Theory of Formal Languages and Automata

Lecture 22

Mahdi Dolati

Sharif University of Technology

Fall 2023

December 22, 2023

Time Complexity

- Solvability (or decidability) is not sufficient for practical usages of an algorithm,
  - Inordinate amount of time or memory,
- Computation complexity theory,
  - Time,
  - Memory,
  - Disk,
  - Message.
- Our objective: Time complexity theory.
  - Measure the time,
  - Classify problems.

Measuring Complexity

- Example:

$$A = \{0^k 1^k | k \geq 0\}$$

- A is decidable.
- How much time does a single-tape TM need to decide A?

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

Measuring Complexity

- The time depends on several parameters,
- We can abstract all problem-specific parameters:
  - Length of the representation of input,
- Worst-case analysis or average-case analysis?
  - Longest possible running time on inputs of length n,
    - Average running time on all inputs of length n.
- Definition:

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Measuring Complexity

Big-O and Small-O Notation

- Hide constant factors and coefficients,
- Avoid complex expressions,
- Use an estimate,
  - A form of estimation: Asymptotic analysis.
    - Consider large inputs,
  - Keep the highest order term,
    - Discard its coefficient,
    - And lower order terms.

- **Example**:
  - $f(n) = 6n^3 + 2n^2 + 20n + 45 \rightarrow n^3$
  - f is asymptotically at most n^3
  - $f(n) = O(n^3)$

Measuring Complexity

Big-O and Small-O Notation

- Definition:

Let f and g be functions $f, g : \mathcal{N} \longrightarrow \mathcal{R}^{+}$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,

$$f(n) \leq c\, g(n).$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.

- $f(n) = O(g(n))$
  - f is less than or equal to g if we disregard differences up to a constant factor.
  - O: a suppressed constant.

Measuring Complexity
Big-O and Small-O Notation

- **Example**:
  - $f_1(n) = 5n^3 + 2n^2 + 22n + 6$
    - $f_1(n) = \mathrm{O}(\mathrm{n}^3)$
      - $f_1(n) \leq 6n^3$ for $n \geq 10$
    - $f_1(n) = O(n^4)$
    - $f_1(n)$ is not $O(n^2)$

- **Example**:
  - Change of the base in logarithm changes the value by a constant factor:
    - $\log_b n = \log_2 n \,/\, \log_2 b$
    - Thus, we omit the base in O notation: $f(n) = O(\log n)$

Measuring Complexity

Big-O and Small-O Notation

- **Example**:
  - We can use the O notation in arithmetic expressions:
    - $f(n) = O(n^2) + O(n) = O(n^2)$
  - Big-O in the exponent has the same meaning:
    - $f(n) = 2^{O(n)} \rightarrow f(n) \leq 2^{cn}$ for some c.
- **Example**:
  - $f(n) = 2^{O(\log n)} \rightarrow f(n) \leq n^c$ for some c.
    - $n = 2^{\log_2 n}$
  - $f(n) = n^{O(1)} \rightarrow f(n) \leq n^c$ for some c.
- **Example**:
  - Polynomial bounds: Have the form n^c
  - Exponential bounds: Have the form 2^{n^δ} for some $\delta > 0$.

- To say that one function is asymptotically less than another, we use small-o notation.
  - big-O and small-o are similar to $\leq$ and $<$
- Definition:

Let f and g be functions $f, g \colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < c\,g(n)$ for all $n \geq n_0$.

- **Example**: $n^2 = o(n^3)$
- **Example**: f(n) is never o(f(n))

Measuring Complexity

- **Example**: $A = \{0^k1^k | \, k \geq 0\}$

M_1 = "On input string w:
1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

- Stage 1:
  - Scan the tape and go back: 2n = O(n)
- State 2 and 3:
  - At most n/2 scans, each takes O(n) steps: (n/2)O(n)=O(n^2)
- State 4:
  - One scan is sufficient to decide: O(n)
- The running time of the machine is O(n^2)

Measuring Complexity

Classification

- Definition:

Let $t\colon \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, $\mathbf{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

- Example: Consider the following language:

$$A = \{0^k 1^k \mid k \geq 0\}$$

  - $A \in \mathrm{TIME}(n^2)$

- We can decide A in O(n log n) with a better algorithm,
- We can decide A in O(n) with a two-tape TM.

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Measuring Complexity
Classification

- We can decide A in O(n log n) with a better algorithm,
- We can decide A in O(n) with a two-tape TM.

M_3 = "On input string w:

1. Scan across tape 1 and *reject* if a 0 is found to the right of a 1.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, *reject*.
4. If all the 0s have now been crossed off, *accept*. If any 0s remain, *reject*."

Measuring Complexity

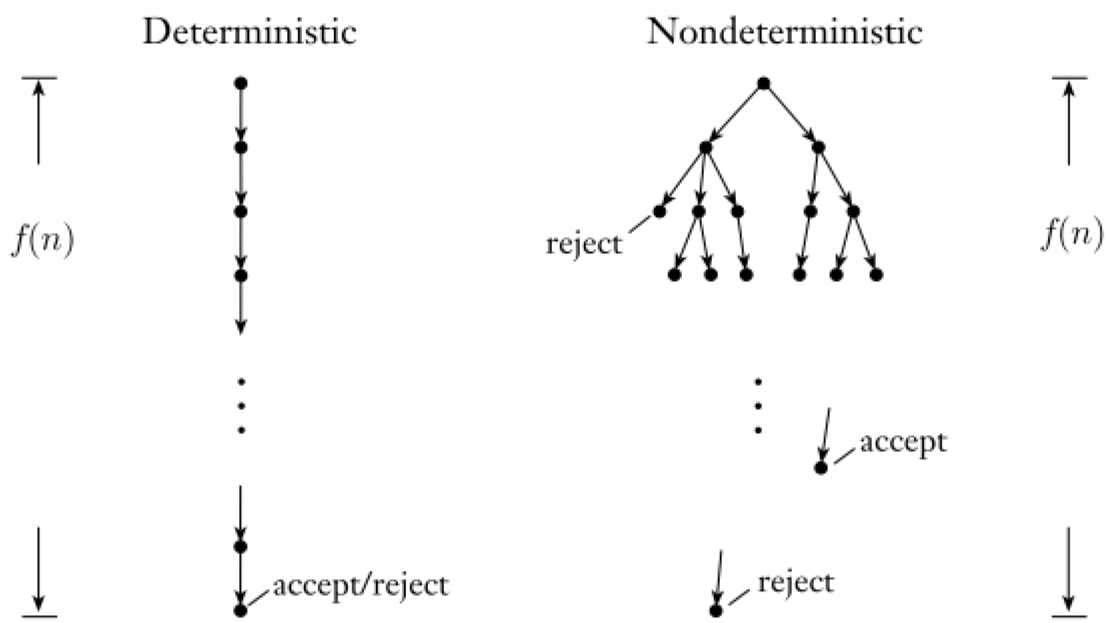
Classification

- We can decide A in O(n log n) with a better algorithm,
- We can decide A in O(n) with a two-tape TM.

- We learned that according to the Church-Turing thesis all reasonable models of computation are equivalent.
- However, in complexity theory, the choice of the model affects the time complexity of the language.
  - Single-tape TM,
  - Multitape TM,
  - Nondeterministic TM.

- Definition:

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.



Theorem

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.

- Proof Idea:
  - Previously we saw it is possible to simulate a multitape TM with a single-tape TM.
  - We analyze the simulation.
  - We show that the single-tape TM can simulate each step of the multitape TM with at most $O(t(n))$ steps.

Theorem

Let t(n) be a function, where $t(n) \geq n$. Then every t(n) time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.

- Proof:
  - Each step of the multitape TM:
    - Initialize the time: $O(n)$
    - Read symbols under all heads: $O(t(n))$
    - Move heads: $O(t(n))$
    - Update the tapes (possibly shift to right): $O(t(n))$
  - The entire simulation involves simulation of t(n) steps of the multitape TM:

$$t(n) \times O\big(t(n)\big) = O(t^2(n))$$

Theorem

Let t(n) be a function, where $t(n) \geq n$. Then every t(n) time nondeterministic single-tape TM has an equivalent $O(2^{O(t(n))})$ time deterministic single-tape TM.

- Proof:
  - N: A nondeterministic TM running in t(n).
  - D: a deterministic TM that simulates N.
  - b: maximum choice by N's transition function.
  - Simulation explores the computation branches of N in a breadth-first search manner.
  - The total number of nodes in the computation tree is bounded by $O(b^{t(n)})$. Reaching each node takes at most $O(t(n))$:
    $$O\left(t(n)b^{t(n)}\right) = 2^{O(t(n))}$$
  - Simulating the tree tapes at most squares the running time:
    $$\left(2^{O(t(n))}\right)^2 = 2^{O(2t(n))} = 2^{O(t(n))}$$

Measuring Complexity

The Class P

- We observe an exponential difference between the time complexity of problems on deterministic and nondeterministic TMs.

- Polynomial difference is small.
- Exponential difference is large.
  - n=1000
  - n^3 is one billion
  - 2^n larger than the number of atoms in the universe
- Exponential time algorithms rarely are useful:
  - brute-force search

Measuring Complexity
The Class P

- **All reasonable deterministic computational models are polynomially equivalent.**

- We focus on fundamental properties of computation:
  - Aspects of time complexity theory that are unaffected by polynomial differences in running time.
  - Develop a theory that doesn't depend on the selection of a particular model of computation.

Measuring Complexity

The Class P

- Definition:

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

- P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine,
  - P is a mathematically robust class.
- P roughly corresponds to the class of problems that are realistically solvable on a computer.

Theorem
PATH $\in$ P.

- Brute force: Examining all potential paths in a graph with m nodes is not efficient. Number of all paths is roughly m^m.
- **Proof**: A polynomial time algorithm:

$M =$ "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

- Stage 1 and 4 are executed once.
- Stage 3 is executed m times (each time one node is marked).
- Each stage can be implemented in polynomial time.