Introducton to Computer Science

Karteek Sreenivasaiah

1st Jan 2021!

A Brief History

Computation

“Algorithms” have been known since ages:

- Earliest known: A division algorithm older then 2000 BC from Mesopotamia.
- Egyptian algorithms for arithmetic ~1550 BC.
- Sieve of Eratosthenes, and Euclid’s gcd algorithm ~300BC.
- Aryabhata’s algorithm for Chinese remainder theorem 6th century AD.
- References to Aryabhata’s algorithm by Bhaskara I and Brahmagupta in the 1st century.

Computation

“Algorithms” have been known since ages:

- Earliest known: A division algorithm older then 2000 BC from Mesopotamia.
- Egyptian algorithms for arithmetic ~1550 BC.
- Sieve of Eratosthenes, and Euclid’s gcd algorithm ~300BC.
- Aryabhata’s algorithm for Chinese remainder theorem 6th century AD.
- References to Aryabhata’s algorithm by Bhaskara I and Brahmagupta in the 1st century.

The notion of *computation* is not new at all!

What is Computation?

What is Computation?

What is an ‘algorithm’

Computation

Some common answers:

- An algorithm is a sequence of steps.
- It's a procedure to compute the value of some function.

Computation

Some common answers:

- An algorithm is a sequence of steps.
- It's a procedure to compute the value of some function.

Well... oook.

So... what exactly is a *step*?
And isn't *"procedure"* just another word for algorithm anyway?

Computabililty Theory

Computability Theory

The main goals of Computability Theory were to answer:

- What is computation?
  - How can we define *computation* mathematically?
  - What are the properties of computation?

Computability Theory

The main goals of Computability Theory were to answer:

- What is computation?
  - How can we define *computation* mathematically?
  - What are the properties of computation?
- What are the limits of computation?
  - Are there functions that we can never compute?
  - If yes, what do these functions look like?

Computability Theory

The main goals of Computability Theory were to answer:

- What is computation?
  - How can we define *computation* mathematically?
  - What are the properties of computation?
- What are the limits of computation?
  - Are there functions that we can never compute?
  - If yes, what do these functions look like?

The answer to these questions started modern computer science.

Course CS2030

The course CS2040 is a course in Mathematics. We begin with:

- Studying a toy computation model called Finite State Automata.
- Defining the model rigorously.
- Defining computation for this model rigorously.
- Exploring the limits of this model.

Then, we will add some computational power to the above model.

Course CS2030

The course CS2040 is a course in Mathematics. We begin with:

- Studying a toy computation model called Finite State Automata.
- Defining the model rigorously.
- Defining computation for this model rigorously.
- Exploring the limits of this model.

Then, we will add some computational power to the above model.

And then... some more.

Course CS2030

Eventually we will arrive at a formal definition of computation:

- It will be independent of hardware.
- It will capture all computation as we know it

  (err... not quantum computation tho)

We will also answer the question on the limits of computation.

Computability Theory

Several mathematicians answered the question:

"What is computation?"

This resulted in several definitions!

Computability Theory

Several mathematicians answered the question:

"What is computation?"

This result in several definitions!

The breakthrough:

- In 1936, Alan Turing's defined his *"A-machine"*.
- Proved the first undecidable language (defined by David Hilbert in ~1900).

The A-machine garnered widespread acceptance and settled the problem. The "A-machine" is called the "Turing machine"

In many ways, this was the start of modern computer science.

Computability Theory

Computability theory is a study of computation with *unlimited* resources.
In Computational Complexity Theory: We want to study *efficient* computation.

Computational Complexity Theory

Computational Complexity - as the name suggests is about studying the complexity (difficulty) of computing functions (that are computable).

The main goals[1] of this area are to understand:

- What functions are *hard* to compute?
- Why are some functions hard to compute?
- How do the amount of resources used affect computation?

[1] There is a misconception that Complexity Theory is about measuring running time of algorithms. It is not!

Computational Complexity Theory

Computational Complexity - as the name suggests is about studying the complexity (difficulty) of computing functions (that are computable).

The main goals[1]of this area are to understand:

- What functions are *hard* to compute?
- Why are some functions hard to compute?
- How do the amount of resources used affect computation?

But wait! What does '*hard to compute*' even mean?
As usual, we will have to define this formally.

Let's understand it intuitively first.

---

[1]There is a misconception that Complexity Theory is about measuring running time of algorithms. It is not!

Computational Complexity Theory

Example

Consider the following two tasks:

- Multiplying two n digit numbers (integers in decimal)
- Adding two n digit numbers (integers in decimal)

Which do you think is harder to compute by hand?

Computational Complexity Theory

Example

Consider the following two tasks:

- Multiplying two n digit numbers (integers in decimal)
- Adding two n digit numbers (integers in decimal)

Which do you think is harder to compute by hand?

Multiplication? Maybe.

- But you might think that because you know the procedure taught in high-school to multiply two integers.
- What if there is some other algorithm that makes multiplication easier than addition, and we just haven't figured it out yet?

Computational Complexity Theory

While studying computation, we can do the following:

- If two functions are equally hard to compute, then create a set for them, and put both of them into it. Give the set a name.
- If two functions seem to be very different in terms of resources used to compute them, then put them into different sets.

Computational Complexity Theory

While studying computation, we can do the following:

- If two functions are equally hard to compute, then create a set for them, and put both of them into it. Give the set a name.
- If two functions seem to be very different in terms of resources used to compute them, then put them into different sets.

These "sets" are called *complexity classes*.

Computational Complexity Theory

While studying computation, we can do the following:

- If two functions are equally hard to compute, then create a set for them, and put both of them into it. Give the set a name.
- If two functions seem to be very different in terms of resources used to compute them, then put them into different sets.

These "sets" are called *complexity classes*.

A natural consequence of such a study is:

- We obtain a rich classification of functions based on their computational complexity.
- As we learn more about various functions, we develop a better understanding of the complexity classes.

Computational Complexity Theory

Example Complexity Class:

Very informal, and not at all precise definition, but bear with me:

DTIME

Let $t : \mathbb{N} \to \mathbb{R}^+$. The complexity class $\mathrm{DTIME}(t(n))$ is the set of all functions that can be computed by in time $O(t(n))$.

For example: $\mathrm{DTIME}(n^2)$

Computational Complexity Theory

Example Complexity Class:

Very informal, and not at all precise definition, but bear with me:

DTIME

Let $t : \mathbb{N} \to \mathbb{R}^+$. The complexity class $\mathrm{DTIME}(t(n))$ is the set of all functions that can be computed by in time $O(t(n))$.

For example: $\mathrm{DTIME}(n^2)$

Trivially, we have:

$$\mathrm{DTIME}(n) \subseteq \mathrm{DTIME}(n^2) \subseteq \mathrm{DTIME}(n^3) \subseteq \cdots$$

$$\text{But Is } \mathrm{DTIME}(n^2) \subsetneq \mathrm{DTIME}(n^3)?$$

Computational Complexity Theory

Example Complexity Class:

Very informal, and not at all precise definition, but bear with me:

DTIME

Let $t : \mathbb{N} \to \mathbb{R}^+$. The complexity class $\mathrm{DTIME}(t(n))$ is the set of all functions that can be computed by in time $O(t(n))$.

For example: $\mathrm{DTIME}(n^2)$

Trivially, we have:

$$\mathrm{DTIME}(n) \subseteq \mathrm{DTIME}(n^2) \subseteq \mathrm{DTIME}(n^3) \subseteq \cdots$$

But Is $\mathrm{DTIME}(n^2) \subsetneq \mathrm{DTIME}(n^3)$?

Computational Complexity Theory

Example Complexity Class:

Very informal, and not at all precise definition, but bear with me:

DTIME

Let $t : \mathbb{N} \rightarrow \mathbb{R}^+$. The complexity class $\mathrm{DTIME}(t(n))$ is the set of all functions that can be computed by in time $O(t(n))$.

For example: $\mathrm{DTIME}(n^2)$
Trivially, we have:

$$\mathrm{DTIME}(n) \subseteq \mathrm{DTIME}(n^2) \subseteq \mathrm{DTIME}(n^3) \subseteq \cdots$$

$$\text{But Is } \mathrm{DTIME}(n^2) \subsetneq \mathrm{DTIME}(n^3)?$$

Notation and jargon aside, this is simply:

Can we compute *more* functions with n^3 time than with n^2 time?

Computational Complexity Theory

The biggest, hardest, most epic

open problem

in the history of everything under the sun

~~is the Riemann Hypothesis~~

is an elegant question in Computational Complexity Theory

P vs NP

P vs NP

NP is the set of problems whose solutions are easy to verify.
P is the set of problems that are easy to solve from scratch.

P vs NP

NP is the set of problems whose solutions are easy to verify.
P is the set of problems that are easy to solve from scratch.

The holy grail question:

$$\text{Is } \mathrm{P} = \mathrm{NP}?$$

P vs NP

Here is an even more informal version:

Is correcting answer sheets easier than actually solving the exam problems you ask?

P vs NP

Here is an even more informal version:

Is correcting answer sheets easier than actually solving the exam problems you ask?

- Most computer scientists think that solving problems from scratch is harder.
- In other words, deterministic polynomial time computation is more restricted than non-deterministic polynomial time computation.
- Hence, most scientists lean towards $P \neq NP$.

Thank you!

Have a great new year!