Applied Programming 1

Algorithms on Lists

Degree on Bioinformatics

Lluís Padró

Computer Science Department – UPC

padro@cs.upc.edu

Algorithms on unsorted lists

Apply a modification to each list element

```
# Pre: 'a' is a list of elements, 'n' is an integer
# Post: A new list is returned, where each element in 'a'
#       has been multiplied by n
def multiply(a,n) :
   new_a = []
   # for each element in the list
   for x in a :
      # add a new element to the new list. With the
      # value in the original list times n
      new_a.append(x*n)

   # return new list
   return new_a
```

Apply a modification to each list element

```
# Pre: 'a' is a list of elements, 'n' is an integer
# Post: Each element in the list has been multiplied by n
def multiply(a,n) :
   # for each position in the list
   for i in range(len(a)) :
      # multiply content of position i by n
      a[i] == a[i] * n

   # No need to return anything, the list has been
modified
```

Find two numbers in a list that add up *x*

```
# Pre: 'a' is a list of elements, 'x' is an integer
# Post: Return the position of two elements in 'a' that sum 'x',
#       or -1,-1 if not found
def locate_pair(a,x) :
   # for each position in the list
   for i in range(len(a)) :
      # check pairing position i with all positions after it.
      for j in range(i+1,len(a)) :
         # if content of positions i and j work, we are done.
         if a[i] + a[j] == x :
            return i,j

   # The loop ended without ever returning, so no pair of
   # working positions was found.
   return -1,-1
```

Rotate a list 1 position right

```
# Pre: 'a' is a non-empty list
# Post: Elements in the list have been shifted right 1 position
def rotate(a) :
    i = len(a)-1 # last position
    x = a[i] # save last element
    # check positions right to left, moving one element at a time
    while i>0 :
        a[i] = a[i-1]
        i = i-1

    # recover saved last element, and put it in first position
    a[0] = x

    # No need to return anything, the list has been modified
```

Find the *k*-th max element in a list

```
# Pre: 'a' is a list, 'k' is integer, len(a)>=k
# Post: The k-th largest element in 'a' is returned
def k_max(a,k) :
   maxs = []    # list of k max values seen so far
   for x in a :  # check each value in the list
      # find in which position of maxs it should go
      i = 0
      while i<len(m) and m[i]<x :
         i = i+1
      # insert x in its position, moving up other elements
      m.insert(i,x)
      # if we have k+1 elements, discard the smallest
      if len(m)>k : maxs = maxs[1:]

   return maxs[0]
```

Algorithms on sorted lists

Find two numbers in a list that add up *x*

```
# Pre: 'a' is a sorted list of integers, 'x' is an integer
# Post: Return the position of two elements in 'a' that sum 'x',
#       or -1,-1 if not found
def locate_pair(a,x) :
   # check each position in the list that is not too big
   i = 0
   while i<len(a) and a[i]<x :
      # pair position 'i' with all positions afer it, as long
      # as the value is not too big
      j = i+1
      while j<len(a) and a[i]+a[j]<x :
         # if content of positions i and j work, we are done.
         if a[i]+a[j] == x : return i,j
         j += 1
      i += 1
   # The loop ended without ever returning, no such pair found.
   return -1,-1
```

Find two numbers in a list that add up *x*

```
# Pre: 'a' is a sorted list of integers, 'x' is an integer
# Post: Return the position of two elements in 'a' that sum 'x',
#       or -1,-1 if not found
def locate_pair(a,x) :
   # check each position in the list that is not too big
   i = 0
   while i<len(a) and a[i]<x :
       # pair position 'i' with all posit
       # as the value is not too big
       j = i+1
       while j<len(a) and a[i]+a[j]<x :
          # if content of positions i and j work, we are done.
          if a[i]+a[j] == x : return i,j
          j += 1
       i += 1
   # The loop ended without ever returning, no such pair found.
   return -1,-1
```

We cut short the list traversal when numbers are too big to sum x.
Since list is sorted, pending numbers are still larger and the sum would be even bigger

Merge of two sorted lists

```
# Pre: v1 and v2 are sorted lists of integers
# Post: Returns a sorted list with all elements of v1 and v2
def merge(v1,v2) :
    result = [];  i=0;  j=0
    while i<len(v1) and j<len(v2) :
        if v1[i]<v2[j] :
            result.append(v1[i])
            i += 1
        elif v1[i]>v2[j] :
            result.append(v2[j])
            j += 1
        else:
            result.append(v1[i])
            result.append(v2[j])
            i += 1
            j += 1
    result.extend(v1[i:]) # add any elements left in v1
    result.extend(v2[j:]) # add any elements left in v2
    return result
```

At each step, the resulting list is enlarged with the smaller of the two pending elements in either v1 or v2.
Variables i and j control respectively which is the next element from v1 and v2 to be added,

Intersection of two sorted lists

```
# Pre: v1 and v2 are sorted lists of integers without duplicates
# Post: Returns a sorted list with common elements of v1 and v2
def intersection(v1,v2) :
   result = []; i=0; j=0
   # reach only the end of the shortest
   while i<len(v1) and j<len(v2) :
      # if different elements, skip the smallest
      if v1[i]<v2[j] :
         i += 1
      elif v1[i]>v2[j] :
         j += 1
      else:
         # if same element, add to result (if not already there)
         result.append(v1[i])
         i += 1
         j += 1
   return result
```

Difference of two sorted lists

```
# Pre: v1 and v2 are sorted lists of integers without duplicates
# Post: Returns a sorted list with all in v1 but not in v2
def difference(v1,v2) :
  result = []
  i,j = 0,0
  while i<len(v1) and j<len(v2) :
     if v1[i]<v2[j] : # element in v1 and not in v2
        result.append(v1[i])
        i += 1
    elif v1[i]>v2[j] : # element in v2 not in v1, skip
        j += 1
    else: # element in both, skip
        i += 1
  # if there are any elements left in v1, add them
  result.extend(v1[i:])
  return result
```

Sorting Algorithms

Need for sorted data

- Having data sorted is a common need when programming

  Advantatges

  - Display results properly sorted
  - Speed up searches (e.g. binary search)
  - Simplify data analysis
  - Reduce storage space

  Disadvantatges

  - Sorting takes time $O(n \cdot \log n)$
    Gain obtained from having sorted data must compensate the extra time.

What can be sorted

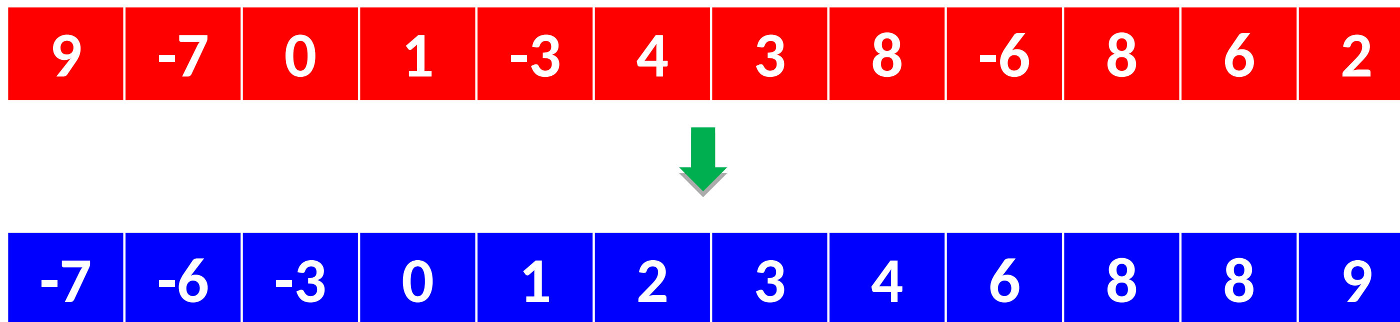
- We only can sort elements when a total order is defined
- A total order is an operation $\leq$ that given two elements a and b can always decide whether $a \leq b$ or $b \leq a$
- Elements to sort can be of any type, as long as we can define their total order
- A fundamental very common problem is sorting a list of elements:

9	-7	0	1	-3	4	3	8	-6	8	6	2

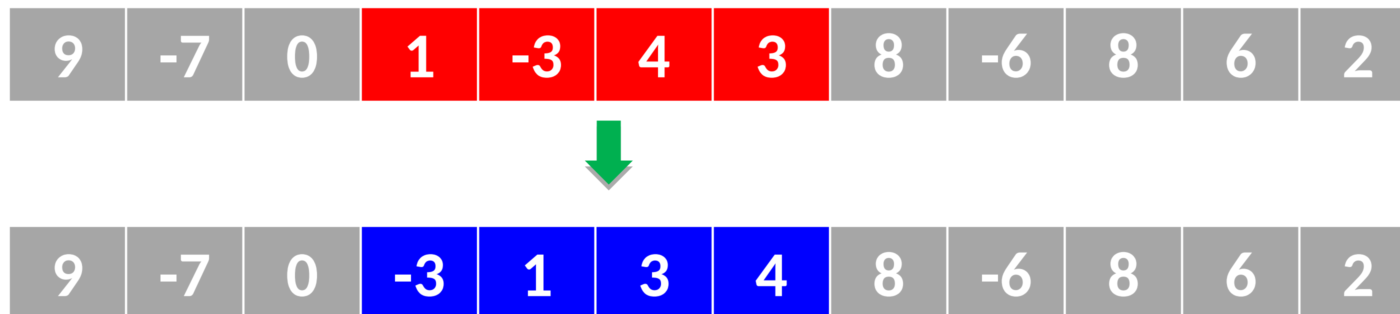
-7	-6	-3	0	1	2	3	4	6	8	8	9

What can be sorted

- A fundamental very common problem is sorting a list of elements.



- Another common task is sorting a part of the list:



Sorting algorithms

- Basic sorting algorithms *$O(n^2)$*
  - Selection sort
  - Insertion sort
  - Bubble sort
- Advanced sorting algorithms *$O(n \cdot log\ n)$*
  - Merge sort
  - Quick sort

Selection Sort

Selection sort

- Main idea:
  - Find the smallest element and place it in a[0]
  - Find the second smallest element and place it in a[1]
  - Find the third smallest element and place it in a[2]
  - ...
  - At the i-th iteration, find the i-th smallest element and place it in a[i]

Selection sort

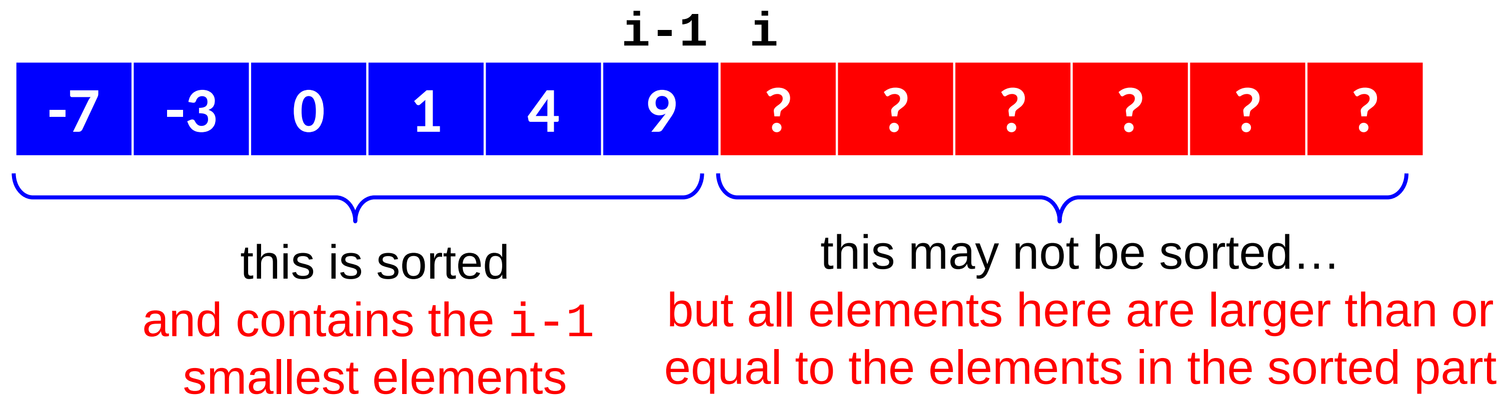
- Main idea:
  - Find the smallest element and place it in a[0]
  - Find the second smallest element and place it in a[1]
  - Find the third smallest element and place it in a[2]
  - ...
  - At the i-th iteration, find the i-th smallest element and place it in a[i]

8
5
2
6
9
3
1
4
0
7

Source: wikipedia

Selection sort

- Selection sort maintains this invariant:



Selection sort

```
# Pre: 'a' is a list of elements for which a
#      total order <= is defined
# Post: 'a' is sorted according to the total order
def selection_sort(a) :
   # for each position in the list
   for i in range(len(a)-1) :
      # find position of i-th smallest element
      # (we only need to check the remaining ones)
      p = i
      for j in range(i+1, len(a)) :
         if a[j]<=a[p] :
            p = j
      # place smallest element in a[i]
      a[i],a[p] = a[p],a[i]
```

Selection sort

Complexity

- At the i-th iteration, selection sort performs `len(a)-i` comparisons, and `1` swap
- The total number of comparisons for a list of size n is

$$(n-1)+(n-2)+(n-3) + \ldots + 2 + 1 = n(n-1)/2$$

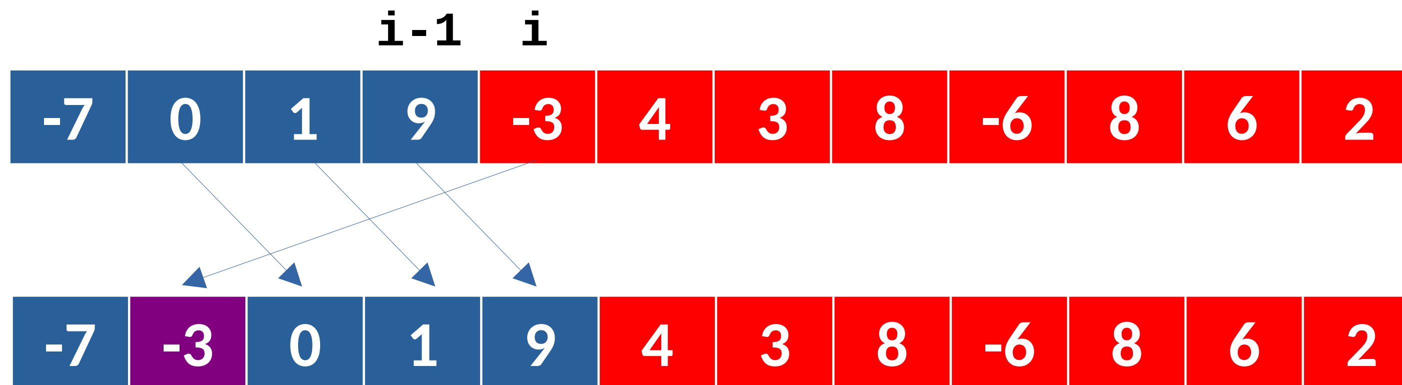
- The total number of swaps is $(n-1)$, thus $3(n-1)$ assignments
- Total cost :

$$n(n-1)/2 + 3(n-1) = (n^2 + 5n - 6)/2 \approx n^2/2 \Rightarrow O(n^2)$$

Insertion Sort

Insertion sort

- Main idea:
  - At the i-th iteration, check element in a[i] and insert it in the right place in the sorted part of the list, moving all the other elements to the right



Insertion sort

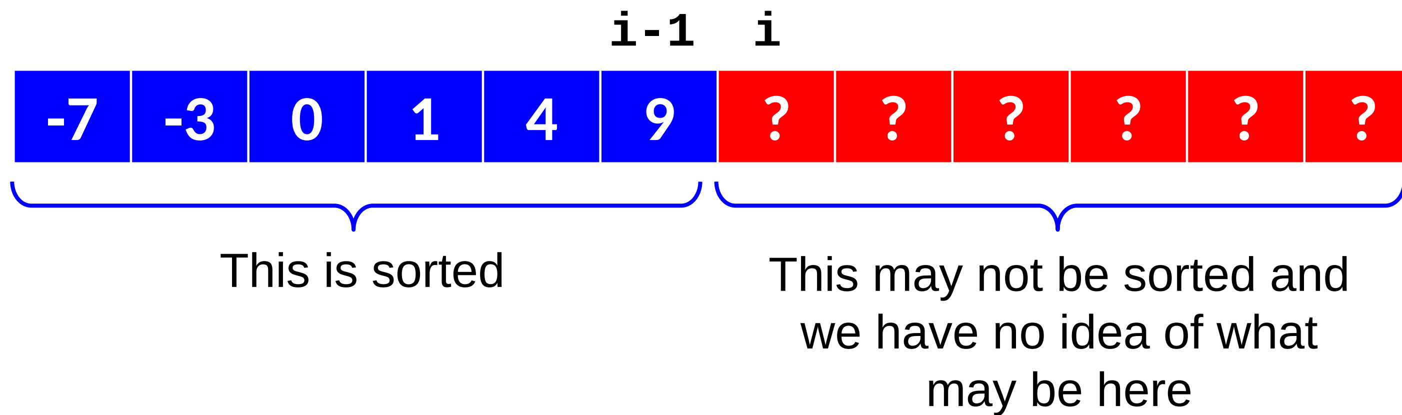
- Main idea:
  - At the i-th iteration, check element in a[i] and insert it in the right place in the sorted part of the list, moving all the other elements to the right

6 5 3 1 8 7 2 4

Source: wikipedia

Insertion sort

- Insertion sort maintains this invariant:



Insertion sort

```
# Pre: 'a' is a list of elements for which a
#      total order <= is defined
# Post: 'a' is sorted according to the total order
def insertion_sort(a) :
   # for each position on the list smallest element
   for i in range(len(a)) :
      # get next element to insert
      elem = a[i]
      # move elements to the right until we find one
      # smaller than 'elem'
      j = i
      while j>0 and elem<=a[j-1] :
          a[j] = a[j-1]
          j -= 1
      # place 'elem' in its place
      a[j] = elem
```

Insertion sort

Complexity

- At the i-th iteration, selection sort performs at most `i` comparisons, and `i+2` assignments
- The total number of comparisons for a list of size n is

$$1 + 2 + ... + (n-1) = n(n-1)/2$$

- The total number of assignments is at most $n^2/2$
- Total cost :

$$n(n-1)/2 + n^2/2 = n^2 - n/2 \approx n^2 \Rightarrow O(n^2)$$

Bubble Sort

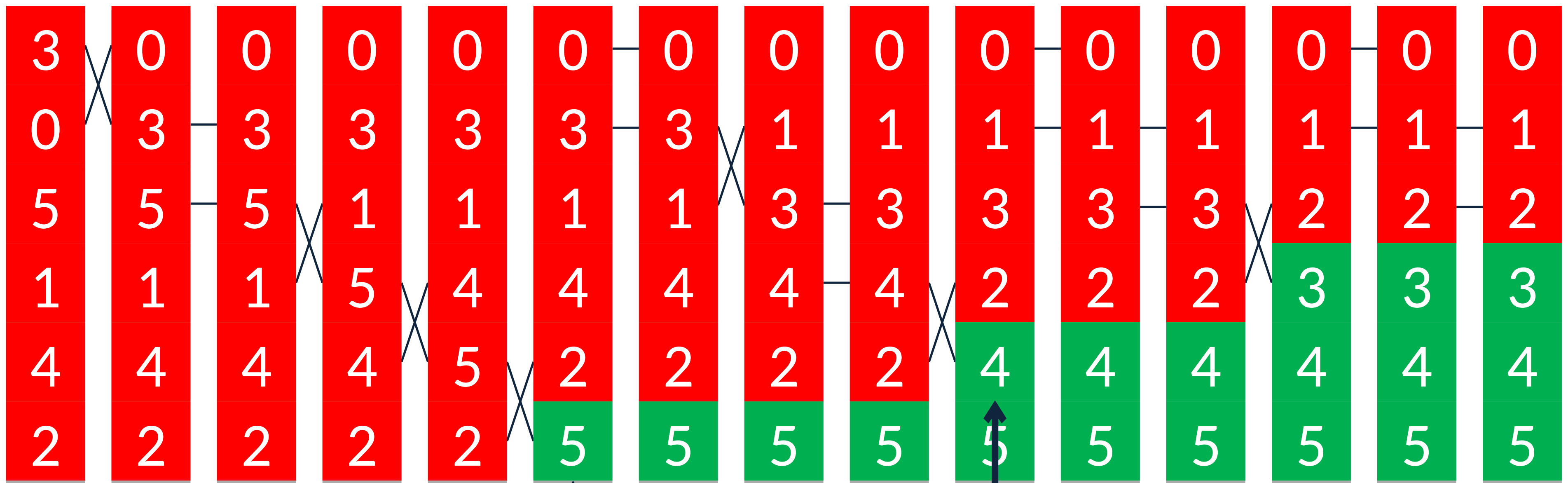
Bubble sort

- Main idea:
  - Traverse the vector many times, swapping adjacent elements when they are in the wrong order.

6 5 3 1 8 7 2 4

Source: wikipedia

Bubble sort



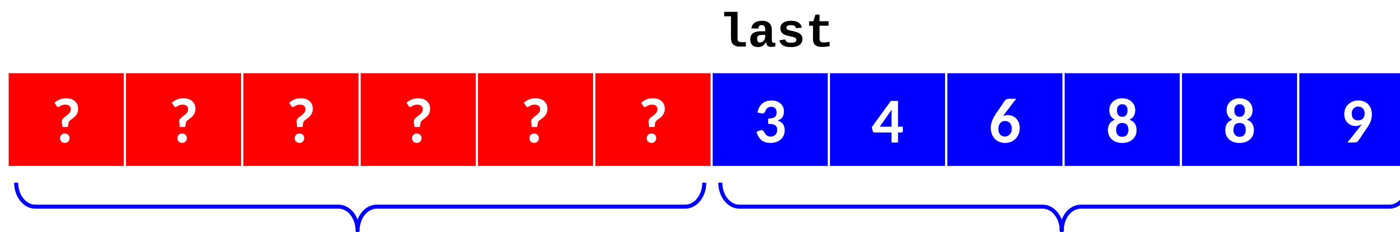
The largest element is well-positioned
after the first iteration.

The second largest element is
well-positioned after the second iteration.

The vector is sorted when no changes
occur during one of the iterations.

Bubble sort

- Bubble sort maintains this invariant:



this may not be sorted…
but all elements here are smaller than or equal to the elements in the sorted part

This is sorted
and contains the *n-last* largest elements

Bubble sort

```
# Pre: 'a' is a list of elements for which a
#      total order <= is defined
# Post: 'a' is sorted according to the total order
def bubble_sort(a) :
   # 'done' will be true when all the vector is sorted
   done = False
   # 'last' marks the position where sorted elements start
   last = len(a)-1
   while not done :
      done = True # ... unless proven otherwise
      for i in range(last) :
          if a[i]>a[i+1] : # unsorted element found:
             a[i],a[i+1] = a[i+1],a[i]  # exchange them.
             done = False               # list was not sorted.

      last -= 1  # element in 'last' is sorted now,
                 # move on to next pass
```

Bubble sort

Complexity

- Worst-case:
  - First pass makes n-1 comparisons and at most n-1 swaps
  - Second pass makes n-2 comparisons and at most n-2 swaps
  - ...
  - Last pass makes 1 comparison and at most 1 swap
- The total number of comparisons for a list of size n is

  $(n-1)+(n-2)+...+2+1 = n(n-1)/2$
- The total number of swaps is also $n(n-1)/2$, i.e $3n(n-1)/2$ assignments
- Total cost :

  $4 \cdot n(n-1)/2 = 2n^2 - 2n \approx 2n^2 \Rightarrow O(n^2)$

Comparison

Algorithm	steps	Largest factor	Complexity class
Selection sort	$(n^2 + 5n - 6)/2$	$n^2/2$	$O(n^2)$
Insertion sort	$n^2 - n/2$	n^2	$O(n^2)$
Bubble sort	$2n^2 - 2n$	$2n^2$	$O(n^2)$

- Despite having the same $O(n^2)$ complexity class, selection sort is x2 faster than insertion sort and x4 faster than bubble sort
- However, if the list is already (almost) ordered, selection sort will take the same time, but the other two will be much faster, about $O(n)$
- In any case, quadratic algorithms are not suitable for big data, and faster approaches are required.

Fast sorting algorithms

Merge sort

Merge sort

- Main idea
  - Split the list in halves until they are small enough.
  - Sort each half.
  - Merge results.

Source: wikipedia

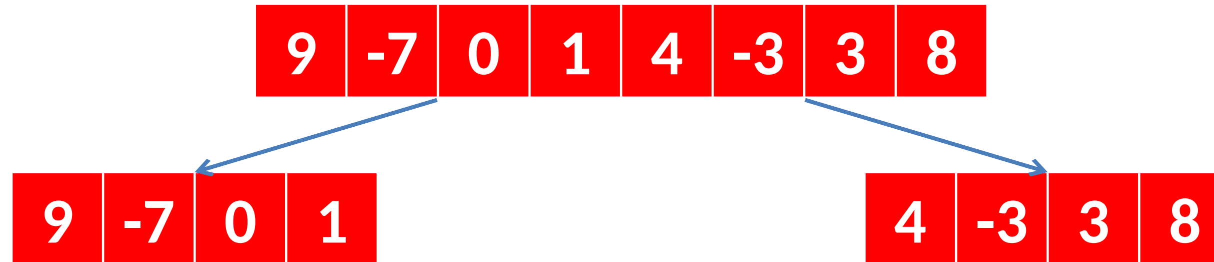
Merge sort

→ : split
→ : merge

9	-7	0	1	4	-3	3	8

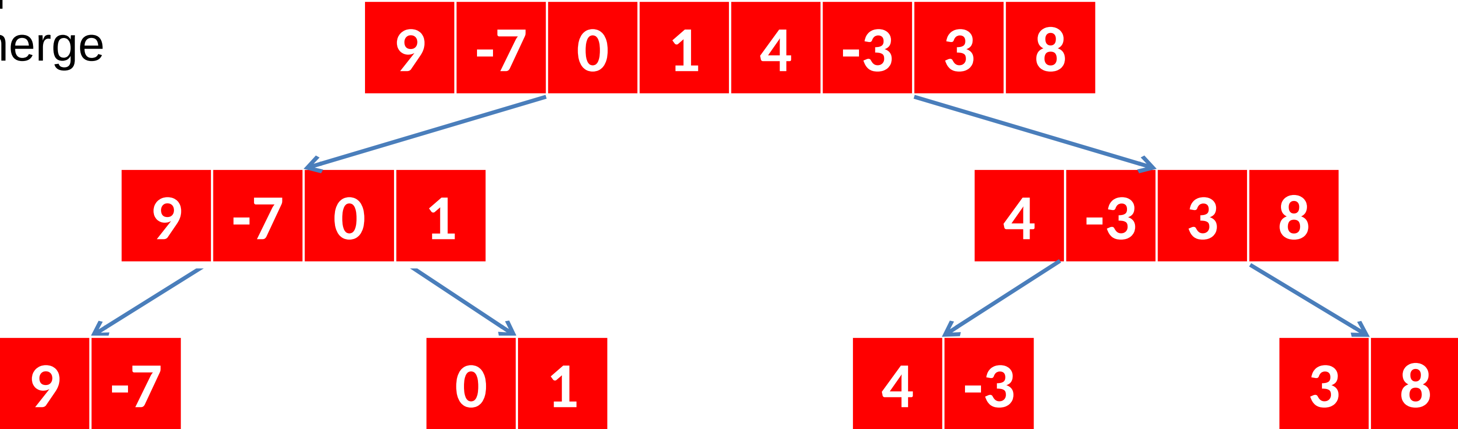
Merge sort

→ : split
→ : merge

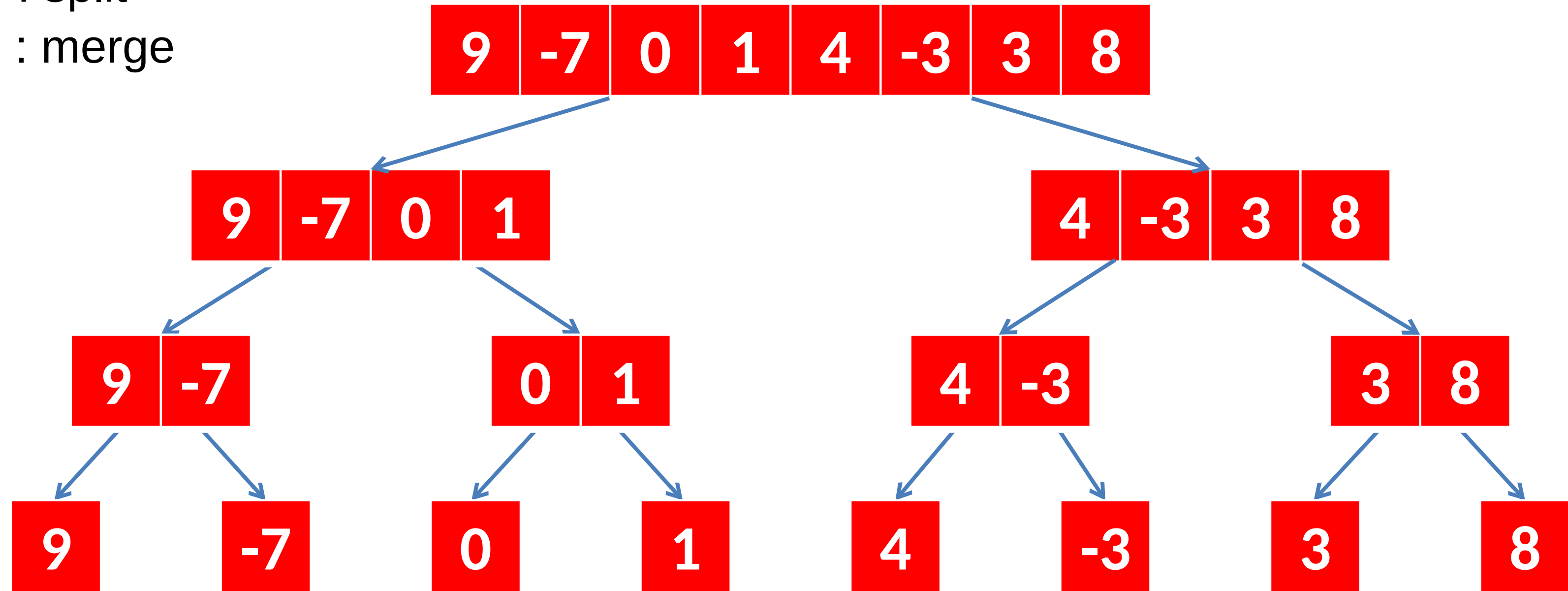


Merge sort

→ : split
→ : merge

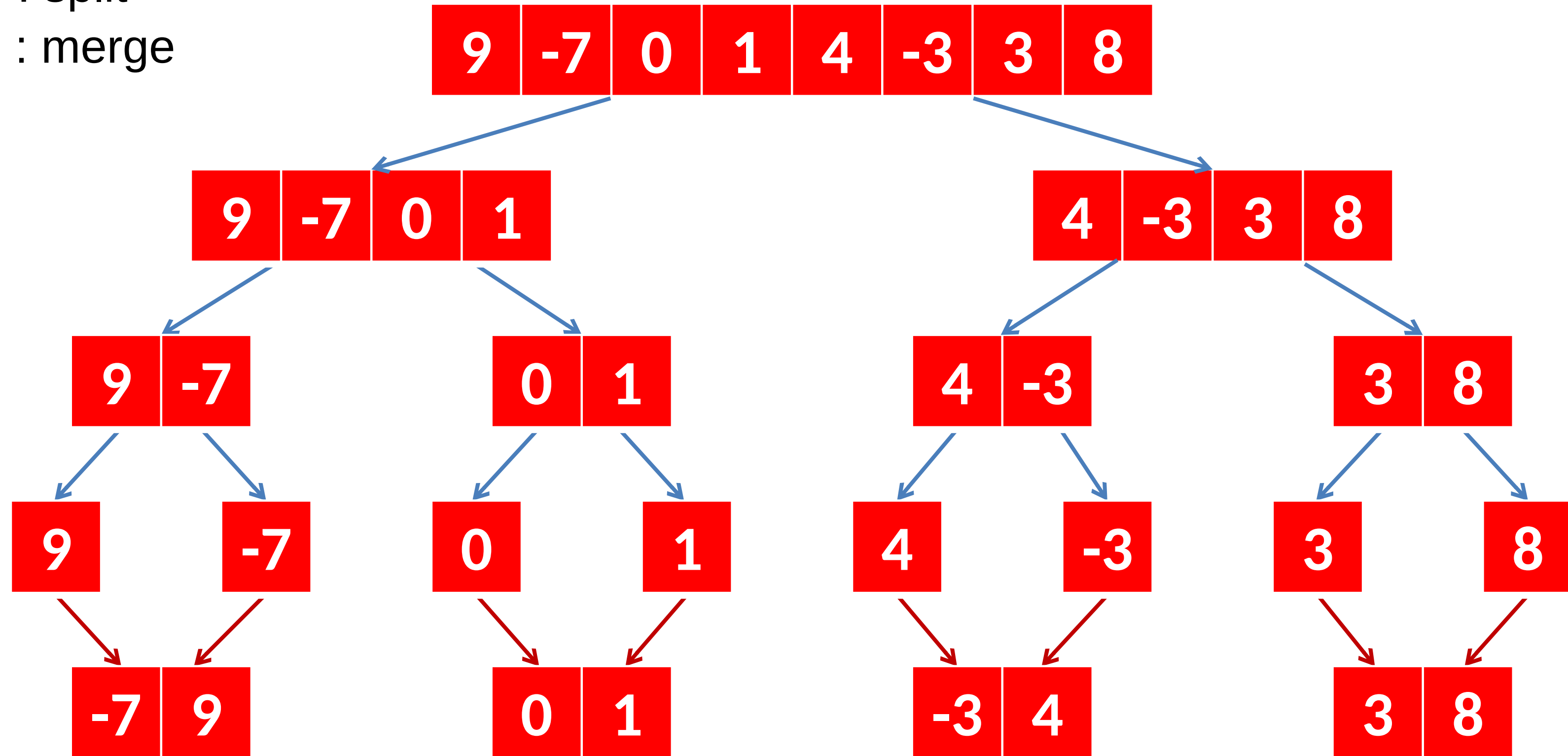


Merge sort



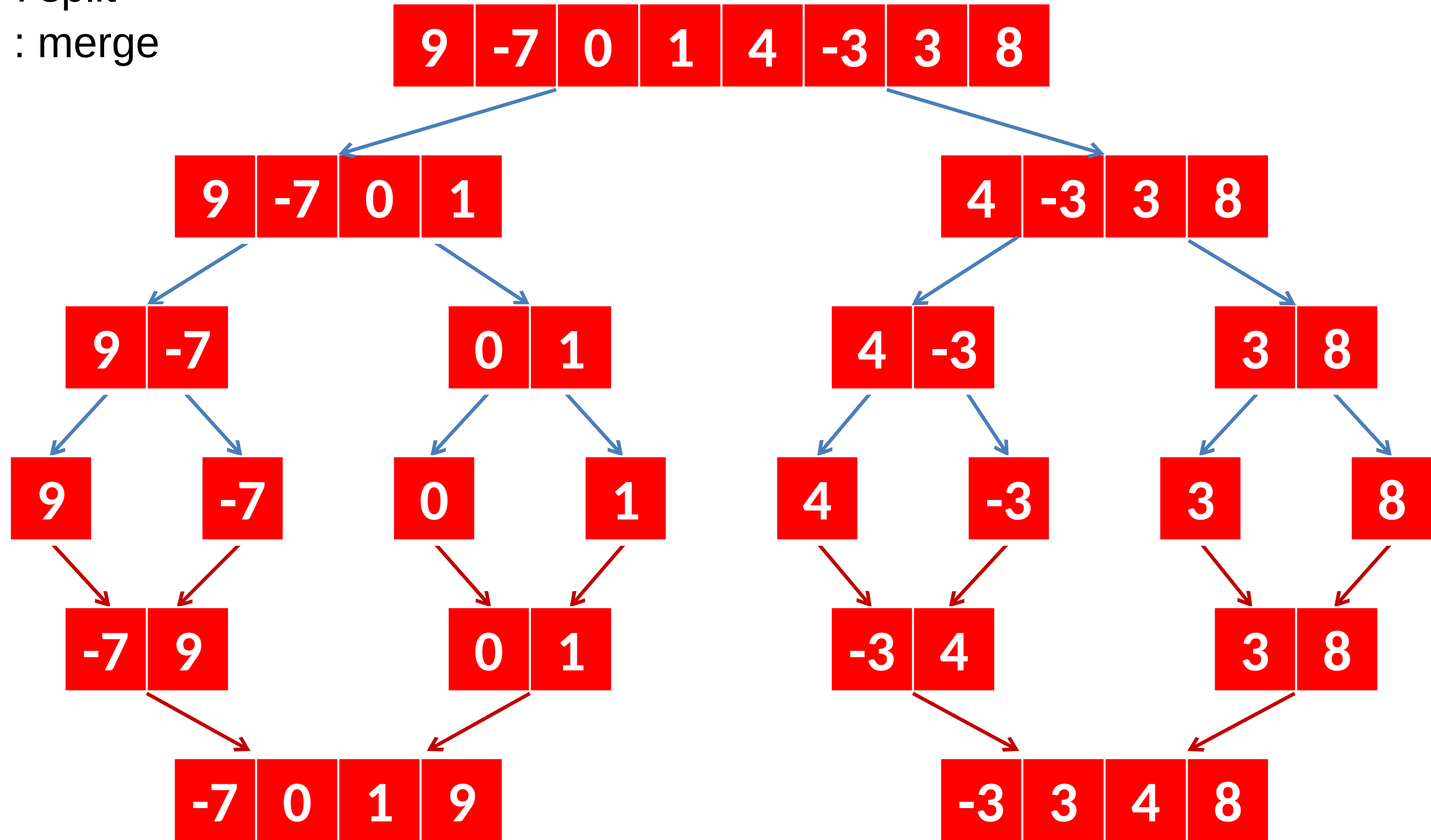
Merge sort

→ : split
→ : merge

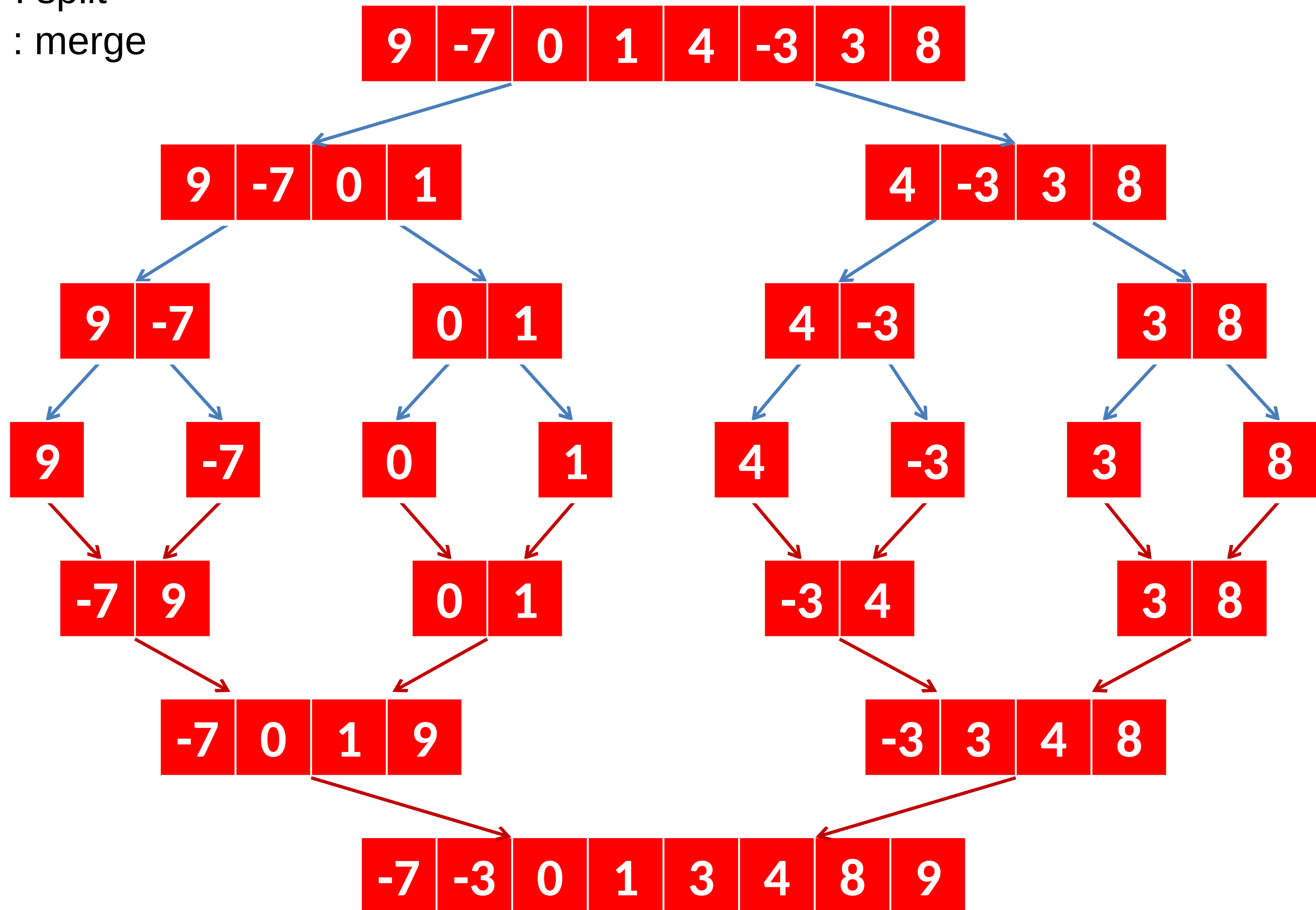


Merge sort

→ : split
→ : merge



Merge sort



Merge sort

- Complexity
  - First merge level (size 1 → size 2): n steps
  - Second merge level (size 2 → size 4): n steps
  - Third merge level (size 4 → size 8): n steps
  - etc

  - Number of merge levels: $log_2 n$
  - Total number of comparisons: $n \cdot log_2 n$ ⇒ $O(n \cdot log\ n)$

Quick sort

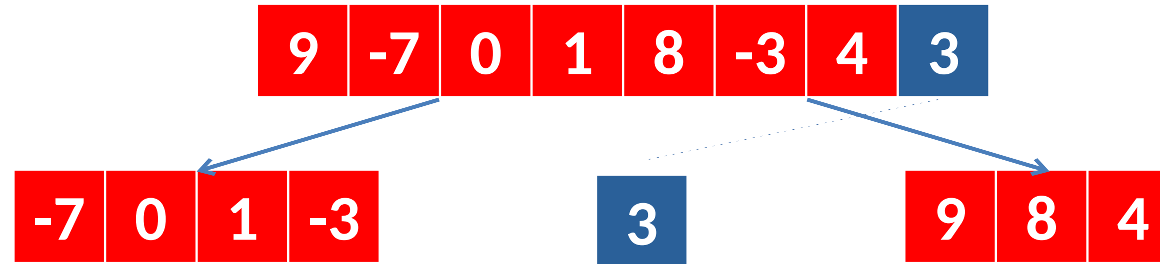
Quick sort

→ : split
→ : sort
→ : join

9	-7	0	1	8	-3	4	3

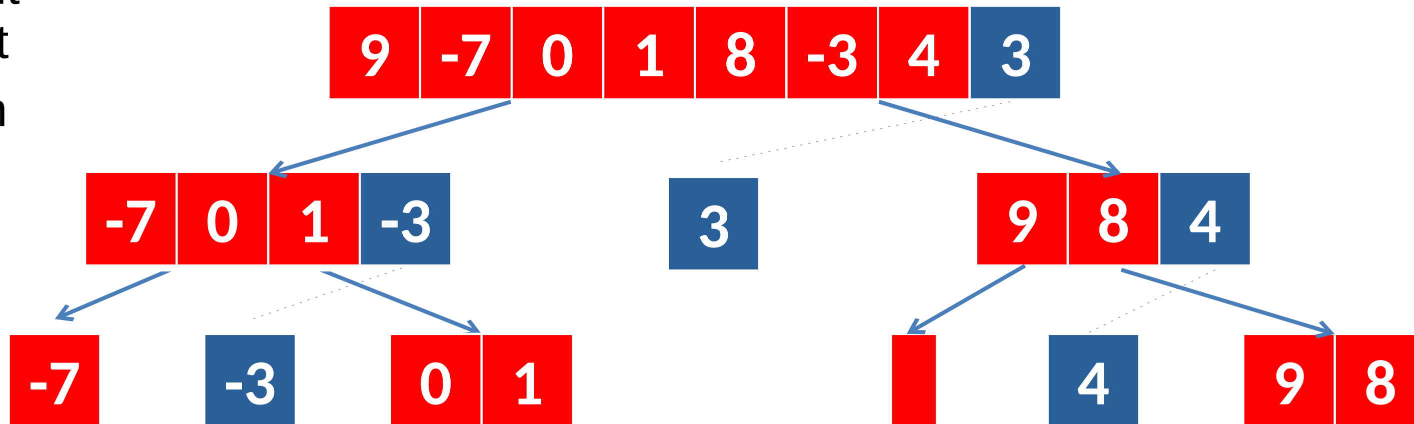
Quick sort

: split
: sort
: join

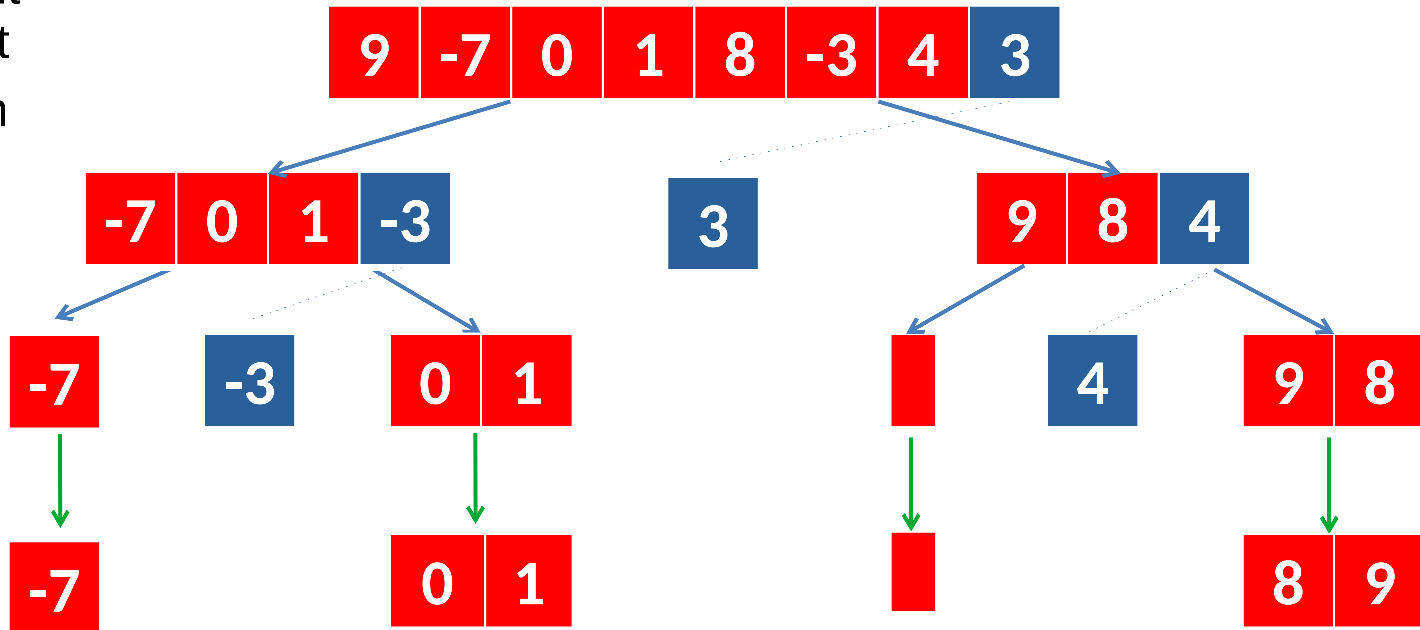


Quick sort

→ : split
→ : sort
→ : join

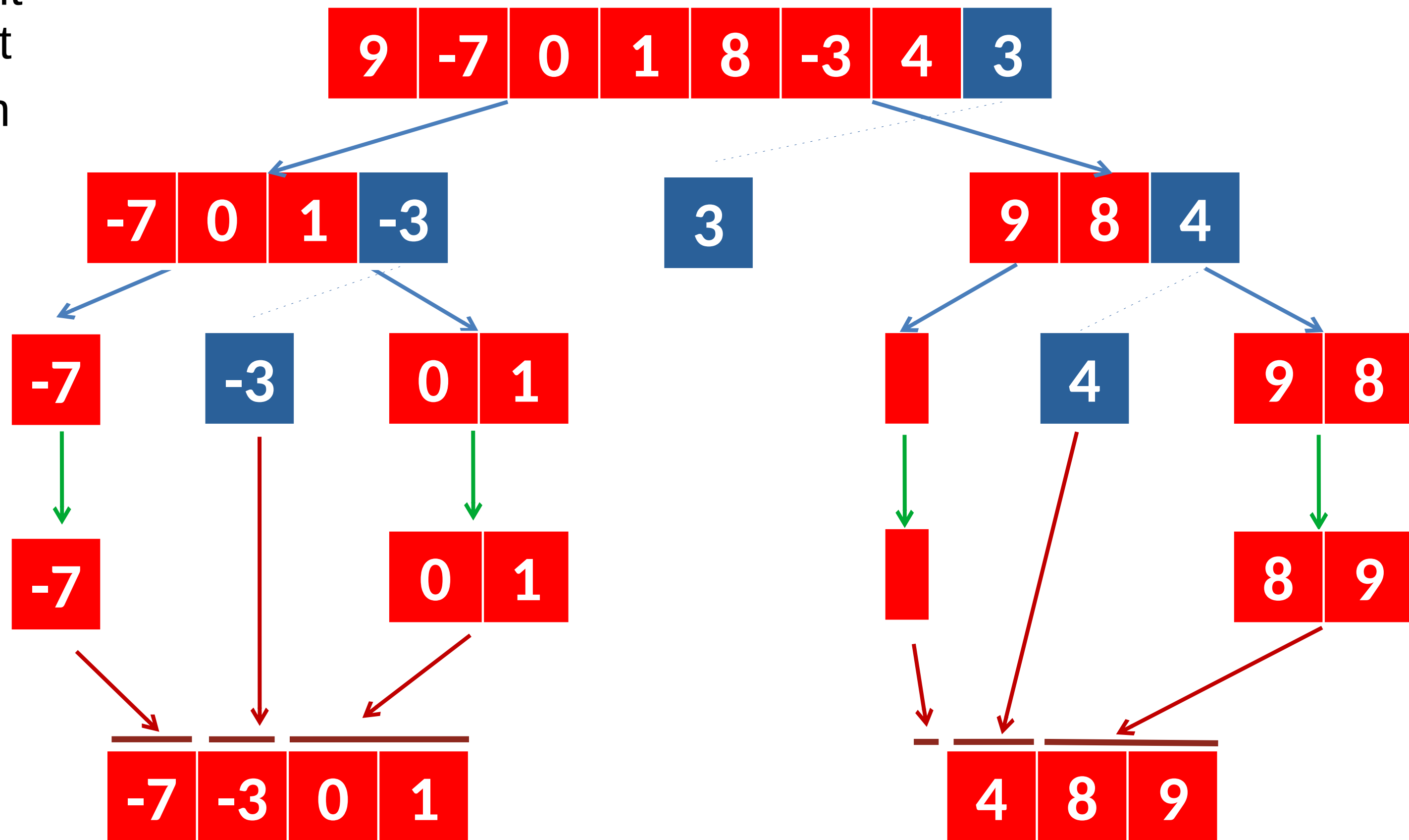


Quick sort

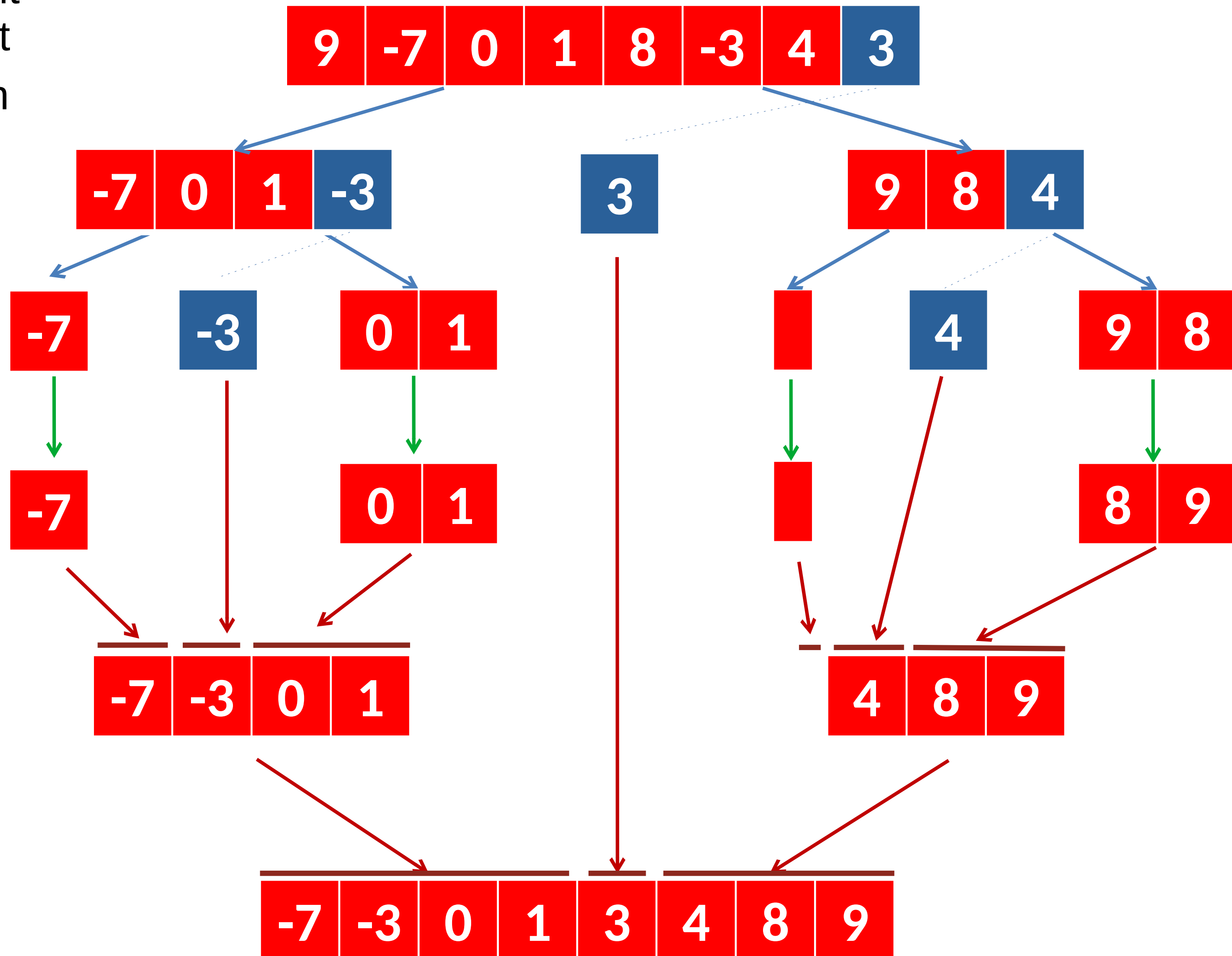


Quick sort

: split
: sort
: join



Quick sort



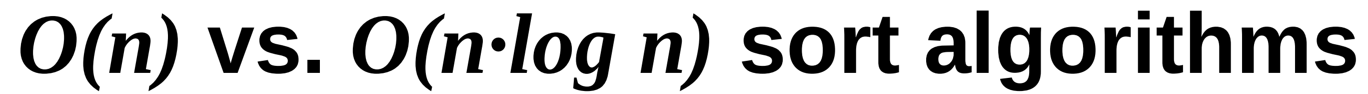
Quick sort

- Complexity
  - First split level (size 1 → size 2): n steps
  - Second split level (size 2 → size 4): n steps
  - Third split level (size 4 → size 8): n steps
  - etc

  - Number of split levels: $log_2 n$
  - Total number of comparisons: $n \cdot log_2 n$ ➡ $O(n \cdot log\ n)$

Comparison

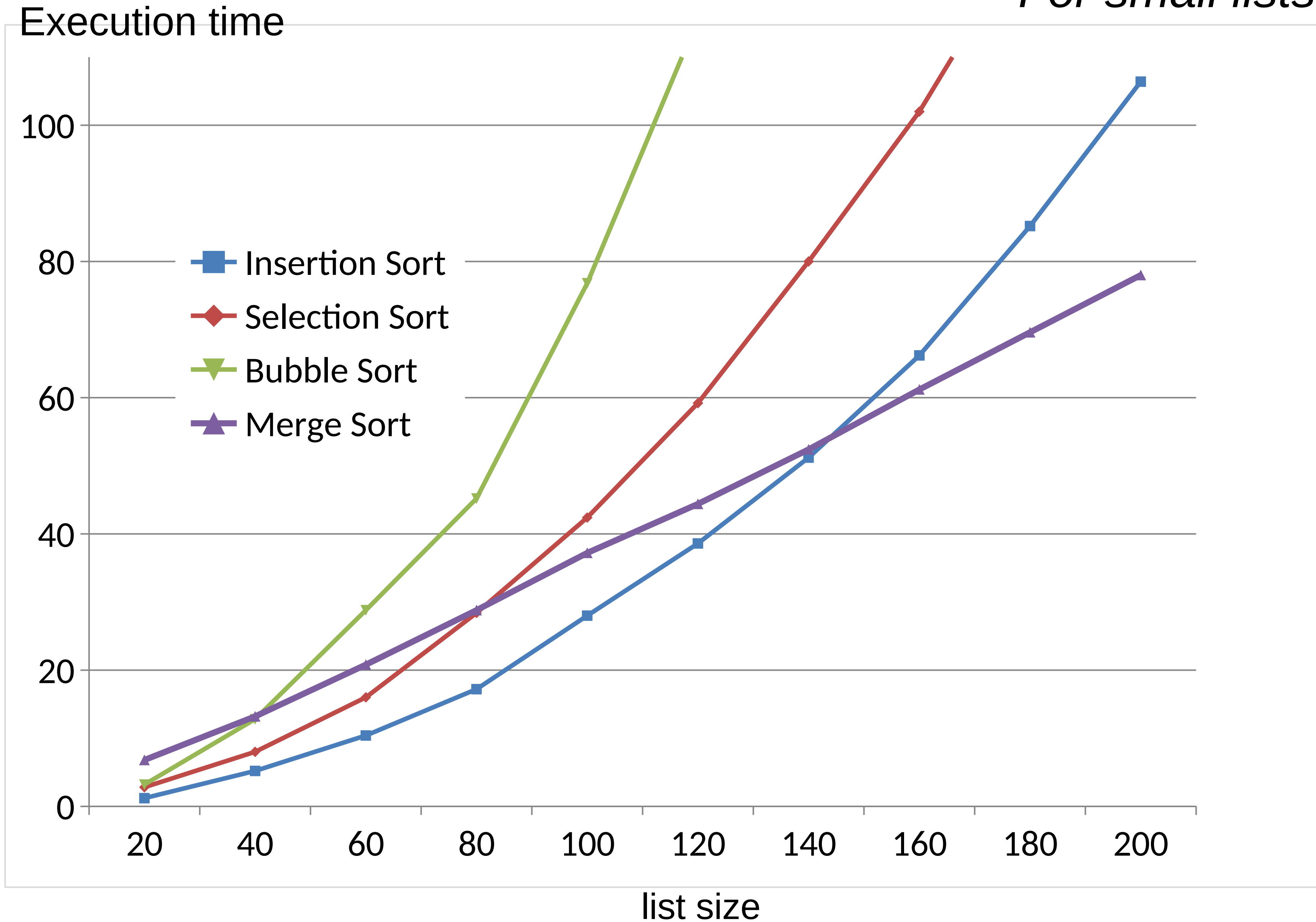
- Merge sort performs fast split levels (divide in half), but $O(n)$ cost merge levels, where two sorted lists have to be intertwined to produce a unique sorted list
- Quick sort has fast fusion levels (just putting togheter two lists), but $O(n)$ cost split levels, where a lists has to be splitted in two, selecting elements larger/smaller than the pivot.
- It can be proven that there is not a general sorting algorithm faster than $O(n \cdot \log n)$
- Faster algorithms exist (radix sort, bucket sort) if the number of different possible values for the list elements is finite and known.


O(n) vs. O(n·log n) sort algorithms

Comparison of sort algorithms

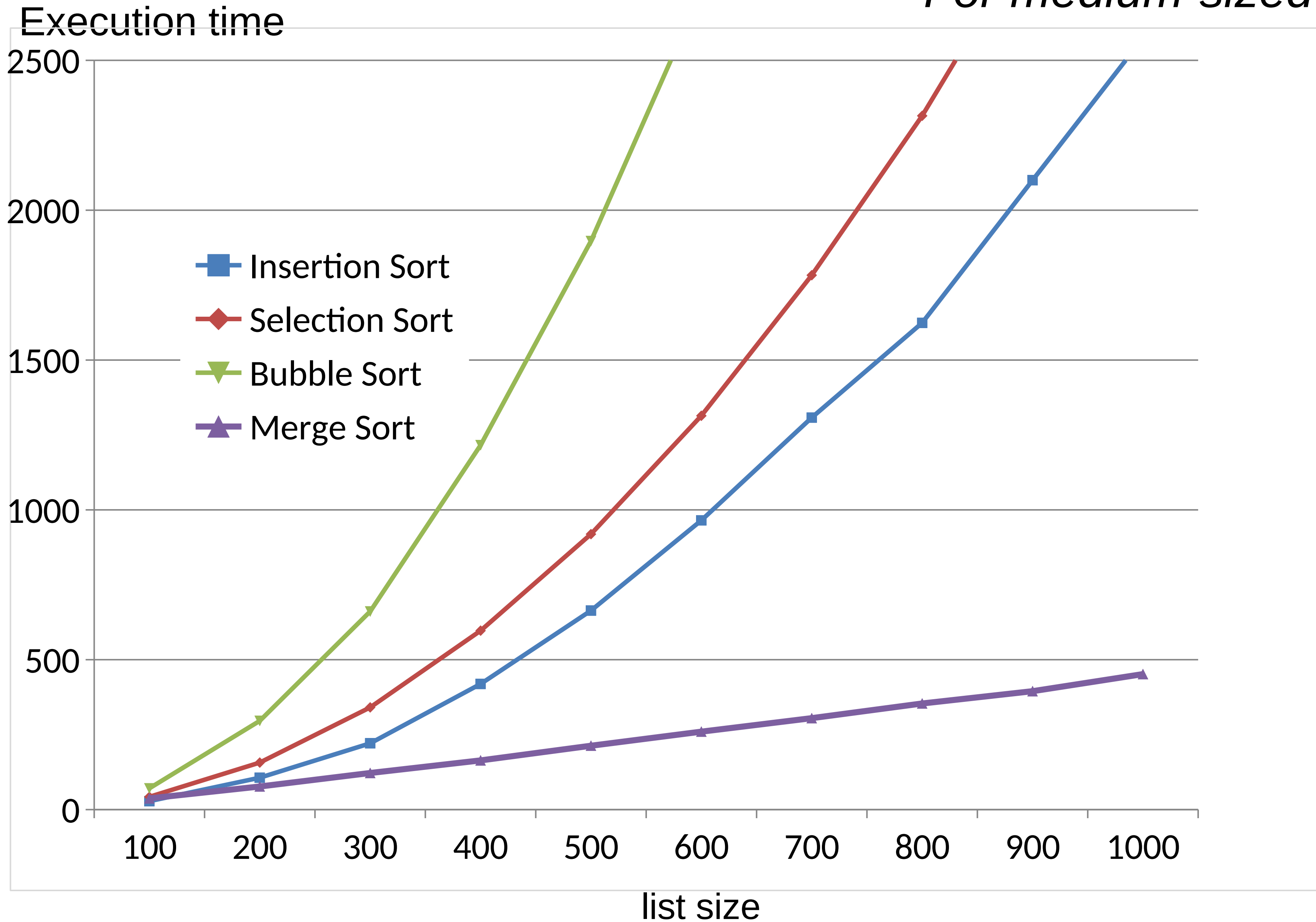
	list length				
Algorithm	**10**	**100**	**1,000**	**10,000**	**100,000**
Bubble sort $\approx 2n^2$	**200**	**20,000**	**2,000,000**	**200,000,000**	**20,000,000,000**
Insertion sort $\approx n^2$	**100**	**10,000**	**1,000,000**	**100,000,000**	**10,000,000,000**
Selection sort $\approx n^2/2$	**50**	**5,000**	**500,000**	**50,000,000**	**5,000,000,000**
Merge Sort, Quick sort $\approx n \cdot \log_2 n$	**33**	**664**	**9,965**	**132,877**	**1,660,964**

Comparison of sort algorithms



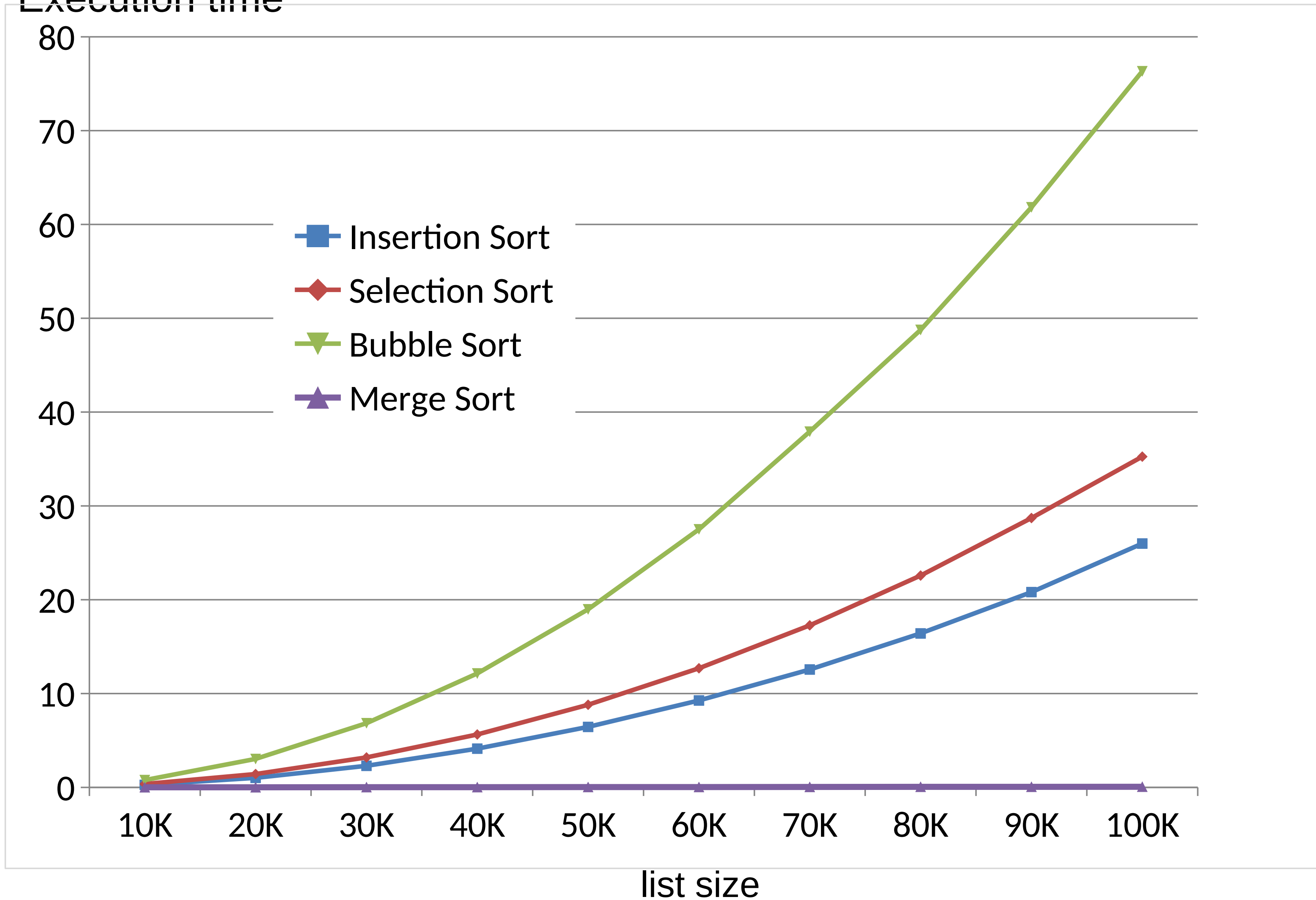
Comparison of sort algorithms

For medium-sized lists



Comparison of sort algorithms

For large lists



Sorting is not always a good idea

- Example: Find the minimum element in a list

```
m = a[0]
for x in a[1:] :
    if x<m :
        m = x
```

n steps (1 comparison for each element in the list)

```
a.sort()
m = a[0]
```

$n \cdot log(n)$ steps (sorting time)

- A simpler program is not always the fastest.

Things to remember

- Strategies used by Insertion/selection sort may be useful to handle lists (e.g. to insert a new element in a sorted list)
- Sorting must use $O(n \cdot log_2 n)$ algorithms for medium or large lists.
- Even with efficient algorithms, sorting is costly and must be avoided or reduced to the minimum necessary.