

Algorithms and Data Structures

Content and Basic Notions

Algorithms and Data Structures

A continuation of **Introduction to Algorithms and Data Structures (INF2 - IADS)**.

Mostly same techniques, more advanced applications.

Divide-and-Conquer, Greedy, Dynamic Programming,

More emphasis on “Algorithms” rather than “Data Structures”.

More **theorem proving**.

The course content

Review of algorithm design basics, including *time and space complexity*, and *asymptotic notation*.

The *divide-and-conquer* paradigm:

Sorting, matrix multiplication, Fourier transform.

Upper and lower bound proofs.

Solving recurrence relations.

The course content (cont)

The *greedy* paradigm:

Minimum spanning trees.

Network flows.

Linear programming.

The *dynamic programming* paradigm:

Matrix-chain multiplication and other examples.

What is an algorithm?

A set of instructions for solving a problem or performing a computation.

Origin of the name: Latinisation of the name given by Persian scholar Muhammad ibn Mūsā al-Khwārizmī.



Example: Sorting

Given a sequence of numbers, put them in increasing order.

6	2	19	4	10	14	17	10	21	24
---	---	----	---	----	----	----	----	----	----



continues the same way...

Is $2 < 6$? Is $19 < 6$? Is $4 < 19$?

Is $4 < 6$?

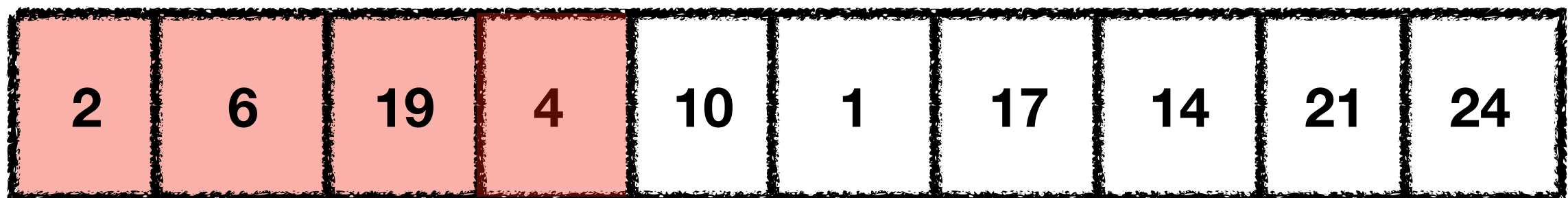
Is $4 < 2$?

Describing algorithms: Pseudocode

```

INSERTION_SORT (A)
1.  FOR  $j \leftarrow 2$  TO  $\text{length}[A]$ 
2.      DO  $\text{key} \leftarrow A[j]$ 
3.      {Put  $A[j]$  into the sorted sequence  $A[1 \dots j-1]$ }
4.       $i \leftarrow j-1$ 
5.      WHILE  $i > 0$  and  $A[i] > \text{key}$ 
6.          DO  $A[i+1] \leftarrow A[i]$ 
7.           $i \leftarrow i-1$ 
8.       $A[i+1] \leftarrow \text{key}$ 
    
```

The algorithm maintains **a sorted array** in each iteration (each time the for loop is executed).



$j=4$

$i=3$

$A[3] = 19 \neq \text{key} = 4$

key=4

$j=5$

$A[4] = 10 \neq 4$

still in the while loop

What should we expect from algorithms?

Correctness: It computes the desired output.

Termination: Eventually terminates (or with high probability).

Efficiency:

The algorithm runs *fast* and/or uses *limited memory*.

The algorithm produces a “good enough” outcome.

Correctness

How do we prove that our algorithm is always correct?

Proof techniques (induction, proof-by-contradiction etc).

For those of you that took INF2-IADS: You did this a lot there!

Running Time / Time Complexity

Different computers have different speeds.

Random Access Machine (RAM) model.

Instructions:

Arithmetic (add, subtract, multiply, etc).

Data movement (load, store, copy, etc).

Control (branch, subroutine call, return, etc).

Each instruction is carried out in constant time.

We can count the number of instructions, or the number of steps.

Example: Running Time of InsertionSort

INSERTION_SORT (A)

```

1.  FOR  $j \leftarrow 2$  TO length[ $A$ ]  $n$  times
2.      DO  $\text{key} \leftarrow A[j]$   $n-1$  times
3.          {Put  $A[j]$  into the sorted sequence  $A[1 \dots j-1]$ }
4.           $i \leftarrow j-1$   $n-1$  times
5.          WHILE  $i > 0$  and  $A[i] > \text{key}$   $\sum_{j=2}^n t_j$  times
6.              DO  $A[i+1] \leftarrow A[i]$ 
7.               $i \leftarrow i-1$   $\sum_{j=2}^n (t_j - 1)$  times
8.           $A[i+1] \leftarrow \text{key}$   $n-1$  times
    
```

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

Best case? **Sorted array, $t_j = 1$**

Bounded by some **cn** for some constant c

Worst case? **Reverse sorted array, $t_j = j$**

Bounded by some **cn^2** for some constant c

Memory Usage / Space Complexity

Each memory cell can hold one element of the input.

Total memory usage = Memory used to hold the input + extra memory used by the algorithm (**auxiliary memory**).

Q: What is the total and the auxiliary memory usage of InsertionSort?

Worst vs Best vs Average Case

Convention: When we say “the running time of Algorithm A”, we mean the **worst-case running time**, over all possible inputs to the algorithm.

We can also measure the **best-case running time**, over all possible inputs to the problem.

In between: **average-case running time**.

Running time of the algorithm on inputs which are chosen at random from some distribution.

The appropriate distribution depends on the application (usually the uniform distribution - all inputs equally likely).

Example: Average Running Time of InsertionSort

INSERTION_SORT (A)

```

1.  FOR  $j \leftarrow 2$  TO  $\text{length}[A]$  n times
2.      DO  $\text{key} \leftarrow A[j]$  n-1 times
3.          {Put  $A[j]$  into the sorted sequence  $A[1 \dots j-1]$ }
4.           $i \leftarrow j-1$  n-1 times
5.          WHILE  $i > 0$  and  $A[i] > \text{key}$   $\sum_{j=2}^n t_j$  times
6.              DO  $A[i+1] \leftarrow A[i]$ 
7.               $i \leftarrow i-1$   $\sum_{j=2}^n (t_j - 1)$  times
8.               $A[i+1] \leftarrow \text{key}$  n-1 times
    
```

for loops, the tests are executed one more time than the loop body

Select an input uniformly at random from all possible sequences with n numbers.

On average, key will be smaller than half of the elements in $A[1, \dots, j]$.

The while loop will look “halfway” through the sorted subarray $A[1, \dots, j]$.

This means that $t_j = \frac{j}{2}$ Bounded by some cn^2 for some constant c

Asymptotic Notation

When n becomes large, it makes less of a difference if an algorithm takes $2n$ or $3n$ steps to finish.

In particular, $3\log n$ steps are fewer than $2n$ steps.

We would like to avoid having to calculate the precise constants.

We use asymptotic notation (next lecture).