COM1009
Introduction to Algorithms and Data Structures

Topic 1: Algorithms

Essential reading:

Chapter 1 and Chapter 2, Sections 2.1-2.2
(skip the problems, exercises, and pseudocode conventions)

Aims of this lecture

- to set the scene for the analysis of algorithms
- to define correctness of algorithms and to demonstrate how to show that an algorithm is correct
- to show how the running time of an algorithm can be analysed

We'll illustrate these ideas by defining and analysing InsertionSort (a simple sorting algorithm)

Algorithms

- An algorithm is a well-defined **computational procedure** that takes some **input** and produces some **output**.
  - It is a tool for solving a well-specified **computational problem**.
- Example: the **sorting** problem
  - **Input**: a sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.
  - **Output**: a permutation (reordering) $\langle a_1', a_2', \dots, a_n' \rangle$ of the input sequence such that $a_1' \leq a_2' \leq \cdots \leq a_n'$.
- A valid input is called an **instance** of the problem.

Example

Given the instance $\langle 31, 41, 59, 26, 41, 58 \rangle$ the sorted output should be $\langle 26, 31, 41, 41, 58, 59 \rangle$. In this example, n = 6.

▶How we describe algorithms

We use an abstract language, **pseudocode**, for two reasons:

1. See that algorithms exist independent from any particular programming language

2. Focus on **ideas** rather than syntax issues, error-handling, etc.

"*If you wish to implement any of the algorithms, you should find the translation of our pseudocode into your favourite programming language to be a fairly straightforward task.*
…
We attempt to present each algorithm simply and directly without allowing the idiosyncrasies of a particular programming language to obscure its essence."

What's an ideal algorithm?

- Does the job
- Does it quickly
- Affordable
- Easy to apply
- Easy to adapt
- ...

We'll be focussing on correctness and efficiency

▶ Correctness ("does the job")

- An algorithm is **correct** if for every input instance it halts with the correct output. A correct algorithm **solves** the problem.

? How do you know whether an algorithm is correct?

? Who would you rather buy from?

  - Person A: "I expect it's correct."
  - Person B: "I tested my algorithm on 3 instances and it worked."
  - Person C: "I can **prove** that my algorithm is **always** correct."

- In this module the algorithms will generally be taught with a proof of correctness

▶How to measure time? ("runs quickly")

- Computers are different (clock rate, speed of memory...)
- Computer architecture can be complex (COM1006: memory hierarchy, pipelining, multi-core...)
- Choice of programming language affects execution time
- We need a model that provides a **good level of abstraction:**
  - Gives a good idea about the time an algorithm needs
  - Allows us to compare different algorithms
  - Without us getting bogged down with details
- We'll model things using an RAM machine...

Random-access machine (RAM) model

- A generic random-access machine; instructions are executed one after another, with no concurrent operations
- Elementary operations:
  - Add, subtract, multiply, divide, remainder
  - Logical operations, shifts, comparisons
  - Data movement: variable assignments
  - Control instructions: loops, subroutine/method calls

Random-access machine (RAM) model (2)

- Common cost model: count the number of elementary operations in the RAM model.
- Assumes all elementary operations take the same time.

Runtime of Algorithm A on instance I:
The number of elementary operations in the RAM model A takes on I.

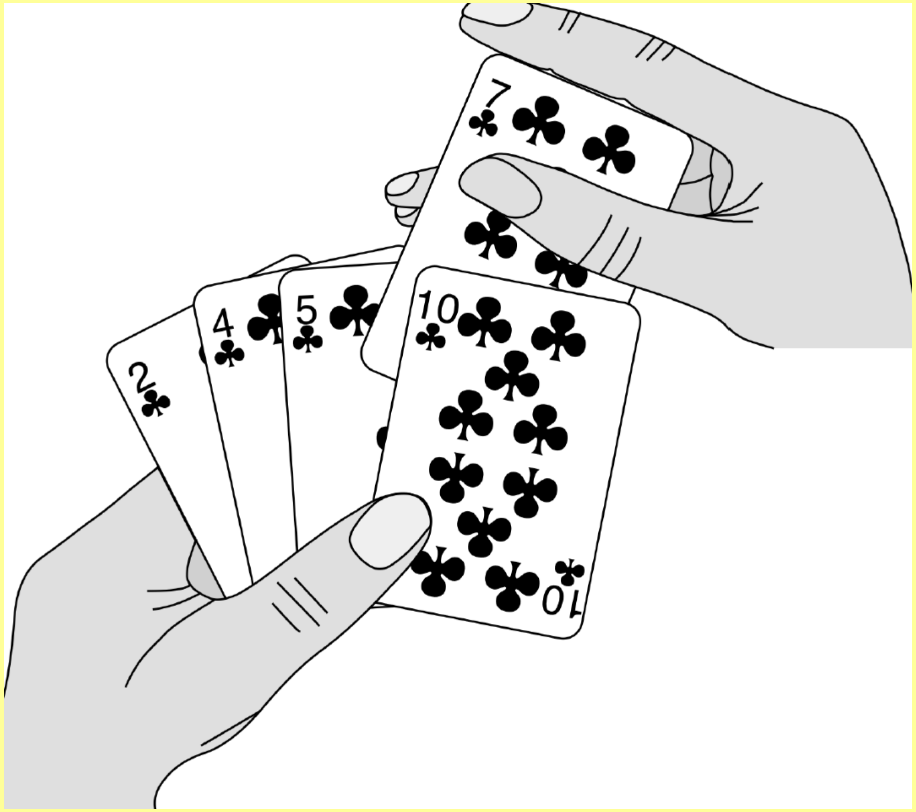
- Don't get obsessed counting operations in detail !
  - We can usually ignore various terms (you'll see why)
  - Focus on **asymptotic growth** of runtime with problem size
  - We'll meet some Greek friends to help us: $\Theta, O, \Omega, o, \omega$

Example: InsertionSort

Idea: build up a sorted sequence by inserting the next element at the right position.

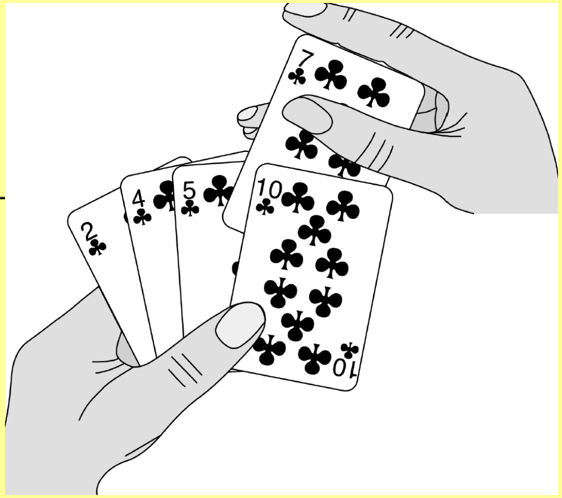
Like sorting a hand of cards!

(find details in the book, Sections 2.1 & 2.2)

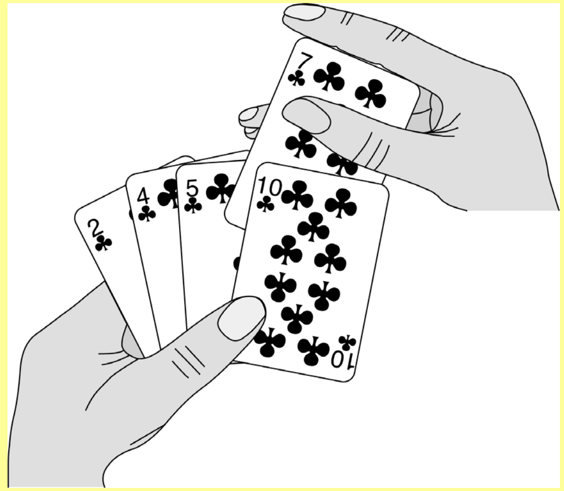


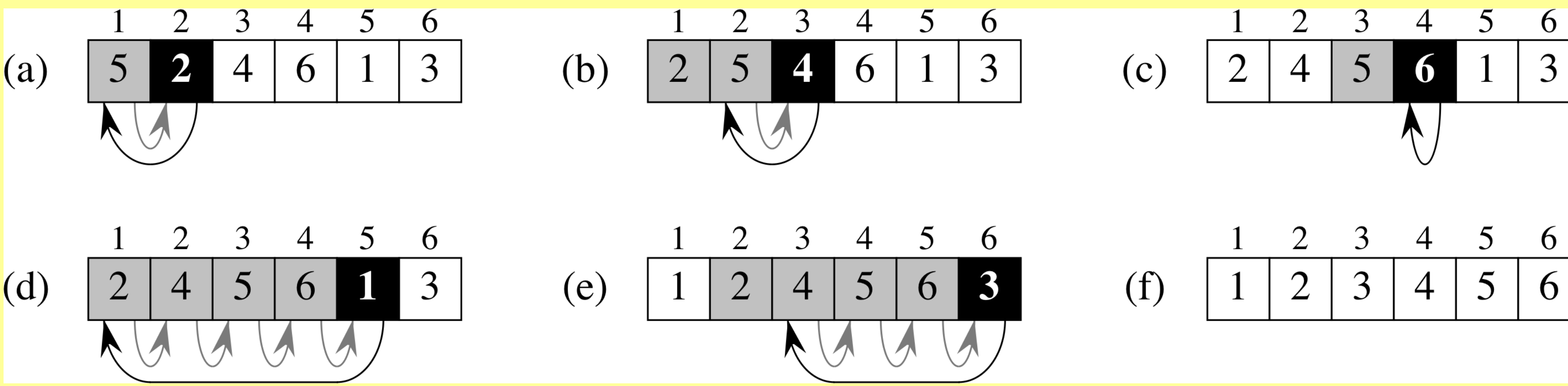
InsertionSort

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



▶Example for InsertionSort





Coming up

1. How do we know whether InsertionSort is always correct?
   - Proof by loop invariant
2. How long does InsertionSort take to run?
   - Naïve and messy approach for now to motivate a cleaner and easier way (next week).

Loop invariants

- A popular way of proving correctness of algorithms with loops.
- A **loop invariant** is a statement that is always true and that reflects the progress of the algorithm towards producing a correct output.
  - Example: "After i iterations of the loop, at least i things are nice."
  - The hard bit is finding out what is "nice" for your algorithm!
  - **Initialisation:** the loop invariant is true at initialisation.
    - Often trivial: "After 0 iterations of the loop, at least 0 things are nice."
  - **Maintenance:** if the loop invariant is true after i iterations, it is also true after $i+1$ iterations.
    - Need to prove that the loop turns i nice things into $i+1$ nice things.
  - **Termination:** when the algorithm terminates, the loop invariant tells that the algorithm is correct.
    - "When terminating, all is nice and that means the output is correct!"

Correctness of InsertionSort

- **Loop invariant:** "At the start of the j'th iteration of the for-loop in lines 1-8, the subarray A[1 .. (j-1)] consists of the elements originally in A[1 .. (j-1)], but in sorted order."

```
InsertionSort(A)
1: for j = 2 to A.length do
2:     key = A[j]
3:     // Insert A[j] into ...
4:     i = j − 1
5:     while i > 0 and A[i] > key do
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

▶ Correctness of InsertionSort (2)

- **Initialisation:** "At the start of the first iteration (j = 2) of the for-loop, the subarray A[1..1] consists of the elements originally in A[1..1], but in sorted order." (trivially true – there's only one element in the array A[1..1]))

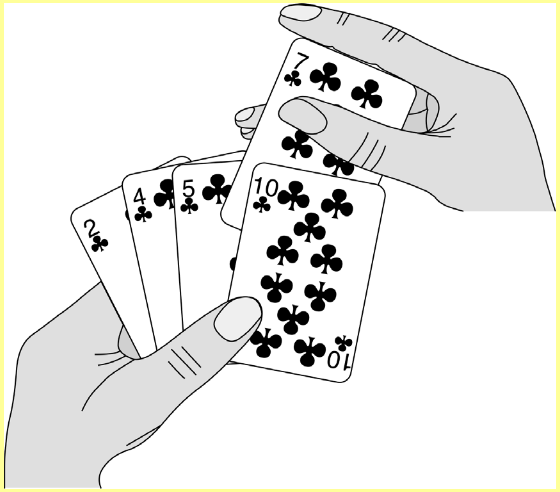
```
InsertionSort(A)
for j = 2 to A.length do
    key = A[j]
    // Insert A[j] into ...
    i = j − 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Correctness of InsertionSort (3)

- **Maintenance:** The while loop moves A[j-1], A[j-2], ... one position to the right and inserts A[j] at the correct position i+1. Then A[1..j] contains the original A[1..j], but in sorted order:

$$\underbrace{A[1] \leq A[2] \leq \cdots \leq A[i-1] \leq A[i]}_{\text{sorted before}} \underbrace{\leq A[i+1] \leq}_{\text{from while loop}} \underbrace{A[i+2] \leq \cdots \leq A[j]}_{\text{sorted before}}$$

```
InsertionSort(A)
for j = 2 to A.length do
    key = A[j]
    // Insert A[j] into ...
    i = j - 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```



Correctness of InsertionSort (4)

- **Termination:** The for loop ends when j=n+1. Then the loop invariant for j=n+1 says that the array contains the original A[1..n] in sorted order.

Since A[1..n] is the whole array, this proves that the sorting algorithm works correctly – by the time it finishes running, the array has been sorted.

InsertionSort(A)

```
for j = 2 to A.length do
    key = A[j]
    // Insert A[j] into ...
    i = j − 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

What about runtime?

```
INSERTIONSORT(A)
1: for j = 2 to A.length do
2:     key = A[j]
3:     // Insert A[j] into ...
4:     i = j - 1
5:     while i > 0 and A[i] > key do
6:         A[i + 1] = A[i]
7:         i = i - 1
8:     A[i + 1] = key
```

Define c_i = the cost of running line i once

Define t_j = the number of times the while loop is executed for that j.

How much does it cost to run each line?
How many times do we run it?
Work out the total cost.

Runtime of InsertionSort

	Cost	How often?
`for j = 2 to n do`	c_1	n
`  key = A[j]`	c_2	$n-1$
`  // comment`	0	(irrelevant)
`  i = j-1`	c_4	$n-1$
`  while i > 0 and A[i] > key do`	c_5	$t_2 + t_3 + \ldots = \sum_{j=2}^{n} t_j$
`    A[i+1] = A[i]`	c_6	$(t_2-1) + (t_3-1) + \ldots = \sum_{j=2}^{n} (t_j-1)$
`    i = i-1`	c_7	$(t_2-1) + (t_3-1) + \ldots = \sum_{j=2}^{n} (t_j-1)$
`  A[i+1] = key`	c_8	$n-1$

Remember: t_j = the number of times the while loop is executed for that j.

▶Runtime of InsertionSort

- Summary of the calculation:
  1. Assume that line i is run in time (cost) c_i.
  2. Count the number of times that line is executed.
     - Use t_j for the number of times the while loop was executed (=number of arrows for moving elements)
  3. Sum up products of costs and times.
- Result (it's messy; our Greek friends will help keep things tidy):

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best case: $t_2 = t_3 = \ldots = 1$ (input is already sorted)

	Cost	How often?
`for j = 2 to n do`	c_1	n
`  key = A[j]`	c_2	$n-1$
`  // comment`	0	(irrelevant)
`  i = j-1`	c_4	$n-1$
`  while i > 0 and A[i] > key do`	c_5	$\sum_{j=2}^{n} t_j = (n-1)$
`    A[i+1] = A[i]`	c_6	**0**
`    i = i-1`	c_7	**0**
`  A[i+1] = key`	c_8	$n-1$

Remember: t_j = the number of times the while loop is executed for that j.

Runtime of InsertionSort: Best case

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \overbrace{\sum_{j=2}^{n} t_j}^{(n-1)} + c_6 \overbrace{\cancel{\sum_{j=2}^{n}(t_j - 1)}}^{0} + c_7 \overbrace{\cancel{\sum_{j=2}^{n}(t_j - 1)}}^{0} + c_8(n-1)$$

Best case: the array is sorted, $t_j = 1$ (1x head of while loop)

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

for constants a, b derived from c_1, c_2, etc.

Best case: a **linear** function in n.

Worst case: $t_j = j$ (input is reverse-sorted)

	Cost	How often?
`for j = 2 to n do`	c_1	n
`  key = A[j]`	c_2	$n-1$
`  // comment`	0	(irrelevant)
`  i = j-1`	c_4	$n-1$
`  while i > 0 and A[i] > key do`	c_5	$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$
`    A[i+1] = A[i]`	c_6	$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1)$
`    i = i-1`	c_7	$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1)$
`  A[i+1] = key`	c_8	$n-1$

Remember: t_j = the number of times the while loop is executed for that j.

▶Runtime of InsertionSort: Worst case

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Worst case: the input is reverse-sorted, and $t_j = j$.

Useful formula (we proved this in com1002): $\displaystyle\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

So...

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

Runtime of InsertionSort: Worst case (2)

Worst case: the array is reverse sorted, $t_j = j$

Using these formulas gives

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

for constants a, b, c composed of c_1, c_2, etc.

Worst case: a **quadratic** function in n

Summary

- Correctness means that an algorithm always produces the intended output.
- Runtime describes the number of **elementary operations in an RAM machine**.
- Seen **InsertionSort** as a first example of an algorithm
  - Idea: build up sorted sequence by slotting in the next element.
  - Used a **loop invariant** to prove that the algorithm is correct.
    - A loop invariant is a statement that is always true.
    - Captures the progress towards producing a correct output at termination.
  - Analysed the **runtime** of InsertionSort.

Reading: read Chapter 1 and Chapter 2, Sections 2.1-2.2, (skip problems, exercises, and pseudocode conventions)