UCSB, Physics 129L, Computational Physics
Lecture notes, Week 3

Zihang Wang (UCSB), zihangwang@ucsb.edu

January 24, 2025

Contents

1 Computational complexity and Turing Machine 1

2 Oracle Machine 2

3 Decision problem and function problems 4

4 Complexity Classes and Algorithmic Complexity 4

4.1 Complexity of decision problems 4

5 Open questions 9

5.1 Generalization of computational complexity to a Function Problem 9

5.2 NP-Complete 9

5.3 NP-Hard 10

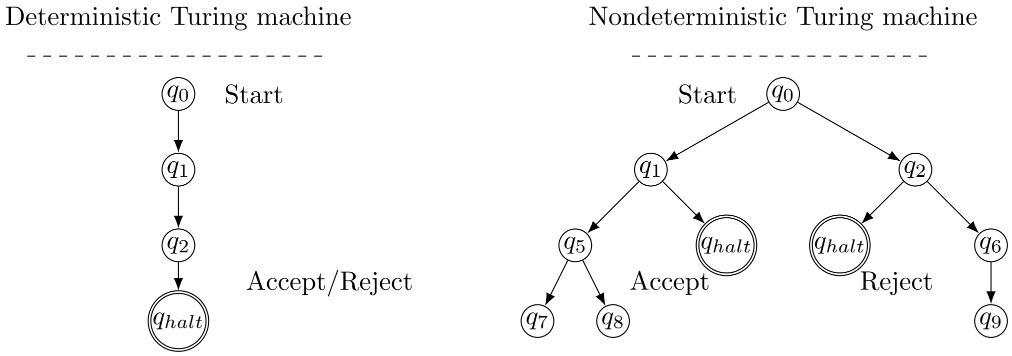
5.4 NP-Intermediate 10

6 Algorithmic Complexity 10

7 Examples on Computation Complexity 11

8 From classical to quantum parallelism: Grover's algorithm 13

1 Computational complexity and Turing Machine

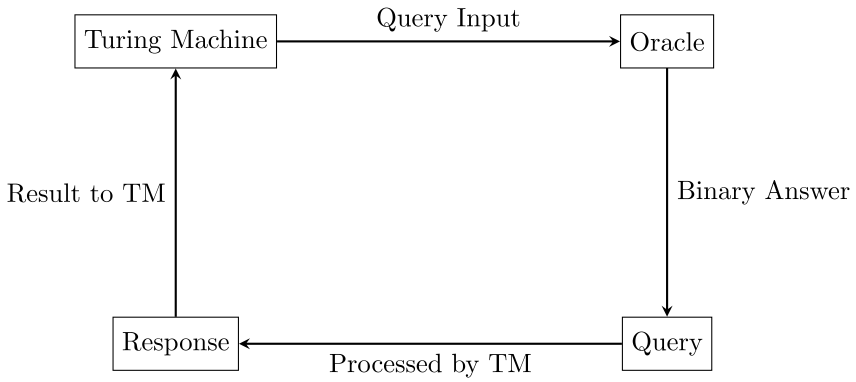


Deterministic Turing machines (TM) and Non-deterministic Turing machines (NTM) are fundamental objects used in quantifying computational complexity: a measure that reflects the amount of resources, such as the number of steps (time) and tape length (space), required for a TM or an NTM to reach a halt state with a given input size n. In particular, the time complexity of an NTM is defined as the maximum number of steps taken on any branch of its computation for an input of size n.

It is worth noting that the input size n is different from the n used in an n-state TM or NTM discussed previously. The space used by a TM or an NTM is often measured by the number of cells on the tape it uses during computation. Unlike in modern computation with random access memory, a Turing machine is only above to move step by step. This restricts the time complexity to be at least n, i.e. you need at least with n steps to scan all elements.

2 Oracle Machine

An **Oracle Machine** is a theoretical extension of the Turing machine, introduced to study decision problems and computational complexity. It operates like a standard Turing machine but includes access to an **oracle**, which is an abstract “black box” capable of solving specific problems instantly, even if those problems are undecidable or computationally hard.



An oracle machine M^O is given by the following,

- A standard Turing machine M, with its components such as states, tapes, and transition rules.
- An additional **oracle tape** and **oracle head**.
- Two special states: the **query** state and the **response** state.

The oracle is represented as a set $O \subseteq \mathbb{N}$. The machine can query whether a number n belongs to O as a binary $0, 1$,

$$O(n) = \begin{cases} 1 & \text{if } n \in O, \\ 0 & \text{if } n \notin O. \end{cases} \tag{1}$$

During computation, the oracle machine alternates between standard operations and consulting the oracle for specific queries.

Time Complexity

The time complexity of an oracle machine depends on:

1. The number of standard computational steps.
2. The number of queries made to the oracle.

If querying the oracle is considered a single step, the overall complexity remains dominated by the non-oracle operations. However, using powerful oracles can collapse certain complexity distinctions (e.g., making undecidable problems solvable). Oracle machines are a fundamental tool in theoretical computer science for examining limits of computation and relationships between complexity classes.

3 Decision problem and function problems

A **decision problem** is a computational problem that can be expressed as a yes-or-no question based on a given input tape. For example, checking if certain symbols exist on a given tape, or checking certain numbers are prime. For decision problems, the output is binary (”yes” or ”no”), which corresponds to the machine halting in one of two states q_{halt} (accept or reject).

A **function problem** requires outputs of certain properties. For example, sorting an array, evaluate mathematical expressions, or finding the index of an element on the tape. The output is typically written on the tape, and the machine halts after completing the computation.

Function problems sometimes contain decision problems. For example, **binary search** and **sorting problem** can be reduced to repeatedly answering individual comparisons (decision problems).

4 Complexity Classes and Algorithmic Complexity

When discussing computational complexity, it is important to distinguish the following,

- Complexity of a decision problem,
- Complexity of a function problem,
- Algorithmic complexity.

A decision problem checks if an input satisfies some property, producing a yes-or-no answer. On a Turing machine, this means halting in either an accept halting state (yes) or reject halting state (no).

A function problem requires producing an output on the tape before halting, which often requires post-processing. It cover a wide range of computational tasks where the goal is to produce a specific result, including constructing, counting, or transforming objects, rather than just determining the existence of a solution in a decision problem.

Algorithmic complexity refers to analyzing how efficiently specific algorithms run on a Turing machine, whether solving decision problems or function problems. Algorithmic complexity usually expressed using Big-O notation (e.g., $\mathcal{O}(n^2)$, $\mathcal{O}(\log n)$), which reflects how efficiently specific algorithms run on Turing machines for either type of problem.

4.1 Complexity of decision problems

While the exact quantification of complexity classes for decision problems remains unsolved, complexity classes is conjectured as the following nested sets,

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \subseteq NEXPTIME \subseteq EXPSPACE, \tag{2}$$

and it can be written as the following diagram from smallest set to the largest.

$$\begin{tikzcd}
\boxed{L} \arrow[r] & \boxed{NL} \arrow[r] & \boxed{P} \arrow[r] & \boxed{NP} \arrow[r] & \boxed{PSPACE} \arrow[d] \\
\boxed{EXPSPACE} & & \boxed{NEXPTIME} \arrow[ll] & & \boxed{EXPTIME} \arrow[ll]
\end{tikzcd}$$

To make the explanation more clear and intuitive, we can adjust the structure, use examples strategically, and provide context for each point. Here's a revised version with improvements:

Loosely speaking, "N{}" refers to problems that involve a nondeterministic Turing machine. In these cases, once the full computational tree is generated, verifying whether a halting state q_{halt} is an accept state (rather than a reject state) requires the resources specified by "{}". For example, NP stands for Nondeterministic Polynomial Time, meaning that a solution guessed by a nondeterministic Turing machine can be verified in polynomial time on a deterministic Turing machine.

This reflects a general pattern: deterministic complexity is a subset of its nondeterministic counterpart. Similarly, the time complexity of decision problems is a subset of their space complexity, as space can often be reused across steps while time cannot.

In the following sections, I will introduce these complexity classes and their relationships one by one, providing examples to illustrate their definitions and significance.

L: Logspace

L is the class of decision problems solvable by a deterministic Turing machine using $O(\log n)$ space on a work tape. Algorithms in L have extremely tight memory constraints. An example problem in L is checking the number of zeros on a tape. The storage required for counting is given by $\text{bit} = \log_2(n)$ since the binary $n = 2^{\text{bit}}$.

NL: Nondeterministic Logspace

NL is the nondeterministic counterpart to L. In NL, a nondeterministic machine can branch into many possible paths of execution, but it must still use $O(\log n)$ space overall. A classic NL problem is 2-Satisfiability Problem (2-SAT), as the example given below. When we navigate on the tape and retrieve data, we need a counter.

P: Polynomial Time

P (Polynomial Time) comprises all decision problems solvable by a deterministic Turing machine in polynomial time with respect to the input size. For example, counting number of 1s on a given tape. While P focuses on time rather than space, any problem in L or NL follows P. This is due to the fact that a Turing machine using $\log n$ space can only access a polynomial number of configurations, which is sufficient to reach the halting state q_{halt} with acceptance if the problem is solvable.

NP: Nondeterministic Polynomial Time

NP contains problems solvable by a nondeterministic Turing machine in polynomial time (if you follow a branch within the computation tree), or equivalently, problems whose solutions can be *verified* in polynomial time by a deterministic machine. Well-known NP problems include

- Satisfiability (SAT): see below,
- Traveling salesman problem (Decision Version): Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? (if there is a route smaller than k)
- **The Hamiltonian Path problem**: a path that visits every vertex in the graph exactly once.
- Decision versions of **prime factorization** are in NP (Does the integer n have a prime factor smaller than k?)

Whether NP is strictly larger than P remains one of the biggest open questions in complexity theory.

Example in NL and NP: Boolean Satisfiability Problem (SAT)

Let's look at an important problem, namely the Boolean Satisfiability Problem (SAT) in detail: determining whether there exists Boolean variables in a given Boolean formula such that the entire formula evaluates to true. A Boolean formula is typically expressed in Conjunctive Normal Form (CNF), which is a conjunction (AND) of one or more clauses, where each clause is a disjunction (OR) of literals (a variable or its negation). For example, the formula $(x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_4)$ is satisfiable if there exists Boolean values x_1, x_2, x_3, and x_4 that makes the formula true. SAT was the first problem proven to be **NP-complete**.

The **2-Satisfiability Problem (2-SAT)** is a special case of SAT: A 2-CNF formula is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of at most two literals. The goal is to determine whether there exists Boolean variables such that the entire formula evaluates to true.

For example:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3), \tag{3}$$

- Conjunctive normal form (CNF) is a conjunction of one or more clauses, where a clause is a disjunction of literals.

- x_1, x_2, x_3 are Boolean variables.
- $\vee$ is a disjunction, i.e. "OR".
- $\wedge$ is a conjunction, i.e. "AND".
- $\neg x_2$ means "not x_2."
- $(x_1 \vee \neg x_2)$ is a **clause**.
- each **clause** contains at most 2 **literals**, e.g. $x_1 \vee \neg x_2$. That is way they are called 2-SAT.
- The formula is satisfied if there exists x_1, x_2, x_3 such that all clauses evaluate to true.

Let's focus on one branch of the computation tree of an NTM that produces a true value for the following input formula on the tape, $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$. An NTM generates a computation tree from the input formula, and we assume in the end, there at least one TM reaches a halting state (q_{halt}). Let's say the output on the tape is given by the following,

$$x_1 = \text{true}, \quad x_2 = \text{false}, \quad x_3 = \text{true}. \tag{4}$$

To verify the halting state (q_{halt}) is an accept state, we must evaluates the formula clause by clause,

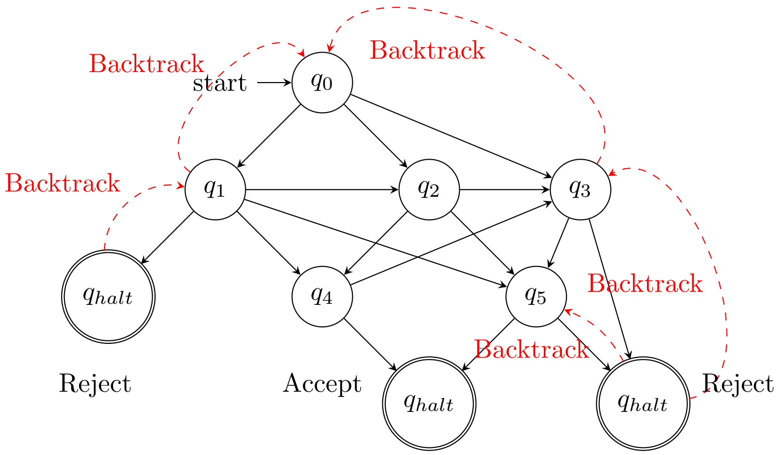
- For $(x_1 \vee \neg x_2)$: Since $x_1 = \text{true}$, this clause is satisfied.
- For $(\neg x_1 \vee x_3)$: Since $x_3 = \text{true}$, this clause is satisfied.

If all clauses are satisfied, halting state (q_{halt}) is an accept state. In other words, the NTM can make guesses and **verify** the correctness of a solution in logarithmic space. The space is not used to compare values but to keep track of where we are in the input. In summary,

- NL consists of decision problems solvable by a nondeterministic Turing machine using $O(\log n)$ space.
- While NL problems allow nondeterminism, deterministic solutions may require more than logarithmic space unless $L = NL$ (an open question in complexity theory).

PSPACE: Polynomial Space

$PSPACE$ is the collection of decision problems solvable with a polynomial amount of space. Even if an algorithm takes exponential time, as long as it uses only polynomial space, it belongs to $PSPACE$. This means that any problem in NP (including NP-complete problems like SAT or Hamiltonian path) can be solved by deterministic TM using polynomial space via the backtracking technique,



As a side note on **Savitch's Theorem**: Any problem solvable nondeterministically in $\mathcal{O}(p[n])$ space can also be solved deterministically in $\mathcal{O}(p[n]^2)$ space.

By Savitch's Theorem, nondeterministic polynomial space (NPSPACE) is equivalent to deterministic polynomial space (PSPACE). However, it does not work for Logspace because $\log(n)$ and $[\log(n)]^2$ are different, i.e. $L \subseteq NL$.

EXPTIME: Exponential Time

$EXPTIME$ consists of problems solvable by a deterministic Turing machine in time bounded by an exponential function of the input size, such as $2^{p(n)}$ for some polynomial p. $EXPTIME$ includes $PSPACE$, because a machine using polynomial space can only run through a limited number of configurations, implying that after exponential time it would either halt or repeat. An example is the decision version of **Generalized Chess**: Given an initial configuration on an $n \times n$ chessboard, can the first player guarantee a win?

- **Input:** Initial board configuration and generalized chess rules.
- **Output:** yes (if the first player has a guaranteed winning strategy) or no.

The problem is in $EXPTIME$, requiring exploration of an exponentially large game tree.

NEXPTIME: Nondeterministic Exponential Time

$NEXPTIME$ extends $EXPTIME$ to nondeterministic machines operating in exponential time. A problem sits in $NEXPTIME$ if a nondeterministic Turing machine can find solutions in time up to $2^{p(n)}$. Or, verification requires exponential time.

EXPSPACE: Nondeterministic Exponential Space

$EXPSPACE$ includes decision problems solvable by a deterministic Turing machine using exponential space, up to $2^{p(n)}$ for some polynomial p. It is believed to be strictly larger than $PSPACE$. Certain highly complex problems involving large search spaces fall naturally into $EXPSPACE$, where even exponential time may not suffice but exponential space is allowed.

5 Open questions

P and NP are fundamental complexity classes that are used in various contexts. One of the central questions in complexity theory is: Does $P = NP$? In Turing machine terms, this asks:

> Can every problem solvable by a non-deterministic Turing machine
> in polynomial time also be solved by a deterministic Turing machine
> in polynomial time?

If $P = NP$, it would mean that finding solutions is no harder than verifying them, revolutionizing fields like cryptography and optimization. However, people currently believe $P \neq NP$, suggesting some problems are inherently harder to solve than to verify.

5.1 Generalization of computational complexity to a Function Problem

As discussed previously, a decision problem is in NP if:

- It can be solved by a **nondeterministic Turing machine** in polynomial time, or equivalently,
- A solution can be **verified** in polynomial time by a deterministic Turing machine.

5.2 NP-Complete

We can further extend the definition of NP to a specific subset NP**-Complete**. A decision problem is NP-Complete if:

- It belongs to NP, and
- Every other problem in NP can be **reduced** to it in polynomial time.

These problems are the "hardest" in NP: if any NP-Complete problem can be solved in polynomial time, then every problem in NP can also be solved in polynomial time ($P = NP$). The Satisfiability Problem (SAT) we discussed previously is NP-Complete. Any problem in NP can be transformed into an instance of SAT in polynomial time.

5.3 NP-Hard

We further extend the NP concept in decision problems to function problems: A general problem is NP**-Hard** if:

- Every problem in NP can be **reduced** to it in polynomial time, but
- It is not required to be in NP (i.e., the problem might not be a decision prolbem).

These problems are at least as hard as NP**-Complete** problems but may not be decision problems. The Traveling Salesman Problem (TSP) in its optimization form (finding the shortest route) is NP-Hard, as it involves finding a solution, in addition to verification.

5.4 NP-Intermediate

Ladner's Theorem (1975): If $P \neq NP$, then there exist problems in NP that are neither in P nor NP-complete. These are called NP-intermediate problems. However, if $P = NP$, NP-Intermediate will not exist.

Summary

- NP: Problems with solutions verifiable in polynomial time.
- NP-Complete: The hardest problems in NP.
- NP-Hard: Problems as hard as NP-Complete, not necessarily in NP.
- NP-Intermediate: Problems are neither in P nor NP-complete.

6 Algorithmic Complexity

Applies to both decision and function problems that focuses on specific algorithms rather than general problem classifications. It also measures resources in time and space, expressed in Big-O notation (e.g., $O(n^2)$, $O(n \log n)$). Binary search implemented on a Turing machine has time complexity $O(\log n)$, regardless of whether it is used for finding an element's index (function problem) or checking if an element exists (decision problem).

In the case of sorting, the minimum number of comparisons required for a Turing machine (TM) to sort a tape of length n using any comparison-based algorithm is $\Omega(n \log n)$. This **lower bound** is a fundamental property of the sorting problem itself and applies to all comparison-based sorting algorithms on a TM. For instance:

- ***Merge Sort***: Time complexity is $\mathcal{O}(n \log n)$ in all cases. Space complexity is $\mathcal{O}(n)$ (in place).

- ***Bubble Sort***: Time complexity is $\mathcal{O}(n^2)$ in the worst case. Space complexity is $\mathcal{O}(1)$ (in place).
- ***Quick Sort***: Time complexity is $\mathcal{O}(n^2)$ in the worst case but $\mathcal{O}(n \log n)$ on average. Space complexity is $\mathcal{O}(\log[n])$ (in place).

In this example, you can see that while *Merge Sort* and *Quick Sort* achieve the optimal $\mathcal{O}(n \log n)$ time for comparison-based sorting, *Bubble Sort* does not. A sorting algorithm is considered **stable** if two equal elements in the input remain in the same relative order in the sorted output.

7 Examples on Computation Complexity

1. Constant Time ($O(1)$)

Example: Read the first symbol on the tape.

A Turing Machine can solve this in $O(1)$ by directly reading the symbol under the head and transitioning to an appropriate state.

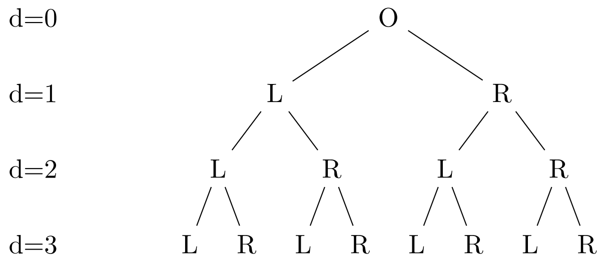
2. Logarithmic Time ($O(\log n)$)

Example: Find the first non-zero element in a sorted tape with a Random-Access Turing Machine

A Random-Access Turing Machine (RATM) can implement **binary search** efficiently by repeatedly halving the input size and accessing the middle point (left-side-included). Using a condition such as 'If value = 0, keep the right side; if value = 1, keep the left side,' it can determine whether an element exists or find its position in $O(\log n)$ time. However, on a standard Turing machine without random access, binary search would require $O(n)$ time due to sequential tape traversal.

$$\log_2(n) = d$$

$$\begin{aligned}
\log_2(1) &= 0 \\
\log_2(2) &= 1 \\
\log_2(4) &= 2 \\
\log_2(8) &= 3 \\
\log_2(16) &= 4 \\
\log_2(32) &= 5 \\
\log_2(64) &= 6
\end{aligned}$$



3. Linear Time $(O(n))$

Example: Count the number of 1's on the tape.

The Turing Machine scans the tape from left to right, counting the number of 1's in a separate register or state.

4. Linearithmic Time $(O(n \log n))$

Example: Sort a binary string using partitioning.

The Turing Machine sorts a binary string by repeatedly partitioning it into half segments of 0's and 1's until reaching the one-element partition. Then, we only compare the "boundary points" with other partitions. This is called **divide-and-conquer**.

5. Quadratic Time $(O(n^2))$

Example: Compare all pairs of symbols on the tape and sort.

The Turing Machine uses nested loops to compare every symbol with every other symbol. This is called **brute-force attack**.

6. Exponential Time $(O(2^n))$

Example: Generate all subsets of a binary string.

The Turing Machine generates all subsets by recursively constructing each combination on the tape.

7. Factorial Time $(O(n!))$

Example: Generate all permutations of a binary string.

The Turing Machine generates all permutations by systematically swapping symbols on the tape.

8 From classical to quantum parallelism: Grover's algorithm

The process of Grover's Algorithm utilizes the quantum parallelism to speedup the calculation,

Initialization: Apply Hadamard gates to all n-qubits to create the uniform superposition:

$$|\psi_0\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle.$$

Oracle Application: Use the oracle U_f to flip the phase of the marked state,

$$U_f|x\rangle = \begin{cases} -|x\rangle & \text{if } x \text{ is the marked state,} \\ |x\rangle & \text{otherwise.} \end{cases} \tag{5}$$

Diffusion Operator: Apply the diffusion operator $D = 2|\psi_0\rangle\langle\psi_0| - I$ to amplify the probability of the marked state with respect to the initial state.

Iteration: Repeat the oracle and diffusion operator $\mathcal{O}(\sqrt{N})$ times to maximize the probability of measuring the marked state, caused by the redistribution of probabilities on the denominator.

For a classical algorithm on a Turing computer (since we have to check one by one), it will take $\mathcal{O}(N)$. This is the advantage of quantum parallelism.