An Introduction to Computational Complexity

Stanley Yang

May 30, 2024

Contents

1 Introduction

Computational complexity theory serves as a cornerstone in understanding the limits and capabilities of computation. In this paper, we're going to explore computational complexity, starting by looking at how we measure complexity, using different math tools to make things clear and precise.

One big question we'll tackle is whether a function, like a computer program, can be computed or not. If it can, we'll try to figure out the best way to do it in terms of time and space.

We'll also look into why some functions are easier or harder to compute than others. We'll introduce different groups of problems, called complexity classes, and explain how they relate to each other.

Along the way, we'll encounter some important ideas. We'll learn about Turing Machines, which show us that some problems just can't be solved by computers. We'll also explore Gödel's Incompleteness Theorem, which tells us there are limits to what math can prove. And we'll study the Time and Space Hierarchy Theorems, which help us understand the different levels of computational power.

Understanding computational complexity is not just for theoretical mathematicians. It's also important for anyone who wants to design efficient algorithms and solve real-world problems. By learning about computational complexity, we can improve our problem-solving skills and understand the relationship between math and computer science better.

2 Backgrounds and Definitions

Our goal is to mathematically capture the concept of computation. We will address the following questions: what is computation? What is the best way to model computation? How can we reason about whether something can be efficiently computed or not? What are the resources that are worth measuring with regards to computational efficiency?

A useful mathematical analogy that represents the idea of computation is function. A function has

some input and will produce some output. Computation will work in the similar way. The underlying mystery happening is represented by the internal definition of the corresponding function.

We also note that things can be encoded in binary representations. Everything is essentially data, and data can be represented as a binary strings of arbitrary length. Here we only consider the most basic set of functions: binary functions, which only outputs true or false. Let 0 represent a false value and 1 represent a true value, then our function definition is:

$$f : \{0,1\}^* \to \{0,1\}$$

More complicated functions can be composed by those basic boolean functions in the same complexity class. We'll show what this means in the later sections.

Because computation takes resources, we shall also define the two most important resources of computational complexity: time complexity and space complexity.

The time complexity measures the minimal time it take for our algorithm to compute the function. "Time" is usually denoted by the number of steps it take for the algorithm to run.

The space complexity measures the minimal memory resource it take for our algorithm to compute the function. The memory usually denoted by the number of active memory storage running in parallel. Because memory storage can be flushed and reused, space complexity measures the maximum memory load of an algorithm over its whole runtime.

Both complexities are measured in big-O bounds, and different bounds can classify complexities into different complexity classes. Since the time and space for the algorithm usually depends on the size of the input data, the binary string input length n is usually the domain of the complexity.

3 Turing Machine

The Turing Machine is a classical and perhaps the most well known model of computability. As a fundamental abstract model it really captures the way things run in computers, so it has huge connection with modern computers as well.

The idea behind it is that any Machine has some internal states, and it has access (read/write) to some memory depending on its state. A Turing Machine is essentially a program written in a particular programming language. The program has access to three arrays and three pointers (sometimes called a tape):

– x which is accessed using the pointer i.

  x is an array that can be read but not written into, also known as the input tape.

– y which is accessed using the pointer j.

  y can be read and written into, also known as the working tape.

– z which is accessed using the pointer k.

  z can only be written into, also known as the output tape.

The Machine is described by its code. Each line of code reads the bits x_i and y_j, and based on those values, the Machine will possibly write new bits into y_j and z_k, and then possibly after incrementing or decrementing i, j, k, jumps to a different line of code or stops computing. Initially, the input is written in x and the goal is for the output to be written in z at the end. i, j, k are all set to 1 to begin with.

3.1 Example programs:

This single line program will copy all the input to the output.

```
Step 1:
    If x_i is empty:
        then HALT
    Else:
        Set z_k = x_i and increment i, k.
        Jump to step 1.
```

This program will output the input bits which are in odd locations.

```
Step 1:
    If x_i is empty:
        then HALT
    Else:
        Set z_k = x_i and increment i, k.
        Jump to step 2.
```

```
7
8  Step 2:
9      If x_i is empty:
10         then HALT
11     Else:
12         Increment i, k.
13         Jump to step 1.
```

3.2 Complexity of Turing Machine

We can then talk about the complexity of computing a particular function $f : \{0,1\}^* \to \{0,1\}$.

Definition 3.1. $|\alpha|$ denotes the **length** of binary string α

The above definition is particularly useful for us to reason about the size of input string.

If we fix a Turing Machine M that computes a function f, we can measure its time and space complexity.

Definition 3.2. The **time complexity** of a Turing Machine M is we how many steps the Turing Machine takes in order to halt. Formally, the Machine has running time $T(n)$ if on every input of length n, it halts within $T(n)$ steps.

Definition 3.3. The **space complexity** of a Turing Machine M is the maximum value of j during the running of the Turing Machine. We say it has space $S(n)$ if on every input of length n, j is always bounded above by $S(n)$.

The following theorem is immediate:

Theorem 3.4. *The space used by a Machine is at most the time it takes for the Machine to run.*

Proof. By our definition, because we only increment or decrement the pointers, the maximum value that the pointer can get for a given time is when the pointer only increments. □

3.3 Extended Church-Turing Thesis

The reason Turing Machines are so important is because of the Extended Church-Turing Thesis.

The thesis says that, **every** efficient computational process can be simulated using an efficient Turing Machine as formalized above. Every "physically realizable" computer can also be carried out by a Turing Machine. We say a Turing Machine is efficient if it carries out the computation in polynomial time.

The Church-Turing Thesis is not a mathematical claim, but a wishy-washy philosophical claim about the nature of the universe. As far as we know so far, it is a sound one.

We can show that, if we changed the above model slightly (for example, increase the number of tapes to more than 3, or by adding more alphabets beyond 0 and 1 just like the real-world programming languages), then we can still simulate any program in the new model using a program in the model we have chosen.

Theorem 3.5. *A program written for an L-tape Machine that runs in time $T(n)$ can be simulated by a program with 1 input tape, 1 work tape and 1 output tape in time $O\left(L \cdot T(n)^2\right)$.*

Proof. The idea is to encode the contents of all the new work arrays into a single work tape.

We can use the first L locations on the work tape to store the first bit from each of the L arrays, then the next L locations to store the second bit from each of the L arrays, and so on. The actual pointer in the new Turing Machine will then do a big left to right sweep of the array to simulate a single operation of the old Machine.

Each operation in the L-tape will take $T(n)$ time to run; by our simulation, we need another $T(n)$ time to "pick up" the original pointer location for each step, as we are jumping around on our only work tape. We need another factor of L to address the expansion of locations, which brings us the time complexity of $O\left(L \cdot T(n)^2\right)$. □

Theorem 3.6. *A program written using symbols from a larger alphabet Γ that runs in time $T(n)$ can be simulated by a Machine using the binary alphabet in time $O(\log|\Gamma| \cdot T(n))$.*

Proof. We encode every element of the old alphabet in binary. This requires $O(\log|\Gamma|)$ bits to encode each alphabet symbol. Each step of the original Machine can then be simulated using $O(\log|\Gamma|)$ steps of the new Machine. □

By the Extended Church-Turing Thesis, we then have that there is a Machine that can compile and run the code of any other Machine efficiently. In rigorous terminologies, we have the following theorem:

Theorem 3.7. *There is a Turing Machine M such that given the code of any Turing Machine α and an input x as input to M, if α takes $T > 1$ steps to compute an output for x, then M computes the same output in $O(C \cdot T \cdot \log T)$ steps, where here C is a constant that depends only on α and not on x.*

Proof. We utilize a simulation method where the Turing Machine M employs a compressed encoding of the tape and state transitions. The key idea is to use an optimized simulation strategy that involves "macro" steps. In these steps, the Turing Machine M processes large blocks of the input and tape transitions at once. By grouping $\log T$ transitions into single steps, M can simulate the behavior of α much more efficiently.

Here is how we construct M:

M takes as input the description of the Turing Machine α and the input x. It reads the description of α and sets up an internal representation of α's transition table, so it simulates α's computation on x by using a compressed representation of α's tape. M groups multiple transitions into macro steps, thereby reducing the number of simulated steps.

Each macro step of M simulates multiple steps of α, with each macro step taking $O(\log T)$ time. Since α takes T steps, the total number of macro steps is $O(T/\log T)$. Therefore, the total time complexity of M is $O(CT \log T)$, where C depends only on α and not on x.

The constant C encapsulates the fixed overhead associated with the specific Turing Machine α, including the overhead of managing the tape encoding and the transition table for the macro steps. Since this constant does not depend on the input x, it remains fixed for a given α.

By leveraging tape compression and macro step simulation, we have shown that there exists a Turing Machine M that can simulate any Turing Machine α on input x and produce the same output in $O(CT \log T)$ steps, where T is the number of steps α takes, and C is a constant depending only on α. This proves the theorem.

Therefore, the existence of such a Turing Machine M is guaranteed by the efficient simulation capabilities of multi-tape Turing Machines and the use of macro steps with logarithmic overhead, completing the proof. □

4 Uncomputable Functions

The definition of uncomputable is essentially a function that cannot be computed for a certain computing model with its restrictions on time and space limits. From now on we mainly focus on computability of Turing Machines.

There is a very philosophically powerful fact. If we assume the Church-Turing Thesis is true, then there are some ways to manipulate information that can never occur in the universe. It seems hard to imagine a physical process that violates the Church-Turing thesis, and it also seems hard to imagine the fact that the universe cannot manipulate information in a particular way, but we will show it below.

4.1 Definition

Given a string α, we write M_α to denote the Turing Machine whose code is α, and we write $M_\alpha(x)$ as the output of Turing Machine whose code is α on input x.

Theorem 4.1. *There is a function that is not computed by any Turing Machine.*

Proof. Consider the function $f : \{0,1\}^* \to \{0,1\}$ defined as:

$$f(\alpha) = \begin{cases} 1 & \text{if } M_\alpha(\alpha) \text{ halts and outputs } 0 \\ 0 & \text{otherwise} \end{cases}$$

No Turing Machine can compute this function.

Assume for the contrary that there is some Machine that could, let γ denote the binary encoding of its code, then we have that $M_\gamma(\gamma) = f(\gamma)$, but this contradicts the definition of f , since if f (γ) = 0, then $M_\gamma(\gamma)$ cannot be 0, and if f (γ) = 1, $M_\gamma(\gamma)$ cannot be 1. □

You may think that this uncomputable function we found above is very artificial and deliberate, but we can make this more intuitive using a diagonalization argument, that we will see in a moment. Some classical natural example is the Halting Problem, which is also impossible to compute using Turing Machines.

4.2 Halting Problem

Define the function $HALT : \{0,1\}^* \to \{0,1\}$ that takes as input two strings α, x, and then decides whether $M_\alpha(x)$ halts or runs forever. This is a very useful function to compute, for example it can help compilers to catch a bug of a program before it gets stuck in an infinite loop. Unfortunately it is uncomputable.

Theorem 4.2. *HALT is not computable by any Turing Machine.*

Proof. Suppose it is. Then consider the Machine M that on input α first simulates HALT(α, α). If the answer is that $M_\alpha(\alpha)$ halts, then M simulates $M_\alpha(\alpha)$ and outputs the opposite of its output. If $M_\alpha(\alpha)$ does not halt, then M outputs 0. Then M would compute the uncomputable function f above, which is false. □

5 Gödel's Incompleteness Theorem

The theorem is a statement about proof systems. It states that, any consistent formal system that is powerful enough to express basic arithmetic cannot be both complete and consistent. In other words, there will always be true statements about natural numbers that cannot be proven within the system.

Definition 5.1. A **proof system** is given by a collection of axioms. Given a list of such axioms, a proof is a sequence of statements that uses the axioms to prove that a statement is true.

For example, here are two axioms about the integers:

1. For any integers a, b, c, $a > b$ and $b > c$ implies that $a > c$.
2. For any integer a, $a + 1 > a$.

To prove that $a > b$ implies that $a + 1 > b$, we can combine the assumption $a > b$ with the axiom $a + 1 > a$ and the first axiom, to prove $a + 1 > b$.

Background: Prior to Gödel's work, mathematicians were trying to axiomatize all of mathematics. They were looking for a set of finite axioms that could be combined to prove any proof statement. Godel proved that this a doomed project.

Definition 5.2. A set of axioms is **consistent** if the axioms don't contradict each other.

Definition 5.3. A set of axioms is **complete** if every true statement can be derived from the set of axioms.

Definition 5.4. Given $x \in \{0,1\}^*$, its **Kolmogorov complexity** $K(x)$ is the length of the shortest program α such that $M_\alpha(.) = x$. Namely it is the length of the shortest program that outputs x.

Theorem 5.5. *For $N \in \mathbb{N}$, then for every N, there is an x for which the statement $K(x) > N$ is true.*

Proof. There are only a finite number of programs of length N, so for each N, there are only a finite number of x's such that $K(x) \leq N$. This means that almost all statements $K(x) > N$ are true. □

Theorem 5.6. *Every consistent finite set of axioms is incomplete.*

Proof. Assume we have a formal system $\mathcal{F}$ that is consistent and complete. This system can express statements about the Kolmogorov complexity $K(x)$ of strings x.

Consider a specific statement within $\mathcal{F}$ of the form S_N: "There exists a string x such that $K(x) > N$." This statement is true for sufficiently large N by the properties of Kolmogorov complexity.

Suppose $\mathcal{F}$ could prove S_N. This means that $\mathcal{F}$ can effectively verify the existence of a string x such that $K(x) > N$.

Consider encoding the statement "The Kolmogorov complexity of this statement is greater than N." This is a self-referential statement, and if $\mathcal{F}$ can prove it, it can generate a string x_S corresponding to this statement.

If $\mathcal{F}$ can prove the statement and hence construct x_S, then x_S must have a complexity greater than N. However, by constructing x_S within the system, $\mathcal{F}$ essentially provides a description (or a program) for x_S, which contradicts the definition of $K(x_S)$ because it would imply that $K(x_S)$ is actually less than or equal to the length of the description used by $\mathcal{F}$ to construct x_S. Therefore, if $\mathcal{F}$ can prove S_N, it leads to a contradiction because $\mathcal{F}$ would be able to provide a shorter description of x than $K(x) > N$ allows. □

This contradiction implies that there exist true statements about the Kolmogorov complexity of certain strings that $\mathcal{F}$ cannot prove without contradicting itself. Thus, $\mathcal{F}$ cannot be both complete and consistent, thereby demonstrating Gödel's incompleteness theorem.

6 Complexity Classes

Definition 6.1. DTIME($t(n)$) is the set of functions computable in time $O(t(n))$.

Definition 6.2. DSPACE($t(n)$) is the set of functions computable in computable in space $O(s(n))$.

Definition 6.3. $P = \cup_{c \geq 1}\text{DTIME}(n^c)$, the polynomial time complexity class.

Definition 6.4. $EXP = \cup_{c \geq 1}\text{DTIME}(2^{n^c})$, the exponential time complexity class.

$L = \text{DSPACE}(\log n)$, the log space complexity class.

Definition 6.5. $Pspace = \cup_{c \geq 1}\text{DSPACE}(n^c)$, the polynomial space complexity class.

Definition 6.6 (Time Constructible Functions)**.** We say that the function $t : N \to N$ is **time constructible** if $t(n) \geq n$ and on input x there is a Turing Machine that computes $t(|x|)$ in time $O(t(|x|))$.

Theorem 6.7 (Time Hierarchy)**.** *If r, t are time-constructible functions satisfying $r(n)\log r(n) = o(t(n))$, then DTIME($r(n)$) $\subsetneq$ DTIME($t(n)$).*

Proof. Consider the function $f : \{0,1\}^* \to \{0,1\}$ defined as:

$$f(\alpha) = \begin{cases} 1 & \text{if } M_\alpha(\alpha) \text{ halts and outputs 0 after } t(|\alpha|) \text{ steps of the simulator} \\ 0 & \text{otherwise} \end{cases}$$

First, f can be computed in time $O(t(n))$.

To compute f, we first compute $t(|\alpha|)$ and then apply Theorem 3.7 above to simulate $M_\alpha(\alpha)$ for $t(|\alpha|)$ steps of the simulator. So, f can be computed in time $O(t(n))$.

Second, f cannot be computed in time $O(r(n))$.

If β is the code of a Machine that computes f in time $c \cdot r(n)$. Let C_β be a constant such that the execution of r steps of the Machine M_β can be simulated in $C_\beta r \log r$ steps by the universal Machine. Then there must be some binary string β' that represents the same Machine as β, but is long enough so that

$$t(|\beta'|) > C_\beta \cdot c \cdot r(|\beta'|)\log r(|\beta'|)$$

This is because by assumption $r(n)\log r(n) = o(t(n))$, and so for large enough n,

$$t(n) > 2C_\beta \cdot c \cdot r(n)\log(r(n)) > C_\beta \cdot c \cdot r(n)\log(c \cdot r(n))$$

Moreover, we can always add redundant lines to the code in β, until the code becomes long enough for $t(|\beta'|) > 2C_\beta \cdot c \cdot r(|\beta'|) \log r(|\beta'|)$.

If $M_\beta(\beta') = 0$, then $M_{\beta'}(\beta') = 0$ and so $f(\beta') = 1$ by the guarantee of Theorem 3.7. If $M_\beta(\beta') = 1, M_{\beta'}(\beta') = 1$, and so $f(\beta') = 0$, which proves that M_β does not compute f.

Because there is some function f in DTIME($r(n)$) but not in DTIME($t(n)$), we have DTIME($r(n)$) $\subsetneq$ DTIME($t(n)$). □

Definition 6.8 (Space Constructible Functions)**.** We say that the function $s : N \to N$ is **space constructible** if $s(n) \geq \log n$ and on input x there is a Turing Machine that computes $s(|x|)$ in space $O(s(|x|))$.

Theorem 6.9 (Space Hierarchy)**.** *If q, s are space-constructible functions satisfying $q(n) = o(s(n))$, then $DSPACE(q(n)) \subsetneq DSPACE(s(n))$.*

Proof. Similarly, consider a function $f : \{0,1\}^* \to \{0,1\}$ decided by a Turing maching M that, for input α, simulates the i-th Turing Machine M_i that uses $q(n)$ space.

If M_i accepts w within $q(n)$ space, M rejects w; if M_i rejects w within $q(n)$ space, M accepts w.

Since $q(n) = o(s(n))$, there exists an n_0 such that for all $n \geq n_0$, $q(n) < s(n)$.

The Machine M can simulate M_i within $q(n)$ space and then perform the acceptance/rejection step, all within $s(n)$ space.

By diagonalization construction, f differs from the language of each M_i on at least one input w. Thus, f cannot be decided by any Turing Machine using $q(n)$ space.

Hence we have, $f \in$ DSPACE($s(n)$) since M decides f using $s(n)$ space, $f \notin$ DSPACE($q(n)$) because it is constructed to differ from any language decided within $q(n)$ space.

Therefore, DSPACE($q(n)$) $\subsetneq$ DSPACE($s(n)$), completing the proof. □

With those theorem, we can achieve a hierarchy relationship between the complexity classes:

Theorem 6.10. $L \subseteq Pspace$

Proof. By the Space Hierarchy theorem, $L \subseteq DSPACE(\log(n))$, $\log(n) = o(n)$, so by the Space Hierarchy theorem, $DSPACE(log(n)) \subseteq DSPACE(n) \subseteq Pspace$. □

Theorem 6.11. $P \subseteq EXP$

Proof. By the Time Hierarchy theorem, $P \subseteq \text{DTIME}(n^{\log(n)})$, $n^{\log(n)} = o(2^n) \subseteq EXP$, so $P \subseteq EXP$. □

Theorem 6.12. $L \subseteq P$

Proof. A function f is in L (logspace) if it can be decided by a deterministic Turing Machine M using only $O(\log n)$ space on its work tape, where n is the length of the input. Because M uses $O(\log n)$ space, it can only access a limited number of configurations, specifically $n^{O(1)}$ configurations, since the number of possible tape contents is $2^{O(\log n)} = n^{O(1)}$.

We can simulate a logspace Machine M using a polynomial-time Machine M'. M' constructs a graph where each node represents a configuration of M, and an edge represents a valid transition between configurations. Since there are only $n^{O(1)}$ configurations, this graph is polynomial in size.

To decide whether M accepts an input, M' performs a reachability analysis on this graph, checking if an accepting configuration is reachable from the initial configuration. This graph traversal can be done in polynomial time since the graph itself has polynomially many nodes and edges.

Thus, any language decidable in logspace can also be decided in polynomial time. □

Theorem 6.13. $P \subseteq Pspace$

Proof. If a function can be decided in polynomial time, it can also be decided using polynomial space. This is because a polynomial-time Machine has a polynomial number of steps, and during each step, it can only use a polynomial amount of space. Thus, any computation that fits within polynomial time constraints must also fit within polynomial space constraints. □

Theorem 6.14. $Pspace \subseteq EXP$

Proof. If a language can be decided in polynomial space, it can also be decided in exponential time. This is because a Turing Machine using polynomial space can have an exponential number of configurations (since the number of configurations is bounded by $2^{p(n)}$ for some polynomial $p(n)$). To decide the language, the Machine can explore all possible configurations, which takes at most exponential time. □

Combining all the theorems above in this section, we have the hierarchy between these complexity classes:

Theorem 6.15. $L \subseteq P \subseteq Pspace \subseteq EXP$.

7 Conclusion

In this paper, we have explored the foundational concepts of Computational Complexity, providing a comprehensive introduction to key topics and theorems in the field. We began by defining the Turing machine model, which serves as the backbone for understanding computational processes and complexity. Through detailed discussions, we examined time and space complexity, outlining how these resources are quantified and bounded during Turing machine execution.

We delved into significant theorems that illustrate the constraints on computational resources, highlighting methods to effectively bound time and space costs. This led us to a critical aspect of computational theory—the existence of uncomputable functions. By presenting the halting problem, we demonstrated a classic example of such functions, emphasizing the limits of algorithmic computation.

Further, we introduced Gödel's Incompleteness Theorem, employing diagonalization as a proof technique to showcase its implications on formal systems and computational theory. This theorem underscores the inherent limitations in formal systems, paralleling the uncomputability results discussed earlier.

Lastly, we explored various complexity classes, elucidating the hierarchical relationships between them. By understanding classes such as P, NP, and beyond, we gain insights into the landscape of computational problems and their relative complexities.

Overall, this paper has laid out a structured pathway from the basic definitions of computational models to the sophisticated hierarchy of complexity classes. These discussions not only provide a fundamental understanding of computational complexity but also underscore the profound implications of these concepts on the broader field of computer science. Through this exploration, we appreciate the intricate balance between what can be computed and the resources required, paving the way for future research and discovery in computational theory.

8 References

1. "Computational Complexity: A Modern Approach" by Sanjeev Arora and Boaz Barak [2007].
2. "CSE 431: Computational Complexity" by Anup Rao [2024].

9 Question 2

I am not comfortable with my paper being shared.