Databases and Algorithms

02-Fundamentals

Instructor: Rossano Schifanella

@SDS

Definition

- An algorithm is an **explicit**, **precise**, **unambiguous**, **mechanically-executable** sequence of elementary instructions.
- Term from the 9th century Persian mathematician al-Khwārizmī (father of the modern word algebra)
- **Algorithm ≠ Problem**
  - a **computational problem** is defined by an input, a desired output and the relationship between them.
- An algorithm describes a **computational procedure** that implements that input/output relationship

Definition

- In general there are **several** algorithms to solve the same computational problem
  - e.g., sorting, searching, integer multiplication, …
- An algorithm is **correct** if, for **every input instance, it halts with the correct output**.
  - if the problem is well specified, it should be possible to formally prove the correctness of an algorithm

Definition

PAPP Model

P ⇨ A ⇨ P ⇨ P

PROBLEM → **A**LGORITHM → **P**ROGRAM → **P**ROCESS

Definition

- As for many natural sciences, we are interested in a **generalization**.
  - examples: animals, plants, minerals, and so on
- Which are the observable properties of an algorithm that we can use for categorizing them in equivalence classes?

A classic problem: SORTING

INPUT:

A sequence of **n** numbers $\langle a_1, a_2, \dots, a_n \rangle$

OUPUT:

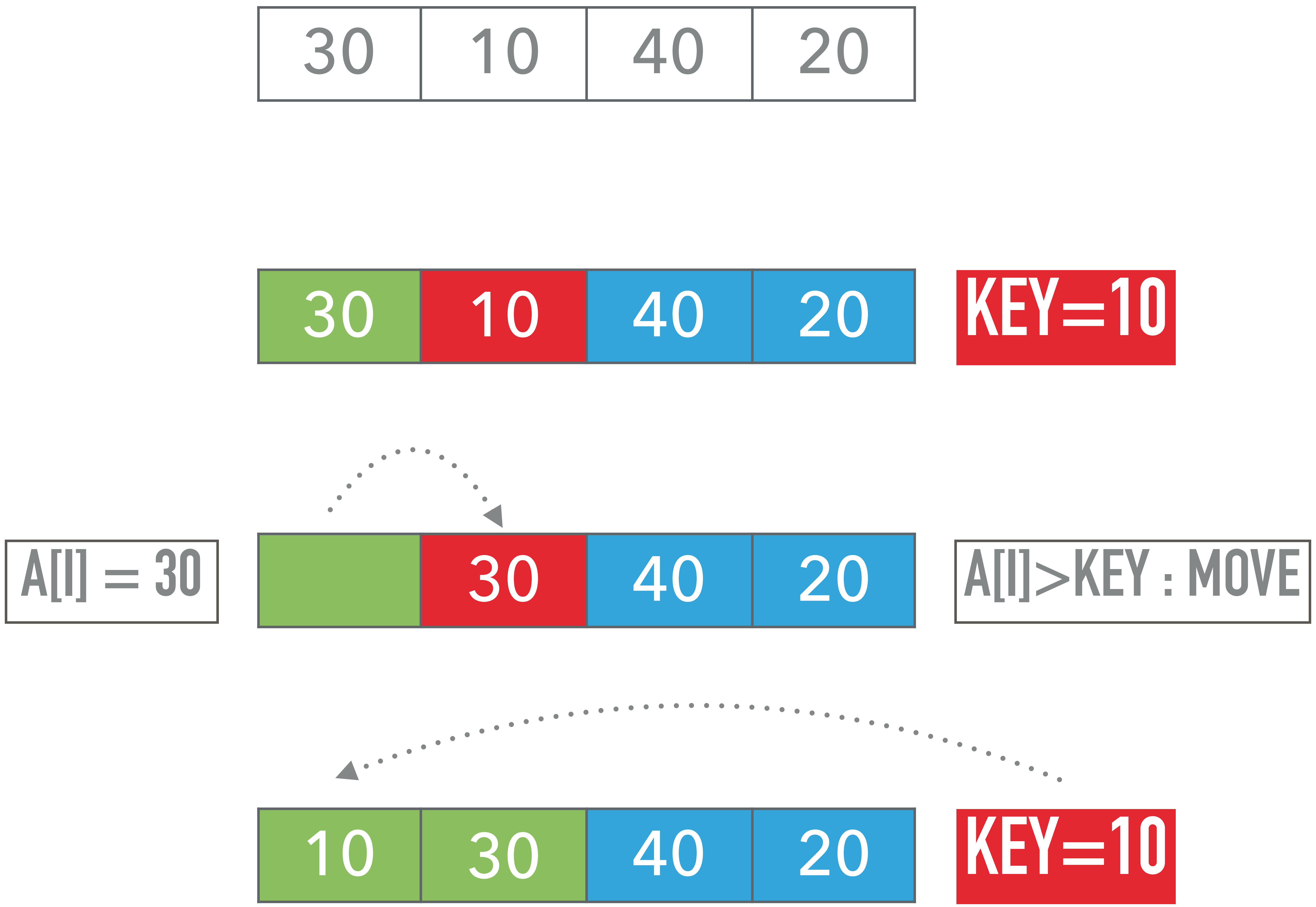
A **permutation** (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$

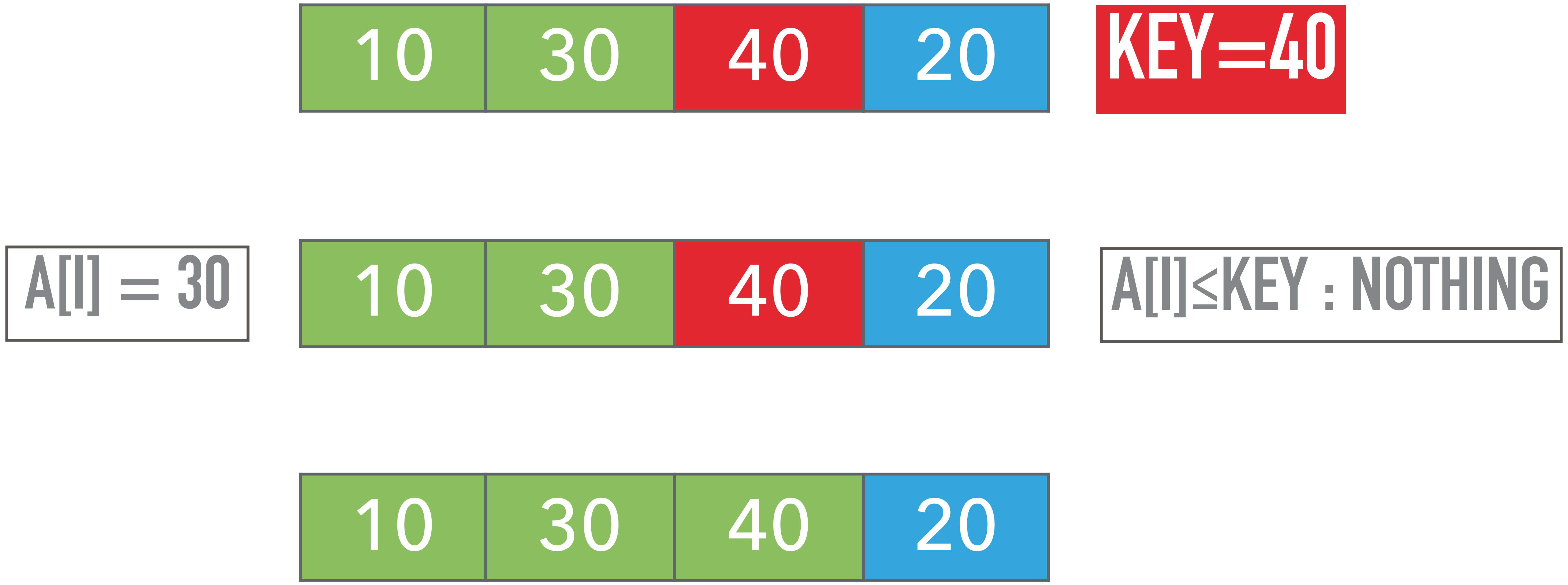
A First Solution: Insertion Sort

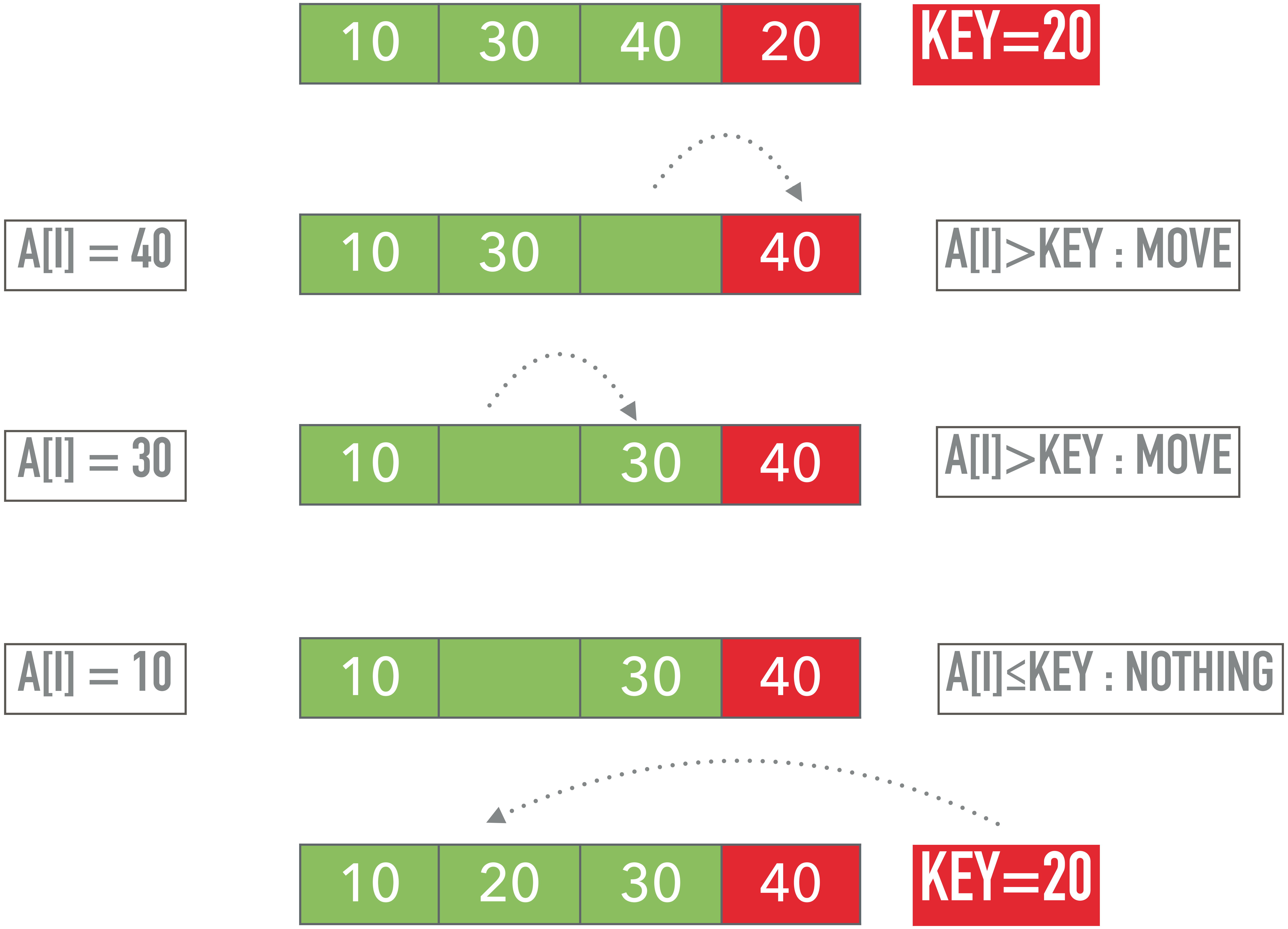
```
INSERTION-SORT(A)
  for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1]
    i = j-1
    while i > 0 and A[i] > key
      A[i+1] = A[i]
      i = i-1
    A[i+1] = key
```

What is a Pseudocode?

- **Approximate language** between the natural language and computer code.
- English phrases or lines of statements that used to solve specific problem by using short commands.
- Why a pseudocode called by this name?
  - Because it is not programming language so, computer didn’t understand this language.
- Why a pseudocode used?
  - Its very easy to produce code by any programming language .


30 10 40 20
30 10 40 20
KEY=10
A[I] = 30
30 40 20
A[I]>KEY : MOVE
10 30 40 20
KEY=10


10 30 40 20
KEY=40
A[I] = 30
10 30 40 20
A[I]≤KEY : NOTHING
10 30 40 20



10 30 40 20
KEY=20
A[I] = 40
10 30 40
A[I]>KEY : MOVE
A[I] = 30
10 30 40
A[I]>KEY : MOVE
A[I] = 10
10 30 40
A[I]≤KEY : NOTHING
10 20 30 40
KEY=20

Insertion Sort: Correctness

- **Loop invariant**

> AT THE START OF EACH ITERATION OF THE FOR LOOP, THE SUBARRAY A[1..J–1] CONSISTS OF THE ELEMENTS ORIGINALLY IN A[1..J–1], BUT IN SORTED ORDER

Does it hold for the proposed solution?

Loop Invariants

- **Statements about an algorithm that remain valid**
- We must show **three** things about loop invariants:
  - **Initialization:** statement is true before first iteration
  - **Maintenance:** if it is true before an iteration, then it remains true before the next iteration
  - **Termination:** when loop terminates the invariant gives a useful property to show the correctness of the algorithm

Analyzing Algorithms

HOW EFFICIENT IS AN ALGORITHM?
- RUNNING TIME
- MEMORY/STORAGE REQUIREMENTS, BANDWIDTH, ETC.

- We use the **RAM model**:
  - All memory equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time
    - Except, of course, function calls
  - Constant word size (space)
    - Unless we are explicitly manipulating bits

Analyzing Algorithms

- Performance depends on **input size**:
  - it depends on the problem definition: #items in the input, #bits, #nodes and #edges in a graph
- **Running time**:
  - number of primitive operations or steps executed
  - each line of the pseudocode takes a constant time c_i

Asymptotic performance

HOW DOES ALGORITHM BEHAVE AS THE PROBLEM SIZE GETS VERY LARGE?

- Depending on the input we can have:
  - **worst-case**: it gives an upper bound on the resources required by the algorithm
  - **average-case**: it characterizes the performance of the algorithm in case of an average input
  - **best-case**: it gives an lower bound on the resources required by the algorithm

Analysis of Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $\quad$ $key = A[j]$	c_2	$n-1$
3 $\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4 $\quad$ $i = j-1$	c_4	$n-1$
5 $\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $\quad\quad$ $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $\quad\quad$ $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $\quad$ $A[i+1] = key$	c_8	$n-1$

t_j: NUMBER OF TIMES THE WHILE LOOP IN ROW 5 IS EXECUTED FOR THAT VALUE OF j

Analysis of Insertion Sort

- The running time of the algorithm is the **sum of running times for each statement executed**; a statement that takes c_i steps to execute and executes n times will contribute $c_i \times n$ to the total running time.

$$
\begin{aligned}
T(n) \;=\; & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
& + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \,.
\end{aligned}
$$

Analysis of Insertion Sort

BEST-CASE => $t_j = 1$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

linear function of n:

$an + b$ for constants a and b that depend on the cost c_i

Analysis of Insertion Sort

WORST-CASE => $t_j = j$

compare each element A[j] with each element in the sorted subarray A[1..j-1]

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\ .
\end{aligned}$$

Analysis of Insertion Sort

WORST-CASE => $t_j = j$

compare each element A[j] with each element in the sorted subarray A[1..j-1]

<u>quadratic function of n</u>:

$an^2 + bn + c$ for constants a, b, and c that depend on the cost c_i

Analysis of Insertion Sort

AVERAGE-CASE => $t_j = j/2$

- Often roughly as bad as the worst case
- On average, half the elements in A[1..j-1] are less than A[j] , and half the elements are greater. On average, therefore, we check half of the subarray A[1..j-1], and so t_j is about j/2.
- Running time turns out to be a **quadratic function** of the input size, just like the worst-case running time.

Order of Growth

- We are not really interested in the abstract costs c_i neither in the constants a, b, or c (e.g., in the worst-case analysis)
- We are interested more in the **rate of growth**, so we consider only the **leading term** of the running time formula
  - lower-order terms are relatively insignificant for large inputs
  - we don’t consider the constant of the leading term since is less significant that the rate of growth for large inputs
- We use the **Θ-notation** (theta) (and others!) for that
  - e.g., worst-case of insertion sort is in **$\Theta(n^2)$**

Questions?

@rschifan

schifane@di.unito.it

http://www.di.unito.it/~schifane