Algorithms and Complexity

What is the course about?

How to solve problems in an algorithmic way.

Three examples:

1. Sorting of numbers ⟵ Can be solved easily. But it can be solved more efficiently.

2. Shortest paths in graphs ⟵ A bit tricky to solve efficiently.

3. Partitioning of a set of numbers. ⟵ Can not be solved efficiently.

There is a set of natural questions concerning algorithms:

How do we measure efficiency?

How do we find algorithms?

How do we describe algorithms?
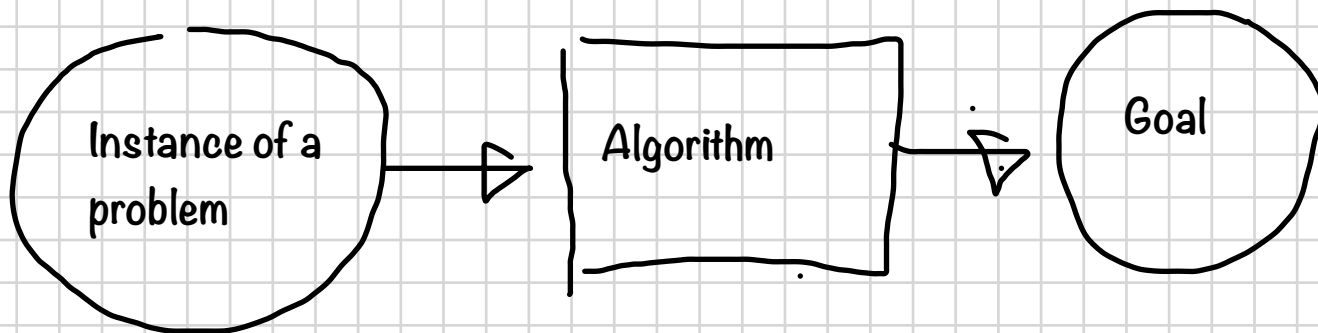
How do we know if an algorithm works?

How do we know if a problem can be solved efficiently?

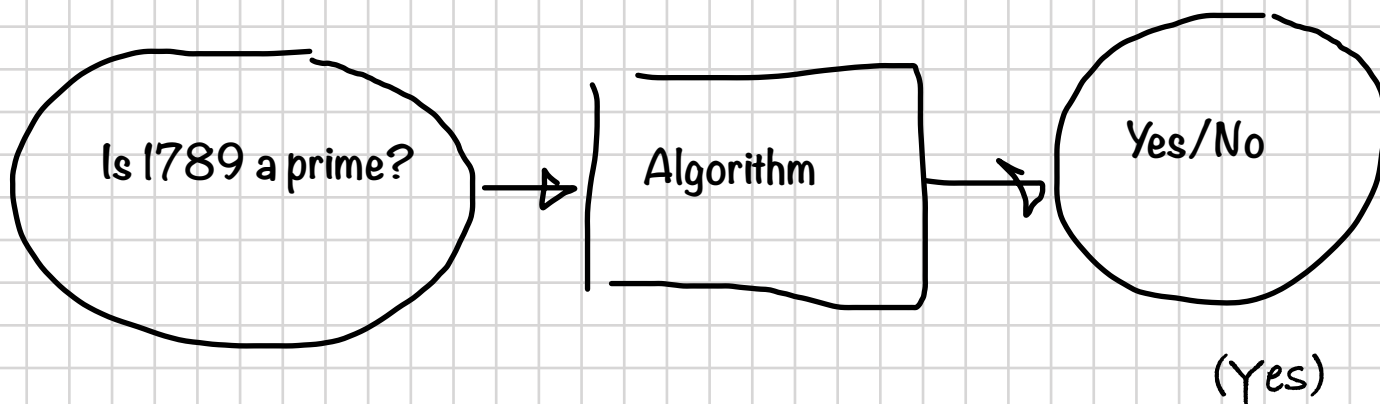How do we know if a problem can be solved algorithmically at all?

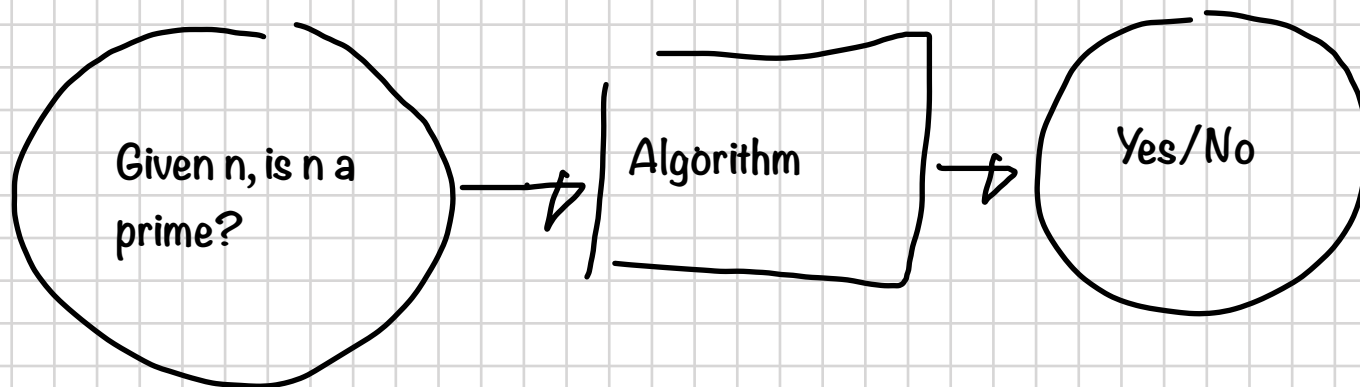By the way, what is a problem?

We will try to provide some answers.

A general picture:

```
┌─────────────┐        ┌───────────┐         ╭────────╮
│ Instance of │ ────▷  │ Algorithm │ ────▷   │  Goal  │
│ a problem   │        │           │         ╰────────╯
└─────────────┘        └───────────┘
```

Ex.

```
╭───────────────╮      ┌───────────┐         ╭──────────╮
│ Is 1789 a     │ ──▷  │ Algorithm │ ──▷     │  Yes/No  │
│ prime?        │      │           │         ╰──────────╯
╰───────────────╯                              (Yes)
```

But an algorithm is usually not designed to solve just one instance.
Normally it solves an infinite numbers of instances:

Given n, is n a prime? → Algorithm → Yes/No

So what is a problem?

A problem consist of Input and Goal

Ex:

PRIME NUMBER

Input: An integer n
Goal: Is n a prime number?

SORTING

Input: A list $L = \{ l_1, l_2, ... \}$ of numbers
Goal: A sorting $L' = \{ l'_1, l'_2, ... \}$ such that $l'_1, l'_2, ...$ is increasing

SHORTEST PATH

Input: A graph G. Two nodes a,b
Goal: A path from a to b with as few edges as possible

How do we desribe an algorithm?

An answer would be to just give a program code.
But that can be inconvenient.

Take for instance the most famous sorting algorithm:
Insertion-Sort.

Informal description:

We start with al list A[1], A[2], ...
Take A[1] and set $L_1 = A[1]$.
Take A[2] and sort into L_1. This gives us a new list L_2.
Take A[3] and sort into L_2. This gives us a new list L_3.
And so on ...

We describe the algorithm with so called pseudocode.

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

How do we know that Insertion-Sort works correctly?

Think like this: At step k we know that

A[1], A[2], ... , A[k] is a part of the start list and
A[1], A[2], ... , A[k] is sorted.

When the algorithm stops A[1], A[2], ... , A[k] must be the sorted list.

Usually, the problem is to decide if a loop works
correctly. Sometimes we can define a loop-invariant I
that is always true for each step in the lopp

Suppose the loop has the form

While B do
    (...)   Something happens here

When the loop ends we have I and not B. If we have choosen I wisely, we will have reached our goal.

What about efficiency? How do we measure it?

Basic idea:
We measure the number of steps needed as a function of the size of the input. This type of measure is called time-complexity.

Insertion-Sort: If we have n elements we need $O(n^2)$ steps.

Problem: How do we define size of input?

Basic idea: We use a string to represent the input. We measure the size of the string.

In practice, we use something that is a convenient approximation of the size of the string.

Ex: If theinput is n elements in al list we normally use n as a measure of the input size.

Ex: If the input is just an integer n we use $\log_2 n$ as the size of the input.

But there is a problem with finding good representations.

A bit more about complexity

Time-complexity is not the only measure of complexity.

We can measure the size of the memory space the algorithm needs to complete the computation. This measure is called space-complexity.

Time-complexity more important but space-complexity is also important.

Later we will show that

high space-complexity $\Rightarrow$ high time-complexity

The opposite is not necessarily true.

Unit cost and bit cost

An interesting question is what counts as a step in a computation.

Unit cost: Each "algebraic operation" counts as one step
Bit cost: Each bit operation counts as a step

Ex:
We want to compute $521 * 394$. What is the cost?

Unit cost: Just one operation! The cost is 1.
Bit cost: 521 has 9 bits. 394 has 8 bits. The cost is $9*8 = 72$
(The answer is 205274.)

Unit cost is easier to work with and we
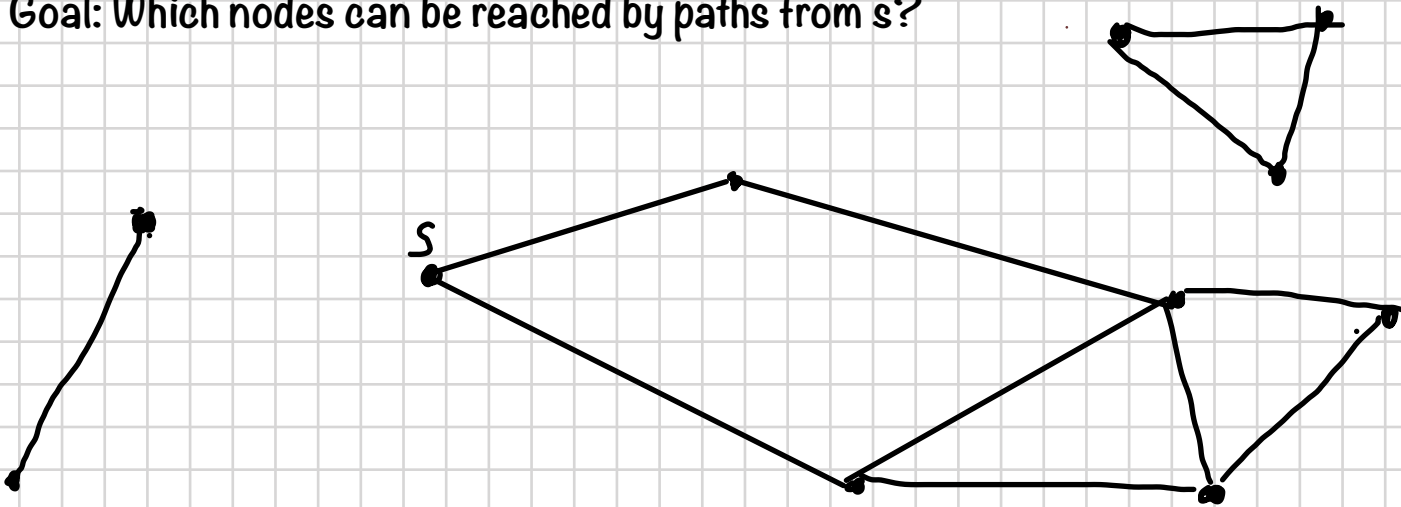normally use it.

Ex: How do we represent a graph?

The natural measure of the input size could be
$|V|$ or $|E|$ or $|V|+|E|$

Usually, the complexity is given as a function of $|V|$
and $|E|$.

We will study a special problem in graph theory

Input: A graph G and a node s.
Goal: Which nodes can be reached by paths from s?

A simple algorithm

$R \leftarrow \{s\}$

    While there is (u,v) such that $u \in R$ and $v \in R$

        Add v to R

End while

When the algorithm stops, R is the set of nodes reachable from s.

But there is an uncertainty. How shall (u,v) be choosen?

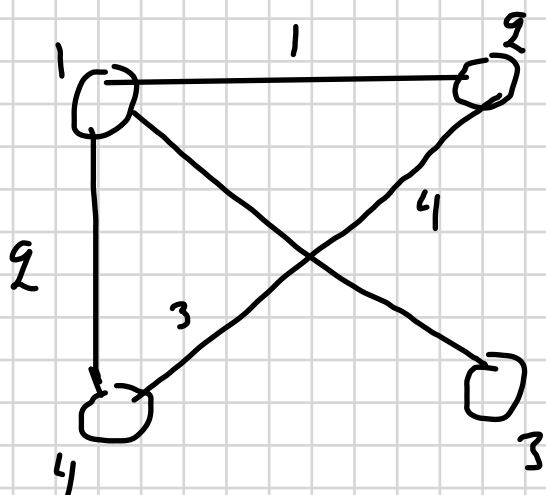## Data structures for graphs

There are basically 3 ways to describe a graph in a form suitable for computation.

 ✗ Adjacency matrix

 ✗ Incidence matrix

 ✗ Adjacency lists

Ex:

adjacency matrix $A(i,j) = 0/1$

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

$A(i,j)=1 \quad \Longleftrightarrow \quad$ There is an edge (i,j)

Incidence matrix

$$I = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$I(i,j) = 0/1$

$I(i,j) = 1 \iff$ nod i is on edge j

Adjacency lists

L1: 2,3,4
L2: 1, 4
L3: 1
L4: 1,2

j is in L i $\iff$ There is an edge (i,j)

We now present an algorithm which is amore detailed solution to our graph problem.

When the algorithm stops, T is a tree containing all nodes that can be reached from s.

The complexity is $O(|V| + |E|)$

For all $v \in V$
   set vis(s) = 0
End for
Set vis(s) = 1
Set count = 0
Set T = Ø

Set L[0] = {s}
While L[i] is not empty
   Set L[i+1] = Ø

   For each u ∈ L[i]
      For each edge (u,v)
        If vis(v) = 0
          Set vis(v) = 1
          Add (u,v) to T
          Add v to L[i+1]
        End if
      End for
   End for
   Set i = i+1
End while

What is an efficient algorithm? We can start by asking what an inefficient algorithm is. If the size of the input is n and the algorithm works in time $O(2^n)$ we say that the algorithm has exponential complexity. Such an algorithm is clearly very inefficient.

In several situations where we have a problem with an input of size n, we have a set of possible solutions of size exponential in n. Just one (or a few) of these possible solutions are real solutions. If we just test all possible solutions to see if any of them are real solutions we get an algorithm withexponential complexity. In order to get a better algorithm we must find a way of "zooming in" on the real solutions.

It seems natural to say that an efficient algorithm is an algorithm which is not exponential. But we don't do this. Instead we use the following definition:

We say that an algorithm has polynomial time-complexity if there is an integer k such that the algorithm, if started with input of size n, runs in $O(n^k)$ steps.

Efficient algorithms:
The standard definition is to say that an algorithm is efficient if and only if it has polynomial time-complexity.

An outline of the course

We have two parts:

Algorithms

Complexity

Some theory

Lots of applications

Lots of theory

Applications
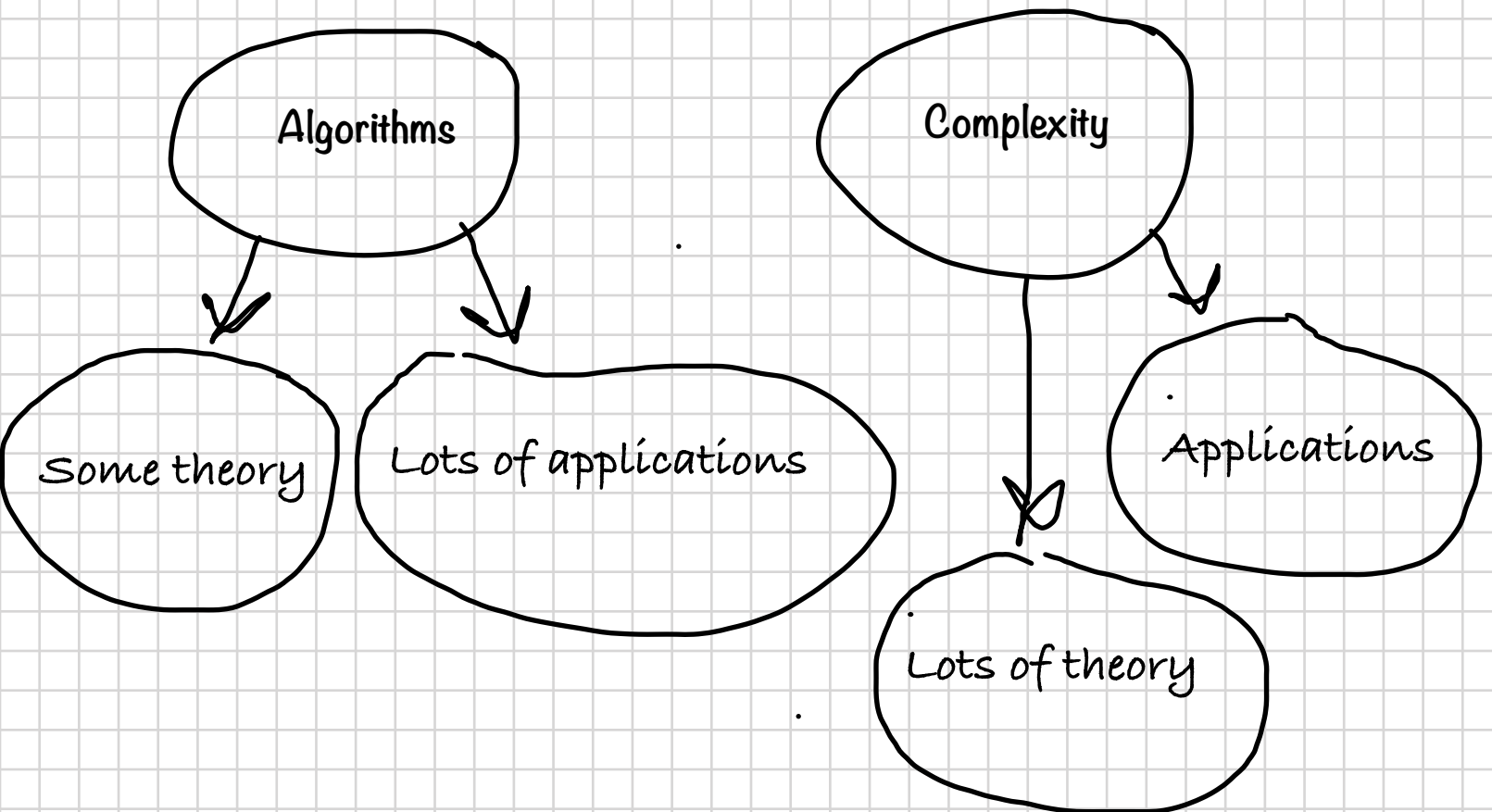
Algorithms
We describe several types of algorithms and ways to construct algorithms.

Complexity

Not all problems can be solved efficiently.
Some problems can not be solved at all!
Some of the things we will do in this part are:

Study Turing Machines and formally define computing

Study uncomputable problems.

Study NP-problems which (probably) can not be solved efficiently.

Study so called approximation algorithms.

We describe one more search algorithm for graphs: Depth-First Search

Set = Ø

For all v ∈ V

    Set vis(v) = 0

End for

DFS(s)

DFS(u):

    Set vis(u) = 1

    Add u to R

    For each v such that v is adjacent to u

        If vis(v) = 0

            DFS(s)

        End if

    End for

The complexity is $O(|V| + |E|)$.
The algorithm is defined
recursively. A non-recursive
definition is:

Set R = Ø

Let S' be a stack and set S = {s}

For all v ∈ V

    Set vis(v) = 0

End for

Set vis(s) = 1

While S ≠ Ø

    Take the top node u from S

    If vis(s) = 0

        Add u to R

        For each v adjacent to u

            Add v to the top of S

        End for

    End if

End while