INFDEV026A - Algoritmiek Unit 2

G. Costantini, F. Di Giacomo, G. Maggiore

costg@hr.nl, giacf@hr.nl, maggg@hr.nl – Office H4.204

Today

- ~~Why is my code slow?~~
  - ~~Empirical and complexity analysis~~
- How do I order my data?
  - **Sorting algorithms**
- How do I structure my data?
  - **Linear, tabular, recursive data structures**
- How do I represent relationship networks?
  - **Graphs**

Sorting algorithms

Insertion sort, Merge sort

http://www.youtube.com/watch?v=INHF_5RIxTE

Sorting algorithms

- Algorithms that put elements of a sequence in a certain order (numerical/lexicographical)
  - fundamental problem in computer science
    - as a standalone algorithm (i.e., producing human readable output)
    - as part of more complex algorithms which require sorted data (i.e., binary search!)
  - usually, data is considered to be stored in an array
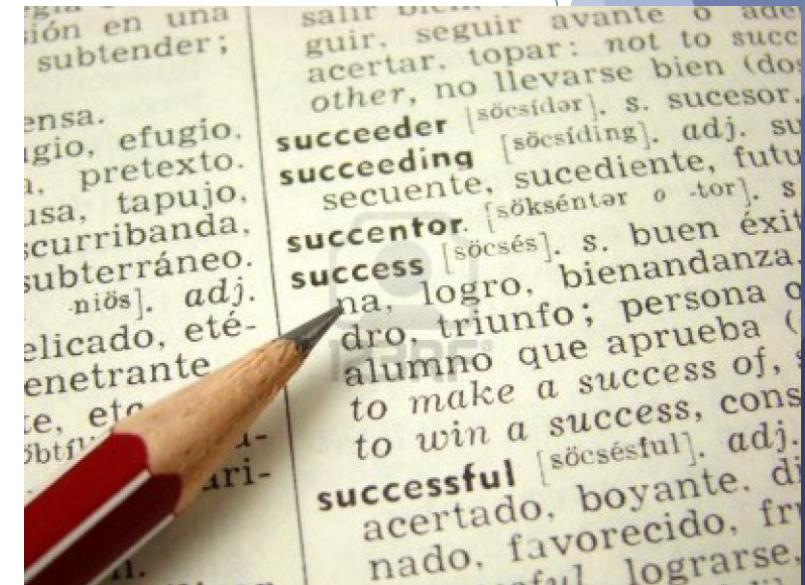
Sorting algorithms

- Popular [sorting algorithms](#)
  - Simple sorts
    - [Insertion sort](#), selection sort
  - Efficient sorts
    - [Merge sort](#), Quick sort, Heap sort
  - Bubble sort and variants
    - Bubble sort, Shell sort, Comb sort
  - Distribution sort
    - Counting sort, Bucket sort, Radix sort
- In practice, a few algorithms predominate

Sorting algorithms

▶ When studying algorithms, we are interested in many aspects

  ▶ **Correctness**, first and foremost

  ▶ **Performance**, especially in certain domains (i.e., games)

    ▶ **Time** & Space

▶ Time complexity… remember from last lesson?

  ▶ Asymptotic notation to express the relationship between computing time and input size

  ▶ Example: suppose an algorithm runs in $0.1n^2 + 10.0n$ milliseconds for an input of size n → its complexity is $O(n^2)$

  ▶ Valid asymptotically (i.e., not for small inputs)

Sorting algorithms

- **Input**
 - a sequence of n numbers $a_1, \dots, a_n$

- **Output**
  - a permutation (reordering) $a_{1'}, \dots, a_{n'}$ of the input sequence ...
 - ... in non-decreasing order (i.e., such that $a_{1'} \leq \cdots \leq a_{n'}$)

- The numbers that we wish to sort are also known as the **keys**

Sorting algorithms
Properties

▶ Interesting properties for a sorting algorithm

    ▶ STABILITY → maintain the relative order of elements with equal keys

    ▶ COMPUTATIONAL COMPLEXITY → how many elements comparisons in terms of the size of the sequence

        ▶ Good behavior is $O(n \log n)$

    ▶ MEMORY USAGE → *in-place* algorithms need only $O(1)$ memory beyond the items being sorted

    ▶ RECURSION → recursive or not

    ▶ ADAPTABILITY → the presortedness of the input affects the running time
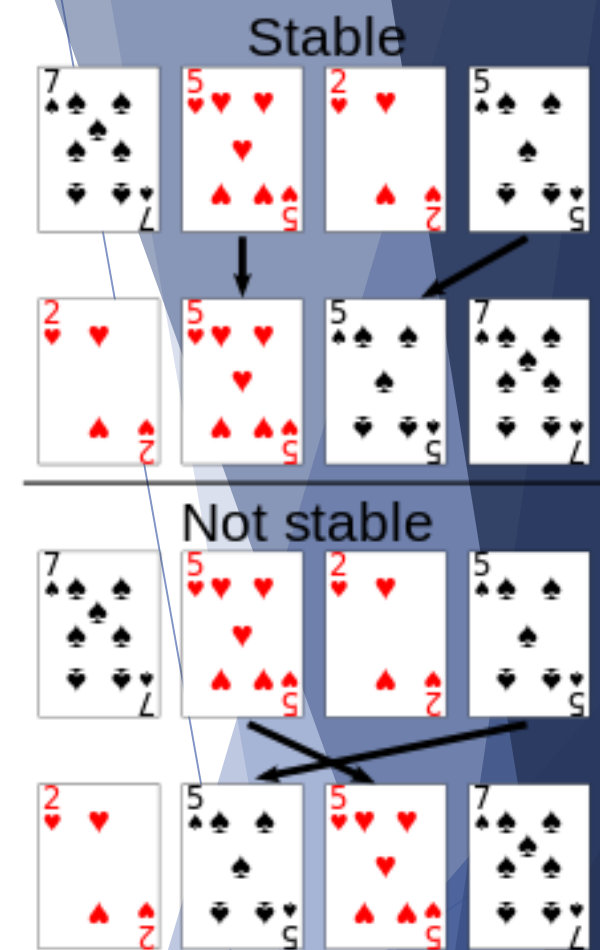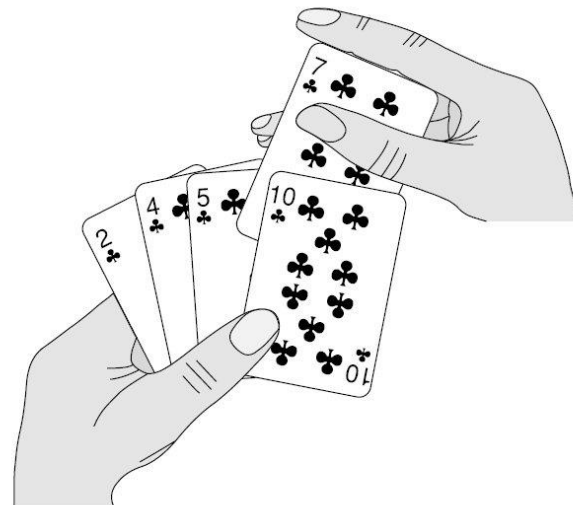
Insertion sort

Insertion sort

- Basic idea
  - When people manually sort something (for example, a deck of playing cards), most use a method that is similar to insertion sort
  - Put one element at a time in its right position in the sorted sub-array
  - The final sorted array (or list) is built one item at a time

Insertion sort

- Basic idea
    - When people manually sort something (for example, a deck of playing cards), most use a method that is similar to insertion sort
    - Put one element at a time in its right position in the sorted sub-array
    - The final sorted array (or list) is built one item at a time

- Advantages
    - very intuitive algorithm ; simple implementation
    - efficient for (quite) small data sets
        - very efficient for data sets that are already substantially sorted and more efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms; the best case (nearly sorted input) is $O(n)$
    - stable
    - in-place
    - online (can sort a sequence *as* it receives it, one element at a time)
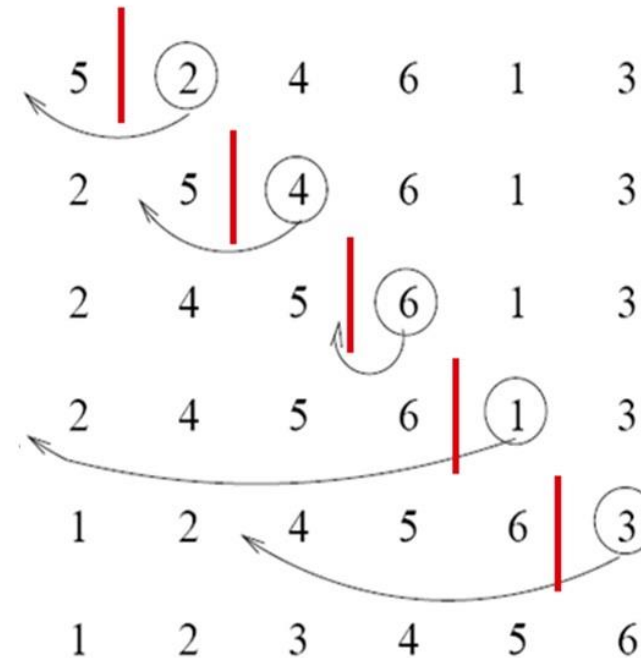
Insertion sort

▶ Graphical example

6 5 3 1 8 7 2 4

Insertion sort

- Iterative algorithm
  - At each iteration one input element is consumed, growing a sorted output sequence

- Iteration
  i. remove one element from the input data
  ii. find the location it belongs within the sorted sequence
  iii. insert it there
- Repeat until no input elements remain

5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

Insertion sort

▶ Sorting is typically done in-place

▶ For each unsorted item

   ▶ shift all the larger values up to make a space
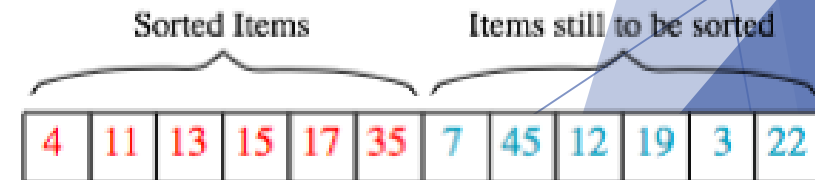
   ▶ then insert it into the correct position

Start with a partially sorted list of items:

Sorted Items   Items still to be sorted

| 4 | 11 | 13 | 17 | 35 | 15 | 7 | 45 | 12 | 19 | 3 | 22 |
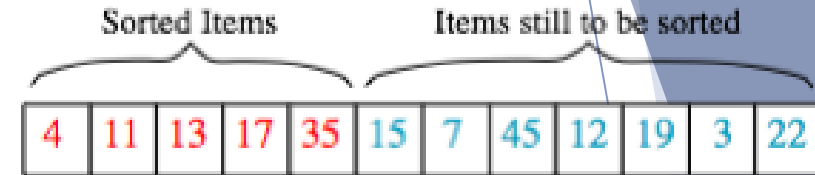
Temp: 15   Copy next unsorted item into Temp, leaving a "hole" in the array.

| 4 | 11 | 13 | 17 | 35 | | 7 | 45 | 12 | 19 | 3 | 22 |

Move items in sorted part of array to make room for Temp.   Temp: 15

| 4 | 11 | 13 | 15 | 17 | 35 | 7 | 45 | 12 | 19 | 3 | 22 |

Sorted Items   Items still to be sorted

| 4 | 11 | 13 | 15 | 17 | 35 | 7 | 45 | 12 | 19 | 3 | 22 |

Now, the sorted part of the list has increased in size by one item.

Insertion sort
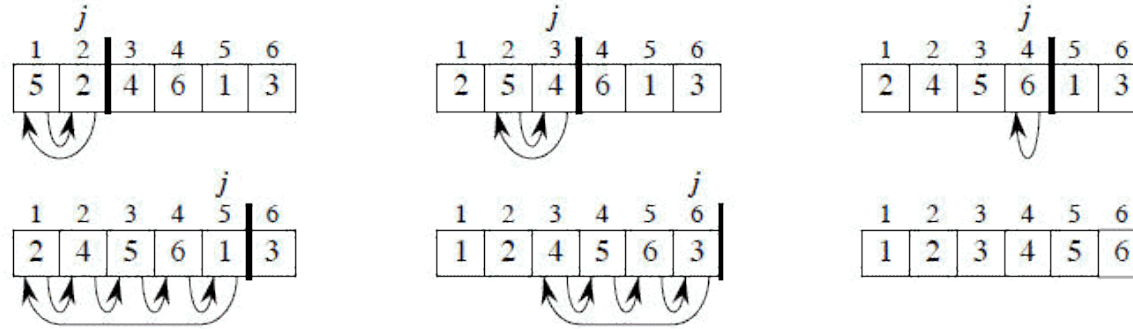
▶ Pseudo-code of the algorithm (<u>supposing the origin of the array is 1</u>)
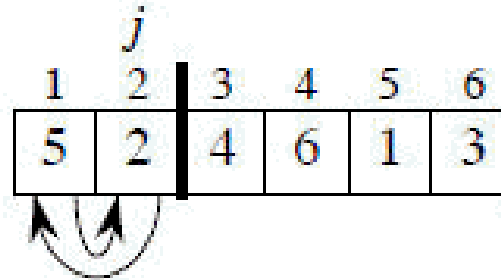
```
FOR j = 2 to length(A)
  key = A[j]
  % put A[j] into the sorted sequence A[1..j-1]
  i = j - 1
  WHILE i > 0 and A[i] > key
    A[i+1] = A[i]
    i = i – 1
  A[i+1] = key
```
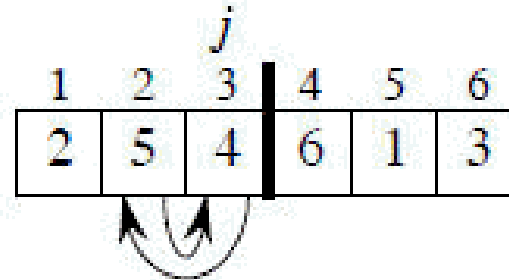
Insertion sort

```
FOR j = 2 to length(A)
    key = A[j]
    % put A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    WHILE i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

- First iteration trace
  - $j = 2$
  - key = A[2] = 2
  - $i = 1$
    - $i > 0$ && A[i] > 2 ? YES
      - A[2] = A[1] → A[2] = 5
      - $i = i - 1 = 0$
    - $i > 0$ && A[i] > 2 ? NO because $i = 0$
  - A[i+1] = 2 → A[1] = 2
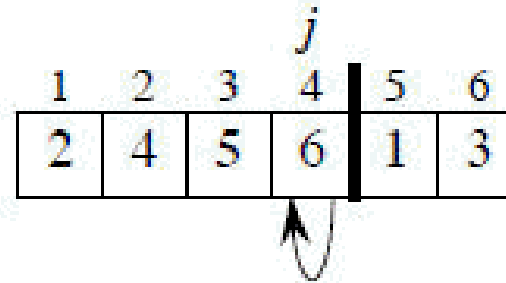
Insertion sort

```
FOR j = 2 to length(A)
    key = A[j]
    % put A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    WHILE i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

- Second iteration trace

  - j = 3

  - key = A[3] = 4

  - i = 2

    - i > 0 && A[i] > 4 ? YES

      - A[3] = A[2] → A[3] = 5

      - i = i – 1 = 1

    - i > 0 && A[i] > 4 ? NO because 2 > 4 is false

  - A[i+1] = 4 → A[2] = 4

Insertion sort

```
FOR j = 2 to length(A)
    key = A[j]
    % put A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    WHILE i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

▶ Third iteration trace

  ▶ j = 4

  ▶ key = A[4] = 6

  ▶ i = 3

    ▶ i > 0 && A[i] > 6 ? NO because 5 > 6 is false

  ▶ A[i+1] = 6 → A[4] = 6

▶ ... and so on ...

Insertion sort

- In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result

Sorted partial result		Unsorted data	
$\leq x$	$> x$	x	...

$\Longrightarrow$

Sorted partial result			Unsorted data
$\leq x$	x	$> x$	...

Insertion sort

▶ **Correctness**

  ▶ After k iterations, the following property holds:

  $$\textit{The first } k + 1 \textit{ entries are sorted}$$

  ("+1" because the first entry is skipped)

  ▶ this property (called **INVARIANT**) holds true for every k, i.e. for the whole run of the algorithm

    ▶ can be proved formally by induction (for us, intuition only)

▶ How many iterations does the algorithm?

  ▶ $length - 1$

  ▶ After the last iteration, then: "*The first* $(length - 1) + 1$ *entries are sorted*" → "*The first* $length$ *entries are sorted*" → All entries are sorted!!!

Insertion sort

▶ **Performance**

▶ Even with the same input *size*, runtime may differ

    ▶ Depends on the _shape_ of the data!

        ▶ what varies is how many times we execute the loop test (t_j)

        ▶ we can distinguish best, **worst**, average case

        ▶ for each one, we can use the Big O notation (upper bound)

Insertion sort

```
FOR j = 2 to length(A)
    key = A[j]
    % put A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    WHILE i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i – 1
    A[i+1] = key
```

▶ Best case (when the array is already sorted)

2	4	5	7

  ▶ The condition of the **while** loop then is false (the body is not executed)

  ▶ We just verify that every element is in the correct position (through the "for" loop)

  ▶ Complexity $O(n)$

Insertion sort

```
FOR j = 2 to length(A)
    key = A[j]
    % put A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    WHILE i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

▶ Worst case (when the array is in reverse sorted order)

7	5	4	2

- ▶ The **while** loop is executed the maximum possible # of times
- ▶ For each element we have to insert it into the right position by shifting all the elements to its left
- ▶ Complexity $O(n^2)$

Insertion sort

- **Complexity** analysis: summary

  - *Best case* → sequence already sorted

3	5	6	9	11	15

    $$O(n)$$

  - *Worst case* → sequence in reverse sorted order

15	11	9	6	5	3

    $$O(n^2)$$

9 3 4 220 1 3 10 5 8 7 2

Divide the data set in half.

9 3 4 220 1 | 3 10 5 8 7 2

9 3 4 220 1   3 10 5 8 7 2

Sort each half.

1 3 4 9 220   2 3 5 7 8 10

Merge the halves to
obtain the sorted set.

1 2 3 3 4 5 7 8 9 10 220
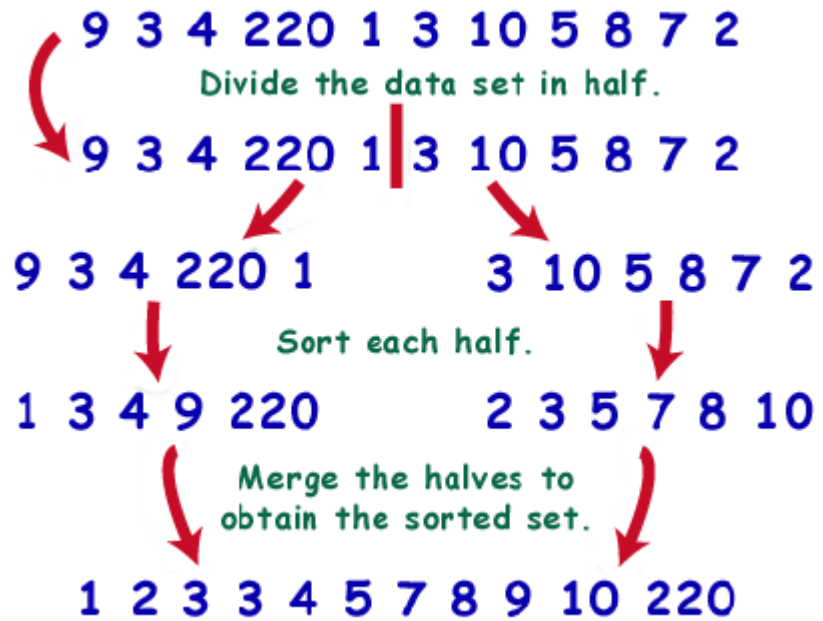
Merge sort

Merge sort

▶ "Divide-and-conquer" approach

1. [DIVIDE] Break problem into smaller sub-problems

2. [CONQUER] Solve the sub-problems recursively

   ▶ If the sub-problems are small enough just solve them straightforwardly

3. [COMBINE] Combine the solutions of the sub-problems

Merge sort

- "Divide-and-conquer" approach
  1. [DIVIDE] Break problem into smaller sub-problems
  2. [CONQUER] Solve the sub-problems recursively
     - If the sub-problems are small enough just solve them straightforwardly
  3. [COMBINE] Combine the solutions of the sub-problems

- Merge sort idea
  1. [DIVIDE] Split the sequence to sort (n elements) into two sub-sequences ($\frac{n}{2}$ elements each)
  2. [CONQUER] Sort the two sub-sequences recursively (using merge sort)
     - If the sub-sequence has length 1, it is already sorted (recursion stops here)
  3. [COMBINE] Merge the two sorted sub-sequences to produce the complete sorted sequence

Merge sort

▶ Merge sort idea

1. [DIVIDE] Split the sequence to sort into two sub-sequences ($\frac{n}{2}$ elements each)

2. [CONQUER] Sort the two sub-sequences recursively (using merge sort)

3. [COMBINE] Merge the two sorted sub-sequences to produce the complete sorted sequence

▶ Pseudocode

```
MERGE-SORT(A,p,r)
    if p < r
        q = (p + r) / 2
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```
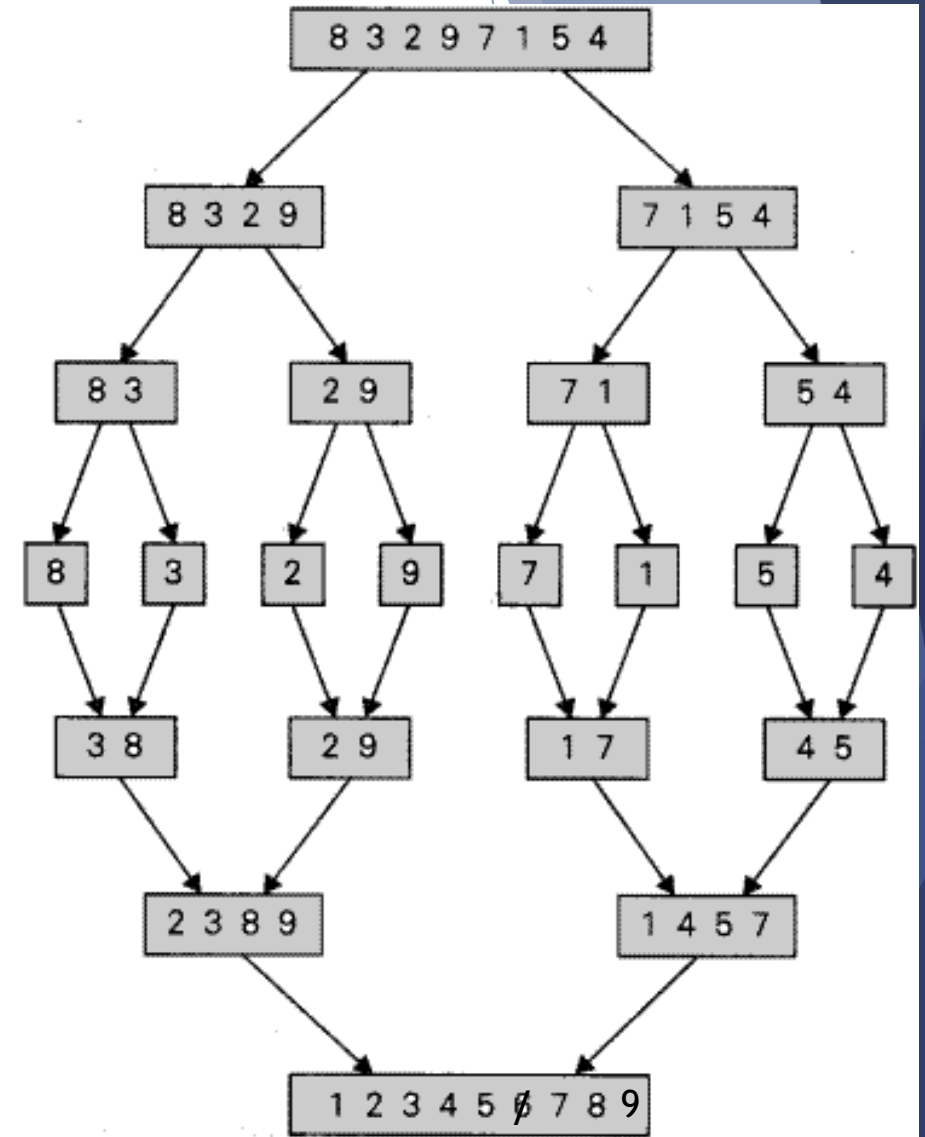
Merge sort

▶ Merge sort idea

1. [DIVIDE] Split the sequence to sort into two sub-sequences ($\frac{n}{2}$ elements each)

2. [CONQUER] Sort the two sub-sequences recursively (using merge sort)

3. [COMBINE] Merge the two sorted sub-sequences to produce the complete sorted sequence

▶ Pseudocode

```
MERGE-SORT(A,p,r)
    if p < r
        q = (p + r) / 2
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```

Merge sort

▶ Merge sort idea

1. [DIVIDE] Split the sequence to sort into two sub-sequences ($\frac{n}{2}$ elements each)

2. [CONQUER] Sort the two sub-sequences recursively (using merge sort)

3. [COMBINE] Merge the two sorted sub-sequences to produce the complete sorted sequence

▶ Pseudocode

```
MERGE-SORT(A,p,r)
    if p < r
        q = (p + r) / 2
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```

Merge sort

- ▶ How to combine two sorted sub-sequences into one?

- ▶ Example
  - ▶ two sorted piles of cards (face-up; smallest card on top)
  - ▶ we want to merge them into a single sorted pile

Merge sort

- How to combine two sorted sub-sequences into one?

- Example

  - two sorted piles of cards (face-up; smallest card on top)

  - we want to merge them into a single sorted pile

- Procedure

  - Choose the smallest of the two cards on top and remove it from its pile

    - Place this card into the output pile (face down)

  - Repeat the previous step until one of the two piles is empty

  - Take the remaining input pile and move it into the output one

Merge sort

▶ Example: merging the two sequences <u>2</u> 3 8 9 and <u>1</u> 4 5 7

    ▶ Compare 2 and 1 → Put 1 into the output sequence

        ▶ S1: <u>2</u> 3 8 9; S2: <u>4</u> 5 7; OUTPUT: **1**

    ▶ Compare 2 and 4 → Put 2 into the output sequence

        ▶ S1: <u>3</u> 8 9; S2: <u>4</u> 5 7; OUTPUT: **1 2**

    ▶ Compare 3 and 4 → Put 3 into the output sequence

        ▶ S1: <u>8</u> 9; S2: <u>4</u> 5 7; OUTPUT: **1 2 3**

    ▶ Compare 8 and 4 → Put 4 into the output sequence

        ▶ S1: 8 9; S2: 5 7; OUTPUT: **1 2 3 4**

    ▶ …

Merge sort

Example: merging the two sequences <u>2</u> 3 8 9 and <u>1</u> 4 5 7

- ...
  - S1: <u>8</u> 9; S2: <u>5</u> 7; OUTPUT: **1 2 3 4**
- Compare 8 and 5 → Put 5 into the output sequence
  - S1: <u>8</u> 9; S2: <u>7</u>; OUTPUT: **1 2 3 4 5**
- Compare 8 and 7 → Put 7 into the output sequence
  - S1: <u>8</u> 9; S2: ∅; OUTPUT: **1 2 3 4 5 7**
- One pile is empty
  - Put all the elements of the other pile (8 9) in the output sequence
  - S1: ∅; S2: ∅; OUTPUT: **1 2 3 4 5 7 8 9**
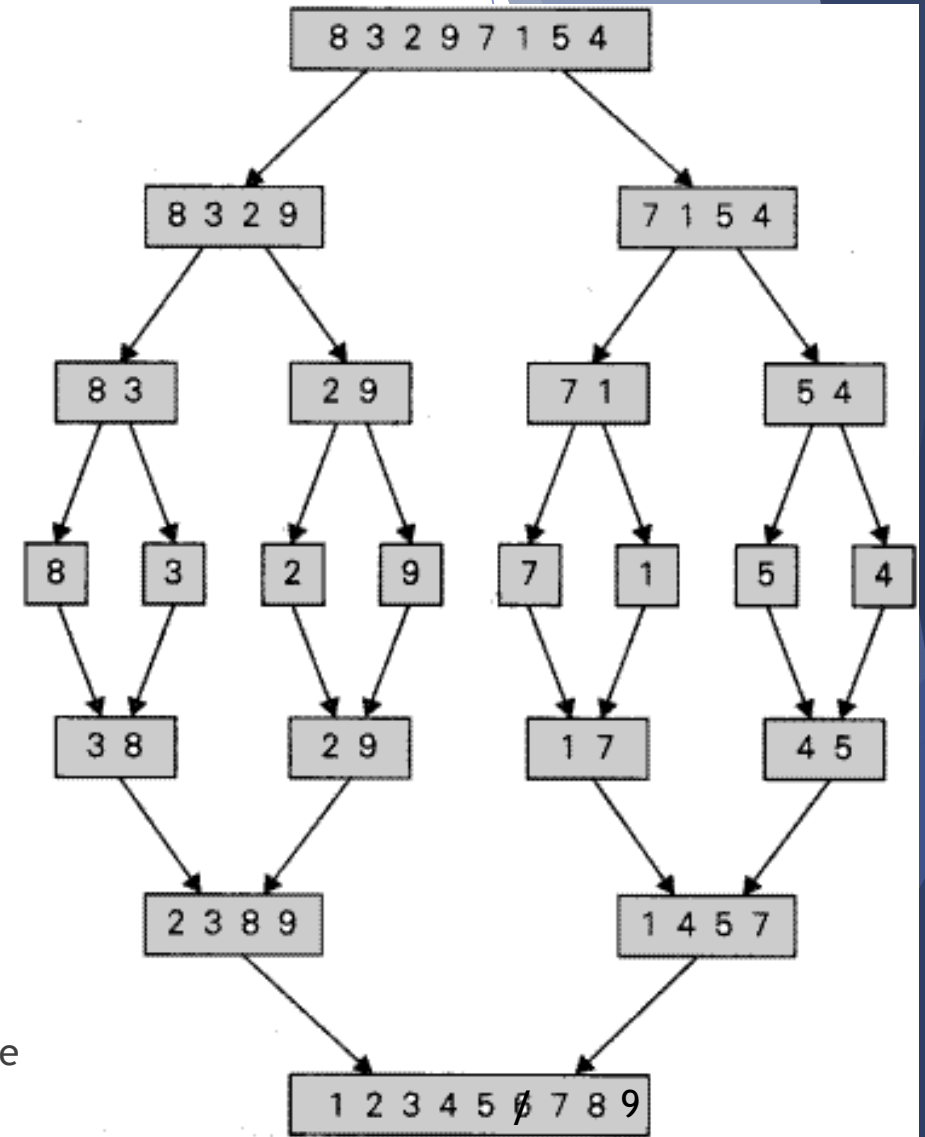
Merge sort

- A possible way to put it into code
  - Start by creating two arrays L,R
    - L contains the left part of A (one pile)
    - R contains the right part of A (other pile)
    - Both parts are sorted!
    - Last element of both L/R is ∞
  - Then, choose & copy the smallest element from the two arrays
    - No need to check if one part is empty, thanks to the use of ∞

```
MERGE(A, p, q, r)
n₁ = q – p + 1
n₂ = r – q
let L[1..n₁ + 1 ] and L[1..n₂ + 1] be new arrays
FOR i = 1 TO n₁
  L[i] = A[p + i - 1]
FOR j = 1 TO n₂
  R[j] = A[q + j]
L[n₁ + 1] = ∞
R[n₂ + 1] = ∞
i = 1
j = 1
FOR k = p TO r
  IF L[i] ≤ R[j]
    A[k] = L[i]
    i = i + 1
  ELSE
    A[k] = R[j]
    j = j + 1
```
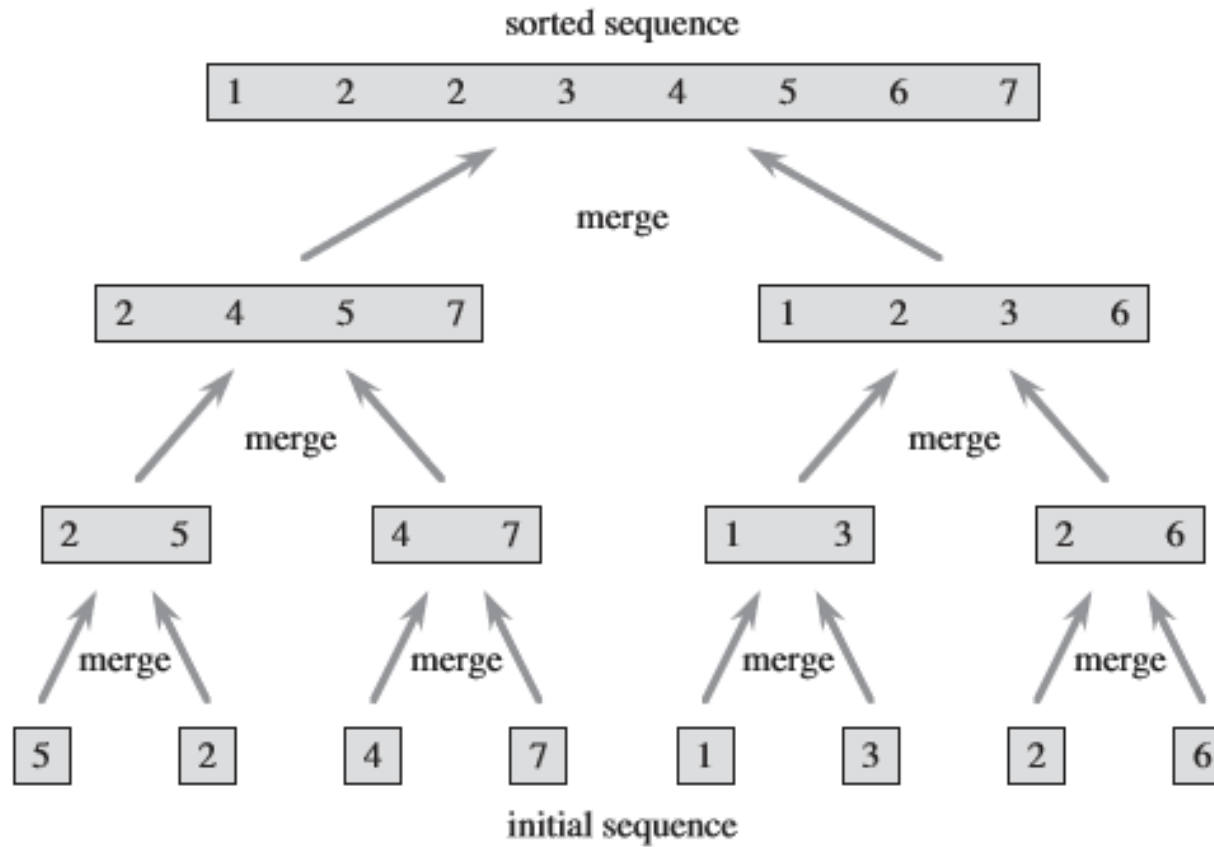
Merge sort
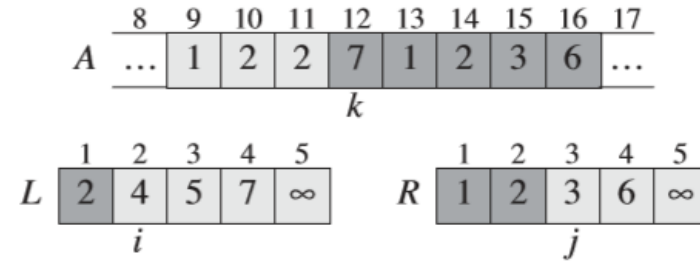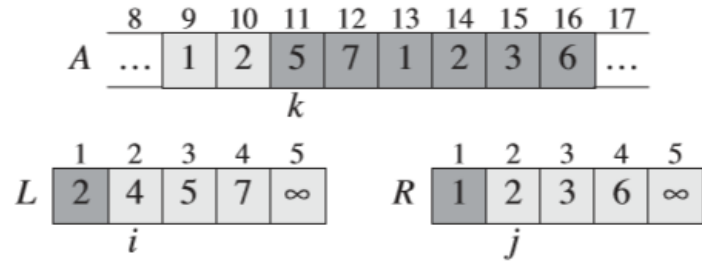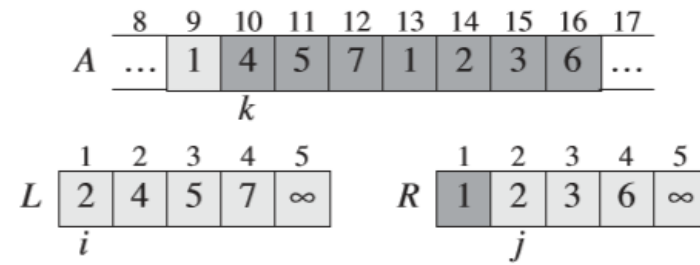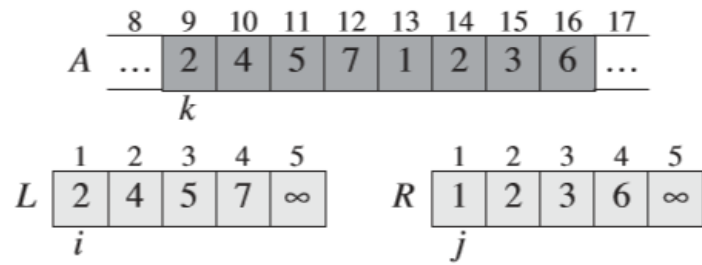
Merge sort
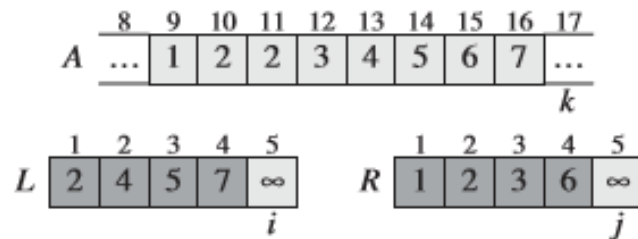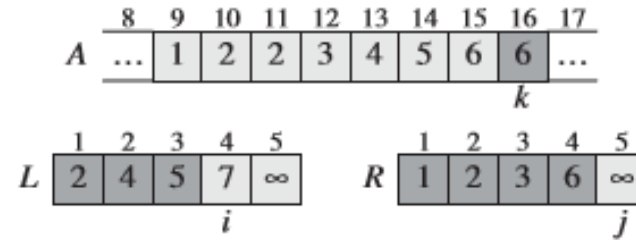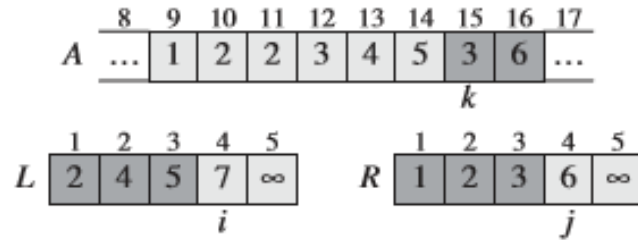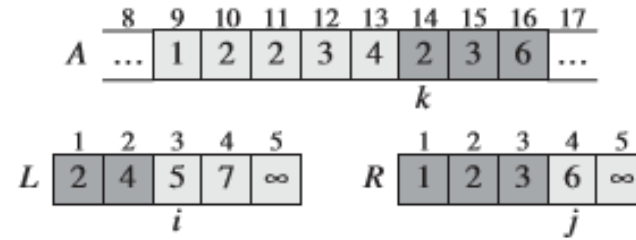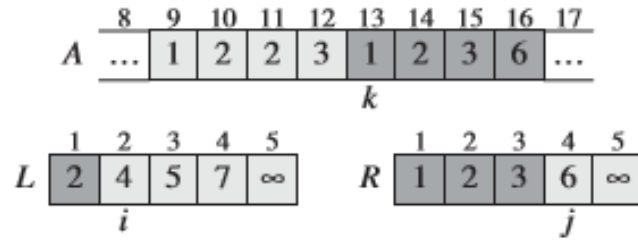
Merge sort

Merge sort

- **Performance**
  - $T(n)$ → running time of Merge Sort on an input of size n
  - The total running time is the sum of…
    - [DIVIDE] compute the middle of the subarray → $O(1)$
    - [CONQUER] recursively solve the two sub-problems, each of size $\frac{n}{2}$ → $2 \times T\left(\frac{n}{2}\right)$
    - [COMBINE] n iterations of the loop, each of which takes constant time → $O(n)$

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + n$$

  - To solve this recurrence, we would need the "master theorem"…
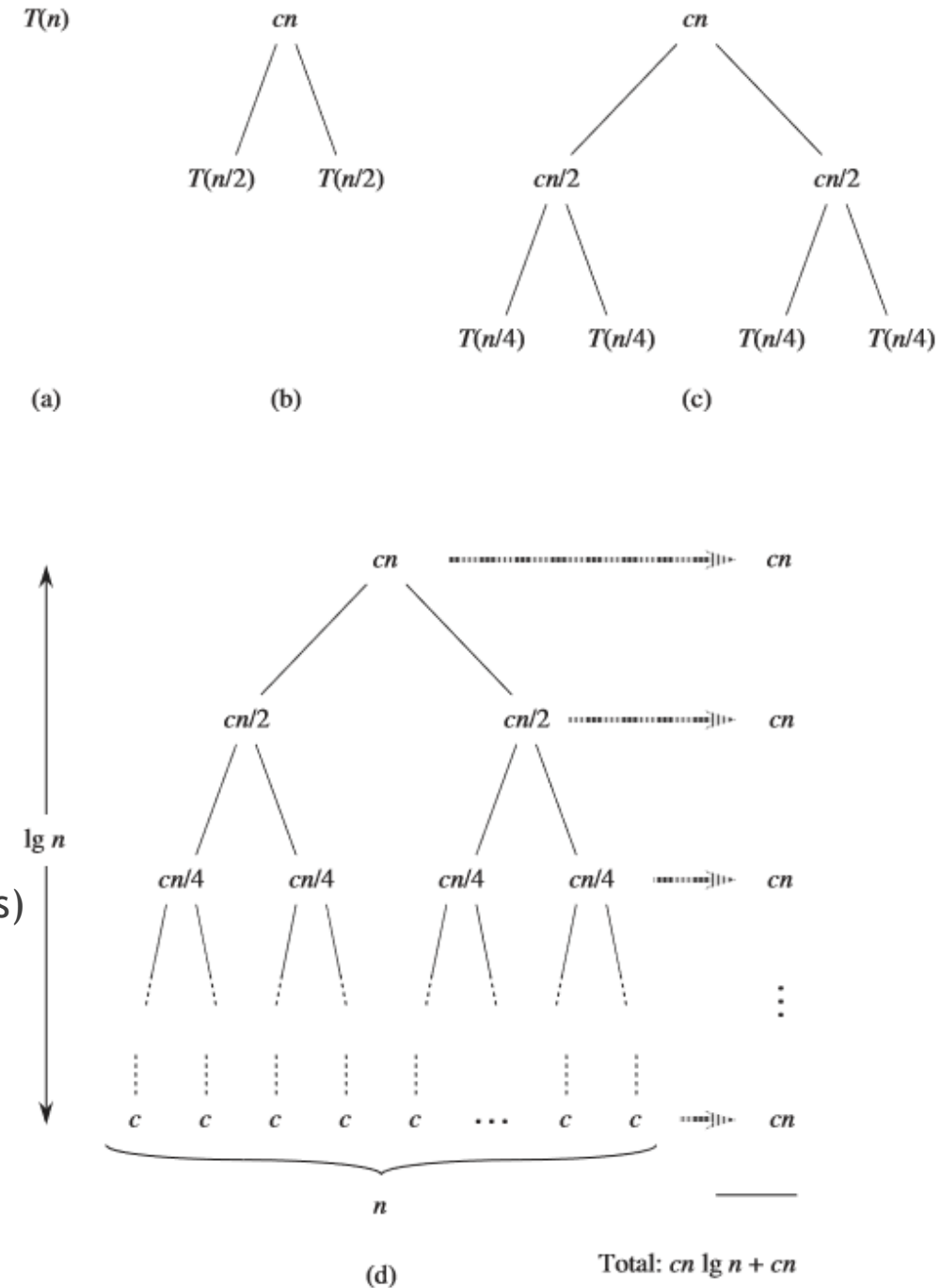 - … here only intuition! (pfiuuuuu ☺)

Merge sort

▶ Performance

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + n$$

▶ for convenience, assume that n is a power of 2

▶ Tree representing the recurrence

▶ Root = top level of recursion

▶ Each node = cost of merging plus cost of sub-problems (subtrees)

▶ Leaves = problems of size 1 (recursion stops)
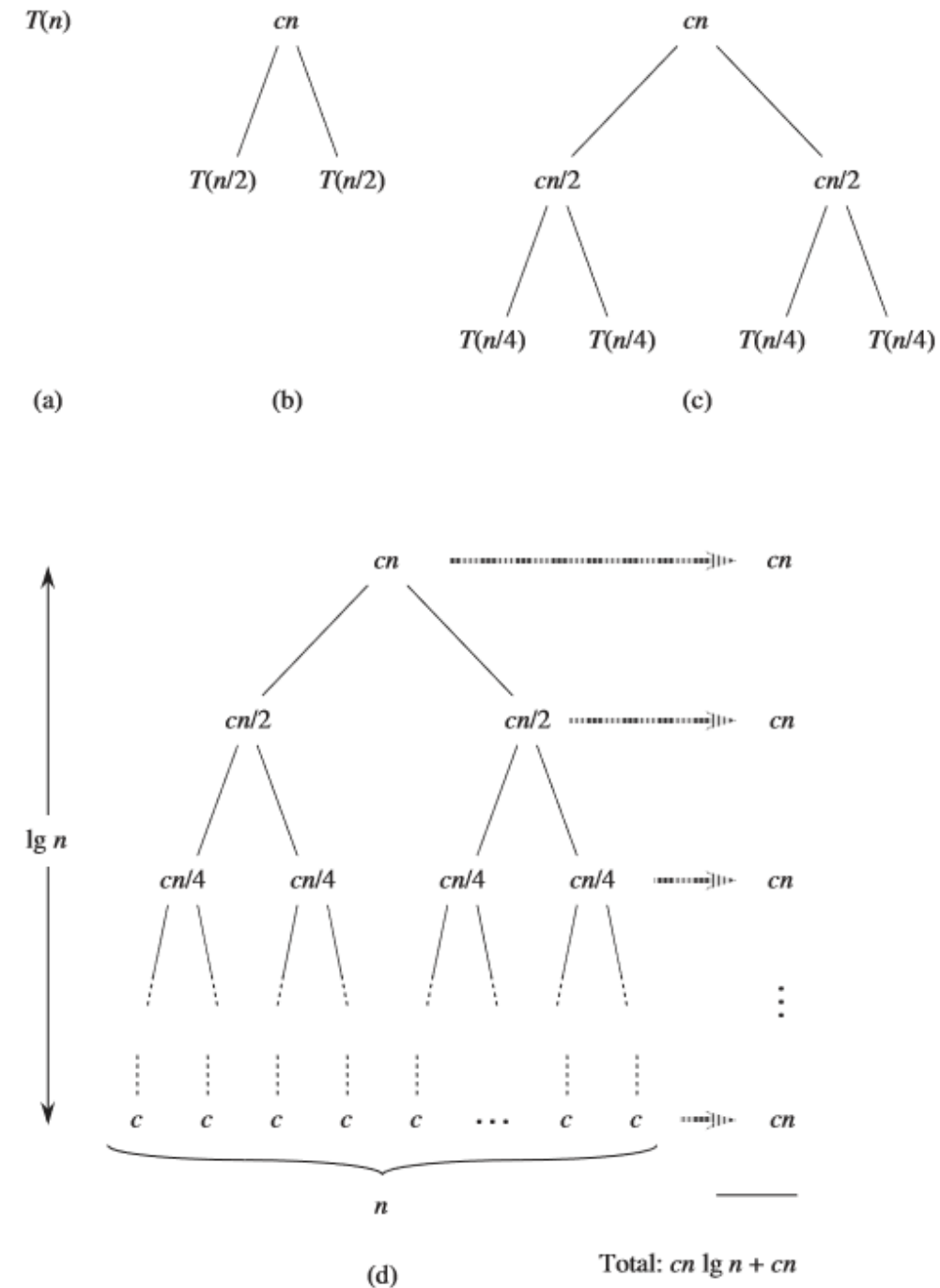
Merge sort

- ▶ Performance

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + n$$

  - ▶ for convenience, assume that n is a power of 2

- ▶ Total cost of the tree?

  - ▶ Cost of each level multiplied by number of levels

  - ▶ What is the cost of each level?

    - ▶ $O(n)$

  - ▶ What is the height of the tree?

    - ▶ $\log_2 n$ because, by definition, $x = \log_2 n \Rightarrow 2^x = n \Rightarrow \frac{n}{2^x} = 1$
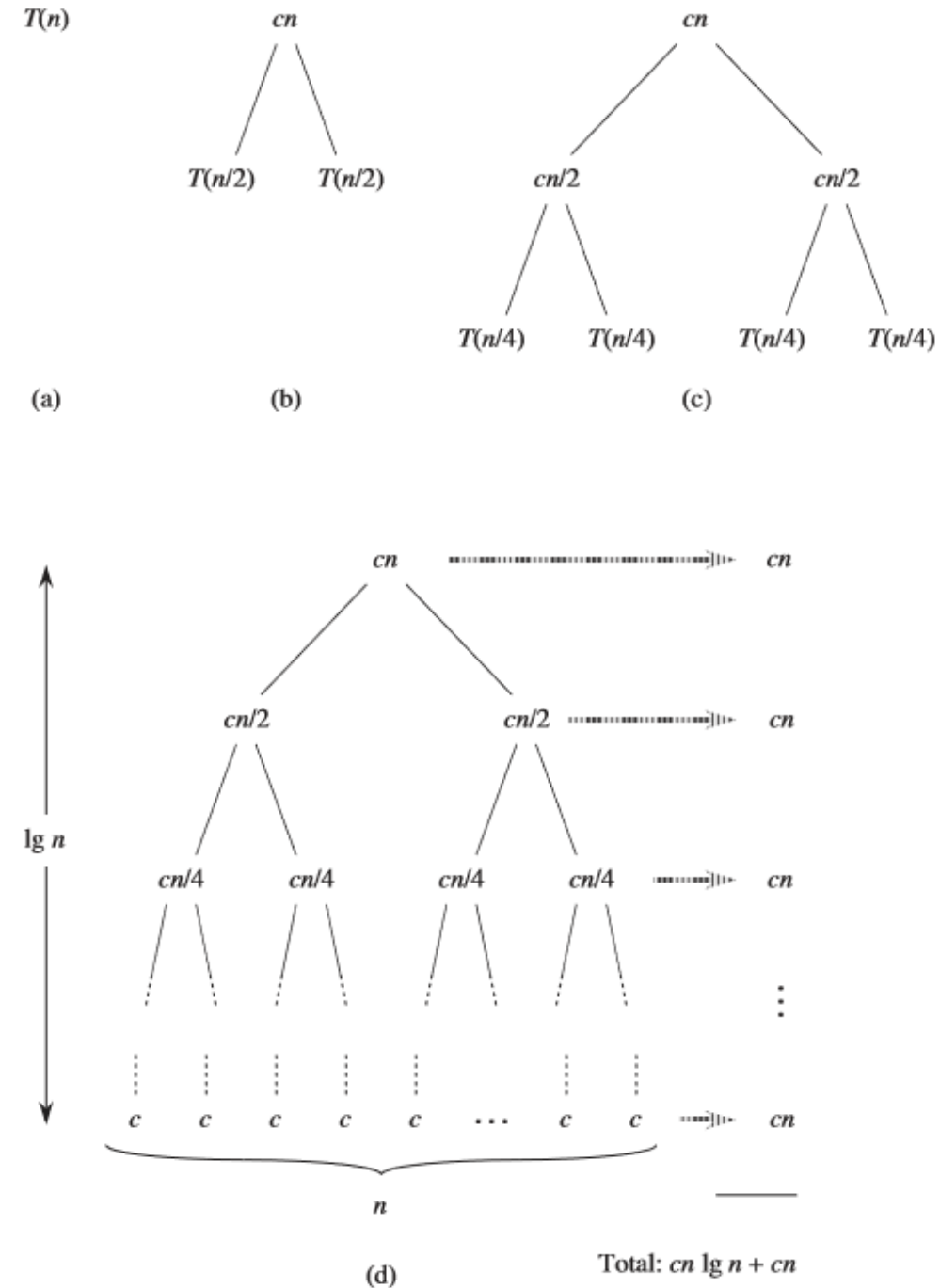
Merge sort

▶ Performance

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + n$$
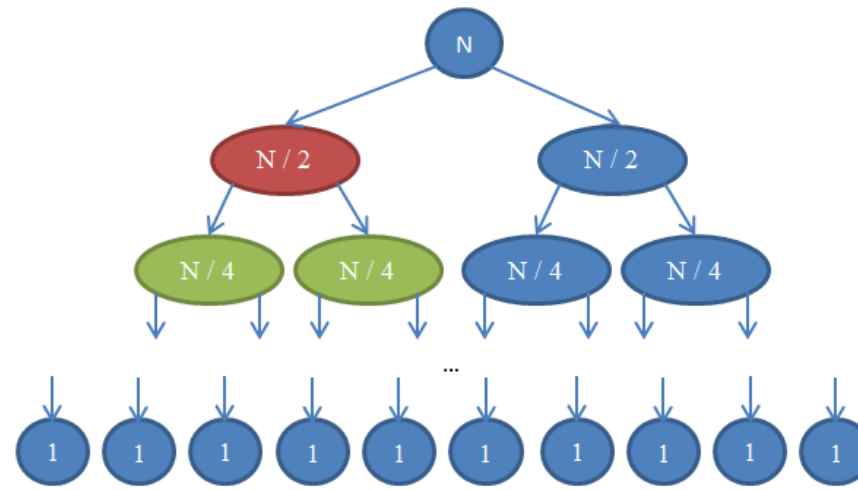
▶ for convenience, assume that n is a power of 2

▶ Total cost of the tree?

▶ Cost of each level → $O(n)$

▶ How many levels? → $\log_2 n + 1$

▶ Ignoring the constant 1 we get

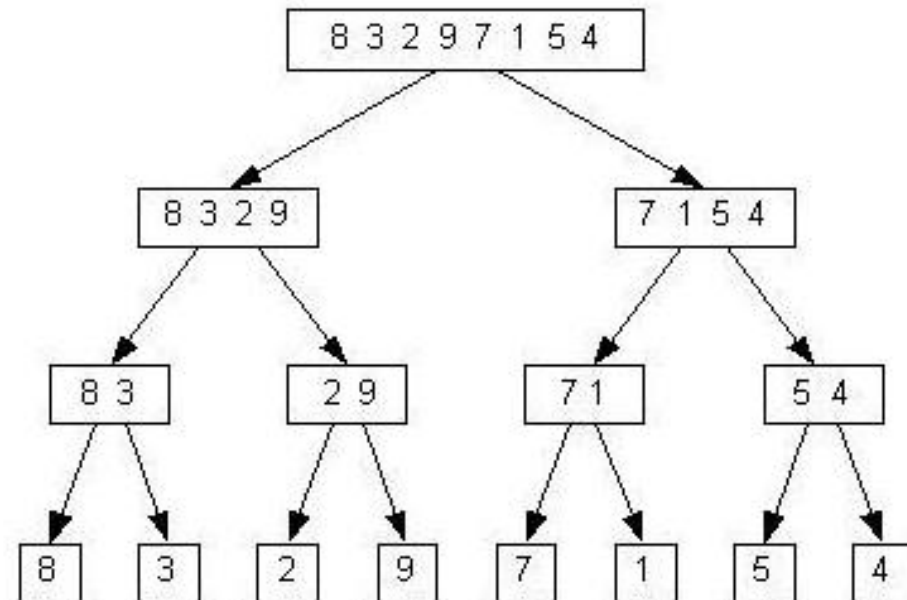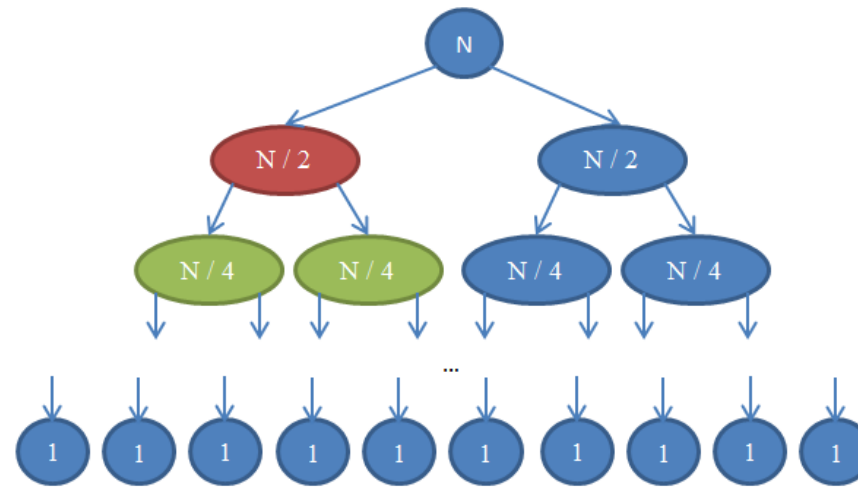$$\boldsymbol{T(n) = O(n \log n)}$$

Merge sort

▶ Recursion tree, example

 ▶ Let $n = 8$

 ▶ Levels of the recursion tree?

   ▶ First level: 1 node with 8 elements

   ▶ Second level: 2 nodes with 4 elements each

   ▶ Third level: 4 nodes with 2 elements each

   ▶ Fourth (and last) level: 8 nodes with 1 element each

 ▶ $\log_2 8 = 3$ because $2^3 = 2 \times 2 \times 2 = 8$

 ▶ $(\log_2 8) + 1 = 4$ → 4 levels

Merge sort

- Recursion tree, example

  - … what if n is not a power of 2?

  - Let $n = 5$

  - Levels of the recursion tree?

    - First level: 1 node with 5 elements

    - Second level: 2 nodes with max 3 elements each

    - Third level: 4 nodes with max 2 elements each

    - Fourth (and last) level: some nodes with max 1 element each

  - $\log_2 5 = 2.321 \ldots \rightarrow$ we round it to the next integer (3)!

  - $\lceil \log_2 5 \rceil + 1 = 4 \rightarrow 4$ levels
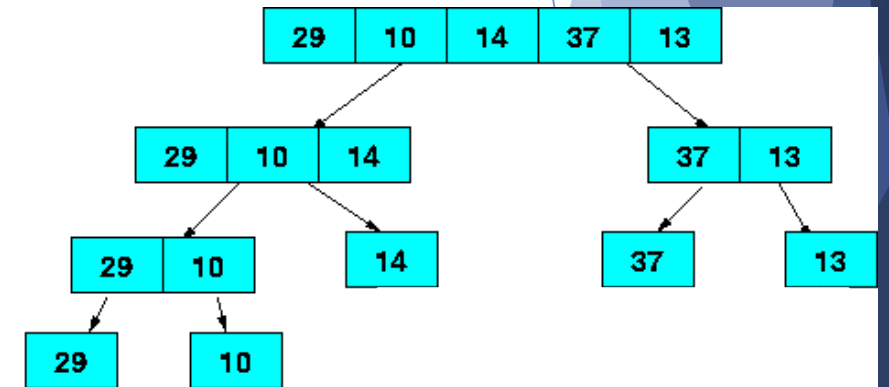
Homework

- Implement the two sorting algorithms
  - *Facultative*: try to make it generic with respect to the type of the elements being sorted (using a comparator)

- **START WITH THE FIRST EXERCISE OF THE PRACTICAL ASSIGNMENT!**