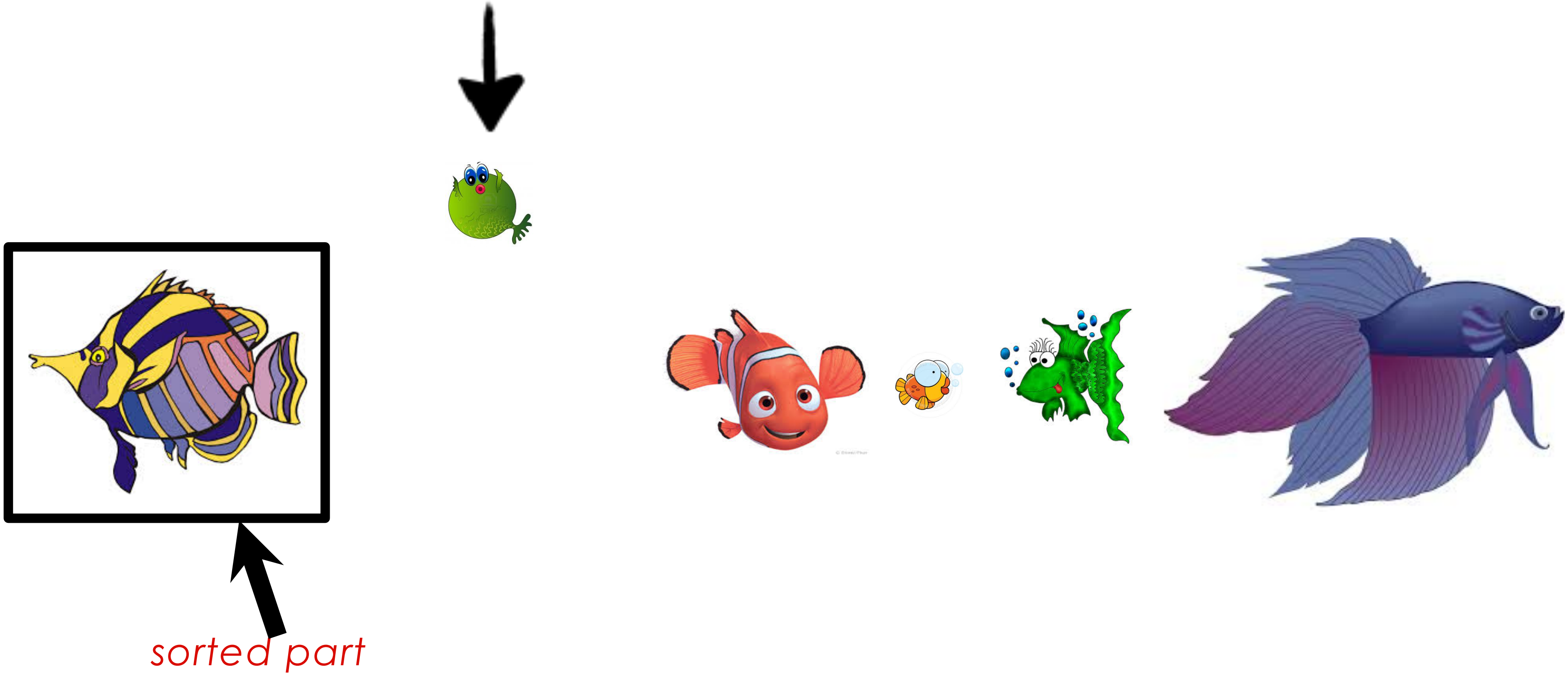
In-place Insertion Sort

- 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.

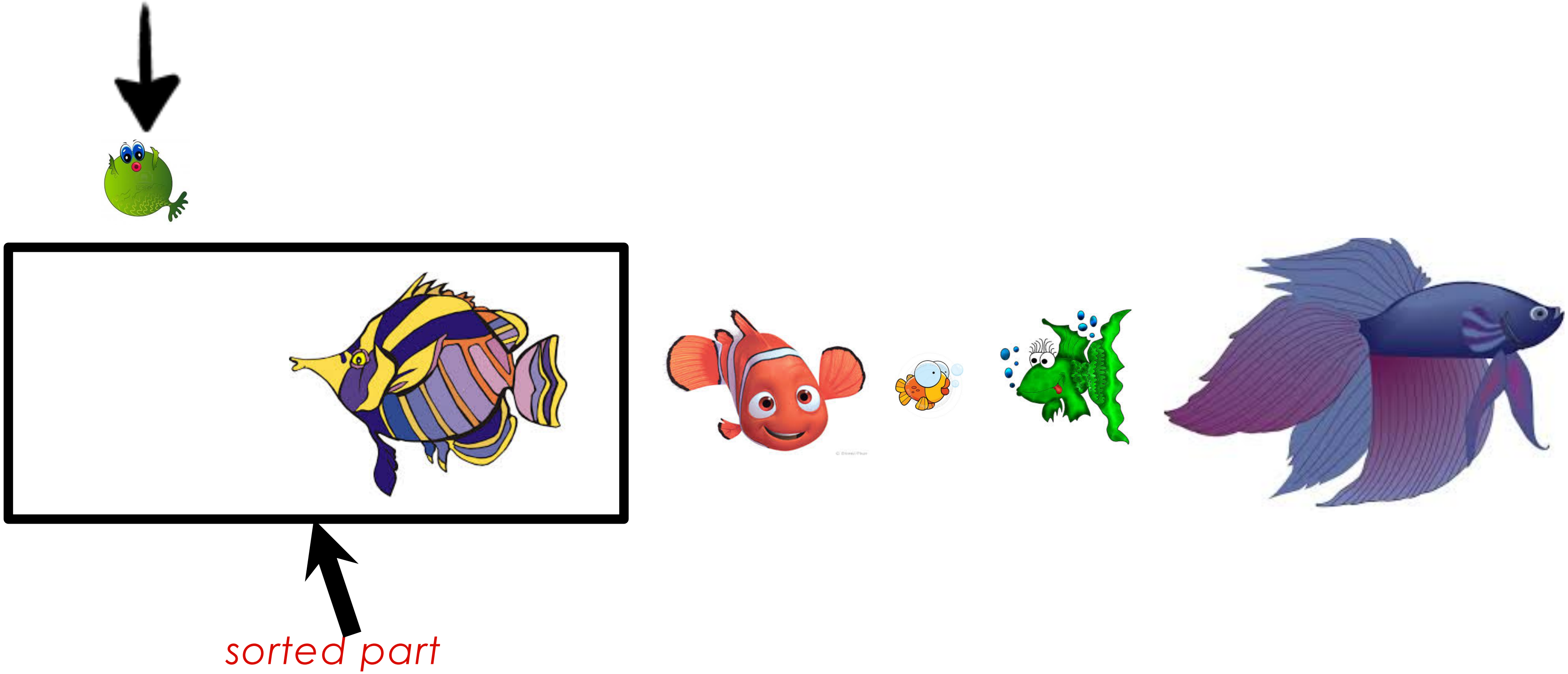
In-place Insertion Sort



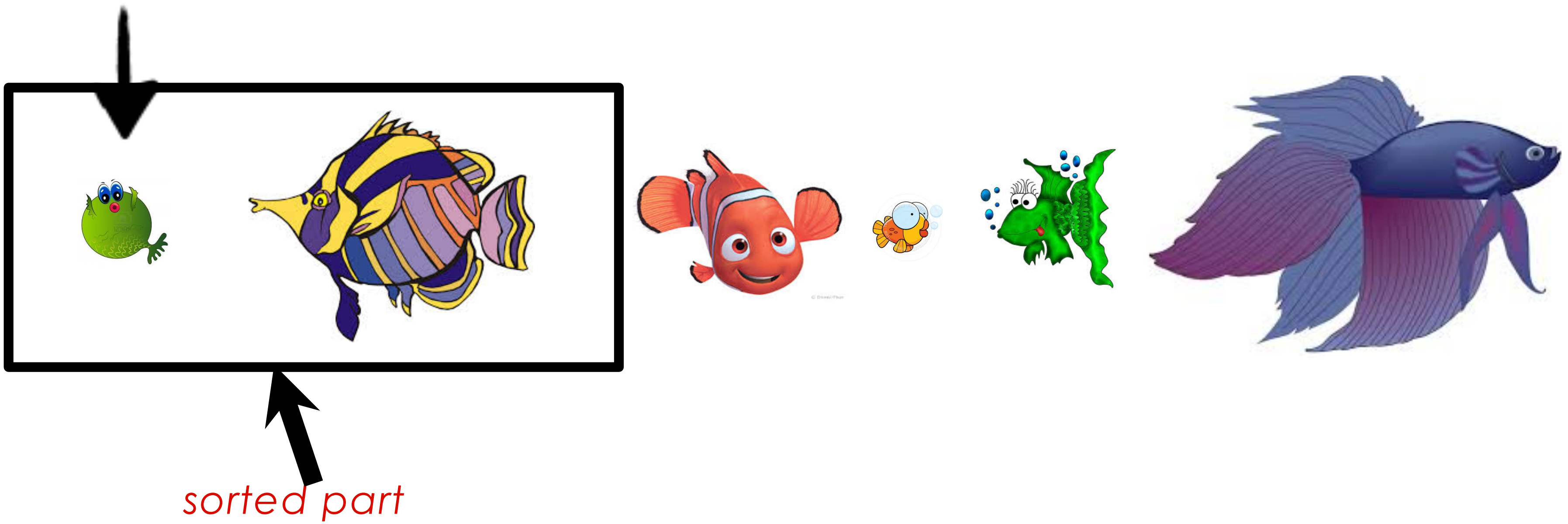
In-place Insertion Sort



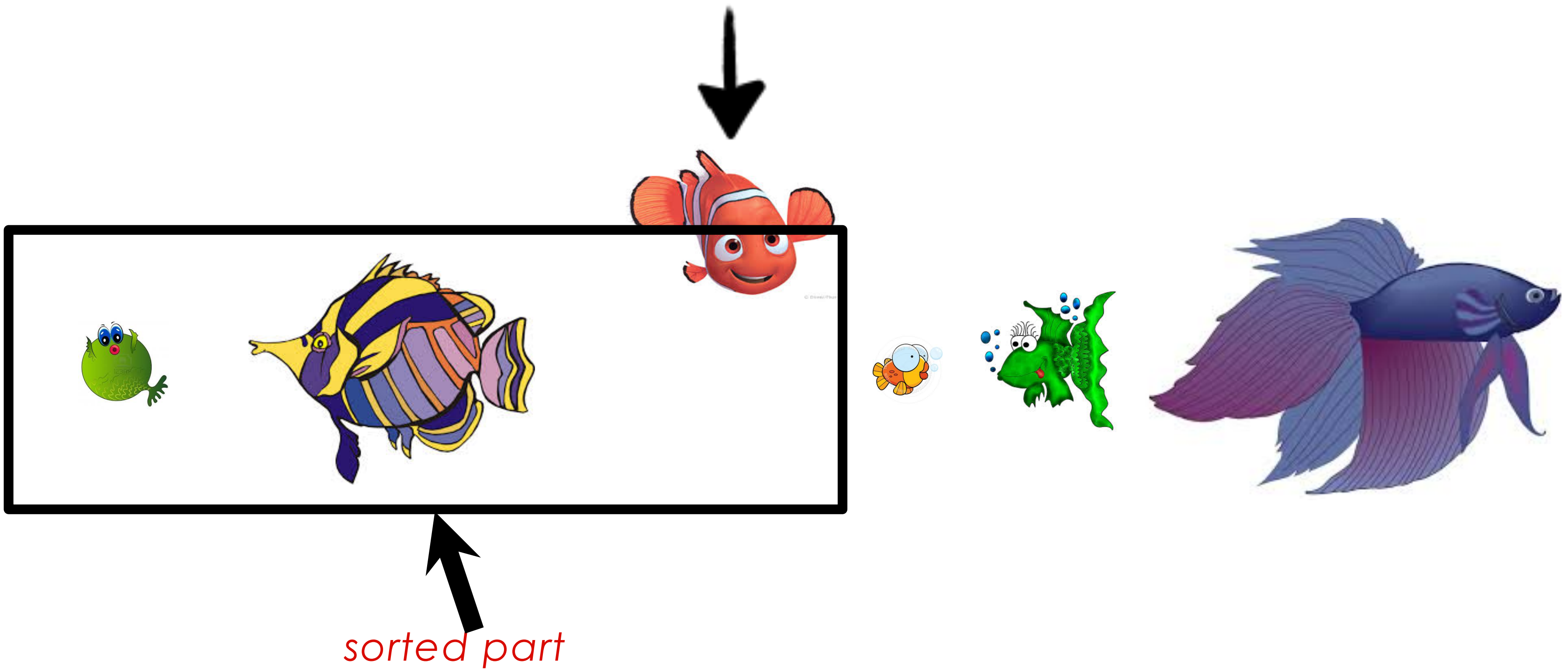
In-place Insertion Sort



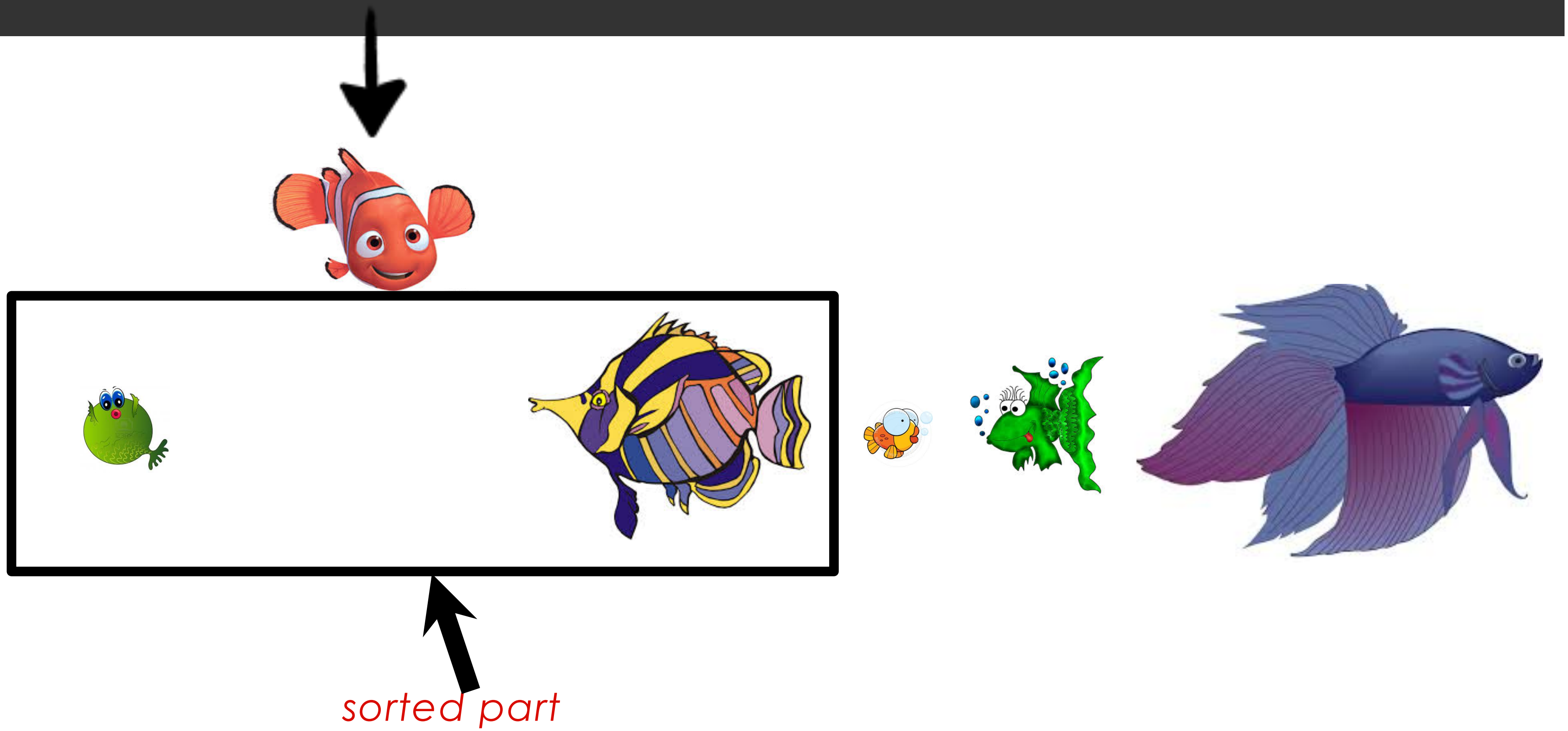
In-place Insertion Sort



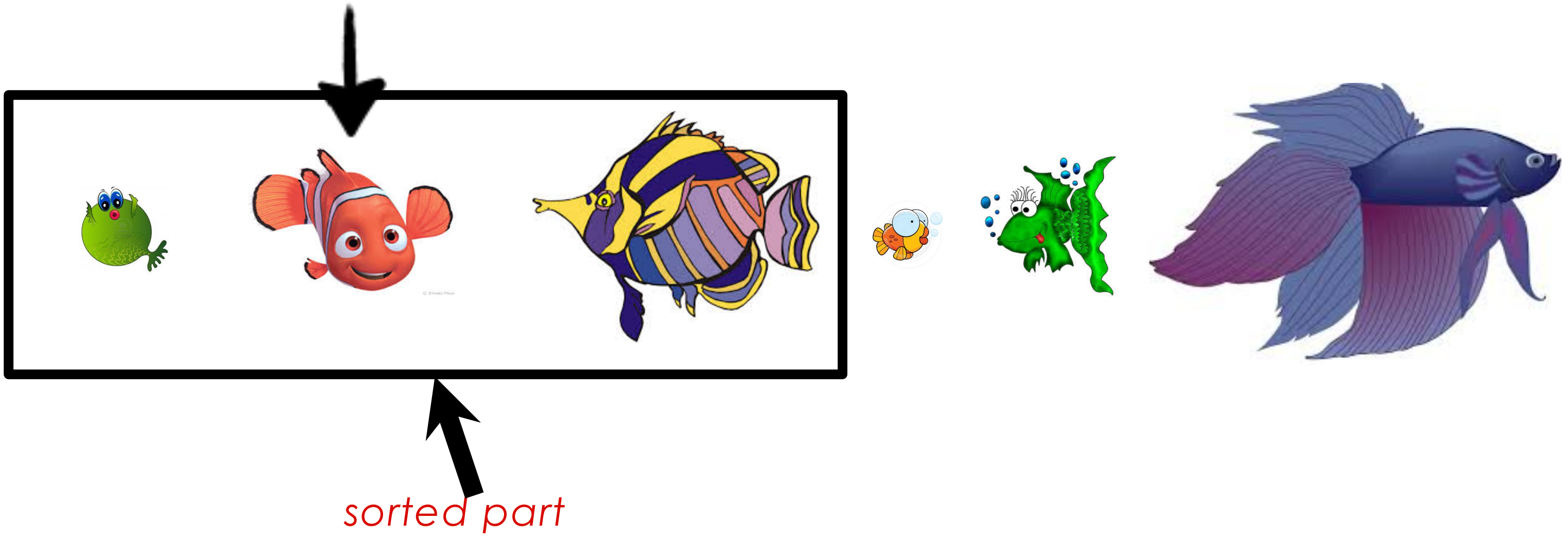
In-place Insertion Sort



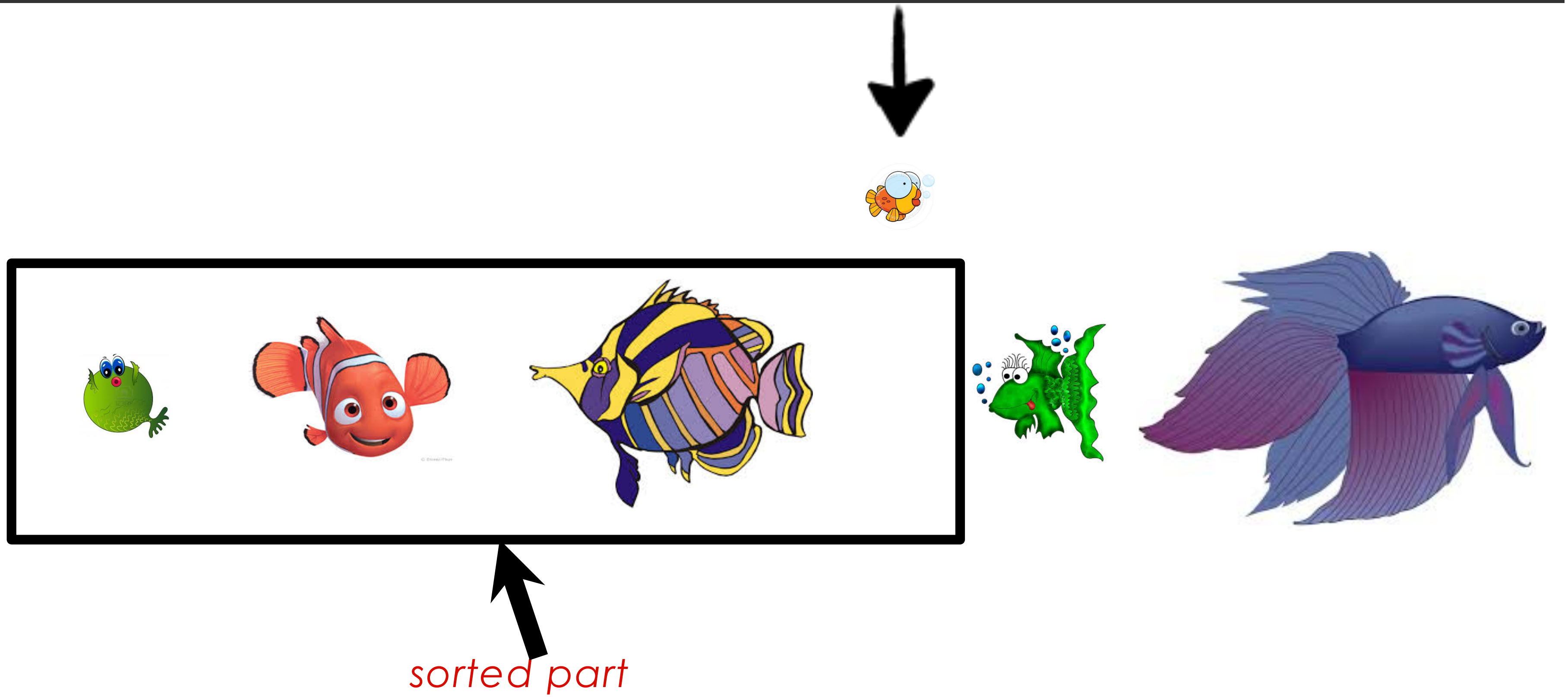
In-place Insertion Sort



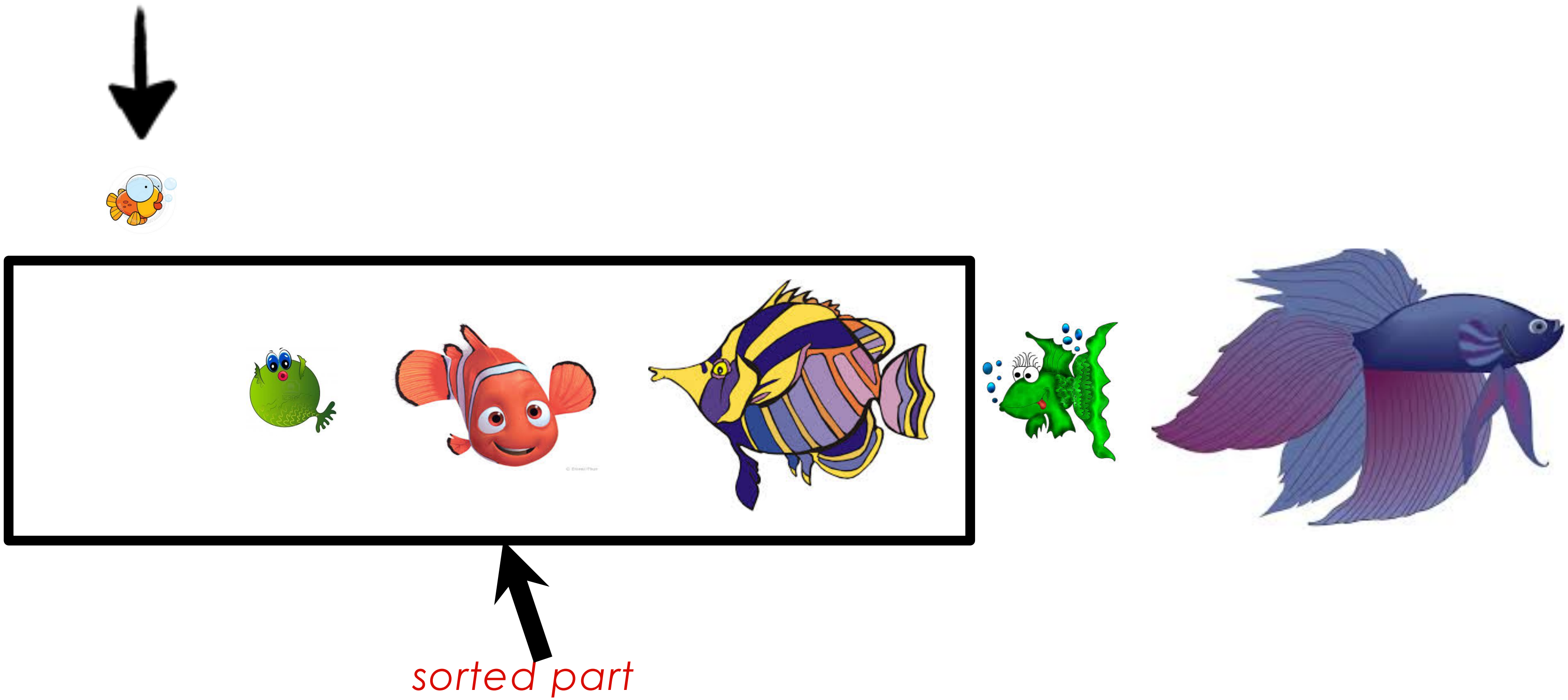
In-place Insertion Sort



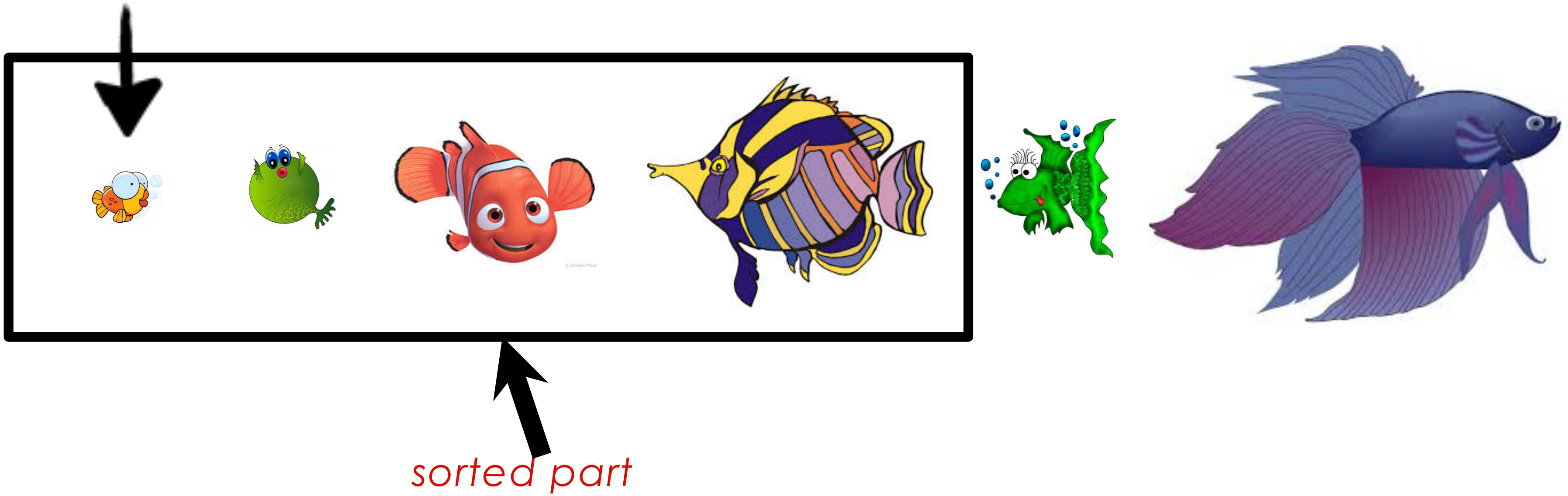
In-place Insertion Sort



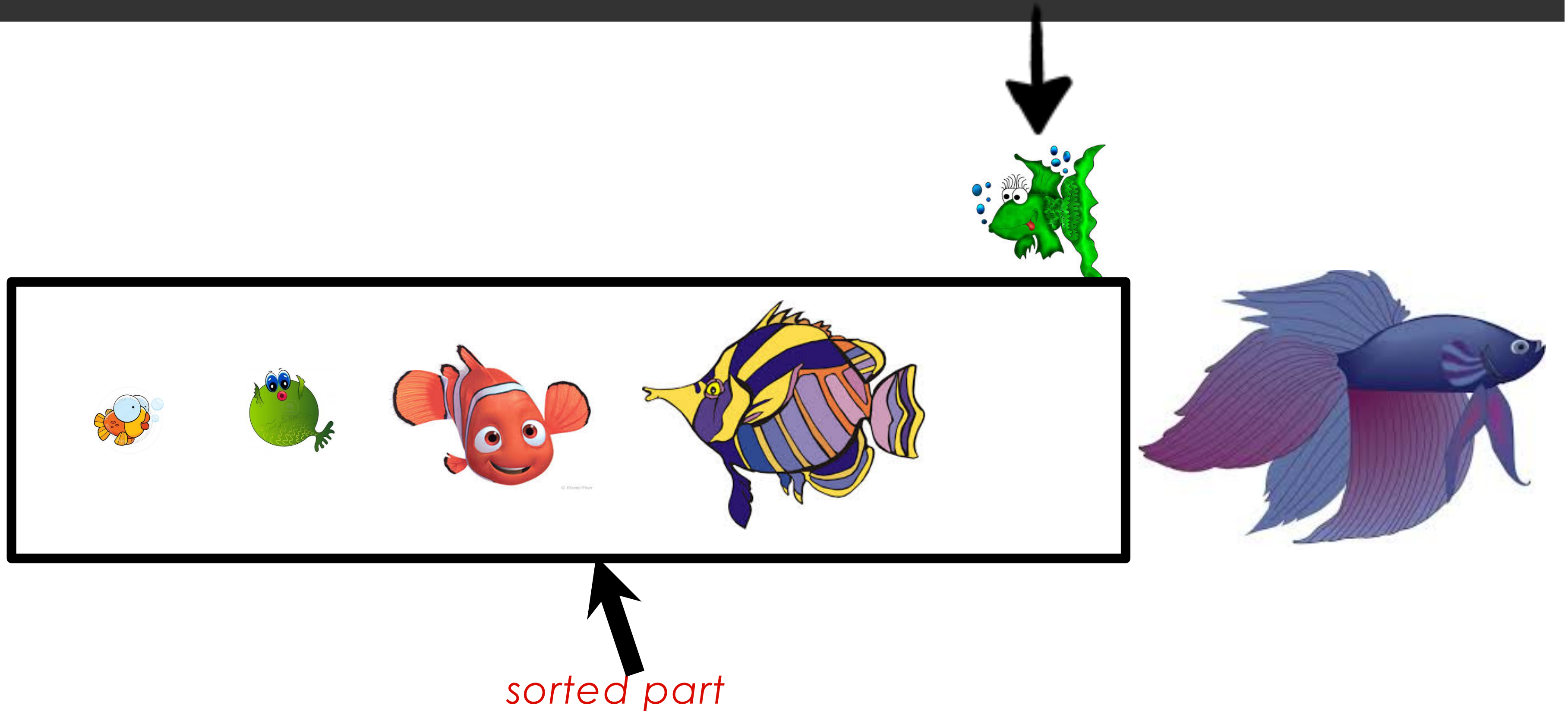
In-place Insertion Sort



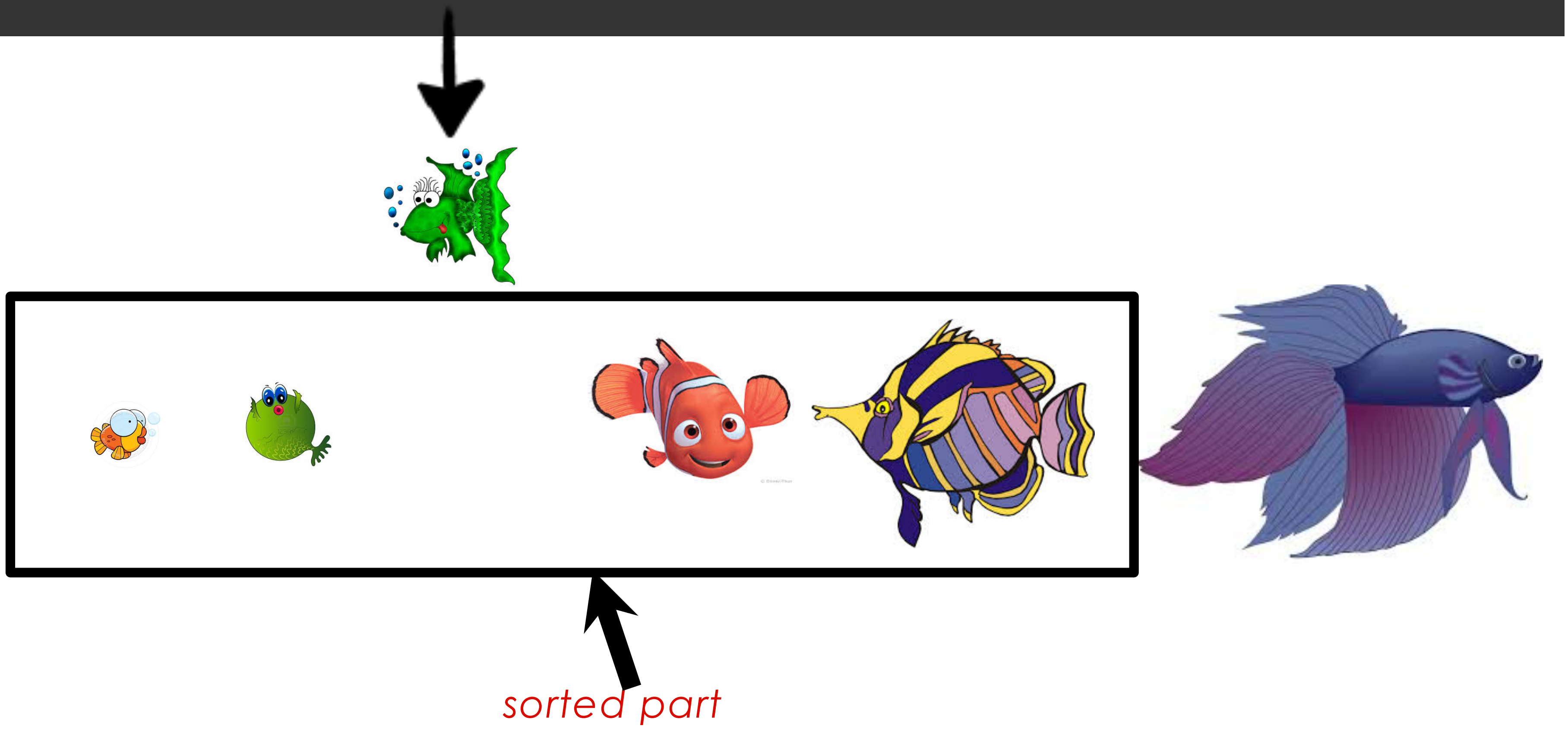
In-place Insertion Sort



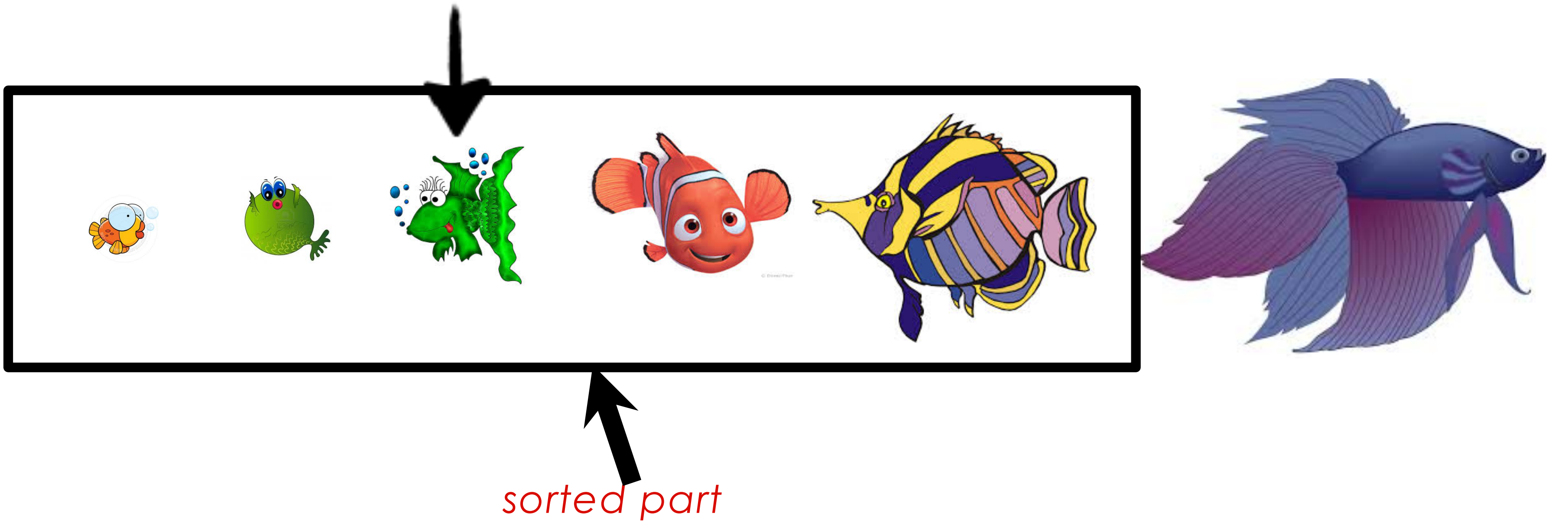
In-place Insertion Sort



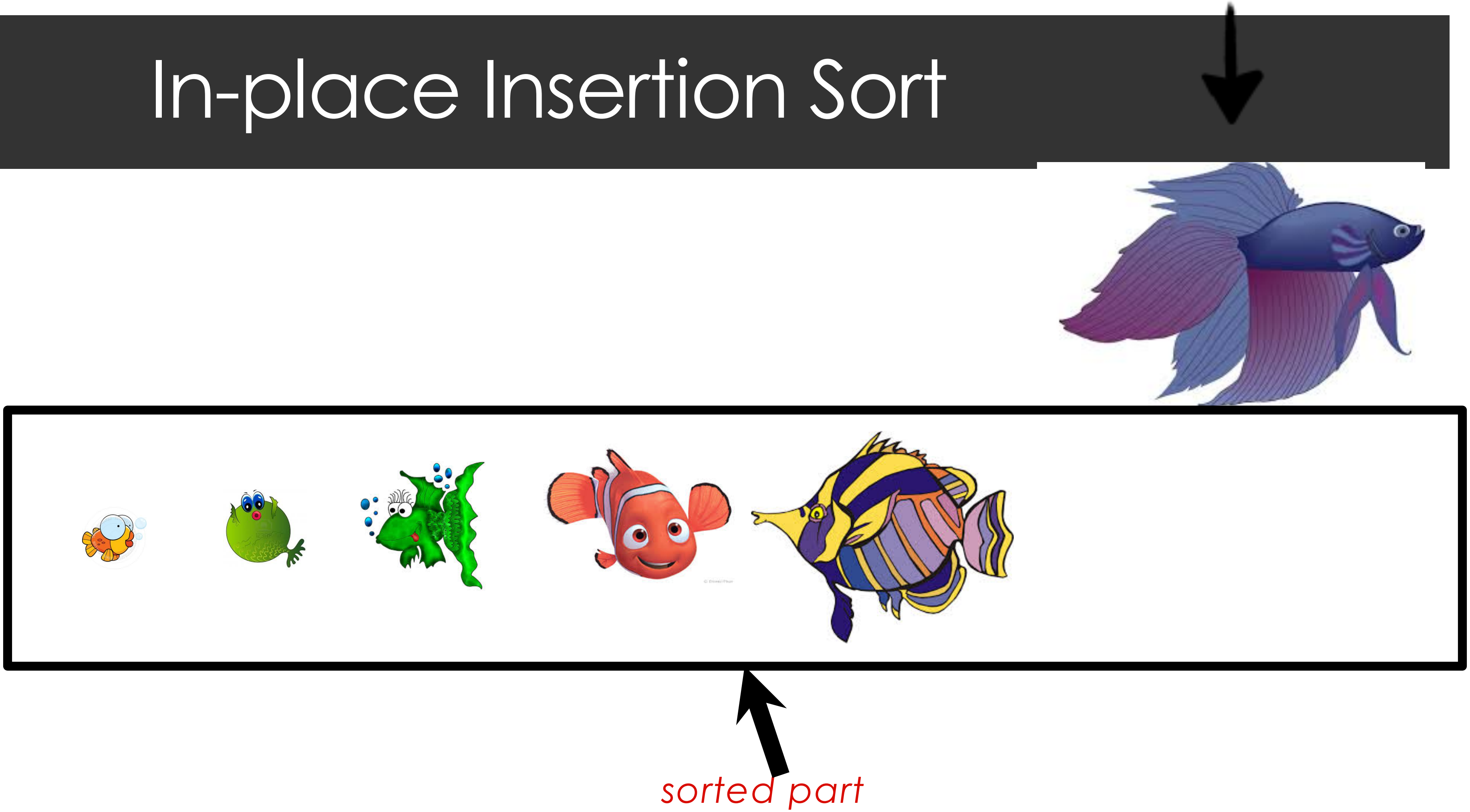
In-place Insertion Sort



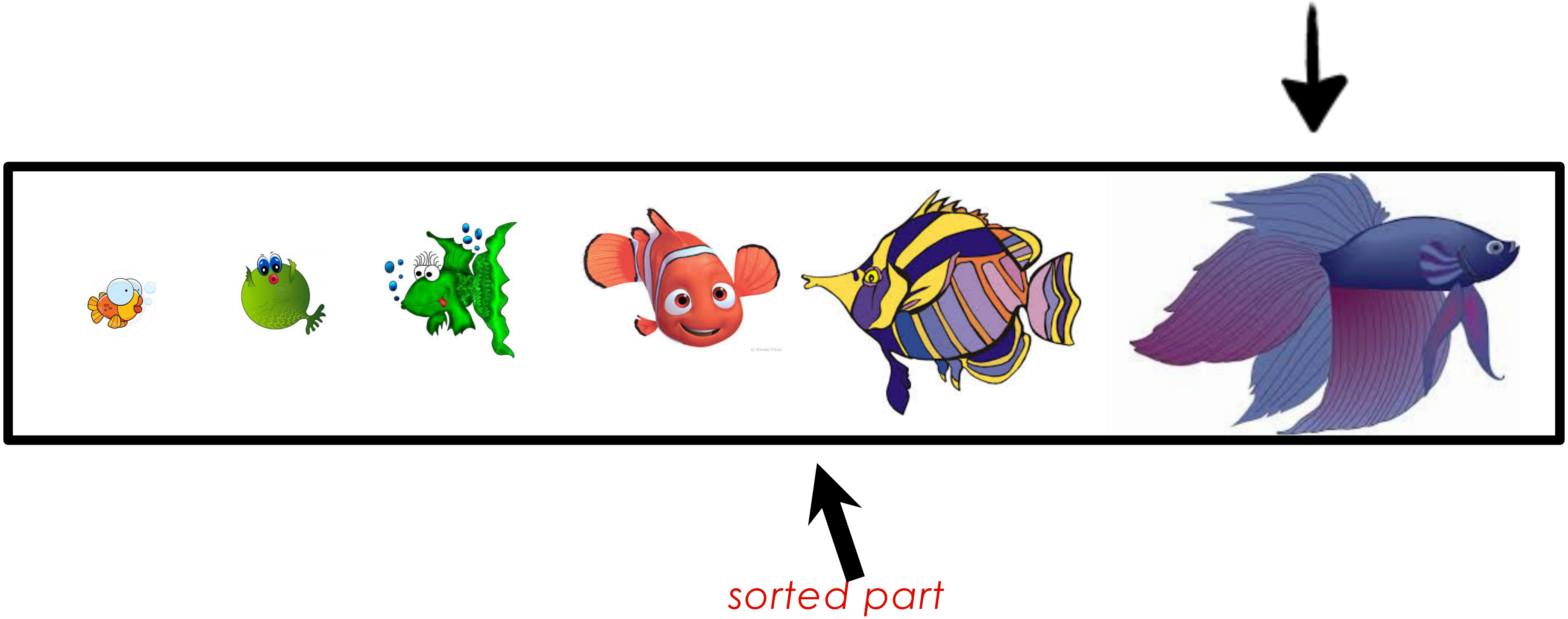
In-place Insertion Sort



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
```

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

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".)

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].

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

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?

Thinking like a computer scientist

Code Analysis

Efficiency

- A computer program should be correct, but it should also
  - execute as quickly as possible (time-efficiency)
  - use memory wisely (storage-efficiency)
- How do we compare programs (or algorithms in general) with respect to execution time?
  - various computers run at different speeds due to different processors
  - compilers optimize code before execution
  - the same algorithm can be written differently depending on the programming paradigm

Counting Operations

- We measure time efficiency by considering "work" done
  - Counting the number of operations performed by the algorithm.
- But what is an "operation"?
  - assignment statements
  - comparisons
  - function calls
  - return statements

Think of it in a machine-independent way

- We think of an operation as any computation that is independent of the size of our input.

Linear Search

```
# let n = the length of list.
def search(list, key):
    index = 0
    while index < len(list):
        if list[index] == key:
            return index
        index = index + 1
    return None
```

Best case: the key is the first element in the list

Linear Search: Best Case

```
# let n = the length of list.
def search(list, key):
   index = 0                                   1
   while index < len(list):                    1
      if list[index] == key:                   1
           return index                        1
      index = index + 1
   return None
                               Total:4
```

Linear Search: Worst Case

```
# let n = the length of list.
def search(list, key):
  index = 0
  while index < len(list):
     if list[index] == key:
          return index
     index = index + 1
  return None
```

Worst case: the key is not an element in the list

Linear Search: Worst Case

```
# let n = the length of list.
def search(list, key):
  index = 0                                        1
  while index < len(list):                         n+1
     if list[index] == key:                        n
          return index
     index = index + 1                             n
  return None                                      1
                                       Total:      3n+3
```

Asymptotic Analysis

- How do we know that each operation we count takes the same amount of time?
  - We don't.
- So generally, we look at the process more abstractly
  - We care about the behavior of a program in the long run (on large input sizes)
  - We don't care about constant factors (we care about how many iterations we make, not how many operations we have to do in each iteration)

What Do We Gain?

- Show important characteristics in terms of resource requirements
- Suppress tedious details
- Matches the outcomes in practice quite well
- As long as operations are faster than some constant (1 ns? 1 µs? 1 year?), it does not matter

Linear Search: Best Case Simplified

```
# let n = the length of list.
def search(list, key):
  index = 0
  while index < len(list):      1 iteration
     if list[index] == key:
          return index
     index = index + 1
  return None
```

Linear Search: Worst Case Simplified

```
# let n = the length of list.
def search(list, key):
  index = 0
  while index < len(list):      n iterations
     if list[index] == key:
          return index
     index = index + 1
  return None
```

Order of Complexity

- For very large n, we express the number of operations as the (time) order of complexity.
- For asymptotic upper bound, order of complexity is often expressed using Big-O notation:

Number of operations	Order of Complexity
n	O(n)
3n+3	O(n)
2n+8	O(n)

Usually doesn't matter what the constants are... we are only concerned about the highest power of n.

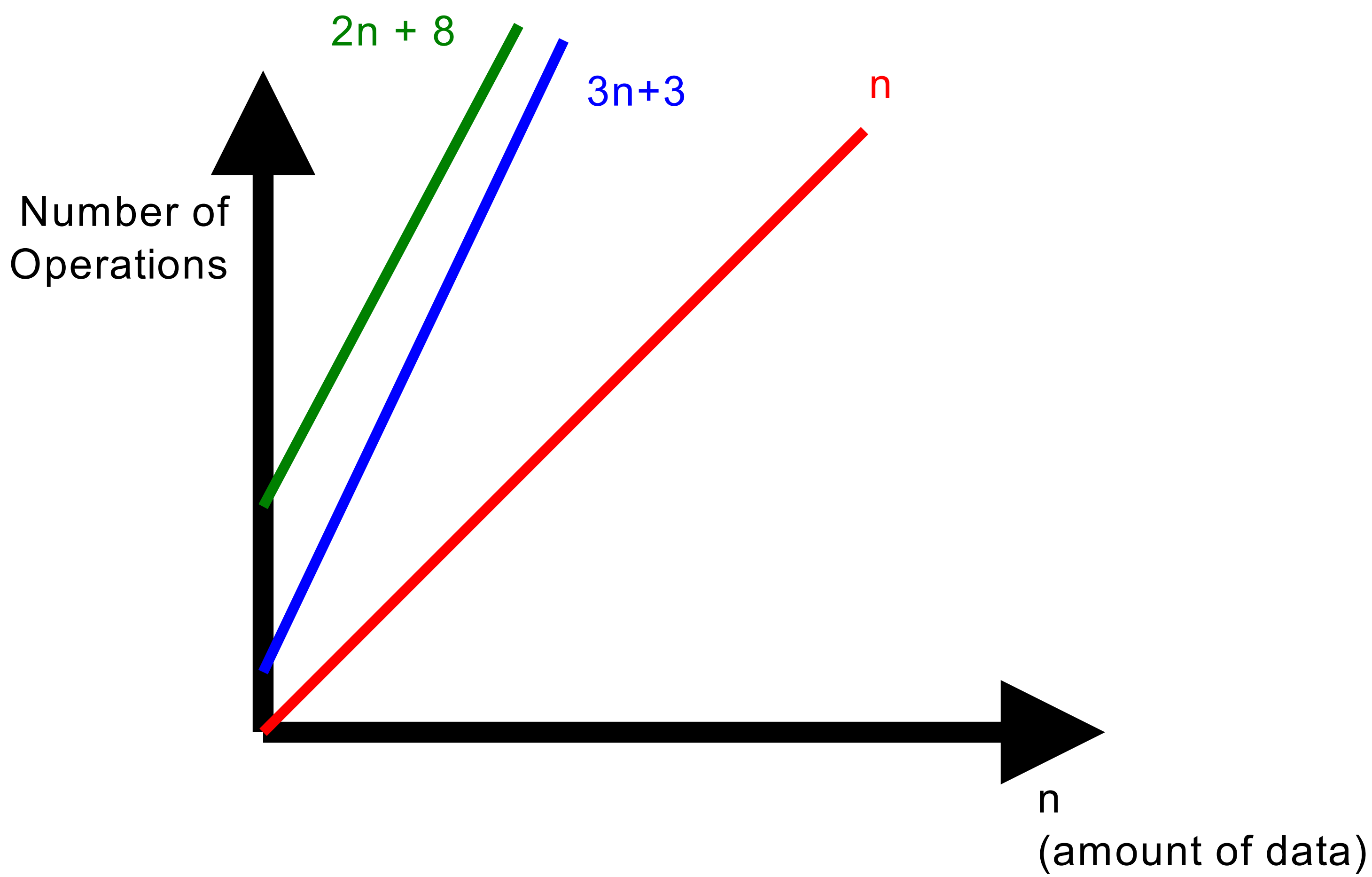
Why don't constants matter?

(n=1) $45n^3 + 20n^2 + 19 = 84$

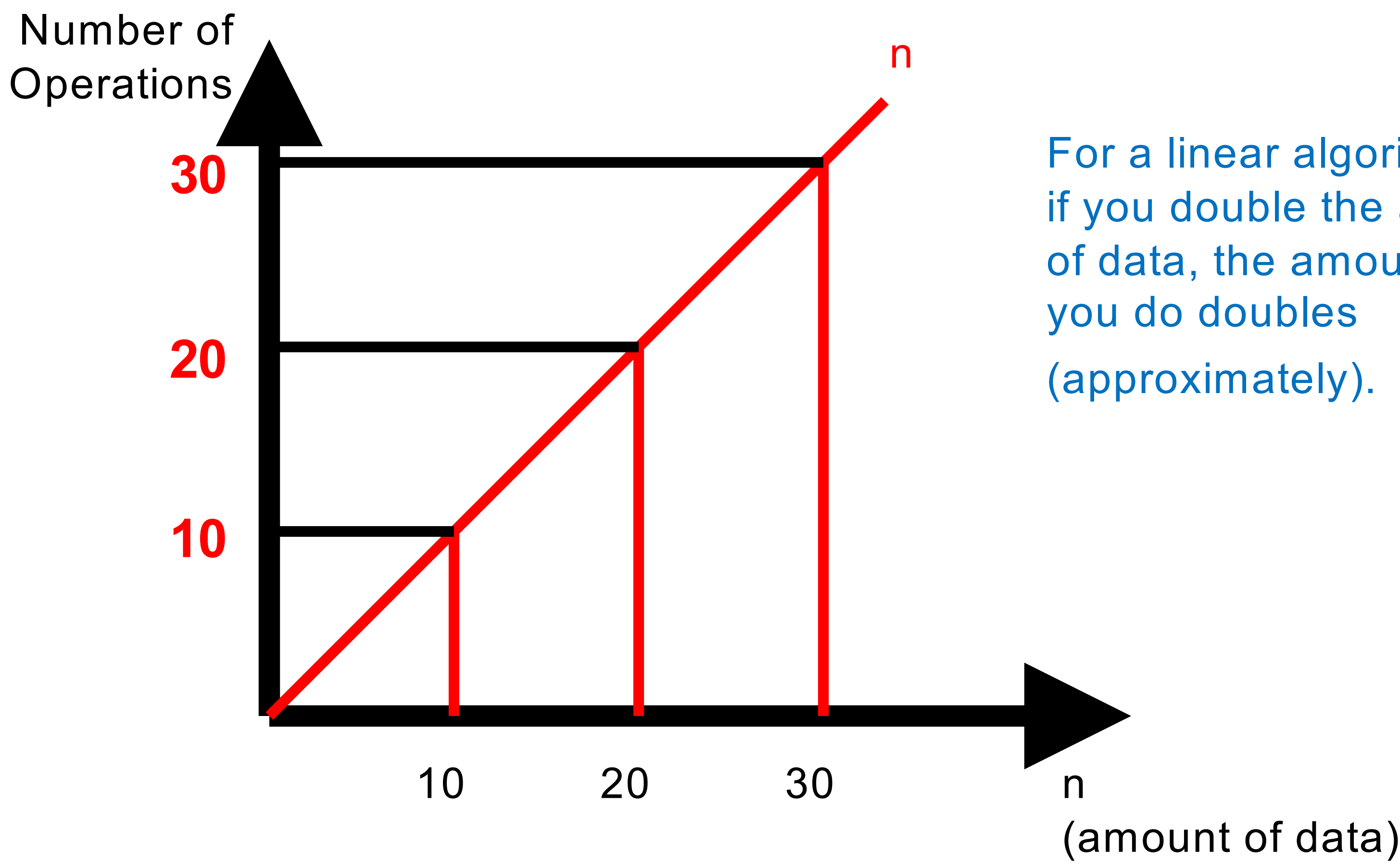
(n=2) $45n^3 + 20n^2 + 19 = 459$

(n=3) $45n^3 + 20n^2 + 19 = 1414$

O(n) ("Linear")



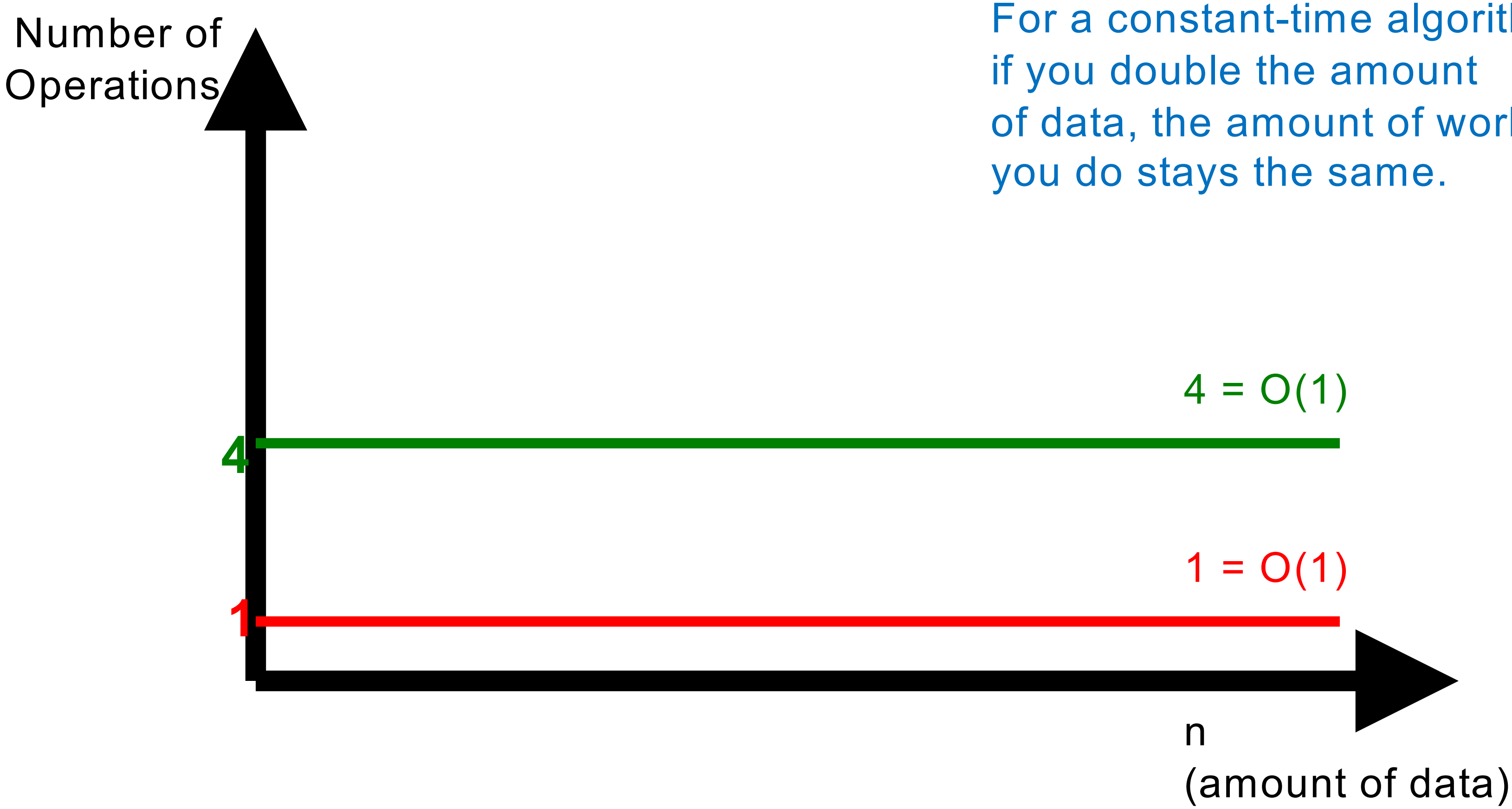
O(n)



For a linear algorithm, if you double the amount of data, the amount of work you do doubles (approximately).

O(1) ("Constant-Time")

For a constant-time algorithm, if you double the amount of data, the amount of work you do stays the same.



Linear Search

- Best Case: O(1)
- Worst Case: O(n)
- Average Case: ?
  - Depends on the distribution of queries
  - But can't be worse than O(n)

Insertion Sort

```
# let n = the length of list.
def isort(list):
    i = 1
    while i != len(list):   n-1 iterations
        move_left(list, i)
        i = i + 1
    return list
```

move_left

```
# let n = the length of list.
def move_left(a, i):
      x = a.pop(i)
      j = i - 1
      while j >= 0 and a[j] > x: i iterations
            j = j - 1
      a.insert(j + 1, x)
```

at most

but how long do **`pop`** and **`insert`** take?

Measuring pop and insert

2 million elements in list, 1000 inserts:	0.7548720836639404 seconds
4 million elements in list, 1000 inserts:	1.6343820095062256 seconds
8 million elements in list, 1000 inserts:	3.327040195465088 seconds
8 million elements in list, 1000 pops:	2.031071901321411 seconds
16 million elements in list, 1000 pops:	4.033380031585693 seconds
32 million elements in list, 1000 pops:	8.06456995010376 seconds

Doubling the size of the list doubles the cost (time) of insert or pop. These functions take **linear time.**

move_left

```
# let n = the length of list.
def move_left(a, i):
    x = a.pop(i)                    n iterations
    j = i - 1
    while j >= 0 and a[j] > x:  i iterations
        j = j - 1
    a.insert(j + 1, x)              n iterations
```

Insertion Sort: what is the cost of move_left?

```
# let n = the length of list.
def move_left(a, i):
      x = a.pop(i)                    n iterations
      j = i - 1
      while j >= 0 and a[j] > x: i iterations
            j = j - 1
      a.insert(j + 1, x)              n iterations
```

Total cost (at most): **n + i + n**

But what is **i**? To find out, look at isort, which calls move_left, supplying a value for **i**

Insertion Sort: what is the cost of the whole thing?

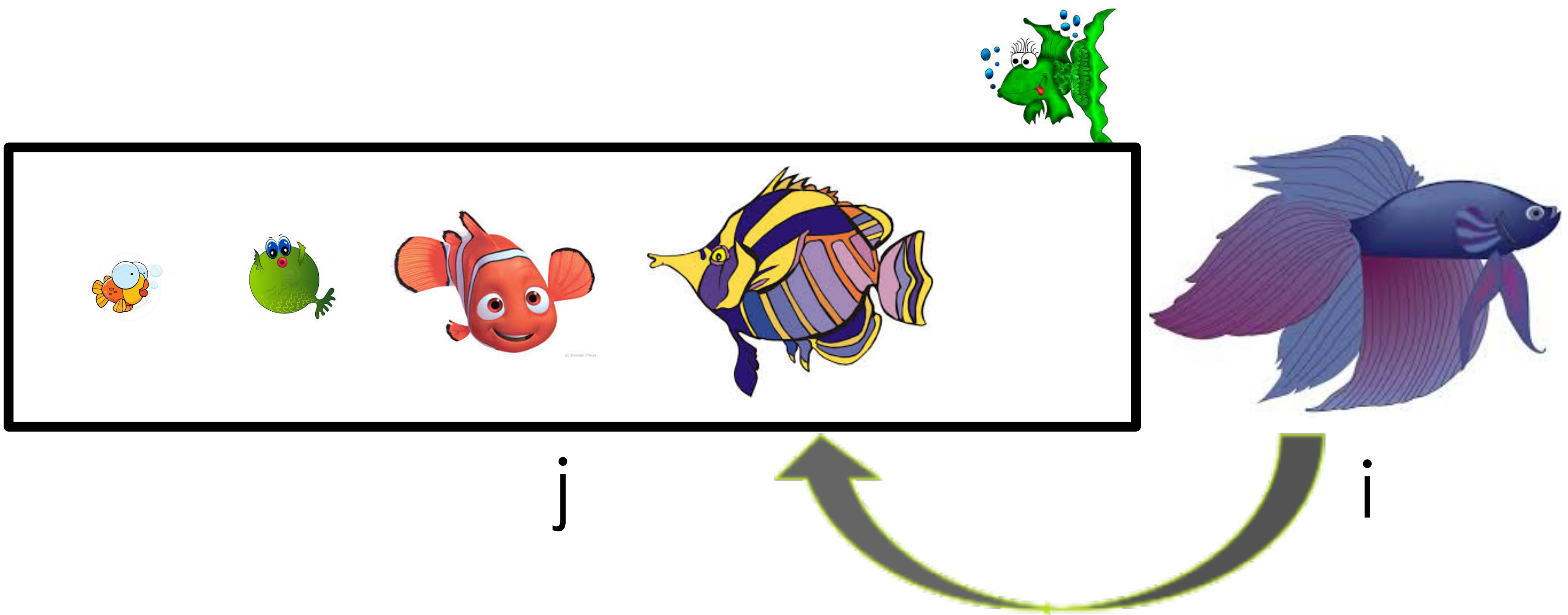
```
# let n = the length of list.
def isort(list):
    i = 1
    while i != len(list):     #n-1 iterations
        move_left(list,i)     #i goes from 1 to n-1
        i = i + 1
    return list
```

Total cost: *cost of move_left as i goes from 1 to n-1*

Cost of all the move_lefts:

```
n + 1 + n
+ n + 2 + n
+ n + 3 + n
...
+ n + n-1 + n
```

In place iSort Worst Case…



- • On iteration i, we need to examine j elements and then shift i-j elements to the right, so we have to do j + (i-j) = i units of work.

Figuring out the sum

- $n + 1 + n$
- $+ n + 2 + n$
- $+ n + 3 + n$
- …
- $+ n + n-1 + n$

$(n-1)*2n$

$+ 1$

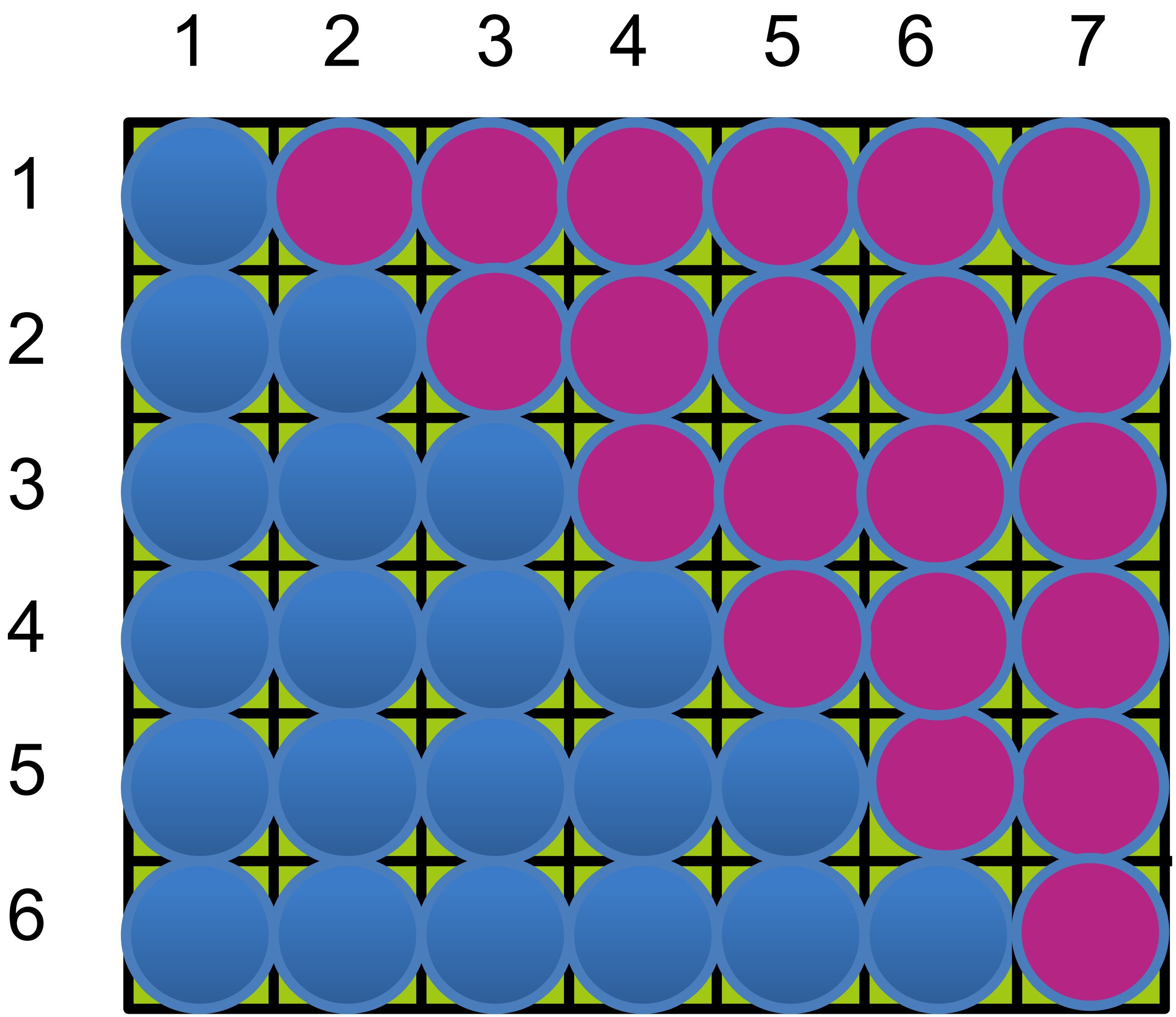
$+ 2$

$+ 3$

…

$+ n-1$

Adding 1 through n-1



(6 * 7) / 2
blue circles

Adding 1 through n-1

- We saw 1 + 2 + ... + 6 = (6 * 7) / 2
- Generalizing, 1 + 2 + ... + n-1 = (n-1)(n) / 2
- So our whole cost is:
- (n-1)*2n + 1 + 2 + 3 ... + n-1
- = (n-1)*2n + (n-1)(n) / 2
- = $2n^2$ - 2n + (n^2 - n) / 2
- = ($5n^2$ - 5n) / 2 = (5/2)n^2 - (5/2)n
- Observe that the highest-order term is n^2

A different way…

- When i=1,we have1 unit of work.
- When i=2, we have 2 units of work.
- …
- When i = n-1, we have n-1 units of work.
- The total amount of work done is:
- 1 + 2 + … + (n-1)
  = n(n-1)/2
  = (n2 - n)/2 (a quadratic function)
  = $O(n^2)$

Order of Complexity

Number of operations	Order of Complexity
n^2	$O(n^2)$
$(5/2)n^2 - (1/2)n$	$O(n^2)$
$2n^2 + 7$	$O(n^2)$

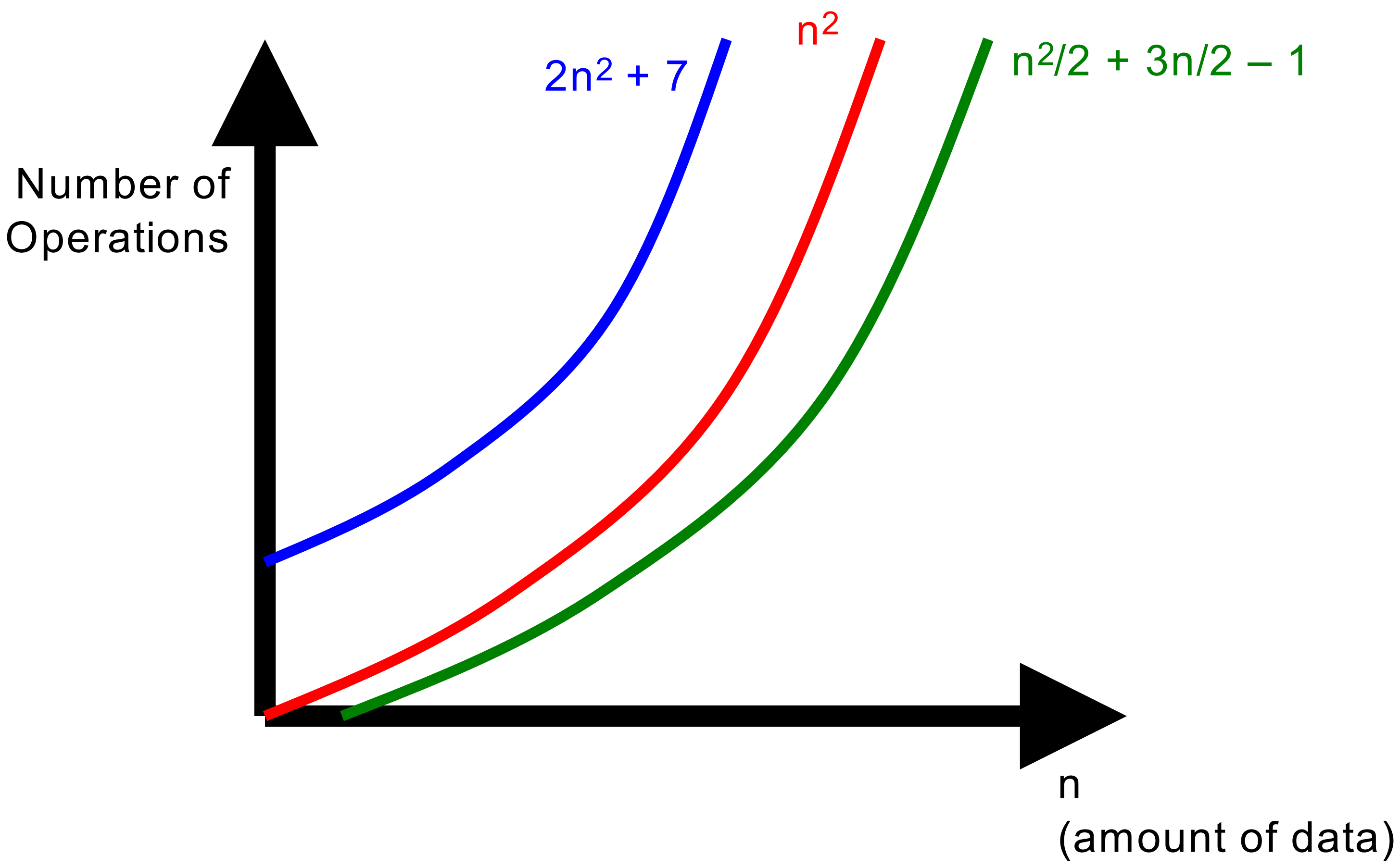
Usually doesn't matter what the constants are…
we are only concerned about the highest power of n.

$f(n)$ is $O(g(n))$ means $f(n) < g(n) \cdot k$ for some positive k

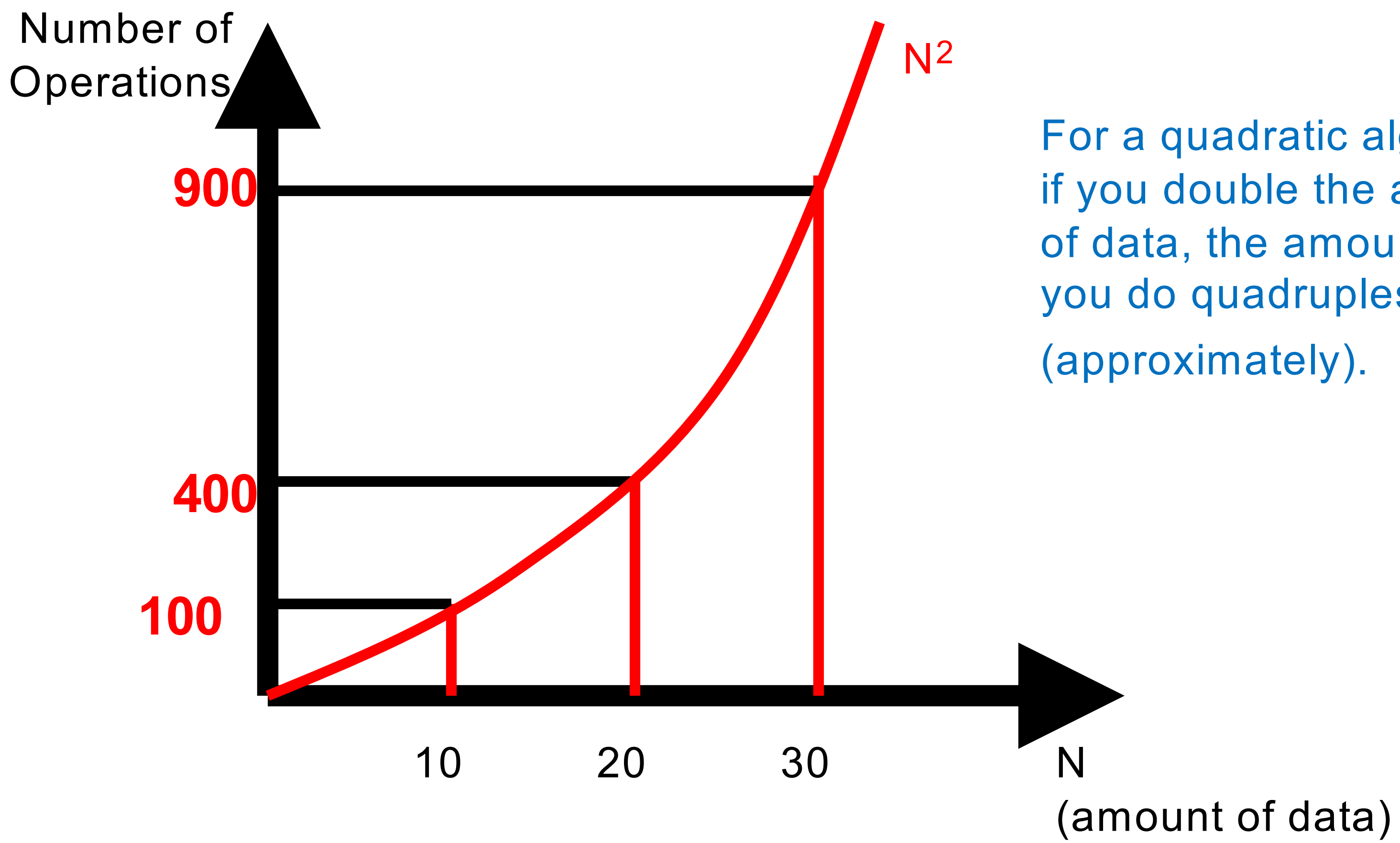
Keep It Simple

- "Big O" notation expresses an upper bound:
  $\boldsymbol{f(n)}$ is $\mathbf{O(}\boldsymbol{g(n)}\mathbf{)}$ means $\boldsymbol{f(n) < g(n) \cdot k}$
  (whenever n is large enough)
- So if $f(x)$ is $\mathrm{O}(n^2)$, then $f(x)$ is $\mathrm{O}(n^3)$ too!
- But we always use the smallest possible function, and the simplest possible.
- We say $3n^2 + 4n + 1$ is $\mathrm{O}(n^2)$, not $\mathrm{O}(n^3)$
- We say $3n^2 + 4n + 1$ is $\mathrm{O}(n^2)$, not $\mathrm{O}(3n^2 + 4n)$
- ...even though all of the above are true

$O(n^2)$ ("Quadratic")



$O(n^2)$



For a quadratic algorithm, if you double the amount of data, the amount of work you do quadruples (approximately).

Insertion Sort

- Worst Case: $O(n^2)$
- Best Case: ?
- Average Case: ?

- We'll compare these algorithms with others soon to see how scalable they really are based on their order of complexities.

Big O

O(1)	constant
O(log *n*)	logarithmic
O(*n*)	linear
O(*n* log *n*)	log linear
O(n^2)	quadratic
O(n^3)	cubic
O(2^n)	exponential