Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU22012: Data Structures and Algorithms II

Lecture 2: Sorting Algorithms

Dr Anthony Ventresque

Outline

- Sorting Problem
- Various algorithms:
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Quick sort
  - Merge sort

Take home message:
Bubble sort, selection sort and insertion sort are not efficient. Quick sort and merge sort are better: O(n log n) in most cases

Sorting

Sort data in order

- Numbers in ascending/descending order
- Strings alphabetically
- Dates chronologically
- etc

Total order

- Ascending $x_0 \leq x_1 \leq x_2 \leq x_3 \leq .... \leq x_{n-1}$

- Descending $x_0 \geq x_1 \geq x_2 \geq x_3 \geq .... \geq x_{n-1}$

Sorting

Input: Sequence n of elements in no particular order
Output: ***Sequence rearranged in as-/de-scending order of elements' values***

Motivation: Fundamental in ***many real-world applications***
Very popular exercise to learn the concepts behind algorithms and data structures

- Numbers in ascending/descending order
- Strings alphabetically
- Dates chronologically
- etc

Ascending $x_0 \leq x_1 \leq x_2 \leq x_3 \leq .... \leq x_{n-1}$
Descending $x_0 \geq x_1 \geq x_2 \geq x_3 \geq .... \geq x_{n-1}$

There are literally hundreds of sorting algorithms

Total Order

$$x_0 \leq x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_{n-1}$$

Is a binary relation ≤ that satisfies

- Antisymmetry: if both $v \leq w$ and $w \leq v$, then $v = w$.
- Transitivity: if both $v \leq w$ and $w \leq x$, then $v \leq x$.
- Totality: either $v \leq w$ or $w \leq v$ or both.

Performance Analysis

Cost models

- Running time
- Memory cost

Methods to measure/express

- Tilde notation, $T(n)$ – counting number of executions of certain operations as a function of input size n

Order of growth classification

- Big Theta $\Theta(n)$ – asymptotic order of growth
- Big Oh $O(n)$ - upper bound
- Big Omega $\Omega(n)$ – lower bound

Performance Analysis

Time complexity

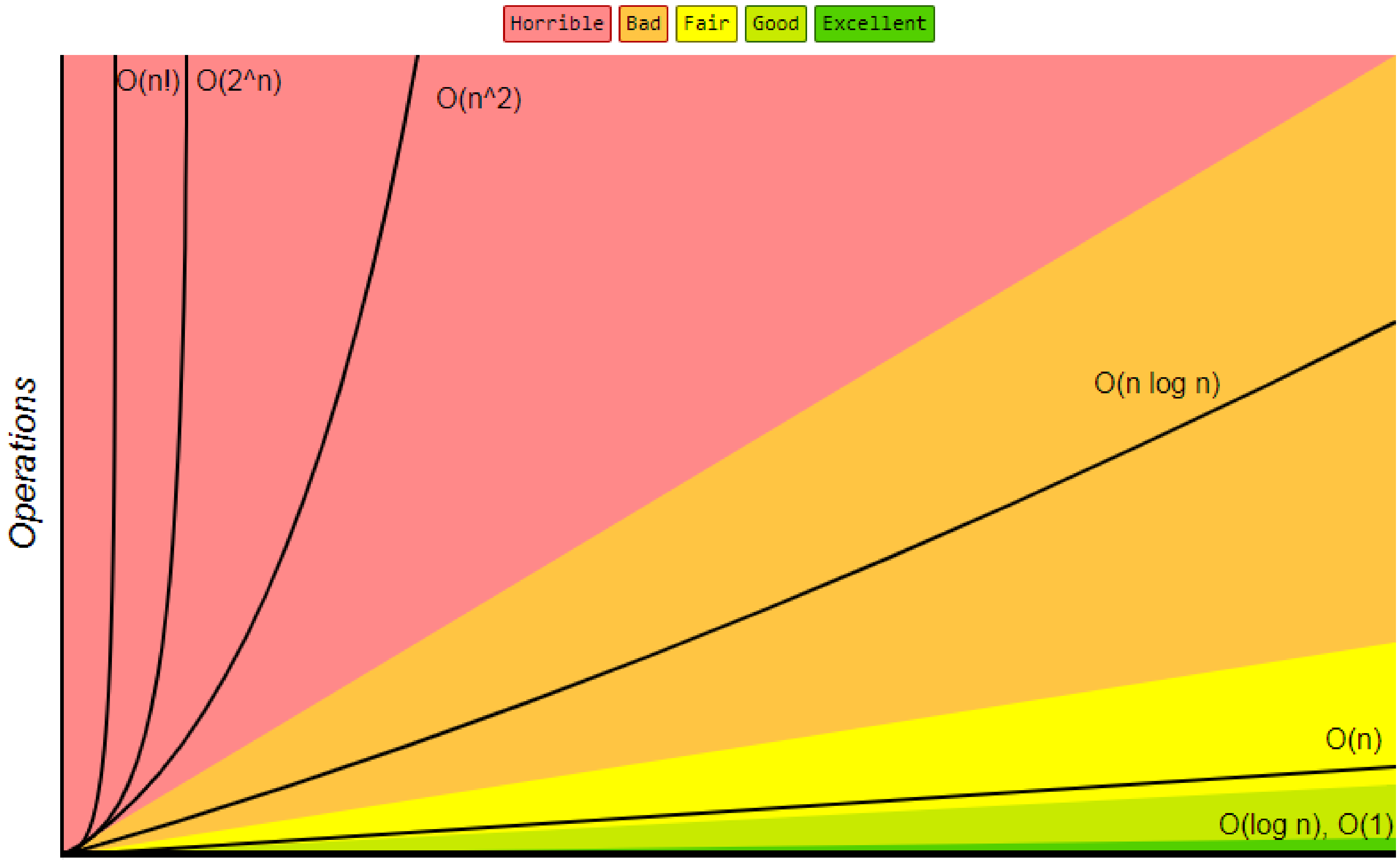
- Worst Case Analysis – usually done
  - Upper bound on running time of an algorithm
  - Must know the case that causes the maximum number of operations to be performed, eg in linear search, if the element is not in the array
- Average – not easy to do in practice
  - Take all possible inputs and calculate computing time for all of the inputs, and average
  - Must know/predict distribution of cases
- Best – is it any use if worst case bad?
  - Lower bound on running time of an algorithm
  - Must know the case that causes the minimum number of operations to be performed

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	**constant**	`a = b + c;`	statement	add two numbers	1
$\log N$	**logarithmic**	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	**linear**	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	**quadratic**	`for (int i = 0; i < N; i++)` `for (int j = 0; j < N; j++)` `{ ... }`	double loop	check all pairs	4
N^3	**cubic**	`for (int i = 0; i < N; i++)` `for (int j = 0; j < N; j++)` `for (int k = 0; k < N; k++)` `{ ... }`	triple loop	check all triples	8
2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

44

Big-O Complexity Chart



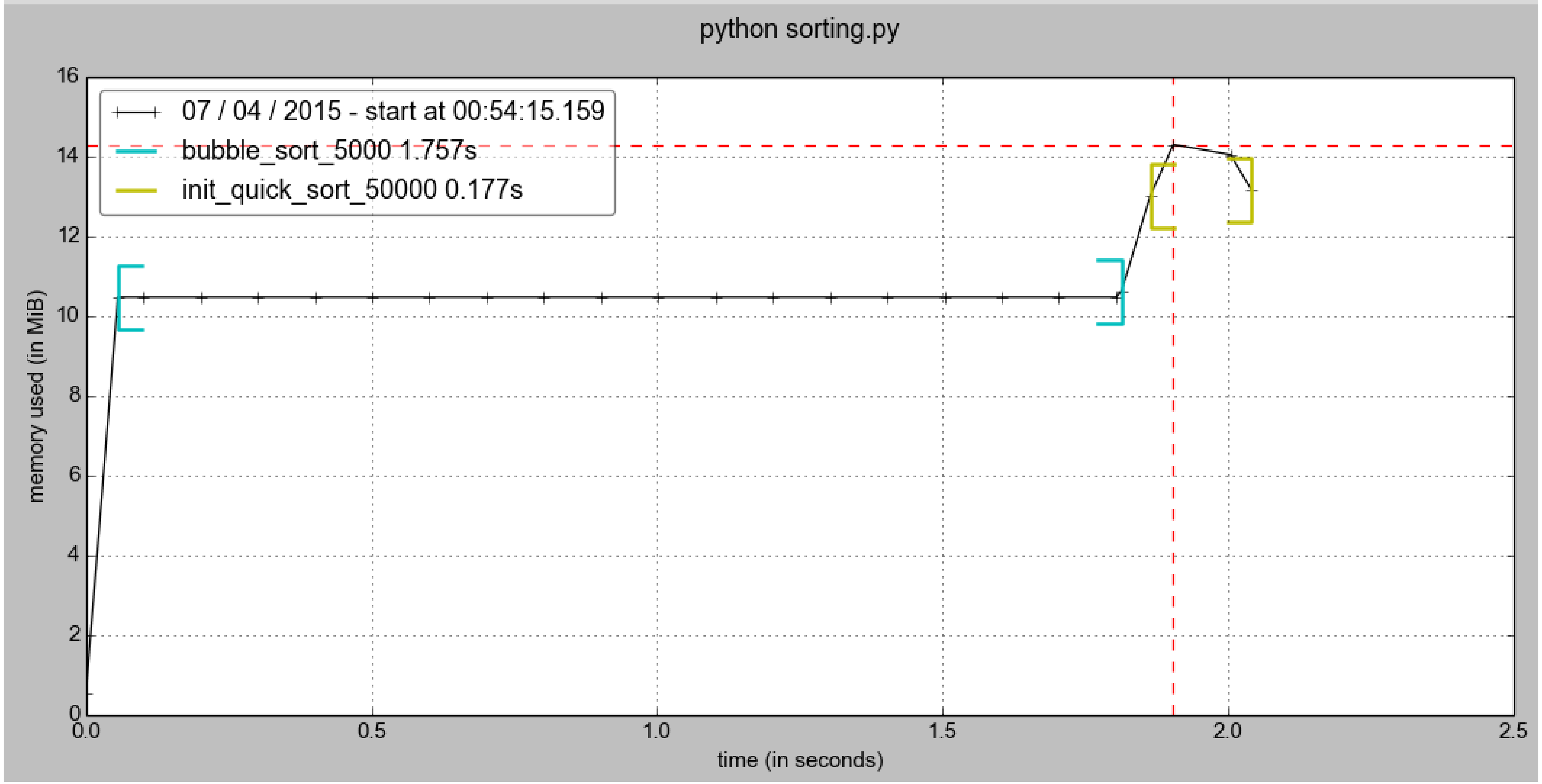
Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort ($N \log N$)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Performance Analysis



Why Do We Need So Many of Them?

No Free Lunch Theorem
Different applications/different behaviour based on input
Examples

- Merge sort – useful for linked lists
- Quicksort – excellent average-case behaviour
- Insertion sort – good if your list is already almost sorted
- Bubble sort – if small enough data set, it is the simplest to implement

Also, a handy way to learn different algorithm design strategies on the same example!

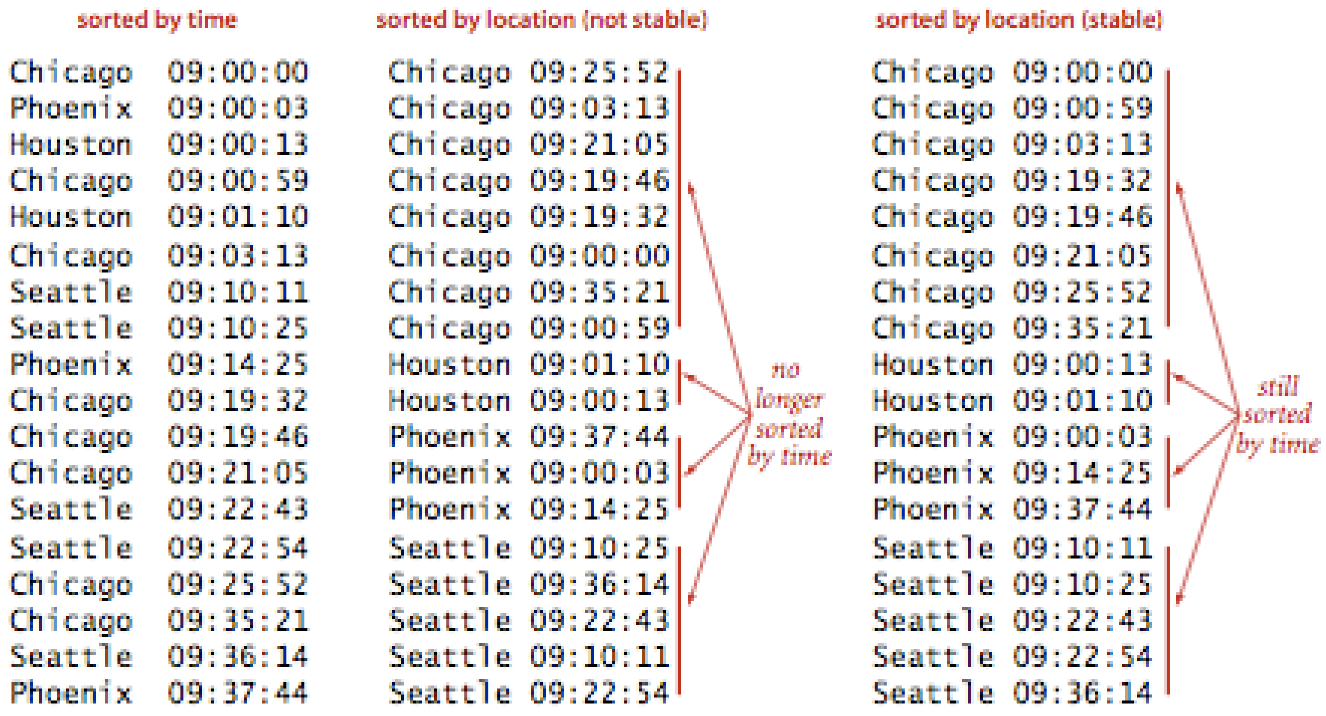
Stability of Sorting Algorithms

Stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted
Do we care?

- NO: When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key
- NO: If all keys are different.
- YES: if duplicate keys and want to maintain original order by eg secondary key.
- When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. Stability is also not an issue if all keys are different.

Stability of Sorting Algorithms

Stable sorting algorithms: Insertion sort, bubble sort, merge sort



sorted by time		sorted by location (not stable)		sorted by location (stable)	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

Stability when sorting on a second key

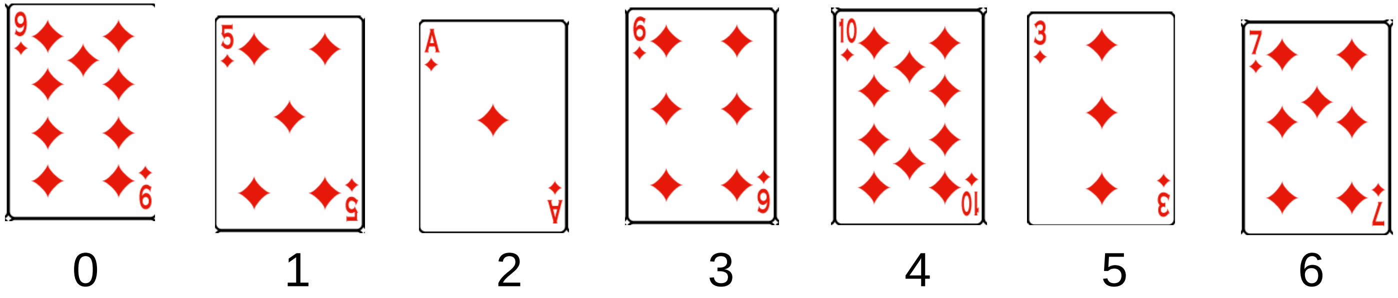
Memory requirements/In-place algorithms

Transforms input without additional auxiliary data structure, eg array
A small amount of extra storage space is allowed for auxiliary variables
The input is usually overwritten by the output as the algorithm executes
In-place algorithm updates input sequence only through replacement or swapping of elements

Affects space complexity of an algorithm
Selection, insertion, shell, quick

Bubble sort

1. Get a hand of unsorted cards
2. Repeat steps 3 through 5 until nothing happens
3. for every couple of neighbouring cards (left-right)
4. If the figure on the left is bigger than the one on the right
5. Swap cards
6. Stop



Bubble Sort

- bubble_sort sorts a sequence (ADT) of values
- Based on a structured pattern of ***comparison-exchange (CE)*** operations
- comparison_exchange(i): Take value in two adjacent slots in the sequence and if the values are out of order (i.e., the larger before the smaller), then swap them around:

$\cdots\underbrace{27\ 13}\cdots \rightarrow \cdots 13\ 27\cdots$(Swap)

$\cdots\underbrace{27\ 44}\cdots \rightarrow \cdots 27\ 44\cdots$(No swap)

Bubble Sort

- bubble_sort involves multiple sweeps through list
- Sweep: For an n-element list, apply n – 1 comparison-exchanges to each pair of adjacent position in left-to-right order.

27	***13***	44	15	12	99	63	57
13	***27***	***44***	15	12	99	63	57
13	27	***44***	***15***	12	99	63	57
13	27	15	***44***	***12***	99	63	57
13	27	15	12	***44***	***99***	63	57
13	27	15	12	44	***99***	***63***	57
13	27	15	12	44	63	***99***	***57***
13	27	15	12	44	63	57	99

Bubble Sort

- bubble_sort involves n − 1 sweeps through the array

Sweep=0	27	13	44	15	12	99	63	57
Seep=1	13	27	15	12	44	63	57	**99**
Sweep=2	13	15	12	27	44	57	**63**	**99**
Sweep=3	13	12	15	27	44	**57**	**63**	**99**
Sweep=4	12	13	15	27	**44**	**57**	**63**	**99**
Sweep=5	12	13	15	**27**	**44**	**57**	**63**	**99**
Sweep=6	12	13	**15**	**27**	**44**	**57**	**63**	**99**
End:	12	**13**	**15**	**27**	**44**	**57**	**63**	**99**

Bubble Sort Pseudo Code

```
algorithm bubble_sort
Input: A an array
Output: A is sorted
for s = 1 to n-1 do
        for current = 0 to n-2 do
                if A[current] > A[current + 1] then
                        swap A[current] and A[current+1]
                endif
        endfor
endfor
```

Bubble Sort Pseudo Code

Multiple sweeps

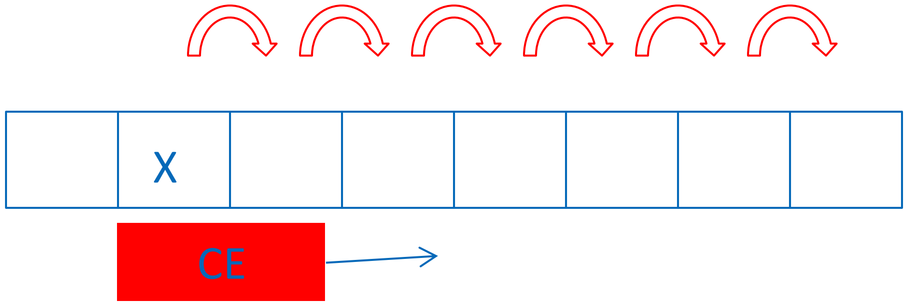
1 sweep

Comparison-exchange (CE)

```
algorithm bubble_sort
Input: A an array
Output: A is sorted
for s = 1 to n-1 do
        for current = 0 to n-2 do
                if A[current] > A[current + 1] then
                        swap A[current] and A[current+1]
                endif
        endfor
endfor
```

Useful Observation

- Consider largest value X:
  - No CE can move X leftwards
  - Every CE with X on LHS moves it rightwards
- First sweep pushes X into very last slot in the list (where it belongs)



- CEs of subsequent sweeps leave it there

Analysis

```
algorithm bubble_sort
Input: A an array
Output: A is sorted
for s = 1 to n-1 do #1 op per loop
      for current = 0 to n-2 do #1 op per loop per loop
            if A[current] > A[current + 1] then
            #3 op per loop per loop
                  swap A[current] and A[current+1]
                  #3 op per loop per loop
            endif
      endfor
endfor
```

$T(n) = 7(n-1)^2 + n-1$, which is $O(n^2)$

Optimising Bubble Sort

Whenever the array is sorted, there is no need to continue running bubble_sort!

Sweep=0	27	13	44	15	12	99	63	57
Seep=1	13	27	15	12	44	63	57	**99**
Sweep=2	13	15	12	27	44	57	**63**	**99**
Sweep=3	13	12	15	27	44	**57**	**63**	**99**
Sweep=4	12	13	15	27	**44**	**57**	**63**	**99**
~~Sweep=5~~	~~12~~	~~13~~	~~15~~	~~**27**~~	~~**44**~~	~~**57**~~	~~**63**~~	~~**99**~~
~~Sweep=6~~	~~12~~	~~13~~	~~**15**~~	~~**27**~~	~~**44**~~	~~**57**~~	~~**63**~~	~~**99**~~
End:	12	**13**	**15**	**27**	**44**	**57**	**63**	**99**

Optimising Bubble Sort (1)

```
algorithm bubble_sort
Input: A an array
Output: A is sorted
for s = 1 to n-1 do
        swapped ← False
        for current = 0 to n-2 do
                if A[current] > A[current + 1] then
                        swap A[current] and A[current+1]
                        swapped ← True
                endif
        endfor
        if not swapped then
                finish
        endif
endfor
```

Optimising Bubble Sort (2)

- After the i-th pass the last (i-1) items are sorted: no need to go through them!

Sweep=0	27	13	44	15	12	99	63	57
Seep=1	13	27	15	12	44	63	57	**99**
Sweep=2	13	15	12	27	44	57	**63**	**99**
Sweep=3	13	12	15	27	44	**57**	**63**	**99**
Sweep=4	12	13	15	27	**44**	**57**	**63**	**99**
Sweep=5	12	13	15	**27**	**44**	**57**	**63**	**99**
Sweep=6	12	13	**15**	**27**	**44**	**57**	**63**	**99**
End:	12	**13**	**15**	**27**	**44**	**57**	**63**	**99**

Optimising Bubble Sort (2)

```
algorithm bubble_sort
Input: A an array
Output: A is sorted
for s = 1 to n-1 do
     swapped ← False
     for current = 0 to n – s - 2  do
          if A[current] > A[current + 1] then
               swap A[current] and A[current+1]
               swapped ← True
          endif
     endfor
     if not swapped then
          finish
     endif
endfor
```

Selection Sort

- selection_sort iteratively looks for the minimum value in an array
- Then swaps it with the leftmost (unsorted) item

Original array:	27	13	44	15	12	99	63	57
	12	13	44	15	*27*	99	63	57
	12	***13***	44	15	27	99	63	57
	12	**13**	***15***	*44*	27	99	63	57
	12	**13**	**15**	***27***	*44*	99	63	57
	12	**13**	**15**	**27**	***44***	99	63	57
	12	**13**	**15**	**27**	**44**	***57***	63	*99*
sorted array:	**12**	**13**	**15**	**27**	**44**	**57**	***63***	**99**

Selection Sort (pseudo-code)

```
Algorithm selection_sort
Input: A an array
Output: A is sorted
for j = 0 to n-2 do
     min ← j
     for i = j + 1 to n-1 do
          if A[min] > A[i] then
               min ← i
          endif
     endfor
     swap a[min], a[j]
endfor
```

Selection Sort (pseudo-code)

For each cell of the array

Find the min

```
Algorithm selection_sort
Input: A an array
Output: A is sorted
for j = 0 to n-2 do
     min ← j
     for i = j + 1 to n-1 do
          if A[min] > A[i] then
               min ← i
          endif
     endfor
     swap a[min], a[j]
endfor
```

Analysis

```
Algorithm selection_sort
Input: A an array
Output: A is sorted
for j = 0 to n-2 do # 1 op per loop
     min ← j # 1 op per loop
     for i = j + 1 to n-1 do # 1 op per loop per loop
          if A[min] > A[i] then # 3 op per loop per loop
               min ← i # 1 op per loop per loop
          endif
     endfor
     swap a[min] and a[j] # 3 op per loop
endfor
```

$T(n) = 5n^2 + 4n$ which is $O(n^2)$

Insertion Sort

- insertion_sort shares with selection_sort the idea of increasing the sorted section at the start of the array
- insertion_sort takes the next item and puts it at the correct position

27	13	44	15	12	99	63	57
27	13	44	15	12	99	63	57
13	27	44	15	12	99	63	57
13	27	***44***	15	12	99	63	57
13	***15***	27	44	12	99	63	57
12	13	15	27	44	99	63	57
12	13	15	27	44	***99***	63	57
12	13	15	27	44	***63***	99	57
12	13	15	27	44	***57***	63	99

Insertion Sort (pseudo-code)

```
algorithm insertion_sort
Input: A an array
Output: A is sorted
for j = 1 to n-1 do
     i ← j
     while i > 0 and A[i-1] > A[i] do
          swap a[i] and a[i-1]
          i ← i - 1
     endwhile
endfor
```

Insertion Sort (pseudo-code)

For each cell of the array

push the elements to the right until item to be inserted

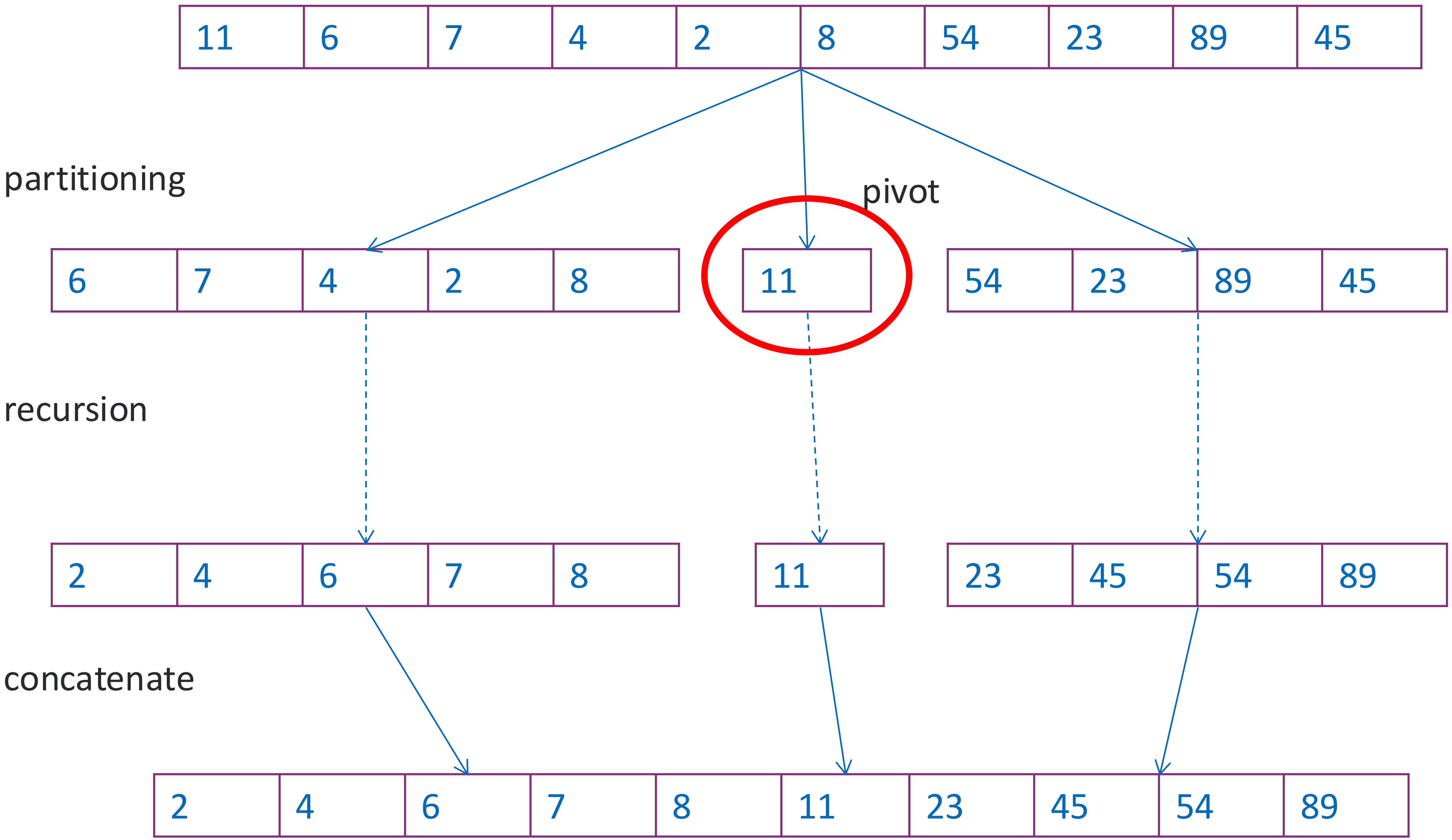
```
algorithm insertion_sort
Input: A an array
Output: A is sorted
for j = 1 to n-1 do
     i ← j
     while i > 0 and A[i-1] > A[i] do
          swap a[i] and a[i-1]
          i ← i - 1
     endwhile
endfor
```

Analysis

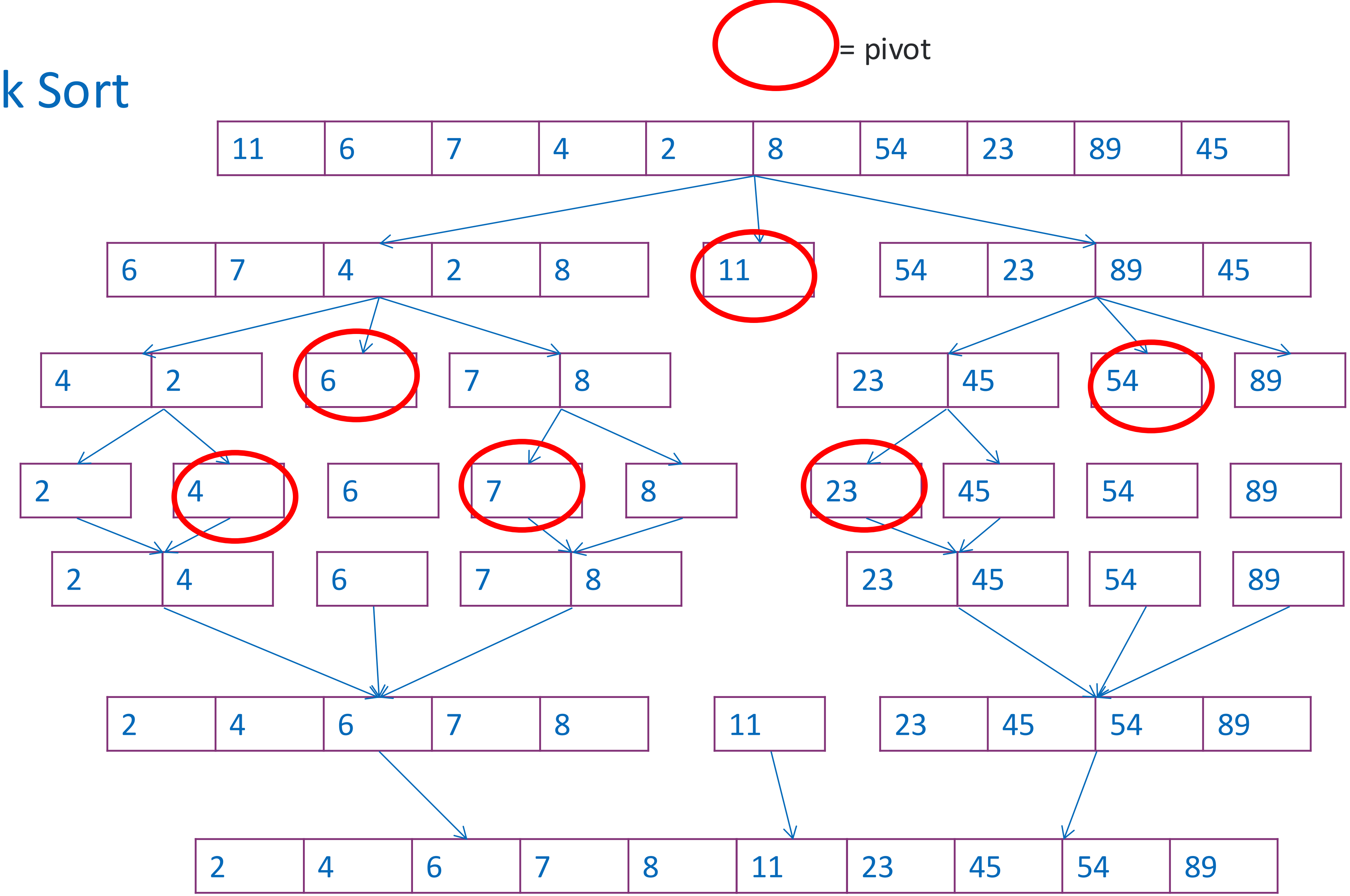
```
algorithm insertion_sort
Input: A an array
Output: A is sorted
for j = 1 to n-1 do # 1 operation per loop
     i ← j # 1 operation per loop
     while i > 0 and A[i-1] > A[i] do
     # 5 operations per loop
          swap a[i] and a[i-1] # 3 operations per loop
          i ← i – 1 # 2 operations per loop per loop
     endwhile
endfor
```

$T(n) = 10n^2 + 2n$ which is $O(n^2)$

Quick Sort


11 6 7 4 2 8 54 23 89 45
partitioning
pivot
6 7 4 2 8
11
54 23 89 45
recursion
2 4 6 7 8
11
23 45 54 89
concatenate
2 4 6 7 8 11 23 45 54 89

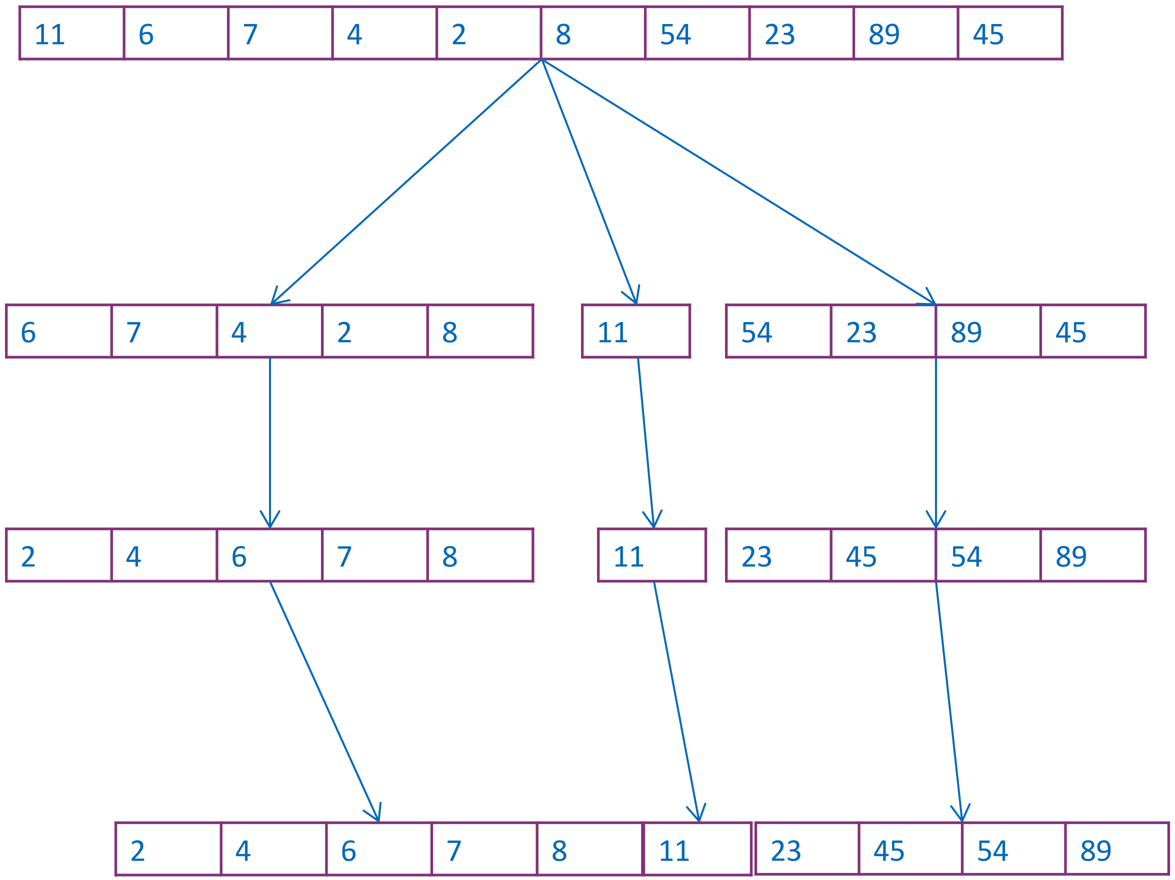
Quick Sort


= pivot
11 6 7 4 2 8 54 23 89 45
6 7 4 2 8
11
54 23 89 45
4 2
6
7 8
23 45
54
89
2
4
6
7
8
23
45
54
89
2 4
6
7 8
23 45
54
89
2 4 6 7 8
11
23 45 54 89
2 4 6 7 8 11 23 45 54 89

Quick Sort

To sort (sub)Array A:

- If A has fewer than two elements, do nothing
- If A has at least two elements,
  - Select a pivot element x from A
  - Remove elements from A and place
    - those less than x in S
    - those equal to x in E, and
    - those greater than x in G
  - Recursively sort S and G
  - Place elements back in A in the order, first the elements of S, then those of E and then those of G.



Quick Sort (pseudocode)

```
algorithm quick_sort:
Input: A an array
Output: A is sorted
if |A| > 1 then
     pivot ←some element from A
     Partition elements of A into lists S (smaller than pivot), E 
(equal) and G (greater than pivot)
     quick_sort(S)
     quick_sort(G)
     Reconstruct A by copying contents of S, E, G (in that order) 
back into A
endif
```

Partitioning elements in S, E and G

```
pivot ← take some element (e.g., first or last) from L and remove
it
E.add(pivot)
while L is not empty do
     elt ← get first element of A and remove it
     if elt < pivot then
          S.add(elt)
     else
          if elt = pivot then
               E.add(elt)
          else
               G.add(elt)
          endif
     endif
endwhile
```

Quick Sort (complete algorithm)

```
algorithm quick_sort:
Input: A an array
Output: A is sorted
if |A| > 1 then
        pivot ← take some element (e.g., first or last) from L and remove it
        E.add(pivot)
        while A is not empty do
            elt ← get first element of A and remove it
            if elt < pivot then
                S.add(elt)
            else
                if elt = pivot then
                        E.add(elt)
                else
                        G.add(elt)
                endif
            endif
        endwhile
        quick_sort (S)
        quick_sort (G)
        Reconstruct A by copying contents of S, E, G (in that order) back into list A
endif
```

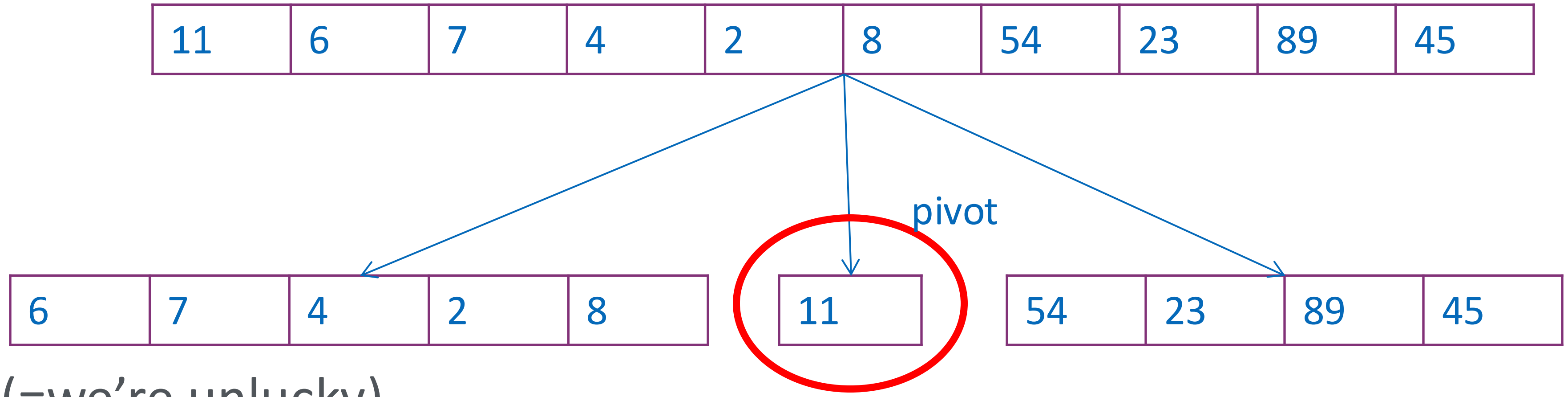
Quick Sort (complete algorithm)

c operations per loop (n elements): cn

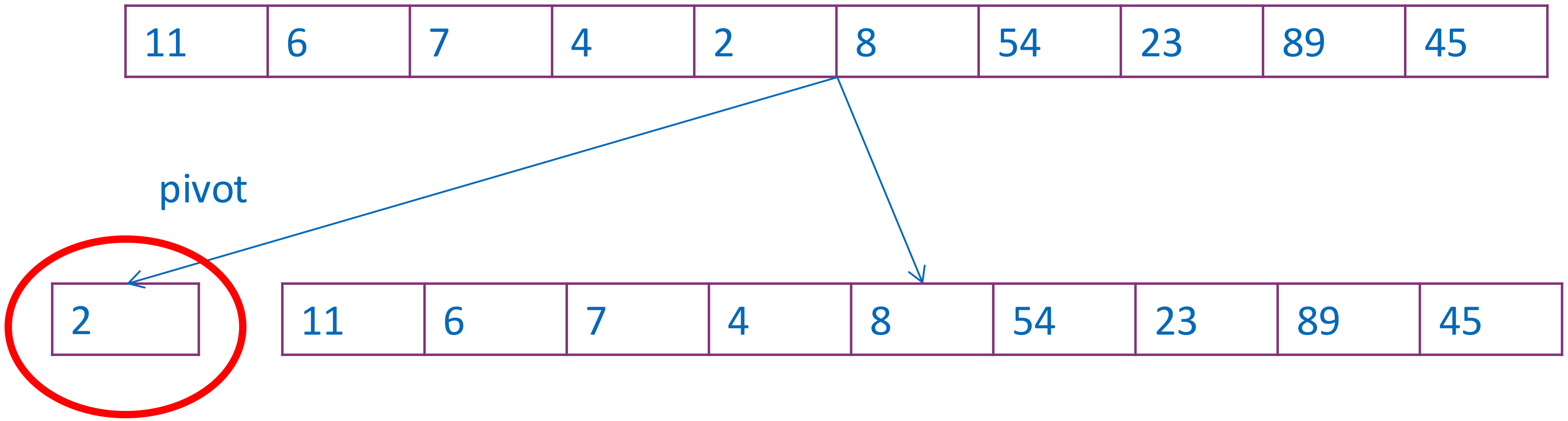
```
algorithm quick_sort:
Input: A an array
Output: A is sorted
if |A| > 1 then
        pivot ← take some element (e.g., first or last) from L and remove it
        E.add(pivot)
        while A is not empty do
            elt ← get first element of A and remove it
            if elt < pivot then
                S.add(elt)
            else
                if elt = pivot then
                        E.add(elt)
                else
                        G.add(elt)
                endif
            endif
        endwhile
        quick_sort (S)
        quick_sort (G)
        Reconstruct A by copying contents of S, E, G (in that order) back into list A
endif
```

How Many Calls?

- “normal”/average

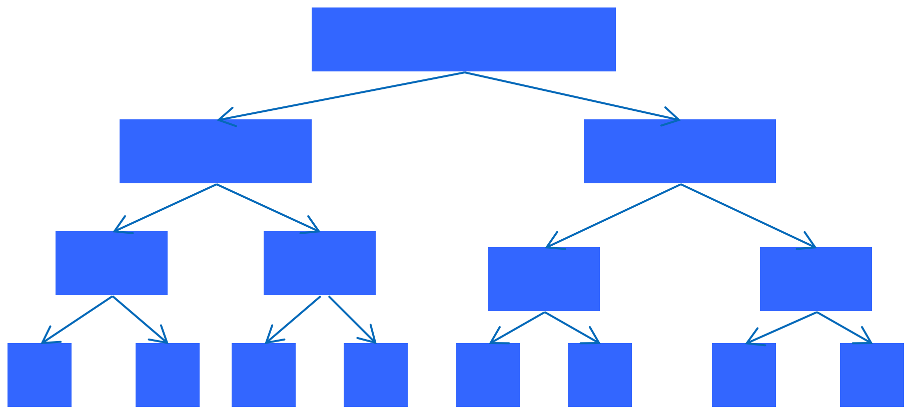


- worst (=we’re unlucky)

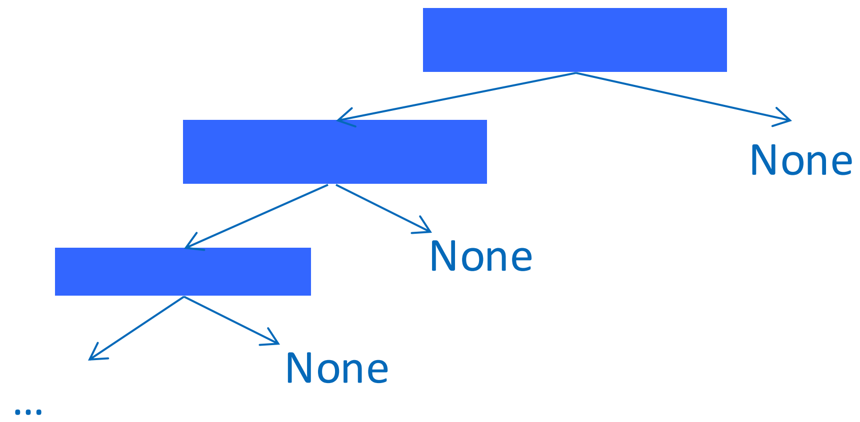


How Many Calls?

- “normal”/average

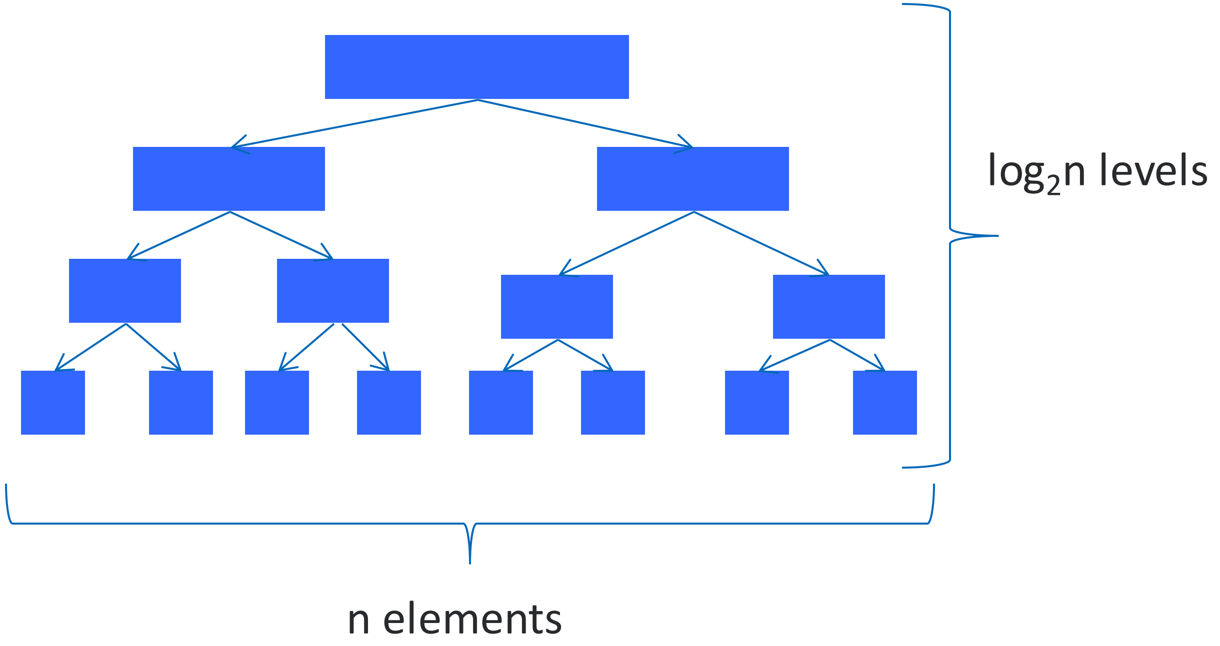


- worst (=we’re unlucky)



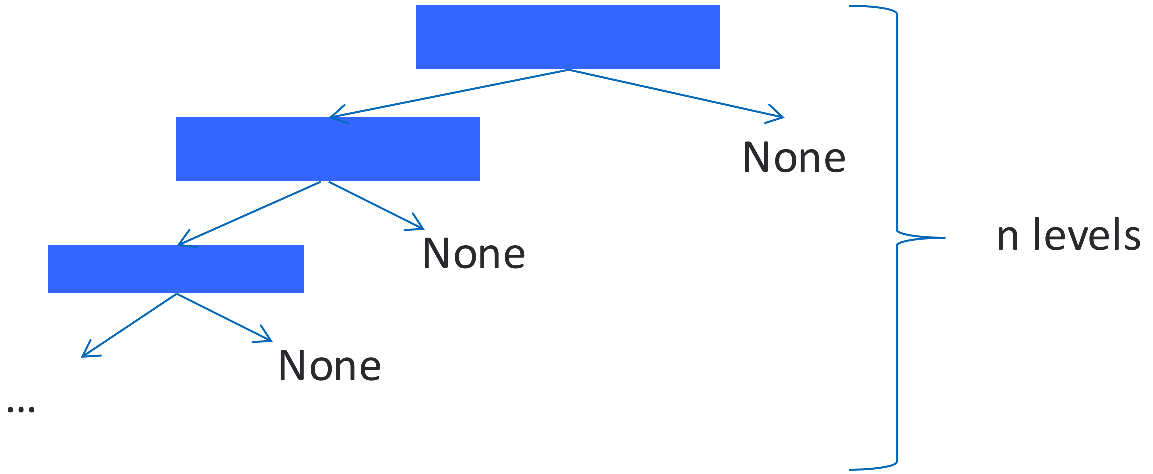
Number of Calls (Average)

at each level we have n elements in total (in each sub-arrays) hence ~cn operations
$\Rightarrow T(n) = cn * \log_2(n)$ which is $O(n \log n)$



Number of Calls (Worst)

at each level we have 1 element less than the previous one
$\Rightarrow T(n)=cn * n$ which is $O(n^2)$



Merge Sort

- There are two ideas behind merge_sort
  - merging two sorted lists is easy
  - sorting by merging, using carefully chosen sequences of merges sounds like a good plan!

The Merge Problem

- Input two sorted arrays A1 and A2:

23	45	54	89

12	25	41	96

- Output Single sorted array containing all values from A1 and A2:

12	23	25	41	45	54	89	96

- Idea Build up A key by key; at each step remove the smallest remaining key from A1 ∪ A2 and append it to the end of A.

Merge Algorithm

compare first elements: 23 > 12?

23	45	54	89	A1
12	25	41	96	A2

output

compare first elements: 23 > 25?

23	45	54	89	A1
25	41	96		A2

output

12							

45	54	89	A1
25	41	96	A2

output

12	23						

Merge Algorithm

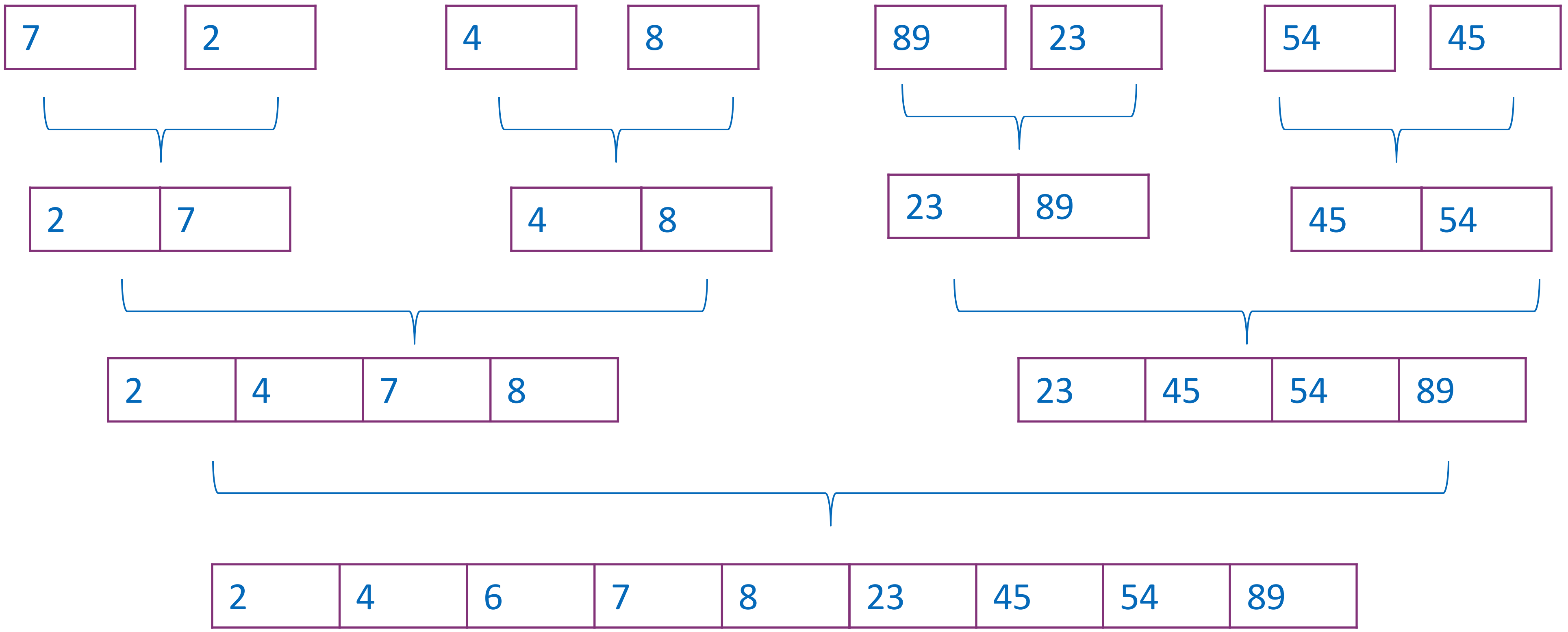
```
algorithm merge(L1, L2, L):
while L1 is not empty and L2 is not empty do
     if L1.get(0) ≤L2.get(0) then
          L.add(L1.remove(0))
     else
          L.add(L2.remove(0))
     endif
endwhile
while L1 is not empty do
     L.add(L1.remove(0))
endwhile
while L2 is not empty do
     L.add(L2.remove(0))
endwhile
```

Complexity Analysis

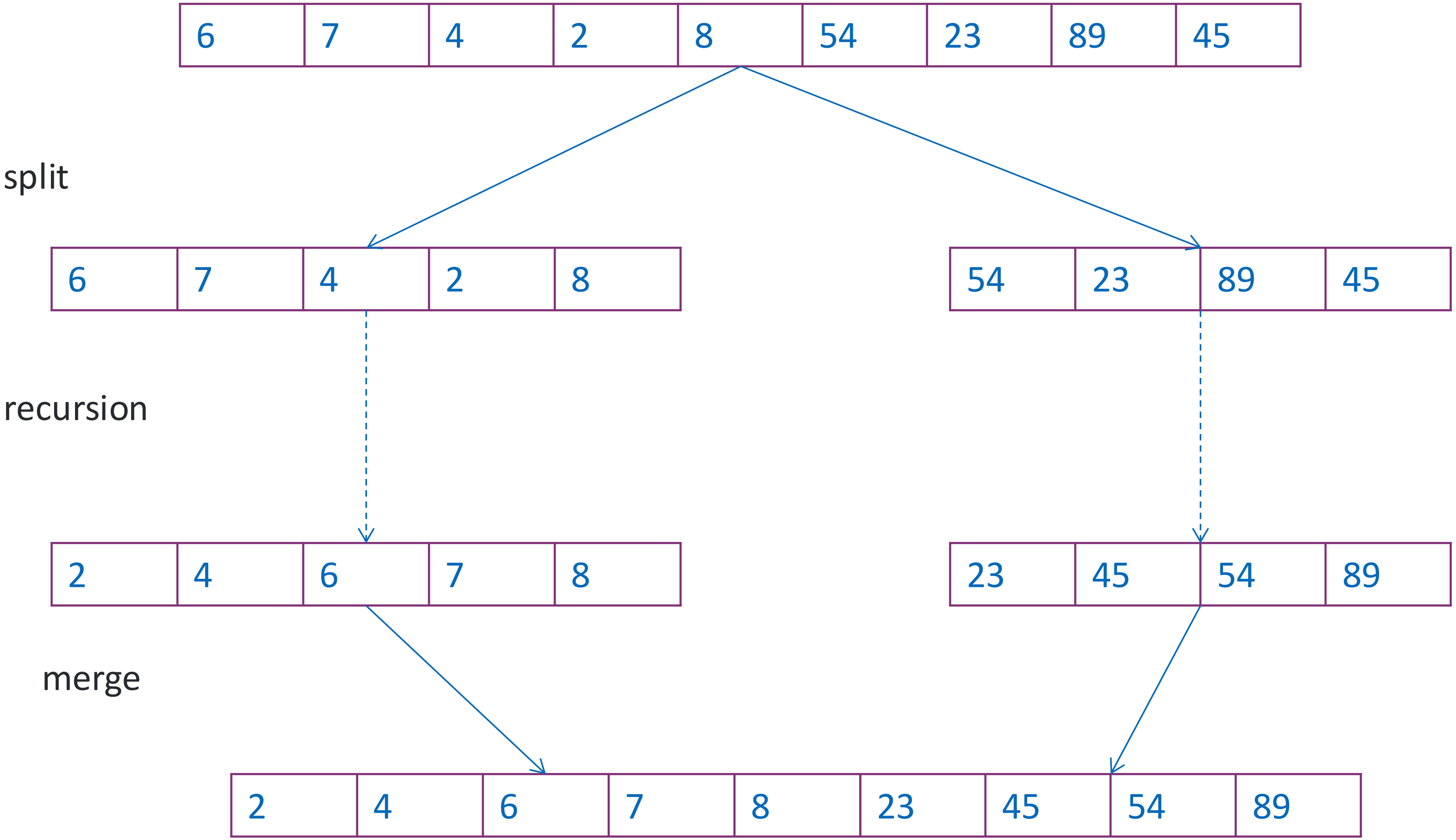
```
algorithm merge(L1, L2, L):
while L1 is not empty and L2 is not empty do # 3 op per loop
       if L1.get(0) ≤L2.get(0) then # 3 op per loop
              L.add(L1.remove(0)) # 2 op per loop
       else
              L.add(L2.remove(0)) # 2 op per loop
       endif
endwhile
while L1 is not empty do # 1 op per loop
       L.add(L1.remove(0)) # 2 op per loop
endwhile
while L2 is not empty do # 1 op per loop
       L.add(L2.remove(0)) # 2 op per loop
endwhile

worst case: 8(n + m), n being L1’s size and m being L2’s size, which is O(n+m) or O(n)
```

Sort by Merging

Idea: can sort using carefully chosen pattern of merges



Merge Sort

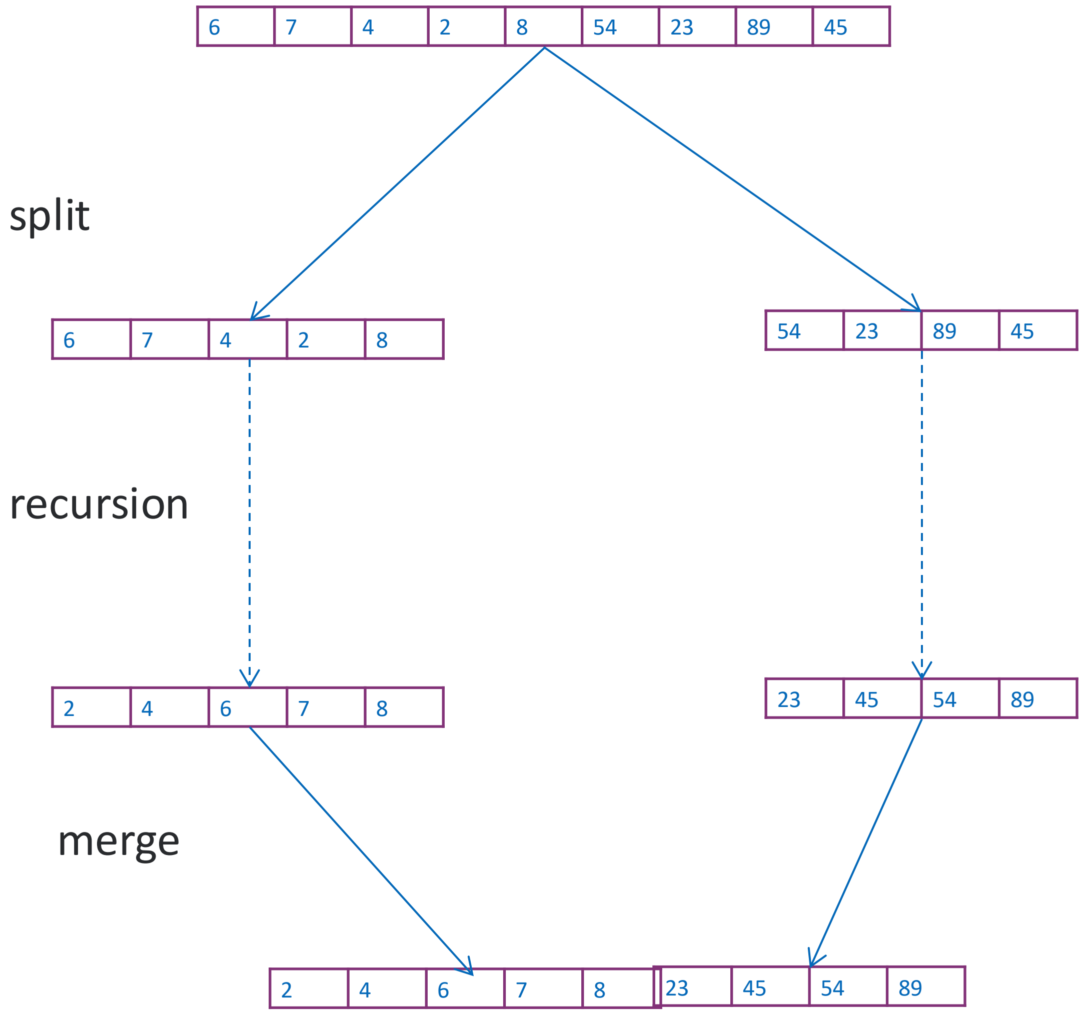

6 7 4 2 8 54 23 89 45
split
6 7 4 2 8
54 23 89 45
recursion
2 4 6 7 8
23 45 54 89
merge
2 4 6 7 8 23 45 54 89

Merge Sort

To sort (sub)array A:

- If A has fewer than two elements, do nothing

- If A has at least two elements,
  - Split A into two arrays A1 and A2 of equal size (+/- 1)
  - Recursively sort A1 and A2
  - Transfer elements back into A by merging (sorted) A1 and (sorted) A2



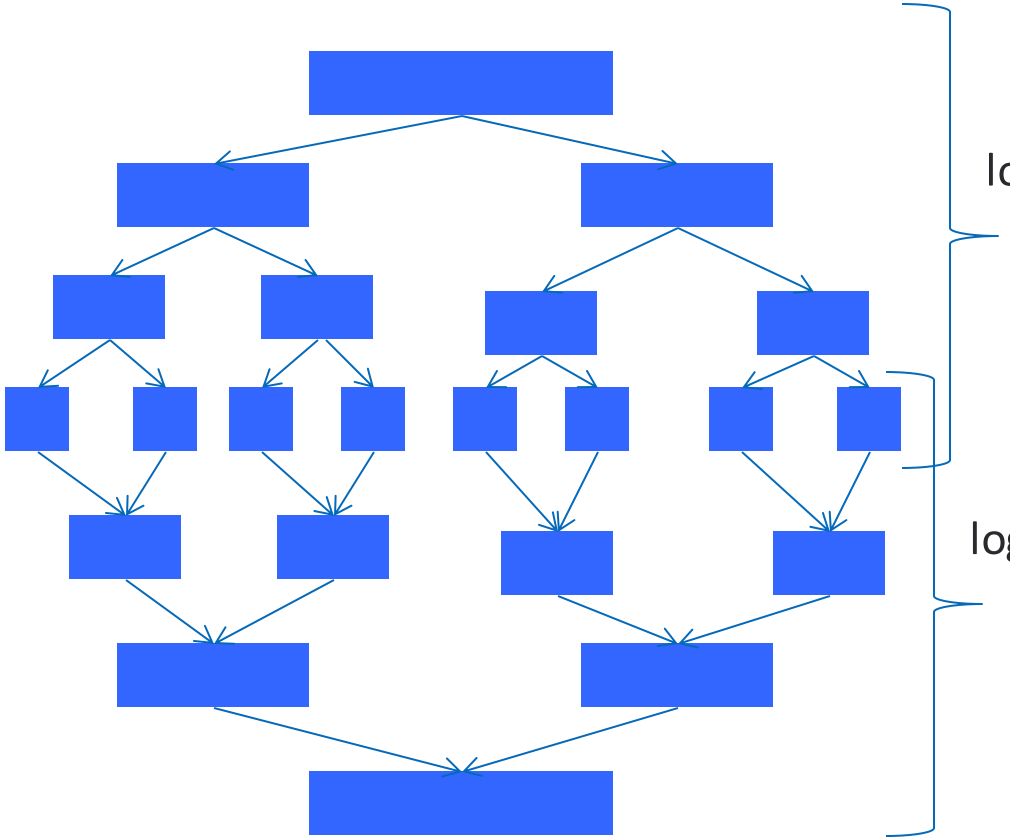
Merge Sort (pseudocode)

```
algorithm merge_sort:
Input: A an array
Output: A is sorted
if |A| > 1 then
     for j ← 0 to |A|/2 do
          add A[j] to A1
     endfor
     for  j ← |A|/2 + 1 to |A| do
          add A[j] to A2
     endfor
     merge_sort(A1)
     merge_sort(A2)
     A = merge(A1, A2)
endif
return A
```

Analysis

```
algorithm merge_sort:
Input: A an array
Output: A is sorted
if |A| > 1 then # 1 operation
       for j ← 0 to |A|/2 - 1 do # 1 operation per loop (n/2)
              add A[j] to A1 # 1 operation per loop
       endfor
       for  j ← |A|/2  to |A|-1 do # 1 operation per loop (n/2)
              add A[j] to A2 # 1 operation per loop
       endfor
       merge_sort(A1) # 1 operation
       merge_sort(A2) # 1 operation
       A = merge(A1, A2)  # T(n) = 8*(n/2 + n/2) = 8n
endif
return A # 1 operation
T(n) = 1 + 2*n/2 + 2*n/2 + 2 + 8n + 1 = 10n + 3 which is O(n)
```

Number of Calls



- at each split level we have ~2n operations $T_{split}(n) = 2n*\log_2(n)$ which is O(n log n)
- at each merge level we have ~8n operations $T_{split}(n) = 8n*\log_2(n)$ which is O(n log n)

=> O(nlogn)