

**T**eaching **L**ondon **C**omputing

## **A Level Computer Science**

# **Topic 2: Searching and Sorting**



**COMPUTING AT SCHOOL**  
EDUCATE · ENGAGE · ENCOURAGE



SUPPORTED BY  
**MAYOR OF LONDON**



# Aims

- Understanding and implement
    - Linear search
    - Binary search of sorted lists
  - Introduce computational complexity
  - Understand sorting algorithms
    - Bubblesort
    - Insertion Sort
    - Quicksort
-

# Why Learn Standard Algorithms?

- Real programmers never implement these!
  - They are in the library
- We are going to learn the importance of a good algorithm

Better a slow computer and a fast algorithms than  
a slow algorithm on a fast computer.

---

# Linear Search

---

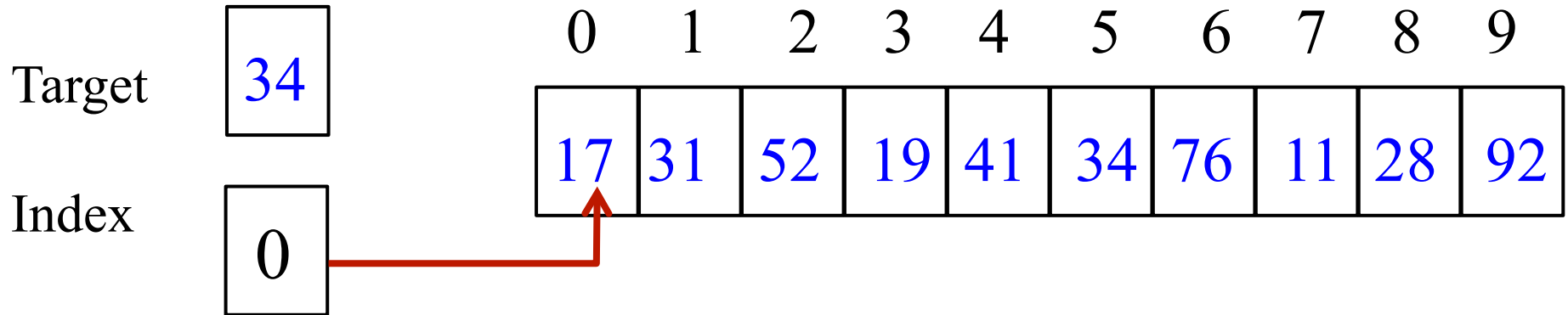
# Search: The Problem

- Find a target value in the list
  - Is it there?
  - If so, at what index?

0	1	2	3	4	5	6	7	8	9	10	11
17	31	52	19	41	34	76	11	28	92	44	61

- Target = 41, found at index 4
  - Target = 27, not found
-

# Linear Search



- Idea: look at each entry in turn
  - Steps
    - Start at index = 0
    - Is `Array[Index]` equal to the target?
    - If yes, stop; otherwise increment the index
    - Stop when index is one less than the length
-

# Linear Search – Algorithm

- Algorithm in pseudo code
- Array is A

```
index = 0
while index < length of array
    if A[index] equals target
        return index
    index = index + 1
return -1 to show not found
```

---

# Exercise 1.1: Code Linear Search

```
index = 0
while index < length of array
    if A[index] equals target
        return index
    index = index + 1
return -1 to show not found
```

- Pseudo code

- Outline of code to complete

```
def findLin(A, target):
    # find the target in array A
    # return index or -1
```

```
    ...
```

```
    ...
```

```
print(findLin([2,3,4], 3))
print(findLin([2,3,4], 7))
```



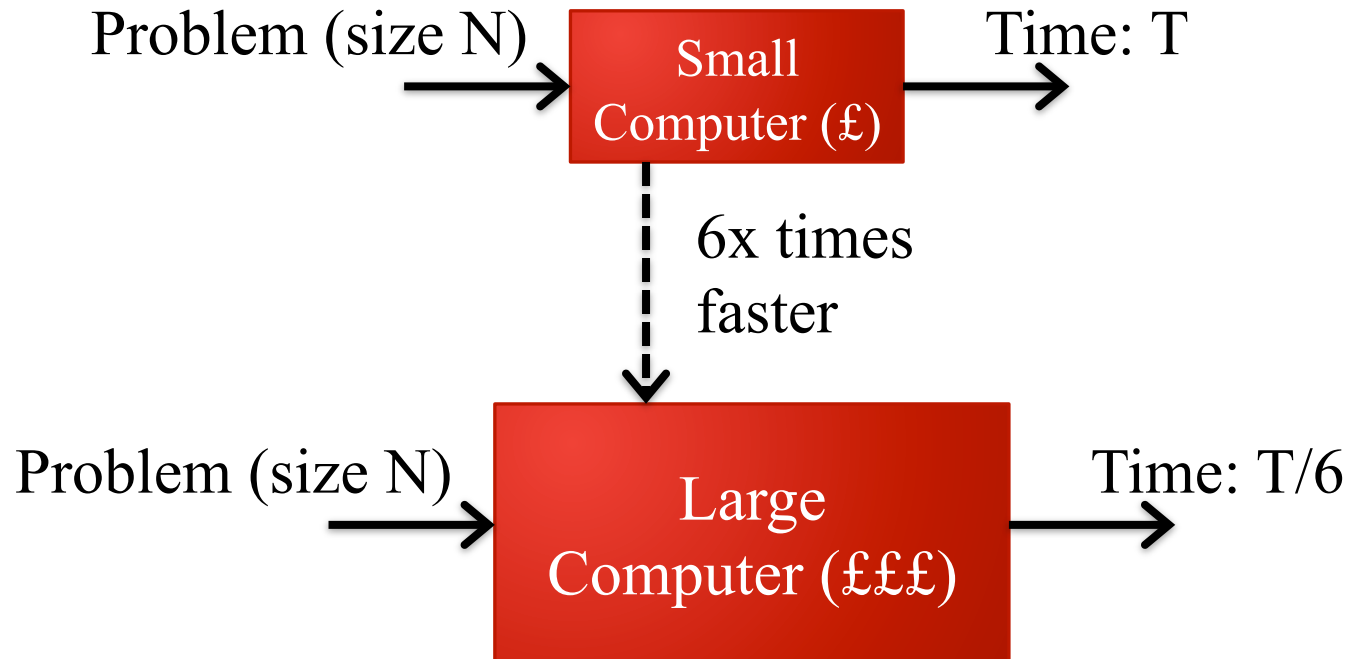
# Computational Complexity

Compare the efficiency of algorithms

---

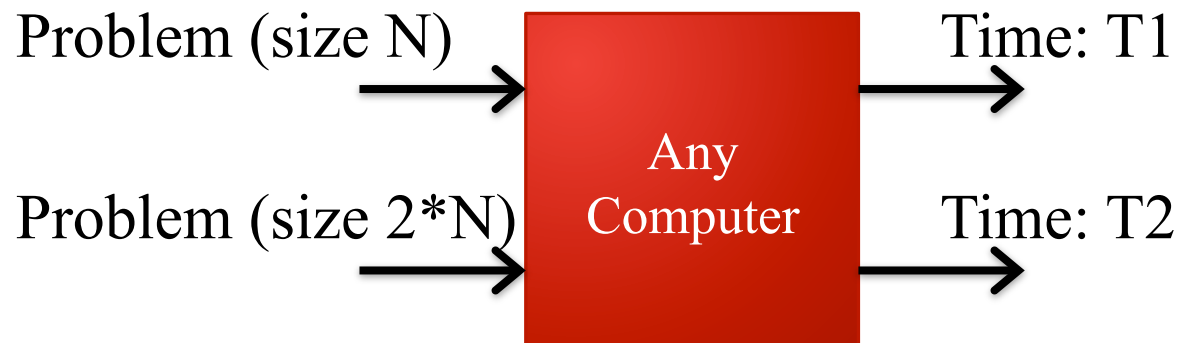
# Efficiency – How Fast?

- Doesn't it depend on the power of computer?



# Efficiency – How Fast?

- We only care about how the time increases
  - Maybe the time stays the same
  - Maybe doubling the size, doubles the time
  - Maybe doubling the size, more than doubles the time



# Linear Search: How Many Steps?

- On average, how many steps?
- Assume:
  - Target is present
  - List length  $N$
- Expect to look at 50% of locations on average
- Complexity
  - Length  $N \rightarrow N/2$  steps
  - It does not matter how long each step takes

We are assuming same time to access any location. True in arrays (not generally in lists).

# Big-0 Notation

- Time (size =  $N$ ) =  $N / 2$
  - Suppose for size 10, i.e. 5 steps, times is 15 ms
    - Size 20  $\rightarrow$  10 steps  $\rightarrow$  30 ms
    - Size 40  $\rightarrow$  20 steps  $\rightarrow$  60 ms
    - etc.
  - BUT
    - We do not care about the exact time
    - We only care how the time increases with the size
  - Linear search has complexity  $O(N)$
-

# Exercise 1.2 Complexity

- Discuss the statements on the complexity of linear search
    - *Which is correct?*
-

# Binary Search

Searching a sorted list

---

# Searching a Sorted List

- Question: why are books in the library kept in order?



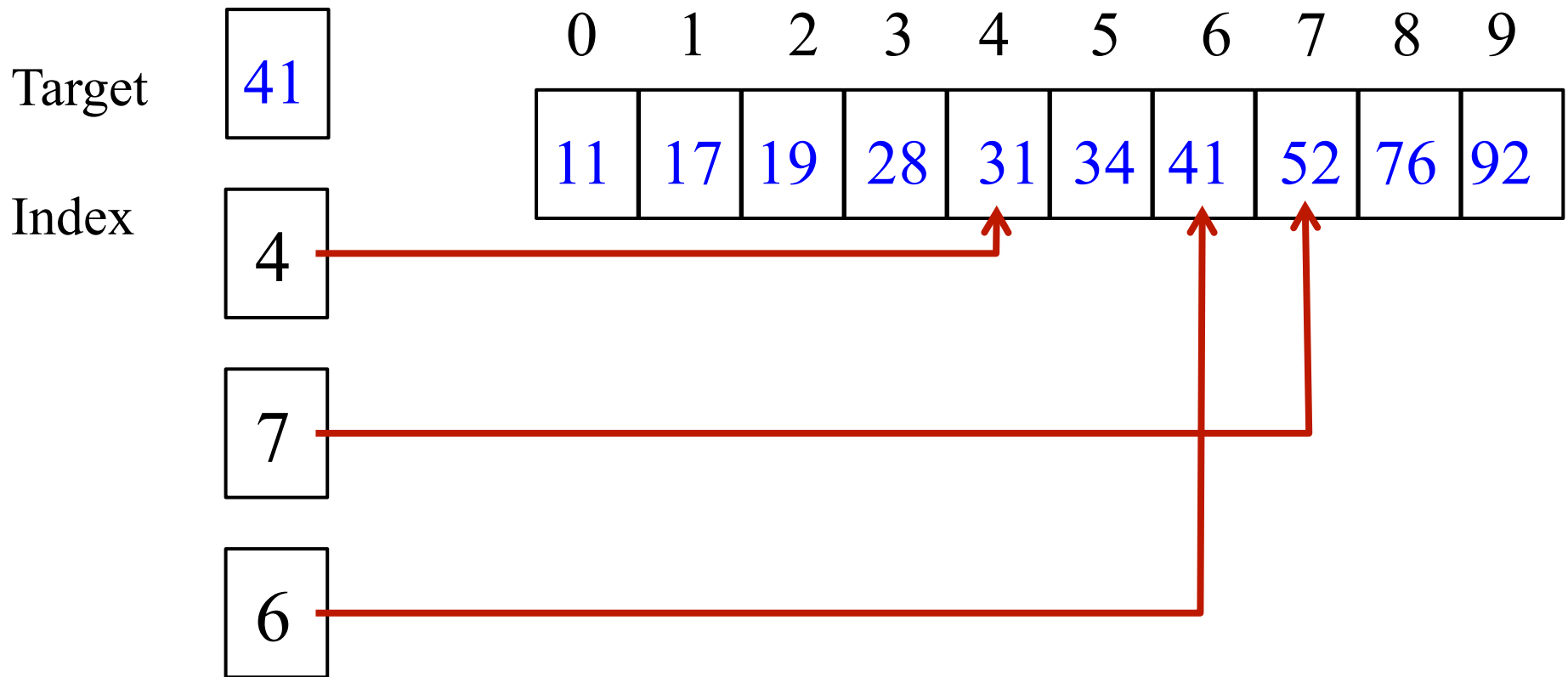


# Searching a Sorted List

- Question: why are books in the library kept in order?
  - *In an ordered array, we do not have to look at every item*
    - “Before this one”
    - “After this one”
    - ... quickly find the correct location
  - What is the best algorithm for looking?
-

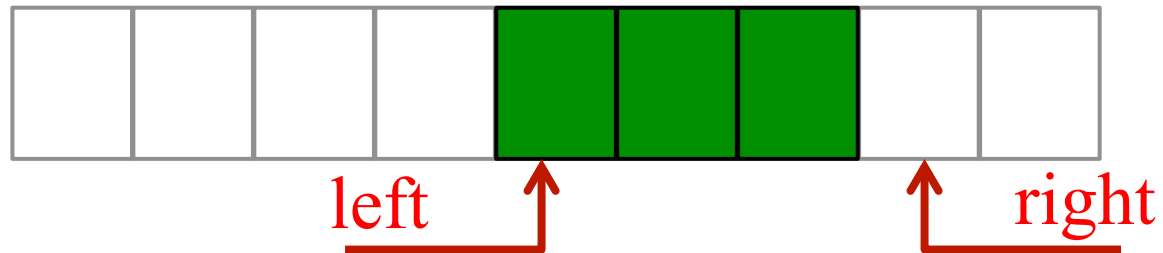
# Binary Search – Sorted Lists

- Which half is it in? Look in the middle.

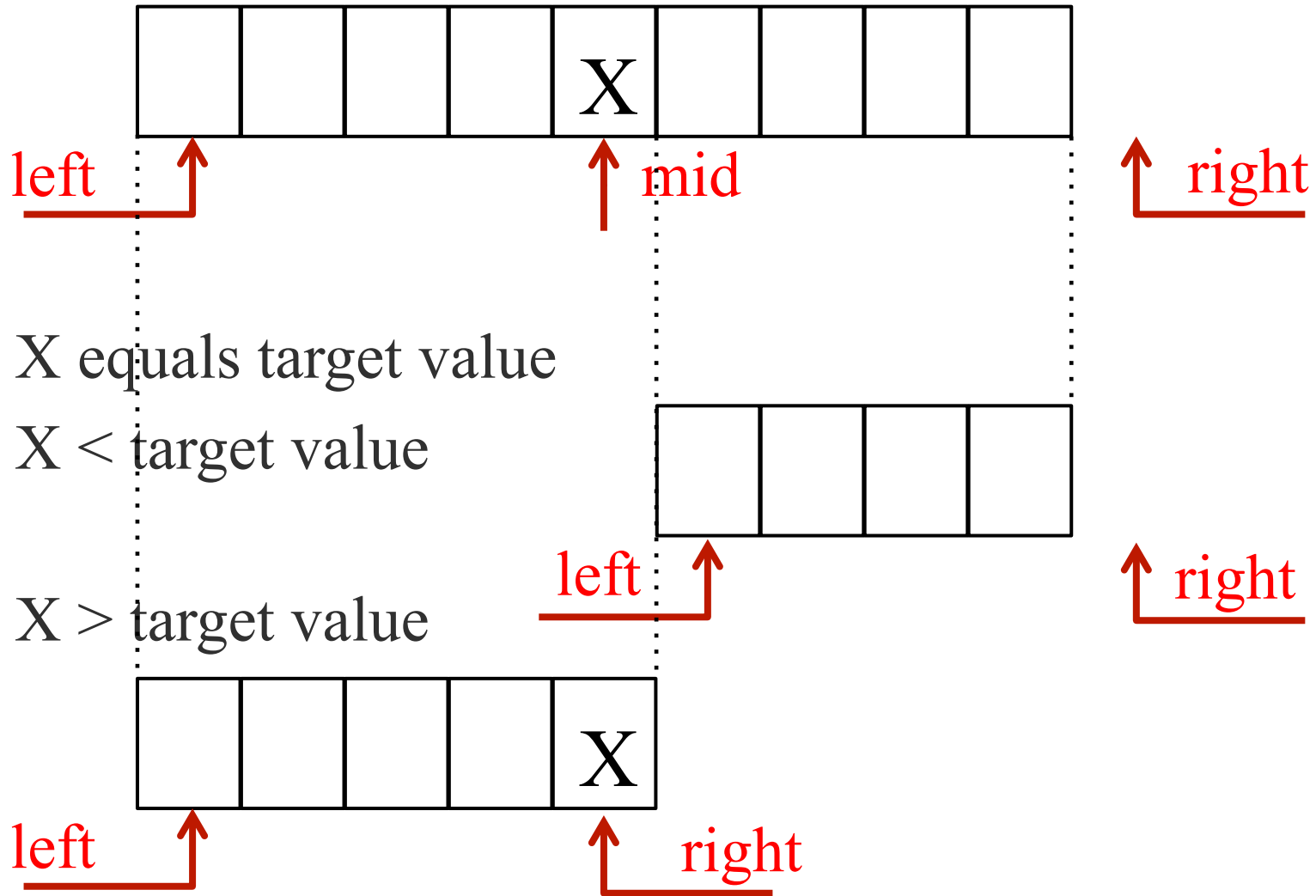


# Binary Search – Algorithm

- Key idea: in which part of the array are we looking?



# Binary Search – 3 Cases



# Binary Search – Algorithm

```
left = 0
right = length of array
while right > left:
    mid = average of left and right
    if A[mid] equals target
        found it at 'mid'
    if A[mid] < target
        search between mid+1 & right
    otherwise
        search between left & mid
return not found
```

---

# Binary Search – Python

```
def BSearch(A, target):  
    left = 0  
    right = len(A)  
    while right > left:  
        mid = (left + right) // 2  
        if A[mid] == target:  
            return mid  
        elif A[mid] < target:  
            left = mid+1  
        else:  
            right = mid  
    return -1
```

# Binary Search – Complexity

0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1

11	17	19	28	31	34	41	52
----	----	----	----	----	----	----	----

- Number of steps = number of binary digit to index
  - $O(\log N)$
-

# Exercise 1.3

- Using playing cards and a pointer (or pointers), show how the following search algorithms work
    - Linear search
    - Binary search
  - *A pen or pencil can be used as a pointer*
  - *Do we need a pointer?*
-



# **How We Describe Algorithms**

---

# Steps for Understanding Algorithms

1. The problem to be solved
  2. The key idea
  3. The steps needed
  4. The state represented
  5. The cases to consider
  6. Pseudo code
  7. Code
-

# Understanding Linear Search

---

## Steps

- *The problem to be solved*
- *The key idea*
- *The steps needed*
- *The state represented*
- *The cases to consider*
- *Pseudo code*
- *Code*

## Application

- Find an item in an unsorted list
  - Look at each item in turn
  - *Show it with e.g. cards*
  - How far we have got
  - Found or not found
  - ...
-

# Understanding Binary Search

---

## Steps

- *The problem to be solved*
- *The key idea*
- *The steps needed*
- *The state represented*
- *The cases to consider*
- *Pseudo code*
- *Code*

## Application

- Find an item in an **sorted** list
  - Halve the part to be searched
  - *Show it with e.g. cards*
  - The two end of the search space
  - Left half, found, right half
  - ...
-

# Sorting

---

# Sorting: The Problem

0	1	2	3	4	5	6	7	8	9
17	31	52	19	41	34	76	11	28	92
11	17	19	28	31	34	41	52	76	92

- Arrange array in order
    - Same entries; in order – swap entries
  - Properties
    - Speed, space, stable,
-


# Discussion

- Sort a pack of cards
- Describe how you do it



# Bubble Sort – Insight

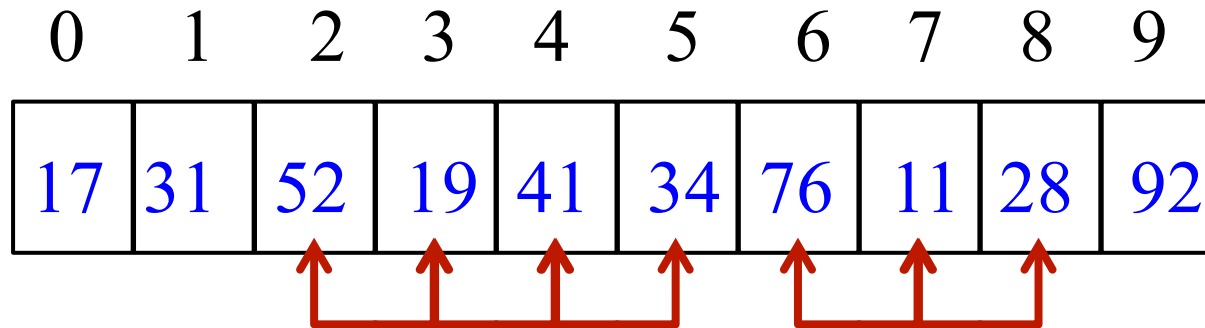
0	1	2	3	4	5	6	7	8	9
17	31	52	19	41	34	76	11	28	92



- Librarian finds two books out of order
  - Swap them over!
  - Repeatedly
-



# Bubble Sort – Description



- Pass through the array (starting on the left)
- Swap any entries that are out of order
- Repeat until no swaps needed

Quiz: show array  
after first pass

# Bubble Sort – Algorithm

- Sorting Array A
  - Assume indices 0 to length-1

```
while swaps happen
    index = 1
    while index < length
        if A[index-1] > A[index]
            swap A[index-1] and A[index]
        index = index + 1
```

---

# Exercise 2.1 Bubble Sort

- Complete the table to show the successive passes of a bubble sort



# Demo

sortingDemo.py

---

# Bubble Sort – Properties

- Stable
- Inefficient
- $O(N^2)$ 
  - Double length – time increases 4-fold

<http://www.sorting-algorithms.com/bubble-sort>

---

# Insertion Sort – Insight

not yet ordered

17	31	52	19	41	34	76
----	----	----	----	----	----	----

ordered

11	28	92
----	----	----


- Imagine part of the array is ordered
- Insert the next item into the correct place

not yet ordered

17	31	52	19	41	34
----	----	----	----	----	----

ordered

11	28	76	92
----	----	----	----



---

# Insertion Sort – Description

not yet ordered

17	31	52	19	41	34	76
----	----	----	----	----	----	----

ordered

11	28	92
----	----	----

- Start with one entry – ordered
  - Take each entry in turn
  - Insert into ordered part by swapping with lower values
  - Stop when all entries inserted
-

# Exercises 2.2 & 2.4

- Using playing cards (e.g. 6) to show the sort algorithms
    - bubble sort
    - insertion sort
-



# Insertion Sort – Algorithm

- Sorting Array A
  - Assume indices 0 to length-1

```
index = 1
while index < length of array
    ix = index
    while A[ix] < A[ix-1] and ix > 0
        swap A[ix] and A[ix-1]
        ix = ix - 1
    index = index + 1
```

- A[0:index] ordered
- Same values

Inner loop: insert into ordered list

# Quicksort – Insight

- How could we share out sorting between two people?
    - Choose a value  $V$
    - Give first person all values  $< V$
    - Give second person all values  $> V$
  - When there is only a single entry – it is sorted
-

# Quicksort Example

17	31	52	19	41	34	76	11	28
----	----	----	----	----	----	----	----	----

28	17	19	11	31	34	76	52	41
----	----	----	----	----	----	----	----	----

all < 31

all  $\geq$  31

11	17	19	28
----	----	----	----

41	34	52	76
----	----	----	----

19	28
----	----

34	41
----	----

---

# Quicksort Description

- Choose a pivot value
  - Partition the array
    - Values less than the pivot to the left
    - The pivot
    - Values greater than the pivot to the right
  - Repeat process on each partition
  - Stop when the partition has no more than one value
-

# Properties

- Insertion sort
  - $O(N^2)$  – same as bubble sort
  - Stable

<http://www.sorting-algorithms.com/insertion-sort>

- Quicksort
  - More efficient:  $O(N \log N)$
  - Not stable

<http://www.sorting-algorithms.com/quick-sort>

---

# Exercises 2.6

- Using playing cards (e.g. 6) to show the sort algorithms
    - quicksort
-

# Quick Sort – Recursive Implementation

```
def quickSort(A):  
    alen = len(A)  
    if alen < 2: return A  
  
    p = A[0]  
    A1 = []  
    A2 = []  
  
    for i in range(1, alen):  
        if A[i] < p:  
            A1.append(A[i])  
        else:  
            A2.append(A[i])  
  
    return quickSort(A1) + [p] + quickSort(A2)
```

Quicksort and Mergesort can be described using recursion: later topic.

# Summary

- Need for algorithms
  - Difference between
    - $O(\log N)$  and  $O(N)$  – searching
    - $O(N \log N)$  and  $O(N^2)$  – sorting
  - Divide and conqueror principle
-