CS217: Data Structures & Algorithm Analysis (DSAA)

Lecture #1

➢ Getting Started

Prof. Pietro Oliveto
Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn
https://faculty.sustech.edu.cn/olivetop

Reading (that means homework!): read Chapter 1 and Chapter 2, Sections 2.1-2.2, (skip 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**
- to analyse **InsertionSort** as 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 sequence like $\langle 31, 41, 59, 26, 41, 58 \rangle$ is called an **instance** of the sorting problem.

- We expect an algorithm to **solve** any instance of the problem

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?**

Correctness

- 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 be?

  – Person A: “I designed an algorithm and I think it is correct.”

  – Person B: “I tested my algorithm on 3 instances and it worked.”

  – Person C: “I can **prove** that my algorithm is **always** correct.”

- Ideally, all algorithms should be taught with a proof of correctness!

How to measure time?

- Computers are different (clock rate, speed of memory…)
- Computer architecture can be complex
  (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

Random-access machine (RAM) model

- A generic random-access machine; instructions are executed one after another, with no concurrent operations
- Elementary operations:
  - **Arithmetic:** Add, subtract, multiply, divide, remainder
  - **Logical** operations, shifts, comparisons
  - **Data movement:** variable assignments (storing, retrieving)
  - **Control instructions:** loops, subroutine/method calls
- The RAM model assumes that each elementary operation takes the same amount of time (a *constant* independent of the problem size)
- The elementary operations are those commonly found in real computers

Runtime

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

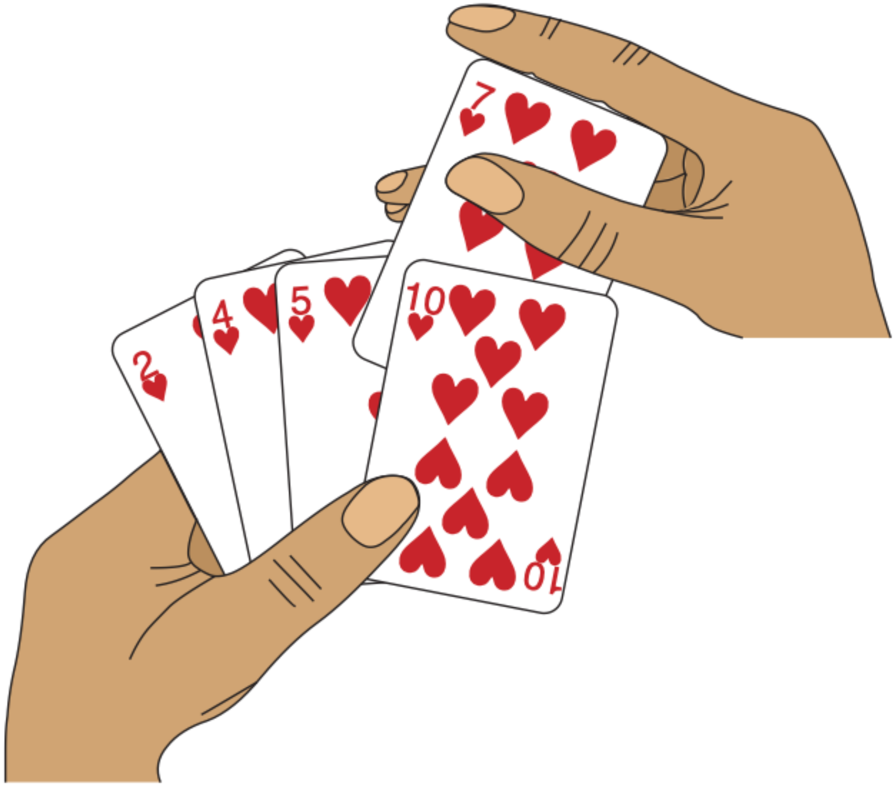
Runtime of Algorithm A on instance I:
The number of elementary operations in the RAM model A takes on I.

- But… don't get obsessed counting operations in detail
- We'll often abstract from constants (you'll see how)
- 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

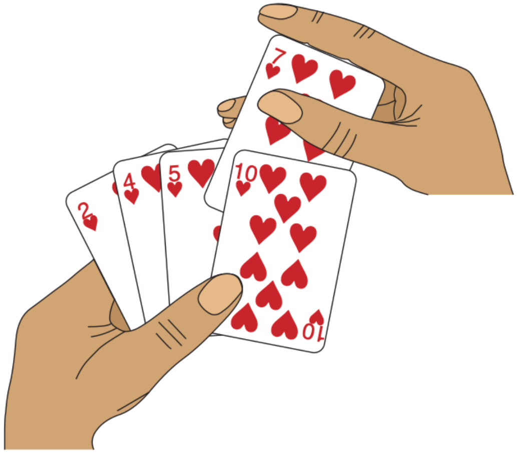
Idea: build up a sorted sequence by inserting the next element at the right position.

Like sorting a hand of cards!

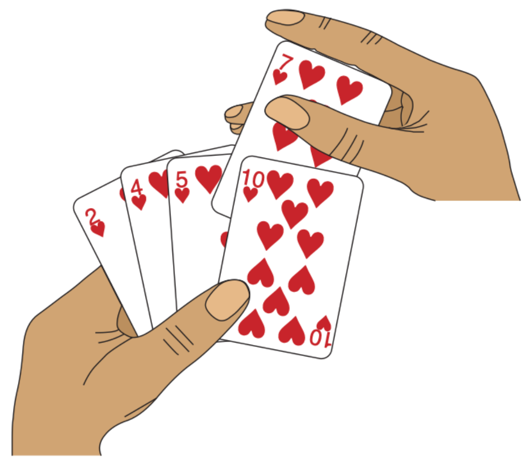


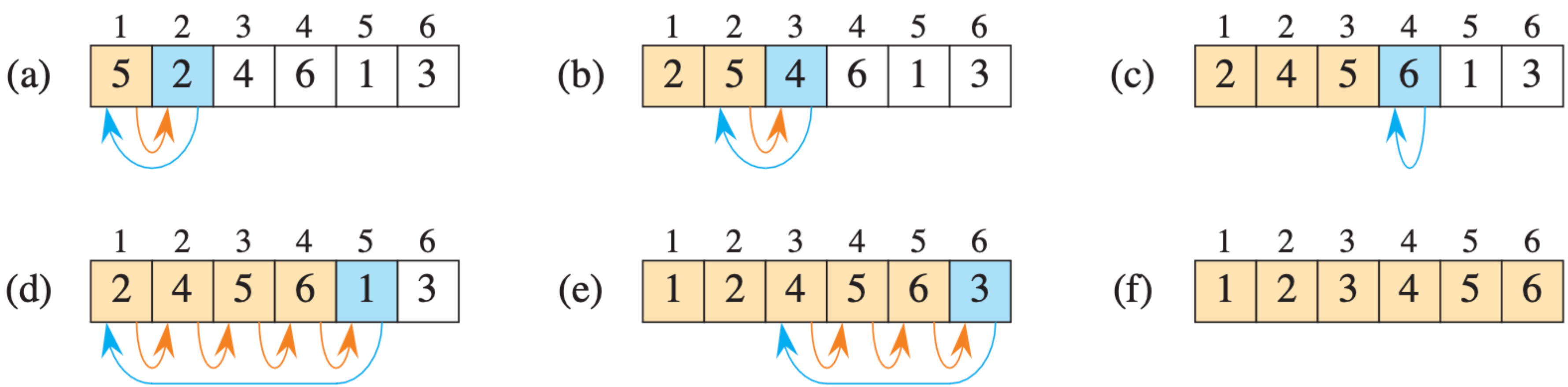
InsertionSort

```
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!”

Loop invariant: Example

INSERT-ALL-FIVES(A, n)

1: **for** $i = 1$ to n **do**
2: $\quad A[i] = 5$

- **Loop invariant:** *"At the start of each iteration of the for loop, each element of the subarray A[1..i-1] is a 5"*
- **Initialisation:** For *i=1* the empty subarray has no elements (trivial).
- **Maintenance:** Loop invariant says that at step i of the *for* loop the subarray *A[1..i-1]* contains *5*s. During the *i_th* iteration we insert a 5 in *A[i]*, so by the end of the iteration the loop invariant still holds for step *i+1*.
- **Termination:** The algorithm terminates when *i=n+1*. Then the loop invariant for *i=n+1* says that all the elements of the subarray *A[1..n]* contain *5*s, so the algorithm returns the correct output!

Correctness of InsertionSort

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

- **Initialisation:** For j=2 the subarray A[1] is the original A[1] and it is sorted (trivially).

- **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:

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

- **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!

Runtime of InsertionSort

```
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
```

Cost

Times

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

Runtime of InsertionSort

```
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
```

Cost	Times
c_1	n
c_2	$n-1$
c_4	$n-1$
c_5	$t_2 + t_3 + \ldots = \sum_{j=2}^{n} t_j$
c_6	$(t_2-1) + (t_3-1) + \ldots = \sum_{j=2}^{n} (t_j-1)$
c_7	$(t_2-1) + (t_3-1) + \ldots = \sum_{j=2}^{n} (t_j-1)$
c_8	$n-1$

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

Runtime of InsertionSort

- How to analyse the runtime of InsertionSort (in a naïve way):
  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
  3. Sum up products of costs and times.
- Result (it's messy; our Greek friends will help keep things tidy):

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

Runtime of InsertionSort: Best case

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

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 composed of c_1, c_2, etc.

Note: $an + b$ is a **linear function** in n.

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 array is reverse sorted, $t_j = j$

The following formula is very helpful:

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

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

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

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

Note: a **quadratic function** in n

Summary

- **Correctness** means that an algorithm always produces the intended output for any input.
- Runtime describes the number of **elementary operations in a 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.