Designing Programs to Check Their Work

Manuel E. Blum

TR-88-009

November, 1988

ABSTRACT

Students, engineers, programmers... are all taught to check their work. Computer programs are not. There are several reasons for this:

1. Computer hardware almost never makes errors -- but that fails to recognize that programmers unfortunately do!
2. Programs are hard enough to write without having to also write program checkers for them -- but that is the price of increased confidence!
3. There is no clear notion what constitutes a good checker. Indeed, the same students and engineers who are cautioned to check their work are rarely informed what it is that makes for a good procedure to do so -- but that is just the sort of problem that computer scientists should be able to solve!

In the view of the author, the lack of correctness checks in programs is an oversight. Programs have bugs that could perfectly well be caught by such checks. This paper urges that programs be written to check their work, and outlines a promising and rigorous approach to the study of this fascinating new area.

1International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, California 94704-1105, and the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94704.
Designing Programs to Check Their Work

Manuel Blum

Computer Science Division
University of California at Berkeley
94720

Summary:
A program correctness checker can provide assurance that a computation is correct. 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.

1. Introduction and Brief History

Students, engineers, programmers... all are taught to check their work. Computer programs are not. There are several reasons for this:

1. Computer hardware almost never makes errors—but that fails to recognize that programmers do!
2. Programs are hard enough to write without having to also write program checkers for them—but that is the price of increased confidence!
3. There is no clear notion what constitutes a good checker. Indeed, the same students and engineers who are cautioned to check their work are rarely informed what it is that makes a procedure good for doing so—but that is just the sort of problem that computer scientists should be able to solve!

In the view of this author, the lack of correctness checks in programs is an oversight. Programs have bugs that could perfectly well be caught by correctness checks. This paper urges that programs be written to check their work, and outlines an approach to the study of this fascinating area.

1.1. History

The ideas in this paper arise from a mixture of 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 Budd and Angluin [BA] and

---

1 This work was supported in part by NSF grant CCR85-13926, in part by IBM Research Division, T.J. Watson Research Center, and in part by the International Computer Science Institute, Berkeley, CA. A preliminary version of this paper appeared in Blum and Raghavan [BR].

Author's e-mail address: blum@ernie.berkeley.edu
Weyuker [W] are indirectly relevant to this paper in that they seek to give program testing a rigorous mathematical basis.

1.2. What This Paper Is About

Section 2 formally defines a "program checker." Informally, such a checker is a program for either certifying the correctness of a given program’s output on a given input or else exhibiting a bug in that program. A theoretical requirement, the "little oh" property, is required to hold of every checker to give evidence that the checker is programmed differently from the program it checks.

Section 3 gives an example of a program checker for (any program for) the graph isomorphism problem. This example, which is based on an interactive proof of Goldreich, Micali, and Wigderson [GMW], shows the relevance of cryptographic ideas to this completely different area of computer science. The reader may feel that this kind of program checker will work only for this and relatively few other problems. The next two sections show that it has much broader applications.

Section 4 describes a technique suggested by Beigel [B] for converting a program checker for one computational problem into a program checker for any other polynomial-time-equivalent problem.

Section 5 presents program checkers for a number of group theoretic problems. These checkers all resemble the checker for graph isomorphism.

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. For example, the author considered designing checkers for programs used in the proof of the 4-color problem but rejected this approach on the grounds that such a study would be too specialized. 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 better and more natural place to focus initial attention.

Another important area on which to focus research is the problems in $P$. This is a vast area, we do a little of it.

Section 6 presents fast checkers for a few carefully chosen problems in $P$. The first example is Euclid’s own checker for his gcd and modular inverses algorithm! The second example is a checker for sorting algorithms: the problem with sorting is not just to check that the output is in order, which is easy, but to check that the output and input sets are the same. This check, it is proved here, can be carried out in probabilistic linear time on any practical (and many impractical) models of computation. For example, a checker for sorting (or multi-set equality) can run in linear time on a 1-tape 1-head Turing machine, even though a lower bound for sorting (or multi-set equality) on a Turing machine is $\Omega(n^2)$ steps. Finally, a checker is constructed to check set containment in linear time on a RAM-machine. This is important since containment arises as a necessary ingredient of checkers for many different problems. Recently, this author learned that this containment checker was constructed independently and earlier in another context by Carter, Floyd, Gill, Markowsky and Wegman [CFGMW].

Section 7 characterizes program checkers in terms of a concept that is close to but not the same as the standard [GMR] concept of IP, the class of problems for which there is an interactive protocol.

Section 8 gives an overview and conclusions.

Since the checker method is developed here with emphasis on computational group problems, it seems appropriate to quote John Leech [L1, p. 38], a pioneer of computational group research, in a published appeal for correctness proofs of the kind provided here:
A proof by coset enumeration (whether by hand or by computer) that a group is finite clearly implies proof of many relations in the group, for example those specifying periods of elements of the group. But it does not readily yield explicit proofs of these relations, particularly when the work is done out of sight on a computer and only the results are presented. This is conspicuous in cases where the group is trivial; computer working commonly destroys its own traces, and one is left with no supporting evidence that the group is trivial to supplement the bare statement that ‘the computer says so’. Since almost any sporadic computer malfunction, and many a subtle program error, will precipitate a collapse of the working to a single coset, one’s confidence in such a conclusion may well be less than complete. Coxeter... reported independent coset enumerations, by Sinkov and himself, purporting to show that the group \((5, 5, 5: 3)\) was trivial, yet later work... showed that this conclusion was wrong....

The methods of this paper solve John Leech’s problem.

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 converges (i.e., 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 inputs, for any instance \(I\) of \(\pi\), and for any positive integer \(k\) (the so-called “security parameter”), \(k\) presented in unary:

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

In the above, 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 \(C_\pi^P(I; k) = \text{TIME}\).

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.

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.

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 \(C_\pi^P(I; k)\) may output \text{CORRECT} or \text{BUGGY}. Intuitively, if

\[2\] This probability is computed over the sample space of all finite sequences of coin flips that \(C\) could have tosses.
\( C_P^c(I;k) = \textit{CORRECT} \), then we may rely with a specific high (but not absolute!) degree of confidence on the output of \( P \) on this particular \( I \); if \( C_P^c(I;k) = \textit{BUGGY} \), then \( P \) definitely (rather than probably) has a bug. Note that in this latter case, \( P(I) \) may nevertheless be correct.

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 major serious problem.\(^3\) One solution is to prove the checker \textit{correct}. Sometimes, this is easier than proving the original program correct, as in the case of the Euclidean \textit{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 \textit{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 \textit{differently} from the program it checks.\(^4\) The little oh property forces the programmer \textit{away} from simply running his program a second time. Whatever else he does, he must think more about his problem:

4. The running time of the checker must be little oh of the running time of the program being checked.

Checkers satisfying condition 4 will be specifically referred to as having the little oh property. The specific examples of program checkers sprinkled throughout this paper all have the little oh property with respect to all known programs for the given problem. The general results (Beigel’s theorem, checker characterization theorem) however, are for program checkers that satisfy conditions 1 and 2 only.

3. Example: Graph Isomorphism

The graph isomorphism decision problem is defined as follows:

\begin{align*}
\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.}
\end{align*}

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:

---

\(^3\) The 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 usually \textit{test} it. Then we 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 \( P \).
\[ C_{G'}^{\mathcal{G}}(G, H; k) = \]

**Input:** \( P \) = a program (supposedly for Graph Isomorphism) that always halts, to be used as an oracle by \( C_{G'}^{\mathcal{G}} \);

\( G, H \) = two graphs, each having \( n \) nodes;

\( k \) = a positive integer.

**Output:** The output must be **CORRECT** or **BUGGY** (sometimes either one is correct). It is only required that the implications hold:

**CORRECT** \( \implies P(G, H) \) is probably correct. This means that the probability that output = **CORRECT** in case \( P(G, H) \neq GI(G, H) \) is at most \( 1/2^k \) (see previous footnote).

**BUGGY** \( \implies P \) definitely has a bug.

*Begin*

Compute \( P(G, H) \).

If \( P(G, H) = \text{YES} \), then

Use \( P \) (as if it were bug-free) to search for an isomorphism from \( G \) to \( H \).

(This is done by attaching an appropriately selected "gadget," say a clique on \( n + 1 \) nodes, to node 1 of \( G \). Denote the resulting graph by \( G' \). Then search for a node of \( H \) such that when the same gadget is attached to that node, the resulting \( H' \) is, according to \( P \), isomorphic to \( G' \), that is, \( P(G', H') = \text{YES} \). If no such \( H' \) is found then return **BUGGY**. Continue as above to build a correspondence between the nodes of \( G' \) and the nodes of \( H' \), until the resulting "isomorphism" is complete. Isomorphism is in quotes here because \( P \) is unreliable.)

Check if the resulting correspondence is an isomorphism.

If not, return **BUGGY**; if yes, return **CORRECT**.

If \( P(G, H) = \text{NO} \), then

Do \( k \) times:

Toss a fair coin.

If coin = heads then

generate a random \(^5\) permutation \( G_i \) of \( G \).

Compute \( P(G, G_i) \).

If \( P(G, G_i) = \text{NO} \), then return **BUGGY**.

If coin = tails then

generate a random permutation \( H_j \) of \( H \).

Compute \( P(G, H_j) \).

If \( P(G, H_j) = \text{YES} \), then return **BUGGY**.

*End do*

Return **CORRECT**.
*End*

The above program checker correctly tests any computer program 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:

---

\(^5\) By a "random" permutation we mean that every permutation of \( G \), i.e., every relabeling of the \( n \) nodes of \( G \) with the integers \( 1, \cdots, n \), is equally likely.
\( C^P_{G_1} \) is efficient.

Let \( P \) be any decision program (a program that halts on all inputs and always outputs YES or NO). Let \( G \) and \( H \) be any two graphs. Let \( k \) be a positive integer.

If \( P \) is a correct program for \( G_1 \), i.e., one without bugs, then \( C^P_{G_1}(G,H;k) \) will definitely output \( \text{CORRECT} \).

If \( P(G,H) \) is incorrect on this input, i.e., \( P(G,H) \neq G_1(G,H) \), then \( \text{prob} \{ C^P_{G_1}(G,H;k) = \text{CORRECT} \} \) is at most \( 1/2^k \).

**Proof:** Clearly, \( C^P_{G_1} \) runs in expected polynomial time.

If \( P \) has no bugs and \( G \) is isomorphic to \( H \), then \( C^P_{G_1}(G,H;k) \) constructs an isomorphism from \( G \) to \( H \) and (correctly) outputs \( \text{CORRECT} \).

If \( P \) has no bugs and \( G \) is not isomorphic to \( H \), then \( C^P_{G_1}(G,H;k) \) tosses coins. It discovers that \( P(G,G_i) = \text{YES} \) for all \( G_i \), and \( P(G,H_j) = \text{NO} \) for all \( H_j \), and so (correctly) outputs \( \text{CORRECT} \).

If \( P(G,H) \) has a bug, there are two cases:

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

Finally, the most interesting case:

2. If \( P(G,H) = \text{NO} \) but \( G \) is isomorphic to \( H \), then what? The only way that \( C \) will return \( \text{CORRECT} \) is if \( P(G,G_i) \) or \( H_j \) = \( \text{YES} \) whenever the coin comes up heads, \( \text{NO} \) when it comes up tails. But \( G \) is isomorphic to \( H \). Since the permutations of \( G \) and \( H \) are random, \( G_i \) has the same probability distribution as \( H_j \). Therefore, \( P \) correctly distinguishes \( G_i \) from \( H_j \) only by chance, i.e., for just 1 of the \( 2^k \) possible sequences of \( T \)'s coin tosses.

Qed

**Remark:** If \( C^P_{G_1}(I;k) = \text{BUGGY} \), then the above program checker must actually have obtained a proof that \( P \) has a bug, e.g., for \( P(G,H) = \text{NO} \) it might have found \( H_j \) such that \( P(G,H_j) = \text{YES} \). This proof points to instances of \( \pi \), in this case \( (G,H) \) and \( (G,H_j) \), that exercise the part of \( P \) that has a bug. In general, one can always modify \( C \), using the results of the next section, so that \( C^P_{\pi}(I;k) \) outputs such a pointer whenever it discovers that \( P \) has a bug.

### 4. Beigel's Trick

Richard Beigel [B] has pointed out to this author 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 \( \pi_2 \).

**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 \( i \in \{1,2\} \).

\( C_{\pi_2}^P(I_2;k) \) 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 \( P_1(\pi_1) = \text{def} P_2(f_{12}(\pi_1)) \) for all instances \( \pi_1 \) of \( \tau_1 \), 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 \( C_{\pi_1}^P(I_1, k) \).

If conditions one or two fail, then \( C_{\pi_2}^P(I_2; k) := \text{BUGGY} \). Otherwise, \( C_{\pi_2}^P(I_2; k) := \text{CORRECT} \).

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 \( C_{\pi_1}^P(I_1; k) = \text{CORRECT} \). So \( C_{\pi_2}^P(I_2; k) = \text{CORRECT} \).

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 \( C_{\pi_2}^P(I_2; k) = \text{BUGGY} \), or else condition 1 holds, whence

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

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

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

\[
= \pi_1(I_2) \quad \text{by} \pi_1(f_{21}(I_2)) = \pi_1(I_1),
\]

in which case \( C_{\pi_1}^P(I_1; k) \), and therefore also \( C_{\pi_2}^P(I_2; k) \) will correctly return \text{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. Subsection 5.2 gives a general approach to checker construction that works particularly well for group theoretic problems, with a hard and beautiful example due to Sampath Kannan.

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 example, \( S \) and \( G \) could be:

1. \( S = \) Set of all labelled \( n \)-node graphs.
   \( G = \) Group generated by all permutations of the node labels.

2. \( S = \) Set of all cubes whose faces are each colored with one of three colors, \( R,W,B \).
   \( G = \) Group generated by the three orthogonal 90 degree rotations.

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

Let \( ESP(S,G) \) denote the

\textit{Equivalence Search Problem}

\textbf{Input:} \( a,b \) in \( S \)

\textbf{Output:} \( g \) such that \( g(a) = b \) if \( a =_G b \);
Theorem: 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 \(C_{\text{ESP}(S,G)}^P\) for the problem ESP(S,G).

Proof: Let \(C_{\text{ESP}(S,G)}^P(a,b;k) = \)

\[
\text{Begin}
\]

\[
\text{If } P(a,b) = \text{NO, then generate } k \text{ random } g \text{'s in } G, \text{ and for each such } g:
\]

\[
\text{Check that } P(a,g(a)) = \text{YES and } P(a,g(b)) = \text{NO.}
\]

\[
\text{If every check succeeds, then return CORRECT; else return BUGGY.}
\]

\[
\text{If } P(a,b) = g, \text{ then return CORRECT or BUGGY depending on whether or not } g(a) = b.
\]

\[
\text{End}
\]

It is easy to see that \(C_{\text{ESP}(S,G)}^P\) is an efficient program checker.

Qed.

The theorem gives conditions under which any program for the equivalence search problem can be tested. Standard examples of such an equivalence search problem include:

1. Graph Isomorphism
2. Quadratic Residuacity:
   \[
   \text{Input: Positive integer } N; \ a \text{ in } Z_N^*.
   \]
   \[
   \text{Output: } x \text{ such that } x^2 \mod N = a \text{ if such } x \text{ exists;}
   \]
   \[
   \text{NO otherwise.}
   \]

Here, \(S = Z_N^*\); \(G = \{g^2 \mod N \mid g \in Z_N^*\}\) under the operation of multiplication \(\mod N\). \(G\) acts on \(S\) by the map \(g: a \rightarrow a \cdot g^2 \mod N\). Note that \(a\) is a quadratic residue \(\mod N\) if and only if \(a \equiv 1 \mod N\).

---

\(^6\) Related to the Equivalence Search Problem is the Equivalence Decision Problem defined by:

\text{Equivalence Decision Problem (EDP)}

\text{Instance: } a, b \text{ in } S

\text{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, \(Z_N^*\) denotes the group of positive integers less than \(N\) that are relatively prime to \(N\) under multiplication \(\mod N\). For \(p\) a prime, let \(S = Z_p^*\) and \(G = Z_{p-1}^*\), 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 \(Z_{p-1}^*\). 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 \textit{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 \(\text{order}(b) | \text{order}(a)\). This is because \(x^\text{order}(a) = 1 \mod p\) has exactly \(\text{order}(a)\) solutions, namely \(a, a^2, \ldots, a^{\text{order}(a)} = 1\). Finally, \(\text{order}(a)\) and \(\text{order}(b)\) can be determined from the factorization of \(p-1\).
3. **Discrete Log Generalization:**
   **Input:**  Positive integer $N$; $a, b$ in $\mathbb{Z}_N^*$.
   **Output:**  $x$ such that $a^x = b \mod N$ if such $x$ exists;
   NO otherwise.

   Here, $S = \mathbb{Z}_N^*$; $G = \mathbb{Z}_{\phi(N)}$ under the operation of addition $\mod \phi(N)$, where $x$ in $G$ maps $a$ in $S$ to $a^x \mod N$.

   Besides Graph Isomorphism, Quadratic Residuacity, and Discrete Log (the usual ones that appear in cryptography), are there any other equivalence search problems not known to be in $P$ for which there are correct checkers? Yes! There are many natural such problems! Nonstandard examples of the equivalence search problem include:

4. **Games (see Berlekamp and Conway (BC)):**
   Rubic’s cube: Suppose Rubic’s cube is (physically) taken apart, then put back together again in a different order. Are the two settings equivalent in the sense that one can be transformed into the other by an allowable sequence of rotations?

   15 puzzle: Given two settings of the puzzle, can one setting be transformed into the other by an allowable sequence of moves?

5. **Examples from computational group theory:**

   We quote from Jeffrey S. Leon [L2, pp. 321-2]:

   Given a combinatorial object $X$, how can we effectively compute the automorphism group of $X$? ... Two related problems, often considered together with the automorphism group problem, are the following:

   1. Determine if two combinatorial objects $X_1$ and $X_2$ are isomorphic;
   2. Given a combinatorial object $X$, find a canonical representative $\text{canon}(X)$ for its isomorphism class (that is, $\text{canon}(X)$ is isomorphic to $X$, and $X_1$ isomorphic to $X_2$ implies $\text{canon}(X_1) = \text{canon}(X_2)$).

   Note that programs for the second of these problems can also be tested by the methods given here. Define the

   **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 $g(a) = c$.

   **Theorem:** 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.

   **Proof:** Observe that the canonical element problem does not specify which element of any given equivalence class shall be the canonical one. 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. With this definition, we have the following program checker:
\( C_{\text{EP}} \)

Input: \( a, c \in S, \ g \in G, \) and a positive integer “security parameter” \( k. \)

Output: \( \text{CORRECT} \iff g(a) = c \) and at least half of the elements of \( [a] \) are mapped by the CEP program to \( c. \) Here, \( [a] = \) the equivalence class containing \( a. \)

As usual, the output \( \text{CORRECT} \) may be in error; the probability of such error is at most \( 1/2^k. \)

\( \text{BUGGY} \iff [g(a) \neq c] \) or \( [g(a) = c \text{ and there is an element } b \text{ of } S \text{ equivalent to } a \text{ such that } g(b) \neq c]. \)

Begin

If \( g(a) \neq c \) then return \( \text{BUGGY}. \)

Do \( k + 1 \) times:

Select a random element \( g' \) of \( G \) and compute \( b = g'(a). \)

If \( g(b) \neq c, \) then return \( \text{BUGGY}. \)

End-do

Return \( \text{CORRECT}. \)

End

It is clear that this program checker works as specified. It runs in expected polynomial time and therefore has the little “oh” property with respect to any of the currently known super polynomial time programs for the general canonical element problem.

Qed

6. Examples from knot theory, block designs, codes, matrices over \( GF(q) \), Latin squares, etc. \[1.2. \text{p.} 321].

7. Examples from applications of Burnside and Polya theorems [PR]:

To spell one out, consider the Boolean function equivalence problem. Here, \( f \) and \( g \) are Boolean functions described by truth tables. Say that \( f \) is equivalent to \( g \) if and only if \( g \) can be obtained from \( f \) by permuting and/or complementing the variables. (The Polya problem is to count the number of “distinctly different” Boolean functions on \( n \) variables, a nontrivial problem.) For example, \( f = ab + c' \) is equivalent to \( g = cb' + a \) under the transformation \( a \rightarrow c, \ b \rightarrow b, \ c \rightarrow a, \) then \( a \rightarrow a', \ b \rightarrow b', \ c \rightarrow c. \)

In our setting, \( S = \) set of Boolean functions on \( n \) variables; \( G = \) group of permutations that permute and/or complement the variables. Deciding if two Boolean functions of \( n \) variables are equivalent by trying all permutations and/or complementations takes \( O(2^{\log_2 N}) \) steps for \( N = 2^n. \) which is superpolynomial.

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.

Sampath Kannan, currently writing his Ph.D. thesis [K1] with me on group theoretic algorithms, has devised polynomial time program checkers for several group theoretic problems in this way. A beautiful example is his checker for
The Group Intersection Problem:
Input: Two permutation groups, $G$ and $H$, specified by generators. The generators are presented as permutations of $[1, \cdots, n]$.
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 Kannan's 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 $[1, \cdots, n]$ 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 $\{x | x = ab \text{ where } a \in G \text{ and } b \in H\}$. Here is the continuation of the protocol.

3. The verifier sends to the prover an element, $x$, of $GH$, which he obtains by selecting random elements of $G$ and $H$ and multiplying them together.

4. The prover sends back a 'factorization' of $x$ as $a'b'$ where $a' \in G$ and $b' \in H$.

**Lemma:** $a^{-1}a'$ is a random element of $G \cap H$.

**Proof:** First we show that $a^{-1}a' \in G \cap H$:

$$ab = x = a'b' \implies a^{-1}a = b'b^{-1}$$

On the left hand side of the last equation we have an element of $G$ equalling an element of $H$ on the right. 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 $x = ab$ be an element of $GH$. Then $x = (ax^{-1}b)$. If the prover sends back this factorization of $x$ 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 $|G \cap H|/2$. 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 using Beigel’s trick 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 \( GH \).
\( a, b \) such that \( a \in G \) and \( b \in H \) and \( ab = \pi \) otherwise.

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

Factorizing 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 \( ab \) such that \( a \in G \) and \( b \in H \)?

We are left with the problem of showing that FDP and FSP are equivalent. We will show how to use FDP to solve FSP. The reduction in the other direction (which is necessary to use Beigel’s trick!) is trivial.

Using FDP to Solve FSP

Assume that the generators for \( G \) and \( H \) are in the special form used in Furst, Hopcroft, and Lux [FHL]. Call this the ‘matrix form.’ This is not a serious assumption because any set of generators can be converted to matrix form in polynomial time.

Here is a brief description of the matrix form \( M_G \) for the generators of a 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 \( ij \)th entry is any such permutation in \( G \). It is convenient (and customary) to make the diagonal entries be the identity permutation.

Some properties of the matrix form representation are given here without proof. Every element of \( G \) can be expressed in a unique way as a product, \( \pi_n \pi_{n-1} \ldots \pi_1 \) 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 \( \pi_1 \) is the entry \( \sigma \pi_1^{-1} \) 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, \( \sigma_n \sigma_{n-1} \ldots \sigma_1 \) 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 $a \sigma_i^{-1}$ is in $G_H$. We can use the oracle for FDP to find out which entry in the first row of $M_H$ has the above property. If this entry is $\sigma_i$, we consider $\pi \sigma_i^{-1}$ and factor it in $G_H$. A factorization in $G_H$ 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 $G$. This completes the reduction and shows that the 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 $C^*_f(I,k)$, it will be denoted by $C^*_f(I,O;k)$. 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 $C^*_f(I,O)$ instead of $C^*_f(I,O;k)$.

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 three problems in $P$: Euclidean-GCD (because it is one of the oldest nontrivial algorithms on the books), Sorting (because it is one of the most frequently run algorithms), and Containment (because many checkers seem to require containment checks).

6.1. A Program Checker for Euclidean GCD

**Euclidean-GCD**

Input: Two positive integers $a,b$.

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

Observe that Euclidean-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 Euclidean-GCD both output $d = \gcd(a,b)$, but Euclidean-GCD also outputs two integers $u,v$ such that $a \cdot u + b \cdot v = d$. We know how to check programs for Euclidean-GCD but not GCD.

For our model of computation, we choose a standard RAM and count only arithmetic operations $+, -, \cdot, /$ as steps.

$C^*_{\text{Euclidean-GCD}}$

Input: positive integers $a,b$; positive integer $d$ and integers $u,v$.

Output: BUGGY if $d \nmid a$ or $d \nmid b$ or $a \cdot u + b \cdot v \neq d$,

CORRECT if the given program $P$ computes Euclidean-GCD correctly.

Begin

1. If $d \nmid a$ or $d \nmid b$, return BUGGY. else
2. If $a \cdot u + b \cdot v \neq d$, return BUGGY. else
3. Return CORRECT.

End

Here, line 1 checks that $d$ is a divisor of $a$ and $b$; line 2 checks that $d$ is the greatest divisor of $a$ and $b$. This checker takes just 3 steps: 2 division steps + 2 multiplication steps + 1 addition step. By comparison, the running time of all known GCD programs requires at least $\log n$ division steps on certain inputs.
6.2. A Program Checker for Sorting

**Sorting**

Input: An array of integers $X = [x_1, \ldots, x_n]$, representing a multiset.
Output: An array $Y$ consisting of the elements of $X$ listed in non-decreasing order.

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

$C_{\text{Sorting}}$

Input: Two arrays of integers $X = [x_1, \ldots, x_n]$ and $Y = [y_1, \ldots, y_m]$.
Output: *BUGGY* if $Y$ is not in order or if $X \neq Y$ as multisets.
CORRECT if the given program $P$ correctly sorts.

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.

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 $n$. Before exhibiting a way to do multiset equality, let us see a precise statement of what is wanted.

**Multiset Equality Test**

Input: Two arrays of integers $X, Y$; and a positive integer $k$.
Output: YES if $X$ and $Y$ represent the same multiset, NO otherwise. with probability of error $\leq 1/2^k$. 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 $1/2^k$.

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.\(^7\)

---

\(^7\) 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 $\sum_{x \in X} h(x_i)$ and compute $\sum_{y \in Y} h(y_i)$. If the two sums are equal, then setting $h(x_1) = 1$ will distinguish $X$ from $Y$; if different, then setting $h(x_1) = 0$ will distinguish the two.
Choosing an easy-to-compute random-enough hash function is difficult. The Wegman and Carter hash function in particular requires an associative 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

\[
m = 1 + \max \{ \text{number of times that element } e \text{ can appear in one of the multisets } e \in X \cup Y \}.
\]

\( m = 2 \) if the multiset is a set; \( m \leq n + 1 \) in general. Let \( a \) = 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 \( \sum m^{x_i} = \sum m^{y_i} \). Indeed, if \( X = Y \), then \( \sum (m^{x_i} \mod p) = \sum (m^{y_i} \mod p) \) for all primes \( p \). If \( X \neq Y \), then \( \sum m^{x_i} \neq \sum m^{y_i} \) and so, as pointed out by Rabin [R] in his man-in-the-moon problem, \( \sum m^{x_i} \neq \sum m^{y_i} \mod p \) for at least half of all primes in the interval \( [1, 3 \cdot a \cdot \log m] \), and therefore we observe that \( \sum (m^{x_i} \mod p) \neq \sum (m^{y_i} \mod p) \) for at least half of all primes in this interval. This proves that \( h \) is a random-enough hash function.

**Method 2:** This idea was first suggested by Lipton [L3] and more recently by Ravi Kannan [K2]. Let \( f(z) = (z - x_1) \cdots (z - x_n) \) and \( g(z) = (z - y_1) \cdots (z - y_n) \). Then \( x = y \) 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 \( f(z) = g(z) \) 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 has at most \( n \) integers, each in the range \([0, a]\). In method 1, primes are \( O(a \log n) \); in method 2, they are \( O(n \log a) \). In method 1, computations take \( O((a \log n)^2 \cdot n) \) steps; in method 2, they take \( O((n \log a)^2 \cdot n) \) steps. From this, it follows that for \( a \geq n \), method 2 is better than 1, while for \( a < n \), 1 is better.

### 6.3. 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 that has a random access memory, one of size \( \Omega(|A| + |B|) \).

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

---

8 A common way to construct hash functions is to select a prime \( p \) and integers \( a, b \) at random, then to set \( h(x) = a \cdot x + b \mod p \). Will this work? Not if 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 \( \sum h(x_i) = \sum h(y_i) \mod p \).

9 This author does 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.
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

End

Return YES.

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 \( 0 < a, b < p \).

7. Checker Characterization Theorem

This section characterizes all program checkers that satisfy conditions 1 and 2. (Program checkers satisfying conditions 1, 2, and 4 have not been characterized: this is an open problem.)

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 \( z \)” in that definition by “for all \( z \)”. 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 (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 \( C_\pi \) exists for decision problem \( \pi \iff \pi \text{ lies in function-restricted IP } \cap \text{function-restricted CO-IP} \).

**Proof:**

\[ \implies \] We must show two things:

1. a prover can convince a verifier of the truth.
2. a (dishonest) prover cannot convince a verifier of a lie.

   a) The verifier wants a proof that \( \pi(I) = \text{NO or YES} \), whichever the case may be, for some instance \( I \) of \( \pi \). The verifier simulates \( C_\pi^F(I:k) \), feeding each request that \( C_\pi \) puts to \( P \) to the prover. If the prover always answers truthfully, then \( C_\pi^F(I) \neq \text{BUGGY} \), therefore = CORRECT. This CORRECT is evidence that with high probability, \( P(I) = \pi(I) \). \( \pi(I) \) is determined by asking the prover for \( P(I) \).

   b) What are the chances for an adversary, a restricted prover to convince a verifier of a falsehood? If prover can do this on some instance \( I \), then we can construct a program \( \tilde{P} \) such that \( \tilde{P}(I) \neq \pi(I) \) and yet \( \tilde{P} \) is accepted by the program checker with probability greater than \( 1/2^k \). This is a contradiction.

\[ \iff \] Define an efficient program checker as follows:

\[ C_\pi^P(z) = \text{"Determine } P(z) \text{, say } P(z) = \text{NO (YES). Use the verifier's scheme in function-restricted co-IP (or function-restricted IP) to check that indeed } \pi(z) = \text{NO (or YES). If proof is convincing, return CORRECT, else Buggy."} \]

Qed

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

Let \( NP\text{-search-solving-decision} \) denote the class of decision problems in NP whose corresponding search problem is Cook-reducible to it. This includes most but not all examples in Garey and Johnson [GJ]. Positive examples include the decision problems SAT and TSP (Traveling Salesman Problem). A candidate negative example is the equivalence decision problem defined in the next footnote (section 6 below).

**Corollary:** Let \( \pi \) be an \( NP\text{-Search-Solving-Decision} \) or \( NP\text{-search} \) problem. An efficient program checker \( C_\pi \) exists for \( \pi \iff \pi \text{ is in function-restricted co-IP} \).

**Proof:** \( \implies \) This follows from the previous theorem.

\[ \iff \] Let \( C_\pi^P(z) = \text{"Determine if } P(z) = \text{NO or YES. If } P(z) = \text{NO, proceed as in the above proof to check that } \pi(z) = \text{NO. If } P(z) = \text{YES, use the fact that } \pi \text{ is in search-solving NP to interrogate the oracle } P \text{ in order to get a short proof that } \pi(z) = \text{YES. If no short proof is reached, then return Buggy. Else return CORRECT."} \]

Qed
The main purpose of the above corollary is to point out that if \( NP \notin \text{function-restricted co-IP} \), as seems likely, then there can be no efficient program checker \( C_w \) (in the above sense) for \( \text{NP-complete problems} \! \)

8. Overview and Conclusions

8.1. An Alternative to Proving Correctness and/or Checking

The thrust of this paper is to show that in many cases, it is possible to check a program's output for 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 \( \text{CORRECT or BUGGY} \)). 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.

8.2. Suggestions for the Program Check Designer

This raises the issue that checkers can be difficult to construct. We offer two suggestions for the program check designer:

1. Just as proofs of correctness are often abandoned for proofs of partial correctness, so too it makes sense to construct partial checkers when complete checkers are hard to write. For example, a partial check of a program to factor a number \( N \) might only check that the output \((p_1, e_1), \ldots, (p_k, e_k)\) has the property that \( p_1^{e_1} \cdots p_k^{e_k} = N \), i.e., that the product is correct. This check would be partial because it does not check that \( p_1, \ldots, p_k \) are primes.

The philosophy here is to check for what you can; warn against the rest.

2. Sometimes, when it is not evident how to construct a checker, there is another possibility:

   - Demand that the program compute certain additional variables, then
   - Construct a partial checker that uses those additional variables to check the correctness of the (originally desired) output.

The checker is partial because it does not necessarily have to check the correctness of the computed additional variables, only the correctness of the originally desired output.

For example, to check that a \( \text{GCD} \) program

Input: positive integers \( a, b \)
Output: \( d = \text{gcd}(a, b) \)

outputs the correct value, it suffices to require it to compute additional integers \( u, v \) such that \( a \cdot u + b \cdot v = d \) and then to check that \( d \mid a \) and \( d \mid b \) and \( a \cdot u + b \cdot v = d \). This example is worked in detail in section 8.

For another example, to check that a \( \text{MOD} \) program
Input: integers $a, b$ with $b > 0$.
Output: $c = a \mod b =_{df} \text{remainder } (a/b)$.

outputs the correct value, it suffices to require it to compute the additional variable $q = [a/b]$ and then to check that $a = bq + c$ and that $0 \leq c < b$.

For a final example, to check that a Max Flow program
Input: A network with one source, one sink, and positive integer capacities assigned to edges.
Output: A nonnegative integer $F$ = the maximum flow that can be pushed from source to sink.

outputs the correct value, it suffices to require the program to output the edge flow $f$, $f$ = an assignment of integers to all edges, then to

2.1. check that $f$ is a flow, that is to say:

2.1.1. the sum of the flows into a node $= \text{ the sum of flows out of that node},$ for all nodes $\notin \{\text{source, sink}\}$, and

2.1.2. the flow in each edge is nonnegative and $\leq$ the capacity of that edge, and

2.2. check that a depth-first search that is started at the source and is turned back at every forward edge that is filled to capacity and at every backward edge that is empty will terminate at the source without ever reaching the sink, and

2.3. check that $F = \text{ the sum of flows into the sink}$.

8.3. The Problems Raised by Software Maintenance

Software maintenance is a problem for correctness proofs and software tests. Both must be questioned when software is modified. Both must in that case be redone.

What is the value of a program checker after software modification? If the modification is for a bug in $P$, the program checker is as valid after the modification as before. Only if the modification is for a bug in the checker $C$ is there a possible negative effect: $C$ could incorrectly announce that a correct $P$ is buggy and/or it could fail to catch a bug in a buggy $P$. Still, we would expect a large program with checkers distributed throughout it to be easy to maintain. The reason for the optimism is that $C$ works completely independently of the program $P$ that it checks, so modifications in $P$ will still be caught by $C$ if those modifications are buggy. If the modifications are made elsewhere than in $P$ or $C$, this should not affect $C$'s “ overseer” property.

8.4. The Numerical Analyst, the Compiler Writer, and the Engineer

Appropriate warnings is something that numerical analysts and compiler writers have inserted into their machines for years: Warning: division by 0! Warning: wrong data type! What is new and different in this paper is the view that errors should be suspected and appropriate responses prepared for all computations. When all possibilities for error have been foreseen, the programmer is better able to decide what the appropriate response should be to detection of an error... be that response a simple error message or a sophisticated computation.

Our approach to software checking is also akin to something that engineers have traditionally been taught to do, i.e., to check their computations, to ask: Does this answer make sense? Is it reasonable that the answer should lie in this range? If not, there may be an error in computation or in derivation. The latter is the equivalent of a software bug.
9. Acknowledgements

This paper originated in Shmuel Winograd's response to the question: "In your view, what is the most important research problem facing IBM?" His answer: "Program testing."\(^\text{10}\) Ashok Chandra made numerous insightful comments. For example, he observed that program testing is closer to Physics than Mathematics: "On seeing that \( A \implies B \), a mathematician will show \( B \) by proving \( A \), while a physicist will show \( A \) by proving \( B \)." I am grateful to Winograd and Chandra for providing me the opportunity to participate in the wonderful research environment at IBM T.J. Watson Research Center. I am grateful to Prabhakar Raghavan for valuable early discussions on this work [BR], and to Richard Beigel for the nice ideas of section 4. Sampath Kannan has done splendid work in this area, exhibiting powerful checkers for many group theoretic problems. Ronitt Rubinfeld, Sandy Irani, and Raimund Seidel did the same with program checkers for Computational Geometry. I am especially grateful to Sampath and Ronitt for marvelous ideas, long conversations, and enthusiastic support.

References


\(^{10}\) Little did the author know, however, that Winograd has more than one answer for this question.
Biographical Sketch:
Manuel Blum is a professor of computer science at the University of California at Berkeley, and a member of the International Computer Science Institute, Berkeley. He is a fellow of the IEEE and the AAAS.

CR Categories and Subject Descriptors:
D.2.4 (Software: Program Verification);
D.2.5 (Software: Testing and Debugging);
F.1.2 (Theory of Computation: Probabilistic Computation);
F.2 (Theory of Computation: Analysis of Algorithms and Problem Complexity);
G.4 (Mathematics of Computation: Certification and Testing, Verification)

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Correctness checkers