Program Correctness Checking
...and the design of programs
that check their work¹

Manuel Blum and Sampath Kannan

TR-88-013

December, 1988

ABSTRACT

A program correctness checker is an algorithm for checking the output of a computation. This paper defines the concept of a program checker. It designs program checkers for a few specific and carefully chosen problems in the class \( P \) of problems solvable in polynomial time. It also applies methods of modern cryptography, especially the idea of a probabilistic interactive proof, to the design of program checkers for group theoretic computations. Finally it characterizes the problems that can be checked.

¹This paper extracts from work in Blum[B2] and Kannan[K2].
Support was provided in part by NSF grant CCR85-13926 and in part by the International Computer Science Institute, Berkeley.

Authors' e-mail addresses: blum@ernie.berkeley.edu and kannan@ernie.berkeley.edu.
Program Correctness Checking ...
and the design of programs that check their work

Manuel Blum
Sampath Kannan
Computer Science Division
University of California at Berkeley
94720

1. History

The ideas in this paper arise from cryptography, probabilistic algorithms, and program testing. Particularly important for this work are the probabilistic interactive proofs of Goldwasser, Micali and Rackoff [GMR], and their spinoffs. As will be seen, several of the correctness checkers constructed in this paper use probabilistic interactive proofs as a kind of scaffolding. Equally important for this work are the papers on randomized algorithms of Rabin [R] and Freivald [F]. The latter, remarkably enough, includes excellent program checkers for integer, polynomial, and matrix multiplication. Finally, the works of Blum and Raghavan [BR], Budd and Angluin [BA], and Weyuker [W] are relevant in that they too seek to give program testing a rigorous mathematical basis.

2. Program Checkers

Let \( \pi \) denote a (computational) decision and/or search problem. For \( x \) an input to \( \pi \), let \( \pi(x) \) denote the output of \( \pi \). Let \( P \) be a program (supposedly) for \( \pi \) that halts on all instances of \( \pi \). We say that such a program \( P \) has a bug if for some instance \( x \) of \( \pi \), \( P(x) \neq \pi(x) \).

Define an (efficient) program checker \( C_\pi \) for problem \( \pi \) as follows: \( C_\pi^P(i ; k) \) is any probabilistic (expected-poly-time) oracle Turing machine that satisfies the following conditions, for any program \( P \) (supposedly for \( \pi \)) that halts on all instances of \( \pi \), for any instance \( i \) of \( \pi \), and for any positive integer \( k \) (the so-called "security parameter") presented in unary:

1. If \( P \) has no bugs, i.e., \( P(x) = \pi(x) \) for all instances \( x \) of \( \pi \), then with probability\(^2\) greater or equal to \( 1 − 1/2^k \), \( C_\pi^P(i ; k) = \text{CORRECT} \) (i.e., \( P(i) \) is \text{CORRECT}).

2. If \( P(i) \neq \pi(i) \), then with probability greater or equal to \( 1 − 1/2^k \), \( C_\pi^P(i ; k) = \text{BUGGY} \) (i.e., \( P \) is \text{BUGGY}).

Some remarks are in order:

i. The running time of \( C \) above includes whatever time it takes \( C \) to submit inputs to and receive outputs from \( P \), but excludes the time it takes for \( P \) to do its computations.

---

1 This paper extracts from work in Blum [B2] and Kannan [K2].
Support was provided in part by NSF grant CCR85-13925 and in part by the International Computer Science Institute at Berkeley.
Authors' e-mail addresses: blum@ernie.berkeley.edu and kannan@ernie.berkeley.edu

2 This probability is computed over the sample space of all finite sequences of coin flips that \( C \) could have tossed.
ii. In the above definition, if \( P \) has bugs but \( P(I) = \pi(I) \), i.e. buggy program \( P \) gives the correct output on input \( I \), then \( C^P_k(I; k) \) may output \text{CORRECT} \text{ or BUGGY}.

It is assumed that any program \( P \) for problem \( \pi \) halts on all instances of \( \pi \). This is done in order to help focus on the problem at hand. In general, however, programs do not always halt, and the definition of a "bug" must be extended to cover programming errors that slow a program down or cause it to diverge altogether. In this case, the definition of a program checker must also be extended to require the additional condition:

3. If \( P(x) \) exceeds a precomputed bound \( \Phi(x) \) on the running time, for \( x = I \) or any other value of \( x \) submitted by the checker to the oracle, then the program checker is to sound a warning, namely \( C^P_k(I; k) = \text{TILE} \).

In the remainder of this paper, it is assumed that any program \( P \) for a problem \( \pi \) halts on all instances of \( \pi \), so condition 3 is everywhere suppressed.

Regarding this model for ensuring program correctness the question naturally arises: if one cannot be sure that a program is correct, how then can one be sure that its checker is correct? This is a very serious problem! One solution is to prove the checker correct. Sometimes, this is easier than proving the original program correct, as in the case of the Extended GCD checker of section 6. Another possibility is to try and make the checker to some extent independent of the program it checks. To this end, we make the following definition: Say that (probabilistic) program checker \( C \) has the \text{little oh} property with respect to program \( P \) if and only if the (expected) running time of \( C \) is little oh of the running time of \( P \). We shall generally require that a checker have this little oh property with respect to any program it checks. The principal reason for this is to ensure that the checker is programmed \text{differently} from the program it checks.

3. Example: Graph Isomorphism

The graph isomorphism decision problem is defined as follows:

\text{Graph Isomorphism (GI):}
\text{Input:} \quad \text{Two graphs } G \text{ and } H.
\text{Output:} \quad \text{YES if } G \text{ is isomorphic to } H; \text{ NO otherwise.}

Our checker is an adaptation of Goldreich, Micali and Wigderson's [GMW] demonstration that Graph Isomorphism has interactive proofs. The [GMW] model relies on the existence of an all-powerful prover. The latter is replaced here by the program being checked. As such, ours is a concrete application of their abstract idea. Indeed, the following program checker is a sensible practical way to check computer programs for graph isomorphism. The checker, \( C^P_{GI}(G, H; k) \), checks program \( P \) on input graphs \( G \) and \( H \):

\begin{verbatim}
Begin
Compute \( P(G, H) \).
\end{verbatim}

---

3 The first author was privileged to hear Dr. Warren S. McCulloch at the First Bionics Symposium, where he described how farmers at a county fair weigh pigs: "First they lay a plank across a rock and set a pig at one end. Then they heap rocks at the other end until the rocks balance the pig. Finally, they guess the weight of the rocks and compute the weight of the pig!"

Our approach to program checking is similar: Instead of proving a program checker correct, we test it. Then we \text{prove} that a (debugged) program checker will discover all output errors.

4 A second reason (to ask that the checker have the "little oh" property) arises whenever the program checker is the type that runs the program \( P \) just once (to determine \( O = P(I) \)). In that case, \( P \) and \( C \) can be run consecutively without increasing the asymptotic running time over that of running just \( P \).
If \( P(G,H) = YES \), then
Use \( P \) (as if it were bug-free) to search for an ‘isomorphism’ from \( G \) to \( H \).
(This is done by a standard self reduction as in Hoffmann [H].)
Check if the resulting correspondence is an isomorphism.
If not, return \emph{BUGGY}; if yes, return \emph{CORRECT}.

If \( P(G,H) = NO \), then
Do \( k \) times:
Toss a fair coin.
If coin = heads then
generate a random\(^5\) permutation \( G' \) of \( G \).
Compute \( P(G,G') \).
If \( P(G,G') = NO \), then return \emph{BUGGY}.

If coin = tails then
generate a random permutation \( H' \) of \( H \).
Compute \( P(G,H') \).
If \( P(G,H') = YES \), then return \emph{BUGGY}.

End-do
Return \emph{CORRECT}.

End

The above program checker correctly tests any computer program \emph{whatsoever} that is purported to solve the graph isomorphism problem. Even the most bizarre program designed specifically to fool the checker will be caught, when it is run on any input that causes it to output an incorrect answer. The following Theorem formally proves this:

**Theorem:** The program checker for Graph Isomorphism runs efficiently and works correctly (as specified). Formally:
\( C_{GI}^P \) is efficient.
Let \( P \) be any decision program (a program that halts on all inputs and always outputs \emph{YES} or \emph{NO}). Let \( G \) and \( H \) be any two graphs. Let \( k \) be a positive integer.
If \( P \) is a correct program for \( GI \), i.e. one without bugs, then \( C_{GI}^P (G,H;k) \) will definitely output \emph{CORRECT}.
If \( P(G,H) \) is incorrect on this input, i.e., \( P(G,H) \neq GI(G,H) \), then \( \text{prob} \{ C_{GI}^P (G,H;k) = \text{CORRECT} \} \) is at most \( 1/2^k \).

**Proof:** Clearly, \( C_{GI}^P \) runs in expected polynomial time.
If \( P \) has no bugs and \( G \) is isomorphic to \( H \), then \( C_{GI}^P (G,H;k) \) constructs an isomorphism from \( G \) to \( H \) and (correctly) outputs \emph{CORRECT}.
If \( P \) has no bugs and \( G \) is not isomorphic to \( H \), then \( C_{GI}^P (G,H;k) \) tosses coins. It discovers that \( P(G,G') = YES \) for all \( G' \), and \( P(G,H') = NO \) for all \( H' \), and so (correctly) outputs \emph{CORRECT}.
If \( P(G,H) \) outputs an incorrect answer, there are two cases:

1. If \( P(G,H) = YES \) but \( G \) is not isomorphic to \( H \) then \( C \) fails to construct an isomorphism from \( G \) to \( H \), and so \( C \) (correctly) outputs \emph{BUGGY}.

Finally, the most interesting case:

---

\(^5\) By a "random" permutation we mean that every permutation of \( G \), i.e., every relabeling of the \( n \) nodes of \( G \) with the integers \( 1, \ldots, n \), is equally likely.
2. If \( P(G,H) = NO \) but \( G \) is isomorphic to \( H \), then the only way that \( C \) will return \textit{CORRECT} is if \( P(G,G' \text{ or } H') = YES \) whenever the coin comes up heads, \( NO \) when it comes up tails. But \( G \) is isomorphic to \( H \). Since the permutations of \( G \) and \( H \) are random, \( G' \) has the same probability distribution as \( H' \). Therefore, \( P \) correctly distinguishes \( G' \) from \( H' \) only by chance, i.e., for just 1 of the \( 2^k \) possible sequences of \( T \)'s coin tosses.

Qed

4. Beigel's Trick

Richard Beigel [B1] has pointed out to these authors the following fundamental fact:

**Theorem (Beigel's trick):** Let \( \pi_1, \pi_2 \) be two polynomial-time equivalent computational (decision or search) problems. Then from any efficient program checker \( C_{\pi_1} \) for \( \pi_1 \) it is possible to construct an efficient program checker \( C_{\pi_2} \) for \( \pi_2 \).

**Remark:** The proof requires that \( \pi_1 \) be equivalent to \( \pi_2 \). Observe that \( \pi_1 < \pi_2 \) (by Karp or Cook reducibility) implies that from any efficient algorithm for \( \pi_2 \) one can construct an efficient algorithm for \( \pi_1 \). This suggests, perhaps, that from any efficient checker for \( \pi_2 \) one should be able to construct an efficient checker for \( \pi_1 \) if only \( \pi_1 < \pi_2 \). This is NOT necessarily true: Observe that Group Isomorphism (GI) < Extended Group Isomorphism (EGI), where

1. groups are given by multiplication tables, and
2. GI differs from EGI in that a YES answer in the former is an explicit
   isomorphism in the latter.

We know an efficient checker for EGI but not for GI.

**Proof:** Our proof of this theorem will be for the special case in which decision problems \( \pi_1, \pi_2 \) are polynomial-time equivalent by Karp-reductions, but it goes through as well for search/optimization problems \( \pi_1, \pi_2 \) that are polynomial-time equivalent by Cook-reductions.

Let \( f_{ij} \) be two polynomial-time functions that map YES-instances of \( \pi_i \) to YES-instances of \( \pi_j \) and NO-instances of \( \pi_i \) to NO-instances of \( \pi_j \), for \( \{i,j\} = \{1,2\} \). In what follows, \( P_i \) will denote a program for \( \pi_i \) and \( I_i \) will denote an instance of \( \pi_i \), for \( i \in \{1,2\} \).

\[ C_{\pi_1}^{P_1} \left( I_{2}; k \right) \] works as follows: it checks if \( P_2 (I_2) = \pi_2 (I_2) \) by verifying two conditions:

1. \( P_2 (I_2) = P_2 (f_{12} (f_{21} (I_2))) \), and
2. setting \( I_1 = \text{def} f_{21} (I_2) \), and defining \( P_1 \) by \( P_1 (x_1) = \text{def} P_2 (f_{12} (x_1)) \) for all instances \( x_1 \) of \( \pi_1 \),
   check the correctness of \( P_1 (I_1) = \pi_1 (I_1) \) (and therefore of \( P_2 (I_2) = \pi_2 (I_2) \)) by using \( C_{\pi_1}^{P_1} \left( I_1; k \right) \).

If conditions one or two fail, then \( C_{\pi_1}^{P_1} \left( I_{2}; k \right) \) := BUGGY. Otherwise, \( C_{\pi_1}^{P_1} \left( I_{2}; k \right) \) := CORRECT.

Observe that if \( P_2 \) is correct (i.e., \( P_2 = \pi_2 \)), then conditions one and two hold. In particular, \( P_1 \) is correct, whence \( C_{\pi_1}^{P_1} \left( I_1; k \right) = \text{CORRECT} \). So \( C_{\pi_1}^{P_1} \left( I_2; k \right) = \text{CORRECT} \).

On the other hand, if \( P_2 (I_2) \neq \pi_2 (I_2) \), then either condition one fails, i.e., \( P_2 (I_2) \neq P_2 (f_{12} (f_{21} (I_2))) \), in which case \( C_{\pi_1}^{P_1} \left( I_{2}; k \right) \) := BUGGY, or else condition 1 holds, whence

\[ P_1 (I_1) = P_1 (f_{21} (I_2)) = P_2 (f_{12} (f_{21} (I_2))) \text{ since } P_1 = \text{def} P_2 f_{12} , \]

\[ = P_2 (I_2) \text{ because condition one holds} \]

\[ = \pi_2 (I_2) \text{ by assumption} \]

\[ = \pi_1 (f_{21} (I_2)) = \pi_1 (I_1) , \]
in which case $C_{i; k}^k (l_1; k)$, and therefore also $C_{i; k}^k (l_2; k)$ will correctly return BUGGY with high probability, i.e., with probability of error $\leq 1/2^k$.

Qed

Problems that are polynomial-time equivalent to Graph Isomorphism include that of finding generators for the automorphism group of a graph, determining the order of the automorphism group of a graph, and counting the number of isomorphisms between two graphs. It follows from Beigel's trick that all these problems have efficient program checkers.

5. Checkers for Group Theoretic Problems

Many group theoretic problems have checkers resembling that for graph isomorphism. Subsection 5.1 shows this for two fairly general classes of examples. 5.2 gives a general approach to checker construction that works particularly well for group theoretic problems.

Why all the work on group theoretic problems? One way to show off the program checker concept is to apply it to a large interesting area in which there is substantial interest to get provably correct results. The group theoretic area, on account of both the enormous energy that has gone into the classification of finite simple groups and the interest of mathematicians to have credible computer generated output, seems a natural place to focus attention.

5.1. The Equivalence Search and Canonical Element Problems

The problems (and corresponding checkers) described in this subsection are all stated in terms of a set $S$ of elements and a group $G$ acting on $S$.

For $a, b$ in $S$, define $a \equiv_G b$ if and only if $g(a) = b$ for some $g$ in $G$.

Let $ESP(S, G)$ denote the

- **Equivalence Search Problem**
  - Input: $a, b$ in $S$
  - Output: $g$ such that $g(a) = b$ if $a \equiv_G b$;
  - NO otherwise.

**Proposition:** Let $ESP(S, G)$ be the Equivalence Search Problem$^6$ for given $S$ and $G$. Suppose there exists an efficient probabilistic algorithm to find a "random" $g$ in $G$, where random means that all $g$ in $G$ are equally likely. Then there is an efficient program checker $C_{ESP(S, G)}$ for the problem $ESP(S, G)$.

Examples of the Equivalence Search Problem include graph Isomorphism, quadratic residuosity, a generalization of discrete log and games such as Rubik's cube. Other examples arise in knot theory, block designs, codes, matrices over GF (q), Latin Squares [L2, p. 32] and in applications of Burnside and Polya.

---

$^6$ Related to the Equivalence Search Problem is the Equivalence Decision Problem defined by:

- **Equivalence Decision Problem (EDP)**
  - Instance: $a, b$ in $S$
  - Question: Is $a \equiv_G b$?

The search problem, not the decision problem, is required in the above theorem since the decision problem is not expected to be an NP-Search-Solving-Decision problem. Why? Recall that for $N$ a positive integer, $\mathbb{Z}_N$ denotes the group of positive integers less than $N$ that are relatively prime to $N$ under multiplication mod $N$. For $p$ a prime, let $S = \mathbb{Z}_p$ and $G = \mathbb{Z}_{p-1}$, where the action of $g$ in $G$ on $a$ in $S$ maps $a$ to $a^g \mod p$. Observe that $a \equiv_G b$ if and only if $b = a^g \mod p$ for some $g$ in $\mathbb{Z}_{p-1}$. To find $g$ is to solve the discrete log problem, which in cryptographic circles is believed to not be solvable in polynomial time, even given an oracle for factoring [A]. On the other hand, the EDP is solvable in polynomial time given an oracle for factoring. The proof consists in showing that $b = a^g \mod p$ for some $g$, if and only if order($b$) | order($a$). This is because $x^{\text{order}(a)} - 1 \mod p$ has exactly order($a$) solutions, namely $a, a^2, \ldots, a^{\text{order}(a)}$. Finally, order($a$) and order($b$) can be determined from the factorization of $p - 1$. 

theorems [PR].

• **Canonical Element Problem (CEP)**
  
  **Input:** \( a \) in \( S \)
  
  **Output:** \((c, g)\) where \( c \) is a canonical element (the unique canonical element) in the equivalence class of \( a \), and \( g \) in \( G \) satisfies \( g(a) = c \).

Proposition: There is an efficient program checker for the canonical element problem, provided there is a probabilistic procedure to select a random \( g \) in \( G \) efficiently.

Remark: If the CEP program should fail by having two or more canonical elements in some class, then we define the (true) canonical element of that class to be the unique element, if any, to which more than half the elements of the class are mapped by the program.

5.2. **The Vorpal Blade Went Snicker-Snack**

A 1-2 approach is useful for constructing program checkers for group theoretic problems:

1. Design an interactive protocol (cf. Goldwasser, Micali, and Rackoff [GMR]) for proving correctness of an algorithm’s output.
2. Modify said interactive protocol into a checker.

It is possible to design checkers for several group theoretic problems in this way. An example is the checker for

**The Group Intersection Problem:**

**Input:** Two permutation groups, \( G \) and \( H \), specified by generators. The generators are presented as permutations of \([1, \ldots, n]\).

**Output:** Generators for \( G \cap H \).

No probabilistic polynomial time algorithm is known for solving this problem, which is not surprising since graph isomorphism is polynomial time reducible to group intersection. Here we present the checker for the group intersection problem as developed using the 1-2 approach.

**IP Protocol**

**Begin**

1. The prover sends the verifier a set of permutations of \([1, \ldots, n]\) which supposedly generate \( G \cap H \).
2. The verifier checks that the elements sent by the prover actually lie in \( G \cap H \). This involves testing membership in \( G \) and \( H \), which the verifier can do. As a consequence, the verifier is convinced that the elements sent by the prover either generate \( G \cap H \) or a proper subgroup of it.

The next phase of the protocol is aimed towards giving the verifier a random element of \( G \cap H \). Before going into it we introduce the following (standard) notation: with \( G \) and \( H \) as above \( GH \) represents the set of permutations \( \{ \pi \mid \pi = ab \text{ where } a \in G \text{ and } b \in H \} \). Here is the continuation of the protocol.

3. The verifier sends to the prover an element, \( \pi \), of \( GH \), which he obtains by selecting random elements\(^7\) of \( G \) and \( H \) and multiplying them together.
4. The prover sends back a ‘factorization’ of \( \pi \) as \( a'b' \) where \( a' \in G \) and \( b' \in H \).

\(^7\) This is doable as explained later in this section or more clearly in [H].
Lemma: \( a^{-1}a' \) is a random element of \( G \cap H \).

Proof: First we show that \( a^{-1}a' \in G \cap H \):
\[
a \ b = \pi = a'b' \Rightarrow a^{-1}a' = b \ b^{-1}
\]

In the last equation we have an element of \( G \) equalling an element of \( H \). Thus this element must belong to \( G \cap H \).

Next, we show that every element of \( G \cap H \) is equally likely to be generated. Let \( x \in G \cap H \) and let \( \pi = ab \) be an element of \( GH \). Then \( \pi = (ax)(x^{-1}b) \). If the prover sends back this factorization of \( \pi \) then the verifier will generate the element \( x \). Furthermore, no other factorization that the prover sends will result in the verifier generating \( x \). Thus for each element of \( G \cap H \) there is a unique factorization that the prover can send that will generate \( x \) for the verifier. Since the verifier initially created the product randomly, the prover's factorization is equally likely to yield any element of \( G \cap H \).

Qed

5. Now that the verifier has a random element of \( G \cap H \), he tests it for membership in the group generated by the elements sent by the prover. If that group is a proper subgroup of \( G \cap H \) its size is at most \( 1G \cap H | \). Hence the verifier has a probability of at least \( 1/2 \) of detecting this. If however the elements actually generate \( G \cap H \) the verifier will be convinced of this after a number of trials.

End

Converting the IP Protocol into a Checker

The verifier in the above protocol asks the prover to 'factor' certain elements of \( GH \). To convert this IP protocol into a checker presents a problem in that a checker is only allowed to run the program on various inputs to the program. Here we are dealing with a group intersection program, so the checker can only ask for the intersection of various pairs of permutation groups. If the Factorization Search Problem (FSP) were shown equivalent to the group intersection problem then we could pose the factorization questions as group intersection questions and we would have our desired checker.

Factorization Search Problem (FSP)

Input: Two permutation groups \( G \) and \( H \) specified by generators, and a permutation \( \pi \).

Output: No, if \( \pi \) is not in \( GH \).

\[ a, b \text{ such that } a \in G \text{ and } b \in H \text{ and } ab = \pi \text{ otherwise.} \]

The Factorization Decision Problem (FDP) is known to be equivalent to group intersection, cf. Hoffmann [H].

Factorization Decision Problem (FDP)

Instance: Generators for two permutation groups \( G \) and \( H \) acting on the set \([1..n]\) and a permutation \( \pi \) in the symmetric group on \( n \) letters.

Question: Can \( \pi \) be factored into \( \text{ab} \) such that \( a \) in \( G \) and \( b \) in \( H \)?

We are left with the problem of showing that FDP and FSP are equivalent. We now show how to use FDP to solve FSP. The reduction in the other direction is trivial.

Using FDP to Solve FSP

Assume that we have strong generators for \( G \) and \( H \) as defined in Furst, Hopcroft, and Lux [FHL]. This is not a serious assumption because any set of generators can be converted to a set of strong
generators in polynomial time.

Here is a brief description of the notion of strong generators \( M_G \) for the group \( G \). \( M_G \) is an \( n \times n \) matrix where \( n \) is the size of the permutation domain. The matrix has no entries below the diagonal. Above the diagonal, in position \( ij \) we have an entry iff there is a permutation in \( G \) that fixes (pointwise) the elements \( 1,2,\ldots,i-1 \) and moves \( i \) to \( j \). In case such a permutation exists, the \( ij \) entry is any such permutation in \( G \). It is convenient (and customary) to make the diagonal entries be the identity permutation.

Some properties of this representation are given here without proof. Every element of \( G \) can be expressed in a unique way as a product, \( \pi = \pi_n \cdots \pi_1 \) where \( \pi_i \) is from row \( i \) of \( M_G \). We are using the convention here that in a string of permutations the leftmost one acts first and the rightmost one last. As a consequence of the previous fact, \(|G|\) is the product of the numbers of non-empty entries in each row of \( M_G \). Another consequence is that a random element of \( G \) can be obtained by multiplying together random elements in each of the rows of \( M_G \). Also, \( G_1 \), the subgroup of \( G \) that fixes the point 1 is generated by the entries in rows 2 through \( n \) of \( M_G \). Finally, membership in \( G \) for a permutation \( \sigma \) can be tested as follows: if \( \sigma \) moves 1 to \( j \), we look in position \( 1j \) for an entry. If none exists \( \sigma \) is not in \( G \). Otherwise, if \( \pi_i \) is the entry \( \sigma_i^{-1} \) fixes the point 1 and we move on to the second row and check it for membership in \( G_1 \). Proceeding thus we will either find that \( \sigma \) is not in \( G \) or find an expression for \( \sigma \) as a product of entries in \( M_G \).

Suppose now that \( \pi \) is in \( GH \). We consider \( H_1 \), the subgroup of \( H \) consisting of all permutations that fix the point 1. Since \( \pi \) is in \( GH \), \( \pi = ab \) with \( a \) in \( G \) and \( b \) in \( H \). \( b \) is equal to some product, \( \sigma_1 \sigma_2 \cdots \sigma_i \) where \( \sigma_i \) is in the \( i \)th row of \( M_H \). Thus there is a permutation, \( \sigma_1 \), in the first row of \( M_H \) such that \( ab \sigma_1^{-1} \) is in \( GH \). We can use the oracle for \( \text{FDP} \) to find out which entry in the first row of \( M_H \) has the above property. If this entry is \( \sigma_1 \), we consider \( \pi \sigma_1^{-1} \) and factor it in \( GH \). A factorization in \( GH \) will yield a factorization in \( G \) of \( \pi \). It can be seen that if this technique is applied recursively it yields a factorization for \( \pi \) in \( GH \). This completes the reduction and shows that the \( \text{IP} \) protocol described can be converted into a checker.

6. Problems in \( P \)

In this section, some program checkers use their oracle just once (to determine \( O = P(I) \)) rather than several times. In such cases, instead of the program checker being denoted by \( C^e_p(I,O,k) \), it will be denoted by \( C_p(I,O,k) \). The latter notation has the advantage of clarifying what must be tested for. In cases where the checker is nonprobabilistic, it will be denoted by \( C_p(I,O) \) instead of \( C_p(I,O,k) \).

Many problems in \( P \) have efficient program checkers, and it is a challenge to find them. In what follows, we give a fairly complete description of program checkers for just four problems in \( P \): Extended GCD (because it has one of the oldest nontrivial algorithms on the books), Sorting (because it is one of the most frequently run algorithms), Matrix rank (because it is most unusual in that it seems to require a multicalle checker with two-sided error), and Containment (because many checkers seem to require containment checks).

6.1. A Program Checker for Extended GCD

**Extended GCD**

**Input:** Two positive integers \( a, b \).

**Output:** \( d = \text{gcd}(a,b) \), and integers \( u, v \) such that \( a \cdot u + b \cdot v = d \).

Observe that **Extended GCD** is a specific computational problem, not an algorithm. In particular, it does not have to be solved by Euclid's algorithm. Problems **GCD** and **Extended GCD** both output \( d = \text{gcd}(a,b) \), but **Extended GCD** also outputs two integers \( u, v \) such that \( a \cdot u + b \cdot v = d \). We know how to check programs for **Extended GCD** but not **GCD**.
For our model of computation, we choose a standard RAM and count only arithmetic operations +, - , * , / as steps.

\text{C}_{\text{Extended GCD}}
\begin{itemize}
  \item \text{Input:} positive integers \( a, b \); positive integer \( d \) and integers \( u, v \).
  \item \text{Output:} BUGGY if \( d \nmid a \) or \( d \nmid b \) or \( a \cdot u + b \cdot v \neq d \),
  otherwise (the given program \( P \) computes \text{Extended GCD} correctly).
\end{itemize}

The desired input-output relationship above can be achieved by a checker that takes just 5 steps: 2 division steps + 2 multiplication steps + 1 addition step. This checker can also be proved correct easily.

6.2. A Program Checker for Sorting

\text{Sorting}
\begin{itemize}
  \item \text{Input:} An array of integers \( X = [x_1, \ldots, x_n] \), representing a multiset.
  \item \text{Output:} An array \( Y \) consisting of the elements of \( X \) listed in non-decreasing order.
\end{itemize}

A checker for \text{Sorting} must do more than just check that \( Y \) is in order. It must also check that \( X = Y \) as multisets.

\text{C}_{\text{Sorting}}
\begin{itemize}
  \item \text{Input:} Two arrays of integers \( X = [x_1, \ldots, x_n] \) and \( Y = [y_1, \ldots, y_n] \)
  \item \text{Output:} BUGGY if \( Y \) is not in order or if \( X \neq Y \) as multisets,
  otherwise (the given program \( P \) correctly sorts).
\end{itemize}

This check is easy in a RAM model of computation, but a RAM does not reflect many sorting scenarios. The following one does.

\textbf{Model of computation:} The computer has a fixed number of tapes, including one that contains \( X \) and another that contains \( Y \). \( X \) and \( Y \) each have at most \( n \) elements, and each such element is an integer in the range \( [0,a] \). The random access memory has \( O(lg n + lg a) \) words of memory, and each of its words is capable of holding any integer in the range \( [0,a] \), in particular each can hold any element of \( X \cup Y \).

- Single precision operations: \( +, -, \times, /, <, \leq \) each take one step. Here, / denotes "integer divide."

- Multi-precision operations: \( +, -, \leq, \) on integers that are \( m \) words long take \( m \) steps;
  \( \times, / \) on integers that are \( m \) words long take \( m^2 \) steps.

In addition the machine can do the usual tape operations: Each shift of a tape, and each copy of a word on tape to a word in RAM or vice versa counts 1 step.

In this model of computation, it is easy to check that \( Y \) is in order in just \( O(n) \) steps. To check that \( X = Y \) as multisets can be done in probabilistic \( O(n) \) steps, but the right method depends in general on the the comparative sizes of \( a \) and \( n \). Here is a precise statement of what is wanted:

\textbf{Multiset Equality Test}
\begin{itemize}
  \item \text{Input:} Two arrays of integers \( X, Y \); and a positive integer \( k \).
  \item \text{Output:} YES if \( X \) and \( Y \) represent the same multiset, NO otherwise, with probability of error \( \leq 1/2^k \).
\end{itemize}

The latter means that for any two sets \( X \) and \( Y \) selected by adversary, the probability of an incorrect output (measured over the distribution of coin-toss-sequences generated by the algorithm) should be at most \( 1/2^k \).

\textbf{Method 1:} This method (but not the specific and important choice of hash function) was first suggested by Wegman and Carter [WC]: Compute \( n = |X| \) and check that \( |Y| = n \). If so, select a hash function \( h : Z \to \{0,1\} \) and compare \( h(x_1) + \cdots + h(x_n) \) to \( h(y_1) + \cdots + h(y_n) \).
If $h$ is random and $X \neq Y$ then with probability at least $1/2$ the above two sums will differ.\footnote{To see this, remove from $X$ and $Y$ any largest sub-multiset of elements that is common to both. The resulting $X$ and $Y$ are still the same size and their intersection is empty. Compute $\sum h(x_i)$ and compute $\sum h(y_j)$. If the two sums are equal, then setting $h(x_i) = 1$ will distinguish $X$ from $Y$; if different, then setting $h(x_i) = 0$ will distinguish the two.}

Choosing an easy-to-compute random-enough hash function is difficult\footnote{A common way to construct hash functions is to select a prime $p$ and integers $a$, $b$ at random, then to set $h(x) = a \cdot x + b \mod p$. This will not work for this problem. To see why not, suppose that addition of hash functions is done mod $p$. In that case, an adversary can choose $X \neq Y$ so that $\sum x_i = \sum y_i$, whence it follows that $\sum h(x_i) \equiv \sum h(y_j) \mod p$. This hash function also fails when ordinary addition is used.}. The Wegman and Carter hash function in particular requires a random access memory, so it cannot be implemented in the above tape model of computation. Here is a new approach that is guaranteed to work: Recall that $n = |X| = |Y|$. Let $m = n + 1$. $a$ is the largest possible value for any element of $X \cup Y$. Select prime $p$ at random from the interval $[1, 3 \cdot a \cdot \log m]$. Set $h(x) = m^x \mod p$. Observe that $X = Y$ if and only if $\sum m^x = \sum m^{x_i}$. Indeed, if $X = Y$, then $\sum (m^x \mod p) = \sum (m^{x_i} \mod p)$ for all primes $p$. If $X \neq Y$, then $\sum m^x \neq \sum m^{x_i}$ and so, as pointed out by Karp and Rabin [KR], $\sum m^x \neq \sum m^{x_i} \mod p$ for at least half of all primes in the interval $[1, 3 \cdot a \cdot \log m]$, and therefore $\sum (m^x \mod p) \neq \sum (m^{x_i} \mod p)$ for those primes. This proves that this $h$ is a random-enough hash function.

**Method 2:** This idea was first suggested by Lipton [L3] and more recently by Ravi Kannan [K1]. Let $f(z) = (z - x_1) \cdots (z - x_n)$ and $g(z) = (z - y_1) \cdots (z - y_n)$. Then $X = Y$ as multisets if and only if $f = g$. Since $f$ and $g$ are polynomials of degree $n$, either $f(z) = g(z)$ for all $z$ (if $f = g$) or $f(z) \neq g(z)$ for at most $n$ values of $z$ (if $f \neq g$). A probabilistic algorithm can decide if $f = g$ by selecting $k$ values for $z$ at random from a set of $2n$ (or more) possibilities, say from $[1, 2n]$, then comparing $f(z)$ to $g(z)$ for these $k$ values. The computations can be kept to reasonable size by doing the subtractions and multiplications modulo randomly chosen (small) primes.

**Comparison of methods 1 & 2 for checking multiset equality:**

Recall that each multiset in the range $[0, a]$. In method 1, primes are $O(a \log n)$; in method 2, they are $O(n \log a)$. The lengths of the primes are respectively $O(\log a + \log \log a)$ and $O(\log a + \log \log a)$. If $n$ is bigger than $2^n$ a simple bucket sort works in linear time and we don't need to use either of these two methods. For $n$ less than $2^n$ the primes fit in a constant number of words in method 1. In method 2 the number of words to hold a prime is $k = O(\max(1, \log n))$. The running time of method 1 is $O(n \log a)$. (We need to perform $\log a$ multiplications to compute $m^n$ each costing $1$ time step.) The running time for method 2 is $O(n k^2)$ (Since we multiply numbers that are $k$ words long).

To achieve the little oh property one must make the choice of method carefully. If $n < a$ method 2 runs in linear time since $k = 1$. We use method 2 as long as $n < a^{\log a}$. The running time of method 2 is then $O(n (\log a)^2)$ which is $O(n (\log \log n)^2)$. When $n$ gets bigger than $a^{\log a}$ the running time of method 1 which is $n \log a$ becomes $o(n \log n)$.

### 6.3. A Program Checker for Matrix Rank

**Matrix Rank**

**Input:** An $m \times n$ matrix $M$ with elements from a finite field $F$.

**Output:** An integer $r$, which is the rank of the matrix.
We assume in our model of computation that we can generate a random element of $F$ in one time step. The difficulty in designing a checker for this problem is achieving the little oh property. For instance, a commonly used operation, that of forming random linear combinations of the columns of the matrix, takes time $O(n^2)$. So this cannot be done too many times. In this subsection we present an outline of the checker.

The checker is in two parts. The first part ensures that the rank of $M$ is at least $r$ and the second part verifies that the rank is at most $r$.

6.3.1. Checking that Rank is at least $r$

To check that the rank is at least $r$, use the program to obtain (by self reduction) an $r$ by $r$ submatrix, $A$, which is supposedly of full rank. It must be checked that $A$ is really of full rank. The key fact used here is that any vector $x$ in $\text{span}(A)$, the span of the columns of $A$ has a unique representation as a linear combination of the columns of $A$ iff $A$ is of full rank. In particular if $A$ is not of full rank there is a column $a_i$ of $A$ such that any $x$ in $\text{span}(A)$ can be expressed as a linear combination of the columns with any desired coefficient for $a_i$.

Create $k$ (= the security parameter) random vectors, $y_1, \ldots, y_k$ in $\text{span}(A)$. Let $c_{ij}$ be the coefficient of column $a_i$ of $A$ in $y_j$. If $A$ is not of full rank, then for column $a_i$, as in the previous paragraph the following two cases are indistinguishable:

1. Setting $y_j = y_j - c_{ij}a_i$ and using the program to find the rank of $A$ without column $a_i$ but with the vector $y_j$ in its place.
2. Setting $y_j = y_j - b_i a_i$, where $b$ is a random element of $F$, $b \neq c_{ij}$, and using the program to find the rank of $A$ without column $a_i$ but with the vector $y_j$.

However when $A$ is of full rank these cases are clearly distinguishable to the program. Using these ideas one can prove

Lemma: Checking that a matrix is of full rank can be done in $O(n^2k)$ time.

6.3.2. Checking that the rank is at most $r$

The program above obtained $r$ independent columns such that all other columns are supposedly dependent on them. Let these independent columns be $a_1, \ldots, a_r$. These columns are vectors in $F^n$. One can augment this set of columns by random vectors in $F^n$ so as to have $n$ vectors in all.

Lemma: If $a_1, \ldots, a_r$ are independent, then with probability greater than a positive constant $\left(\frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \ldots = .28, \ldots\right)$, the $n$ vectors obtained by augmenting $a_1, \ldots, a_r$ with random vectors $b_{r+1}, \ldots, b_n$ forms a basis for $F^n$.

Two other facts are used in the construction of this checker.

1. To check that the originally given $a_{r+1}, \ldots, a_m$ are dependent on $a_1, \ldots, a_r$, it is sufficient (probabilistically) to check that each of $k$ random linear combinations of $a_{r+1}, \ldots, a_n$ is dependent.
2. If $x$ is a vector independent of $a_1, \ldots, a_r$, then in its (unique) expression as

$$x = \sum_{i=1}^{r} c_i a_i + \sum_{i=r+1}^{n} c_i b_i,$$

each $c_i$ for $r+1 \leq i \leq n$ has a probability $\geq 1/2$ of being nonzero. Here the probability is over the possible choices of $b_{r+1}, \ldots, b_n$.

Using these results one can prove

Theorem: There is a checker for Matrix Rank running in time $O(n^2k^2)$.
6.4. A Program Checker for Containment

Checking containment \((A \subseteq B)\) is an important function of many checkers. For example, to check that a program for minimum spanning tree or maximum matching in a graph has worked correctly, one must check (among other things) that the output edges are a subset of the set of all edges of the graph. Fortunately, it is possible to construct a probabilistic linear time containment checker, as we show next, in any reasonable model of computation\(^{10}\) that has a random access memory, one of size \(\Omega((\|A\| + \|B\|))\).

The following algorithm for multisets containment was discovered by Carter, Floyd, Gill, Markowsky & Wegman [CFGMW] (see their 'Approximate Membership Tester 3') and independently by these authors. The algorithm is simplest to explain in terms of a set of independent (truly) random hash functions \(\{h_1, \ldots, h_k\}\):

**Containment Checker**

**Input:** 2 multisets of equal size \(A = \{a_1, \ldots, a_n\}\) and \(B = \{b_1, \ldots, b_n\}\), and a positive integer \(k\).

**Output:** *YES* if \(A\) is contained in \(B\) (meaning that each element appears in \(B\) at least as often as it does in \(A\)),

*NO* otherwise;

An output error may occur only if \(A\) is not contained in \(B\), in which case the probability of error is less than or equal to \(1/e^k\).

**Idea of the algorithm for sets:**

If the \(n\) elements of set \(B\) are hashed into \(n\) buckets, then the expected number of empty buckets will be \(1/e\). As a consequence, if \(A\) contains an element \(a\) that is not in \(B\), then the probability that \(a\) is hashed into an empty bucket is \(1/e\).

**Begin**

Comment: \(A\) and \(B\) are multisets

For \(i = 1\) to \(k\), do:

Use hash function \(h_i\) to hash the \(n\) elements of \(B\) into \(n\) buckets.

In each bucket, store only the number of elements that hash into it.

(If \(A\) and \(B\) are known to be sets, then it suffices to store just a single bit instead of a number. The bit indicates whether the number of elements hashed into that bucket is 0 or greater than 0.)

Hash the \(n\) elements of \(A\) in the same way into the same \(n\) buckets.

If a bucket contains more elements of \(A\) than of \(B\), Return *NO*.

**End do**

Return *YES*.

**End**

The reason this algorithm works is that each element \(b\) of \(B\) has a probability \(1/e\) to hash into its own "private" bucket. A private bucket is one that is mapped to by just one element of \(B\) (and any other copies of that element in \(B\)) and no others. Thus if some element occurs in \(A\) more times than in \(B\), then each iteration of the above algorithm has a \(1/e\) chance to discover that fact.

[CFGMW] have pointed out that a universal2 set of hash functions will work in the above algorithm provided the algorithm is modified so that multisets of size \(n\) are hashed into \(2n\) (rather than \(n\)) buckets. A typical universal2 collection of hash functions based on any given prime \(p\) is \(h_{a,b}(x) = ax + b \mod p\)

where \(a\) and \(b\) range over the integers in the interval \(0 < a,b < p\).

---

\(^{10}\) We do not know how to check containment in linear time in the more limited and demanding tape model of computation used above to decide multiset equality.
7. Checker Characterization Theorem

We take as our definition of IP (Interactive Proof-System) the definition appearing in Goldwasser, Micali, and Rackoff [GMR], except that we replace "for all sufficiently large x" in that definition by "for all x". This modification of GMR conforms with the commonly accepted definition of IP as it appears, for example, in Goldwasser and Sipser [GS], and Tompa and Woll [TW].

Define function-restricted IP (CO-IP) = the set of all decision problems for which there is an interactive probabilistic expected-polynomial-time proof system for YES-instances (NO-instances) of \( \pi \) satisfying the conditions that prover (= any honest prover) must compute the function \( \pi \) and prover (= any dishonest prover) must be a function from the set of instances of \( \pi \) to \{YES, NO\}. This restriction implies two things:

1. verifier may only ask questions that are instances of \( \pi \), and
2. prover (and prover) must answer each of verifier's questions with an answer that is independent of prover's previous history of questions and answers.

Because of restriction 1 on verifier and 2 on prover, it is not clear whether IP is contained in function-restricted IP. Because of restriction 2 on prover, it is not clear whether function-restricted IP is contained in IP.

Theorem: An efficient program checker \( C_x \) exists for decision problem \( \pi \) lies in function-restricted IP \( \supseteq \) function-restricted CO-IP.

Let NP-search denote the class of problems \( \pi \) such that \( \pi(x) = NO \) if \( x \) is a NO-instance; YES together with a proof that \( x \) is a YES-instance otherwise.

Corollary: Let \( \pi \) be an NP-search problem. An efficient program checker \( C_x \) exists for \( \pi \) is in function-restricted CO-IP.

The main purpose of the above corollary is to point out that if \( NP \subseteq \) function-restricted CO-IP, as seems likely, then there can be no efficient program checker \( C_x \) (in the above sense) for NP-complete problems!

8. Overview and Conclusions

The thrust of this paper is to show that in many cases, it is possible to check a program's output on a given input, thereby giving quantitative mathematical evidence that the program works correctly on that input. By allowing the possibility of an incorrect answer (just as one would if computations were done by hand), the program designer confronts the possibility of a bug and considers what to do if the answer is wrong. This gives an alternative to proving a program correct that may be achievable and sufficient for many situations.

A proper way to develop this theory would be to require that the program checker itself be proved correct. This paper, however, is about pure checking, meaning no proofs of correctness whatsoever. Instead, we require the checker \( C \) to be different from the program \( P \) that it checks in two ways: First, the input-output specifications for \( C \) are different from those for \( P \) (\( C \) gets \( P \)'s output and it responds CORRECT or BUGGY). Second, we demand that the running time of the checker be \( o(S) \), where \( S \) is the running time of the program being checked. This prevents a programmer from undercutting this approach, which he could otherwise do by simply running his program a second time and calling that a check. Whatever else the programmer does, he must think more about his problem.

9. Acknowledgements

We are grateful to Ronitt Rubinfeld for many long conversations and marvelous ideas, including her extension of our checking ideas to parallel computation. She, Sandy Irani, and Raimund Seidel have designed interesting checkers for various problems in computational geometry such as convex hull. We