

[Open Peer Review on Qeios](#)

# NP on Logarithmic Space

Frank Vega

**Funding:** No specific funding was received for this work.

**Potential competing interests:** No potential competing interests to declare.

## Abstract

P versus NP is considered as one of the most important open problems in computer science. This consists in knowing the answer of the following question: Is P equal to NP? It was essentially mentioned in 1955 from a letter written by John Nash to the United States National Security Agency. However, a precise statement of the P versus NP problem was introduced independently by Stephen Cook and Leonid Levin. Since that date, all efforts to find a proof for this problem have failed. Another major complexity classes are L and NL. Whether  $L = NL$  is another fundamental question that it is as important as it is unresolved. We prove the breakthrough result that  $L = NL$ . Besides, we show that every NP problem is in L with oracle access to L.

## Frank Vega

*Research Department, NataSquad, 10 rue de la Paix, Paris, 75002, France*

[vega.frank@gmail.com](mailto:vega.frank@gmail.com)

**2012 ACM Subject Classification:** Theory of computation → Complexity classes; Theory of computation → Problems, reductions and completeness; Theory of computation → Abstract machines.

**Keywords:** Complexity Classes, Completeness, Polynomial Time, Reduction, Logarithmic Space.

## 1. Introduction

In 1936, Turing developed his theoretical computational model<sup>[1]</sup>. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation<sup>[1]</sup>. A deterministic Turing machine has only one next action for each step defined in its program or transition function<sup>[1]</sup>. A nondeterministic Turing machine could contain more than one action defined for each step of its program, where this one is no longer a function, but a relation<sup>[1]</sup>.

Let  $\Sigma$  be a finite alphabet with at least two elements, and let  $\Sigma^*$  be the set of finite strings over  $\Sigma$ <sup>[2]</sup>. A Turing machine  $M$

has an associated input alphabet  $\Sigma$  [2]. For each string  $w$  in  $\Sigma^*$  there is a computation associated with  $M$  on input  $w$  [2]. We say that  $M$  accepts  $w$  if this computation terminates in the accepting state, that is  $M(w) = \text{"yes"}$  [2]. Note that,  $M$  fails to accept  $w$  either if this computation ends in the rejecting state, that is  $M(w) = \text{"no"}$ , or if the computation fails to terminate, or the computation ends in the halting state with some output, that is  $M(w) = y$  (when  $M$  outputs the string  $y$  on the input  $w$ ) [2].

Another relevant advance in the last century has been the definition of a complexity class. A language over an alphabet is any set of strings made up of symbols from that alphabet [3]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [3]. The language accepted by a Turing machine  $M$ , denoted  $L(M)$ , has an associated alphabet  $\Sigma$  and is defined by:

$$L(M) = \{w \in \Sigma^* : M(w) = \text{"yes"}\}.$$

Moreover,  $L(M)$  is decided by  $M$ , when  $w \notin L(M)$  if and only if  $M(w) = \text{"no"}$  [3]. We denote by  $t_M(w)$  the number of steps in the computation of  $M$  on input  $w$  [2]. For  $n \in \mathbb{N}$  we denote by  $T_M(n)$  the worst case run time of  $M$ ; that is:

$$T_M(n) = \max\{t_M(w) : w \in \Sigma^n\}$$

where  $\Sigma^n$  is the set of all strings over  $\Sigma$  of length  $n$  [2]. We say that  $M$  runs in polynomial time if there is a constant  $k$  such that for all  $n$ ,  $T_M(n) \leq n^k + k$  [2]. In other words, this means the language  $L(M)$  can be decided by the Turing machine  $M$  in polynomial time. Therefore,  $P$  is the complexity class of languages that can be decided by deterministic Turing machines in polynomial time [3]. A verifier for a language  $L_1$  is a deterministic Turing machine  $M$ , where:

$$L_1 = \{w : M(w, u) = \text{"yes"} \text{ for some string } u\}.$$

We measure the time of a verifier only in terms of the length of  $w$ , so a polynomial time verifier runs in polynomial time in the length of  $w$  [2]. A verifier uses additional information, represented by the string  $u$ , to verify that a string  $w$  is a member of  $L_1$ . This information is called certificate.  $NP$  is the complexity class of languages defined by polynomial time verifiers [4].

It is fully expected that  $P \neq NP$  [4]. Indeed, if  $P = NP$  then there are stunning practical consequences [4]. For that reason,  $P = NP$  is considered as a very unlikely event [4]. Certainly,  $P$  versus  $NP$  is one of the greatest open problems in science and a correct solution for this incognita will have a great impact not only in computer science, but for many other fields as well [5]. Whether  $P = NP$  or not is still a controversial and unsolved problem [6]. We provide some results in order to understand better this outstanding problem in computer science.

## 1.1. The Hypothesis

A function  $f: \Sigma^* \rightarrow \Sigma^*$  is a polynomial time computable function if some deterministic Turing machine  $M$ , on every input  $w$ , halts in polynomial time with just  $f(w)$  on its tape [1]. Let  $\{0, 1\}^*$  be the infinite set of binary strings, we say that a language  $L_1 \subseteq \{0, 1\}^*$  is polynomial time reducible to a language  $L_2 \subseteq \{0, 1\}^*$ , written  $L_1 \leq_p L_2$ , if there is a polynomial time computable function  $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$  such that for all  $x \in \{0, 1\}^*$ :

$$x \in L_1 \text{ if and only if } f(x) \in L_2.$$

An important complexity class is NP--complete [7]. If  $L_1$  is a language such that  $L' \leq_p L_1$  for some  $L' \in \text{NP--complete}$ , then  $L_1$  is NP--hard [3]. Moreover, if  $L_1 \in \text{NP}$ , then  $L_1 \in \text{NP--complete}$  [3]. A principal NP--complete problem is SAT [7]. An instance of SAT is a Boolean formula  $\phi$  which is composed of:

1. Boolean variables:  $x_1, x_2, \dots, x_n$ ;
2. Boolean connectives: Any Boolean function with one or two inputs and one output, such as  $\wedge$  (AND),  $\vee$  (OR),  $\neg$  (NOT),  $\Rightarrow$  (implication),  $\Leftrightarrow$  (if and only if);
3. and parentheses.

A truth assignment for a Boolean formula  $\phi$  is a set of values for the variables in  $\phi$ . A satisfying truth assignment is a truth assignment that causes  $\phi$  to be evaluated as true. A Boolean formula with a satisfying truth assignment is satisfiable. The problem SAT asks whether a given Boolean formula is satisfiable [7]. We define a *CNF* Boolean formula using the following terms:

A literal in a Boolean formula is an occurrence of a variable or its negation [3]. A Boolean formula is in conjunctive normal form, or *CNF*, if it is expressed as an AND of clauses, each of which is the OR of one or more literals [3]. A Boolean formula is in 2-conjunctive normal form or *2CNF*, if each clause has exactly two distinct literals [3]. For example, the Boolean formula:

$$(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2) \wedge (\neg x_1 \vee \neg x_3)$$

is in *2CNF*. The first of its three clauses is  $(x_1 \vee \neg x_2)$ , which contains the two literals  $x_1$ , and  $\neg x_2$ .

A logarithmic space Turing machine has a read-only input tape, a write-only output tape, and read/write work tapes [1]. The work tapes may contain at most  $O(\log n)$  symbols [1]. In computational complexity theory,  $L$  is the complexity class containing those decision problems that can be decided by a deterministic logarithmic space Turing machine [4].  $NL$  is the complexity class containing the decision problems that can be decided by a nondeterministic logarithmic space Turing machine [4]. The complexity class  $coNL$  can be defined as the set of languages such that every element inside of the language will be accepted for every possible path by a nondeterministic logarithmic space Turing machine [4].

A function  $f: \Sigma^* \rightarrow \Sigma^*$  is a logarithmic space computable function if some deterministic Turing machine  $M$ , on every input  $w$ , halts using logarithmic space in its work tapes with just  $f(w)$  on its output tape [1]. Let  $\{0, 1\}^*$  be the infinite set of binary strings, we say that a language  $L_1 \subseteq \{0, 1\}^*$  is logarithmic space reducible to a language  $L_2 \subseteq \{0, 1\}^*$ , written  $L_1 \leq_l L_2$ , if there is a logarithmic space computable function  $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$  such that for all  $x \in \{0, 1\}^*$ :

$$x \in L_1 \text{ if and only if } f(x) \in L_2.$$

The logarithmic space reduction is used for the completeness of the complexity classes  $L$ ,  $NL$  and  $P$  among others.

The two-way Turing machines may move their head on the input tape into two-way (left and right directions) while the one-way Turing machines are not allowed to move the input head on the input tape to the left [8]. Hartmanis and Mahaney have investigated the classes  $1L$  and  $1NL$  of languages recognizable by deterministic one-way logarithmic space Turing

machine and nondeterministic one-way logarithmic space Turing machine, respectively [8]. They have shown that  $1L \neq 1NL$  (by looking at a uniform variant of the string non-equality problem from communication complexity theory) and have defined a natural complete problem for  $1NL$  under deterministic one-way logarithmic space reductions [8]. Furthermore, they have proven that  $1NL \subseteq L$  if and only if  $L = NL$  [8].

We can give a certificate-based definition for  $NL$  [2]. The certificate-based definition of  $NL$  assumes that a logarithmic space Turing machine has another separated read-only tape, that is called "read-once", where the head never moves to the left on that special tape [2].

**Definition 1.** A language  $L_1$  is in  $NL$  if there exists a deterministic logarithmic space Turing machine  $M$  with an additional special read-once input tape polynomial  $p: \mathbb{N} \rightarrow \mathbb{N}$  such that for every  $x \in \{0, 1\}^*$  :

$$x \in L_1 \Leftrightarrow \exists u \in \{0, 1\}^{p(|x|)} \text{ then } M(x, u) = \text{"yes"}$$

where by  $M(x, u)$  we denote the computation of  $M$ ,  $x$  is placed on its input tape, the certificate string  $u$  is placed on its special read-once tape, and  $M$  uses at most  $O(\log |x|)$  space on its read/write tapes for every input  $x$  where  $|\dots|$  is the bit-length function. The Turing machine  $M$  is called a logarithmic space verifier.

An oracle Turing Machine  $M$  has an additional tape, the oracle tape, and three states  $q_?$ ,  $q_{yes}$  and  $q_{no}$  [9]. When  $M$  enters  $q_?$  ( $M$  is said to query the oracle), then  $M$  goes to the state  $q_{yes}$  or the state  $q_{no}$  according to whether the string written in the oracle tape belongs or does not belong to a set called the oracle [9]. A language accepted by an oracle Turing Machine  $M$  with oracle  $A$  is denoted by  $L^A(M)$  [9]. The class of languages accepted by deterministic and nondeterministic oracle Turing Machine  $M$  working in space  $S(n)$ , with oracle  $A$ , is denoted by  $DSPACE^A(S(n))$  and  $NSPACE^A(S(n))$ , respectively [9]. In this definition, we bound the space of the oracle tape by a space  $2^{O(S(n))}$  [9]. A nondeterministic oracle Turing machine can query  $2^{2^{O(S(n))}}$  strings in the tree of all possible computations [9].

There is another definition such that the oracle tape is not space-bounded and the machine works deterministically from the time it begins to write on the oracle tape [9]. The complexity classes  $DSPACE^{(A)}(S(n))$  and  $NSPACE^{(A)}(S(n))$  are the respective complexity classes based on this definition on an oracle  $A$  [9]. It is trivial to see that  $DSPACE^{(A)}(S(n)) = DSPACE^A(S(n))$  [9]. Moreover,  $L = NL$  if and only if

- $\forall S(n) \forall A DSPACE^A(S(n)) = NSPACE^A(S(n))$
- and  $\forall S(n) \forall A DSPACE^{(A)}(S(n)) = NSPACE^{(A)}(S(n))$

for space constructible  $S(n) \geq \log n$  [9].

We state the following Hypothesis:

**Hypothesis 1.** There is a language  $L_1 \in 1NL$ -complete that is in  $L$ . Moreover, there is a nonempty language  $L_2 \in coNL$ , such that there is another language  $L_3$  which is closed under logarithm space reductions in  $NP$ -complete with a deterministic logarithmic space Turing machine  $M$  using an additional special read-once input tape polynomial  $p: \mathbb{N} \rightarrow \mathbb{N}$ , where:

$$L_3 = \{w: M(w, u) = y, \exists u \in \{0, 1\}^{p(|w|)} \text{ such that } y \in L_2\}$$

when by  $M(w, u)$  we denote the computation of  $M$ ,  $w$  is placed on its input tape, and the certificate string  $u$  is placed on the special read-once tape of  $M$ . In this way, there is a  $\text{NP}$ -complete language defined by a logarithmic space verifier  $M$  such that when the input is an element of the language, then there exists a certificate  $u$  such that  $M$  outputs a string which belongs to a single language in  $\text{coNL}$ .

We show the principal consequences of this Hypothesis:

**Theorem 2.** If the Hypothesis 1 is true, then  $L = \text{NL}$  and  $\text{NP} \subseteq L^{(L)}$ .

*Proof.* If there is a language  $L_1 \in \text{1NL}$ -complete in  $L$ , then  $L = \text{NL}$  [8]. We can simulate the computation  $M(w, u) = y$  in the Hypothesis 1 by a nondeterministic logarithmic space oracle Turing machine  $N$  such that the string  $y$  is written in the oracle tape in the computation of  $N(w)$ , since we can read the certificate string  $u$  within the read-once tape by a work tape in a nondeterministic logarithmic space generation of symbols contained in  $u$  [4]. Certainly, we can simulate the reading of one symbol from the string  $u$  into the read-once tape just nondeterministically generating the same symbol in the work tapes using a logarithmic space [4]. We could remove each symbol or a logarithmic amount of symbols generated in the work tapes, when we try to generate the next symbol contiguous to the right on the string  $u$ . In this way, the generation will always be in logarithmic space. This proves that  $L_3$  is in  $\text{NL}^{\text{coNL}}$  since the string  $y$  written in the oracle tape is queried whether  $y \in L_2$  or not. That is equivalent to say that  $L_3$  is in  $L^{(L)}$  when the Hypothesis 1 is true, since  $\text{NL}^{\text{coNL}} = \text{NL}^L = L^L = L^{(L)}$  as a consequence of  $L = \text{NL}$  [9]. Due  $L_3$  is closed under logarithmic space reductions in  $\text{NP}$ -complete, then every  $\text{NP}$  problem is logarithmic space reduced to  $L_3$ . This implies that  $\text{NP} \subseteq L^{(L)}$  since  $L$  is closed under logarithmic space reductions as well.  $\square$

## 1.2. The Problems

We describe the problems that we use and their complexity properties. We will say that the representation of a directed, acyclic graph,  $G$  is topological sorted if for any pair of edges  $(a, b)$  and  $(b, c)$  in  $G$ ,  $(a, b)$  is listed before  $(b, c)$  [8].

### Definition 3. TAGAP

*INSTANCE:* A source vertex  $s$ , a sink vertex  $t$  and a directed and acyclic graph  $G$  that is a topological sorted representation.

*QUESTION:* Is there a directed path from  $s$  to  $t$  in  $G$ ?

*REMARKS:* TAGAP  $\in \text{1NL}$ -complete [8]. In a directed graph, one can distinguish the outdegree (number of outgoing edges), from the indegree (number of incoming edges); a source vertex is a vertex with indegree zero, while a sink vertex is a vertex with outdegree zero. Juris Hartmanis and Stephen Ross Mahaney proved that TAGAP  $\in \text{1NL}$ -complete using an initial configuration as the source vertex and any halting configuration as the sink vertex [8].

A subpath is a path making up part of a larger path

**Definition 4. SUBPATH TAGAP (SPG)**

*INSTANCE:* Two vertices  $s$  and  $t$  and a directed and acyclic graph  $G$  that is a topological sorted representation.

*QUESTION:* Is every path in  $G$  a subpath of some directed path from  $s$  to  $t$ ?

*REMARKS:* We know that  $\text{SPG} \in \text{coNL}$ . Certainly, we can add the single edge  $(t, s)$  and decide whether there is always a cycle that contains any subpath in the modified graph by a nondeterministic logarithmic space Turing machine.

The logic operator  $\oplus$  (XOR) is used in some Boolean formulas instead of using  $\vee$  (OR).

**Definition 5.  $\oplus$  2UNSAT**

*INSTANCE:* A Boolean formula  $\phi$  that is the conjunction of a set of clauses  $c_1, c_2, \dots, c_m$  where each  $c_i$  consists of either a literal or is the XOR (EXCLUSIVE OR) of two literals.

*QUESTION:* Is it the case that  $\phi$  is not satisfiable?

*REMARKS:*  $\oplus$  2UNSAT  $\in L$  <sup>[10], [11]</sup>.

An independent set of an undirected graph  $G$  is a set of vertices of  $G$  such that no two vertices in the independent set are joined by an edge in  $G$ .

**Definition 6. INDEPENDENT SET (ISET)**

*INSTANCE:* A positive integer  $K$  and an undirected graph  $G$ .

*QUESTION:* Does  $G$  contain an independent set with  $K$  vertices or more?

*REMARKS:* ISET  $\in$  NP--complete <sup>[7]</sup>.

## 2. Results

**Theorem 7. TAGAP  $\in L$ .**

*Proof.* Consider a general directed and acyclic graph  $G$  that is a topological sorted representation with a source vertex  $s$  and a sink vertex  $t$ . We reduce it to a CNF expression  $\phi$  such that for each edge  $(a, b)$  in  $G$ , we create the clause  $(\neg x_a \oplus x_b)$ . Finally, we add the two clauses with a single literal  $(x_s)$  and  $(\neg x_t)$ . Since the graph  $G$  is topological sorted, then a directed path  $s, v, w, \dots, t$  is logically equivalent to

$$x_s \Rightarrow x_v \Rightarrow x_w \Rightarrow \dots \Rightarrow x_t$$

in the CNF expression  $\phi$ . However, that is false when the clauses  $(x_s)$  and  $(\neg x_t)$  are satisfied in the Boolean formula  $\phi$  at the same time. If there is no directed path between the vertices  $s$  and  $t$ , then  $\phi$  can be satisfiable since the vertices reachable from  $s$  can be assigned in their variable representations as true and the vertices that reach to  $t$  can be assigned

in their variable representation as false when  $s$  is a source vertex and  $t$  is a sink vertex. For that reason, there is a directed path from  $s$  and  $t$  if and only if  $\phi$  is not satisfiable. This reduction can be made in logarithmic space and thus,  $\text{TAGAP} \in L$  because of  $\oplus 2\text{UNSAT} \in L$ .  $\square$

**Theorem 8.** *There is a deterministic logarithmic space Turing machine  $M$ , where:*

$$\text{ISET} = \{w: M(w, u) = y, \exists u \text{ such that } y \in \text{SPG}\}$$

when by  $M(w, u)$  we denote the computation of  $M$ ,  $w$  is placed on its input tape,  $u$  is placed on the special read-once tape of  $M$ , and  $u$  is polynomially bounded by  $w$ .

*Proof.* The input could be a positive integer  $K$  and an undirected graph  $G$  with  $n$  vertices such that each vertex is represented by a unique integer between 1 and  $n$ . We can create a certificate array  $A$  which contains  $\frac{(K+1) \cdot (K+2)}{2}$  edges that represents a directed and acyclic graph  $G'$  that is a topological sorted representation, every vertex is represented by an integer between 0 and  $n + 1$  and for any pair of edges  $(a, b)$  and  $(a, c)$  in  $G'$  such that  $b < c$ ,  $(a, b)$  is listed before  $(a, c)$  and for any pair of edges  $(a, b)$  and  $(c, d)$  in  $G'$  such that  $a < c$  or  $a < d$ ,  $(a, b)$  is listed before  $(c, d)$ . We read at once the edges of the array  $A$  and we reject when this is not the described graph  $G'$ . Besides, we check that the first vertex contains  $K + 1$  edges (that vertex is represented by 0 in  $G'$ ), the second vertex contains  $K$  edges (that is the vertex that represents the minimum integer greater than 0 in  $G'$ ) and so on until we reach the penultimate vertex (that is the vertex that represents the maximum integer lesser than  $n + 1$  in  $G'$ ) that contains 1 edge (that's why the number of edges is  $(K + 1) + K + \dots + 2 + 1 = \frac{(K+1) \cdot (K+2)}{2}$  in  $G'$ ). While we read the edges of the array  $A$  using the index  $i$ , we check those constraints in  $G'$  and verify that every edge in  $G'$  is not in  $G$ . In this way, we output two vertices and the same certificate (i.e. the edges of the array  $A$ ), where the edges in  $G'$  do not exist in the current input  $G$ .

We obtain that all:

$$(K, G) \in \text{ISET} \Leftrightarrow \exists A \text{ such that } (0, n + 1, A) \in \text{SPG}$$

because of when  $(0, n + 1, A) \in \text{SPG}$ , then this would mean that  $G'$  is a complete graph after a conversion of the directed edges to undirected and we guarantee that those vertices are exactly an independent set of size  $K$  in  $G$  during the computation of the logarithmic space verifier  $M$  (we exclude the vertices represented by 0 and  $n + 1$  in  $G'$  inside of the independent set in  $G$ ). Indeed, we can create this verifier that only uses a logarithmic space in the work tapes such that the array  $A$  is placed on the special read-once tape, because we read at once the edges in the array  $A$ . Hence, we only need to iterate from the cells of the array  $A$  to verify whether the array is an appropriated certificate according to the constraints of  $G'$  and check that every edge in  $G'$  does not exist in  $G$ .

This logarithmic space verifier with output will be the Algorithm 1. We introduce some constraints in the Algorithm 1 in order to guarantee the theoretical procedure. For example, we assume that a value does not exist in the array  $A$  into the cell of a position  $i$  when  $A[i] = \text{null}$ . In addition, we immediately reject when the mentioned comparisons between the vertices in  $G'$  do not hold at least into one single binary digit. That means the machine enters into the rejecting state when the certificate is not valid. Note that, the vertex 0 would be the source vertex and  $n + 1$  is the sink vertex in the instance

$(0, n + 1, A) \in \text{SPG}$ .  $\square$

■ **Algorithm 1** Logarithmic space verifier with output

```

1: /*A valid instance for ISET with its certificate*/
2: procedure VERIFIER( $(K, G), A$ )
3:   /*Initialize the previous left vertex*/
4:    $left \leftarrow 0$ 
5:   /*Initialize the previous right vertex*/
6:    $right \leftarrow 0$ 
7:   /*Initialize three new variables*/
8:    $j \leftarrow 0$ 
9:    $total \leftarrow (K + 1)$ 
10:   $size \leftarrow \frac{(K+1) \cdot (K+2)}{2}$ 
11:  /*Output the source and sink vertices*/
12:  output  $[0, n + 1]$ 
13:  /*Iterate for the edges of the certificate array A*/
14:  for  $i \leftarrow 1$  to  $size$  do
15:    /*Assign the current edge*/
16:     $(v, w) \leftarrow A[i]$ 
17:    if  $(v, w) = null \vee v > n + 1 \vee w < left \vee w > n + 1$  then
18:      return "no"
19:    else if  $i = 1 \wedge v \neq 0$  then
20:      return "no"
21:    else if  $i = size \wedge (A[i + 1] \neq null \vee w \neq n + 1)$  then
22:      return "no"
23:    else if  $v \neq left$  then
24:      if  $v \leq left \vee j \neq total \vee right \neq n + 1$  then
25:        return "no"
26:      end if
27:       $left \leftarrow v$ 
28:       $right \leftarrow w$ 
29:       $j \leftarrow 0$ 
30:       $total \leftarrow total - 1$ 
31:    else if  $w \leq right \vee j > total \vee v < 0$  then
32:      return "no"
33:    else if  $(v, w)$  is an edge in  $G$  then
34:      return "no"
35:    else
36:       $right \leftarrow w$ 
37:       $j \leftarrow j + 1$ 
38:    end if
39:    /*Output the edge  $(v, w)$ */

```



```
40:     output (v, w)
41: end for
42: end procedure
```

---

**Theorem 9.**  $L = NL$  and  $NP \subseteq L^{(L)}$ .

*Proof.* This is a directed consequence of Theorems 2, 7 and 8. Certainly, ISET is closed under logarithm space reductions in NP--complete. Indeed, we can reduced SAT to ISET in logarithmic space and every NP problem could be logarithmic space reduced to SAT by the Cook's Theorem Algorithm<sup>[7]</sup>.

## References

1. [a, b, c, d, e, f, g, h](#) Michael Sipser. *Introduction to the Theory of Computation, volume 2*. Thomson Course Technology Boston, USA, 2006.
2. [a, b, c, d, e, f, g, h, i, j, k](#) Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, USA, 2009.
3. [a, b, c, d, e, f, g, h, i](#) Thomas Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, USA, 3rd edition, 2009.
4. [a, b, c, d, e, f, g, h, i](#) Christos Harilaos Papadimitriou. *Computational complexity*. Addison-Wesley, USA, 1994.
5. <sup>^</sup> Stephen Arthur Cook. *The P versus NP Problem*. <http://www.claymath.org/sites/default/files/pvsnp.pdf>, April 2000. Clay Mathematics Institute. Accessed 9 January 2023.
6. <sup>^</sup> Scott Aaronson. *P ? NP*. *Electronic Colloquium on Computational Complexity, Report No. 4*, 2017.
7. [a, b, c, d, e](#) Michael Randolph Gare and David Stifler Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman and Company, USA, 1 edition, 1979.
8. [a, b, c, d, e, f, g, h](#) Juris Hartmanis and Stephen Ross Mahaney. *Languages Simultaneously Complete for One- Way and Two-Way Log-Tape automata*. *SIAM Journal on Computing*, 10(2):383–390, 1981. doi:10.1137/0210027.
9. [a, b, c, d, e, f, g, h, i, j, k](#) Pascal Michel. *A survey of space complexity*. *Theoretical computer science*, 101(1):99–132, 1992. doi:10.1016/0304-3975(92)90151-5.
10. <sup>^</sup> Neil Jones, Edmund Lien, and William Laaser. *New problems complete for nondeterministic log space*. *Mathematical systems theory*, 10(1):1–17, 1976. doi:10.1007/BF01683259.
11. <sup>^</sup> Omer Reingold. *Undirected connectivity in log-space*. *Journal of the ACM (JACM)*, 55(4):1–24, 2008. doi:10.1145/1391289.1391291.