Teoria degli Algoritmi

Corso di Laurea Magistrale in Matematica Applicata
a.a. 2020-21

Gabriele Tolomei

Dipartimento di Informatica
Sapienza Università di Roma
tolomei@di.uniroma1.it

Lecture 1: Introduction to Computability

Table of Contents

1. Course Information
2. Background
3. Types of Computational Problems
4. Goals
5. Summary

Table of Contents

1. Course Information
2. Background
3. Types of Computational Problems
4. Goals
5. Summary

Useful Information

Class Schedule

- **Wednesday:** 9 a.m. – 11 a.m.
- **Thursday:** 11 a.m. – 1 p.m.

Useful Information

Class Schedule

- **Wednesday:** 9 a.m. – 11 a.m.
- **Thursday:** 11 a.m. – 1 p.m.

Office Hours

Arranged online by appointment

Useful Information

Class Schedule

- **Wednesday:** 9 a.m. – 11 a.m.
- **Thursday:** 11 a.m. – 1 p.m.

Office Hours

Arranged online by appointment

Contacts

- **email:** tolomei@di.uniroma1.it
- **website:** https://github.com/gtolomei/theory-of-algorithms
- **moodle:** https://elearning.uniroma1.it/course/view.php?id=13101

SAPIENZA UNIVERSITÀ DI ROMA

Course Outline

- The course is divided into **2 main parts**:
  - **Foundational** (about 25%)
  - **Functional** (about 75%)

Course Outline

- The course is divided into **2 main parts**:
  - **Foundational** (about 25%)
  - **Functional** (about 75%)
- The **Foundational** part is devoted to provide you with (some of) the background in the theory of computation:
  - ***Computability:*** *Are all the problems we may think of computable?*

Course Outline

- The course is divided into **2 main parts**:
  - **Foundational** (about 25%)
  - **Functional** (about 75%)
- The **Foundational** part is devoted to provide you with (some of) the background in the theory of computation:
  - ***Computability:*** *Are all the problems we may think of computable?*
  - ***Computational Complexity:*** *How much resources do we need to compute a problem?*

Course Outline

- The course is divided into **2 main parts**:
  - **Foundational** (about 25%)
  - **Functional** (about 75%)
- The **Foundational** part is devoted to provide you with (some of) the background in the theory of computation:
  - ***Computability:*** *Are all the problems we may think of computable?*
  - ***Computational Complexity:*** *How much resources do we need to compute a problem?*
- The **Functional** part allows you to fully grasp the implications of the theoretical results discussed above in the real-world, with a focus on **machine learning** and **artificial intelligence**:
  - algorithms that *learn* algorithms

Exam

Still trying to figure it out...

Exam

Still trying to figure it out...

But I needed to hear a few things from you first!

Table of Contents

1. Course Information
2. Background
3. Types of Computational Problems
4. Goals
5. Summary

Theory of Computation in One Slide

- The subject of interest to many mathematicians around the 1930's
  - Alonzo Church, Kurt Gödel, Stephen Kleene, Emil Post, and Alan Turing (just to name a few) have all made remarkable contributions to the field

Theory of Computation in One Slide

- The subject of interest to many mathematicians around the 1930's
  - Alonzo Church, Kurt Gödel, Stephen Kleene, Emil Post, and Alan Turing (just to name a few) have all made remarkable contributions to the field
- They were all interested in understanding and defining what it means (for a problem) to be **computable**

Theory of Computation in One Slide

- The subject of interest to many mathematicians around the 1930's
  - Alonzo Church, Kurt Gödel, Stephen Kleene, Emil Post, and Alan Turing (just to name a few) have all made remarkable contributions to the field
- They were all interested in understanding and defining what it means (for a problem) to be **computable**
  - How do we **define** a computable problem?

Theory of Computation in One Slide

- The subject of interest to many mathematicians around the 1930's
  - Alonzo Church, Kurt Gödel, Stephen Kleene, Emil Post, and Alan Turing (just to name a few) have all made remarkable contributions to the field
- They were all interested in understanding and defining what it means (for a problem) to be **computable**
  - How do we **define** a computable problem?
  - Are **all** the problems computable?

Theory of Computation in One Slide

- The subject of interest to many mathematicians around the 1930's
  - Alonzo Church, Kurt Gödel, Stephen Kleene, Emil Post, and Alan Turing (just to name a few) have all made remarkable contributions to the field
- They were all interested in understanding and defining what it means (for a problem) to be **computable**
  - How do we **define** a computable problem?
  - Are **all** the problems computable?
  - How **effectively** can we compute a problem?

Theory of Computation in One Slide

- The subject of interest to many mathematicians around the 1930's
  - Alonzo Church, Kurt Gödel, Stephen Kleene, Emil Post, and Alan Turing (just to name a few) have all made remarkable contributions to the field
- They were all interested in understanding and defining what it means (for a problem) to be **computable**
  - How do we **define** a computable problem?
  - Are **all** the problems computable?
  - How **effectively** can we compute a problem?
- In the following, we will try to answer all the questions above

What is a Computational Problem?

- Informally, any mathematical problem that can be solved by a **computing device** through an **effective method** (more on this later)

What is a Computational Problem?

- Informally, any mathematical problem that can be solved by a **computing device** through an **effective method** (more on this later)
- A **computing device** can be a human or any other physical machine

What is a Computational Problem?

- Informally, any mathematical problem that can be solved by a **computing device** through an **effective method** (more on this later)
- A **computing device** can be a human or any other physical machine

Example (Integer Factorization)

Given $n \in \mathbb{Z}$ such that $n > 1$, find all non-trivial prime factors of n

What is a Computational Problem?

Definition

A computational problem P is a **relation** between a possibly infinite set of **instances** (i.e., input) $\mathcal{X}$ and their set of **solutions** (i.e., output) $\mathcal{Y}$:

$$P \subseteq \mathcal{X} \times \mathcal{Y}$$

What is a Computational Problem?

Definition

A computational problem P is a **relation** between a possibly infinite set of **instances** (i.e., input) $\mathcal{X}$ and their set of **solutions** (i.e., output) $\mathcal{Y}$:

$$P \subseteq \mathcal{X} \times \mathcal{Y}$$

Note

Let $x \in \mathcal{X}$ be an **instance**, then *any* $y \in \mathcal{Y}$ such that $(x, y) \in P$ is a **solution** to instance x of the problem P

What is a Computational Problem?

Definition

A computational problem P is a **relation** between a possibly infinite set of **instances** (i.e., input) $\mathcal{X}$ and their set of **solutions** (i.e., output) $\mathcal{Y}$:

$$P \subseteq \mathcal{X} \times \mathcal{Y}$$

Note

Let $x \in \mathcal{X}$ be an **instance**, then *any* $y \in \mathcal{Y}$ such that $(x, y) \in P$ is a **solution** to instance x of the problem P

Example (Integer Factorization)

$x = 42$ is an **instance**;
$y_1 = 2$; $y_2 = 3$; $y_3 = 7$ are *all* valid **solutions**, that is:

$$(x, y_1) \in P \wedge (x, y_2) \in P \wedge (x, y_3) \in P$$

Input/Output Representation

- In the integer factorization example, both instances and solutions are a subset of the set of integer numbers, i.e., $\mathcal{X} = \mathcal{Y} = \{n \in \mathbb{Z} \mid n > 1\}$

Input/Output Representation

- In the integer factorization example, both instances and solutions are a subset of the set of integer numbers, i.e., $\mathcal{X} = \mathcal{Y} = \{n \in \mathbb{Z} \mid n > 1\}$
- In practice, though, many problems operate on different mathematical objects: i.e., not only integers but also graphs, texts, images, etc.

Input/Output Representation

- In the integer factorization example, both instances and solutions are a subset of the set of integer numbers, i.e., $\mathcal{X} = \mathcal{Y} = \{n \in \mathbb{Z} \mid n > 1\}$
- In practice, though, many problems operate on different mathematical objects: i.e., not only integers but also graphs, texts, images, etc.
- It is therefore customary to **encode** both instances of and solutions to problems into some **finite representations**

Input/Output Representation

- In the integer factorization example, both instances and solutions are a subset of the set of integer numbers, i.e., $\mathcal{X} = \mathcal{Y} = \{n \in \mathbb{Z} \mid n > 1\}$
- In practice, though, many problems operate on different mathematical objects: i.e., not only integers but also graphs, texts, images, etc.
- It is therefore customary to **encode** both instances of and solutions to problems into some **finite representations**
- A common input/output encoding is given by the set of finite-length **binary strings**

Binary String Encoding

Definition: Kleene Closure

Let $\Sigma = \{0, 1\}$ be an *alphabet* of 2 symbols, and let Σ^n be the set of all the strings over Σ whose length is $n \in \mathbb{N} \cup \{0\}$. We define $\Sigma^* = \{0, 1\}^*$ the **Kleene closure** of Σ as the infinite set of all the finite-length strings over Σ:

$$\Sigma^* = \bigcup_{n \in \mathbb{N} \cup \{0\}} \Sigma^n$$

Binary String Encoding

Definition: Kleene Closure

Let $\Sigma = \{0, 1\}$ be an *alphabet* of 2 symbols, and let Σ^n be the set of all the strings over Σ whose length is $n \in \mathbb{N} \cup \{0\}$. We define $\Sigma^* = \{0, 1\}^*$ the **Kleene closure** of Σ as the infinite set of all the finite-length strings over Σ:

$$\Sigma^* = \bigcup_{n \in \mathbb{N} \cup \{0\}} \Sigma^n$$

Definition: Representation Scheme

Let $\mathcal{O}$ be a set of *objects*. A representation scheme is a *one-to-one* function $e : \mathcal{O} \longmapsto \Sigma^*$

Binary String Encoding

- It is easy to show that such a binary representation scheme exists for the set of natural numbers ($\mathbb{N}$) as well as that of integers ($\mathbb{Z}$)

Binary String Encoding

- It is easy to show that such a binary representation scheme exists for the set of natural numbers ($\mathbb{N}$) as well as that of integers ($\mathbb{Z}$)
- With a slight modification to the alphabet Σ, it is also possible to map every rational number ($\mathbb{Q}$) to a finite binary string

Binary String Encoding

- It is easy to show that such a binary representation scheme exists for the set of natural numbers ($\mathbb{N}$) as well as that of integers ($\mathbb{Z}$)
- With a slight modification to the alphabet Σ, it is also possible to map every rational number ($\mathbb{Q}$) to a finite binary string
- Any real number ($x \in \mathbb{R}$), instead, can be approximated by a rational number ($q = a/b$) or using floating-point scheme

Binary String Encoding

- It is easy to show that such a binary representation scheme exists for the set of natural numbers ($\mathbb{N}$) as well as that of integers ($\mathbb{Z}$)
- With a slight modification to the alphabet Σ, it is also possible to map every rational number ($\mathbb{Q}$) to a finite binary string
- Any real number ($x \in \mathbb{R}$), instead, can be approximated by a rational number ($q = a/b$) or using floating-point scheme
- Similarly, objects like vectors (i.e., composition of numbers), graphs, texts, etc. can all be described by binary strings

Binary String Encoding

- It is easy to show that such a binary representation scheme exists for the set of natural numbers ($\mathbb{N}$) as well as that of integers ($\mathbb{Z}$)
- With a slight modification to the alphabet Σ, it is also possible to map every rational number ($\mathbb{Q}$) to a finite binary string
- Any real number ($x \in \mathbb{R}$), instead, can be approximated by a rational number ($q = a/b$) or using floating-point scheme
- Similarly, objects like vectors (i.e., composition of numbers), graphs, texts, etc. can all be described by binary strings

Note

Binary encoding is just a useful convention; other representation schemes are indeed possible and the results we will show are independent of the specific representation

What Does it Mean to Solve a Computational Problem?

- At its core, it means executing a procedure that transforms an input (e.g., a string of bits) into some output (e.g., a string of bits)

What Does it Mean to Solve a Computational Problem?

- At its core, it means executing a procedure that transforms an input (e.g., a string of bits) into some output (e.g., a string of bits)
- In other words, it means computing a **function** that solves the problem!

What Does it Mean to Solve a Computational Problem?

- At its core, it means executing a procedure that transforms an input (e.g., a string of bits) into some output (e.g., a string of bits)
- In other words, it means computing a **function** that solves the problem!
- This input-to-output transformation can be done using a modern computing device, a human following instructions, or any other means

What Does it Mean to Solve a Computational Problem?

- At its core, it means executing a procedure that transforms an input (e.g., a string of bits) into some output (e.g., a string of bits)
- In other words, it means computing a **function** that solves the problem!
- This input-to-output transformation can be done using a modern computing device, a human following instructions, or any other means

Algorithm (effective method)

An **algorithm** consists of a finite number of exact, finite instructions that always transforms a given problem instance (**input**) into a correct solution for it (**output**) after a finite number of steps.

What Does it Mean to Solve a Computational Problem?

- At its core, it means executing a procedure that transforms an input (e.g., a string of bits) into some output (e.g., a string of bits)
- In other words, it means computing a **function** that solves the problem!
- This input-to-output transformation can be done using a modern computing device, a human following instructions, or any other means

Algorithm (effective method)

An **algorithm** consists of a finite number of exact, finite instructions that always transforms a given problem instance (**input**) into a correct solution for it (**output**) after a finite number of steps.

- It turns out that an **algorithm** must be:
  - **Correct:** it must *always* produce the correct solution
  - **Finite:** it must *always* terminate, eventually

Algorithm: Formal Definition

Definition: Algorithm

Let $P \subseteq \mathcal{X} \times \mathcal{Y}$ be a problem. An algorithm $\mathcal{A}$ **solves** P iff $\mathcal{A}$ computes a function $f_{\mathcal{A}}$, such that:

- $\text{domain}(f_{\mathcal{A}}) \supseteq \mathcal{X}$
- $\forall x \in \mathcal{X} : (x, f_{\mathcal{A}}(x)) \in P$

Algorithm: Formal Definition

Definition: Algorithm

Let $P \subseteq \mathcal{X} \times \mathcal{Y}$ be a problem. An algorithm $\mathcal{A}$ **solves** P iff $\mathcal{A}$ computes a function $f_{\mathcal{A}}$, such that:

- $\text{domain}(f_{\mathcal{A}}) \supseteq \mathcal{X}$
- $\forall x \in \mathcal{X} : (x, f_{\mathcal{A}}(x)) \in P$

- The first condition says that the algorithm must be defined *at least* on all the possible instances of the problem it aims to solve
- The second condition ensures the algorithm is correct (and finite)

Algorithm: Formal Definition

Definition: Algorithm

Let $P \subseteq \mathcal{X} \times \mathcal{Y}$ be a problem. An algorithm $\mathcal{A}$ **solves** P iff $\mathcal{A}$ computes a function $f_{\mathcal{A}}$, such that:

- $\text{domain}(f_{\mathcal{A}}) \supseteq \mathcal{X}$
- $\forall x \in \mathcal{X} : (x, f_{\mathcal{A}}(x)) \in P$

- The first condition says that the algorithm must be defined *at least* on all the possible instances of the problem it aims to solve
- The second condition ensures the algorithm is correct (and finite)

Note

We assumed that an algorithm takes an abstract instance as input and returns an abstract solution as output. In fact, the real input and output are finite encodings (e.g., binary strings) of these abstract objects

Function vs. Algorithm

- Always separate between **specifications** and **implementations** or, equivalently, between **functions** and **algorithms**

Function vs. Algorithm

- Always separate between **specifications** and **implementations** or, equivalently, between **functions** and **algorithms**
- More formally, $\mathcal{A} \neq f_{\mathcal{A}}$, namely an algorithm is **not** the function it computes!

Function vs. Algorithm

- Always separate between **specifications** and **implementations** or, equivalently, between **functions** and **algorithms**
- More formally, $\mathcal{A} \neq f_{\mathcal{A}}$, namely an algorithm is **not** the function it computes!
- The same function can be computed by several different algorithms (i.e., the same specification can be implemented is different ways)

Function vs. Algorithm

Example

Consider the function $mult : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ that maps a pair of natural numbers (n_1, n_2) to their product $n_1 \cdot n_2$
The following two algorithms (`mult1` and `mult2`) computes exactly the same function $mult$, defined above

Function vs. Algorithm

Example

Consider the function *mult* : $\mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ that maps a pair of natural numbers (n_1, n_2) to their product $n_1 \cdot n_2$
The following two algorithms (`mult1` and `mult2`) computes exactly the same function *mult*, defined above

```
def mult1(x,y):
    res = 0
    while y>0:
        res += x
        y   -= 1
    return res
```

Function vs. Algorithm

Example

Consider the function $mult : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ that maps a pair of natural numbers (n_1, n_2) to their product $n_1 \cdot n_2$

The following two algorithms (`mult1` and `mult2`) computes exactly the same function *mult*, defined above

```python
def mult1(x,y):
    res = 0
    while y>0:
        res += x
        y   -= 1
    return res
```

```python
def mult2(x,y):
    a = str(x) # represent x as string in decimal notation
    b = str(y) # represent y as string in decimal notation
    res = 0
    for i in range(len(a)):
        for j in range(len(b)):
            res += int(a[len(a)-i])*int(b[len(b)-
             ↪ j])*(10**(i+j))
    return res
```

Table of Contents

1. Course Information
2. Background
3. Types of Computational Problems
4. Goals
5. Summary

Examples: Decision Problems

Example (Primality Testing)

- Given (a representation of) an integer $z > 1$, determine whether z is prime or not
- Using the binary encoding scheme above, this corresponds to compute a function $f : \Sigma^* \mapsto \Sigma$

Examples: Decision Problems

Example (Primality Testing)

- Given (a representation of) an integer $z > 1$, determine whether z is prime or not
- Using the binary encoding scheme above, this corresponds to compute a function $f : \Sigma^* \mapsto \Sigma$

Note

This is a special case of a computational problem where the goal is to compute a **boolean function**, whose output is a single bit $\{0, 1\}$.
Since this corresponds to answering a YES/NO question, this task is also known as a **decision problem**

Decision Problems and Formal Languages

- It turns out that there is a nice connection between the theory of computation and formal linguistics developed by Noam Chomsky

Decision Problems and Formal Languages

- It turns out that there is a nice connection between the theory of computation and formal linguistics developed by Noam Chomsky
- Given any boolean function $f : \Sigma^* \mapsto \Sigma$ and $x \in \Sigma^*$, the task of computing $f(x)$ is equivalent to determine if $x \in L$, where:

$$L = \{x \mid f(x) = 1\}$$

Decision Problems and Formal Languages

- It turns out that there is a nice connection between the theory of computation and formal linguistics developed by Noam Chomsky
- Given any boolean function $f : \Sigma^* \mapsto \Sigma$ and $x \in \Sigma^*$, the task of computing $f(x)$ is equivalent to determine if $x \in L$, where:

$$L = \{x \mid f(x) = 1\}$$

- L is known as the **language** corresponding to the function f

Decision Problems and Formal Languages

- It turns out that there is a nice connection between the theory of computation and formal linguistics developed by Noam Chomsky
- Given any boolean function $f : \Sigma^* \mapsto \Sigma$ and $x \in \Sigma^*$, the task of computing $f(x)$ is equivalent to determine if $x \in L$, where:

$$L = \{x \mid f(x) = 1\}$$

- L is known as the **language** corresponding to the function f
- Hence, this is also referred to as **deciding a language**

Examples: Search Problems

Example (Integer Factorization)

- Given (a representation of) an integer $z > 1$, compute its prime factors, i.e., the list of primes $p_1 \leq \ldots \leq p_k$ such that $z = p_1 \cdot \ldots \cdot p_k$
- Using the binary encoding scheme above, this corresponds to compute a function $f : \Sigma^* \mapsto \Sigma^*$

Examples: Search Problems

Example (Integer Factorization)

- Given (a representation of) an integer $z > 1$, compute its prime factors, i.e., the list of primes $p_1 \leq \ldots \leq p_k$ such that $z = p_1 \cdot \ldots \cdot p_k$
- Using the binary encoding scheme above, this corresponds to compute a function $f : \Sigma^* \mapsto \Sigma^*$

Note

This is known as a **search problem**, as it aims to find a solution with certain properties, providing that such a solution exists

SAPIENZA
UNIVERSITÀ DI ROMA

Examples: Optimization Problems

Example (Shortest Path)

- Given (a representation of) a graph $G = (V, E)$ and two nodes $s, t \in V$, find the **shortest path** from s to t
- Using the binary encoding scheme above, this corresponds to compute a function $f : \Sigma^* \mapsto \Sigma^*$

Examples: Optimization Problems

Example (Shortest Path)

- Given (a representation of) a graph $G = (V, E)$ and two nodes $s, t \in V$, find the **shortest path** from s to t
- Using the binary encoding scheme above, this corresponds to compute a function $f : \Sigma^* \mapsto \Sigma^*$

Note

This is an **optimization problem**, asking to find the "best possible" solution among the set of all possible solutions to a search problem

Decision vs. Search vs. Optimization Problems

- There are standard techniques for transforming search problems and optimization problems into decision problems

Decision vs. Search vs. Optimization Problems

- There are standard techniques for transforming search problems and optimization problems into decision problems
- Most of the results in the theory of computation are based on the decision version of a problem

Decision vs. Search vs. Optimization Problems

- There are standard techniques for transforming search problems and optimization problems into decision problems
- Most of the results in the theory of computation are based on the decision version of a problem

Note

Intuitively, the goal is to change any instance x of a search or an optimization problem into another instance x' of the corresponding decision problem

Decision vs. Search vs. Optimization Problems

Example

- **search problem:** "Given input x compute $f(x)$ (if it exists)"

Decision vs. Search vs. Optimization Problems

Example

- **search problem:** "Given input x compute $f(x)$ (if it exists)"

$$\Downarrow$$

- **decision problem:** "Given input (x, y) tests if $y = f(x)$ or not"

Decision vs. Search vs. Optimization Problems

Example

- **search problem:** "Given input x compute $f(x)$ (if it exists)"

$$\Downarrow$$

- **decision problem:** "Given input (x, y) tests if $y = f(x)$ or not"

Example

- **optimization problem:** "Given input x compute $f(x)$, such that $f(x)$ is minimum/maximum (if it exists)"

Decision vs. Search vs. Optimization Problems

Example

- **search problem:** "Given input x compute $f(x)$ (if it exists)"

$$\Downarrow$$

- **decision problem:** "Given input (x, y) tests if $y = f(x)$ or not"

Example

- **optimization problem:** "Given input x compute $f(x)$, such that $f(x)$ is minimum/maximum (if it exists)"

$$\Downarrow$$

- **decision problem:** "Given input (x, y, k) tests if $y = f(x)$ and $y \leq k$ (minimization) or $y \geq k$ (maximization)"

Table of Contents

1. Course Information
2. Background
3. Types of Computational Problems
4. Goals
5. Summary

Goals

Premise

We know that for every function f, there can be several possible algorithms that computes it. The following questions arise:

Goals

Premise

We know that for every function f, there can be several possible algorithms that computes it. The following questions arise:

- Is there any function f for which there is no algorithm that computes it? In other words, is there any function f that is **not computable**?

Goals

Premise

We know that for every function f, there can be several possible algorithms that computes it. The following questions arise:

- Is there any function f for which there is no algorithm that computes it? In other words, is there any function f that is **not computable**?
- If there is an algorithm for computing f, what is the best one (according to some measure of performance)?

Goals

Premise

We know that for every function f, there can be several possible algorithms that computes it. The following questions arise:

- Is there any function f for which there is no algorithm that computes it? In other words, is there any function f that is **not computable**?
- If there is an algorithm for computing f, what is the best one (according to some measure of performance)?
- Can f be "practically uncomputable", i.e., every algorithm that computes it requires a prohibitively large amount of resources?

Goals

Premise

We know that for every function f, there can be several possible algorithms that computes it. The following questions arise:

- Is there any function f for which there is no algorithm that computes it? In other words, is there any function f that is **not computable**?
- If there is an algorithm for computing f, what is the best one (according to some measure of performance)?
- Can f be "practically uncomputable", i.e., every algorithm that computes it requires a prohibitively large amount of resources?
- Can we show equivalence between different functions f and f' (therefore problems), meaning that either they are both "easy" or both "hard" to compute?

Table of Contents

1 Course Information

2 Background

3 Types of Computational Problems

4 Goals

5 Summary

Summary

- We have formalized the definition of **computational problem**

Summary

- We have formalized the definition of **computational problem**
- We have categorized problems into **3 classes**:
  - ***decision*** problems
  - ***search*** problems
  - ***optimization*** problems

Summary

- We have formalized the definition of **computational problem**
- We have categorized problems into **3 classes**:
  - ***decision*** problems
  - ***search*** problems
  - ***optimization*** problems
- We have defined the notion of **algorithm** as an **effective procedure** for computing a **function**

Summary

- We have formalized the definition of **computational problem**
- We have categorized problems into **3 classes**:
  - ***decision*** problems
  - ***search*** problems
  - ***optimization*** problems
- We have defined the notion of **algorithm** as an **effective procedure** for computing a **function**
- We have opened up a number of questions on the **computability** of functions