www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 17(1), Jan-April 2026, 299-306. ISSN: 2007-1558. https://doi.org/10.61467/2007.1558.2026.v17i1.1172

Design of a recursive algorithm for DFA implementation: computational complexity analysis

Ashley Torres-Pérez, Eduardo Cornejo-Velázquez [2], Mireya Clavel-Maqueda[2]
[1] Instituto de Ciencias Básicas e Ingeniería, Universidad Autónoma del Estado de Hidalgo
[2] Universidad Autónoma del Estado de Hidalgo.

E-mails: to476602@uaeh.edu.mx, ecornejo@uaeh.edu.mx, Mireya Clavel-Maqueda

Abstract. Computational complexity, in its practical application, quantifies the time and memory space required for an algorithm's execution, thereby allowing the assessment of its efficiency and limitations with respect to input size. In this work, the design of a recursive algorithm for implementing a deterministic finite automaton (DFA) to recognise regular languages is presented. The divide-and-conquer technique was employed to select appropriate data structures and to define a recursive function for resolving the transition functions between states. The recursive algorithm was implemented in the C++ language and exhibits linear temporal and spatial computational complexity, which is intended to support different configurations for the evaluation of multiple input strings. The proposed algorithm is consistent with the theoretical definitions, and the computational complexity analysis provides support for its classification based on the input parameters.
Keywords: Computational complexity, DFA, programming.

Article Info
Received July 8, 2025
Accepted July 10, 2025

1 Introduction

Computational complexity focuses on the study of the intrinsic difficulty of computational problems and their classification according to the resources needed to solve them: time and memory. It is fundamental to understand the limits of what computers can and cannot do (Dean, 2016).

The concept of computational complexity arises from Alan Turing's work in 1936, in which he proposed the Turing machine as a theoretical model of computation that made it possible to formally define which problems are computable or decidable (Cook, 1983). In 1965, Hartmanis and Stearns laid the theoretical foundation of computational complexity by introducing the idea of measuring the time and space required by an algorithm as a function of the size of the input (Fortnow & Homer, 2014).

The importance of computational complexity lies in its practical application to evaluate the efficiency and limitations of algorithms. Complexity theory has influenced various areas of technology and the sciences, especially with the concepts of feasibility and in the distinction between solving and verifying solutions (Dean, 2019).

For the purpose of measuring the performance or behavior of a computational program, time and space complexity have been defined as fundamental measures to evaluate the efficient use of computational resources (Gnatenko et al., 2024).

Time complexity refers to the number of steps or the amount of time an algorithm takes to complete as a function of the input size. This measure helps determine the degree of scalability and efficiency of an algorithm for large inputs (Mohan, 2019).

While space complexity refers to the amount of memory or storage, an algorithm needs to be used relative to the input size. It is of utmost significance to understand resource requirements and feasibility in memory constrained environments (Hashimoto, et al., 2017).

Low space complexity does not necessarily imply low time complexity, and vice versa. There are problems that can be solved quickly, but require a lot of memory, and others that use little memory, but take a long time to solve. Therefore, computational complexity is essential for analyzing tradeoffs between memory and time efficiency (Carl, 2019; Latkin, 2023).
In the deterministic Turing machine model, it has been shown that deterministic exponential time and deterministic polynomial space can coincide with certain problems (Latkin, 2023). In continuous-time models of computation, spatial complexity may correspond to the required accuracy of computations, linking memory usage to numerical stability (Blanc & Bournez, 2024).
In engineering and physical systems, the concepts of spatial and time complexity help to describe predictability and loss of information, especially in systems with spatial and time dynamics (Schertzer & Lovejoy, 2004).

To evaluate the efficiency of algorithms and understand computational limits, asymptotic analysis is used to evaluate the behavior of an algorithm as the input size tends to infinity (Karp, 1972). Big O notation is used to determine how an algorithm behaves in execution time and memory usage in the worst-case situation with respect to input size. This analysis allows the design of algorithms that achieve a balance between speed and resource efficiency (Zhang, 2022).

On the other hand, automata theory and computational complexity are closely linked because the former provides the conceptual framework for understanding computational problems and how they can be solved with abstract machines, while the latter analyzes the efficiency and bounds of these problems (Kozen, 1997).

The simplest model is a finite state machine or finite automaton which is described as a computer with a very limited amount of memory (Sisper, 2006).

Finite automata can be classified as deterministic and nondeterministic. A deterministic finite automaton (DFA) is a mathematical model of computation consisting of a finite set of states and well-defined rules for transiting between them based on an input. In each state and for each input symbol, there is only one possible transition, and therefore, there is no ambiguity in the execution. Given an input string, the automaton follows a single path in its state diagram.

In this paper we address the implementation of a DFA as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ that recognizes regular languages. Where Q is a finite set of states, Σ is a finite set of characters or symbols called the alphabet, δ are transition functions of the form $Q\ x\ \Sigma \rightarrow$ Q, q_0 is the start state and F is the set of acceptance states or final states such that $F \subseteq Q$ (Sisper, 2006).

2 Experimental procedures

Let M be a DFA defined by $M = (Q, \Sigma, \delta, q_0, F)$ and a string of symbols $w = w_0, w_1, w_2, \cdots, w_n$ where $w_i \in \Sigma$, then M accepts w if there is a sequence of states $r_0, r_1, r_2, \cdots, r_n$ such that $r_i \in Q$ and the following conditions are satisfied:

1. $r_0 = q_0$
2. $\delta(r_i,\ w_{i+1}) = r_{i+1} \mid 0 \leq i \leq n - 1$
3. $r_n \subseteq F$

The DFA recognizes a language A if it accepts all the words w that belong to that language. The purpose is to design and implement an algorithm which, given a language A and an automaton M, determine whether a word $w \in A$ is accepted or rejected by M.
For the development of the following sections, we consider the case of DFA defined by $M = (Q, \Sigma, \delta, q_0, F)$ with $Q = \{1, 2, 3, 4\}$, $\Sigma = \{a, b, c\}$, $F = \{2, 4\}$, $q_0 = 1$ and $\delta(I, X) = J\ where\ I, J \in Q$ and $X \in \Sigma$ with transitions:
$\delta(1, a) = 1;\ \delta(1, b) = 3;\ \delta(1, c) = 3;\ \delta(2, a) = 2;\ \delta(2, b) = 3;\ \delta(2, c) = 2;\ \delta(3,\ a) = 3;\ \delta(3, b) = 4; \delta(3, c) = 4;\ \delta(4, a) = 4;\ \delta(4, b) = 4;\ \delta(4, c) = 2$. Fig. 1. shows the state diagram for M.

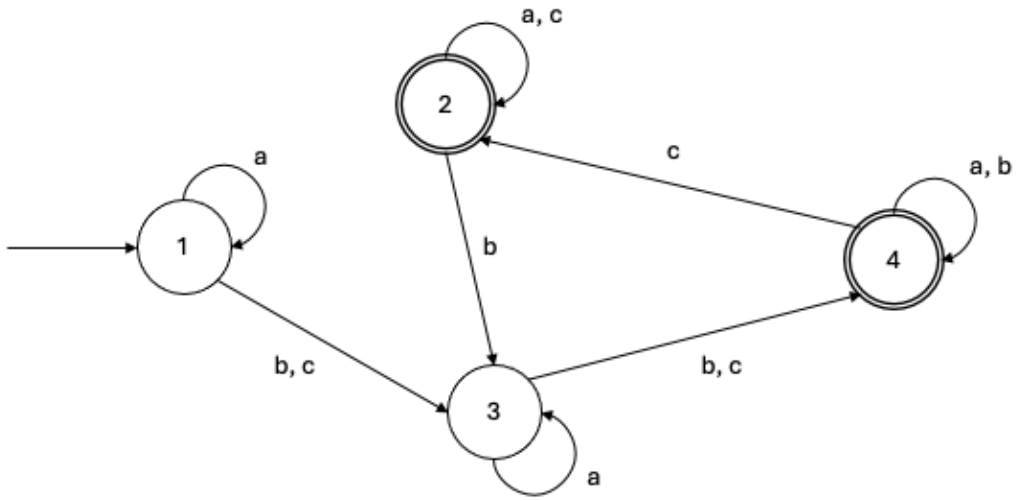

Fig. 1. State diagram of DFA M.

For the purpose of choosing the appropriate data structures for the DFA in the C++ programming language, the following analysis was performed:

- The automaton M has a set of states Q of size N and one or more of these are defined as end states F. A vector<bool> was used to represent this set, where the index i corresponds to a state in Q and the value stored at each position indicates whether the state is final (true) or not (false) as presented in Figure 2a. This data structure simplifies the verification of the acceptance states for an input sequence: the sequence will be valid if the last state reached is a final state marked with a true value. By definition, the automaton M has exactly one initial state q_0 and is represented by a single integer variable.
- For state transitions defined by function $\delta(I, X) = J$ where $I, J \in Q$ and $X \in \Sigma$, a dynamic linear data structure storing hash tables was adopted. A dynamic linear data structure that stores hash tables was implemented using a vector<unordered_map<char, int>> structure to store all state transitions, where I from the transition function corresponds to the index i of the vector, X is the key of the hash table unordered_map at the $i - th$ index and J is the integer hash table value that represents the state to which the automaton moves when it has as input the X character. The chosen data structure is described in Fig. 2b.

a. vector<bool> Q

i	1	2	3	4
boolean	0	1	0	1

b. vector<unordered_map<char,int>> T

i	1			2			3			4		
hash table	key	a	b	c	key	a	b	c	key	a	b	c	key	a	b	c
	value	1	3	3	value	2	3	2	value	3	4	4	value	4	4	2

Fig. 2. Data structures for Q states and δ transition functions.

For the design of the algorithm to verify whether the string $W =$ "abaacbac" is accepted by the automaton M the divide-and-conquer strategy was adopted to split the problem into smaller, more manageable subproblems, then solve the subproblems recursively and combine the solutions to create a final solution to the original problem (Cormen et al., 2022).

For the transition function $\delta(I, X) = J$ is defined a recursive function $f(n)$ for a $k < n$, in accordance with Koshy (2004), to ensure the convergence of the function. The following steps were defined:

Step 1. Initial call to the recursive function $f\big(\delta(I, X)\big) = J$ with initial state $q_0 = 1$ for $w_0 = \text{a}$, is used $f(s, n)$ where $s \in Q$ and n is the position of the symbol in the string W. $f(1, 0) = 1$ is a valid transition.

Step 2 For each valid transition, a new call is made to the recursive function with $n + 1$. Recursive calls continue as long as valid transitions are encountered and $n < |W|$. When n is equivalent to the size of string W, a base case is called to halt the recursive calls and begin returning the solution for each subproblem created. For the chain W the sequence is built $f(1, 0) \rightarrow f(1, 1) \rightarrow f(3, 2) \rightarrow f(3, 3) \rightarrow f(3, 4) \rightarrow f(4, 5) \rightarrow f(4, 6) \rightarrow f(2, 7) \rightarrow f(2, 8)$.

Step 3. The base case is reached when $n = |W|$, then it is checked whether the state reached corresponds to an acceptance state. If $s \in F$ the function returns true. For each call on the recursion stack the function returns true backward within the sequence $f(2, 8) \rightarrow f(2, 7) \rightarrow f(4, 6) \rightarrow f(4, 5) \rightarrow f(3, 4) \rightarrow f(3, 3) \rightarrow f(3, 2) \rightarrow f(1, 1) \rightarrow f(1, 0)$. The initial call $f(1, 0)$ receives true, which leads to the conclusion that the string W is accepted by the automaton M.

Note that the recursive chain follows a single path that coincides with the behavior of deterministic computation (Sisper, 2006).

Fig. 3. shows the recursion graph of the proposed function, the process of recursive calls until reaching the base case, and the return value in the recursion stack.

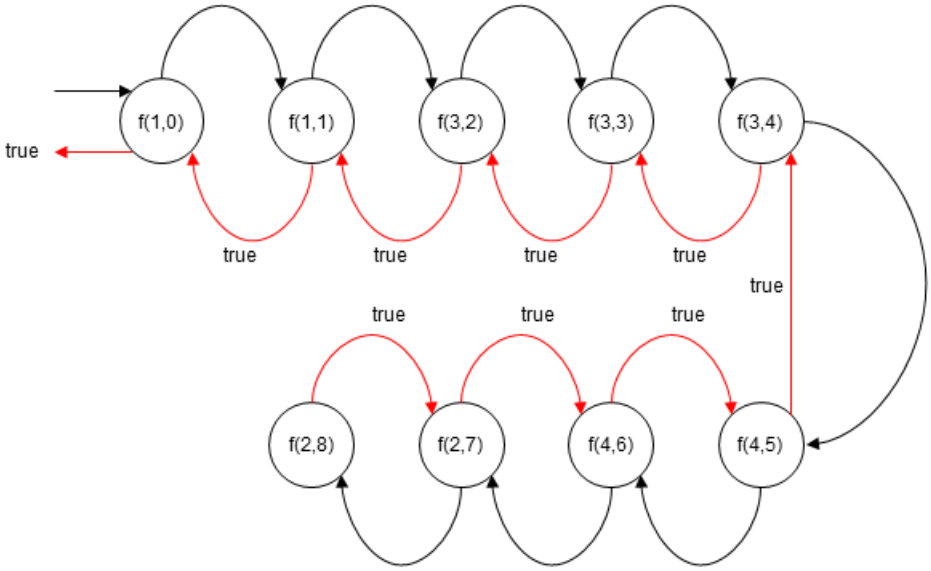

Fig. 3. Recursion graph of $f(s, n)$.

3 Results

The pseudocode of the algorithm to implement the designed recursive function is presented in Figure 4. In line 2 the first base case is considered, when there are no more symbols left in the string W to process, return the boolean value corresponding to current state $Q[s]$. Line 5 includes the second base case: if there is no valid transition function in the current state s for symbol $W[n]$, return false. The recursive case is presented in line 8, the recursive call is made for the target state $T[s][W[n]]$ with the symbol $W[n + 1]$, this call is repeated until one of the base cases is reached.

```
Input: T, s, W, n, Q
Output: true or false
1  function verify()
2     if n == |W| then
3         return Q[s]
4     end if
```

```
5     if T[s].find(W[n]) == false then
6         return false
7     end if
8     return verify(T, T[s][W[n]], W, n + 1, Q)
9  end function
```

Fig. 4. Pseudocode for the recursive function `verify()`.

The implemented recursive function verify() is shown in Fig. 5, which uses the parameters: vector <unordered_map<char, int>>& T, string W and vector<bool> Q to establish the base cases and to perform simple comparison operations. While int wIdx corresponds to the $n-th$ position in string W and int currentState is the current state s.

```
1  bool verify(vector<unordered_map<char, int>>& T, int currentState, string W,
   int wIdx, vector<bool> Q) {
2        if(wIdx == W.size()) return Q[currentState];
3        if(T[currentState].find(W[wIdx]) == T[currentState].end()) return false;
4        return verify(T, T[currentState][W[wIdx]], W, wIdx+1, Q);
5  }
```

Fig. 5. Recursive function to verify the string W.

In line 2 the first base case is considered to establish that the recursion should stop when wIdx == W.size() and return the boolean value stored in Q[currentSt]. Line 3 implements the second base case, using the function .find() of the data structure unordered map to search for the key W[wIdx] on the map T[currentState]. If not found, .find() returns T[currentState].end(), and then the recursion stops with a returned value of false at the bottom of the call stack. The recursive case is presented in line 4: the recursive call is made for the destination state T[currentState][W[wIdx]] with the symbol wIdx + 1. The call is repeated until a base case is reached. The recursive function verify() corresponding to $f(s,n)$ recognizes the following cases:

$$f(s,n) = \begin{cases} \textbf{\textit{false}} \text{ if } n = |W| \text{ and s is not a final state} \\ \textbf{\textit{true}} \text{ if } n = |W| \text{ and s is a final state} \\ \textbf{\textit{true}} \text{ if there exists a valid transition from } s \text{ the next state with the character in the } nth \text{ position} \\ \boldsymbol{f(\delta(s, W(n)), n+1)} \text{ if there is a valid transition and recursion is made with the following symbol} \\ \textbf{\textit{false}} \text{ if there is no valid transition for the symbol at } n \text{ position} \end{cases}$$

Within the main function main() the input from the terminal of the automaton parameters is considered: size N of Q, D number of transition functions, initial state e and number of end F states, with them the input of values for the structures vector<bool> Q and vector <unordered_map<char, int>> T is performed as shown in Fig. 6.

```
1  vector<bool> Q(N + 1, false);
2  for (int i = 1; i <= F; i++) {
3        int s;
4        cin >> s;
5        Q[s] = true;
6  }
7  vector<unordered_map<char, int>> T(D);
8  int I, J;
9  char X;
10 for (int i = 1; i <= D; i++) {
11       cin >> I >> X >> J;
12       T[I][X] = J;
```

```
13  }
14  string W;
15  getline(cin, W);
```

Fig. 6. Code to load parameters and values of the automaton.

In line 1 the data structure vector *<bool> Q(N, false)* is declared to store the number N of states of the automaton and all of them are initialized to false. The for-cycle from lines 2 to 6 is used to enter the number F of final states, and in line 5 for each state the value is changed to true in the structure.

In line 7, the data structure vector <unordered_map<char, int>> T(D) is declared to store the number D of transition functions. The loop for from lines 10 to 13 is used to enter the key-value pairs corresponding to each transition function using the variables declared in lines 8 and 9. In line 15, the string W is read using a getline(), which is used in consideration that the input W may be an empty string.

For the complete implementation of the automaton $M = (Q, \Sigma, \delta, q_0, F)$, limiting cases are considered for the proposed algorithm based on the following constraints $1 \leq |Q|, |\Sigma| \leq 100, 1 \leq |\delta| \leq 10^4, 1 \leq q_0 \leq |Q|, 0 \leq |F| \leq |Q|, 0 \leq |W| \leq 100$. This is done considering extreme value inputs in the expected ranges (boundary values) or unusual but valid ones.
The constraints explicitly state that the algorithm must be capable of processing the empty string $|W| = 0$ and the empty set of final states $|F| = 0$. The implemented code is presented in Fig. 7.

```
1  if (F == 0) {
2      cout << "REJECTED\n";
3  }
4  else {
5       if (Q[e] && W=="") {
6            cout << "ACCEPTED\n";
7       }
8       else {
9            if (verify(T, e, W, 0, Q)) cout <<" ACCEPTED \n";
10           else cout <<" REJECTED \n";
11      }
12 }
```

Fig. 7. Implementation of special constraints.

Line 1 considers the limiting case $|F| == 0$, by definition, if there are no acceptance states in the automaton, no string entered will be accepted. If the condition F == 0 any input string is rejected, and the program terminates. Otherwise, line 5 checks whether the initial e state is also an end state and the limiting case where W is an empty string; if the condition is met, then the input string is accepted.

If none of the limiting cases occur, then in line 9 the recursive function is called verify(T, e, W, 0, Q) with the initial parameters for the symbol $W[0]$. If the recursive function returns true, string W is accepted; otherwise, it is rejected.

For the time complexity analysis, the input parameters for the automaton are considered: N Q states, D transition functions, F final states and the chain W of length L. Based on these values it can be established that N + D + F + L operations are performed. According to the defined restrictions, at most 100 + 104 + 100 + 100 operations are performed, which can be approximated to 104. Therefore, it is concluded that the algorithm has a linear time complexity $\mathcal{O}(N)$.

On the other hand, the data structures used, vector<bool> and vector<unordered map<char, int>>, have constant time complexity $\mathcal{O}(1)$ for search, insertion and deletion operations (GeeksforGeeks, 2024).

For processing the string W one symbol at a time is considered, therefore, a maximum of $|W|$ recursive calls are made. The verify() performs only one operation, either base case or recursion; therefore, to process the entire string, L operations are performed. The time complexity of the recursive function is constant $\mathcal{O}(L)$.

In the case of spatial complexity, the algorithm uses fixed variables that do not change their value during its execution, occupying a constant memory space $\mathcal{O}(1)$. For the case of vector<bool> and vector<unordered map<char, int>> data structures, the number of elements depends on the number N of Q states and the number D of transition functions, respectively. Based on the defined constraints, the maximum values for the structures are 100 and 104 memory spaces. Both structures have linear space complexity $\mathcal{O}(N)$.

The recursive function each time it calls itself needs memory space for the recursion stack until it reaches the size L of $|W|$. The constraint defined for the maximum size of the memory space is 100, so the function has a linear space complexity $\mathcal{O}(L)$. Based on the analyses performed, it is concluded that the proposed algorithm has a linear time complexity $\mathcal{O}(N)$, as well as linear space complexity $\mathcal{O}(N)$.

Finally, in order to evaluate the proposed algorithm, the omegaUp competitive programming platform was used to solve the challenge 15320. Simulation of a Deterministic Finite Automaton (omegaUp, 2025) with the constraints defined in the virtual judge with time limit (case) 1 second, input limit 10Kb, memory limit 32 Mb and time limit (total) 1 minute 0 seconds. The result of the evaluation is presented in Fig. 8.

Submissions

Date and Time	Language	Percentage	Execution	Output	Memory	Runtime	Actions
2025-02-23 17:38	cpp11-gcc	100.00%	Finished	Correct	3.37 MB	0.01 s	

Fig. 8. OmegaUp virtual judge results for the proposed algorithm.

The proposed algorithm achieved 100% of the challenge score, all outputs for the test cases were correct, it occupied 3.37 MB of memory, and the execution time was 0.01 seconds.

4 Conclusions

Finite automata are fundamental tools in the theory of computation and have practical applications in the lexical analysis phase in compilers, as well as in pattern recognition in regular expressions. By definition, a deterministic finite automaton (DFA) consists of a set of states, an alphabet, a set of transition functions, an initial state and a set of final states. In this work, we designed and implemented a recursive algorithm capable of simulating a deterministic finite automaton (DFA) by applying the divide-and-conquer technique to select the appropriate data structures and define a recursive transition function. This approach allows the DFA to be fully configurable based on the input parameters, allowing it to support different automaton structures and to evaluate multiple input strings.

The implementation in C++ demonstrated that the recursive strategy complies with the theoretical definitions of DFAs while maintaining predictable behavior during state transitions. A DFA in each state has exactly one transition function for each input symbol, which ensures a unique and predictable path. The proposed recursive algorithm succeeded in implementing a DFA that can be configured based on the input parameters, and the evaluation performed on the omegaUp platform by its virtual judge verified the algorithm's correct functionality in a practical setting, reinforcing its applicability for solving problems involving regular language recognition.

The computational complexity analysis shows that the algorithm achieves linear time complexity $\mathcal{O}(N)$, with respect to the number of states, number of transition functions, number of accepting states, length of the input string, and the recursive call stack. Each symbol is processed in constant time, ensuring that execution grows proportionally with the size of the input. Likewise, the space complexity remains linear O(N) due to the data structures employed and the depth of the recursion stack.

The computational complexity analysis of the algorithm was based on processing the input string, symbol by symbol, changing from one state to another according to the transition function. For each symbol, an operation is performed that takes constant time. Although different DFA designs may vary in the number of states or the behavior of transitions defined by the transition function,

the overall behavior of the algorithm remains linear as long as the transitions are processed symbol by symbol. This implies that a practical implication such as lexical analysis can be handled by the recursive design with $\mathcal{O}(N)$ complexity.
The linear computational complexity of DFAs makes them very efficient for processing regular languages. The time required by the DFA to process the input string will always be proportional to its length.

Overall, the time and spatial complexity analysis demonstrate that the proposed recursive DFA algorithm is both theoretically sound and practically efficient. Its linear computational complexity makes it suitable for scalable applications involving regular languages and emphasizes the effectiveness of combining divide-and-conquer techniques with recursive design in automata processing.

References

Blanc, M., & Bournez, O. (2024). *The complexity of computing in continuous time: Space complexity is precision*. *arXiv*. https://doi.org/10.48550/arXiv.2403.02499

Carl, M. (2019). *Space and time complexity for infinite time Turing machines*. *Journal of Logic and Computation, 30*, 1239–1255. https://doi.org/10.1093/logcom/exaa025

Cook, S. (1983). *An overview of computational complexity*. *Communications of the ACM, 26*, 400–408. https://doi.org/10.1145/358141.358144

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.

Dean, W. (2019). *Computational complexity theory and the philosophy of mathematics*. *Philosophia Mathematica*. https://doi.org/10.1093/philmat/nkz021

Dean, W. (2021). *Computational complexity theory*. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Fall 2021 ed.). https://plato.stanford.edu/archives/fall2021/entries/computational-complexity/

Fortnow, L., & Homer, S. (2014). *Computational complexity*. In *Handbook of the History of Logic* (Vol. 9, pp. 495–521). North-Holland.

GeeksforGeeks. (2024). *unordered_map::find() in C++ STL*. https://www.geeksforgeeks.org/unordered-mapfind-in-c-stl/

Gnatenko, A., Kutz, O., & Troquard, N. (2024). *Building an ontology of computational complexity*. In *Proceedings of the Joint Ontology Workshops (JOWO) – Episode X* (FOIS 2024). CEUR-WS. https://ceur-ws.org/Vol-3882/foust-9.pdf

Hashimoto, K., Iizuka, N., & Sugishita, S. (2017). *Time evolution of complexity in Abelian gauge theories*. *Physical Review D, 96*, 126001. https://doi.org/10.1103/PhysRevD.96.126001

Karp, R. M. (2021). *Reducibility among combinatorial problems*. In H. R. Lewis (Ed.), *Ideas that created the future: Classic papers of computer science* (pp. 141–156). MIT Press. https://doi.org/10.7551/mitpress/12274.003.0038

Koshy, T. (2004). *Discrete mathematics with applications*. Elsevier Science & Technology.

Kozen, D. C. (1997). *Automata and computability*. Springer.

Latkin, I. (2023). *A correspondence between the time and space complexity*. *arXiv*. https://doi.org/10.48550/arXiv.2311.01184

Mohan, N. (2019). *Understanding time and space complexity in algorithms*. *International Journal of Science and Research (IJSR)*. https://doi.org/10.21275/sr24923134130

OmegaUp. (2025). *15320. Simulación de un autómata finito determinista*. https://omegaup.com/arena/problem/Simulacion-de-DFA/

Schertzer, D., & Lovejoy, S. (2004). *Space–time complexity and multifractal predictability*. *Physica A: Statistical Mechanics and Its Applications, 338*, 173–186. https://doi.org/10.1016/j.physa.2004.04.032

Sipser, M. (2006). *Introduction to the theory of computation* (2nd ed.). MIT Press.

Zhang, N. (2022). *50 years of computational complexity: Hao Wang and the theory of computation*. *arXiv*. https://doi.org/10.48550/arXiv.2206.05274