Part 9: Limits of Computation

Contents



[document finalized]



## Limits of Computation

So far we haven't spent much thought on whether certain problems can or can't be solved by computers.

The part of computing science that deals with this question is called **Computability Theory** or **Recursion Theory**

This field of study originated with work of K. Gödel, A. Church, A. Turing, S. Kleene, and E. Post in the 1930s.

CMPUT 474 covers this topic in some depth.

Another important question is what can be computed under certain resource limitations, such as computation time or memory constraints.

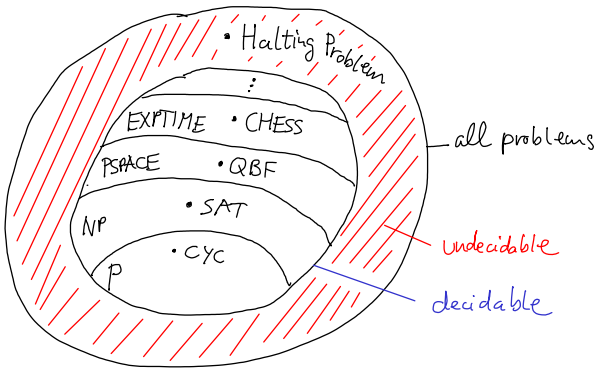
The field that studies this question is called **Computational Complexity Theory**, which originated in the 1960s and 1970s with work of J. Myhill, M. Blum, S. Cook, and R. Karp.



In this course we have seen some lower bound arguments which point in this direction. For instance, any sorting algorithm based on pairwise key comparisons has runtime $\Omega(n \log n)$. Which means that we can't sort using key comparisons if we restrict the runtime to $O(n)$.

CMPUT 304 and CMPUT 474 have more to say about this.






Complexity Classes:
(what can be computed under various resource limitations?)

P: polynomial time

NP: non-deterministic polynomial time

PSPACE: polynomial space

EXPTIME: exponential time

Problems (check whether input is in one of the following sets):

CYC: graphs with cycles

SAT: satisfiable Boolean formulas (e.g. $(x_1 \wedge \neg x_2) \vee x_3$))

QBF: true quantified Boolean formulas (e.g. $\forall x_1 \exists x_2 (x_1 \vee x_2)$)

CHESS: generalized chess positions won by white

We know

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$$

and that one of these inclusions is proper, but we don't know which.

Tough problem — open for almost 40 years:

$$\mathsf{P} = \mathsf{NP} \ ?$$

If so, we could solve thousands of important problems in polynomial time, for which we currently only have exponential time algorithms.



## Non-Computable Functions

In what follows we will first introduce word functions and then prove that certain word functions can't be computed by pseudo-code programs.

**Definition:** An **alphabet**

$$\Sigma = \{s_1, \ldots, s_n\}$$

is a finite set of **symbols** $s_i$ that can be used to construct **words**, which are sequences of symbols.

The **length** of a word is the number of symbols it consists of.

For alphabet $\Sigma$, $\Sigma^*$ denotes the set of all finite words over $\Sigma$.

The **empty word** is denoted $\varepsilon$. It has length 0, and if we print it we don't see anything, not even a space.

Examples:

$\Sigma = \{a, b, c, \ldots, z, \sqcup\}$ ($\sqcup$ is the space symbol)

Words over $\Sigma$: $apple \quad this \sqcup is \sqcup an \sqcup example$



$\Gamma = \{0, 1\}$

Words over $\Gamma$: $\varepsilon \quad 00000 \quad 10101010$

$\Gamma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$

**Definition:** Let $\perp$ denote "undefined".

For an alphabet $\Sigma$ that does not contain $\perp$, we call word function

$$f : \Sigma^* \to \Sigma^* \cup \{\perp\}$$

**computable** iff there exists a pseudo-code program $p$ that for $x \in \Sigma^*$

- stops and returns $f(x)$ if $f(x) \neq \perp$, or
- does not stop if $f(x) = \perp$.

Note, that this computability notion subsumes pseudo-code programs that work only on integers, because those, too, can be represented as $0/1$-words.



Example 1:

For $\Sigma = \{0, 1\}$

```
function double(x)
return xx                    // returns concatenated strings
```

for $x \in \{0, 1\}^*$ computes $f(x) = xx$, i.e. the input concatenated with itself:

$$f(0) = 00, \quad f(100) = 100100, \quad f(\varepsilon) = \varepsilon$$

Example 2:

```
function forever(x)
while 0 ≠ 1 do
end
```

computes the function that is undefined everywhere, i.e. $f(x) = \perp$ for all $x \in \Sigma^*$.

Are all word functions computable?

This may come as a surprise, but the answer is "no" and the proof of this fact is actually quite simple, once we understand how words and programs can be enumerated.

Lecture 36 Consider alphabet $\Sigma = \{0, 1\}$. Words in $\Sigma^*$ can be listed in **lexicographical ordering** like so:

$$\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$$

i.e., by non-decreasing word length. This defines an infinite sequence $(w_i)$ of words:

$$w_0 = \varepsilon,\ w_1 = 0, w_2 = 1, \ldots$$

This enumeration scheme works for any alphabet.

Thus, because pseudo-code programs are just words over an alphabet that contains

$$\sqcup\ a\ b \ldots z\ A\ B \ldots\ Z\ 0 \ldots\ 9 = < > + - * (\ ) \ \ldots$$

we can enumerate programs

$$p_0, p_1, p_2, \ldots$$

where we only list words that are syntactically correct programs, e.g.

$$A(*$$

is certainly not a pseudo-code program — so it doesn't appear in the list — but

$$\text{function}\sqcup\text{f(x)}\sqcup\text{return}\sqcup\text{x}$$

is a program, say $p_{42}$ in our list.



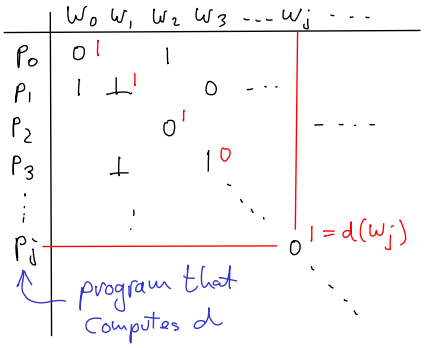
**Theorem:** There exists a non-computable function.

**Proof:** Suppose $\Sigma = \{0, 1\}$ is used for program inputs and outputs.

Consider function $d : \Sigma^* \to \Sigma^*$ with

$$d(w_i) = \begin{cases} 1, & p_i(w_i) \neq 1 \\ 0, & \text{otherwise} \end{cases}$$

Here, $p_i(x)$ denotes the value computed by program $p_i$ on input $x$.




Assume for the purpose of a contradiction, that $d$ is computable. This means that there is a program $p_j$ that computes $d$.



What happens if we use $w_j$ as input for $p_j$?

From the fact that $p_j$ computes $d$ $(*)$ and the definition of $d$ $(**)$ we get:

$$p_j(w_j) = 1 \overset{(*)}{\Leftrightarrow} d(w_j) = 1 \overset{(**)}{\Leftrightarrow} p_j(w_j) \neq 1$$

This is a contradiction. So, there can't be any program that computes $d$. □

The proof technique we used here is called "diagonalization". It goes back to G. Cantor who proved in 1874 that there are uncountable sets and as a corollary, that there are more real numbers than natural numbers.

The original diagonalization idea works as follows:

Suppose the set of infinite sequences of digits $0..9$ is countable, i.e. there is an infinite sequence $(s_i)_{i=0}^{\infty}$ that contains all such sequences: Say



$$s_0 = (0, 1, 2, 0, 9, 1, 3, \ldots)$$
$$s_1 = (2, 4, 9, 0, 1, 3, 5, \ldots)$$
$$s_2 = (7, 6, 9, 1, 5, 5, 5, \ldots)$$
$$s_3 = (1, 0, 0, 7, 0, 0, 0, \ldots)$$
$$\ldots$$

To construct an infinite sequence that is not on the list, change the main diagonal elements $(s_i)_i$, say by adding 1 and wrapping around to 0 when 10 is reached:

$$\tilde{s}_0 = (\underline{1}, 1, 2, 0, 9, 1, 3, \ldots)$$
$$\tilde{s}_1 = (2, \underline{5}, 9, 0, 1, 3, 5, \ldots)$$
$$\tilde{s}_2 = (7, 6, \underline{0}, 1, 5, 5, 5, \ldots)$$
$$\tilde{s}_3 = (1, 0, 0, \underline{8}, 0, 0, 0, \ldots)$$
$$\ldots$$

Now consider the sequence

$$\tilde{s} = ((\tilde{s}_0)_0, (\tilde{s}_1)_1, (\tilde{s}_2)_2, \ldots) = (1, 5, 0, 8, \ldots)$$

It is clear that $\tilde{s}$ is not equal to any $s_i$ because it differs from it at entry $i$.

Therefore, there is no list of infinite sequences that contains all infinite sequences.

In the context of functions and programs the diagonalization proof establishes that there are more functions than programs.

Function $d$ we used in the proof is quite artificial.

Next we will see two more natural examples which are relevant to program verification, i.e. checking whether a program is correct, for which we have to show that the program terminates and when it does, produces correct results.

We will see that the termination question corresponds to a non-computable function, which means that program verification can't be automated.

Again, suppose $\Sigma = \{0, 1\}$ and consider function $h' : \Sigma^* \to \{0, 1\}$ defined by

$$h'(w_i) = \begin{cases} 1, \text{ } p_i \text{ halts on } w_i \\ 0, \text{ otherwise} \end{cases}$$



**Theorem:** $h'$ is not computable.

**Proof:** Suppose program $p_j$ computes $h'$.

Then create a new program $p$ from $p_j$ in which each statement of the form `return` $y$ is replaced by

```
while y ≠ 0 do
end
return 0
```

Let the resulting program be $p_k$.

Then with these modifications $(*)$ and the definition of $h'$ $(**)$:

$$p_k \text{ halts on } w_k \overset{(*)}{\Leftrightarrow}$$
$$p_j \text{ on } w_k \text{ computes } 0 \overset{(**)}{\Leftrightarrow}$$
$$p_k \text{ does not halt on } w_k$$

which is a contradiction. □

Again, the proof is based on diagonalization. Here we change the "diagonal element" by a program modification which turns stopping with result 1 into non-termination.



Function $h'$ is more related to the program verification task than $d$, but still too specialized.

The next theorem will address the general halting problem:

**Theorem:** The following function $h$ is not computable: $h : \{0, 1, \#\}^* \to \{0, 1\}$ given by

$$h(w_i \# x) = \begin{cases} 1, \text{ } p_i \text{ halts on input } x \\ 0, \text{ otherwise} \end{cases}$$

where $w_i, x \in \{0, 1\}^*$, and $h(x) = \perp$ if $x$ does not contain exactly one $\#$ symbol, which acts as a delimiter.

**Proof:** We present a proof based on **reduction**:

We show: if $h$ is computable, then we can use a program that computes $h$ as a blackbox to compute $h'$.

This contradicts the previous result. Thus, $h$ is not computable.



Suppose $h$ is computable by program $p$ which w.l.o.g. does not call itself. Construct a new program $p'$ from $p$ like so:

```
function p'(x)
x ← x#x
p without ''function p(x)'' line
```

Then:

$p'(w_i) = 1 \Leftrightarrow$

$p(w_i \# w_i) = 1 \Leftrightarrow$

$p_i$ halts on $w_i \Leftrightarrow$

$h'(w_i) = 1$

which proves that $h'$ is computable — a contradiction. □

So, in general, we can't tell whether a program given by its index halts when run on a particular input. Of course, that doesn't preclude the possibility of proving termination in specific cases. It just means there is no program that can answer the halting question for arbitrary (program index, input) pairs.

An even more general result was proved by H.G. Rice in 1953:

"All non-trivial semantic program properties are undecidable."

(**semantic** means "related to the function the program computes", and a set $A$ is called **decidable** iff its characteristic function

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ 0, & x \notin A \end{cases}$$

is computable) In detail:

**Theorem:** Let $R$ be the set of computable functions and $S \subseteq R$ ("semantic property"), with $S \neq \emptyset$ and $S \neq R$ (i.e. $S$ is non-trivial).

Then the following set is undecidable:

$$\{w_i \mid p_i \text{ computes a function in } S\}$$

Examples of non-decidable program properties:

- Does $p$ compute a constant function?
- Does $p$ compute a total function (defined on every input)?
- Is $p(0) = 0$?



Final Exam:

Dec. 15 2pm in CSC B2

2 hours, closed book

Format similar to term exams

Everything is relevant: all lecture notes, assignments, seminars

Study assignment and seminar solutions

There will be problem-solving questions

Monday, Dec. 13 Office Hour @ 2pm

Good luck!

FIN