Lecture 1: Introduction
(CLRS 1, 2.1-2.2)

1   Introduction

- Class is about *designing* and *analyzing algorithms*

    - *Algorithm*: A well-defined procedure that takes an input and computes some output.

        * Not a program (but often specified like it): An algorithm can often be implemented in several ways.

    - *Design*: Methods/ideas for developing (efficient) algorithms.

    - *Analysis*: Abstract/mathematical comparison of algorithms (without actually implementing them). Think of analysis as a measure of the quality of your algorithm and use it to justify design decisions when you write programs.

- In this class we do all these:

    - come up with solutions for a problem

    - prove that it is correct

    - analyze its running time

- Hopefully the class will show that **algorithms matter!**

2   Algorithm example: Insertion-sort

The problem of sorting is defined as:

- Input: n integers in array $A[1..n]$

- Output: A sorted in increasing order

Insertion-sort works similarly with sorting a deck of cards. The algorithm is described below in a "pseudo-code" that we will use to describe algorithms.

```
INSERTION-SORT(A)

    For j = 2 to n DO
        key = A[j]
        i = j − 1
        WHILE i > 0 and A[i] > key DO
            A[i + 1] = A[i]
            i = i − 1
        OD
        A[i + 1] = key
    OD
```

How does it work? Example:

5   2   4   6   1   3   j=2   i=1   key=2

5   5   4   6   1   3   i=0

2   5 | 4   6   1   3

2   5   4   6   1   3   j=3   i=2   key=4

2   5   5   6   1   3   i=1

2   4   5 | 6   1   3

2   4   5   6   1   3   j=4   i=3   key=6

2   4   5   6 | 1   3

2   4   5   6   1   3   j=5   i=4   key=1

2   4   5   6   6   3   i=3

2   4   5   5   6   3   i=2

2   4   4   5   6   3   i=1

2   2   4   5   6   3   i=0

1   2   4   5   6 | 3

1   2   4   5   6   3   j=6   i=5   key=3

1   2   4   5   6   6   i=4

1   2   4   5   5   6   i=3

1   2   4   4   5   6   i=2

1   2   3   4   5   6 |

2.1 Correctness

We prove correctness by finding and proving certain conditions that hold at some point in the algorithm *for any input.* These are called *invariants.*

- Prove the following loop invariant: "A[1..j-1] is sorted" holds at the beginning of each iteration of FOR-loop.

    - When j=n+1 (*Termination*) we have the correct output.

- The loop invariant can be proved by induction (try it!).

- Note: In many cases it is harder to find the right invariant(s) than to prove it (them).

2.2 Analysis

- We want to predict the resource use of the algorithm.

- We can be interested in different resources (like main memory, bandwidth), but normally *running time.*

- To analyze running time without actually implementing the algorithm we need a mathematical model of a computer:

  > **Random-access machine (RAM) model**:
  > - Instructions executed sequentially one at a time
  > - All instructions take unit time:
  >   * Load/Store
  >   * Arithmetics (e.g. $+, -, *, /$)
  >   * Logic (e.g. $>$)
  > - Main memory is infinite

- > **The running time of an algorithm is the number of instructions it executes in the RAM model of computation.**

- RAM model not completely realistic, e.g.

    - main memory not infinite (even though we often imagine it is when we program)
    - not all memory accesses take same time (cache, main memory, disk)
    - not all arithmetic operations take same time (e.g. multiplications expensive)
    - instruction pipelining
    - other processes

- But RAM model often enough to give relatively realistic results (if we don't cheat too much).

- Running time of insertion-sort depends on many things

- How sorted the input is
- How big the input is, etc etc

- Normally we are interested in running time as a function of *input size*

   - in insertion-sort: n.

- **Best-case running time:** The shortest running time for any input of size n. The algorithm will never be faster than this.

- **Worst-case running time:** The longest running time for *any* input of size n. The algorithm will never be slower than this.

- **Average-case running time:** Be careful: average over what? Must assume an input distribution.

- Let us analyze insertion-sort by assuming that line i in the program use c RAM instructions.

   - How many times are each line of the program executed?
   - Let t_j be the number of times line 4 (the WHILE statement) is executed in the j'th iteration.

	cost	times
FOR $j = 2$ to n DO	c	n
$key = A[j]$	c	$n - 1$
$i = j - 1$	c	$n - 1$
WHILE $i > 0$ and $A[i] > key$ DO	c	$\sum_{j=2}^{n} t_j$
$A[i + 1] = A[i]$	c	$\sum_{j=2}^{n}(t_j - 1)$
$i = i - 1$	c	$\sum_{j=2}^{n}(t_j - 1)$
OD		
$A[i + 1] = key$	c	$n - 1$
OD		

- Running time: (depends on t_j) $T(n) = cn + 2c(n - 1) + c\sum_{j=2}^{n} t_j + 2c\sum_{j=2}^{n}(t_j - 1) + c(n - 1)$

   - **Best case:** $t_j = 1$ (already sorted)
$$\begin{aligned} T(n) &= cn + 2c(n - 1) + c(n - 1) + c(n - 1) \\ &= 5cn - 4c \\ &= k_1 n - k_2 \end{aligned}$$
   Linear function of n

   - **Worst case:** $t_j = j$ (sorted in decreasing order)
$$\begin{aligned} T(n) &= cn + 2c(n - 1) + c\sum_{j=2}^{n} j + 2c\sum_{j=2}^{n}(j - 1) + c(n - 1) \\ &= cn + 2c(n - 1) + c(\tfrac{n(n+1)}{2} - 1) + 2c(\tfrac{(n-1)n}{2}) + c(n - 1) \\ &= \dots \\ &= k_3 n^2 + k_4 n - k_5 \end{aligned}$$

Quadratic function of n

Note: We used $\boxed{\sum_{j=1}^{n} j = \frac{n(n+1)}{2}}$ (Next week!)

– **Average case**: We assume n numbers chosen randomly $\Rightarrow t_j = j/2$

$$T(n) = k_6 n^2 + k_7 n + k_8$$

Still **Quadratic function of** n

- Note:

  – We will normally be interested in worst-case running time.

    * For some algorithms, worst-case occur fairly often.
    * Average case often as bad as worst case (but not always!).

  – We will only consider order of growth of running time:

    * We already ignored cost of each statement and used the constants c.
    * We even ignored c and used k_i.
    * We simply said that best case was *linear in* n and worst/average case *quadratic in* n.

  $\Rightarrow O$-notation (best case $O(n)$, worst/average case $O(n^2)$) (next lecture!)