SIEVE ALGORITHMS FOR PERFECT POWER TESTING

by

Eric Bach
Jonathan Sorenson

Computer Sciences Technical Report #852

June 1989

Sieve Algorithms for Perfect Power Testing

Eric Bach and Jonathan Sorenson
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
USA

June 19, 1989

Abstract.

A positive integer n is a perfect power if there exist integers x and k, both at least 2, such that $n = x^k$. The usual algorithm to recognize perfect powers computes approximate kth roots for $k \leq \log n$, and runs in time $O(\log^3 n)$.

We present an algorithm that avoids kth root computations by seeing if the input n is a perfect kth power modulo small primes. If n is chosen uniformly from a large enough interval, the average running time is $O(\log^2 n)$. We also give an algorithm that incorporates trial division and has an average running time of $O(\log^2 n / \log^2 \log n)$, and a median running time of $O(\log n)$.

The first two average time bounds assume that a table of small primes is precomputed. We give a heuristic argument and computational evidence that the largest prime in the table is $O(\log^{1+\epsilon} n)$; assuming the Extended Riemann Hypothesis primes up to $O(\log^{2+\epsilon} n)$ suffice. The table can be computed in time proportional to the largest prime it contains.

We also present computational results indicating that our new algorithms perform very well in practice.

Sponsored by NSF [Grants DCR-8552596 and DCR-8504485]

University of Wisconsin Computer Sciences Technical Report #852

1. Introduction.

This paper presents fast and practical algorithms for deciding if a positive integer n is a perfect power, that is, can be expressed as x^k for integers $x, k > 1$. By trying all possible powers, this problem is solvable in $O(\log^3 n)$ steps. (This slightly improves results known to us and seems new.) Unfortunately, the average running time for this method is not much better than the worst-case running time. We give algorithms in this paper that perform much better on typical inputs. One of our methods has an average running time of $O(\log^2 n)$, and another runs in $O(\log^2 n / \log^2 \log n)$ average time, with a median running time of $O(\log n)$. Our average-case results assume that certain tables are precomputed; as a practical matter these tables are small, but we need to assume the ERH to bound the values of their entries.

In number theory, most analyses of algorithms address worst case complexity, although there are some studies of average behavior [8, 11, 13, 19, 27]. However, we are unaware of any average-case results for this problem.

Before describing our methods, we indicate some applications for them. The fastest known methods for integer factorization [10, 23] find non-obvious solutions to the congruence $x^2 \equiv y^2$ modulo n; if x and y are solutions with x different from $\pm y$, then $\gcd(x-y, n)$ splits n. However, if n is odd, such x and y will only exist if n is not a prime power, and this condition should be checked before attempting to factor n. It is simplest to check that n is not a perfect power, for if it is, then we have a factorization.

Similar comments apply to many other factoring algorithms [2, 3, 13, 20]; it is common when analyzing them to make the assumption that the input is not a perfect power.

We also mention an application where the average case behavior of a perfect power algorithm is significant; in fact, it is the source of the present problem. Earlier, one of us published an efficient algorithm for generating random integers in factored form [1]; we will not discuss this algorithm in detail except to say that it repeatedly draws integers (according to some distribution) and rejects them when they are not prime powers, saving the prime powers for further processing. Since the perfect powers are rare, any perfect power test that performs well on the average will be useful in this context.

Our algorithm is based on the following idea: if a number n is not a pth power mod q for some small prime q, then it cannot be a pth power. (This is a simple application of the local-global principle.) The time to check this condition is roughly proportional to the length of n, much less than the time needed to compute a pth root of n to high precision. Of course, this advantage is offset somewhat by the fact that a pth power modulo q need not be a pth power. Hence, tests using more than one q are necessary; our algorithm does enough of these tests to ensure that a pth root computation will be rare.

We also present an algorithm that combines perfect power testing and trial division; this algorithm performs even better on the average than the method outlined above. This latter algorithm is useful as a preprocessing step for factorization, since factoring programs usually start by performing trial division. Indeed, in practical factoring one often learns that a number is not a perfect power as a by-product of trial division. This perfect power algorithm is the most efficient one known to us.

Sieving ideas for testing pth powers have been suggested by Cobham [7] and V. Miller [21]. Cobham, assuming the ERH, showed that a sieve method for testing if a number

is a perfect square is optimal to within a constant factor in its use of work space. To prove this, he showed that if a number is not a perfect square, then it must fail to be a square mod q for some small q. Miller generalized this last result to pth powers of algebraic numbers; for certain fields, he found that a sieve algorithm outperforms methods based on root approximation in the worst case.

In contrast, we do not attempt to make the sieve part of the algorithm completely foolproof. Instead, we use pth power tests modulo q to make the relatively expensive root computations rare on the average. Hence our algorithms are always correct, and our probabilistic results only apply to the running time.

Throughout this paper we discuss algorithms to solve the decision problem for the set of perfect powers. Any of them could be easily modified to output roots and exponents when the input is a perfect power, or compute the value of Golomb's arithmetic function $\gamma(n)$ (which counts the positive integral solutions to $n = a^b$) [12], but we leave such modifications to the reader.

2. Notation and Definitions.

Let k be a positive integer. We call a positive integer n a *perfect kth power* if $n = x^k$ for some integer $x > 1$. In this case, we refer to k as the *exponent*, and x as the *root*. If n is a perfect kth power for some integer $k > 1$, then we say n is a *perfect power*.

We will use $\log n$ to denote the logarithm of n to the base 2, and $\ln n$ to denote the natural logarithm. Note that the length of a positive integer n, written in binary, is $\lfloor \log n \rfloor + 1$.

In our analysis, we will assume that arithmetic on large numbers is done by classical methods. Hence the complexity of basic arithmetic we take to be as follows (see [18]):

1. To compute $a \times b$, $\lfloor a/b \rfloor$, or $a \bmod b$ takes time $O(\log a \log b)$.
2. To compute $a \pm b$ takes time $O(\log a + \log b)$.

These imply that when a and b are positive integers, computing $c = a^b$ takes $O(\log^2 c)$ steps, and computing $a^b \bmod m$ takes $O(\log a \log m + \log b \log^2 m)$ steps.

In the sequel, p and q will always be prime. For $x > 0$, $\pi(x)$ denotes the number of primes less than or equal to x, and $\pi_p(x)$ the number of primes less than x and congruent to 1 modulo p. The notation $p \,||\, n$ means that $p \mid n$ (p divides n), but not $p^2 \mid n$. By the Extended Riemann Hypothesis (ERH) we mean the Riemann Hypothesis applied to Dirichlet L-functions (see [9]).

In asymptotic bounds expressed with O, o, and Θ, the implied constants are absolute.

3. Root Computation Algorithms.

In this section, we review the usual root-finding algorithm for perfect powers, which seems to be folklore. We present a slight variation of one given by Shallit [26].

The idea is very simple. If n is a perfect kth power, $k \leq \log n$. So, we compute an integer approximation x of $n^{1/k}$ for each such k, starting with $k = 2$. If for some k, $x^k = n$, we accept n and halt. If we reach $k = \log n$ without finding an exact root x, we reject n and halt. We can do a little better by noting that only prime values of k need be tried. Putting this together we get Algorithm A.

Algorithm A.
Input: positive integer n.
  For each prime $p \leq \log n$:
    Compute $x = \lfloor n^{1/p} \rfloor$
    If $n = x^p$ accept and halt
  Reject and halt.

To compute $x = \lfloor n^{1/p} \rfloor$, we first use the value of $\log n$ to get a rough upper limit for x, say $2^{\lfloor \log n/p \rfloor + 1}$. Then we do a binary search between 2 and this limit. To test each possible pth root y encountered in the search, we can compute y^p using any reasonable powering algorithm.

THEOREM 3.1 [26]. *Given as input a positive integer n, Algorithm A will decide if n is a perfect power in time $O(\log^3 n \log\log\log n)$.*

PROOF: Clearly Algorithm A is correct.

Each time a pth root is computed, binary search will iterate $O(\frac{1}{p}\log n)$ times. During each iteration, an approximation y of the pth root of n is raised to the pth power. This takes time $O(\log^2 n)$. So then for a fixed exponent p, $\lfloor n^{1/p} \rfloor$ can be computed in time $O(\frac{1}{p}\log^3 n)$. Summing over all prime exponents p gives

$$\sum_{p \leq \log n} \frac{1}{p} \log^3 n = O\left(\log^3 n \log\log\log n\right)$$

since $\sum_{p \leq b} 1/p = O(\log\log b)$, where the sum is over primes (see Rosser and Schoenfeld [25]).

That takes care of everything, except for finding all the primes below $\log n$. Using the Sieve of Eratosthenes, this can be done in time $O(\log n \log\log\log n)$. Hence the overall running time is $O(\log^3 n \log\log\log n)$, as we claimed. ∎

Note that there are more efficient algorithms than the Sieve of Eratosthenes for finding all the primes below a bound m. Pritchard [24] discusses some of these, and presents an algorithm which uses only $O(m)$ additions and $O(\sqrt{m})$ space.

By replacing binary search with Newton's method, we can improve the running time of Algorithm A. However, without a good starting point, Newton's method is no better than binary search. So we first prove that if the first $\log p$ bits of $n^{1/p}$ are provided, Newton's method converges quadratically.

LEMMA 3.2. *Let $f(x) = x^p - n$, and let $r > 0$ satisfy $f(r) = 0$. Suppose x_0 satisfies $0 \leq (x_0/r) - 1 \leq 2^{-\log p}$. Then, Newton's method, using x_0 as an initial estimate for r, will obtain a estimate of absolute error less than 1, in $O(\log(\frac{1}{p}\log n))$ iterations.*

PROOF: Newton's method computes successive approximations $x_1, x_2, \ldots$ to r using the iteration $x_{n+1} = g(x_n)$, where $g(x) = x - f(x)/f'(x)$. Note that $g(r) = r$ and $g'(r) = 0$. Using a Taylor expansion for g around $x = r$, the mean value theorem implies that for

some α, $r \le \alpha \le x_n$,

$$g(x_n) - r \quad = \quad \frac{g''(\alpha)(x_n - r)^2}{2!} \quad = \quad \frac{(p-1)n}{2\alpha^{p+1}}(x_n - r)^2 \quad \le \quad \frac{p}{2\alpha}(x_n - r)^2.$$

Since $g(x_n) = x_{n+1}$, dividing by r gives

$$\frac{x_{n+1}}{r} - 1 \quad \le \quad \frac{p}{2(\alpha/r)}\left(\frac{x_n}{r} - 1\right)^2 \quad \le \quad \frac{p}{2}\left(\frac{x_n}{r} - 1\right)^2.$$

Thus, if $(x_n/r) - 1 \le 2^{-(\log p + i)}$ for some $i \ge 0$, then $(x_{n+1}/r) - 1 \le 2^{-(\log p + 2i + 1)}$. In other words, at each step we double the number of bits of r we have found. To get an error less than 1 means we only have to match $O(\frac{1}{p}\log n)$ bits, so the number of iterations is $O(\log(\frac{1}{p}\log n))$. ∎

Note that this result still holds if we use $\lfloor g(x) \rfloor$ for the iteration function, which is easy to compute. Now we assume that $\lfloor n^{1/p} \rfloor$ is computed by obtaining the first $\log p$ bits of $n^{1/p}$ with binary search, and the rest with Newton's method. Calling this the *Modified Newton Method*, the following theorem holds.

THEOREM 3.3. *Using the Modified Newton Method to find roots, Algorithm A runs in time* $O(\log^3 n)$.

PROOF: For each p, the number of approximations needed is $\log p + \log(\frac{1}{p}\log n) = O(\log \log n)$. Since the cost of each iteration is $O(\log^2 n)$, the total running time is

$$\sum_{p \le \log n} O(\log^2 n \log \log n) \quad = \quad O(\log^3 n).$$

∎

COROLLARY 3.4. *Computing integer approximations of* k*th roots (for* $k > 1$ *an integer) and solving the recognition problem for the set of perfect powers are both in logspace-uniform* NC^2.

PROOF: Let n be the input. From Lemma 3.2 we know that $O(\log \log n)$ iterations of binary search, followed by $O(\log \log n)$ iterations of the modified Newton method, suffice to compute an integer approximation to the kth root of n. Each iteration requires a power computation and possibly a long division. By methods of Beame, Cook, and Hoover [4] (see also [17]), these two tasks can be done by a circuit of depth $O(\log \log n)$ and size polynomial in $\log n$.

These circuits are not known to be logspace uniform, but the only nonuniformity is a requirement for certain products of small primes, which can be generated easily by logspace uniform circuits of $O(\log^2 \log n)$ depth. So we use an NC^2 circuit for generating these, followed by $O(\log \log n)$ levels of binary search and $O(\log \log n)$ levels of Newton's method, to compute kth roots. The total depth is $O(\log^2 \log n)$.

For recognizing perfect powers, we note that the circuit size for computing pth roots can be bounded independently of p, and simply compute pth roots of the input for every prime $p \leq \log n$, in parallel. ∎

Of course, finding an approximate pth root of n is a special case of finding roots of polynomials over the integers. This more general problem has efficient sequential methods (see, for example, Pan [22]), but parallel solutions seem more difficult. Ben-Or, Feig, Kozen, and Tiwari [5] showed that, if a polynomial has only real roots, then all of its roots can be found in NC. However, it is not known if finding all the complex roots of a polynomial, or even the real roots when some are complex, can be done in NC.

One might ask whether the modified Newton method helps in practice. The answer is yes; in section 7 we support this with timing results comparing binary search with the modified Newton method when used in Algorithm A.

4. A Simple Sieve Algorithm.

In this section we present a second algorithm that improves on Algorithm A's average running time. Our main idea is the following. Most numbers are not perfect powers, so the algorithm will run better if it can quickly reject them. We will do this using the following lemma.

LEMMA 4.1. *Let p and q be primes, with $q \equiv 1 \pmod{p}$. Further suppose that n is a perfect pth power, and $\gcd(n, q) = 1$. Then $n^{(q-1)/p} \equiv 1 \pmod{q}$.*

PROOF: If $n = x^p$, then by Fermat's little theorem, $n^{(q-1)/p} = x^{q-1} \equiv 1 \pmod{q}$. ∎

In other words, if $n^{(q-1)/p} \not\equiv 1 \pmod{q}$, then n cannot be a perfect pth power, assuming all the other hypotheses are satisfied. We will call this computation a *sieve test for exponent* p, and the prime modulus q the *sieve modulus*. We say n *passes* the test if $n^{(q-1)/p} \equiv 0, 1 \pmod{q}$; it *fails* the test otherwise. (Note that we are evaluating the pth power residue symbol mod q; see [15].)

We modify Algorithm A as follows. Before computing an approximate pth root, we do a certain number of sieve tests on n for exponent p. The number of tests, $\lceil 2 \log\log n / \log p \rceil$, will be justified later; it results from balancing the goals of accuracy (few non-perfect powers should pass all the sieve tests) and efficiency (not too many sieve tests should be done). These modifications give the following procedure.

Algorithm B.
Input: positive integer n.
  For each prime $p \leq \log n$:
    Perform up to $\lceil 2 \log\log n / \log p \rceil$ sieve tests on n
    If n passed all the tests, then:
      Compute $x = \lfloor n^{1/p} \rfloor$
      If $n = x^p$ accept and halt
    End if
  Reject and halt.

To analyze this procedure, two questions must be answered.

1. Do enough sieve moduli exist, and if they do, how large are they? Moreover, how do we find them?
2. What are the chances n will pass all the sieve tests for a fixed p?

Regarding the first question, Dirichlet showed that any "reasonable" arithmetic progression contains infinitely many primes (see [15]). Although this guarantees the existence of enough sieve moduli, it says nothing about their size. To reasonably bound this we will have to assume the ERH; in section 6 we will prove the following result.

LEMMA 4.2 [ERH]. *If every input for Algorithm B is less than or equal to n, then the largest sieve modulus needed is $O(\log^2 n \log^4 \log n)$.*

This suggests that the required sieve moduli are all small, and our experience with the algorithm corroborates this. In fact, we believe that the above result is still an overestimate, and offer in section 7 a heuristic argument and numerical evidence for a sharper estimate of $\log n \ln^2(\log n)$. In practice, the sieve of Eratosthenes will suffice to quickly generate the primes less than this bound, and hence the required list of sieve moduli.

Regarding the second question, we first argue informally. The chance an integer n is a pth power modulo q is about $1/p$. If we perform $2 \log\log n / \log p$ sieve tests whose results are independent, the chance n passes them all is about $(1/p)^{\frac{2\log\log n}{\log p}} = 1/\log^2 n$. However, the tests are not quite independent and we must modify this rough argument. In section 6 we will prove the following result, which uses the ERH only to bound the magnitude of the sieve moduli.

LEMMA 4.3 [ERH]. *Let n be an integer chosen uniformly from an interval of length L, and assume for every such n, $L \geq (\log n)^{3 \log\log\log n}$. Then the probability n passes $\lceil \frac{2\log\log n}{\log p} \rceil$ different sieve tests for a fixed exponent p in Algorithm B is bounded above by*

$$O\left(\frac{\log\log n}{\log^2 n}\right).$$

We expect that any implementation of our algorithm will have available a list of sieve moduli. For this reason, we will analyze Algorithm B under the assumption that they have been precomputed. In particular, we assume that Algorithm B has available a table, called the *sieve table*, which contains, for each prime $p \leq \log n$, a list of the first $\lceil \frac{2\log\log n}{\log p} \rceil$ sieve moduli for p. Note that the number of entries in this table is at most

$$\sum_{p \leq \log n} \left\lceil \frac{2\log\log n}{\log p} \right\rceil \quad = \quad O\left(\frac{\log n}{\log\log n}\right)$$

and so the total space used by the table, once computed, is $O(\log n)$. Using the Sieve of Eratosthenes, or a variant, the table can be constructed in $O(\log^{2+\epsilon} n)$ time. In practice this is pessimistic, as we demonstrate in section 7.

We also note that computing the exact values of $\lceil \frac{2\log\log n}{\log p} \rceil$ for each prime $p \leq \log n$ can be done using the methods of Brent [6] easily within the above time bound. In practice, of course, this is not a concern.

Assuming the above lemmas, we now present our average-case result.

THEOREM 4.4 [ERH]. *Let n and L be as in Lemma 4.3, and assume that a sieve table is available. Then Algorithm B will decide if n is a perfect power in expected time $O(\log^2 n)$.*

PROOF: The correctness of Algorithm B follows immediately from Lemma 4.1; it is independent of the ERH.

To get an upper bound on the running time, we may assume that all the possible sieve tests are actually done. By Lemma 4.2, $\log q = O(\log p)$. Since the sieve table is precomputed, the time to find each sieve modulus q is $O(1)$. Computing $n^{(q-1)/p} \bmod q$ can be done using one division and then modular exponentiation in time $O(\log n \log q + \log^3 q) = O(\log n \log p + \log^3 p) = O(\log n \log p)$ since $p \leq \log n$. If $\lceil \frac{2 \log \log n}{\log p} \rceil$ sieve tests are performed, the total time spent is at most $O(\log n \log \log n)$ for each prime exponent p.

From the proof of Theorem 3.3, the time needed to approximate the pth root of n and compute its pth power is $O(\log^2 n \log \log n)$. However, by Lemma 4.3, the probability that we even have to make the computation is $O(\log \log n / \log^2 n)$. Thus, the average time spent is $O(\log^2 \log n)$.

Hence, for each prime exponent p, the average time spent is $O(\log n \log \log n)$. Summing over all primes below $\log n$ gives the average time of $O(\log^2 n)$, and the proof is complete. ∎

In connection with Algorithm B, the following question is also of interest: "Can we guarantee that n is a perfect power by only performing sieve tests?" By applying quadratic reciprocity and Ankeny's theorem, Cobham [7] showed that $O(\log^2 n)$ sieve tests suffice to check that n is a square, if the ERH is true. V. Miller [21] has recently extended this result: if one assumes the Riemann hypothesis for certain Hecke L-functions then $O(p^2 \log^2(np))$ sieve tests for the exponent p will prove that n is a pth power. Since we are only interested in $p \leq \log n$, this would imply that $O(\log^{5+\epsilon} n)$ tests suffice to check perfect powers. Note that this leads to another NC algorithm for perfect power testing.

It is also of interest to ask what can be proved without assuming the ERH. The main difficulty with this seems to be in finding a large number of sieve moduli efficiently. When the sieve moduli are not bounded by a small polynomial in $\log n$, then the Sieve of Eratosthenes and its variants are no longer practical for finding them. As an alternative, one might use probabilistic search, but then the sieve moduli found must be *proved* prime. It is an interesting theoretical question as to how this might be done, but such an approach is unnecessary in practice, and we discuss it no further here.

5. A Sieve Algorithm with Trial Division.

In the previous section we discussed Algorithm B, a substantial improvement over Algorithm A, which used a simple sieving idea to weed out non-perfect powers. In this section, we will use trial division by small primes to further improve Algorithm B. The resulting method may be of use in situations, such as factorization, where trial division is done anyway.

Algorithm C.
Input: positive integer n.
  Compute the smallest integer b such that $b \log^2 b \geq \log n$
  $S \leftarrow \{p : \ p \leq \log n / \log b\}$
  Trial divide n by each prime $r \leq b$:
    If a divisor r is found then:
      Find e such that $r^e \mid\mid n$
      $S \leftarrow \{p : p \mid e\}$
      Stop trial division
    End if
  While $S \neq \emptyset$ do:
    $p \leftarrow$ the smallest element of S
    Perform up to $\lceil 2 \log\log n / \log p \rceil$ sieve tests on n for p:
      If for some sieve modulus q, $q \mid n$, then:
        Find e such that $q^e \mid\mid n$
        $S \leftarrow S \cap \{p : p \mid e\}$
      End if
    If n passed all the tests and $p \in S$, then:
      Compute $x = \lfloor n^{1/p} \rfloor$
      If $n = x^p$ accept and halt
    End if
    $S \leftarrow S - \{p\}$
  Reject and halt.

Our basic idea is the following. To test n, we see if n has any small prime divisors, by checking all primes less than or equal to b. The trial division bound b is much smaller than n; we take b to be about $\log n$. There are two ways this modification can help:

1. If we find a prime r that divides n, we can compute the integer e such that $r^e \mid\mid n$. Then if n is a perfect pth power, p must divide e. Since e will typically be quite small, this will greatly reduce the number of possible prime exponents p that must be checked.
2. If we do not find any divisors of n below b, then if n is a perfect pth power, its pth root x must be larger than b. Hence $p \leq \log_b n = \log n / \log b$, which will also save time.

Our new algorithm does trial division up to some bound b, and then applies either 1. or 2. above, depending on whether or not a divisor is found. It then uses an appropriately modified version of Algorithm B. The procedure above, Algorithm C, gives the details.

To analyze this algorithm we will need a technical lemma.

LEMMA 5.1. *Let P be a set of primes, and for a positive integer n, let $e(n)$ denote the largest e such that $p^e \mid n$ for some $p \in P$. If n is chosen uniformly from an interval of length L, then the expected value of $e(n)$ is at most $\frac{|P| \log n}{L} + O(1)$.*

PROOF: At most $L/p^e + 1$ integers in the interval are divisible by p^e. Since $e(n) \geq k$ if

and only if $p^k \mid n$ for some $p \in P$, $\Pr[e(n) \geq k] \leq \sum_{p \in P} 1/p^k + |P|/L$. Using this,

$$\mathrm{E}[e(n)] = \sum_{k=1}^{\lfloor \log n \rfloor} \Pr[e(n) \geq k] \leq \frac{|P| \log n}{L} + 1 + \sum_{p \in P} \sum_{k=2}^{\infty} \frac{1}{p^k} < \frac{|P| \log n}{L} + 2.$$

∎

THEOREM 5.2 [ERH]. *Let n be an integer chosen uniformly from an interval satisfying the hypotheses of Lemma 4.3. Then Algorithm C will decide if n is a perfect power in time*

$$O\left(\frac{\log^2 n}{\log^2 \log n}\right)$$

on the average.

PROOF: Correctness follows from Theorem 4.4 and from the two observations made at the beginning of this section. All that remains is to prove the average running time bound.

First we note that b is $\Theta(\log n / \log^2 \log n)$, from which it follows that $\log b = \Theta(\log \log n)$.

By a large sieve estimate of Jurkat and Richert [16, 4.2] the probability that no prime below b divides n is

$$\prod_{p \leq b} \left(\frac{p-1}{p}\right)\left(1 + O\left(\frac{1}{\log L}\right)\right),$$

provided that L, the interval length, satisfies $\log b \leq (\log L)/(2 \log \log(3L))$. By our choice of L, this holds for sufficiently large n, so by Mertens's theorem (see [14, p. 351]), the probability of escaping trial division is $O(1/\log b)$, a fact we will need later.

We break the running time into four parts: the time spent on trial division, the time spent computing maximal exponents e for various primes, the time spent on approximating pth roots and computing pth powers of these approximations, and the time spent performing sieve tests.

The time spent on trial division is $O(\sum_{p \leq b} \log n \log p)$. Combining this with our estimate for b shows that the expected time for trial division is $O(\log^2 n / \log^2 \log n)$.

By Lemma 4.2 no sieve modulus or trial divisor is larger than $O(\log^{2+\epsilon} n)$, so the time spent finding a maximal exponent e for any such base is $O(e \log n \log \log n)$. By Lemma 5.1 the expected value of e is $O(1)$, so the expected work for finding maximal exponents is negligible.

To estimate the third part, we note that the time spent computing pth roots and their pth powers is no more than the time Algorithm B would spend in the same task, on a given input. Hence the argument used to prove Theorem 4.4 applies, and we find that the total expected time for this part is $\sum_{p \leq \log n} O(\log^2 \log n)$, which is $O(\log n \log \log n)$.

To estimate the expected time for sieve tests, we condition on whether or not a divisor is found. If a divisor r of n is found, its maximal exponent e is at most $\log n$. Thus e has at most $\log \log n$ prime divisors, which is how many exponents remain to be tested. From the proof of Theorem 4.4, we know the maximum time spent on each possible prime

exponent is $O(\log n \log\log n)$. Thus the average sieve time, given that a divisor is found, is $O(\log n \log^2 \log n)$. If no divisor is found, the average sieve time is similarly found to be $O(\pi(\log n/\log b)\log n \log\log n) = O(\log^2 n/\log b)$. Multiplying this by the probability that no divisor is found and using the asymptotic value of $\log b$ gives the result. ∎

The distribution of Algorithm C's running time exhibits the following anomaly. Although its expected running time is high (consider the fraction $1/\log b$ of inputs for which all trial divisions are performed), its median running time is much lower, in fact $O(\log n)$. We prove this below.

THEOREM 5.3. *Let n be chosen uniformly from an interval of length L, and assume for every such n, $L \geq f(n)$, where $\lim_{x\to\infty} f(x) = \infty$. Then the median running time of Algorithm C is $O(\log n)$.*

PROOF: We must show that there is a set of inputs, of probability at least $1/2$, on which the running time is $O(\log n)$. First consider the asymptotic probability that some prime less than B divides n, and the least such prime divides n exactly once. This is

$$\sum_{p<B} \Pr[\text{ no } q < p \text{ divides } n \text{ and } p \,||\, n\,] = \sum_{p<B} \left(\frac{1}{p} \prod_{q\leq p} \left(1 - \frac{1}{q}\right)\right).$$

If $B = 20$, this is some constant α, greater than $1/2$. Hence with probability $\alpha + o(n)$, $e = 1$ at the end of the trial division phase. Given that this happens, no further work will be required. Furthermore, the work of trial division is $O(\log n)$, since no r greater than 20 and no exponent larger than 2 was ever used on these inputs. ∎

6. Technical Results.

In this section we will prove Lemmas 4.2 and 4.3 with the aid of the ERH. Recall Lemma 4.2 states that the largest sieve modulus needed by Algorithm B is $O(\log^{2+\epsilon} n)$.

Let $\pi_p(x)$ denote the number of primes less than or equal to x which are congruent to 1 modulo p or, what is the same thing, are possible sieve moduli for p. Below we give two estimates for the density of such primes; the first is due to Titchmarsh [28], and the second, which we need later, is similar to a result of Turán [29].

THEOREM 6.1 [ERH]. *Let x be a positive integer, and p a prime. There is a constant $A > 0$, independent of p and x, such that*

$$\pi_p(x) \geq \frac{1}{p-1}\int_2^x \frac{dt}{\ln t} - A\sqrt{x}\ln x.$$

PROOF: See Theorem 6 in [28]. ∎

COROLLARY 6.2 [ERH]. *Let p be a prime and $x = Cp^2\log^4 p$. Then*

$$\pi_p(x) \geq \frac{1}{p-1}\frac{x}{\ln x}\left(1 - O\left(\frac{\ln^2 C}{\sqrt{C}}\right)\right).$$

PROOF: Since $\int_2^x dt/\ln t \geq (x-2)/\ln x$,

$$\pi_p(x) \geq \frac{1}{p-1}\frac{x}{\ln x}\left(1-\frac{2}{x}-\frac{A\ln x}{\sqrt{x}}\right).$$

The result follows from noting that $1/x \leq 1/C$, $\ln x/\sqrt{x} \leq \sqrt{C}(\ln x/\log p)^2$, and $\ln x/\log p \leq \ln C + 6$. ∎

We now prove the sieve modulus bound.

LEMMA 4.2 [ERH]. *If every input for Algorithm B is less than or equal to n, then the largest sieve modulus needed is $O(\log^2 n \log^4 \log n)$.*

PROOF: Let A be the constant from Theorem 6.1 and choose B so that $\sqrt{B} > 4A\ln^2 2$. Let $x = B\log^2 n(\log\log n)^4$. Then

$$\pi_p(x) \geq \frac{1}{\log n}\frac{x-2}{\ln x} - A\sqrt{x}\ln x \sim \left(\frac{B}{2\ln 2} - 2A\sqrt{B}\ln 2\right)\log n(\log\log n)^3.$$

By the choice of B, the right-hand expression is larger than $\lceil 2\log\log n/\log p\rceil$ for all sufficiently large n. ∎

Now we will prove Lemma 4.3, which states that an input n to Algorithm B is unlikely to pass multiple sieve tests.

LEMMA 4.3 [ERH]. *Let n be an integer chosen uniformly from an interval of length L, and assume for every such n, $L \geq (\log n)^{3\log\log\log n}$. Then the probability n passes $\lceil \frac{2\log\log n}{\log p}\rceil$ different sieve tests for a fixed exponent p in Algorithm B is bounded above by*

$$O\left(\frac{\log\log n}{\log^2 n}\right).$$

PROOF: Let T be the set of sieve moduli from the sieve table for p. Define $m = \prod_{q\in T} q$, and note that $|T| = \lceil \frac{2\log\log n}{\log p}\rceil$. Write $L = dm + r$ where d and r are integers satisfying $d > 0$, $0 \leq r < m$. The chance that n passes all $\lceil \frac{2\log\log n}{\log p}\rceil$ sieve tests is, by the Chinese Remainder Theorem, at most

$$\frac{dm}{L}\prod_{i=1}^{|T|}\left[\frac{q_i-1}{q_i}\left(\frac{1}{p}\right)+\frac{1}{q_i}\right]+\frac{r}{L} \quad = \quad \frac{dm}{L}\prod_{i=1}^{|T|}\frac{1}{p}\left(1+\frac{p-1}{q_i}\right)+\frac{r}{L},$$

where $q_1, \ldots, q_{|T|}$ are the primes in T in increasing order. Since $q_i > ip$, $1 + (p-1)/q_i < (i+1)/i$, so the first term is at most

$$\frac{1}{\log^2 n}\prod_{i=1}^{|T|}\frac{i+1}{i} \quad = \quad O\left(\frac{\log\log n}{\log^2 n}\right).$$

To estimate the second term, let $x = 16p^2(\log^5 p)|T|\log|T|$. Noticing that $C = 16(\log p)|T|\log|T| > 2\log\log n \to \infty$ with n, by Corollary 6.2,

$$\pi_p(x) \geq \frac{x}{p\ln x}(1-o(1)) \geq p\log^4 p\,|T| > |T|$$

for sufficiently large n. So we can assume that the largest sieve modulus in T is at most x. Since $r < m$, it suffices to show $m/L = o(1/\log^2 n)$, which is true since

$$m = \prod_{q\in T} q \leq x^{|T|} = (\log n)^{2\log\log\log n(1+o(1))}.$$

∎

It would be interesting to show that this result holds for intervals of "polynomial size," that is, $L \geq \log^c n$ for some $c > 0$. We require slightly larger intervals, in which only the last $\Omega(\log\log n\log\log\log n)$ bits of the input vary.

Finally, we remark that it would be easy to modify our algorithm so that a corresponding result held for the interval $[1, n]$.

7. Implementation Results.

In this section we give empirical results on our algorithms. As the performance of Algorithms B and C is sensitive to the size of the entries in the sieve table, we also give a heuristic argument, backed up by experimental data, that this table is efficiently computable.

Lemma 4.2 indicates that the sieve table will have entries bounded by $O(\log^{2+\epsilon} n)$. In practice, we have found this bound to be pessimistic, and believe that $\log n\ln^2\log n$ is a more accurate estimate. Below we give a heuristic argument, patterned after one by Wagstaff [30], that results in this bound.

Let p be the largest prime less than or equal to $B = \log n$. We consider a sieve modulus bound $m > B$ and estimate the probability that this suffices to obtain enough sieve moduli for all primes up to B. Call a prime "small" if it is less than $(\log n)^{1/3}$, and "large" otherwise. If n is sufficiently large, then Lemma 4.2 guarantees enough sieve moduli for small primes. For large primes, we need at most t sieve moduli, where $t = 6$ (the particular constant does not matter). We now make the heuristic assumption that an integer x is prime with probability $1/\ln x$, and estimate the probability that more than t sieve moduli exist for all large primes. This probability is at least

$$\Pr[\ t \text{ sieve moduli exist for } p\ ]^B,$$

assuming that the chances of success for each p are independent and monotonically decreasing. Now consider a sequence of m/p integers $p+1, 2p+1, \ldots, m$. The probability that there are at most t primes in this sequence is at most

$$\binom{m/p}{t}\left(1-\frac{1}{\ln m}\right)^{m/p-t} \leq (m/p)^t e^{(t-m/p)/\ln m}.$$

Write the right-hand expression as e^y where $y = t\ln(m/p) + (t - m/p)/\ln m$; then

$$\Pr[\text{at least } t \text{ sieve moduli exist for all } p \le B] \ge (1 - e^y)^B.$$

We wish this estimate to be e^{-c} for some small positive c. Setting these equal we find that $e^y = 1 - e^{-c/B} \sim c/B$; taking logarithms of both sides and simplifying we find that

$$\frac{m}{\ln m} \sim B\ln B - B\ln c + \frac{Bt}{\ln m} + Bt\ln(\frac{m}{B})$$

(note that $p \sim B$). If c is a constant, this implies that

$$m \sim B\ln^2 B = \log n \ln^2(\log n).$$

(We can also take $c = 1/(\ln B)^k \to 0$ and get the same result.)

Although its derivation is suspect, this bound seems reasonably precise, as Table 7.1 shows.

Table 7.1.

Decimal Digits	Maximum Modulus	Heuristic Bound	Table Memory	CPU Time in sec.
10	373	407	48	0.005
25	1609	1622	94	0.017
50	3769	4341	147	0.050
100	9767	11197	245	0.116
250	24229	37525	498	0.350
500	78577	91327	865	0.850
1000	152017	218398	1510	2.084
2500	527591	676371	3289	6.700
5000	1449271	1568522	5981	16.166
10000	2839301	3600522	10977	38.700
25000	9731863	10655493	24781	120.333
50000	21021569	23998967	46169	282.466

In this table, the first column lists various values of d for $n = 10^d$. *Maximum Modulus* is the largest sieve modulus, where the first $\lceil 2\log\log n/\log p\rceil$ were taken for each $p \le \log n$. *Heuristic Bound* is the value of $\log n \ln^2 \log n$. *Table Memory* gives the number of integers needed to store all the sieve moduli (the space in words was about double this). Finally, *CPU Time* indicates the number of CPU seconds used to construct the sieve moduli from scratch and store them in a table.

In conclusion, we needed the ERH to give provable bounds on the size of the sieve moduli, and the resulting bounds forced us to use precomputation to construct the table. In practice this is not a problem, as Table 7.1 demonstrates.

Next we give the results of our implementations of Algorithms A, B, and C. We coded all three algorithms using the same multiple precision arithmetic routines on a DEC VAXstation 3200. Table 7.2 gives these results.

Table 7.2.

Decimal Digits	A-BS CPU sec	A-MN CPU sec	B CPU sec	C CPU sec
10	0.155	0.309	0.033	0.046 (10)
25	0.519	0.984	0.048	0.065 (8)
50	1.751	2.960	0.079	0.093 (7)
75	4.058	6.030	0.121	0.091 (9)
100	8.150	10.193	0.173	0.112 (9)
150	22.034	22.645	0.299	0.128 (10)
250	87.453	68.540	0.626	0.229 (9)
500	636.780	349.803	2.218	0.312 (10)
750	2109.134	1012.668	8.845	0.442 (10)
1000	4936.074	2096.609	28.676	0.699 (9)
1500	16256.909	6263.153	131.138	0.826 (10)
2000	-	-	321.581	1.540 (9)

Decimal Digits is the size of the inputs; we ran each algorithm on the same 10 pseudo-random integers and tabulated the average running time for each algorithm. There are two columns for algorithm A: the first gives timings using binary search, and the second gives timings using the modified Newton method. Note that Algorithm B's running time seems relatively large for the last few input sizes; we believe this is caused by multi-word sieve moduli. For Algorithm C we also give the number of times (out of 10) a divisor was found during trial division; this accounts for the irregularities in Algorithm C's running time.

For Algorithm B we did not precompute the sieve table. Instead, a modified version of the Sieve of Eratosthenes found all the primes below the heuristic bound, which were then stored in a bit vector. When sieve moduli were needed, we searched for them sequentially in this bit vector.

For Algorithm C, we first found the primes below b, the trial division bound. Only when trial division failed did we find all the primes below the heuristic bound.

The system's timing clock has unreliable low order digits, so we ran each algorithm several times on the same input and timed the whole group. The number of times per input was inversely proportional to an algorithm's average running time. The program which generated the data in the table took about four days to run.

References

1. E. Bach, *How to generate factored random numbers*, SIAM J. Comput. **2** (1988), 179–193.

2. E. Bach, G. Miller, and J. Shallit, *Sums of divisors, perfect numbers, and factoring*, SIAM J. Comput. **4** (1986), 1143–1154.
3. E. Bach and J. Shallit, *Factoring with cyclotomic polynomials*, Math. Comp. **52** (1989), 201–219.
4. P. W. Beame, S. A. Cook, and H. J. Hoover, *Log depth circuits for division and related problems*, SIAM J. Comput. **15** (1986), 994–1003.
5. M. Ben-Or, E. Feig, D. Kozen, and P. Tiwari, *A fast parallel algorithm for determining all roots of a polynomial with real roots*, SIAM J. Comput. **17** (1988), 1081–1092.
6. R. P. Brent, *Multiple precision zero-finding methods and the complexity of elementary function evaluation*, in "Analytic Computational Complexity," J. F. Traub, Ed., Academic Press, 1976, pp. 151–176.
7. A. Cobham, *The recognition problem for the set of perfect squares*, IBM Research Report RC 1704 (1966).
8. G. E. Collins, *Computing multiplicative inverses in $GF(p)$*, Math. Comp. **23** (1969), 197–200.
9. H. Davenport, "Multiplicative Number Theory," Springer-Verlag, New York, 1980.
10. J. D. Dixon, *Asymptotically fast factorization of integers*, Math. Comp. **36** (1981), 255–260.
11. S. Goldwasser and J. Kilian, *Almost all primes can be quickly certified*, 18th Ann. ACM Symp. Theory Comp. (1986), 316–329.
12. S. W. Golomb, *A new arithmetic function of combinatorial significance*, J. Number Theory **5** (1973), 218–223.
13. J. L. Hafner and K. S. McCurley, *On the distribution of running times of certain integer factoring algorithms*, J. Algorithms (to appear).
14. G. H. Hardy and E. M. Wright, "An Introduction to the Theory of Numbers," 5th ed., Oxford University Press, 1979.
15. K. Ireland and M. Rosen, "A Classical Introduction to Modern Number Theory," Springer-Verlag, New York, 1982.
16. W.B. Jurkat and H.-E. Richert, *An improvement of Selberg's sieve method I*, Acta Arith. **11** (1965), 217–240.
17. R. Karp and V. Ramachandran, *A survey of parallel algorithms for shared memory machines*, Technical Report UCB/CSD 88/408, Computer Science Division, University of California, (1988). To appear in "Handbook of Theoretical Computer Science," North-Holland.
18. D. E. Knuth, "The Art of Computer Programming," vol. 2, 2nd edition, Addison-Wesley, Reading, Mass., 1981.
19. D. E. Knuth and L. Trabb Pardo, *Analysis of a simple factorization algorithm*, Theor. Comp. Sci. **3** (1976), 321–348.
20. H. W. Lenstra Jr., *Factoring integers with elliptic curves*, Ann. Math. **126** (1987), 649–673.
21. V. Miller, *Private communication.*
22. V. Pan, *Fast and efficient algorithms for sequential and parallel evaluation of polynomial zeros and of matrix polynomials*, 26th Ann. IEEE Symp. Foundations Comp. Sci. (1985), 522–531.
23. C. Pomerance, *Fast, rigorous factorization and discrete logarithm algorithms*, in "Discrete Algorithms and Complexity: Proceedings of the Japan-US Joint Seminar," eds. D. S. Johnson, A. Nishizeki, A. Nozaki, H. S. Wilf, Academic Press, London, 1987, pp. 119–143.
24. P. Pritchard, *Fast compact prime number sieves (among others)*, J. Algorithms **4** (1983), 332–344.
25. J. B. Rosser and L. Schoenfeld, *Approximate formulas for some functions of prime numbers*, Ill. J. Math **6** (1962), 64–94.
26. J. Shallit, *Course notes for Number Theory and Algorithms*, Dartmouth College.
27. V. Shoup, *On the deterministic complexity of factoring polynomials over finite fields*, Info. Proc. Letters (to appear).
28. E. C. Titchmarsh, *A divisor problem*, Rend. Circ. Mat. Palermo **54** (1930), 414–429.
29. P. Turán, *Über die Primzahlen der arithmetischen Progression*, Acta Sci. Math. **8** (1936/1937), 226–235.
30. S. S. Wagstaff Jr., *Greatest of the least primes in arithmetic progressions having a certain modulus*, Math. Comp. **33** (1979), 1073–1080.