Python & C++ Program Design

-- Algorithm & Function: Sorting!

Junjie Cao @ DLUT

Summer 2022

https://github.com/jjcao-school/c

Content

- Data structure & algorithm – a brief introduction
- Sorting algorithm
  - Why sorting
  - Selection Sorting
  - Insertion Sorting
- Pseudo Code

Problem Solving by Human

input

↓

Manipulation

↓

output

Problem Solving by Computer



input

↓

Algorithm

↓

output



What is an Algorithm?

Data input

Data manipulation

Data output

Purpose of learning the course: Solving problems by computer

- Data: information being analyzed

  e.g.: numbers, words, movies

- Algorithm: a computational procedure for solving a problem

- Data structure: The way the data is organized

Why Study DA?

- important for all other branches of computer science
- plays a key role in modern technological innovation
- provides novel "lens" on processes outside of computer science and technology
  - quantum mechanics, economic markets, evolution
- challenging (i.e., good for the brain!)
- fun

Why sorting?

The Sorting 排序 Problem

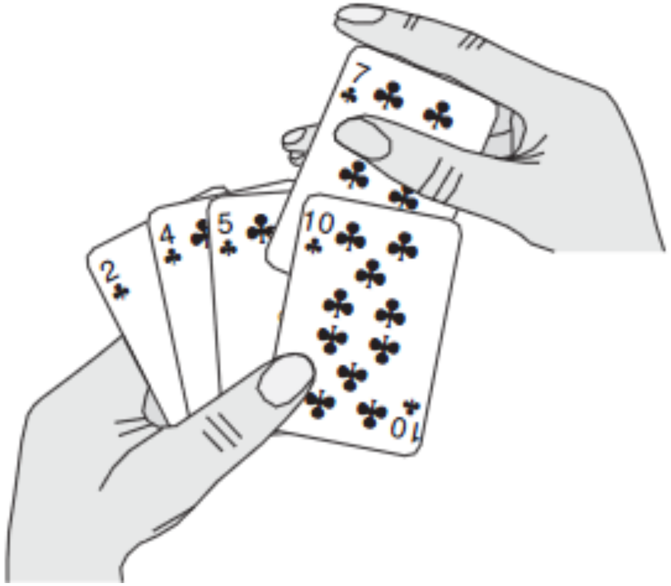
- Example:

- Input: A sequence of n numbers $< a_1, a_2, \ldots, a_\mathrm{n} >$

- Output: A permutation (reordering) $< b_1, b_2, \ldots, b_\mathrm{n} >$ of the input sequence such that $b_1 \leq b_2 \ \ldots \leq b_\mathrm{n}$

Why is sorting worth so much attention?

1. Sorting is **basic** building block



2. Historically, ¼ mainframe cycles were spent sorting data [Knu98]. It remains most **ubiquitous** in practice.
    - Could you make an example?

Algorithm Efficiency

To sort 10 million numbers:

- Insertion sort: $O(c_1 n^2)$, c_1?=2

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{\boxed{10^{10}} \text{ instructions/second}} = 20{,}000 \text{ seconds (more than 5.5 hours)}$$

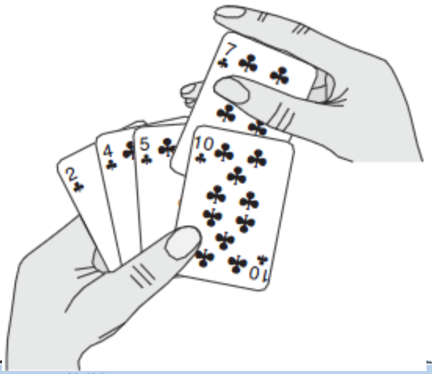
- Merge sort: $c_2 n \log n$, c_2?=50

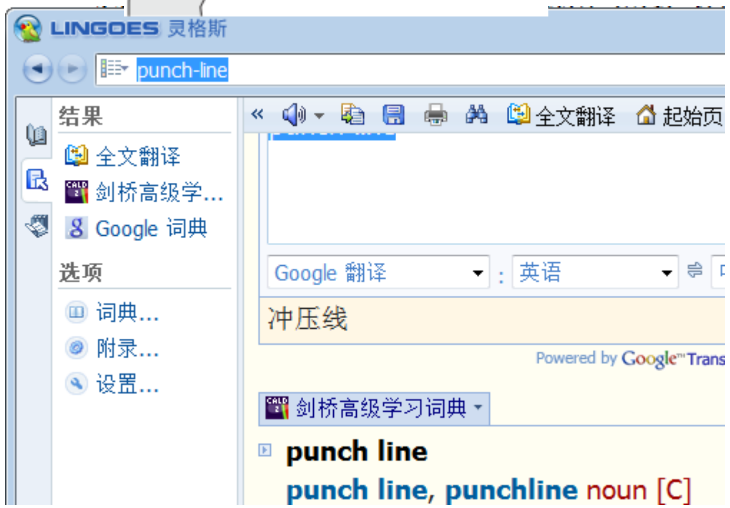
$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{\boxed{10^7} \text{ instructions/second}} \approx 1163 \text{ seconds (less than 20 minutes)}$$

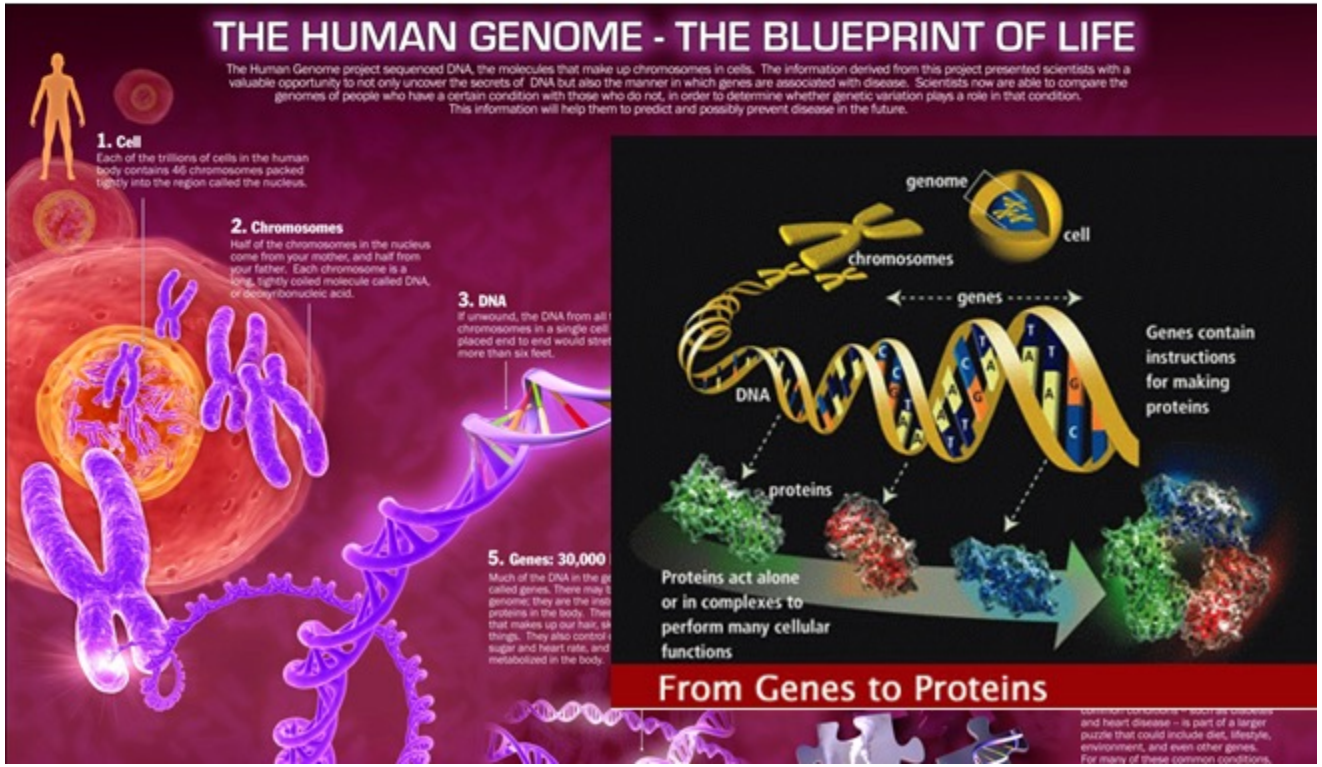
- Many important problems can be reduced to sorting, so we can use our clever *O*(*n* log *n*) algorithms to do work

Applications of Sorting - Searching

- Sorting an array can make searching an array more efficient, not only for humans, but also for computers.
- Binary search tests whether an item is in a dictionary in $O(\log n)$ time, provided the keys are all sorted







Applications of Sorting - Closest pair

- Given a set of *n* numbers, how do you find the pair of numbers that have the smallest difference between them?

5	1	12	-5	16	2	12	14

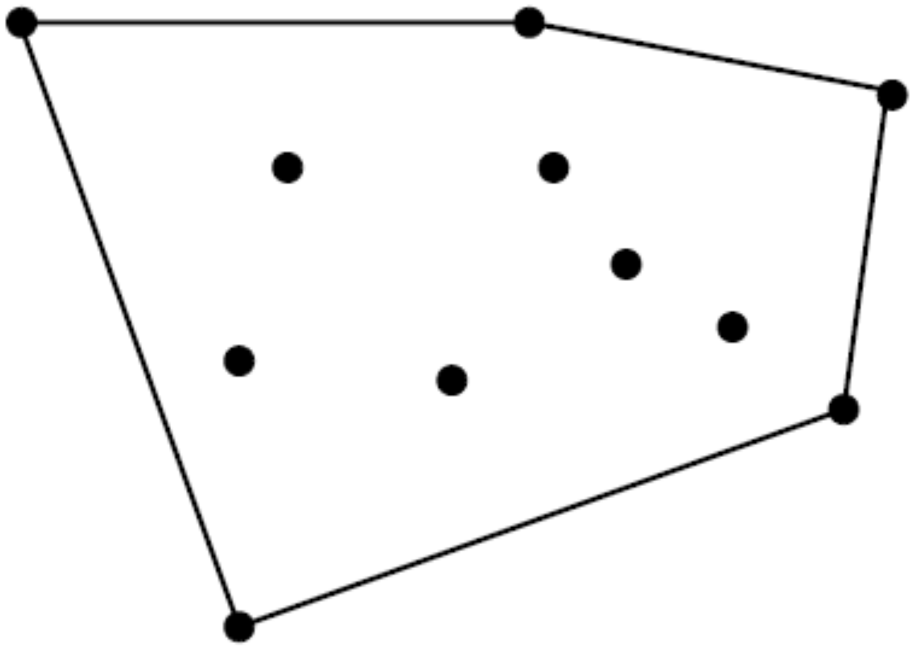
- Once the numbers are sorted, the closest pair of numbers must lie next to each other somewhere in sorted order. Thus, a linear-time scan through them completes the job: **$O(n) + O(n\log n) = O(n\log n)$**

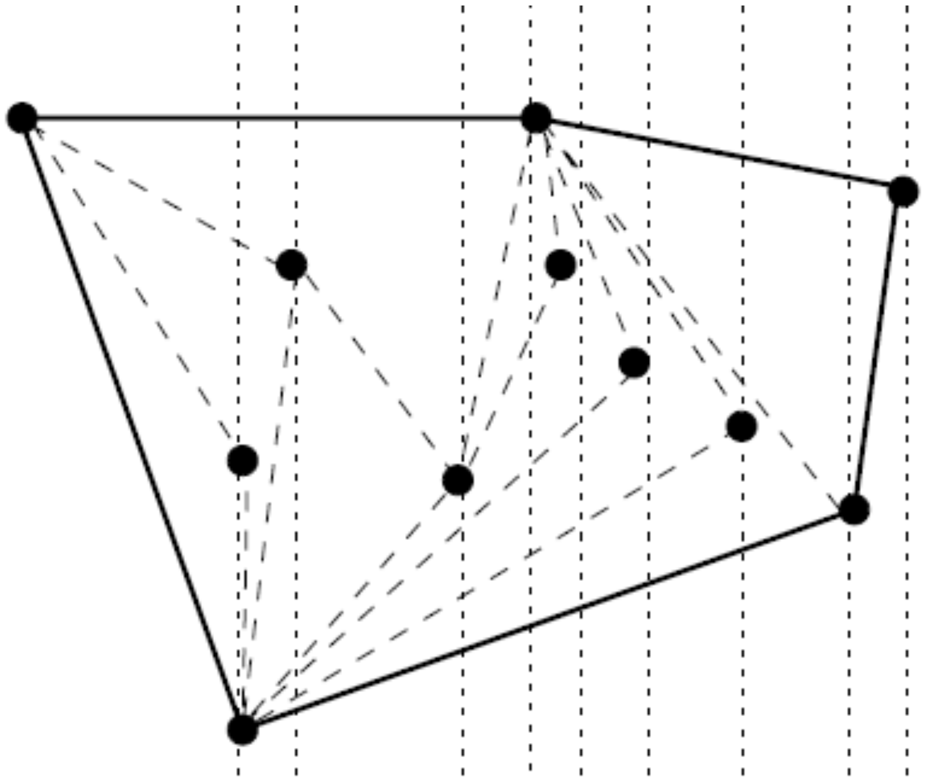
-5	1	2	5	12	12	14	16

- For point cloud processing: k-nearest neighbors or neighbors in ball

Applications of Sorting – Convex Hall

- rubber band stretched over the points
- Collision detection in Game, etc.
- The total time is linear after the sorting has been done (by *x*-coordinate)





[Skiena2] ch4.1

Why review standard sorting when you are better off *not* implementing them and using built-in library functions instead?

1. Most of interesting **ideas** appear in sorting:
   - divide-and-conquer, data structures, randomized algorithms

1. Typical computer science students study the basic sorting algorithms at least **three times** before they graduate: first in **introductory programming**, then in **data structures**, and finally in their **algorithms** course.

2. Sorting is thoroughly studied. Dozens of diff algorithms exist, most of which possess some **particular advantage** over others in certain situations.

Sorting!!

- Never be afraid to spend time sorting, provided you use an efficient sorting routine O(nlogn)

- **Sorting lies at the heart of many algorithms. Sorting the data is one of the first things any algorithm designer should try in the quest for efficiency.**

Sorting

Could you think of an algorithm now?

|31	41	59	26	41	58
26	|31	41	59	41	58
26	31	|41	41	59	58
26	31	41	41	58	59

First Idea in Our Head

sort A

```
for i = 1:n
    [B[i], pos] <- min A
    A.remove(pos)
end
```

min A

```
for i = 1:n
    if tmpv > A[i]
        tmpv = A[i]
        tmpi = i
    end
end
```

A.remove(pos)

```
A.Size <- A.size-1
for i = pos:A.size
    A[i] <- A[i+1]
end
```

"Noble's method", easy to think, high cost for implementation:

1. Additional storage space
2. High cost operation for array, even list $O(c_1 n^2)$

Selection Sort **(When no additional storage space is allowed)**

1. $O(c_1 n^2)$, quite inefficient for sorting large data volumes
2. Notable for **simple-to-code**

Algorithm:

1. **sorted one** and **unsorted one**
2. At each step, algorithm finds **minimal element** in the **unsorted part** and adds it to the end of the **sorted one**

```
for i = 1:n,
    k = i
    //invariant: a[k] smallest of a[i..n]
    for j = i+1:n
        if a[j] < a[k] then k = j
    //invariant: a[1..i] in final position
    swap a[i,k]
end
```

Several Sort Algorithms

http://www.sorting-algorithms.com

Selection Sort - Complexity analysis

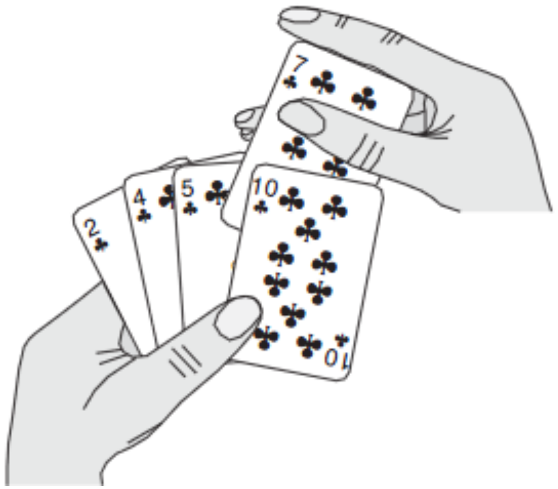
- **Selection sort performs *n* iterations, where the average iteration takes *n/2* steps, for a total of $O(n^2)$ time**

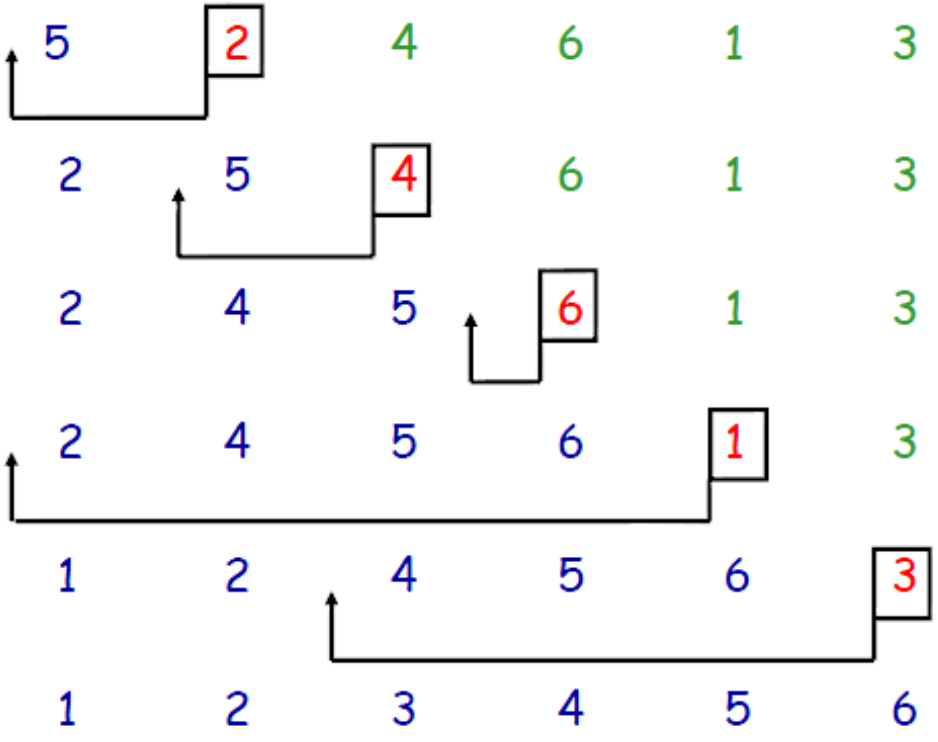
```
for i = 1:n,
    k = i
    for j = i+1:n
        if a[j] < a[k] then k = j
    swap a[i,k]
end
```

- Number of swaps may vary from zero (in case of sorted array) to n - 1 (in case array was sorted in reversed order), which results in O(n) number of swaps.
- So selection sort requires **n - 1 number of swaps at most**, makes it very efficient in situations, **when write operation is significantly more expensive, than read** operation.

Insertion Sort

1. **Although $O(n^2)$, it is applied in practice for sorting a small number of elements (8-12 elements).**
2. Outperforms most of quadratic sorting algorithms, like selection & bubble
3. **works the way many people sort a hand of playing cards**
4. **Somewhat resembles selection sort**

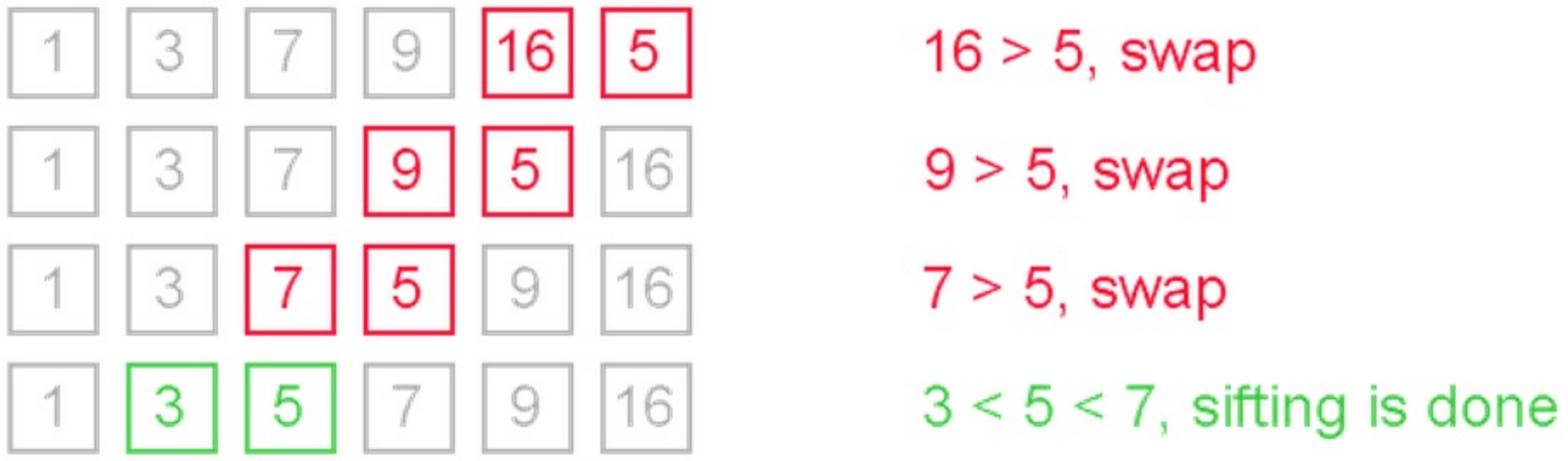




Insertion Sort, $O(c_1 n^2)$

1. **sorted one** and **unsorted one**
2. At the beginning, **sorted part** contains **first element** of the array and **unsorted one** contains the rest.
3. At each step, algorithm takes **first element** in the **unsorted part** and **inserts** it to the right place of the **sorted one.**

Ideas of Insertion

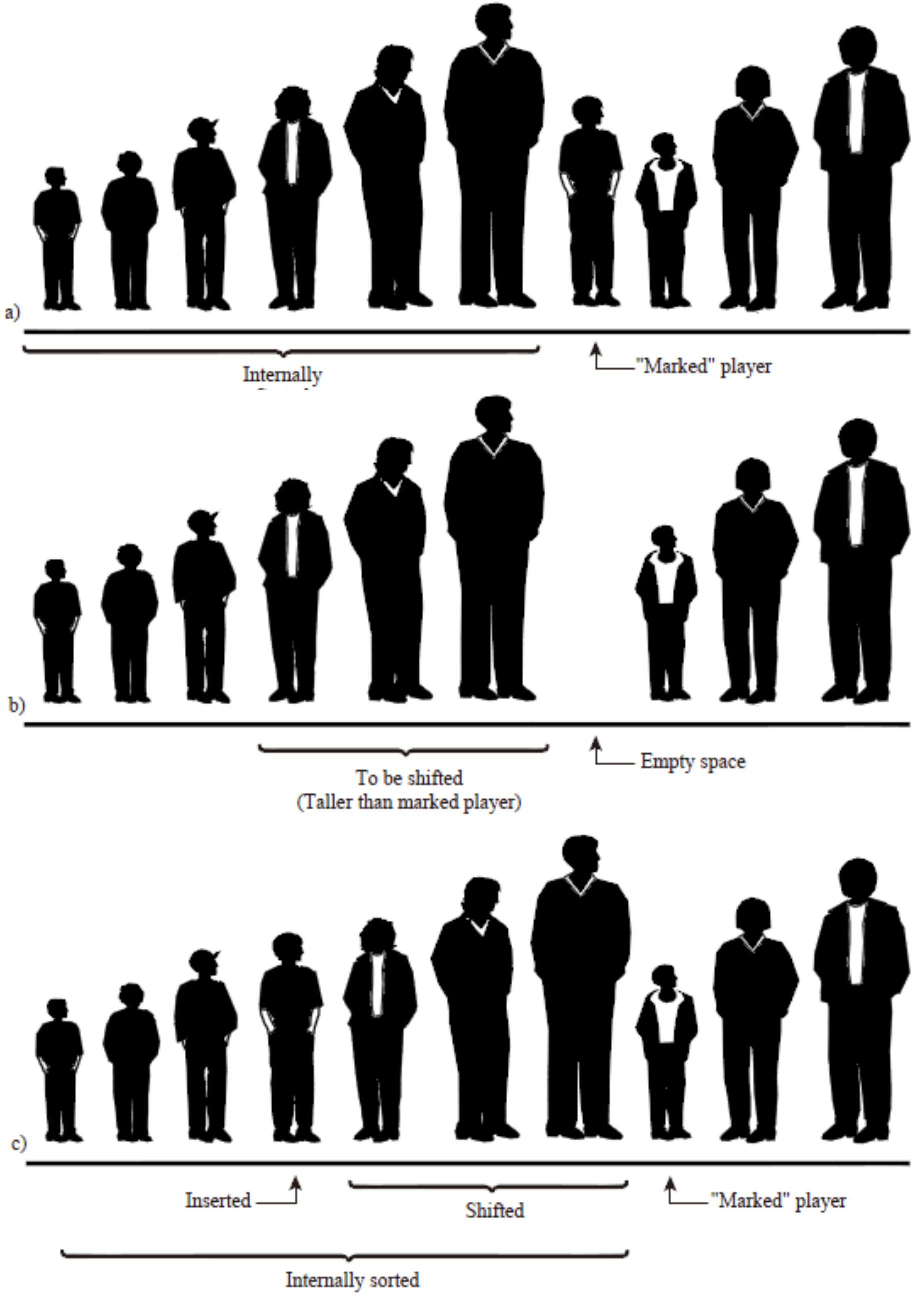



```
for i = 2:n,
   //invariant: a[1..i] is sorted end
   for (k = i; k > 1 && a[k] < a[k-1]; k--)
         swap a[k,k-1]
end
```

http://www.sorting-algorithms.com

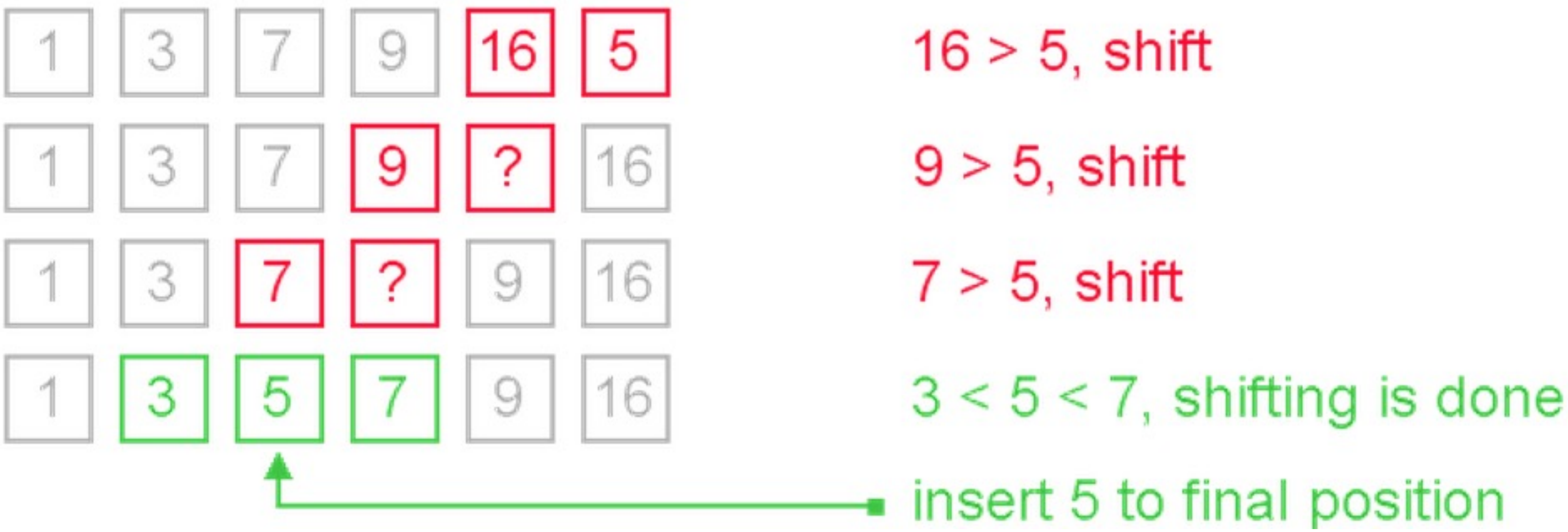
Next implementation **eliminates** those unnecessary writes.

Shifting instead of swapping



[Lafore24]

Shifting instead of swapping




```
For j=2:n
    key<-A[j]
    %Insert A[j] into the sorted sequence A[1 .. j-1]
    for (i = j-1:0; A[i]>key; --i)
        A[i+1] <-A[i]
    A[i+1] <-key
end
```

Selection & Insertion Sort

- Both $O(n^2)$, but insertion sort is faster.
- It uses the order info to reduce the average n/2 operation furthermore.

```
for i = 1:n,
    k <- i
    for j = i+1:n
        if a[j] < a[k] then k <- j
    swap a[i,k]
end
```

becomes

Pseudo Code

- A schematic description of the algorithm which can be **easily translated** to a “real” programming language
  - Common in the literature
  - Easily translated into real code

```
for i = 1:n,
    k <- i
    for j = i+1:n
        if a[j] < a[k] then k <- j swap a[i,k]
end
```

  - More information can be found in [CLRS] book

Pseudo Code – Basic Operators

```
for i = 1:n,
    k <- i
    for j = i+1:n
        if a[j] < a[k] then k <- j swap a[i,k]
end
```

- "←" – assignment operator
- "for", "while", "repeat-until" – loop operators
- "if-else", "case" – condition operators
- "return" – return from procedure
- Lots of other operators with their meaning obvious from their names, e.g.,
  - "swap", "delete", "new", '==' etc.

Pseudo Code - Conventions

- Indentation indicates block structure
- Variables are local to a given procedure
- A[i] denotes the i-th element of the array A
  - A[i…j] denotes the subarray containing elements A[i], A[i+1],…, A[j]
- An empty pointer value is denoted by NIL
- Fields (and methods) in the composite objects are accessed by ".",
  - e.g. Person.id, Person.FirstName, Person.FamilyName, etc.