Sorting
Algorithmic Complexity

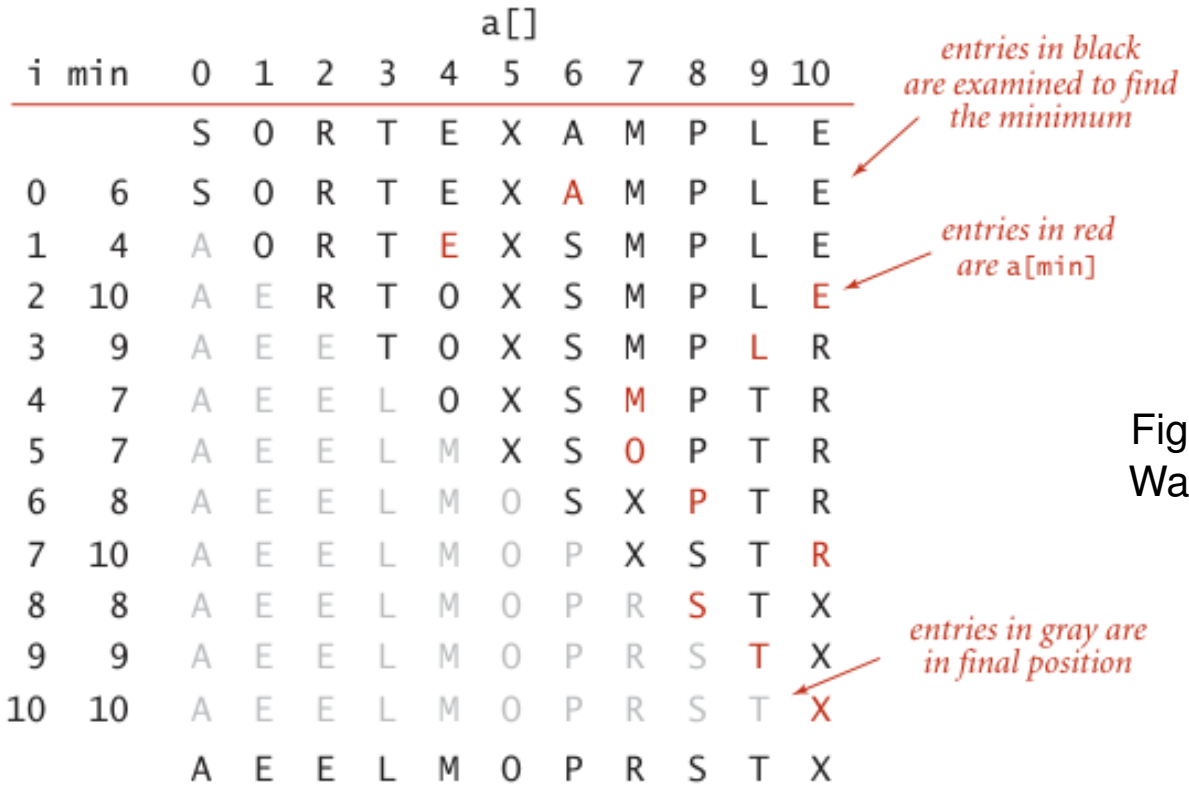
Intro. to computing

`https://mandarm.github.io/computing.html`

Selection sort

Algorithm

1. Find the smallest item in the array and exchange it with the first entry.
2. Find the next smallest item and exchange it with the second entry.
3. Continue in this way until the entire array is sorted.



		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Figure from *Algorithms*, Sedgewick and Wayne, Addison-Wesley, 4th ed.

Selection sort: implementation

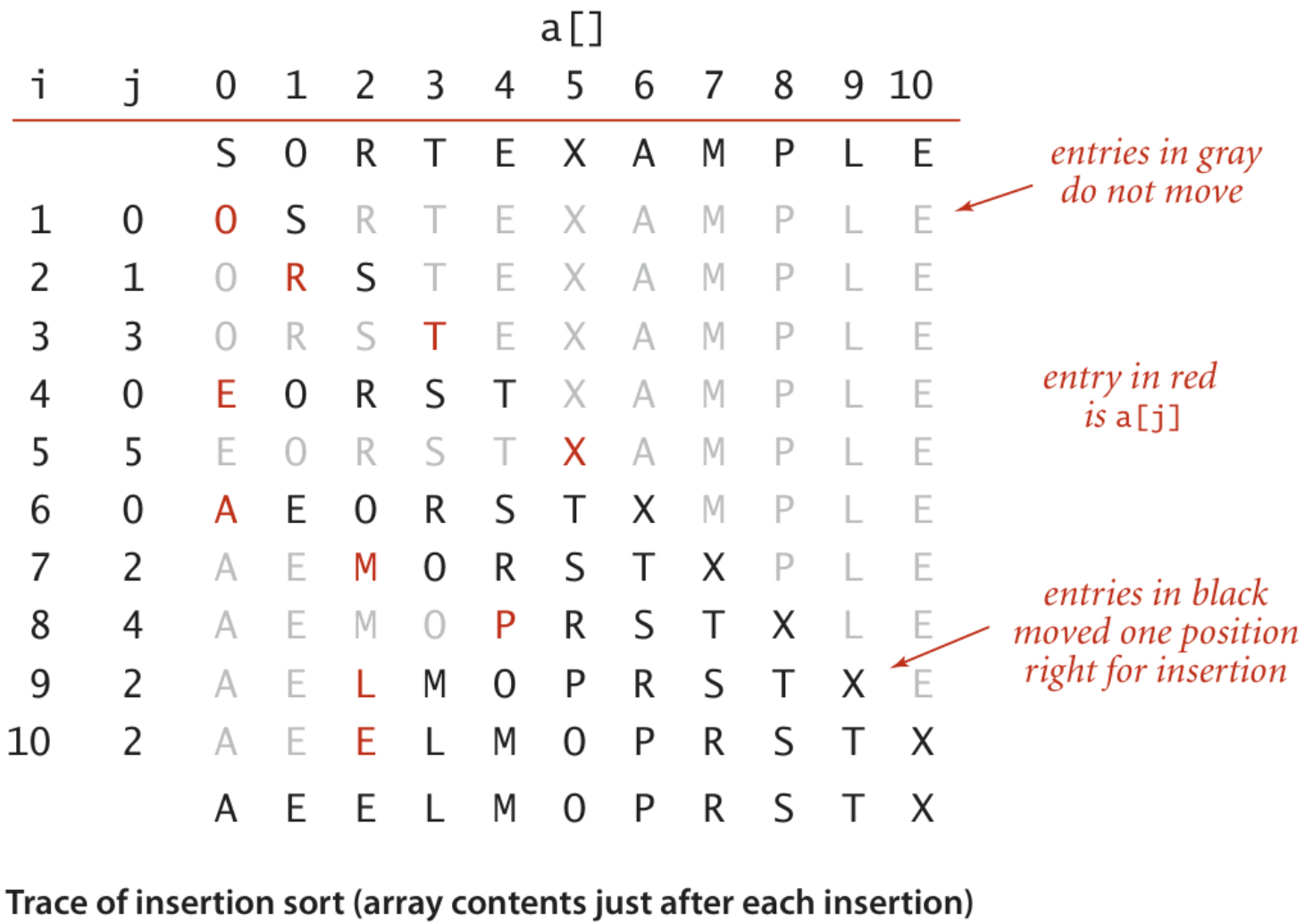
```
def selection_sort(l):
    N = len(l)
    for i in range(N):
        # Find the minimum element in l[i:]
        min_index = i
        for j in range(i+1, N):
            if l[j] < l[min_index]:
                min_index = j
        # Exchange a[i] with smallest entry in a[i+1...N).
        l[i], l[min_index] = l[min_index], l[i]
    assert((l == sorted(l)).all())
```

Insertion sort

Algorithm: sort the elements starting from the left, one-by-one.

1. Insert the second element into its proper position w.r.t. the first element.
2. Insert the third element into its proper position w.r.t. the first **two** elements (already sorted).
3. Insert the fourth element into its proper position w.r.t. the first **three** elements (already sorted).
4. Continue in this way until the entire array is sorted.

Insertion sort: example



		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Trace of insertion sort (array contents just after each insertion)

Figure from *Algorithms*, Sedgewick and Wayne, Addison-Wesley, 4th ed.

Insertion sort: implementation

```
def insertion_sort(l):
    N = len(l)
    for i in range(1,N):
        # insert l[i] in correct position of l[:i] (already sorted)
        for j in range(i, 0, -1):
            if l[j] < l[j-1]:
                l[j], l[j-1] = l[j-1], l[j]
            else: break
    assert((l == sorted(l)).all())
```

Bubble sort

```
def bubble_sort(l):
    N = len(l)
    for i in range(N):
        for j in range(N-1, i, -1):
            if l[j] < l[j-1]:
                l[j], l[j-1] = l[j-1], l[j]
    assert((l == sorted(l)).all())
```

Merge sort

(Recursive) algorithm:

1. If the list is of length 1, it is already sorted.
2. Otherwise, split the list into 2 approx. equal parts.
3. Sort each part *recursively*.
4. Merge the two sorted parts into a single, new sorted list.
5. Copy the sorted list into the original list.

```
def merge_sort(l):
    global aux
    N = len(l)
    if N == 1: return
    l1 = l[:N//2]     # NOTE: numpy slices
    l2 = l[N//2:]     # are views not copies
    merge_sort(l1)
    merge_sort(l2)
    merge(l1, l2, aux)
    l[:] = aux[:N]
```

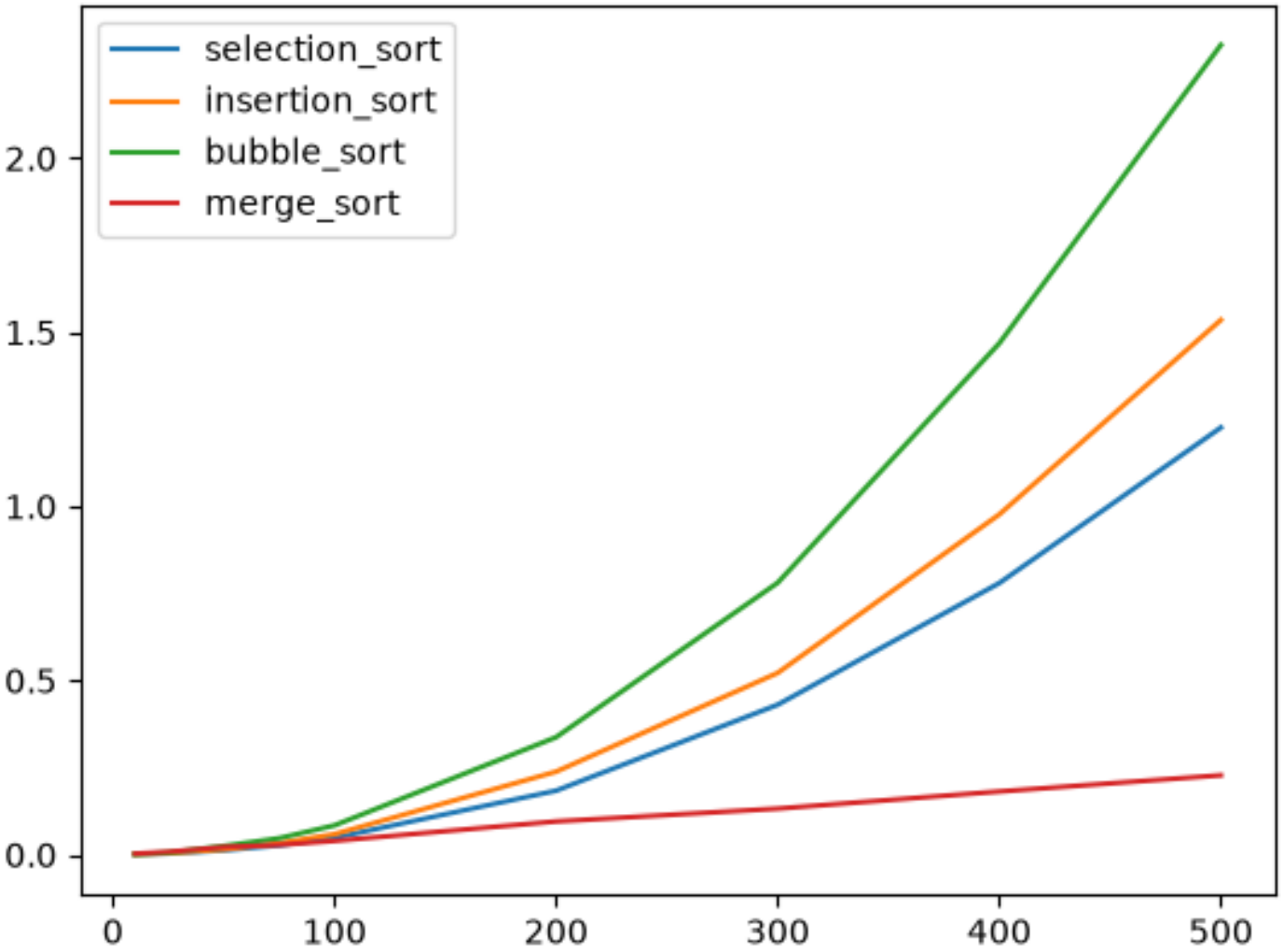
Merging two sorted lists I

1. Place markers at the starting positions of the two sorted lists, and the new (output) list.
2. Pick the smaller element from the two sorted list, and insert it into the output list.
   If both elements are the same, insert both into the output list.
3. Move the corresponding markers forward.
4. Continue until one sorted list is exhausted.
5. If the other sorted list still has elements, copy them into the output list one by one.

Merging two sorted lists II

```
def merge(l1, l2, l):
    n1 = len(l1)
    n2 = len(l2)
    i = i1 = i2 = 0
    while i1 < n1 and i2 < n2:
        if l1[i1] < l2[i2]:
            l[i] = l1[i1]
            i, i1 = i+1, i1+1
        elif l1[i1] > l2[i2]:
            l[i] = l2[i2]
            i, i2 = i+1, i2+1
        else:
            l[i], l[i+1] = l1[i1], l2[i2]
            i, i1, i2 = i+2, i1+1, i2+1
    while i1 < n1:
        l[i] = l1[i1]
        i, i1 = i+1, i1+1
    while i2 < n2:
        l[i] = l2[i2]
        i, i2 = i+1, i2+1
    return
```

Measuring the time taken



(Asymptotic) time complexity

Choosing an algorithm for a task

- Criteria
  - Time
  - Space / memory / storage
  - Communication bandwidth, no. of messages

Choosing an algorithm for a task

- Criteria
  - *Time*
  - Space / memory / storage
  - Communication bandwidth, no. of messages

Choosing an algorithm for a task

- Criteria
  - *Time*
  - Space / memory / storage
  - Communication bandwidth, no. of messages
- Objective: compute *time complexity* $T(n)$, where
  - $n =$ size of the input — needs to be carefully specified, depending on problem
  - $T(n) =$ time taken by the algorithm to run on an input of size n.
- $T(n)$ may depend on *nature* of input, not just size
  $\Rightarrow$ **best** case, **worst** case, **average** case / **expected** running time

  Example: Insertion sort
  - Best case: input list is sorted
  - Worst case: input list is sorted in *reverse* order
- $T(n)$ generally difficult to calculate exactly $\Rightarrow$ focus on **order** / **rate** of growth

Order notation I

- $O(f(n)) = \{g(n) \mid \exists c, n_0$ such that $g(n) \leq cf(n)$ for all $n \geq n_0\}$

  set of functions that grow no faster than $f(n)$

- $\Omega(f(n)) = \{g(n) \mid \exists c, n_0$ such that $g(n) \geq cf(n)$ for all $n \geq n_0\}$

  set of functions that grow at least as fast as $f(n)$

- $\Theta(f(n)) = \{g(n) \mid \exists c_1, c_2, n_0$ such that $c_1 f(n) \leq g(n) \leq c_2 f(n)$ for all $n \geq n_0\}$

  set of functions that grow like $f(n)$

c, c_1, c_2 are all positive constants.

Order notation II

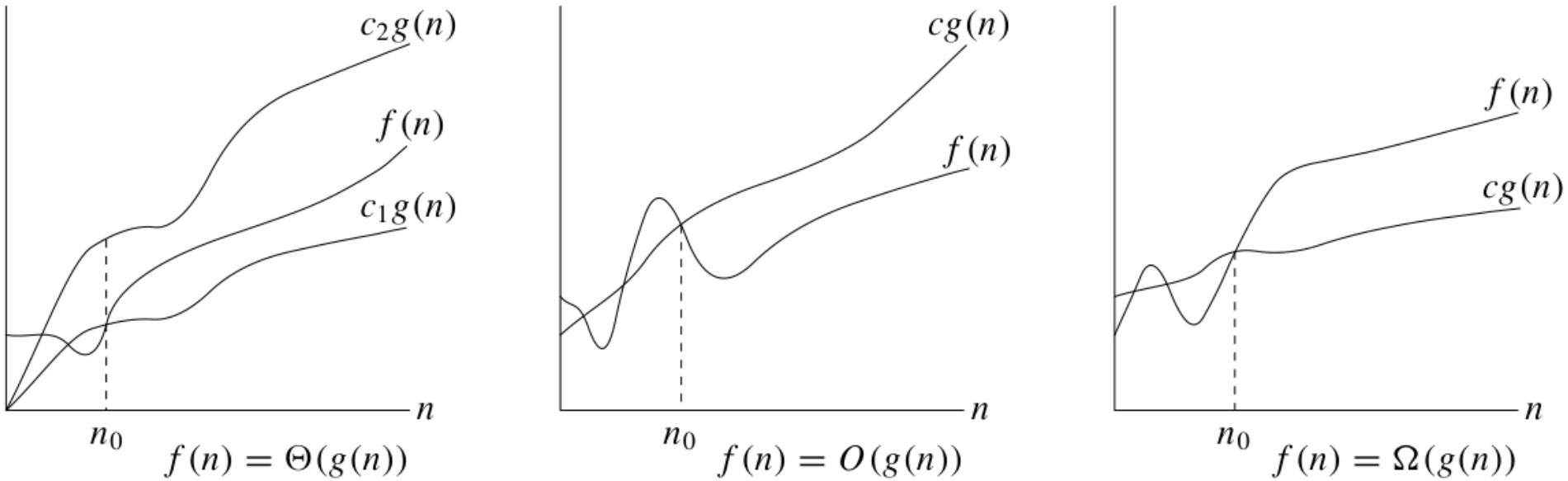


Figure from *Introduction to algorithms*, Cormen, Leiserson, Rivest, Stein, MIT Press, 2nd ed.

Warning: accepted “abuse" of notation

- $X(f(n))$ where $X = O, \Omega$ or Θ denotes a *set*.
- $g(n)$ is $X(f(n))$ or $g(n) = X(f(n))$ means $g(n) \in X(f(n))$

Order notation III

Relating order notation to calculus

Reasonable assumptions: functions of interest ($T(n)$ or $f(n)$, $g(n)$) are non-negative, monotonic.

- If $g(n) \in O(f(n))$, then $\lim_{n\to\infty} \frac{g(n)}{f(n)}$ exists, and may be 0.
- If $g(n) \in \Omega(f(n))$, then $\lim_{n\to\infty} \frac{g(n)}{f(n)}$ is $c > 0$ or ∞.
- If $g(n) \in \Theta(f(n))$, then $\lim_{n\to\infty} \frac{g(n)}{f(n)}$ is $c > 0$.

Examples:

$$O(1) \subset O(\log n) \subset O(n^k)(k < 1) \subset O(n) \subset O(n^l)(l > 1) \subset O(a^n)(a > 1)$$

Complexity analysis: example

```
min_index = 0
for i in range(1, N):
    if l[i] < l[min_index]:
        min_index = i
```

Selection sort: complexity analysis

```
def selection_sort(l):
    N = len(l)
    for i in range(N):
        # Find the minimum element in l[i:]
        min_index = i
        for j in range(i+1, N):
            if l[j] < l[min_index]:
                min_index = j
        # Exchange a[i] with smallest entry in a[i+1...N).
        l[i], l[min_index] = l[min_index], l[i]
    assert((l == sorted(l)).all())
```

Insertion sort: complexity analysis

```
def insertion_sort(l):
    N = len(l)
    for i in range(1,N):
        # insert l[i] in correct position of l[:i] (already sorted)
        for j in range(i, 0, -1):
            if l[j] < l[j-1]:
                l[j], l[j-1] = l[j-1], l[j]
            else: break
    assert((l == sorted(l)).all())
```

Merge sort: complexity analysis

```
def merge_sort(l):
    global aux
    N = len(l)
    if N == 1: return
    l1 = l[:N//2]    # NOTE: numpy slices
    l2 = l[N//2:]    # are views not copies
    merge_sort(l1)
    merge_sort(l2)
    merge(l1, l2, aux)
    l[:] = aux[:N]
```