# LOGSPACE vs P

Frank Vega

FOR REVIEW ONLY
*Computational Complextiy*

# LOGSPACE vs P

Frank Vega

*January 29, 2019*

**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? 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. Given a positive integer x and a collection S of positive integers, MAXIMUM is the problem of deciding whether x is the maximum of S where S is represented by an array. We prove this problem is complete for P. Another major complexity classes are LOGSPACE and coNP. Whether LOGSPACE = P is a fundamental question that it is as important as it is unresolved. We show the problem MAXIMUM can be decided in logarithmic space. Consequently, we demonstrate the complexity class LOGSPACE is equal to P. We define a problem called CIRCUIT-MAXIMUM. CIRCUIT-MAXIMUM is nothing else but the instances of MAXIMUM represented by a positive integer x and a Boolean circuit C which represents the collection S. We show this version of MAXIMUM is in coNP-complete. In addition, CIRCUIT-MAXIMUM contains the instances of MAXIMUM that can be represented by an exponentially more succinct way. In this way, we show the succinct representation of a P-complete problem is indeed in coNP-complete.

## 1 Introduction

The *P* versus *NP* problem is a major unsolved problem in computer science [3]. This is considered by many to be the most important open problem in the field [3]. It is one of the seven Millennium Prize

**ACM Classification:** F.1.3.3, F.1.3.2

**AMS Classification:** 68Q15, 68Q17

**Key words and phrases:** complexity classes, polynomial time, logarithmic space, complete problem, boolean circuit, succinct representation

Problems selected by the Clay Mathematics Institute to carry a US$1,000,000 prize for the first correct solution [3]. It was essentially mentioned in 1955 from a letter written by John Nash to the United States National Security Agency [1]. However, the precise statement of the $P = NP$ problem was introduced in 1971 by Stephen Cook in a seminal paper [3].

In 1936, Turing developed his theoretical computational model [8]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [8]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [8]. 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 [8].

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 [4]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [4].

In the computational complexity theory, the class $P$ contains those languages that can be decided in polynomial time by a deterministic Turing machine [5]. The class $NP$ consists in those languages that can be decided in polynomial time by a nondeterministic Turing machine [5]. The biggest open question in theoretical computer science concerns the relationship between these classes: Is $P$ equal to $NP$?

A logarithmic Turing machine has a read-only input tape, a write-only output tape, and a read/write work tape [8]. The work tape may contain $O(\log n)$ symbols [8]. $LOGSPACE$ is the complexity class containing those decision problems that can be decided by a deterministic logarithmic Turing machine [8]. The $LOGSPACE$ versus $P$ problem is another of the most remarkable problems in complexity theory which remains unsolved [1]. In this work, we prove the complexity class $LOGSPACE$ is equal to $P$.

On the other hand, $LOGSPACE$ is a subclass of $NLOGSPACE$, which is the class of languages decidable in a nondeterministic logarithmic Turing machine [8]. It is known $LOGSPACE \subseteq NLOGSPACE \subseteq P$ [7]. In this way, our proof implies the complexity class $LOGSPACE$ is equal to $NLOGSPACE$ which was a problem that remained open for a long time [7].

## 2 Basic Definitions

Let $\Sigma$ be a finite alphabet with at least two elements, and let $\Sigma^*$ be the set of finite strings over $\Sigma$ [2]. 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 $M(w) =$ "*yes*" [2]. Note that $M$ fails to accept $w$ either if this computation ends in the rejecting state, that is $M(w) =$ "*no*", or if the computation fails to terminate [2].

The language accepted by a Turing machine $M$, denoted $L(M)$, has an associated alphabet $\Sigma$ and is defined by

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

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 running 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]. The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers [4]. Such notations are convenient for describing the worst and better case running time functions, which is usually defined only on integer input sizes [4]. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{f(n) : \textit{ There exist positive constants } c \textit{ and } n_0 \textit{ such that } 0 \leq f(n) \leq c \times g(n) \textit{ for all } n \geq n_0\}$$

where $O$-notation provides an asymptotic upper bound [4]. 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 accepted by the Turing machine $M$ in polynomial time or more specific in a running time $O(n^k)$ for some constant $k$ [2]. Therefore, $P$ is the complexity class of languages that can be accepted in polynomial time by deterministic Turing machines [4]. A verifier for a language $L$ is a deterministic Turing machine $M$, where

$$L = \{w : M(w,c) = \text{“}yes\text{”} \textit{ for some string } c\}.$$

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 symbol $c$, to verify that a string $w$ is a member of $L$. This information is called certificate. *NP* is also the complexity class of languages defined by polynomial time verifiers [7]. If *NP* is the class of problems that have succinct certificates, then the complexity class *coNP* must contain those problems that have succinct disqualifications [7]. That is, a “*no*” instance of a problem in *coNP* possesses a short proof of its being a “*no*” instance [7].

A logarithmic space transducer is a Turing machine with a read-only input tape, a write-only output tape, and a read/write work tape [2]. The work tapes must contain at most $O(\log n)$ symbols [2]. A logarithmic space transducer $M$ computes a function $f : \Sigma^* \to \Sigma^*$, where $f(w)$ is the string remaining on the output tape after $M$ halts when it is started with $w$ on its input tape [2]. We call $f$ a logarithmic space computable function [2]. 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 exists a logarithmic space computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

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

An important complexity class is *P–complete* [6]. A language $L \subseteq \{0,1\}^*$ is *P–complete* if

- $L \in P$, and
- $L' \leq_l L$ for every $L' \in P$.

If $L$ is a language such that $L' \leq_l L$ for some $L' \in$ *P–complete*, then $L$ is *P–hard* [7]. Moreover, if $L \in P$, then $L \in$ *P–complete* [7]. A Boolean formula $\phi$ is composed of

1. Boolean variables: $x_1, x_2, \ldots, x_n$;
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 $\phi$. 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 [4]. 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 [4]. A Boolean formula is in 3-conjunctive normal form or 3*CNF*, if each clause has exactly three distinct literals [4].

For example, the Boolean formula:

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

is in 3*CNF*. The first of its three clauses is $(x_1 \vee \neg x_1 \vee \neg x_2)$, which contains the three literals $x_1$, $\neg x_1$, and $\neg x_2$.

For every $n, m \in \mathbb{N}$ a Boolean circuit $C$ with $n$ inputs and $m$ outputs is a directed acyclic graph [2]. It contains $n$ nodes with no incoming edges; called the input gates and $m$ nodes with no outgoing edges, called the output gates [2]. All other nodes are labeled with one of $\vee$, $\wedge$ or $\neg$ (in other words, the logical operations OR, AND, and NOT) [2]. The $\vee$ and $\wedge$ nodes have fanin (i.e., number of incoming edges) of 2 and the $\neg$ nodes have fanin 1. The size of $C$ is the number of nodes in it [2].

## 3 Results

**Definition 3.1. MAXIMUM**

INSTANCE: A positive integer $x$ and a collection $S$ of positive integers. The collection $S$ could not be a set, since by the definition of a collection, this can contain repeated elements. We will represent the collection $S$ as an array of positive integers $A$ such that some positive integer $i$ is in $S$ if and only if $i$ is an element of $A$. Moreover, the amount of times which is repeated a positive integer in $S$ is the same number of times where this one is contained in $A$.

QUESTION: Is $x$ the maximum number in $S$?

**Lemma 3.2.** *$MAXIMUM \in P$.*

*Proof.* How many comparisons are necessary to determine whether a positive integer $x$ is the maximum of a collection of $n$ positive integers? We can easily obtain an upper bound of $n$ comparisons: examine each element of the collection in turn and keep track of the largest element seen so far and finally, we compare the ultimate result with $x$. In the following procedure, we assume that the collection resides in an array $A$ of length $n$.

Is this the best amount of comparisons we can do? Yes, since we can obtain a lower bound of $n-1$ comparisons for the problem of determining the maximum and one another comparison to check whether this is equal to $x$ [4]. Think of any algorithm that determines the maximum as a tournament among the elements [4]. Each comparison is a match in the tournament in which the larger of the two elements wins [4]. The key observation is that every element except the winner must lose at least one match [4]. Finally, we compare the winner with $x$ [4]. Hence, $n$ comparisons are necessary to determine whether $x$ is the maximum of the collection of positive integers, and the algorithm *MAXIMUM* is optimal with respect to the number of comparisons performed [4]. □

**Algorithm 1** MAXIMUM's Polynomial Time Algorithm

```
1:  procedure MAXIMUM(x, A)
2:      // Assign the first element
3:      max ← A[0]
4:      // Iterate for the elements of the collection
5:      for i ← 1 to n − 1 do
6:          // When the element A[i] is greater than max
7:          if max < A[i] then
8:              // Update the new value of max
9:              max ← A[i]
10:         end if
11:     end for
12:     // If the number x is equal to the maximum of the collection
13:     if max = x then
14:         // Accept
15:         return "yes"
16:     else
17:         // Otherwise reject
18:         return "no"
19:     end if
20: end procedure
```

**Definition 3.3. Unweighted, Not–All–Equal Clauses, 3SAT/FLIP**

INSTANCE: A Boolean formula $\phi$ in $3CNF$ and a truth assignment $T$. Each clause has a weight of 1. The clauses are *not–all–equals* clauses with positive literals. A truth assignment satisfies a clause $c$ under the *not–all–equals* criterion if it is such that $c$ has at least one true and one false literal.

QUESTION: Is the truth assignment $T$ the maximum cost assignment of $\phi$ over all neighbors of $T$? The cost of the assignment is the sum of the weights of the clauses it satisfies. The neighbors of $T$ are truth assignments that differ from $T$ in one bit position.

REMARKS: We denote this language as $U3NSATFLIP$ [6].

**Theorem 3.4.** $U3NSATFLIP \leq_l MAXIMUM$.

*Proof.* Given a Boolean formula $\phi$ in $3CNF$ and a truth assignment $T$, we can calculate the cost assignment of $T$ based on the *not–all–equals* criterion in a logarithmic space algorithm. In the following function $COST$, we assume the truth assignment $T$ is a dictionary that maps every variable in $\phi$ to 1 or 0 (true or false).

**Algorithm 2** COST's Logarithmic space algorithm

```
1:  function COST(φ, T)
2:      // Initialize the cost assignment to 0
3:      num ← 0
4:      // For each clause in φ
5:      for all c ∈ φ do
6:          // The clause c is equal to (p ∨ q ∨ r)
7:          if 0 < T[p] + T[q] + T[r] < 3 then
8:              // Increment num because c complies with the not–all–equals criterion
9:              num ← num + 1
10:         end if
11:     end for
12:     // Return the cost assignment
13:     return num
14: end function
```

This function uses logarithmic space in its work tapes and assumes the clauses contain only positive literals. Certainly, the calculation of $T[p]+T[q]+T[r]$ can be made storing a constant amount of space where $p$, $q$ and $r$ are the positive literals of each clause $c$ in $\phi$. In addition, if $m$ is the number of clauses in $\phi$, then the number $num$ will not exceed the number $m$ and thus, the work tapes will contain at most $O(\log m)$ space.

On the other hand, we can reduce an instance of $U3NSATFLIP$ into another of $MAXIMUM$ in logarithmic space. For this purpose, we are going to use the function $COST$ into a new algorithm. In the following function $REDUCE$, we represent the input instance as a given Boolean formula $\phi$ in $3CNF$ of $n-1$ variables with a truth assignment $T$ and the output instance as a positive integer $x$ with an array $A$ filled with $n$ elements of a collection of positive integers. We will assume the truth assignment $T$ given in the input is a dictionary that maps every variable in $\phi$ to 1 or 0 (true or false) as well.

**Algorithm 3** REDUCE's Logarithmic space algorithm

**function** $REDUCE(\phi, T)$
    // Create an empty array A
    $A \leftarrow [\ldots]$
    // Initialize the index of $A$ in 0
    $i \leftarrow 0$
    // For each variable $y$ in $\phi$
    **for all** $y \in \phi$ **do**
        // Flip the value of $T[y]$ (0 to 1 or 1 to 0)
        $T[y] \leftarrow (T[y]-1) \times (-1)$
        // Calculate the cost of the flipped $T$ based on the *not–all–equals* criterion
        $num \leftarrow COST(\phi, T)$
        // Assign the cost assignment of the neighbor of $T$ after flipping over $T$
        $A[i] \leftarrow num$
        // Increment the index to store the new neighbor cost assignment of $T$
        $i \leftarrow i+1$
        // Return the value of $T[y]$ to the original bit number
        $T[y] \leftarrow (T[y]-1) \times (-1)$
    **end for**
    // Calculate the cost of $T$ based on the *not–all–equals* criterion
    $x \leftarrow COST(\phi, T)$
    // Assign the cost assignment of the original $T$ without flipping any bit position
    $A[i] \leftarrow x$
    // Return the reduction
    **return** $(x, A)$
**end function**

Is this a logarithmic space reduction from $U3NSATFLIP$ to $MAXIMUM$? Given a Boolean formula $\phi$ in $3CNF$ and a truth assignment $T$, we will obtain the positive integer $x$ as the cost assignment of $\phi$ in $T$ and in the array $A$ the cost assignment of $\phi$ from all the neighbors of $T$ included the cost assignment of $T$. In this way, if $x$ is the maximum in the collection of positive integers represented by $A$, then $\langle \phi, T \rangle$ belongs to $U3NSATFLIP$. However, if $x$ is the maximum in the collection of positive integers represented by $A$ (remember that $A$ contains $x$), then this will be an element of the language $MAXIMUM$ as well. Certainly, $\langle \phi, T \rangle$ is in $U3NSATFLIP$ if and only if $x$ is the maximum in the collection of positive integers in $A$. The function $REDUCE$ uses logarithmic space since the bit-length of the index $i$ is $O(\log n)$ because there are $n-1$ variables and thus, there are at most $n$ costs assignments that we need to calculate which is the cost of the original truth assignment $T$ and the $n-1$ truth assignment after flipping one bit position in $T$. Moreover, the bit position that we flip in $T$ will use at most two symbols encoded in binary over the work tapes: the new bit value and the variable. In addition, the algorithm $COST$ runs in logarithmic space in relation to $\phi$ and the truth assignment $T$ with at most one bit flipped. The algorithm $COST$ will take into account the original truth assignment $T$ which remains in the input tape and the changed bit position which is stored in the work tapes. After the computation of $COST$ over each iteration, we will erase from the work tapes the at most $O(\log m)$ space that could contain those tapes where $m$ is the number of clauses in $\phi$. Furthermore, we do not need to store the value of the elements of $A$ in the work tapes since they can be written directly to the output tape. The array $A$ can be written to the output tape as the pairs $(i, v_i)$ where $i$ is an index between 0 and $n-1$ and $v_i$ is equal to the positive integer $A[i]$. We also write the binary string of the number $x$ to the output tape where this string contains at most $O(\log m)$ space. Consequently, we demonstrate $U3NSATFLIP \leq_l MAXIMUM$. □

**Theorem 3.5.** *$MAXIMUM \in P$–complete.*

*Proof.* We prove $U3NSATFLIP$ can be logarithmic reduced to $MAXIMUM$ and $U3NSATFLIP$ is in *P–complete* under logarithmic space reductions [6], thus $MAXIMUM$ belongs to *P–hard*. Moreover, since $MAXIMUM \in P$, then $MAXIMUM$ is in *P–complete*. □

**Theorem 3.6.** *$MAXIMUM \in LOGSPACE$.*

*Proof.* Given a positive integer $x$ and a collection $S$ of positive integers, we are going to demonstrate we can decide this problem in logarithmic space. In the following procedure, we assume that the collection resides in array $A$ of length $n$. Besides, we assume the function $length$ calculates the bit-length of a binary string and uses a logarithmic space for the calculation.

Is this a logarithmic space algorithm? Yes, since we compare the value of the functions $length(x)$ and $length(A[i])$ (the $i^{th}$ element of $A$) using a logarithmic space although we could partially calculate the $length(A[i])$. In addition, the calculated bit-length of $x$ only uses at most $O(\log x)$ space. Besides, in the comparison with the bit-length of $A[i]$ and $x$ we halt and reject immediately when $length(A[i])$ exceeds $length(x)$ at least in one digit and thus, we do not need to calculate completely the $length(A[i])$ to reject. In this way, we just keep at most $O(\log x)$ space in the calculation of $length(A[i])$. Finally, when both bit-lengths are equal, then we compare the elements $A[i]$ and $x$ bit by bit. For this purpose, we compare only two bits in the input tape over the same position $j$ from $x$ and $A[i]$ in a descending order for each step. Notice, that we start to compare from the last bit position in a descending order. For example, in the binary string 100 which represents the number 4, we start iterating from the last bit element, that is the

**Algorithm 4** MAXIMUM's Logarithmic space algorithm

```
1:  procedure MAXIMUM(x, A)
2:      // Initialize the variable answer
3:      answer ← "no"
4:      // Iterate for each element of the collection
5:      for i ← 0 to n − 1 do
6:          // If the bit-length of x is lesser than the bit-length of element A[i]
7:          if length(x) < length(A[i]) then
8:              // Reject because A[i] is greater than x
9:              return "no"
10:             // If the bit-length of x is greater than the bit-length of element A[i]
11:         else if length(x) > length(A[i]) then
12:             // Continue to the next iteration on i
13:             continue
14:             // If the bit-length of x is equal to the bit-length of element A[i]
15:         else
16:             // Assign the index to the last bit element
17:             j ← length(x) − 1
18:             // While there are bits to compare
19:             while j ≥ 0 do
20:                 // Compare the bit in the position j of x with the bit in the position j of A[i]
21:                 if x[j] < A[i][j] then
22:                     // Reject because A[i] is greater than x
23:                     return "no"
24:                 else if x[j] > A[i][j] then
25:                     // Continue to the next iteration on i
26:                     break
27:                 else
28:                     // Decrement the bit position j of x and A[i]
29:                     j ← j − 1
30:                 end if
31:             end while
32:             // After iterating from all the bits of x and A[i]
33:             if j < 0 then
34:                 // x is equal to A[i]
35:                 answer ← "yes"
36:             end if
37:         end if
38:     end for
39:     // Accept if answer = "yes" and reject when answer = "no"
40:     return answer
41: end procedure
```

bit 1. Moreover, we store the position $j$ in the work tapes and this value has at most $O(\log x)$ space. If it would be the case that $A[i]$ and $x$ have the same bit-length, but $A[i]$ is greater than $x$, then we reject. We continue the iteration with the next value $i$ while the property that $x$ is the maximum number in the array remains as true. However, we only accept when the value of the variable *answer* is "*yes*" when initially has the value of "*no*" by default. The value will be "*yes*" in the variable *answer* after the whole iteration for each element in the array if and only if there is at least one element $A[i]$ that is equal to $x$. Furthermore, if the iteration is completed until the last item, then $x$ is greater than or equal to every element in the array $A$. To sum up, we show we can decide whether $x$ is the maximum of the collection represented by the array $A$ in logarithmic space and thus, $MAXIMUM \in LOGSPACE$. □

**Theorem 3.7.** *$LOGSPACE = P$.*

*Proof.* As result of Theorems 3.5 and 3.6 we obtain $LOGSPACE = P$, because the complexity class *LOGSPACE* is closed under logarithmic space reductions [7]. □

**Definition 3.8. CIRCUIT–MAXIMUM**

INSTANCE: A positive integer $x$ and a collection $S$ of positive integers such that the collection $S$ is represented by a Boolean circuit $C$ where some positive integer $i$ belongs to $S$ if and only if $C(i)$ accepts.

QUESTION: Is $x$ the maximum number in $S$?

**Theorem 3.9.** *$CIRCUIT–MAXIMUM \in coNP$.*

*Proof.* The language of *CIRCUIT–MAXIMUM* is in *coNP*. Certainly, we can check in polynomial time a disqualification from an instance $\langle x, C\rangle$ of this language that is a positive integer $y$ where $x < y$ and $y$ is in $S$ or we can simply verify in polynomial time when $x$ is not in $S$ where $\langle \ldots \rangle$ is the binary encoding. Indeed, we can check whether the both evaluations of $y$ and $x$ in $C$ accept and check later whether $x < y$ or we can just verify when $C(x)$ does not accept. Certainly, we can polynomially make the verification when $\langle x, C\rangle$ is a "*no*" instance of the problem *CIRCUIT–MAXIMUM*, because the evaluation in the Boolean circuit can be done in polynomial time as well. □

Given a Boolean circuit $C$, the problem *coCIRCUIT–SAT* consists in deciding whether there is not any input such that $C$ accepts [7].

**Theorem 3.10.** *$CIRCUIT–MAXIMUM \in coNP–complete$.*

*Proof.* Given a Boolean circuit $C$ we can check whether $C(0)$ does not accept. In that case, we create a succinct Boolean circuit $C'$ which only accepts the input string 0 and has the same number of input gates of $C$. We combine $C$ with $C'$ through the input gates into a new Boolean circuit $C''$ which accepts only when $C$ or $C'$ accept. This is possible just adding a gate *OR* between the output gates of $C$ and $C'$. The instance of the positive integer 0 and the final Boolean circuit $C''$ belongs to *CIRCUIT–MAXIMUM* if and only if $C$ is in *coCIRCUIT–SAT*. Certainly, 0 is the maximum of the collection that represents $C''$ if there is not any other input which $C''$ accepts. In addition, $C''$ accepts the positive integer 0 because of the construction of $C'$ on $C$. Since we can create the succinct Boolean circuit $C'$ and evaluate $C$ on the input 0 in polynomial time, then we can reduce *coCIRCUIT–SAT* to *CIRCUIT–MAXIMUM* in polynomial time. *coCIRCUIT–SAT* is a known *coNP–complete* problem [7]. Hence, the language *CIRCUIT–MAXIMUM* is in *coNP–hard* [7]. As result of Theorem 3.9, we obtain *CIRCUIT–MAXIMUM* is also in *coNP* and thus, the proof is completed. □

**Theorem 3.11.** *CIRCUIT–MAXIMUM is a succinct representation of the language MAXIMUM.*

*Proof.* Every Boolean circuit $C$ could always be a succinct representation of some collection of positive integers $S$. Indeed, this will happen since there is always a collection $S$ which could contain more than or approximately to $2^m$ elements (remember that a collection could contain repeated elements) if we represent it by a Boolean circuit of $m$ input gates. In addition, since a collection could contain any amount of repeated elements, then every instance of *CIRCUIT–MAXIMUM* is a succinct representation of another instance of *MAXIMUM*. Certainly, *CIRCUIT–MAXIMUM* is nothing else but a language that contains the instances of the problem *MAXIMUM* which could be represented by an exponentially more succinct input in relation to $S$ [7]. □

## References


[1] SCOTT AARONSON: P $\stackrel{?}{=}$ NP. *Electronic Colloquium on Computational Complexity, Report No. 4*, 2017. 2

[2] SANJEEV ARORA AND BOAZ BARAK: *Computational complexity: a modern approach*. Cambridge University Press, 2009. 2, 3, 4

[3] STEPHEN A COOK: The P versus NP Problem, April 2000. at `http://www.claymath.org/sites/default/files/pvsnp.pdf`. 1, 2

[4] THOMAS H CORMEN, CHARLES E LEISERSON, RONALD L RIVEST, AND CLIFFORD STEIN: *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. 2, 3, 4

[5] ODED GOLDREICH: *P, NP, and NP-Completeness: The basics of computational complexity*. Cambridge University Press, 2010. 2

[6] RAYMOND GREENLAW, H. JAMES HOOVER, AND WALTER L. RUZZO: *A Compendium of Problems Complete for P*. Oxford University Press, 1993. 3, 6, 8

[7] CHRISTOS H PAPADIMITRIOU: *Computational complexity*. Addison-Wesley, 1994. 2, 3, 10, 11

[8] MICHAEL SIPSER: *Introduction to the Theory of Computation*. Volume 2. Thomson Course Technology Boston, 2006. 2

## AUTHOR

Frank Vega
Computational Researcher
Joysonic
Belgrade, Serbia
vega.frank@gmail.com
https://uh-cu.academia.edu/FrankVega

## ABOUT THE AUTHOR

FRANK VEGA is essentially a back-end programmer graduated in Computer Science since 2007. In August 2017, he was invited as a guest reviewer for a peer-review of a manuscript about Theory of Computation in the flagship journal of IEEE Computer Society. In October 2017, he contributed as co-author with a presentation in the $7^{th}$ International Scientific Conference on economic development and standard of living ("EDASOL 2017 - Economic development and Standard of living"). In February 2017, his book "Protesta" (a book of poetry and short stories in Spanish) was published by the Alexandria Library Publishing House. He was also Director of two IT Companies (Joysonic and Chavanasoft) created in Serbia.