P versus NP

Frank Vega
CopSonic, 1471 Route de Saint-Nauphary 82000 Montauban, France
vega.frank@gmail.com

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. It is one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute to carry a US 1,000,000 prize for the first correct solution. Another major complexity class is Sharp-P-complete. A polynomial time algorithm for solving a Sharp-P-complete problem, if it existed, then this would solve the P versus NP problem by implying that P and NP are equal. We demonstrate there is a problem in Sharp-P-complete that can be solved in polynomial time. In this way, we prove the complexity class P is equal to NP.

2012 ACM Subject Classification Theory of computation → Complexity classes; Theory of computation → Problems, reductions and completeness

Keywords and phrases complexity classes, completeness, polynomial time, reduction, logarithmic space, one-way

1 Introduction

The P versus NP problem is a major unsolved problem in computer science [7]. This is considered by many to be the most important open problem in the field [7]. The precise statement of the $P = NP$ problem was introduced in 1971 by Stephen Cook in a seminal paper [7]. In 2012, a poll of 151 researchers showed that 126 (83%) believed the answer to be no, 12 (9%) believed the answer is yes, 5 (3%) believed the question may be independent of the currently accepted axioms and therefore impossible to prove or disprove, 8 (5%) said either do not know or do not care or don't want the answer to be yes nor the problem to be resolved [13].

The $P = NP$ question is also singular in the number of approaches that researchers have brought to bear upon it over the years [10]. From the initial question in logic, the focus moved to complexity theory where early work used diagonalization and relativization techniques [10]. It was showed that these methods were perhaps inadequate to resolve P versus NP by demonstrating relativized worlds in which $P = NP$ and others in which $P \neq NP$ [4]. This shifted the focus to methods using circuit complexity and for a while this approach was deemed the one most likely to resolve the question [10]. Once again, a negative result showed that a class of techniques known as "Natural Proofs" that subsumed the above could not separate the classes NP and P, provided one-way functions exist [22]. There has been speculation that resolving the $P = NP$ question might be outside the domain of mathematical techniques [10]. More precisely, the question might be independent of standard axioms of set theory [10]. Some results have showed that some relativized versions of the $P = NP$ question are independent of reasonable formalizations of set theory [14].

In 1936, Turing developed his theoretical computational model [23]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [23]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [23]. 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 [23]. 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 [8]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [8]. NP is the complexity class which contains those languages that can be decided in polynomial time by nondeterministic Turing machines.

A major complexity class is *Sharp–P* (denoted as $\#P$) [24]. This can be defined by the class of function problems of the form "*compute* $f(x)$", where f is the number of accepting paths of a nondeterministic Turing machines, where this machine always accepts in polynomial time [24]. In previous years there has been great interest in the verification or checking of computations [18]. Interactive proofs introduced by Goldwasser, Micali and Rackoff and Babi can be viewed as a model of the verification process [18]. Dwork and Stockmeyer and Condon have studied interactive proofs where the verifier is a space bounded computation instead of the original model where the verifier is a time bounded computation [18]. In addition, Blum and Kannan have studied another model where the goal is to check a computation based solely on the final answer [18]. More about probabilistic logarithmic space verifiers and the complexity class NP has been investigated on a technique of Lipton [18]. We show some results about the logarithmic space verifiers applied to the class $\#P$. In this way, we provide a proof to solve the outstanding P versus NP problem.

2 Materials & Methods

2.1 Polynomial time verifiers

Let Σ be a finite alphabet with at least two elements, and let Σ^* be the set of finite strings over Σ [3]. A Turing machine M has an associated input alphabet Σ [3]. For each string w in Σ^* there is a computation associated with M on input w [3]. We say that M accepts w if this computation terminates in the accepting state, that is $M(w) =$ "yes" [3]. 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, 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) [3].

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

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

Moreover, $L(M)$ is decided by M, when $w \notin L(M)$ if and only if $M(w) =$ "no" [8]. We denote by $t_M(w)$ the number of steps in the computation of M on input w [3]. 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 Σ^n is the set of all strings over Σ of length n [3]. We say that M runs in polynomial time if there is a constant k such that for all n, $T_M(n) \leq n^k + k$ [3]. 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 [8]. A verifier for a language L_1 is a deterministic Turing machine M, where:

$$L_1 = \{w : M(w, c) = \text{“}yes\text{”} \textit{ for some string } c\}.$$

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 [3]. A verifier uses additional information, represented by the symbol c, to verify that a string w is a member of L_1. This information is called certificate. NP is also the complexity class of languages defined by polynomial time verifiers [21].

A decision problem in NP can be restated in this way: There is a string c with $M(w, c) =$ "*yes*" if and only if $w \in L_1$, where L_1 is defined by the polynomial time verifier M [21]. The function problem associated with L_1, denoted FL_1, is the following computational problem: Given w, find a string c such that $M(w, c) =$ "*yes*" if such string exists; if no such string exists, then reject, that is, return "*no*" [21]. The complexity class of all function problems associated with languages in NP is called FNP [21]. FP is the complexity class that contains those problems in FNP which can be solved in polynomial time [21].

To attack the P versus NP question the concept of *NP–completeness* has been very useful [12]. A principal *NP–complete* problem is SAT [12]. An instance of SAT is a Boolean formula ϕ which 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 ϕ is a set of values for the variables in ϕ. On the one hand, a satisfying truth assignment is a truth assignment that causes ϕ to be evaluated as true. On the other hand, a truth assignment that causes ϕ to be evaluated as false is a unsatisfying truth assignment. A Boolean formula with a satisfying truth assignment is satisfiable. The problem SAT asks whether a given Boolean formula is satisfiable [12].

An important complexity is *Sharp–P* (denoted as $\#P$) [24]. We can also define the class $\#P$ using polynomial time verifiers. Let $\{0,1\}^*$ be the infinite set of binary strings, a function $f : \{0,1\}^* \to \mathbb{N}$ is in $\#P$ if there exists a polynomial time verifier M such that for every $x \in \{0,1\}^*$,

$$f(x) = |\{y : M(x, y) = \text{“}yes\text{”}\}|$$

where $|\cdots|$ denotes the cardinality set function [3]. *#P–complete* is another complexity class. A problem is *#P–complete* if and only if it is in $\#P$, and every problem in $\#P$ can be reduced to it by a polynomial time counting reduction [21].

2.2 Logarithmic space verifiers

A logarithmic space Turing machine has a read-only input tape, a write-only output tape, and read/write work tapes [23]. The work tapes may contain at most $O(\log n)$ symbols [23]. In computational complexity theory, L is the complexity class containing those decision problems that can be decided by a deterministic logarithmic space Turing machine [21]. NL is the complexity class containing the decision problems that can be decided by a nondeterministic logarithmic space Turing machine [21].

A logarithmic space transducer is a Turing machine with a read-only input tape, a write-only output tape, and read/write work tapes [23]. The work tapes must contain at most $O(\log n)$ symbols [23]. 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 [23]. We call f a logarithmic space computable function [23]. 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 $x \in \{0,1\}^*$:

$x \in L_1$ *if and only if* $f(x) \in L_2$.

The logarithmic space reduction is used in the definition of the complete languages for the classes L and NL [21]. 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 [8]. 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 [8]. A Boolean formula is in 2-conjunctive normal form or $2CNF$, if each clause has exactly two distinct literals [8]. There is a problem called $2SAT$, where we asked whether a given Boolean formula ϕ in $2CNF$ is satisfiable. $2SAT$ is complete for NL [21].

We can give a certificate-based definition for NL [3]. The certificate-based definition of NL assumes that a logarithmic space Turing machine has another separated read-only tape [3]. On each step of the machine, the machine's head on that tape can either stay in place or move to the right [3]. In particular, it cannot reread any bit to the left of where the head currently is [3]. For that reason, this kind of special tape is called "read-once" [3].

▶ **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 $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$:*

$$x \in L_1 \Leftrightarrow \exists\, u \in \{0,1\}^{p([x])} \text{ such that } M(x,u) = \text{“}yes\text{”}$$

where by $M(x,u)$ we denote the computation of M where x is placed on its input tape, and the certificate 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 $[\ldots]$ is the bit-length function [3]. M is called a logarithmic space verifier [3].

An interesting complexity class is *Sharp–L* (denoted as $\#L$). $\#L$ has the same relation to L as $\#P$ does to P [2]. We can define the class $\#L$ using logarithmic space verifiers as well.

▶ **Definition 2.** *Let $\{0,1\}^*$ be the infinite set of binary strings, a function $f : \{0,1\}^* \to \mathbb{N}$ is in $\#L$ if there exists a logarithmic space verifier M such that for every $x \in \{0,1\}^*$,*

$$f(x) = |\{u : M(x,u) = \text{“}yes\text{”}\}|$$

where $|\cdots|$ denotes the cardinality set function [2].

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 head on the input tape to the left [20]. 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 [15].

▶ **Lemma 3.** *NL is closed under nondeterministic logarithmic space reductions to every language in $1NL$.*

Proof. Suppose, we have two languages L_1 and $L_2 \in 1NL$, such that there is a nondeterministic logarithmic space Turing machine M which makes a reduction from $x \in L_1$ into $M(x) \in L_2$. Besides, we assume there is a nondeterministic one-way logarithmic space

Turing machine M' which decides L_2. Hence, we only need to prove that $M'(M(x))$ is a nondeterministic logarithmic space Turing machine. The solution to this problem is simple: We do not explicitly store the output result of M in the work tapes of M'. Instead, whenever M' needs to move the head on the input tape (this tape will be the output tape of M), then we continue the computation of M on input x long enough for it to produce the new output symbol; this is the symbol that will be the next scanned symbol on the input tape of M'.

If M' only needs to read currently from the work tapes, then we just pause the computation of M on the input x and continue the computation of M' until this needs to move to the right on the input tape. We can always continue the simulation, because M' never moves the head on the input tape to the left. We only accept when the machine M enters in the halting state and M' enters in the accepting state otherwise we reject. It is clear that this simulation indeed computes $M'(M(x))$ in a nondeterministic logarithmic space. In this way, we obtain $x \in L_1$ if and only if $M'(M(x)) =$ "yes" which is a clear evidence that L_1 is in NL. ◀

We can give an equivalent definition for NL, but this time the output is a string which belongs to a language in $1NL$.

▶ **Definition 4.** *A language L_1 is in NL if there exists another nonempty language $L_2 \in 1NL$ and a deterministic logarithmic space Turing machine M with an additional special read-once input tape polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$:*

$$x \in L_1 \Leftrightarrow \exists\ u \in \{0,1\}^{p([x])} \text{ such that } M(x,u) = y, \text{ where } y \in L_2$$

and by $M(x,u) = y$ we denote the computation of M where x is placed on its input tape, and y is the remaining string in the output tape on M after the halting state, and the certificate 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 $[\ldots]$ is the bit-length function [3]. We call M a one-way logarithmic space verifier. This definition is still valid, because of Lemma 3.

According to the previous definition, we can redefine $\#L$ as follows:

▶ **Definition 5.** *Let $\{0,1\}^*$ be the infinite set of binary strings, a function $f : \{0,1\}^* \to \mathbb{N}$ is in $\#L$ if there exists another nonempty language $L_2 \in 1NL$, and a nondeterministic one-way logarithmic space Turing machine M' which decides L_2, and a one-way logarithmic space verifier M such that for every $x \in \{0,1\}^*$,*

$$f(x) = |\{(u,p) : M(x,u) = y, \text{ where } y \in L_2 \text{ and } p \text{ is an accepting path of } M'(y)\}|$$

and $|\cdots|$ denotes the cardinality set function. This definition is still valid under the result of Lemma 3.

3 Results

We define a new problem:

▶ **Definition 6.** ***NOT–A–SET***

INSTANCE: A unary string 0^q and a collection of binary strings, such that each element in the collection represents a power number in base 2 with a bit-length lesser than or equal to q. The collection of numbers is represented by an array N.

QUESTION: Is there a repeated element in N?

▶ **Theorem 7.** *NOT–A–$SET \in 1NL$.*

Proof. Given an instance $(0^q, N)$ of *NOT–A–SET*, then we can read its elements from left to right on the input tape, verify that every element in the collection is a binary string, and finally check whether every element in N has a bit-length lesser than or equal to q. In addition, we can nondeterministically pick a binary integer d between 1 and q and accept in case of there exists the number 2^{d-1} at least twice in N.

We can make all this computation in a nondeterministic one-way using logarithmic space. Certainly, the verification of the membership of 2^{d-1} in N could be done in logarithmic space, since it is trivial to check whether a binary string represents the power 2^{d-1}. Besides, we can store a logarithmic amount of symbols, because of d has an exponential more succinct representation in relation to the unary string 0^q [21]. Moreover, the variables that we could use for the iteration of the elements in N have a logarithmic space in relation to the length of the instance $(0^q, N)$.

We never need to move to the left on the input tape for the acceptance or rejection of the elements in *NOT–A–SET* in a nondeterministic logarithmic space. We describe this nondeterministic one-way logarithmic space computation in the Algorithm 1. In this algorithm, we assume a value does not exist in the array N into the cell of some position i when $N[i] =$ *undefined*. To sum up, we actually prove that *NOT–A–SET* is in $1NL$. ◀

Let's consider an interesting problem:

▶ **Definition 8.** ***#CLAUSES–2UNSAT***

INSTANCE: Two natural numbers n, m, and a Boolean formula ϕ in $2CNF$ of n variables and m clauses, such that the clauses can contain the constant true value. The clauses are represented by an array C, such that C represents a set of m set elements, where $C[i] = S_i$ if and only if S_i is exactly the set of literals or constant true values into a clause c_i in ϕ for $1 \leq i \leq m$. Besides, each variable in ϕ is represented by a unique integer between 1 and n. In addition, a negative or positive integer represents a negated or non-negated literal, respectively. The constant true value is represented by the number 0.

ANSWER: Count the number of unsatisfied clauses between all the truth assignments in ϕ.

▶ **Theorem 9.** *$\#CLAUSES$–$2UNSAT \in FP$.*

Proof. We are going to show there is a deterministic Turing machine M, where:

$$\#CLAUSES\text{–}2UNSAT = \{w : M(w, u) = y, \exists\ u \text{ such that } y \in NOT\text{–}A\text{–}SET\}$$

when M runs in logarithmic space in the length of w, u is placed on the special read-once tape of M, and u is polynomially bounded by w. Given an instance (n, m, C) of *#CLAUSES–2UNSAT*, we firstly check whether this instance has an appropriate representation according to the constraints introduced in the Definition 8. The constraints for the Definition 8 are the following ones:

1. The array C must contain exactly m sets and,
2. each variable must be represented by a unique integer between 1 and n and,
3. there are no two equals sets inside of C and finally,
4. every set must contain exactly two distinct literals or a literal and the constant true value.

ALGORITHM 1: *ONE–WAY–ALGO*

Data: $(0^q, N)$ where $(0^q, N)$ is an instance of *NOT–A–SET*
Result: A nondeterministic acceptance or rejection in one-way logarithmic space

```
// Get the length of the unary string 0^q as a binary string
q ⟵ length(0^q);
// Generate nondeterministically an arbitrary integer between 1 and q
d ⟵ random(1, q);
t ⟵ 0;
// Initial position in N
i ⟵ 1;
while N[i] ≠ undefined do
    s ⟵ 0;
    // N[i][j] represents the j^th digit of the string in N[i]
    for j ← 1 to q + 1 do
        if j = q + 1 then
            if N[i][j] ≠ undefined then
                // There exists an element with bit-length greater than q
                return "no";
            end
        end
        else if (j = 1 ∧ N[i][j] ≠ 1) ∨ (j > 1 ∧ N[i][j] = 1) ∨ N[i][j] ∉ {0, 1, undefined} then
            // The element N[i] is not a binary string
            return "no";
        end
        else if N[i][j] = undefined then
            // Break the current for loop statement
            break;
        end
        else
            // Store the current position of digit N[i][j] in N[i]
            s ⟵ s + 1;
        end
    end
    if s = d ∧ t < 2 then
        // The element N[i] is equal to 2^(d−1)
        t ⟵ t + 1;
    end
    i ⟵ i + 1;
end
if t = 2 then
    // The element 2^(d−1) appears at least twice in N
    return "yes";
end
else
    return "no";
end
```

All these requirements are verified in the Algorithm 2, where this subroutine decides whether the instance has an appropriate representation according to the Definition 8. After that verification, we use a certificate as an array A, such that this consists in an array A which contains n different integer numbers in ascending absolute value order. We read at once the elements of the array A and we reject whether this is not an appropriate certificate: That is, when the absolute value of the numbers are not sorted in ascending order, or the array A does not contain exactly n elements, or the array A contains a number that its absolute value is not between 1 and n, since every variable is represented by an integer between 1 and n in C.

While we read each element x of the array A, then we copy the binary numbers 2^{j-1} that represent the sets $C[j]$ which contain the literal x just creating another instance $(0^q, N)$ of *NOT–A–SET*, where the value of q is equal to m. Since the array A does not contain repeated elements, then we could correspond each certificate A to a truth assignment for ϕ with a representation of all the variables in ϕ, such that the literals in A are false. We know a set $C[j]$ that represents a clause is false if and only if the two literals in $C[i]$ are false: This does not include the clauses that contain the constant true value. Therefore, the evaluation as false into the literals of the array A corresponds to a unsatisfying truth assignment in ϕ if and only if we write some number 2^{j-1} twice to the output tape, where 2^{j-1} represents a set $C[j]$ for some $1 \leq j \leq m$.

Furthermore, we can make this verification in logarithmic space such that the array A is placed on the special read-once tape, because we read at once the elements in the array A. Indeed, the variables that we could use for the iteration of the elements in A and C have a logarithmic space in relation to the length of the instance (n, m, C). Hence, we only need to iterate from the elements of the array A to verify whether the array is an appropriate certificate and write to the output tape the representation as a power of two of the sets in C that contain the literals in A. This logarithmic space verification will be the Algorithm 3. We assume whether a value does not exist in the arrays A or C into the cell of some position i when $A[i] = \textit{undefined}$ or $C[i] = \textit{undefined}$, respectively.

The Algorithm 3 is a one-way logarithmic space verifier, since this never moves the head on the special read-once tape to the left, where it is placed the certificate A. Moreover, for every unsatisfying truth assignment represented by the array A, then the output of this one-way logarithmic space verifier will always belong to the language *NOT–A–SET*, where we know that $\textit{NOT–A–SET} \in 1NL$ as a consequence of Theorem 7. In addition, every appropriate certificate A is always polynomially bounded by the instance (n, m, C).

Consequently, we demonstrate that *#CLAUSES–2UNSAT* belongs to the complexity class $\#L$ under the Definition 5. Certainly, every truth assignment in ϕ corresponds to a single certificate in our one-way logarithmic space verifier. In addition, the number of accepting paths in the Algorithm 1 for the generated instance $(0^q, N)$ of *NOT–A–SET* is exactly the number of unsatisfied clauses for a single truth assignment.

The number of accepting paths in the Algorithm 1 for a single instance is equal to the number of different powers of two that exist at least twice in the array N. Actually, this corresponds to the number of unsatisfied clauses for the truth assignment that represents the certificate A. We know that $\#L$ is contained in the class FP [2], [6], [3]. As result, $\#L$ remains in the class FP under the Definition 5 as a consequence of Lemma 3. In conclusion, we show that *#CLAUSES–2UNSAT* is indeed in FP. ◀

We show a previous known *#P–complete* problem:

▶ **Definition 10.** ***#MONOTONE–2UNSAT***

ALGORITHM 2: *CHECK–ALGO*

Data: (n, m, C) where (n, m, C) is an instance of *#CLAUSES–2UNSAT*
Result: A logarithmic space subroutine

```
if n < 1 ∨ m < 1 then
    // n or m is not a natural number
    return "no";
end
for i ← 1 to m + 1 do
    if (i < m + 1 ∧ C[i] = undefined) ∨ (i = m + 1 ∧ C[i] ≠ undefined) then
        // C does not contain exactly m sets
        return "no";
    end
end
for i ← 1 to n do
    t ⟵ 0;
    foreach j ← 1 to m; C[j] = {x, y} do
        // abs(...) denotes the absolute value operation
        if abs(x) > n ∨ abs(y) > n then
            // C does not contain exactly n variables from 1 to n
            return "no";
        end
        if t = 0 ∧ (i = abs(x) ∨ i = abs(y)) then
            // Store the existence of the variable i in C
            t ⟵ 1;
        end
    end
    if t = 0 then
        // C does not contain the variable i
        return "no";
    end
end
for i ← 1 to m − 1 do
    for j ← i + 1 to m do
        // ∩ denotes the intersection set operation
        if C[i] ∩ C[j] = C[i] then
            // The array C is not exactly a "set" of set elements
            return "no";
        end
        // |···| denotes the cardinality set function
        if |C[i]| + |C[j]| ≠ 4 then
            // C[i] or C[j] does not contain exactly two elements
            return "no";
        end
    end
end
// The instance (n, m, C) is appropriate for #CLAUSES-2UNSAT
return "yes";
```

ALGORITHM 3: *VERIFIER–ALGO*

Data: (n, m, C, A) where (n, m, C) is an instance of $\#CLAUSES\text{–}2UNSAT$ and A is a certificate

Result: A one-way logarithmic space verifier

```
if CHECK–ALGO(n, m, C) = "no" then
    // (n, m, C) is not an appropriate instance of #CLAUSES-2UNSAT
    return "no";
end
else
    output 0^m;
end
// Minimum current variable during the iteration of the array A
x ⟵ 0;
for i ← 1 to n + 1 do
    if i = n + 1 then
        if A[i] ≠ undefined then
            // The array A contains more than n elements
            return "no";
        end
    end
    else if A[i] = undefined ∨ abs(A[i]) < 1 ∨ abs(A[i]) > n ∨ abs(A[i]) ≤ x then
        // The certificate A is not appropriate
        return "no";
    end
    else
        // abs(...) denotes the absolute value operation
        x ⟵ abs(A[i]);
        y ⟵ A[i];
        for j ← 1 to m do
            if y ∈ C[j] then
                // Output the number 2^(j−1) when the set C[j] contains the literal y
                output , 1;
                if j − 1 > 0 then
                    output 0^(j−1);
                end
            end
        end
    end
end
```

INSTANCE: Two natural numbers n, m, and a Boolean formula ϕ in $2CNF$ of n variables and m clauses, such that there is no clause in ϕ which contains a negated variable [25]. The clauses are represented by an array C, such that C represents a set of m set elements, where $C[i] = S_i$ if and only if S_i is exactly the set of variables into a clause c_i in ϕ for $1 \leq i \leq m$. Besides, each variable in ϕ is represented by a unique integer between 1 and n.

ANSWER: Count the number of unsatisfying truth assignments in ϕ.

REMARKS:$\#MONOTONE\text{–}2UNSAT \in \#P$–complete [25].

▶ **Theorem 11.** *$\#MONOTONE\text{–}2UNSAT \in FP$.*

Proof. Given an instance (n, m, C) of *$\#MONOTONE\text{–}2UNSAT$* that represents a Boolean formula ϕ, then we can compute the counting solutions of (n, m, C) in polynomial time using the algorithm for *$\#CLAUSES\text{–}2UNSAT$* according to the Theorem 9. Given a clause c in ϕ, then the number of unsatisfying truth assignments for the Boolean formula ϕ is always equal to the number of unsatisfying truth assignments for the Boolean formula ϕ when c is unsatisfied plus the number of unsatisfying truth assignments for the Boolean formula ϕ when c is satisfied.

We can count the number of unsatisfying truth assignments in ϕ when c is unsatisfied. We can easily calculate this just calling the polynomial time algorithm *POLY–ALGO* that would solve an instance of *$\#CLAUSES\text{–}2UNSAT$*. Certainly, this will be equal to the number of unsatisfied clauses between all the truth assignments in ϕ with the clause c minus the number of unsatisfied clauses between all the truth assignments in ϕ without the clause c. Hence, we can apply the algorithm *POLY–ALGO* for the both sides of this subtraction.

Indeed, the number of unsatisfied clauses for a truth assignment in ϕ when the clause c is unsatisfied minus the number of unsatisfied clauses for the same truth assignment in ϕ without the clause c is equal to 1. Moreover, the number of unsatisfied clauses for a truth assignment in ϕ when the clause c is satisfied minus the number of unsatisfied clauses for the same truth assignment in ϕ without the clause c is equal to 0. In this way, we count only once every unsatisfying truth assignment for the Boolean formula ϕ when the clause c is unsatisfied. This will only be possible, if we guarantee the Boolean formula ϕ without the clause c contains the same variables of ϕ. For that reason, we add the clauses $(x \vee true)$ and $(true \vee y)$ to the Boolean formula ϕ' when the clause c is equal to $(x \vee y)$ and ϕ' would be the remaining formula after the removal of the clause c in ϕ, where x and y are variables and "*true*" is the constant true value.

Besides, we can count the number of unsatisfying truth assignments in ϕ when c is satisfied. If the clause c is equal to $(x \vee y)$, then the number of unsatisfying truth assignments in ϕ when c is satisfied would be equal to the sum of the number of unsatisfying truth assignments in ϕ' with exactly one the clauses c_2, c_3 or c_4 when they are unsatisfied, where the clauses c_2 and c_3 and c_4 are equal to $(\neg\, x \vee y)$, $(x \vee \neg\, y)$ and $(\neg\, x \vee \neg\, y)$, respectively. Indeed, c is satisfied for some truth assignment if and only if exactly one the clauses c_2, c_3 or c_4 is unsatisfied for the same truth assignment. In this way, we can calculate the number of unsatisfying truth assignments in ϕ' when exactly one the clauses c_2, c_3 or c_4 is unsatisfied as the subtraction between the number of unsatisfied clauses of all the truth assignments in ϕ' with exactly one the clauses c_2, c_3 or c_4 and the number of unsatisfied clauses of all the truth assignments in ϕ'.

Since the number of unsatisfying truth assignments for the Boolean formula ϕ when c is unsatisfied is mutually disjoint with the number of unsatisfying truth assignments for the Boolean formula ϕ when c is satisfied, then we are able to count the number of unsatisfying truth assignments for the Boolean formula ϕ using the previous steps. We do this just

selecting a clause $(x \vee y)$, counting the number of unsatisfying truth assignments when the clauses $(x \vee y)$,$(\rightharpoonup x \vee y)$, $(x \vee \rightharpoonup y)$ and $(\rightharpoonup x \vee \rightharpoonup y)$ are unsatisfied, and finally returning the sum of all these values. We show this polynomial time computation in the Algorithm 4, where we also check whether the given instance is appropriate according to the Definition 10. ◀

ALGORITHM 4: *SOLUTION–ALGO*

Data: (n, m, C) where (n, m, C) is an instance of $\#MONOTONE\text{–}2UNSAT$
Result: A polynomial time algorithm

if $CHECK\text{–}ALGO(n, m, C) =$ "no" **then**
  // (n, m, C) is not an appropriate instance of $\#MONOTONE\text{-}2UNSAT$
  return "no";
end
if $m = 1$ **then**
  return 1;
end
// Create an array C' of length $m + 1$
$C' \longleftarrow Array(m + 1)$;
$\{x, y\} \longleftarrow C[m]$;
foreach $i \leftarrow 1$ **to** $m - 1$; $C[i] = \{p, q\}$ **do**
  if $x \leq 0 \vee y \leq 0 \vee p \leq 0 \vee q \leq 0$ **then**
    // (n, m, C) is not an appropriate instance of $\#MONOTONE\text{-}2UNSAT$
    return "no";
  end
  $C'[i] \longleftarrow C[i]$;
end
// 0 represents the constant true value
$C'[m] \longleftarrow \{x, 0\}$;
$C'[m + 1] \longleftarrow \{0, y\}$;
/* Count the number of unsatisfied clauses for all the truth assignments without the clause $C[m]$ */
$temp \longleftarrow POLY\text{–}ALGO(n, m + 1, C')$;
// Count the number of unsatisfying truth assignments when $C[m]$ is unsatisfied
$count1 \longleftarrow POLY\text{–}ALGO(n, m, C) - temp$;
// Count the number of unsatisfying truth assignments when $C[m]$ is satisfied
$C[m] \longleftarrow \{-x, y\}$;
$count2 \longleftarrow POLY\text{–}ALGO(n, m, C) - temp$;
$C[m] \longleftarrow \{x, -y\}$;
$count3 \longleftarrow POLY\text{–}ALGO(n, m, C) - temp$;
$C[m] \longleftarrow \{-x, -y\}$;
$count4 \longleftarrow POLY\text{–}ALGO(n, m, C) - temp$;
// Final result
return $count1 + count2 + count3 + count4$;

▶ **Theorem 12.** $P = NP$.

Proof. It is known that if some *#P–complete* is in FP, then every *#P–complete* is in FP, because FP is closed under polynomial time counting reductions [21]. However, if this happens, then $P = NP$, due to $\#SAT$ is in $\#P$ [24]. Certainly, we will be able to solve SAT in polynomial time, since a Boolean formula is satisfiable if and only if the number of satisfying truth assignments is greater than 0. Therefore, this is a direct consequence of Theorem 11. ◀

4 Conclusions

No one has been able to find a polynomial time algorithm for any of more than 300 important known *NP–complete* problems [12]. A proof of $P = NP$ will have stunning practical consequences, because it leads to efficient methods for solving some of the important problems in NP [7]. The consequences, both positive and negative, arise since various *NP–complete* problems are fundamental in many fields [7].

Cryptography, for example, relies on certain problems being difficult. A constructive and efficient solution to an *NP–complete* problem such as SAT will break most existing cryptosystems including: Public-key cryptography [16], symmetric ciphers [19] and one-way functions used in cryptographic hashing [9]. These would need to be modified or replaced by information-theoretically secure solutions not inherently based on *P–NP* equivalence.

There are positive consequences that will follow from rendering tractable many currently mathematically intractable problems. For instance, many problems in operations research are *NP–complete*, such as some types of integer programming and the traveling salesman problem [12]. Efficient solutions to these problems have enormous implications for logistics [7]. Many other important problems, such as some problems in protein structure prediction, are also *NP–complete*, so this will spur considerable advances in biology [5].

Since all the *NP–complete* optimization problems become easy, everything will be much more efficient [11]. Transportation of all forms will be scheduled optimally to move people and goods around quicker and cheaper [11]. Manufacturers can improve their production to increase speed and create less waste [11]. Learning becomes easy by using the principle of Occam's razor: We simply find the smallest program consistent with the data [11]. Near perfect vision recognition, language comprehension and translation and all other learning tasks become trivial [11]. We will also have much better predictions of weather and earthquakes and other natural phenomenon [11].

There would be disruption, including maybe displacing programmers [17]. The practice of programming itself would be more about gathering training data and less about writing code [17]. Google would have the resources to excel in such a world [17]. But such changes may pale in significance compared to the revolution an efficient method for solving *NP–complete* problems will cause in mathematics itself [7]. Research mathematicians spend their careers trying to prove theorems, and some proofs have taken decades or even centuries to find after problems have been stated [1]. For instance, Fermat's Last Theorem took over three centuries to prove [1]. A method that is guaranteed to find proofs to theorems, should one exist of a "reasonable" size, would essentially end this struggle [7].

References

1 Scott Aaronson. P $\stackrel{?}{=}$ NP. *Electronic Colloquium on Computational Complexity, Report No. 4*, 2017.

2 Carme Álvarez and Birgit Jenner. A Very Hard Log-Space Counting Class. *Theor. Comput. Sci.*, 107(1):3–30, January 1993. doi:10.1016/0304-3975(93)90252-O.

3 Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

4 Theodore Baker, John Gill, and Robert Solovay. Relativizations of the $\mathcal{P} =? \mathcal{NP}$ Question. *SIAM Journal on computing*, 4(4):431–442, 1975. doi:10.1137/0204037.

5 Bonnie Berger and Tom Leighton. Protein Folding in the Hydrophobic-Hydrophilic (HP) Model is NP-complete. *Journal of Computational Biology*, 5(1):27–40, 1998. doi:10.1145/279069.279080.

6 Allan Borodin, Stephen A. Cook, and Nick Pippenger. Parallel Computation for Well-Endowed Rings and Space-Bounded Probabilistic Machines. *Inf. Control*, 58(1–3):113–136, July 1984. doi:10.1016/S0019-9958(83)80060-6.

7 Stephen A. Cook. The P versus NP Problem, April 2000. In Clay Mathematics Institute at http://www.claymath.org/sites/default/files/pvsnp.pdf. Retrieved June 26, 2020.

8 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

9 Debapratim De, Abishek Kumarasubramanian, and Ramarathnam Venkatesan. Inversion Attacks on Secure Hash Functions Using SAT Solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 377–382. Springer, 2007. doi:10.1007/978-3-540-72788-0_36.

10 Vinay Deolalikar. P $\neq$ NP, 2010. In Woeginger Home Page at https://www.win.tue.nl/~gwoegi/P-versus-NP/Deolalikar.pdf. Retrieved June 26, 2020.

11 Lance Fortnow. The Status of the P Versus NP Problem. *Commun. ACM*, 52(9):78–86, September 2009. doi:10.1145/1562164.1562186.

12 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman and Company, 1 edition, 1979.

13 William I. Gasarch. Guest column: The second P $\stackrel{?}{=}$ NP poll. *ACM SIGACT News*, 43(2):53–77, 2012. doi:10.1145/2261417.2261434.

14 Juris Hartmanis and John E. Hopcroft. Independence Results in Computer Science. *SIGACT News*, 8(4):13–24, October 1976. doi:10.1145/1008335.1008336.

15 Juris Hartmanis and Stephen R. 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.

16 Satoshi Horie and Osamu Watanabe. Hard instance generation for SAT. *Algorithms and Computation*, pages 22–31, 1997. doi:10.1007/3-540-63890-3_4.

17 Russell Impagliazzo. A personal view of average-case complexity. In *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*, pages 134–147. IEEE, 1995. doi:10.1109/SCT.1995.514853.

18 Richard J. Lipton. Efficient checking of computations. In *STACS 90*, pages 207–215. Springer Berlin Heidelberg, 1990. doi:10.1007/3-540-52282-4_44.

19 Fabio Massacci and Laura Marraro. Logical Cryptanalysis as a SAT Problem. *Journal of Automated Reasoning*, 24(1):165–203, 2000. doi:10.1023/A:1006326723002.

20 Pascal Michel. A survey of space complexity. *Theoretical computer science*, 101(1):99–132, 1992. doi:10.1016/0304-3975(92)90151-5.

21 Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

22 Alexander A. Razborov and Steven Rudich. Natural Proofs. *J. Comput. Syst. Sci.*, 55(1):24–35, August 1997. doi:10.1006/jcss.1997.1494.

23 Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

24 Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, (2):189–201, 1979. doi:10.1016/0304-3975(79)90044-6.

25 Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, (3):410–421, 1979. doi:10.1137/0208032.