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

Zihang Wang (UCSB), zihangwang@ucsb.edu

January 22, 2025

Contents

1 Computation Complexity

2 Turing machine

A Turing machine M can be formally described as a quintuple (I swap the notation Γ, Σ if you want to ask why it is different from other sources you might find online):

$$M = (Q, \Gamma, \Sigma, \delta, q_0)$$

where:

- Q: Finite set of **states**: state space of a Turing machine.
- Γ: A finite set of **symbols** as the input alphabet that are directly used for computation. The set contains the symbol of the first kind (e.g. 0, 1).
- Σ: A finite set of **symbols**: state space of the tape where a Turing machine operates on. In particular, the set includes both blanks B and has Γ as a subset. Initially, all tape squares are blank except the input.

- $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R, N\}$: $\delta(Q, \Sigma)$ is a **map** (transition function): a deterministic algorithm that takes in two parameters: the current Turing machine state $q \in Q$ and the current tape symbol $s \in \Sigma$ as inputs, and outputs one new Turing machine state q', a modified tape value s', and a directional instruction left, right or hold $\mathcal{D} \in \{L, R, N\}$. The direction instruction can be loosely understood as the change in Turing machine's coordinate: left, right movement, or hold. When shifting to the right (left), it is represented by a right (left) notation,

$$\delta(q, s) = (q', s', \mathcal{D}), \tag{1}$$

  which is defined over finite **sets**: machine states, tape symbols, and directions.

  The above transition function has an alternative expression, represented by the following quintuple,

$$qss'\mathcal{D}q'; \tag{2}$$

  and note the semi-colons ";" that separates different transition rules. In this way, we are able to write all transition rules on a tape, which is critical for the **universal Turing machine**.

- $q_0 \in Q$: The initial state.

2.1 Notations

For example, let's consider a simple algorithm that replaces all s_2 by s_1. The **Turing machine** operates at an initial state q_0 on a tape symbol $B \in \Sigma$, (q_0, B),

$$M = (Q = \{q_0, q_1, q_2, q_{halt}\}, \Gamma = \{s_1, s_2\}, \Sigma = \{B, s_1, s_2\}, \delta, q_0). \tag{3}$$

The transition function is given by $\delta(q_0, B) = (q_1, s_1, R)$, which can be understood as follows:

1. The Turing machine, in state q_0, reads the symbol $B \in \Sigma$ on the tape.
2. The Turing machine changes its state to $q_1 \in Q$ and writes the symbol $s_1 \in \Sigma$ onto the tape, replacing B.
3. The Turing machine moves its head to the right (R), resulting in a new configuration where the machine is in state q_1 with its head positioned over the next symbol to the right.

2.1.1 Transition Table (single blank symbol)

The above description can be expressed via the following table (there is a better way),

Current State	Read	Write	Move	Next State
q_0	B	B	R	q_1
q_1	s_1	s_1	R	q_1
q_1	s_2	s_1	L	q_2
q_2	s_1	s_1	L	q_2
q_2	B	B	R	q_1
q_1	B	B	R	q_{halt}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(4)

or as an one-line expression, as proposed by Turing,

$$q_1 s_1 s_1 R q_1; q_1 s_2 s_1 L q_2; q_2 s_1 s_1 R q_2; q_2 BBR q_1; \tag{5}$$

The term q_{halt} above represents the terminal state of the Turing machine. If q_{halt} can represent accept or reject for complex programs. In modern computation, it can be understood as the status of the “standard error”, 0 or 1 if the program completes or fails. Therefore, the “detection” of error is directly coded in the Turing machine.

2.1.2 Standard Form

This can be written in the **standard form**. We replace the symbol $s_i = DC\cdots$ by ”D” followed by ”C” repeated i times (this D is not the direction $\mathcal{D}$ mentioned previously). Similarly, the machine state $q_i = DA\cdots$ is replaced by ”D” followed by ”C” repeated i times. For example, $q_0 = D$,$q_1 = DA$,$q_2 = DAA$, $B = D$,$s_1 = DA$,$s_2 = DAA$, and the quintuple has the following expression,

$$\begin{aligned} &q_0 B s_1 R q_1; q_0 s_1 s_1 R q_1; q_1 s_1 s_1 R q_1; q_1 s_2 s_1 L q_2; \\ &= DDDCRDA; DDCDCRDA; DDCDCRDA; DADCDCRDA; DADCCDCRDAA; \end{aligned} \tag{6}$$

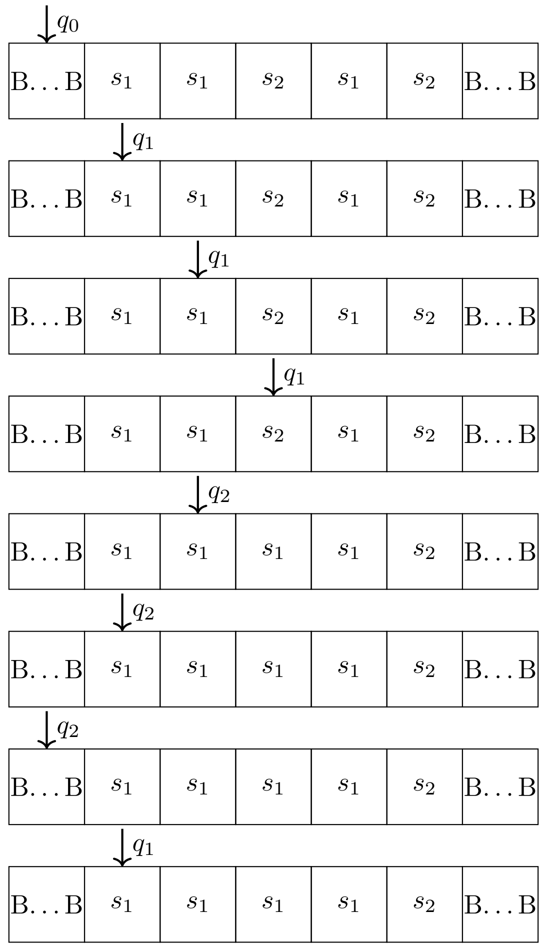
In other words, any Turing machine instruction can be written as combinations of the following 7 letters,

$$A, C, D, L, R, N, ; \tag{7}$$

If we replace ”A” by ”1” , ”C” by ”2” , ”D” by ”3” , ” L” by ”4” , ”R” by ’5” , ”N” by ”6” , and ”;” by ”7” we have a description of the Turing machine in the form of numerals. They are called **description numbers**.

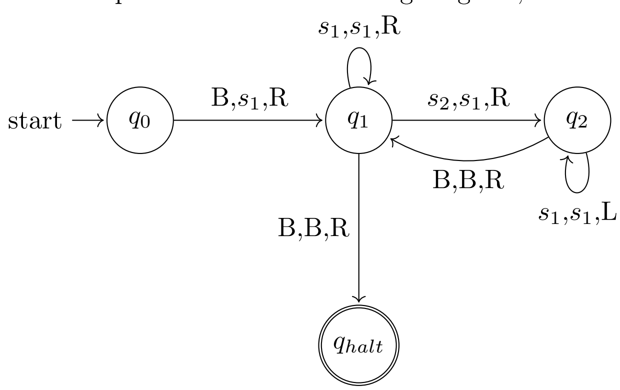
2.1.3 Examples with Tapes

Let’s look at the Turing machine in action: The Turing machine moves its head position, and read and write on the tape depending the above algorithm. The following shows the algorithm,



2.1.4 Transition Diagram

The above expression has the following diagram,



By the end of the process (reaching q_{halt}), the computation result can be read off from the tape or the halt state, $q_{halt} = q_{accept}, q_{reject}$.

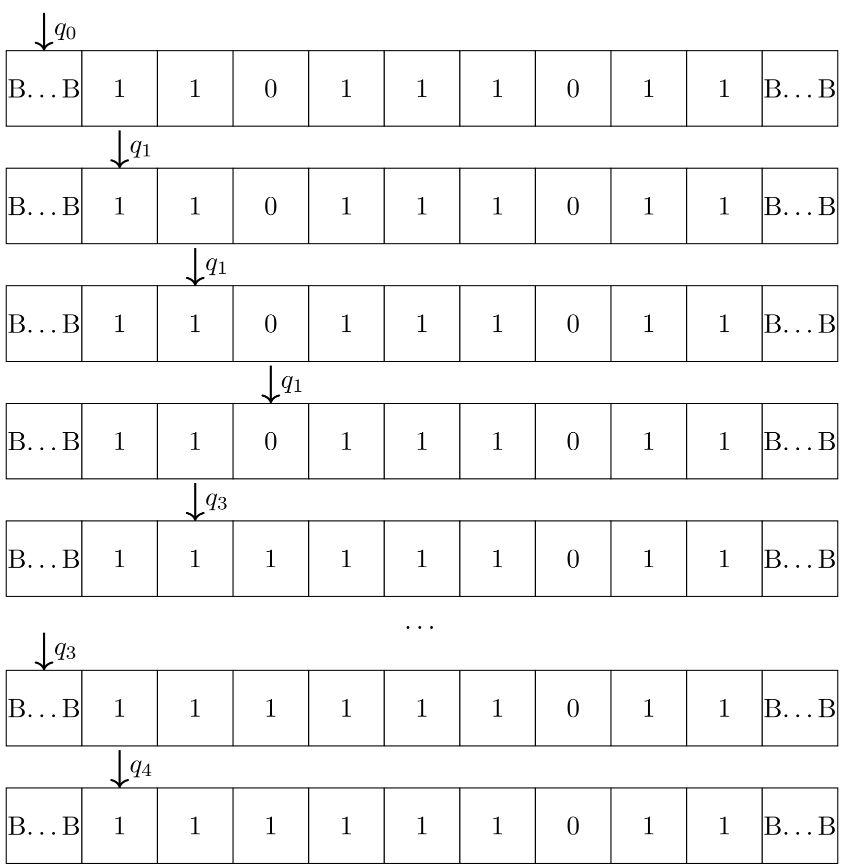
In particular, if $B, s_1, \cdots \in \{0, 1\}$ with n possible states $\{q_i\}$ (state space dimension), the Turing machine is a **2-symbol, n-state Turing machine**.

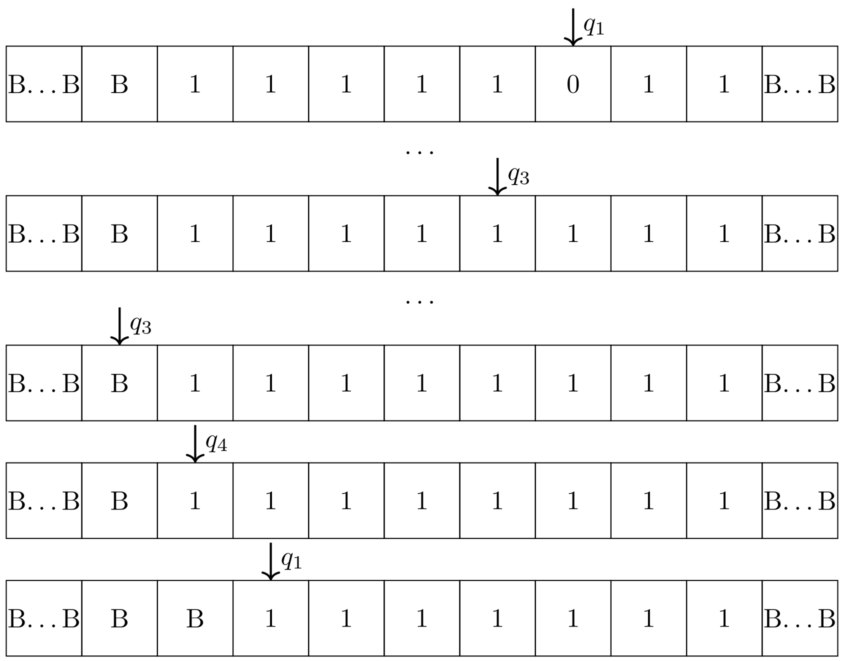
For example, let's consider the following addition problem, as shown below,

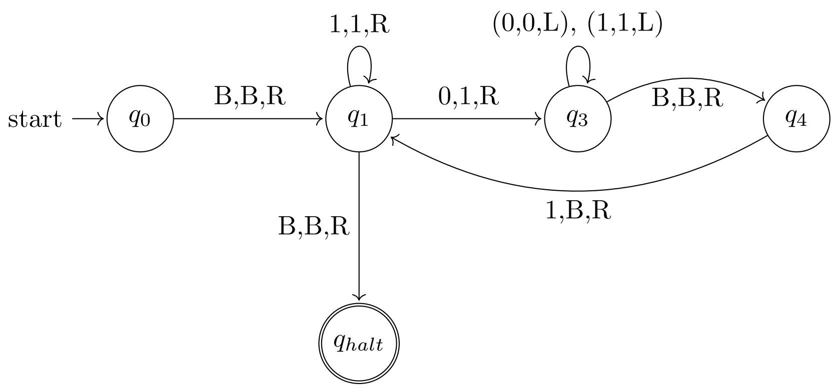
Current State	**Read**	**Write**	**Move**	**Next State**
q_0	B	B	R	q_1
q_1	0	1	R	q_3
q_1	1	1	R	q_1
q_1	B	B	R	q_{halt}
q_3	0	0	L	q_3
q_3	1	1	L	q_3
q_3	B	B	R	q_4
q_4	1	B	R	q_1
q_4	0	0	R	q_4
q_4	B	B	R	q_{halt}

(8)

And a typical binary tape looks like the following,







Loosely speaking, a Turing machine is an algorithm-specific machine under fixed rules, defined within its state space.

The **Church-Turing thesis** proposes that anything computable by an algorithm can be computed via a Turing machine. For example, a real number is Turing computable if there exists a Turing machine or algorithm capable of computing an arbitrarily precise approximation of that number. All of the algebraic numbers and important constants, such as e and π that can be determined via roots finding algorithms are **Turing-computable**.

Let's consider the famous **halting problem** that is **not Turing-computable**: Can we design an algorithm that check the number of steps a turning machine takes to halt (with an initial tape). It turns out that the halting problem is not Turing-computable since the halting problem is undecidable. In other words, we cannot design a Turing machine that preforms this task.

Then, we want to ask, what will be the maximum number of a n-state Turing machine can achieve before it halts? This is the famous **Busy Beaver problem**: It involves finding the n-state Turing machine that performs the maximum "work" on a given tape.

The value is call the **Beaver function** (BB), and it can be accessed via the enumerative search algorithm, where we scan all possible n-state Turing machines. The Beaver function is much-much faster than **any** Turing-computable functions, and here are the value (we are only able to calculate it upto 6),

$$\begin{aligned} BB(1) &= 1, \\ BB(2) &= 6, \\ BB(3) &= 21, \\ BB(4) &= 107, \\ BB(5) &= 47,176,870. \end{aligned} \tag{9}$$

For larger values where exact results are unknown, we have lower bounds,

$$\begin{aligned} BB(6) &> 10^{865}, \\ BB(7) &> 10^{10^{10^{10^{18,705,352}}}}. \end{aligned} \tag{10}$$

This is so far the fastest growing function ever existed.

As a side note, lambda calculus is an alternative approach to the Turing machine, and they emphasize different aspects of computation: lambda calculus focuses on function abstraction and symbolic manipulation, whereas Turing machines provide a step-by-step mechanical process for computation.

Lambda function, denoted by the symbol λ, represents an **anonymous function** that takes an input variable and maps to an output that follows certain rules. Given a function $\lambda x.M$ and an argument N, the result of applying the function is the expression M with x substituted by N, written as $(\lambda x.M)\,N$.

3 Nondeterministic Turing machine

The above examples are deterministic Turing machines since from a given initial condition, the trajectory is known by the transition functions.

A nondeterministic Turing machine can solve certain problems more efficiently by exploring multiple paths concurrently.

You can think it in a probabilistic way: each transition function follows a probability mass function (PMF), and the same drawn from those PMF eventually leads to a halt state.

On the other hand, you can imagine the Turing machine "branches" into many copies at each step, based on some rules. In other words, it effectively performs a **breadth-first search**.

4 Universal Turing machine

We should note that a Turing machine is designed for executing a single algorithm, and it is single-purposed. Can we make a Turing machine that can simulate other Turing machines? This is called the **universal Turing machine**, and it is able to read a tape that contains the following information,

- Description of another Turing machine M (its states, symbols, and transition rules).
- An input string w that the described machine M would process.

The UTM uses these inputs to simulate M's behavior on w, producing the same result as M would.

To achieve this, we must put both the building instruction for M and the input w on the tape.

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.

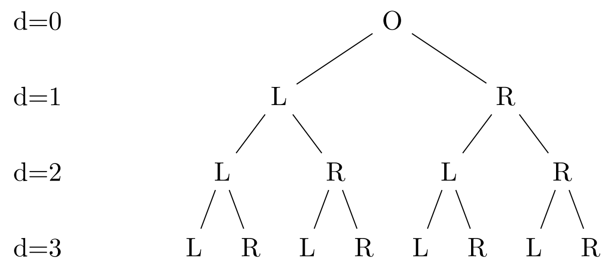
2. Logarithmic Time ($O(\log n)$)

Example: find the first non-zero element in a sorted tape

The Turing Machine repeatedly halves the input size, and get the value at the middle point with the following binary condition: If the value 0, keep the right side, and if the first index is 1, keep the left size. This is called the **binary search**.

$$\log_2(n) = d$$

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



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.