DETECTING PERFECT POWERS IN ESSENTIALLY LINEAR TIME

Daniel J. Bernstein

950518 (draft T)

Abstract. We give complete details of an algorithm to compute approximate kth roots. We use this in an algorithm that, given an integer $n > 1$, either writes n as a perfect power or proves that n is not a perfect power. Using Loxton's theorem on multiple linear forms in logarithms, we prove that this perfect-power decomposition algorithm runs in time $(\log n)^{1+o(1)}$.

1. Introduction

An integer $n > 1$ is a **perfect power** if there are integers x and $k > 1$ with $n = x^k$. Note that $k \le \log_2 n$; also, the minimal k is prime.

A **perfect-power detection algorithm** is an algorithm that, given an integer $n > 1$, figures out whether n is a perfect power. A **perfect-power decomposition algorithm** does more: if n is a perfect power, it finds x and $k > 1$ with $n = x^k$. A **perfect-power classification algorithm** does everything one could expect: it writes n in the form x^k with k maximal.

Theorem 1. *There is a perfect-power classification algorithm that for $n > 2$ uses time at most $(\log_2 n)^{1+o(1)}$.*

A more precise bound is $(\log_2 n)\exp(O(\sqrt{\log\log n \log\log\log n}))$ for $n > 16$.

This paper is organized as a proof of Theorem 1. In Part I we review integer and floating-point arithmetic. In Part II we develop an algorithm to compute kth roots. In Part III we present a perfect-power decomposition algorithm, Algorithm X. We bound the run time of Algorithm X in terms of a function $F(n)$. In Part IV and Part V we analyze $F(n)$. We complete the proof of Theorem 1 by showing that $F(n)$ is essentially linear in $\log n$. In Part VI we present a 2-adic variant of Algorithm X.

A table of notation appears in section 32.

Motivation. Before attempting to factor n with algorithms such as the number field sieve [15], one should make sure that n is not a perfect power, or at least not a prime power. This is a practical reason to implement *some* power-testing method, though not necessarily a quick one.

1991 *Mathematics Subject Classification.* Primary 11Y16; Secondary 11J86, 65G05.

This paper was included in the author's thesis at the University of California at Berkeley. The author was supported in part by a National Science Foundation Graduate Fellowship.

Speed is more important in other applications. According to [17] there is a theoretically interesting method of finding all small factors of n (to be presented in a successor to [18]) for which perfect-power classification can be a bottleneck.

See [3, section 1] for another example. Here average performance (for n chosen randomly from a large interval) is more important than worst-case performance. We consider the average performance of Algorithm X in section 21.

We digress briefly into applied computational complexity theory. It is fashionable to ask, given a computational problem, whether there is an algorithm that solves the problem in polynomial time; here "polynomial" means "polynomial in the number of input bits." We suggest a different question: is there an algorithm that solves the problem in essentially linear time? Here **linear** means "linear in the number of input bits plus the number of output bits," and **essentially** means "allowing a $1+o(1)$ exponent." For many broad classes of problems the answer is yes. Theorem 1 says that the answer is yes for perfect-power classification.

For readers who want to compute high-precision inverses and roots. One of our major tools is of independent practical interest. In section 11 we present complete theoretical and practical details of an algorithm to compute $y^{-1/k}$ to b bits. Our presentation is designed for the benefit of implementors; see the notes in section 11 for further discussion.

For readers interested in transcendental number theory. Another of our major tools is of independent theoretical interest. Section 26 contains a corrected proof of a bound on the number of perfect powers in a short interval. Both the bound and the corrected proof are due to Loxton; the underlying theorem about linear forms in logarithms is also due to Loxton. Sections 23, 24, 25, and 26 may be read independently of the rest of the paper.

2. Acknowledgments

Many thanks to Hendrik W. Lenstra, Jr., John Loxton, Michael Parks, Igor Shparlinski, and Jon Sorenson for their helpful comments. Lenstra pointed out that the method of Part III had a p-adic analogue; Shparlinski suggested that [19] would help in the $F(n)$ analysis; Loxton supplied the corrected proof of [19, Theorem 1] shown in section 26.

Part I. Arithmetic

3. Summary of Part I

Section 4: $M(b)$ is an upper bound on the time used to compute the product of two b-bit integers; M-time means time spent multiplying integers. $M(b)$ is bounded by $b\mu(b)$, where μ is a nondecreasing function.

Section 5: A floating-point number is an integer divided by a power of 2; the pair (a, n) represents the floating-point number $2^a n$. We define $\operatorname{mul}(r, k) = kr$.

Section 6: To speed up floating-point computations we often truncate a floating-point number r to b bits; the result is written $\operatorname{trunc}_b r$. We can calculate the product ss', with $s = \operatorname{trunc}_b r$ and $s' = \operatorname{trunc}_b r'$, in M-time at most $M(b)$. We also define $\operatorname{div}_b(r, k)$ as a b-bit floating-point approximation to r/k, for k a positive integer.

Section 7: Let k and b be positive integers. Then $\operatorname{pow}_b(r, k)$ is a floating-point approximation to r^k: it satisfies $1 \le r^k/\operatorname{pow}_b(r, k) < (1 + 2^{1-b})^{2k-1}$. We can compute $\operatorname{pow}_b(r, k)$ in M-time at most $P(k)M(b)$, where $P(k) \le 2\lfloor \lg k \rfloor$.

4. Integer arithmetic

Inside a computer we represent a positive integer n as a string of bits giving n's binary expansion. It is easy to add and subtract integers in this form.

A **b-bit number** is a positive integer smaller than 2^b. The string representing such an integer has at most b bits.

Let $M(b)$ be an upper bound on the time used to compute the product of two b-bit numbers. We think of integer multiplication as a black box. We write **M-time** for time spent inside this black box. Then the M-time used by any algorithm will be bounded by a sum of various $M(b)$'s.

There is an algorithm that, for realistic computers in some time scale, computes the product of two b-bit numbers in time $b \lg 2b \lg \lg 4b$, so we may take $M(b) = b \lg 2b \lg \lg 4b$.

Define $\mu(b) = \max \{M(j)/j : 1 \le j \le b\}$. Then $\mu(b)$ is a nondecreasing function of b, with $M(b) \le b\mu(b)$. If $M(b)/b$ is already nondecreasing then $\mu(b) = M(b)/b$.

Lemma 4.1. *If $K(t) \in t^{o(1)}$ and $L(v) = \max \{K(t) : t \le v\}$ then $L(v) \in v^{o(1)}$.*

So if $M(b) \in b^{1+o(1)}$ then $\mu(b) \in b^{o(1)}$. When $\mu(b) \in b^{o(1)}$ we say that we are using **fast multiplication**.

Proof. Fix $\epsilon > 0$. Select $u > 1$ such that $-\epsilon \lg t < \lg K(t) < \epsilon \lg t$ for $t > u$. Fix $v > \max \{u, L(u)^{1/\epsilon}\}$. For $t \le u$ we have $K(t) \le L(u)$ so $\lg K(t) \le \lg L(u) < \epsilon \lg v$; and for $u < t \le v$ we have $\lg K(t) < \epsilon \lg t \le \epsilon \lg v$. Hence $\lg L(v) < \epsilon \lg v$. On the other hand $\lg L(v) \ge \lg K(v) > -\epsilon \lg v$. □

Notes. See [28] or [1] for a multiplication algorithm taking time $b \lg 2b \lg \lg 4b$. See [14, section 4.3.3] for a discussion of fast multiplication algorithms.

Note that it is possible to build a different $b \lg 2b \lg \lg 4b$ multiplication algorithm out of the method of [23].

Later we will need algorithms for multiplying and dividing by *small* integers. Instead of our fast black box we use the methods presented in, e.g., [14, section 4.3.1]; we do *not* count these operations as taking M-time. In practice one can use [14, exercise 4.3.1–13] for multiplication and [14, exercise 4.3.1–16] for division.

The time we spend on perfect-power testing is almost exclusively M-time. We generally omit discussion of non-M-time; the reader may verify that operations other than multiplication do not take much time.

5. Floating-point arithmetic

A **positive floating-point number** is a positive integer divided by a power of 2. The computer can store a pair (a, n), with a an integer and n a positive integer, to represent the positive floating-point number $2^a n$. (In practice $|a|$ is always very small, so a can be stored as a machine integer.)

Notice that (a, n) and $(a-1, 2n)$ represent the same number. The computer can shift among representations. It could repeatedly divide n by 2 until it is odd, for example, to make the representation unique.

Let (a, n) and (a', n') represent the positive floating-point numbers $r = 2^a n$ and $r' = 2^{a'} n'$ respectively. Set $f = \min \{a, a'\}$. Then $r + r' = 2^f (2^{a-f} n + 2^{a'-f} n')$ is represented by the pair $(f, 2^{a-f} n + 2^{a'-f} n')$. Similarly, if $r > r'$, then $r - r'$ is a positive floating-point number represented by the pair $(f, 2^{a-f} n - 2^{a'-f} n')$.

Multiplication is easier: $(a+a', nn')$ represents the product rr'. If n and n' are both b-bit numbers then the M-time here is at most $M(b)$.

We define $\operatorname{mul}(r,k) = kr$ for r a positive floating-point number and k a positive integer. By convention we do not compute $\operatorname{mul}(r,k)$ with our black box for integer multiplication, so time spent computing mul is not M-time. Instead we use an algorithm designed for multiplying by small integers.

Notes. A typical computer has hardware designed to handle a finite set of floating-point numbers. One may study the extent to which operations on the real numbers can be approximated by operations on such a small set [14, section 4.2.2]; the difference is called "roundoff error." Rewriting $2^{a-1}(2n)$ as $2^a n$ is often called "denormalization"; it is rarely considered useful.

Our point of view is somewhat different. We do not worry too much about computer hardware, and we do not work within any particular finite set. We regard approximation not as causing "error" but as limiting the precision used for intermediate operations, thus speeding up our computation. We can work with n more efficiently than $2n$, so we happily rewrite $2^{a-1}(2n)$ as $2^a n$.

A floating-point number is also known as a **dyadic rational** [22, page 435].

6. Floating-point truncation

In this section we define **truncation to b bits**, written trunc_b, and show that $r/\operatorname{trunc}_b r$ is between 1 and $1+2^{1-b}$. More generally, for any positive integer k, we define $\operatorname{div}_b(r,k)$ as a b-bit floating-point approximation to r/k, so that $r/k\operatorname{div}_b(r,k)$ is between 1 and $1+2^{1-b}$.

Fix $b \ge 1$. Set $\operatorname{div}_b(a,n,k) = (a+f-\lceil \lg k\rceil - b, \lfloor n/2^{f-\lceil \lg k\rceil - b}k\rfloor)$, where $2^{f-1} \le n < 2^f$. (Note that $f - \lceil \lg k\rceil - b$ may be negative.) This map induces a map, which we also denote div_b, upon positive floating-point numbers:

$$\operatorname{div}_b(2^a n, k) = 2^{a+f-\lceil \lg k\rceil - b}\lfloor n/2^{f-\lceil \lg k\rceil - b}k\rfloor \qquad \text{if } 2^{f-1} \le n < 2^f.$$

To compute $\operatorname{div}_b(r,k)$ we use an algorithm designed for dividing by small integers; time spent computing div_b, like time spent computing mul, is not counted as M-time.

Lemma 6.1. *Fix $b \ge 1$ and $k \ge 1$. Let r be a positive floating-point number, and set $s = \operatorname{div}_b(r,k)$. Then $s \le r/k < s(1+2^{1-b})$.*

Proof. Put $r = 2^a n$ and define f by $2^{f-1} \le n < 2^f$. Also write $g = f - \lceil \lg k\rceil - b$ and $m = \lfloor n/2^g k\rfloor$, so that $s = 2^{a+g}m$. We have $m \le n/2^g k < m+1$; furthermore

$$m \ge \left\lfloor \frac{2^{f-1}}{2^g k}\right\rfloor = \left\lfloor \frac{2^{\lceil \lg k\rceil}2^{b-1}}{k}\right\rfloor \ge \lfloor 2^{b-1}\rfloor = 2^{b-1}.$$

Thus $2^{a+g}m \le 2^a n/k < 2^{a+g}(m+1) = 2^{a+g}m(1+1/m) \le 2^{a+g}m(1+2^{1-b})$. So $s \le r/k < s(1+2^{1-b})$ as desired. □

For $k=1$ we abbreviate $\operatorname{div}_b(r,k)$ as $\operatorname{trunc}_b r$. So $\operatorname{trunc}_b 2^a n = 2^{a+f-b}\left\lfloor n/2^{f-b}\right\rfloor$ if $2^{f-1} \le n < 2^f$. This map is called **truncation to b bits**. Observe that $\left\lfloor n/2^{f-b}\right\rfloor$ is a b-bit number.

Lemma 6.2. *Fix $b \ge 1$. Let r be a positive floating-point number, and set $s = \operatorname{trunc}_b r$. Then $s \le r < s(1 + 2^{1-b})$.*

Proof. Take $k = 1$ in Lemma 6.1. □

Warning. We will often assert that the computer can calculate the product ss', with $s = \operatorname{trunc}_b r$ and $s' = \operatorname{trunc}_b r'$, in M-time at most $M(b)$. Unfortunately, a malicious computer could use unnecessarily large representations for s and s' so as to slow down the multiplication. We tacitly assume that either (1) values of trunc_b are always represented as pairs (a, n) with n a b-bit number or (2) numbers are scaled appropriately before multiplication.

Notes. For most computers a base such as 2^{32} is more convenient than base 2. It is tempting to replace trunc by a function that keeps a few extra bits "up to the word boundary." One may safely succumb to this temptation: beyond Lemma 6.2, the only property we use of trunc is that two values of trunc_b may be multiplied in M-time $M(b)$.

7. Approximate powers

Let r be a positive floating-point number, and let k and b be positive integers. Then $\operatorname{pow}_b(r, k)$, the **$b$-bit approximate kth power of** r, is a floating-point approximation to r^k. In this section we show how to compute $\operatorname{pow}_b(r, k)$ in M-time at most $P(k)M(b)$, where $P(k) \le 2\lfloor \lg k \rfloor$.

We define $P(k)$ for $k \ge 1$ as follows: $P(1) = 0$; $P(2k) = P(k) + 1$; $P(2k+1) = P(2k) + 1$.

Lemma 7.1. $P(k) \le 2\lfloor \lg k \rfloor$.

Proof. For $k = 1$, $P(k) = 0$ and $\lg k = 0$. If $P(k) \le 2\lfloor \lg k \rfloor$ then $P(k) + 2 \le 2\lfloor \lg k \rfloor + 2 = 2\lfloor \lg 2k \rfloor$, so $P(2k) = P(k) + 1 < P(k) + 2 \le 2\lfloor \lg 2k \rfloor$ and $P(2k+1) = P(k) + 2 \le 2\lfloor \lg 2k \rfloor \le 2\lfloor \lg(2k+1) \rfloor$. □

We define $\operatorname{pow}_b(r, k)$ for $k \ge 1$ as follows:

$$\begin{aligned} \operatorname{pow}_b(r, 1) &= \operatorname{trunc}_b r \\ \operatorname{pow}_b(r, 2k) &= \operatorname{trunc}_b(\operatorname{pow}_b(r, k)^2) \\ \operatorname{pow}_b(r, 2k+1) &= \operatorname{trunc}_b(\operatorname{pow}_b(r, 2k) \operatorname{pow}_b(r, 1)). \end{aligned}$$

Lemma 7.2. $\operatorname{pow}_b(r, k) \le r^k < \operatorname{pow}_b(r, k)(1 + 2^{1-b})^{2k-1}$.

Proof. For $k = 1$ we have $\operatorname{trunc}_b r \le r < (\operatorname{trunc}_b r)(1 + 2^{1-b})$ by Lemma 6.2. For $k > 1$ there is, by definition of pow, some partition $i + j = k$ with $\operatorname{pow}_b(r, k) = \operatorname{trunc}_b(\operatorname{pow}_b(r, i) \operatorname{pow}_b(r, j))$. If $\operatorname{pow}_b(r, i) \le r^i < \operatorname{pow}_b(r, i)(1 + 2^{1-b})^{2i-1}$ and $\operatorname{pow}_b(r, j) \le r^j < \operatorname{pow}_b(r, j)(1 + 2^{1-b})^{2j-1}$ then

$$\begin{aligned} \operatorname{pow}_b(r, k) &\le \operatorname{pow}_b(r, i) \operatorname{pow}_b(r, j) \le r^i r^j \\ &< \operatorname{pow}_b(r, i) \operatorname{pow}_b(r, j)(1 + 2^{1-b})^{2(i+j)-2} \\ &< \operatorname{pow}_b(r, k)(1 + 2^{1-b})^{2k-1} \end{aligned}$$

as desired. □

The definition of $\text{pow}_b(r,k)$ immediately suggests the following algorithm:

Algorithm P. Given a positive floating-point number r and two positive integers b, k, we print $\text{pow}_b(r,k)$.
1. If $k=1$, print $\text{trunc}_b\, r$ and stop.
2. If k is even: Compute $\text{pow}_b(r,k/2)$ by Algorithm P. Print $\text{trunc}_b(\text{pow}_b(r,k/2)^2)$ and stop.
3. Compute $\text{pow}_b(r,k-1)$ by Algorithm P. Print $\text{trunc}_b(\text{pow}_b(r,k-1)\,\text{trunc}_b\, r)$.

Lemma 7.3. *Algorithm P computes* $\text{pow}_b(r,k)$ *in* M*-time at most* $P(k)M(b)$.

Proof. We count the number of multiplications. For $k=1$ there are $P(1)=0$ multiplications. If we use at most $P(k)$ multiplications for $\text{pow}_b(r,k)$, then we use at most $P(k)+1=P(2k)$ multiplications for $\text{pow}_b(r,2k)$, and we use at most $P(2k)+1=P(2k+1)$ multiplications for $\text{pow}_b(r,2k+1)$. $\square$

Notes. Algorithm P is the **left-to-right binary method**, which comes from a broad class of powering algorithms indexed by **addition chains** [14, section 4.6.3]. Lemma 7.2 would remain true if we replaced $\text{pow}_b(r,k)$ with the output from any algorithm in this class. For many k one can find an algorithm using fewer than $P(k)$ multiplications; this is useful in practice. See [14, section 4.6.3] and [10, section 1.2] for further discussion.

For large k it is probably better to compute r^k as $\exp(k \log r)$ by the methods of [8], which take essentially linear time.

Part II. Roots

8. Summary of Part II

In section 11 we present an algorithm to compute reciprocals and kth roots to arbitrary precision. Various useful inequalities appear in sections 9 and 10.

9. Well-known inequalities

Lemma 9.1. *If* $u, v > 0$ *then* $(1+1/uv)^u < e^{1/v}$.

Proof. $e^t \ge 1$ for $t \ge 0$. Integrate twice: $e^t \ge 1+t+t^2/2$. Hence $e^t > 1+t$ for $t>0$. In particular $e^{1/uv} > 1+1/uv$, so $e^{1/v} > (1+1/uv)^u$. $\square$

Lemma 9.2. *If* $0 \le t < 1$ *then* $t/(1-t)+\log(1-t) \ge 0$.

Proof. $t/(1-t)^2 \ge 0$ for $0 \le t < 1$. Integrate. $\square$

Lemma 9.3. *If* $\kappa \ge 1$ *and* $1+\epsilon > 0$ *then* $(1+\epsilon)^\kappa \ge 1+\kappa\epsilon$.

Proof. The function $(1+\epsilon)^\kappa - 1 - \kappa\epsilon$ is zero when $\epsilon = 0$. Its derivative $\kappa((1+\epsilon)^{\kappa-1}-1)$ is nonnegative for $\epsilon > 0$ and nonpositive for $-1 < \epsilon < 0$. $\square$

Lemma 9.4. *If* $\kappa \ge 1$ *and* $0 < \epsilon < 1$ *then* $(1-\epsilon)^\kappa < 1/(1+\kappa\epsilon)$.

Proof. $1 > (1-\epsilon^2)^\kappa = (1+\epsilon)^\kappa(1-\epsilon)^\kappa \ge (1+\kappa\epsilon)(1-\epsilon)^\kappa$ by Lemma 9.3. $\square$

Lemma 9.5. *If* $\kappa > 0$ *and* $1+\epsilon > 0$ *then* $(1+\epsilon)^{-\kappa} \ge 1-\kappa\epsilon$.

Proof. The function $(1+\epsilon)^{-\kappa} - 1 + \kappa\epsilon$ is zero when $\epsilon = 0$. Its derivative $\kappa(1-(1+\epsilon)^{-\kappa-1})$ is positive for $\epsilon > 0$ and negative for $-1 < \epsilon < 0$. $\square$

10. Overly specific inequalities

Lemma 10.1. *If* $\kappa > 0$ *and* $0 < \epsilon < 1$ *then* $(1 + \epsilon/4\kappa)^{2\kappa} < 1 + \epsilon$.

Proof. $(1 + \epsilon)(1 - \epsilon/2) = 1 + (1 - \epsilon)\epsilon/2 > 1$. By Lemma 9.5, $(1 + \epsilon/4\kappa)^{2\kappa} \le 1/(1 - \epsilon/2) < 1 + \epsilon$. □

Lemma 10.2. *If* $\kappa \ge 1$ *then* $7/8 \le (1 - 1/8\kappa)^\kappa$.

Proof. The function $(1-1/8\kappa)^\kappa$ increases as κ increases: the derivative of $\kappa \log(1-1/8\kappa)$ is $1/(8\kappa-1)+\log(1-1/8\kappa)$, which is positive by Lemma 9.2. So $(1-1/8\kappa)^\kappa \ge (1 - 1/8)^1$ for $\kappa \ge 1$. □

Lemma 10.3. *If* $0 < t < 1/36$ *then* $(1 + 3t)(1 + t)(1 + 32t/3) < 1 + 16t$.

Proof. $(1+3t)(1+t)(1+32t/3)-(1+16t) = t(32t^2+(137/3)t-4/3)$ is negative. □

Lemma 10.4. *If* $\kappa \ge 1$ *and* $0 < t < 1/(4\kappa+4)$ *then* $(1+t)^{2\kappa+3}-1 < 16t(7\kappa-2)/9$.

Proof. It suffices to compare derivatives: $(2\kappa+3)(1+t)^{2\kappa+2}$ versus $16(7\kappa-2)/9$. By Lemma 9.1, $(1+t)^{2\kappa+2} < (1+1/(4\kappa+4))^{2\kappa+2} < e^{1/2} < 16/9$; and $2\kappa+3 \le 7\kappa-2$. □

11. Approximate roots

In this section we consider the problem of **root extraction**: computing $y^{1/k}$, given a positive floating-point number y and a positive integer k. We also consider the problem of **inversion**: computing y^{-1}. We solve both problems by showing how to compute $y^{-1/k}$. Then $y^{1/k} = y(y^{-1/k})^{k-1}$. (Alternatively $y^{1/k} = (y^{-1/1})^{-1/k}$; more generally $y^{1/k} = (y^{-1/k_1})^{-1/k_2}$ if $k = k_1k_2$.)

For each positive integer b we construct a floating-point number $\mathrm{nroot}_b(y, k)$ satisfying

$$\mathrm{nroot}_b(y, k)(1 - 2^{-b}) < y^{-1/k} < \mathrm{nroot}_b(y, k)(1 + 2^{-b}).$$

Our method, in brief, is a binary search for small b, and then Newton's method with increasing precision for all larger b.

Binary search: the idea. We are trying to find a root z of $z^ky - 1$. Binary search means guessing the bits of z, one by one. Given an interval I surrounding the root, we evaluate $z^ky - 1$ at the midpoint of I. Depending on the sign of the answer we replace I by either the left half of I or the right half of I. We repeat until I is sufficiently small.

To speed up the computation, we only approximate z^ky-1. If the answer is too close to 0 for us to be sure about its sign, we replace I with the *middle half* of I.

Binary search: the algorithm. For $b \le 3 + \lceil \lg k \rceil$ we define and construct $\mathrm{nroot}_b(y, k)$ by Algorithm B below. For the time spent by Algorithm B, see Lemma 11.1. For the accuracy of its output, see Lemma 11.3.

We comment briefly on the constant 993/1024 in Algorithm B. In the proof of Lemma 11.2 we will need the fact that 993/1024 is between 32/33 and $e^{-1/33}$. It is the "simplest" floating-point number in this range.

Algorithm B. We compute $\mathrm{nroot}_b(y, k)$ for $1 \le b \le \lceil \lg 8k \rceil$. In advance, find the exponent g satisfying $2^{g-1} < y \le 2^g$, and set $a = \lfloor -g/k \rfloor$, so that $2^a \le y^{-1/k} < 2^{a+1}$. Also set $B = \lceil \lg(66(2k + 1)) \rceil$.
1. Set $z \leftarrow 2^a + 2^{a-1}$, $j \leftarrow 1$.

2. (See Lemma 11.2 for an invariant.) Now $\operatorname{nroot}_j(y,k)=z$. If $j=b$, stop.
3. Compute $r \leftarrow \operatorname{trunc}_B(\operatorname{pow}_B(z,k)\operatorname{trunc}_B y)$.
4. If $r \le 993/1024$, set $z \leftarrow z + 2^{a-j-1}$.
5. If $r > 1$, set $z \leftarrow z - 2^{a-j-1}$.
6. Set $j \leftarrow j+1$. Go back to step 2.

Note that, if $993/1024 < r \le 1$, we have an excellent bound on $y^{-1/k}$; z will remain forever unchanged and we can immediately terminate the algorithm.

Lemma 11.1. *For $b \le \lceil \lg 8k \rceil$, Algorithm B computes $\operatorname{nroot}_b(y,k)$ in M-time at most $(b-1)(P(k)+1)M(\lceil \lg(66(2k+1))\rceil)$.*

Proof. We get to $\operatorname{nroot}_b(y,k)$ in $b-1$ iterations. Each iteration takes time at most $P(k)M(B)$ to find $\operatorname{pow}_B(z,k)$, by Lemma 7.3, and time at most $M(B)$ to multiply by $\operatorname{trunc}_B y$. □

Lemma 11.2. $2^a \le z - 2^{a-j} \le y^{-1/k} < z + 2^{a-j} \le 2^{a+1}$ *at step 2 of Algorithm B.*

Proof. We prove this invariant by induction. When $j=1$ we have $z - 2^{a-1} = 2^a \le y^{-1/k} < 2^{a+1} = z + 2^{a-1}$.

Assume the invariant true for j. Then, at step 3, the computation of r gives $r \le z^k y < r(1+2^{1-B})^{2k+1}$ by Lemma 6.2 and Lemma 7.2. By choice of B we have

$$(1+2^{1-B})^{2k+1} < \left(1+\frac{1}{33(2k+1)}\right)^{2k+1} < e^{1/33} < \frac{1024}{993}$$

so $r \le z^k y < r(1024/993)$. We consider three cases separately: $r>1$; $r \le 993/1024$; and $993/1024 < r \le 1$.

When $r>1$ we have $z^k y > 1$. We replace z by $z' = z - 2^{a-j-1}$; and $z' - 2^{a-j-1} = z - 2^{a-j} \le y^{-1/k} < z = z' + 2^{a-j-1}$.

When $r \le 993/1024$ we have $z^k y < (993/1024)(1024/993) = 1$. We replace z by $z' = z + 2^{a-j-1}$; and $z' - 2^{a-j-1} = z < y^{-1/k} < z + 2^{a-j} = z' + 2^{a-j-1}$.

When $993/1024 < r \le 1$ we leave z unchanged. We have $j \le \lceil \lg 8k \rceil - 1 < \lg 8k$ so $2^{-j-2} \ge 1/32k$. Thus $(1+2^{-j-2})^k \ge 1 + 1/32$ by Lemma 9.3, so

$$\begin{aligned}(z+2^{a-j-1})^k y &= z^k y\left(1+\frac{2^{a-j-1}}{z}\right)^k > z^k y\left(1+\frac{2^{a-j-1}}{2^{a+1}}\right)^k \\ &\ge z^k y \frac{33}{32} \ge \frac{993}{1024}\frac{33}{32} > 1.\end{aligned}$$

On the other hand $(1-2^{-j-2})^k < 32/33$ by Lemma 9.4. So

$$\begin{aligned}(z-2^{a-j-1})^k y &= z^k y\left(1-\frac{2^{a-j-1}}{z}\right)^k < z^k y\left(1-\frac{2^{a-j-1}}{2^{a+1}}\right)^k \\ &< z^k y \frac{32}{33} < \frac{1024}{993}\frac{32}{33} < 1.\end{aligned}$$

Hence $z - 2^{a-j-1} < y^{-1/k} < z + 2^{a-j-1}$ as desired. □

Lemma 11.3. $\mathrm{nroot}_b(y,k)(1-2^{-b}) < y^{-1/k} < \mathrm{nroot}_b(y,k)(1+2^{-b})$ *for* $b \le \lceil \lg 8k \rceil$.

Proof. $\mathrm{nroot}_b(y,k)$ appears as z in step 2 of Algorithm B when $j=b$. By Lemma 11.2, $2^a \le z - 2^{a-b} \le y^{-1/k} < z + 2^{a-b} \le 2^{a+1}$. Then $z2^{-b} > 2^{a-b}$ so $z(1-2^{-b}) < z - 2^{a-b} \le y^{-1/k} < z + 2^{a-b} < z(1+2^{-b})$. □

Newton's method: the idea. We are trying to find a root of $h(z) = z^{-k}y^{-1} - 1$. In Newton's method, we replace our first guess, z, by a much better guess, $z - h(z)/h'(z) = z + (z - yz^{k+1})/k$. We repeat until z has the desired accuracy.

Newton's method roughly doubles the number of correct digits on each iteration. To speed the computation, we compute the full-precision answer only on the last step; we work with only 1/2 the digits in the previous step, 1/4 in the step before that, and so on.

Newton's method: the algorithm. For $b \ge 4 + \lceil \lg k \rceil$ we define and construct $\mathrm{nroot}_b(y,k)$ by Algorithm N below. For the accuracy of $\mathrm{nroot}_b(y,k)$, see Lemma 11.7. For the time spent by Algorithm N, see Lemma 11.4.

Algorithm N. We compute $\mathrm{nroot}_b(y,k)$ for $b \ge \lceil \lg 8k \rceil + 1$. In advance set $b' = \lceil \lg 2k \rceil + \lceil (b - \lceil \lg 2k \rceil)/2 \rceil$ and $B = 2b' + 4 - \lceil \lg k \rceil$. Note that $b' < b$.
1. Compute $z \leftarrow \mathrm{nroot}_{b'}(y,k)$, by Algorithm B if $b' \le \lceil \lg 8k \rceil$ or by Algorithm N if $b' \ge \lceil \lg 8k \rceil + 1$.
2. Set $r_2 \leftarrow \mathrm{mul}(\mathrm{trunc}_B\, z, k+1)$.
3. Set $r_3 \leftarrow \mathrm{trunc}_B(\mathrm{pow}_B(z, k+1)\, \mathrm{trunc}_B\, y)$.
4. Set $r_4 \leftarrow \mathrm{div}_B(r_2 - r_3, k)$. Now $\mathrm{nroot}_b(y,k) = r_4$.

Lemma 11.4. *For* $b \ge \lceil \lg 8k \rceil + 1$*, Algorithm N computes* $\mathrm{nroot}_b(y,k)$ *in* M*-time at most* $T + KU$*, where* $T = \lceil \lg 4k \rceil\, (P(k)+1) M(\lceil \lg(66(2k+1)) \rceil)$*,* $K = P(k+1)+1$*, and* $U = (2b + 10 + \lceil 8 + \lg k \rceil \lceil \lg(b - \lceil \lg 2k \rceil) - 3 \rceil)\mu(b+6)$*.*

Proof. We have $B - \lceil \lg 2k \rceil - 5 = 2(b' - \lceil \lg 2k \rceil) = 2\lceil (b - \lceil \lg 2k \rceil)/2 \rceil$, so B is either $b+5$ or $b+6$. Hence $B + 2b' \le b + 6 + 2(b' - \lceil \lg 2k \rceil) + 2\lceil \lg 2k \rceil \le b + 7 + (b - \lceil \lg 2k \rceil) + 2\lceil \lg 2k \rceil = 2b + \lceil 8 + \lg k \rceil$. Note that $\lceil \lg(b' - \lceil \lg 2k \rceil) \rceil = \lceil \lg(b - \lceil \lg 2k \rceil) - 1 \rceil$.

We have $b' \ge \lceil \lg 8k \rceil$. If $b' > \lceil \lg 8k \rceil$, Algorithm N calls itself to compute $\mathrm{nroot}_{b'}(y,k)$. By induction, this call takes M-time at most $T + KU'$, where $U' = (2b' + 10 + \lceil 8 + \lg k \rceil \lceil \lg(b' - \lceil \lg 2k \rceil) - 3 \rceil)\mu(b'+6)$.

If $b' = \lceil \lg 8k \rceil$, Algorithm N calls Algorithm B to compute $\mathrm{nroot}_{\lceil \lg 8k \rceil}(y,k)$. By Lemma 11.1, this call takes M-time at most T. Note that in this case $U' = 0$, since $2\lceil \lg 8k \rceil + 10 + \lceil 8 + \lg k \rceil \lceil \lg(\lceil \lg 8k \rceil - \lceil \lg 2k \rceil) - 3 \rceil = 0$.

So in either case step 1 of Algorithm N takes M-time at most $T + KU'$. By Lemma 7.3, step 3 of Algorithm N uses M-time at most $P(k+1)M(B) + M(B) = KM(B)$. Thus the total M-time used by Algorithm N is at most

$$\begin{aligned}
&T + KM(B) + KU' \\
&\quad = T + KM(B) + K\big(2b' + 10 + \lceil 8 + \lg k \rceil \lceil \lg(b' - \lceil \lg 2k \rceil) - 3 \rceil\big)\mu(b'+6) \\
&\quad \le T + K\big(B + 2b' + 10 + \lceil 8 + \lg k \rceil \lceil \lg(b' - \lceil \lg 2k \rceil) - 3 \rceil\big)\mu(b+6) \\
&\quad \le T + K\big(2b + 10 + \lceil 8 + \lg k \rceil + \lceil 8 + \lg k \rceil \lceil \lg(b' - \lceil \lg 2k \rceil) - 3 \rceil\big)\mu(b+6) \\
&\quad = T + K\big(2b + 10 + \lceil 8 + \lg k \rceil \lceil \lg(b - \lceil \lg 2k \rceil) - 3 \rceil\big)\mu(b+6) = T + KU
\end{aligned}$$

as claimed. □

Lemma 11.5. *Define* $w = ((k+1)z - z^{k+1}y)/k$. *If* $z(1+\epsilon) = y^{-1/k}$ *and* $\epsilon > -1/8k$ *then* $w \le y^{-1/k} \le w(1 + 4k\epsilon^2/3)$.

Proof. $1 - k\epsilon \le (1+\epsilon)^{-k}$ by Lemma 9.5, so

$$k + 1 - z^k y = k + 1 - (1+\epsilon)^{-k} \le k + k\epsilon = \frac{k}{zy^{1/k}},$$

so $w = (z/k)(k + 1 - z^k y) \le y^{-1/k}$.

By Lemma 9.3, $(1+\epsilon)^k \ge 1 + k\epsilon \ge 1 - 1/8 > 1/2$, so $k + 1 \ge 2 > (1+\epsilon)^{-k} = z^k y$, so $w > 0$.

Again, $(1+\epsilon)^k \ge 1 + k\epsilon$, so

$$\begin{aligned}\frac{y^{-1/k}}{w} - 1 &= \frac{1 - (1-k\epsilon)(1+\epsilon)^k}{(k+1)(1+\epsilon)^k - 1} = \frac{1 - (1-k\epsilon)/(k+1)}{(k+1)(1+\epsilon)^k - 1} - \frac{1-k\epsilon}{k+1} \\ &\le \frac{1 - (1-k\epsilon)/(k+1)}{(k+1)(1+k\epsilon) - 1} - \frac{1-k\epsilon}{k+1} = \frac{k\epsilon^2}{1 + \epsilon + k\epsilon} \le \frac{4}{3}k\epsilon^2\end{aligned}$$

since $1 + (k+1)\epsilon \ge 1 - (k+1)/8k \ge 1 - 1/4 = 3/4$. $\square$

Lemma 11.6. *If* $z(1 - 2^{-b'}) < y^{-1/k} < z(1 + 2^{-b'})$ *in Algorithm N then* $r_4(1 - 2^{-b}) < y^{-1/k} < r_4(1 + 2^{-b})$.

Proof. Define $w = ((k+1)z - z^{k+1}y)/k$. The idea is that w is very close to $y^{-1/k}$, $(r_2 - r_3)/k$ is very close to w, and r_4 is very close to $(r_2 - r_3)/k$.

Define ϵ by $z(1+\epsilon) = y^{-1/k}$, so that $-2^{-b'} < \epsilon < 2^{-b'}$. We have $b' \ge \lceil \lg 8k \rceil \ge \lg 8k$ so $2^{-b'} \le 1/8k$ so $-1/8k < \epsilon < 1/8k$.

Note that B is either $b + 5$ or $b + 6$, so $2^{5-B} \le 2^{-b}$. Abbreviate $\delta = 2^{1-B}$. Then $\delta < 1/36$, $\delta < 1/(4k+4)$, and $8(1+\delta) < 9$. Also $(2k)2^{-2b'} \le 2^{1+\lceil \lg k \rceil - 2b'} = 2^{5-B} = 16\delta$.

By construction $r_2 \le (k+1)z < r_2(1+\delta)$, $r_3 \le z^{k+1}y < r_3(1+\delta)^{2k+3}$, and $r_4 \le (r_2 - r_3)/k < r_4(1+\delta)$.

By Lemma 10.2, $7/8 \le (1 - 1/8k)^k < (1+\epsilon)^k = y^{-1}z^{-k}$, so $z^k y < 8/7$. So

$$\frac{r_3}{r_2} \le \frac{z^{k+1}y(1+\delta)}{(k+1)z} = \frac{z^k y(1+\delta)}{k+1} < \frac{8}{7}\frac{1+\delta}{k+1} < \frac{9}{7(k+1)} < \frac{2}{3}.$$

By Lemma 11.5, $w \le y^{-1/k}$, so

$$\begin{aligned}\frac{y^{-1/k}}{r_4} &\ge \frac{w}{r_4} \ge \frac{kw}{r_2 - r_3} = \frac{(k+1)z - z^{k+1}y}{r_2 - r_3} \ge \frac{r_2 - r_3(1+\delta)^{2k+3}}{r_2 - r_3} \\ &= 1 - \frac{(1+\delta)^{2k+3} - 1}{r_2/r_3 - 1} > 1 - \frac{(1+\delta)^{2k+3} - 1}{(7k-2)/9} > 1 - 16\delta\end{aligned}$$

by Lemma 10.4.

On the other hand, by Lemma 11.5, $y^{-1/k} \le w(1 + 4k\epsilon^2/3)$, so

$$\begin{aligned}
\frac{y^{-1/k}}{r_4} &\le \frac{w(1+4k\epsilon^2/3)}{r_4} < \frac{w(1+(2/3)(2k)2^{-2b'})}{r_4} \le \frac{w(1+(32/3)\delta)}{r_4} \\
&< \frac{k(1+\delta)w(1+(32/3)\delta)}{r_2 - r_3} = \frac{(1+\delta)((k+1)z - z^{k+1}y)(1+(32/3)\delta)}{r_2 - r_3} \\
&< \frac{(1+\delta)(r_2(1+\delta) - r_3)(1+(32/3)\delta)}{r_2 - r_3} \\
&= (1+\delta)\left(1 + \frac{32}{3}\delta\right)\left(1 + \frac{\delta}{1 - r_3/r_2}\right) < (1+\delta)\left(1+\frac{32}{3}\delta\right)(1+3\delta) \\
&< 1 + 16\delta
\end{aligned}$$

by Lemma 10.3. □

Lemma 11.7. $\operatorname{nroot}_b(y,k)(1-2^{-b}) < y^{-1/k} < \operatorname{nroot}_b(y,k)(1+2^{-b})$ *for all* b.

Proof. For $b \le \lceil \lg 8k \rceil$ this is Lemma 11.3. For $b \ge \lceil \lg 8k \rceil + 1$ we compute $\operatorname{nroot}_b(y,k)$ as r_4 in Algorithm N. By induction $z(1-2^{-b'}) < y^{-1/k} < z(1+2^{-b'})$, so $r_4(1-2^{-b}) < y^{-1/k} < r_4(1+2^{-b})$ by Lemma 11.6. □

Notes. High-precision inverses and roots show up in many contexts other than perfect-power detection, so Algorithms B and N are arguably the most important algorithms in this paper. The author's goal in writing this section was to produce something "ready to run" and immediately useful in practice.

Algorithms B and N are reasonably "tight": they do not use unnecessarily high precision. As the reader can see, we pay for this tightness in complex proofs, though the algorithms themselves are short and straightforward.

The basic outline of our method is well known, as is its approximate run time. For Newton's method with increasing precision see [8] (which popularized [7]) or [6, section 6.4]. For the specific case of inversion see also [14, Algorithm 4.3.3–R] or [1, page 282]. For a controlled number of steps of binary search as preparation for Newton's method see [3, section 3].

However, it is difficult to find a complete error analysis in the literature, let alone an algorithm carefully tuned for speed in light of such an analysis. An algorithm with results of unknown accuracy—or an algorithm not even stated explicitly—is of little value for implementors.

A notable exception for $k = 1$ is [14, Algorithm 4.3.3–R], which is stated in full detail and supported by tight error bounds; but Algorithm N will be faster, because it pays close attention to the desired final precision.

For Newton's method generally see [25, section 9.4]. The inequalities in Lemma 11.5 follow from general facts of the form "when Newton's method is applied to the following class of nice functions, the iterates exhibit the following nice behavior."

Binary search as a root-finding technique is also known as **bisection**. For bisection generally see [25, section 9.1]. Our use of binary search is closer in spirit to [25, section 9.1] than to [3, section 3] since we limit the precision of intermediate calculations.

There are many other general root-finding methods; see [25]. There are even more methods specific to kth roots.

For large k, just as r^k is probably best computed as $\exp(k \log r)$, $r^{1/k}$ is probably best computed as $\exp((\log r)/k)$ by the methods of [8], which take essentially linear time.

Part III. Power testing

12. Summary of Part III

Section 13: Algorithm C tests if $n = x^k$ in time essentially linear in the number of leading digits needed to distinguish n from x^k.

Section 14: Algorithm K tests if n is a kth power. It finds an approximation to $n^{1/k}$ and then applies Algorithm C.

Section 15: Algorithm X tests if n is a perfect power by running Algorithm K for each possible prime exponent.

Section 16: We define a function $F(n)$, and we survey results on $F(n)$ from Part IV and Part V. The run time of Algorithm X is essentially linear in $F(n)$, if we use fast multiplication.

Section 17: We prove Theorem 1.

13. How to test if $n = x^k$

In this section we consider the problem of testing, given positive integers n, x, and k, whether $n = x^k$.

We could simply compute x^k and check whether it equals n. But we can eliminate most (x, n) more efficiently, by checking whether $n = x^k$ is consistent with the first few digits of n and x.

Our solution, Algorithm C, inspects at most about twice as many leading bits of n as are needed to distinguish n from x^k. See Lemma 13.5 for a precise run-time bound. See Lemma 13.2 for a proof that Algorithm C works.

Algorithm C. Given positive integers n, x, k, we compute the sign of $n - x^k$. In advance set $f = \lfloor \lg 2n \rfloor$.
1. Set $b \leftarrow 1$.
2. Compute $r \leftarrow \mathrm{pow}_{b+\lceil \lg 8k \rceil}(x, k)$.
3. If $n < r$, print -1 and stop.
4. If $r(1 + 2^{-b}) \le n$, print 1 and stop.
5. If $b \ge f$, print 0 and stop.
6. Set $b \leftarrow \min\{2b, f\}$. Go back to step 2.

The farther apart n and x^k are, the more quickly Algorithm C can tell them apart. Define a distance d on integers as $d(i, j) = 0$ when $i = j$, $d(i, j) = \lfloor \lg |i - j| \rfloor$ when $i \ne j$; below we express the speed of Algorithm C in terms of $d(n, x^k)$.

Lemma 13.1. *Set $f = \lfloor \lg 2n \rfloor$. At the start of step 3 of Algorithm C, $r \le x^k < r(1 + 2^{-b})$. At the start of step 4, $r \le n$ and $x^k < r + 2^{f-b}$. At the start of step 5, $n < r + 2^{f-b}$.*

Proof. By Lemma 7.2 and Lemma 10.1,

$$r \le x^k < r\left(1 + \frac{1}{2^{b+\lceil \lg 4k \rceil}}\right)^{2k-1} < r\left(1 + \frac{1}{2^b 4k}\right)^{2k-1} < r(1 + 2^{-b}).$$

If we do not stop in step 3, then $r \le n < 2^f$, so $r(1 + 2^{-b}) < r + 2^{f-b}$. If we also do not stop in step 4, then $n < r(1 + 2^{-b})$. $\square$

Lemma 13.2. *When Algorithm C stops, it prints the sign of $n - x^k$.*

Proof. We use each piece of Lemma 13.1. If we stop in step 3 then $n < r$. But $r \le x^k$ so $n < x^k$ as desired.

If we stop in step 4 then $r(1 + 2^{-b}) \le n$. But $x^k < r(1 + 2^{-b})$ so $x^k < n$ as desired.

If we stop in step 5 then $b \ge f$, so $r \le n < r + 1$ and $r \le x^k < r + 1$. Hence $|n - x^k| < 1$. Both n and x^k are integers, so $n = x^k$ as desired. $\square$

Lemma 13.3. *Set $f = \lfloor \lg 2n \rfloor$ and $g = \max\left\{1, f - d(n, x^k)\right\}$. If $b \ge g$ then Algorithm C stops before step 6.*

Proof. We prove the contrapositive. If Algorithm C gets to step 6 then $b < f$. By Lemma 13.1, $r \le n < r + 2^{f-b}$ and $r \le x^k < r + 2^{f-b}$, so $\left|n - x^k\right| < 2^{f-b}$. If $n = x^k$ then $d(n, x^k) = 0 < f - b$; if $n \ne x^k$ then $d(n, x^k) = \left\lfloor \lg \left|n - x^k\right| \right\rfloor \le \lg \left|n - x^k\right| < f - b$. Either way $b < f - d(n, x^k) \le g$. $\square$

Lemma 13.4. *Set $f = \lfloor \lg 2n \rfloor$, $b_j = \min\left\{2^j, f\right\}$, and $g = \max\left\{1, f - d(n, x^k)\right\}$. Then Algorithm C takes M-time at most $P(k) \sum_{0 \le j \le \lceil \lg g \rceil} M(b_j + \lceil \lg 8k \rceil)$.*

Proof. Each iteration of step 2 uses time at most $P(k)M(b + \lceil \lg 8k \rceil)$, by Lemma 7.3. Notice that $b_{\lceil \lg f \rceil - 1} < f$ but $b_{\lceil \lg f \rceil} = f$. So Algorithm C uses first $b \leftarrow b_0$, then $b \leftarrow b_1$, and so on through at most $b \leftarrow b_{\lceil \lg f \rceil}$. If $j > \lceil \lg g \rceil$ then $j \ge 1$ and $b_{j-1} = \max\left\{2^{j-1}, f\right\} \ge \max\left\{2^{\lceil \lg g \rceil}, g\right\} \ge g$, so Algorithm C stops before step 6 with $b \leftarrow b_{j-1}$, so it never gets to b_j. $\square$

Lemma 13.5. *Set $f = \lfloor \lg 2n \rfloor$ and $g = \max\left\{1, f - d(n, x^k)\right\}$. Then Algorithm C takes M-time less than $P(k)(4g + \lceil \lg 2g \rceil \lceil \lg 8k \rceil)\mu(2g + \lceil \lg 8k \rceil)$.*

Proof. Set $b_j = \min\left\{2^j, f\right\}$. If $j \le \lceil \lg g \rceil$ then $b_j \le 2^j < 2g$, so

$$M(b_j + \lceil \lg 8k \rceil) \le (b_j + \lceil \lg 8k \rceil)\mu(2g + \lceil \lg 8k \rceil) \le (2^j + \lceil \lg 8k \rceil)\mu(2g + \lceil \lg 8k \rceil).$$

Now by Lemma 13.4 the M-time is at most

$$\begin{aligned} P(k) \sum_{0 \le j \le \lceil \lg g \rceil} M(b_j + \lceil \lg 8k \rceil) &\le P(k) \sum_{0 \le j \le \lceil \lg g \rceil} (2^j + \lceil \lg 8k \rceil)\mu(2g + \lceil \lg 8k \rceil) \\ &< P(k)(2^{\lceil \lg 2g \rceil} + \lceil \lg 2g \rceil \lceil \lg 8k \rceil)\mu(2g + \lceil \lg 8k \rceil) \\ &< P(k)(4g + \lceil \lg 2g \rceil \lceil \lg 8k \rceil)\mu(2g + \lceil \lg 8k \rceil) \end{aligned}$$

as claimed. $\square$

Notes. Our use of increasing precision is at the heart of our improvement over [3]. A 50-digit number that starts with 9876 is not an 11th power; we don't need to look at the remaining 46 digits to see this. In general we do not inspect many more bits of n than are necessary to distinguish n from x^k. As in Newton's method, the last step dominates the run time.

In Newton's method it is natural to double the precision at each step. But in Algorithm C we could use any vaguely geometric progression. In practice we should modify our b sequence to take into account the speed of multiplication and the distribution of x and n.

Lemma 7.3 is hopelessly pessimistic about the time needed to compute x^k to high accuracy. Since x has very few bits, our first few multiplications use relatively low precision. In fact, the $P(k)$ factor in Lemma 13.4 should disappear as g grows. Similarly, the bound from Lemma 7.2 is somewhat loose. A careful analysis would show that, when b is large, we can use fewer than $\lceil \lg 8k \rceil$ guard bits in step 2 of Algorithm C. This is probably not worth the added complexity in practice.

If we know n modulo a small prime q, we can compute $x \bmod q$ and check whether $x^k \bmod q = n \bmod q$. See section 31 for further discussion.

In step 3 of Algorithm C we compare a high-precision number, n, to a low-precision floating-point number, r. The alert reader may have observed that this is a potential bottleneck. The M-time in Algorithm C is essentially the precision of r; this may be much less time than it takes us to read the digits of n. Fortunately we can check whether $n < r$ in time proportional to the size of r, so there is no difficulty. (Similar comments apply to step 4.) Imagine if, instead, we had to test whether $n \le r$. If n and r agreed out to the precision of r, we would have to keep reading n to see whether any of the remaining bits were nonzero. There are many solutions—in practice n will always be odd; more generally our comparison routine could insist that n be represented as a fully denormalized floating-point number; or we could truncate n and use slightly wider bounds—but it is comforting that for us the problem does not arise.

14. How to test if n is a kth power

Let n and k be integers larger than 1. In this section we show how to check whether n is a kth power. The idea is to compute a floating-point approximation r to $n^{1/k}$; say $|n^{1/k} - r| < 1/4$. Then, if r is within $1/4$ of an integer x, we check whether $x^k = n$.

Algorithm K. Given integers $n \ge 2$ and $k \ge 2$, and a positive floating-point number y (see Lemma 14.2), we see if n is a kth power. In advance set $f = \lfloor \lg 2n \rfloor$ and $b = 3 + \lceil f/k \rceil$.
1. Compute $r \leftarrow \mathrm{nroot}_b(y, k)$.
2. Find an integer x with $|r - x| \le 5/8$.
3. If $x = 0$ or $|r - x| \ge 1/4$, print 0 and stop.
4. Compute the sign of $n - x^k$ with Algorithm C.
5. If $n = x^k$, print x and stop.
6. Print 0.

Lemma 14.3 shows that Algorithm K is correct. Lemma 14.4 gives an upper bound for its run time.

Lemma 14.1. *If* $0 \le t \le 1/10$ *then* $(1+t)^{1/2}/(1-t) \le 1+2t$ *and* $(1-t)^{1/2}/(1+t) \ge 1-2t$.

Proof. We have $1-3t-4t^2+4t^3 \ge 1-0.3-0.04-0.004 > 0$. Hence $t-3t^2-4t^3+4t^4$ is nonnegative. Thus $1+t \le 1+2t-3t^2-4t^3+4t^4 = (1+t-2t^2)^2$, so $\sqrt{1+t} \le (1-t)(1+2t)$. Also $0 < 1-t \le 1$ so $\sqrt{1-t} \ge 1-t \ge 1-t-2t^2 = (1+t)(1-2t)$. □

Lemma 14.2. *If* $y(1-2^{-b}) < n^{-1} < y(1+2^{-b})$ *in Algorithm K then* $\left|r - n^{1/k}\right| < 1/4$.

Proof. Write $\delta = 2^{-b}$. By hypothesis, $y(1-\delta) < n^{-1} < y(1+\delta)$, so $y^{-1/k}(1+\delta)^{-1/k} < n^{1/k} < y^{-1/k}(1-\delta)^{-1/k}$. By Lemma 11.7, $r(1-\delta) < y^{-1/k} < r(1+\delta)$.

Note that $\delta \le 1/16 < 1/10$. We apply Lemma 14.1 twice: first

$$\begin{aligned} r &< \frac{y^{-1/k}}{1-\delta} < n^{1/k}\frac{(1+\delta)^{1/k}}{1-\delta} \le n^{1/k}\frac{(1+\delta)^{1/2}}{1-\delta} = n^{1/k} + n^{1/k}\left(\frac{(1+\delta)^{1/2}}{1-\delta} - 1\right) \\ &\le n^{1/k} + n^{1/k}2\delta < n^{1/k} + 2^{f/k}2\delta = n^{1/k} + 2^{f/k}2^{-2-\lceil f/k\rceil} \le n^{1/k} + \frac{1}{4}; \end{aligned}$$

second

$$\begin{aligned} r &> \frac{y^{-1/k}}{1+\delta} > n^{1/k}\frac{(1-\delta)^{1/k}}{1+\delta} \ge n^{1/k}\frac{(1-\delta)^{1/2}}{1+\delta} = n^{1/k} + n^{1/k}\left(\frac{(1-\delta)^{1/2}}{1+\delta} - 1\right) \\ &\ge n^{1/k} - n^{1/k}2\delta > n^{1/k} - 2^{f/k}2\delta = n^{1/k} - 2^{f/k}2^{-2-\lceil f/k\rceil} \ge n^{1/k} - \frac{1}{4}. \end{aligned}$$

Hence $\left|r - n^{1/k}\right| < 1/4$. □

Lemma 14.3. *Set $f = \lfloor \lg 2n \rfloor$ and $b = 3 + \lceil f/k \rceil$. Assume that $y(1-2^{-b}) < n^{-1} < y(1+2^{-b})$. If n is a kth power, Algorithm K prints $n^{1/k}$. If n is not a kth power, Algorithm K prints* 0.

Proof. In Algorithm K we find an integer x with $|r - x| \le 5/8$. By Lemma 14.2, $\left|r - n^{1/k}\right| < 1/4$, so $\left|x - n^{1/k}\right| < 1/4 + 5/8 < 1$. If n is a kth power then x and $n^{1/k}$ are both integers so $x = n^{1/k}$. Then $|r - x| = \left|r - n^{1/k}\right| < 1/4$ and $x > 0$, so Algorithm K does not stop in step 3; in step 5 it prints x.

On the other hand, if n is not a kth power, then certainly $n \ne x^k$, so Algorithm K does not stop in step 5. So it prints 0. □

Let t be a real number such that $t - 1/2$ is not an integer. We write $\operatorname{round} t$ for the **nearest integer to** t: the unique integer i with $|i - t| < 1/2$.

Lemma 14.4. *Set $f = \lfloor \lg 2n \rfloor$, $g = \max\left\{1, f - d(n, (\operatorname{round} n^{1/k})^k)\right\}$, and $b = 3 + \lceil f/k \rceil$. Assume that $y(1-2^{-b}) < n^{-1} < y(1+2^{-b})$. Then Algorithm K uses M-time less than*

$$\begin{aligned} &\big(4g + \lceil \lg 2g \rceil \lceil \lg 8k \rceil\big)P(k)\mu(2g + \lceil \lg 8k \rceil) + \lceil \lg 4k \rceil (P(k)+1)M(\lceil \lg(66(2k+1)) \rceil) \\ &\qquad + \big(2\lfloor f/k \rfloor + \lfloor \lg f \rfloor \lceil 8 + \lg k \rceil\big)(P(k+1)+1)\mu(\lceil f/k \rceil + 9). \end{aligned}$$

Proof. Define $T = \lceil \lg 4k \rceil (P(k)+1)M(\lceil \lg(66(2k+1)) \rceil)$. Note that $b - \lceil \lg 2k \rceil = \lceil f/k \rceil - \lceil \lg k \rceil + 2 \le \lceil f/2 \rceil + 1 \le f$ since $f \ge 2$.

In Algorithm K we first compute $\operatorname{nroot}_b(y, k)$. If $b \le \lceil \lg 8k \rceil$ then, by Lemma 11.1, we use M-time at most T. If $b > \lceil \lg 8k \rceil$ then, by Lemma 11.4, we use M-time at most

$$\begin{aligned} &T + (P(k+1)+1)(2(b - \lceil \lg 8k \rceil) + \lceil 8 + \lg k \rceil \lceil \lg(b - \lceil \lg 2k \rceil) - 1 \rceil)\mu(b+6) \\ &\quad \le T + (2(\lceil f/k \rceil - 1) + \lceil 8 + \lg k \rceil \lceil \lg f - 1 \rceil)(P(k+1)+1)\mu(\lceil f/k \rceil + 9) \\ &\quad \le T + (2\lfloor f/k \rfloor + \lceil 8 + \lg k \rceil \lfloor \lg f \rfloor)(P(k+1)+1)\mu(\lceil f/k \rceil + 9). \end{aligned}$$

After computing $r = \operatorname{nroot}_b(y, k)$ we construct an integer x. We may invoke Algorithm C; if we do, then $|r - x| < 1/4$, and $\left|r - n^{1/k}\right| < 1/4$ by Lemma 14.2, so $\left|x - n^{1/k}\right| < 1/2$, so $x = \operatorname{round} n^{1/k}$. Finally, by Lemma 13.5, Algorithm C uses M-time at most $P(k)(4g + \lceil \lg 2g \rceil \lceil \lg 8k \rceil)\mu(2g + \lceil \lg 8k \rceil)$. □

Notes. Say we know n modulo a small prime q with $q \bmod k = 1$. Then we can check in advance whether $n^{(q-1)/k}$ equals 0 or 1 modulo q. If not, then n cannot be a kth power. The idea of [3] is to compute n modulo several small primes for each k, so as to quickly weed out most non-powers. See section 31 for further discussion.

15. How to test if n is a perfect power

To see whether n is a perfect power, we run through the primes $p \le \lg n$. For each p, we check whether n is a pth power. Lemma 15.2 shows that our algorithm is correct. For a time analysis see the next section.

Algorithm X. Given an integer $n \ge 2$, we attempt to decompose n as a perfect power. In advance set $f = \lfloor \lg 2n \rfloor$.
1. Compute $y \leftarrow \mathrm{nroot}_{3+\lceil f/2 \rceil}(n, 1)$.
2. For each prime number $p < f$:
3. Apply Algorithm K to (n, p, y); let x be the result.
4. If $x > 0$, print (x, p) and stop.
5. Print $(n, 1)$.

Lemma 15.1. *$y(1 - 2^{-3-\lceil f/p \rceil}) < n < y(1 + 2^{-3-\lceil f/p \rceil})$ in Algorithm X.*

Proof. $y(1 - 2^{-3-\lceil f/2 \rceil}) < n < y(1 + 2^{-3-\lceil f/2 \rceil})$ by Lemma 11.7; and $p \ge 2$. □

Lemma 15.2. *If n is a perfect power, Algorithm X prints a prime number p and a positive integer x such that $x^p = n$. If n is not a perfect power, Algorithm X prints $(n, 1)$.*

Proof. By Lemma 15.1, $y(1 - 2^{-3-\lceil f/p \rceil}) < n < y(1 + 2^{-3-\lceil f/p \rceil})$. If Algorithm X stops in step 4 then, by Lemma 14.3, $x^p = n$.

Conversely, if n is a perfect power then n is a pth power for some prime $p \le \lg n < f$. By Lemma 14.3, Algorithm X stops in step 4. □

Notes. The result of [3] is a perfect-power classification algorithm that runs in time $\log^3 n$; on average, under reasonable assumptions, it runs in time $\log^2 n / \log^2 \log n$.

The run time of Algorithm X is much better: it is essentially linear in $\log n$, if we use fast multiplication. The proof uses transcendental number theory. It is much easier to prove that the average run time is essentially linear. For further discussion see section 16.

Algorithm X is not new. It is stated in, for instance, [16, section 2.4]. But God is in the details: without good methods for computing $n^{1/k}$ and for checking whether $x^k = n$, Algorithm X is not so attractive. The authors of [16] go on to say that one can "save time" by adding a battery of tests to Algorithm X. Variants of the same algorithm are also dismissed in [10, page 38] ("This is clearly quite inefficient") and [3].

We observe that, by putting enough work into the subroutines, we have made Algorithm X quite fast—so fast, in fact, that typical modifications will *slow it down.*

To enumerate the primes $p < f$ we may use the Sieve of Eratosthenes. See [26] for faster methods. Note that we do not have to run through the primes in increasing order. We should, in principle, reorder the operations in Algorithm X, depending on the distribution of inputs, so that we detect perfect powers as soon as possible.

16. Introduction to $F(n)$

In this section we introduce a function $F(n)$. This function should be thought of as the difficulty of determining that n is not a pth power, summed over all possible prime exponents p. If we use fast multiplication, the run time of our perfect-power decomposition algorithm is, at worst, essentially linear in $F(n)$.

We define

$$F(n) = \sum_{2\le p\le \lg n} (\lg p) \max\left\{1, \lg n - d(n, (\operatorname{round} n^{1/p})^p)\right\}$$

for $n \ge 2$. Here round t means an integer within $1/2$ of t.

Informal comments: $F(n)$ has about $\lg n/\log\lg n$ terms. Each term has a minor $\lg p$ factor that reflects the effort spent computing pth powers. The main factor $\max\left\{1, \lg n - d(n, (\operatorname{round} n^{1/p})^p)\right\}$ says how many bits of n agree with a nearby pth power. If n is very close to a pth power then this factor is close to $\lg n$.

$F(n)$ is the subject of Part IV and Part V. In Part IV we give lower and upper bounds for F. Then we prove that the normal and average behaviors of F are comparable to the lower bound. See section 18 for a more complete summary.

In Part V we prove that $F(n)$ is bounded by $(\lg n)^{1+\epsilon(n)}$ for a certain function $\epsilon \in o(1)$. The approach is through the following application of transcendental number theory: *there cannot be many perfect powers in a short interval.* This means that there are not many perfect powers close to n, so not many of the main factors in $F(n)$ are near $\lg n$. Note that our exponent $1+\epsilon(n)$, albeit theoretically satisfying, is ridiculously large for any reasonable value of n.

Connection with Algorithm X. Our interest in $F(n)$ is based on the following lemma.

Lemma 16.1. *For $n \ge 2$, Algorithm X takes M-time at most*

$$(8F(n) + 6f\lfloor \lg 16f\rfloor^3)\mu(2f + \lfloor \lg 128f\rfloor)$$

where $f = \lfloor \lg 2n\rfloor$.

The reader may verify that Algorithm X does not use much non-M-time. Hence Algorithm X takes time essentially linear in $F(n) + \log n$, provided that we use a fast multiplication algorithm. Since $F(n)$ is essentially linear in $\log n$, the run time of Algorithm X is essentially linear in $\log n$.

Proof. Write $c = \lfloor \lg f\rfloor$. Note that $\lceil \lg f\rceil \le c+1$. Note also that $\sum_{2\le p<f} 1/p < c$.

In step 1 we compute $y = \operatorname{nroot}_{3+\lceil f/2\rceil}(n, 1)$. By Lemma 11.4 this takes M-time at most $2M(8) + 2(f + 8\lfloor \lg 4f\rfloor)\mu(\lceil f/2\rceil + 9)$.

In each iteration of step 3 we invoke Algorithm K for a prime number $p < f$. Write $g = g_p = \max\left\{1, f - d(n, (\operatorname{round} n^{1/p})^p)\right\}$. By Lemma 15.1, Lemma 14.4, and Lemma 7.1, Algorithm K takes M-time less than

$$\begin{aligned}
&\big(4g + \lceil \lg 2g\rceil\lceil \lg 8p\rceil\big)P(p)\mu(2g + \lceil \lg 8p\rceil) + \lceil \lg 4p\rceil\,(P(p)+1)M(\lceil \lg(66(2p+1))\rceil)\\
&\qquad + \big(2\lfloor f/p\rfloor + \lfloor \lg f\rfloor\lceil 8 + \lg p\rceil\big)(P(p+1)+1)\mu(\lceil f/p\rceil + 9)\\
&\quad < 8g\lfloor \lg p\rfloor\mu(2f + c + 4) + 4\lfloor f/p\rfloor(c+1)\mu(f+9)\\
&\qquad + 2(c+1)\left((c+2)(c+4) + (c+3)(c+9) + c(c+9)\right)\mu(2f + c + 7)\\
&\quad < \big(8g\lg p + 4(c+1)f/p + 6(c+1)(c^2 + 9c + 12)\big)\,\mu(2f + c + 7).
\end{aligned}$$

Our total M-time is then less than $\mu(2f + c + 7)$ times

$$\begin{aligned}
&2f + 16(c+3) + \sum_{2\le p\le f-1}\big(8g_p\lg p + 4(c+1)f/p + 6(c+1)(c^2 + 9c + 12)\big)\\
&\quad < 2f + 16(c+3) + 8F(n) + 4(c+1)fc + 6f(c+1)(c^2 + 9c + 12)\\
&\quad \le 8F(n) + 2f\left(1 + 4(c+3) + 2c(c+1) + 3(c+1)(c^2 + 9c + 12)\right)\\
&\quad < 8F(n) + 6f(c+4)^3
\end{aligned}$$

as claimed. □

Notes. As we will see in Part IV, $F(n)$ is generally not the dominant term in Lemma 16.1. The reader may be tempted to chop one or more $\lg f$ factors out of the other term by, for example, using various familiar functions from section 19, or focusing on particular multiplication speeds such as $\mu(b) = \lg 2b \lg\lg 4b$. However, several variants of Algorithm X appear in Part VI, and it is undoubtedly easier to achieve any desired run-time goal with one of those variants than with the original algorithm.

17. Proof of Theorem 1

In this section we combine all our results to prove Theorem 1: there is a perfect-power classification algorithm that uses time at most $(\lg n)^{1+o(1)}$ for $n > 2$.

Let $T(n)$ be an upper bound on the time taken by Algorithm X for $n \ge 2$. As discussed in section 16, we may take $T(n) \in (\lg n)^{1+o(1)}$ for $n > 2$, provided that we use fast multiplication.

We define $U(n) = \max\{T(m)/\lg m : m \le n\} \lg n$ for $n \ge 2$.

We have $T(m)/\lg m \in (\lg m)^{o(1)}$ for $m > 2$, so $U(n)/\lg n \in (\lg n)^{o(1)}$ for $n > 2$ by Lemma 4.1. Hence $U(n) \in (\lg n)^{1+o(1)}$ for $n > 2$.

To finish the proof we exhibit a perfect-power classification algorithm, Algorithm PPC, and prove that it runs in time $2U(n)$.

Algorithm PPC. Given $n \ge 2$ we print (x, k) such that (1) $x^k = n$ and (2) x is not a perfect power.
1. Apply Algorithm X to n; let (x, p) be the result.
2. If $(x, p) = (n, 1)$, print $(n, 1)$ and stop.
3. (Note that $2 \le x < n$.) Apply Algorithm PPC to x; let (c, k) be the result.
4. Print (c, kp).

Lemma 17.1. *If $n = x^p$ then $pU(x) \le U(n)$.*

Proof. $U(n)/\lg n$ is a nondecreasing function of n, so $pU(x)/\lg x \le pU(n)/\lg n = pU(n)/p\lg x = U(n)/\lg x$. □

Lemma 17.2. *Algorithm PPC spends time at most $2U(n)$ plus housekeeping.*

Proof. We spend time at most $T(n)$ in step 1. If n is a perfect power then we call Algorithm PPC recursively; by induction this takes time at most $2U(x)$. The total time is at most $T(n) + 2U(x) \le U(n) + pU(x) \le 2U(n)$ by Lemma 17.1. □

Part IV. Analytic methods

18. Summary of Part IV

In section 20 we consider the function $F(n)$ introduced in section 16. We explain why $F(n)$ is roughly $\lg n \lg\lg n$. Several functions arise naturally in our analysis of $F(n)$; we describe these functions first, in section 19.

In section 21 we study $F(n)$ in detail. We give a lower bound of about $\lg n \lg\lg n$. We show that the normal and average behaviors of $F(n)$ are also about $\lg n \lg\lg n$.

19. Some number-theoretic functions

Our function $F(n)$ is a sum over primes p. In this section we supply notation for three simpler sums: $\vartheta(t)$, $\vartheta_2(t)$, and $\ell(t)$. We break the problem of understanding $F(n)$ into (1) relating $F(n)$ to the functions $\vartheta, \vartheta_2, \ell$ and (2) understanding those functions. The next two sections address (1). We address (2) very briefly in this section.

Here are the functions:

$$\begin{aligned}
\vartheta(t) &= \sum_{2\le p\le t} \log p \approx t,\\
\vartheta_2(t) &= \sum_{2\le p\le t} \log^2 p \approx t\log t - t,\\
\ell(t) &= \sum_{2\le p\le t} \frac{\log p}{p} \approx \log t.
\end{aligned}$$

One can easily derive the approximations on the right by replacing $\sum_{p\le t} h(p)$ with $\sum_{k\le t} h(k)/\log k$; a "random" number k has a $1/\log k$ chance of being prime.

Notes. See [27] for bounds on ϑ.

20. Intuition about $F(n)$

In this section we give some motivation for the facts about $F(n)$ proved in the next section. Our theme here is that $F(n)$ is roughly $\lg n \lg\lg n$.

Define u_p as follows: n is $u_p p n^{1-1/p}$ away from the nearest pth power. Then u_p is, intuitively, a random number between 0 and $1/2$. Indeed, if n is randomly selected from the interval $[x^p, (x+1)^p]$, then its distance to the nearest endpoint ranges uniformly from 0 to $((x+1)^p - x^p)/2 \approx (1/2)px^{p-1} \approx (1/2)pn^{1-1/p}$.

These approximations break down when $x \approx n^{1/p}$ is smaller than p, so let's assume for the moment that p is at most $\lg n/\lg\lg n$.

Now the number of bits we need to distinguish n from the nearest pth power is about $\lg n - \lg u_p p n^{1-1/p} = (1/p)\lg n - \lg p - \lg u_p$. If u is a random number uniformly distributed between 0 and $1/2$ then the *average* value of $\lg u$ is

$$2\int_0^{1/2} \lg u\, du = \frac{2}{\log 2}\left(\frac{1}{2}\log\frac{1}{2} - \frac{1}{2}\right) = -1 - \frac{1}{\log 2}.$$

So our number of bits is, on average, about $(1/p)\lg n - \lg p + 1 + 1/\log 2$. Note that this is positive, since $p < n^{1/p}$.

As p grows past $\lg n/\lg\lg n$, on the other hand, the pth powers become so widely spaced that we usually need only a single bit of n.

Now we consider $F(n)$. $F(n)$ compares n with the pth power of the integer closest to $n^{1/p}$; this is usually the nearest pth power to n. So we estimate that, on average,

$$\begin{aligned}
F(n) &\approx \sum_{p \le \lg n/\lg\lg n} \left(\frac{\lg p}{p} \lg n - \lg^2 p + \left(1 + \frac{1}{\log 2}\right) \lg p \right) + \sum_{\lg n/\lg\lg n < p \le \lg n} \lg p \\
&\approx \frac{1}{\log^2 2} \left(\ell\left(\frac{\lg n}{\lg\lg n}\right) \log n - \vartheta_2\left(\frac{\lg n}{\lg\lg n}\right) + \vartheta\left(\frac{\lg n}{\lg\lg n}\right) \log 2e \right) + \frac{\vartheta(\lg n)}{\log 2} \\
&\approx \lg n \lg\lg n - \lg n \lg\lg\lg n + \frac{\lg n}{\lg\lg n} \frac{\log\lg\lg n + \log 2 + 2}{\log^2 2} \\
&\approx \lg n \lg\lg n.
\end{aligned}$$

What makes $F(n)$ difficult to analyze is that u_p is occasionally very close to 0. Then $-\lg u_p$ is much larger than its usual value. If this happens for a few primes p—as it does, for example, when $n \approx 32768$—then $F(n)$ will be noticeably larger than expected. We will get a *lower* bound on $F(n)$ by changing u_p to 1, but we cannot get an upper bound in any analogous way.

21. Analysis of $F(n)$

In this section we present various facts about

$$F(n) = \sum_{2 \le p \le \lg n} (\lg p) \max\left\{1, \lg n - d(n, (\operatorname{round} n^{1/p})^p)\right\},$$

in terms of the functions defined in section 19.

Lemma 21.1 gives a lower bound for $F(n)$, roughly $(\lg n)(\lg\lg n - \lg\lg\lg n - 1/\log 2)$. Lemma 21.2 gives a weak (quadratic) upper bound for $F(n)$. Lemma 21.4 gives a much better upper bound for the normal behavior of $F(n)$, roughly $(\lg n \lg\lg n)(1 + 2/\log 2)$. Lemma 21.5 gives a similar upper bound for the average behavior of $F(n)$, roughly $(\lg n)(2\lg\lg n + 12/\log 2)$.

In light of Lemma 16.1, these results translate directly into facts about the run time of our perfect-power decomposition algorithm, Algorithm X. For example, by Lemma 21.4 and Lemma 21.5, the normal and average run times of Algorithm X are within a factor of about $\lg^3(16 \lg n)$ of multiplication speed.

Lemma 21.1. *If $n \ge 4$ then*

$$F(n) > \frac{1}{\log 2} \ell\left(\frac{\lg n}{\lg\lg n}\right) \lg n - \frac{1}{\log^2 2} \vartheta_2\left(\frac{\lg n}{\lg\lg n}\right).$$

Proof. Note that $\lg\lg n \ge 1$. Fix $p \le \lg n/\lg\lg n$, so that $p \le \lg n \le n^{1/p}$. Set $x = \operatorname{round} n^{1/p}$.

If $n \ge x^p$ then

$$\begin{aligned}
n - x^p &= (n^{1/p} - x)(n^{1-1/p} + x n^{1-2/p} + \cdots + x^{p-1}) \\
&\le \frac{1}{2}(n^{1-1/p} + n^{1-1/p} + \cdots + n^{1-1/p}) = \frac{p}{2} n^{1-1/p}.
\end{aligned}$$

If $n < x^p$ then

$$\begin{aligned}
x^p - n &= (x - n^{1/p})(x^{p-1} + x^{p-2} n^{1/p} + \cdots + n^{1-1/p}) \le \frac{p}{2} x^{p-1} \\
&< \frac{p}{2}\left(n^{1/p} + \frac{1}{2}\right)^{p-1} < \frac{p}{2} n^{1-1/p} \left(1 + \frac{1}{2p}\right)^p < \frac{p}{2} n^{1-1/p} e^{1/2}
\end{aligned}$$

by Lemma 9.1.

Either way $|n - x^p| < pn^{1-1/p}$, so $d(n, x^p) < (1 - 1/p) \lg n + \lg p$. Hence

$$F(n) > \sum_{p \le \lg n/\lg\lg n} (\lg p) \left(\frac{1}{p} \lg n - \lg p \right) = \ell \left(\frac{\lg n}{\lg\lg n} \right) \frac{\lg n}{\log 2} - \vartheta_2 \left(\frac{\lg n}{\lg\lg n} \right) \frac{1}{\log^2 2}$$

as claimed. □

Lemma 21.2. *$F(n) \le \vartheta(\lg n) \lg n/\log 2$.*

Proof. $\lg n - d(n, (\operatorname{round} n^{1/p})^p) \le \lg n$, and $\sum_{2 \le p \le \lg n} \lg p = \vartheta(\lg n)/\log 2$. □

For the next two lemmas we say that n is **exceptional for** p if it is within $n^{1-1/p}/\lg^2 n$ of a pth power. We say that n is **exceptional** if it is exceptional for some prime $p \le \lg n$.

Lemma 21.3. *There are at most $2^{3+f}/(f-1) + 2^{2+f/2}(f-1)$ exceptional integers n in the interval $2^{f-1} \le n < 2^f$.*

So the exceptional integers have natural density 0.

Proof. Fix p. Set $T = 2^{f-f/p}/(f-1)^2$; if n is exceptional for p then n differs from some pth power by less than T.

Let S be the set of integers x between $\lceil 2^{(f-1)/p} - 1 \rceil$ and $\lfloor 2^{f/p} + 1 \rfloor$ inclusive. Say $|n - y^p| < T$ with n in our interval. We construct $x \in S$ such that $|n - x^p| < T$: if $y \in S$, set $x = y$; if $y > \lfloor 2^{f/p} + 1 \rfloor$, set $x = \lfloor 2^{f/p} + 1 \rfloor$; if $y < \lceil 2^{(f-1)/p} - 1 \rceil$, set $x = \lceil 2^{(f-1)/p} - 1 \rceil$.

There are $\lfloor 2^{f/p} \rfloor - \lceil 2^{(f-1)/p} \rceil + 3 < 2^{2+f/p}$ elements $x \in S$. Each x produces at most $2T + 1$ integers exceptional for p. Thus there are, in our interval, at most $2^{2+f/p}(2T + 1) \le 2^{3+f}/(f-1)^2 + 2^{2+f/2}$ integers exceptional for p.

There are at most $f-1$ primes p, so there are at most $2^{3+f}/(f-1)+2^{2+f/2}(f-1)$ exceptional integers in our interval. □

Lemma 21.4. *$F(n) < \ell(\lg n) \lg n/\log 2 + (2 \lg\lg n + 1)\, \vartheta(\lg n)/\log 2$ if n is not exceptional and $n \ge 4$.*

Proof. By hypothesis $d(n, x^p) > \lg n^{1-1/p} - 2 \lg\lg n - 1$ for any x and any $p \le \lg n$. So $\lg n - d(n, x^p) < (1/p) \lg n + 2 \lg\lg n + 1$. Thus

$$\begin{aligned} F(n) &< \sum_{p \le \lg n} (\lg p)((1/p) \lg n + 2 \lg\lg n + 1) \\ &= \sum_{p \le \lg n} \frac{\lg p}{p} \lg n + (2 \lg\lg n + 1) \lg p = \frac{\ell(\lg n)}{\log 2} \lg n + \frac{2 \lg\lg n + 1}{\log 2} \vartheta(\lg n); \end{aligned}$$

note that the sum is nonempty since $n \ge 4$. □

Lemma 21.5. *$2^{1-f} \sum_{2^{f-1} \le n < 2^f} F(n) \le (2/\log 2) f\, \ell(f-1) + (12/\log 2)\, \vartheta(f-1)$ for $f \ge 10$.*

Proof. First we fix $p < f$ and consider the sum

$$C = \sum_{2^{f-1} \le n < 2^f} \max \left\{ 1, f - d(n, (\operatorname{round} n^{1/p})^p) \right\}.$$

Let S be the set of values of $\operatorname{round} n^{1/p}$ for $2^{f-1} \le n < 2^f$. If $1 < n < 2^f$ then $1 < n^{1/p} < 2^{f/p} \le \lceil 2^{f/p} \rceil$ so $1 \le \operatorname{round} n^{1/p} \le \lceil 2^{f/p} \rceil$. Hence $\#S \le \lceil 2^{f/p} \rceil < 2^{f/p+1}$.

We return to our sum C. Write $g = \max\left\{1, f - d(n, (\operatorname{round} n^{1/p})^p)\right\}$; this is an integer between 1 and f inclusive. We analyze $C = \sum g$ by considering how often each possible g can appear.

If $g = f$ then $d(n, (\operatorname{round} n^{1/p})^p) = 0$, so n is within 1 of a pth power. There are at most $\#S$ relevant pth powers, so $g = f$ occurs at most $3\#S$ times.

If $2 \le g < f$ then $d(n, x^p) = f - g > 0$ where $x = \operatorname{round} n^{1/p} \in S$. Thus $2^{f-g} \le |n - x^p| < 2^{f-g+1}$. For each $x \in S$ there are at most 2^{f-g+1} integers n satisfying this condition. So g occurs at most $2^{f-g+1}\#S$ times.

We lump together all $g \le \lceil \lg 4\#S \rceil$, including $g = 1$. There are at most 2^{f-1} such terms, and we estimate each term as $\lceil \lg 4\#S \rceil$. Note that $\lceil \lg 4\#S \rceil \le \lceil f/p + 3 \rceil \le f/2 + 4 \le f - 1$.

Now

$$
\begin{aligned}
C &= \sum_{2^{f-1} \le n < 2^f} \max\left\{1, f - d(n, (\operatorname{round} n^{1/p})^p)\right\} \\
&\le 2^{f-1} \lceil \lg 4\#S \rceil + (3\#S)f + \sum_{\lceil \lg 4\#S \rceil + 1 \le g \le f-1} (2^{f-g+1}\#S)g \\
&= 2^{f-1} \lceil \lg 4\#S \rceil + (3\#S)f + \#S\left(2^{f - \lceil \lg 4\#S \rceil + 1}(\lceil \lg 4\#S \rceil + 2) - 4(f+1)\right) \\
&< 2^{f-1} \lceil \lg 4\#S \rceil + (\#S)2^{f-1-\lceil \lg \#S \rceil} \lceil \lg 16\#S \rceil \\
&< 2^{f-1} \lceil \lg 16\#S \rceil (1 + (\#S)2^{-\lceil \lg \#S \rceil}) \\
&< 2 \cdot 2^{f-1} \lceil \lg 16\#S \rceil < 2 \cdot 2^{f-1}(f/p + 6).
\end{aligned}
$$

Finally we analyze the sum of $F(n)$:

$$
\begin{aligned}
\sum_{2^{f-1} \le n < 2^f} F(n) &< \sum_{2^{f-1} \le n < 2^f} \sum_{2 \le p \le f-1} (\lg p) \max\left\{1, f - d(n, (\operatorname{round} n^{1/p})^p)\right\} \\
&= \sum_{2 \le p \le f-1} (\lg p) \sum_{2^{f-1} \le n < 2^f} \max\left\{1, f - d(n, (\operatorname{round} n^{1/p})^p)\right\} \\
&< \sum_{2 \le p \le f-1} (\lg p) \left(2 \cdot 2^{f-1}\right)(f/p + 6) \\
&= \frac{2}{\log 2} 2^{f-1} f\, \ell(f-1) + \frac{12}{\log 2} 2^{f-1}\, \vartheta(f-1)
\end{aligned}
$$

as stated. □

Notes. $F(n)$ usually behaves like $\lg n \lg\lg n$, but it behaves more like $2 \lg n \lg\lg n$ when n is a power of 2 with a sufficiently smooth exponent.

We ask whether $F(n)/\lg n \lg\lg n$ is bounded. If this is true it will not be easy to prove: we sweat and strain in Part V to prove merely $F(n) \in (\lg n)^{1+o(1)}$. But if it is false perhaps there is an easy disproof.

Part V. Transcendental methods

22. Summary of Part V

In sections 23 and 24 we quote special cases of two theorems from transcendental number theory.

In section 26 we use these theorems to prove that there cannot be many perfect powers in a short interval. Various necessary inequalities, mostly rather loose, appear in section 25.

In section 27 we complete our analysis of $F(n)$: it is essentially linear in $\log n$.

23. Multiplicative dependence

We say that $x_0, \dots, x_n$ are **multiplicatively dependent** if there are integers $a_0, \dots, a_n$, not all zero, with $x_0^{a_0} \cdots x_n^{a_n} = 1$. In this section we quote without proof a special case of a theorem of Loxton and van der Poorten on multiplicative dependence. We will use this theorem in section 26.

Lemma 23.1. *Let $x_0, \dots, x_n$ be multiplicatively dependent positive integers with $x_j \ge 3$. Then there are integers $a_0, \dots, a_n$, not all zero, with $x_0^{a_0} \cdots x_n^{a_n} = 1$, and*

$$|a_j| < 3n^n (\log x_0) \cdots (\log x_n).$$

Notes. Lemma 23.1 follows from [20, Theorem 5(A)], with $D = 1$, $w(\mathbf{Q}) = 2$, and $\lambda(1) = \log 2$; note that $1 < \log x_j$ and $2(n!/(\log 2)^n) < 3n^n$.

24. Linear forms in logarithms

A **linear form in logarithms** is an expression of the form $\beta_1 \log \alpha_1 + \cdots + \beta_n \log \alpha_n$, where α_j and β_j are algebraic numbers. In this section we quote without proof a special case of theorem of Loxton on linear forms in logarithms. We will use this theorem in section 26.

The **height** of a nonzero rational number α is $H(\alpha) = \max\{|i|, |j|\}$, if $\alpha = i/j$ in lowest terms. The height of 0 is 0.

Lemma 24.1. *Fix $c \ge 1$, $n \ge 1$. Let $\alpha_1, \dots, \alpha_n$ be multiplicatively independent positive rational numbers. Let*

$$\begin{pmatrix} \beta_{11} & \beta_{12} & \cdots & \beta_{1n} \\ \beta_{21} & \beta_{22} & \cdots & \beta_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{c1} & \beta_{c2} & \cdots & \beta_{cn} \end{pmatrix}$$

be a rank-c matrix of rational numbers. Fix $A_j \ge 4$ and $B \ge 4$ such that $H(\alpha_j) \le A_j$ and $H(\beta_{ij}) \le B$. Write $\Omega = (\log A_1) \cdots (\log A_n)$. Write

$$\begin{pmatrix} \Lambda_1 \\ \Lambda_2 \\ \vdots \\ \Lambda_c \end{pmatrix} = \begin{pmatrix} \beta_{11} & \beta_{12} & \cdots & \beta_{1n} \\ \beta_{21} & \beta_{22} & \cdots & \beta_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{c1} & \beta_{c2} & \cdots & \beta_{cn} \end{pmatrix} \begin{pmatrix} \log \alpha_1 \\ \log \alpha_2 \\ \vdots \\ \log \alpha_n \end{pmatrix}.$$

Then, for some i,

$$|\Lambda_i| > \exp(-(16n)^{200n} (\Omega \log \Omega)^{1/c} \log B\Omega).$$

Notes. A central theorem of Baker [4] states that a single nonzero linear form in logarithms cannot be exceedingly close to 0, or in fact to any algebraic number. Loxton's theorem [19, Theorem 4] generalizes Baker's theorem to handle several independent linear forms in the same set of logarithms. Lemma 24.1 follows from [19, Theorem 4] with $d = 1$.

The constants 16 and 200 here can easily be reduced.

25. More inequalities

Lemma 25.1. *If $x^k \in [L, U]$ and $x'^{k'} \in [L, U]$ then $|k \log x - k' \log x'| \le \log(U/L)$.*

Proof. Both $\log x^k$ and $\log x'^{k'}$ are between $\log L$ and $\log U$, so the difference is at most $\log U - \log L = \log(U/L)$. $\square$

Lemma 25.2. *For $u \ge 1000$ set $T = (1/10)\sqrt{u/\log 2.56u}$. Then $T > 1$, $4T + 2 < \sqrt{u}$, $200T \log 16T < u/T = 10\sqrt{u \log 2.56u}$, $6T < e^u$, and $T(7 + \lg T + u/\log 2 - \lg \log 2) < u^3$.*

This real number T is selected to balance $(16T)^{200T}$ with $e^{u/T}$.

Proof. The function $u/\log 2.56u$ increases when $\log 2.56u > 1$: its derivative is $(\log 2.56u - 1)/\log^2 2.56u$. Hence $T \ge (1/10)\sqrt{1000/\log 2560} > 1$. On the other hand $T < (1/10)\sqrt{u}$. Thus

$$200T \log 16T < 200T \log\left(1.6\sqrt{u}\right) = 100T \log 2.56u = 10\sqrt{u \log 2.56u} = u/T.$$

Also $4T + 2 < 6T < \sqrt{u} < e^u$. Finally $T(7 + \lg T + u/\log 2 - \lg \log 2) < u(7 + T + 2u + 1) < u(8 + 3u) < u^3$. $\square$

Lemma 25.3. *If $v \ge 1$ and $t \ge 5$ then*

$$\log(t^v + t^{v-1}) < -2v \log\log \frac{t+1}{t-1}.$$

Proof. The function $\log((t+1)/(t-1))$ is decreasing. In fact $(t+1)\log((t+1)/(t-1))$ is decreasing: its derivative is $-2/(t-1) + \log(1 + 2/(t-1))$, which is negative by Lemma 9.1. Hence $(t+1)\log^2((t+1)/(t-1)) \le 6\log^2(6/4) < 1$. So

$$\begin{aligned}\log(t^v + t^{v-1}) + 2v \log\log \frac{t+1}{t-1} &= v \log t + \log(1 + 1/t) + 2v \log\log \frac{t+1}{t-1} \\ &\le v\left(\log(t+1) + 2\log\log \frac{t+1}{t-1}\right) < 0\end{aligned}$$

as desired. $\square$

Lemma 25.4. *If $\log\log 16 \le t \le 1600$ then $t - \log\log 2 < 40\sqrt{t \log t}$.*

Proof. Define $h(t) = (t - \log\log 2)^2/t - 1600 \log t$. At $t = \log\log 16$ we have $h(t) < 0$. For $0 < t \le 1600$ we have $t^2 h'(t) = t(t - 1600) - (\log\log 2)^2 < 0$ so $h'(t) < 0$. Hence $h(t) < 0$ for $\log\log 16 \le t \le 1600$. $\square$

Lemma 25.5. *For $n \ge \exp\exp 1000$ write $t = \log\log n$ and $u = \log\log 2n$. Then $6u^3 \exp(30\sqrt{u \log 2.56u}) < \exp(40\sqrt{t \log t})$.*

Proof. We have $\log 2 < e^t(e^{0.1t} - 1)$ so $u = \log(e^t + \log 2) < 1.1t$. Also $3t < t^{1.2}$ so $u \log 2.56u < u \log 3t < 1.4t \log t$ so $30\sqrt{u \log 2.56u} < 36\sqrt{t \log t}$. Finally $6 < u^3 < 3t^3 < t^{3.2} = \exp(3.2 \log t) < \exp(\sqrt{t \log t})$. $\square$

26. Powers in short intervals

In this section we use the theorems stated in the previous three sections to show that a short interval $[L, U]$ cannot contain many perfect powers. (These results are due primarily to John Loxton. See the notes at the end of the section.)

What we really count is the number of exponents k such that there is a kth power in $[L, U]$. Lemma 26.2 is our workhorse: it says that there can be very few "large" exponents k. Lemma 26.4 gives an upper bound for the number of *prime* exponents k. We will use this in section 27. Corollary 26.5, included here for historical reasons, counts the number of perfect powers in $[L, U]$ when $U = L + \sqrt{L}$.

Lemma 26.1. *The matrix*

$$\begin{pmatrix} k_1 + ta_1 & ta_2 & \cdots & ta_m \\ ta_1 & k_2 + ta_2 & \cdots & ta_m \\ \vdots & \vdots & \ddots & \vdots \\ ta_1 & ta_2 & \cdots & k_m + ta_m \end{pmatrix}$$

has determinant $k_1 \cdots k_m(1 + ta_1/k_1 + ta_2/k_2 + \cdots + ta_m/k_m)$ *for* $k_1, \ldots, k_m \neq 0$.

Proof. Subtract the first row from all succeeding rows; divide column i by k_i; add each column to the first column. The resulting matrix is upper triangular, with $1+ta_1/k_1+ta_2/k_2+\cdots+ta_m/k_m$ in the top left and 1 elsewhere on the diagonal. □

Lemma 26.2. *Fix an interval* $[L, U]$ *with* $1 < U/e < L < U$. *Fix an integer* $C \geq 1$. *Fix* $K \geq 4$ *such that*

$$K \geq (16C)^{200C} \frac{\log U}{-\log\log(U/L)} (\log U)^{1/C}((C+1)\log\log U)^2$$

and

$$K \geq (16C)^{200C} \frac{\log U}{-\log\log(U/L)} (\log 6C^C + (2C+1)\log\log U)^2.$$

Let S *be a set of integer pairs* (x, k) *with* $x^k \in [L, U]$, $x \geq 4$, $k \geq K$. *Assume that* k *and* k' *are coprime whenever* (x, k) *and* (x', k') *are two distinct pairs in* S. *Then* $\#S \leq C + \lg C! + C \lg\lg U$.

Note that $\log U > 1$ and $-\log\log(U/L) > 0$.

Proof. This is a long proof, so we begin with an outline.

We say that $(x_1, k_1), \ldots, (x_m, k_m)$ are multiplicatively dependent if $x_1, \ldots, x_m$ are multiplicatively dependent. Fix a maximal multiplicatively independent subset $(x_1, k_1), \ldots, (x_m, k_m)$ of S. First we show that $m \leq C$. Then, as preparation, we construct a certain small nonzero integer, $\det Q$. If (x_0, k_0) is any element of S, there is a relation $x_0^{a_0} \cdots x_m^{a_m} = 1$, with each a_j reasonably small. We show that $a_0/k_0 + \cdots + a_m/k_m = 0$. So if (x_0, k_0) is different from $(x_1, k_1), \ldots, (x_m, k_m)$ then k_0 divides a_0, which in turn implies that k_0 divides $\det Q$. Hence the number of pairs is at most m plus the maximum number of coprime divisors of $\det Q$.

Step 1. Let $(x_1, k_1), \ldots, (x_m, k_m) \in S$ be multiplicatively independent. We show that $m \leq C$.

Suppose not: suppose we have $m \geq C + 1$ multiplicatively independent pairs $(x_1, k_1), \ldots, (x_m, k_m) \in S$. Then, in particular, $x_1, \ldots, x_{C+1}$ are multiplicatively independent. Put $B = \max\{k_j : 1 \le j \le C+1\}$ and $\Omega = \prod_{1\le i\le C+1} \log x_j$. Notice that

$$B\Omega \le \prod_{1\le i\le C+1} k_j \log x_j \le (\log U)^{C+1}.$$

Now

$$\begin{pmatrix} k_1 \log x_1 - k_{C+1}\log x_{C+1} \\ k_2 \log x_2 - k_{C+1}\log x_{C+1} \\ \vdots \\ k_C \log x_C - k_{C+1}\log x_{C+1} \end{pmatrix} = \begin{pmatrix} k_1 & 0 & \cdots & 0 & -k_{C+1} \\ 0 & k_2 & \cdots & 0 & -k_{C+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & k_C & -k_{C+1} \end{pmatrix} \begin{pmatrix} \log x_1 \\ \vdots \\ \log x_{C+1} \end{pmatrix}.$$

The conditions of Lemma 24.1 are met: each x_j is a positive integer; the matrix has rank C; $x_j \ge 4$ and $B \ge K \ge 4$; $H(x_j) = x_j$; $H(0) = 0 \le B$; and $H(-k_j) = H(k_j) = k_j \le B$. Hence, for some i,

$$|k_i \log x_i - k_{C+1}\log x_{C+1}| > \exp(-(16C)^{200C}(\Omega\log\Omega)^{1/C}\log B\Omega).$$

Apply Lemma 25.1 and take logarithms:

$$\log\log(U/L) > -(16C)^{200C}(\Omega\log\Omega)^{1/C}\log B\Omega.$$

Hence

$$\begin{aligned} K(-\log\log(U/L)) &< (16C)^{200C} K\Omega^{1/C}(\log\Omega)^{1/C}\log B\Omega \\ &< (16C)^{200C} K^{1+1/C}\Omega^{1/C}(\log B\Omega)^{1/C}\log B\Omega \\ &= (16C)^{200C}\left(K^{C+1}\prod_{1\le i\le C+1}\log x_i\right)^{1/C}(\log B\Omega)^{1+1/C} \\ &\le (16C)^{200C}\left(\prod_{1\le i\le C+1} k_i\log x_i\right)^{1/C}(\log B\Omega)^2 \\ &\le (16C)^{200C}\left(\prod_{1\le i\le C+1}\log U\right)^{1/C}((C+1)\log\log U)^2 \\ &= (16C)^{200C}(\log U)^{1+1/C}((C+1)\log\log U)^2 \\ &\le K(-\log\log(U/L)). \end{aligned}$$

Contradiction.

Step 2. Now fix a *maximal* multiplicatively independent subset of S, say $(x_1,k_1),(x_2,k_2),\dots,(x_m,k_m)$. We know $m \le C$. We may assume $m \ge 1$, since otherwise S is empty and we are done.

We construct a matrix as follows. Consider the primes q dividing $x_1x_2\cdots x_m$. Our matrix has one row for each q, namely $\operatorname{ord}_q x_1, \operatorname{ord}_q x_2, \dots, \operatorname{ord}_q x_m$.

The m columns of this matrix are independent. Indeed, if $a_1, a_2, \dots, a_m$ are integers such that $a_1 \operatorname{ord}_q x_1 + \cdots + a_m \operatorname{ord}_q x_m = 0$ for every q, then $\operatorname{ord}_q \prod_j x_j^{a_j} = 0$, so $\prod_j x_j^{a_j} = 1$. The x_j's are independent, so every a_j must be 0.

Hence the matrix has m independent rows. Fix $q_1, q_2, \ldots, q_m$ such that the corresponding rows are independent. Write

$$Q = \begin{pmatrix} \operatorname{ord}_{q_1} x_1 & \operatorname{ord}_{q_1} x_2 & \cdots & \operatorname{ord}_{q_1} x_m \\ \operatorname{ord}_{q_2} x_1 & \operatorname{ord}_{q_2} x_2 & \cdots & \operatorname{ord}_{q_2} x_m \\ \vdots & \vdots & \ddots & \vdots \\ \operatorname{ord}_{q_m} x_1 & \operatorname{ord}_{q_m} x_2 & \cdots & \operatorname{ord}_{q_m} x_m \end{pmatrix}$$

for the matrix formed from these rows.

By construction $\det Q$ is a nonzero integer. Each entry of Q is bounded by $\lg U$, so $|\det Q| \le m!(\lg U)^m \le C!(\lg U)^C$.

Step 3. Now we show that, for any $(x, k) \in S$ other than $(x_1, k_1), \ldots, (x_m, k_m)$, k must divide $\det Q$.

Let (x_0, k_0) be any element of S different from $(x_1, k_1), \ldots, (x_m, k_m)$. Then k_0 is coprime to $k_1, \ldots, k_m$, by hypothesis on S.

Since $(x_1, k_1), \ldots, (x_m, k_m)$ is a maximal multiplicatively independent subset of S, $x_0, x_1, \ldots, x_m$ must be multiplicatively dependent. The hypotheses of Lemma 23.1 are satisfied: each x_j is a positive integer larger than 3. Hence there are integers $a_0, \ldots, a_m$, not all zero, with $x_0^{a_0} \cdots x_m^{a_m} = 1$, and

$$|a_j| < 3m^m (\log x_0) \cdots (\log x_m).$$

Since $x_1, \ldots, x_m$ are independent, a_0 must be nonzero. We may assume that $\gcd\{a_0, \ldots, a_m\} = 1$: if not, divide each a_j by the common gcd.

Suppose that $a_0/k_0 + a_1/k_1 + \cdots + a_m/k_m \neq 0$. Consider the matrix

$$\Theta = \begin{pmatrix} k_1 + k_0 a_1/a_0 & k_0 a_2/a_0 & \cdots & k_0 a_m/a_0 \\ k_0 a_1/a_0 & k_2 + k_0 a_2/a_0 & \cdots & k_0 a_m/a_0 \\ \vdots & \vdots & \ddots & \vdots \\ k_0 a_1/a_0 & k_0 a_2/a_0 & \cdots & k_m + k_0 a_m/a_0 \end{pmatrix}.$$

By Lemma 26.1, Θ has determinant

$$\frac{k_0 k_1 \cdots k_m}{a_0} \left(\frac{a_0}{k_0} + \frac{a_1}{k_1} + \cdots + \frac{a_m}{k_m} \right),$$

which is nonzero. Hence Θ has rank m.

Next observe that

$$\begin{pmatrix} k_1 \log x_1 - k_0 \log x_0 \\ k_2 \log x_2 - k_0 \log x_0 \\ \vdots \\ k_m \log x_m - k_0 \log x_0 \end{pmatrix} = \Theta \begin{pmatrix} \log x_1 \\ \vdots \\ \log x_m \end{pmatrix}.$$

Indeed,

$$\begin{aligned} k_i \log x_i - k_0 \log x_0 &= k_i \log x_i + \frac{k_0}{a_0}(-a_0 \log x_0) \\ &= k_i \log x_i + \frac{k_0}{a_0}(a_1 \log x_1 + \cdots + a_m \log x_m). \end{aligned}$$

Put $B = 6m^m(\log x_0)\cdots(\log x_m)\max\{k_j : 0 \le j \le m\}$ and $\Omega = \prod_{1\le i\le m}\log x_i$. Notice that

$$B\Omega \le 6m^m(\log U)^{2m+1} \le 6C^C(\log U)^{2C+1}.$$

Again the conditions of Lemma 24.1 are met: each x_j is a positive integer; the matrix has rank m; $x_j \ge 4$ and $B \ge 4$; and the matrix entries have height at most B. Hence, for some i, $|k_i \log x_i - k_0 \log x_0| \ge \exp(-(16m)^{200m}(\Omega\log\Omega)^{1/m}\log B\Omega)$. Apply Lemma 25.1 and take logarithms:

$$\log\log(U/L) > -(16m)^{200m}(\Omega\log\Omega)^{1/m}\log B\Omega.$$

So

$$\begin{aligned}
K(-\log\log(U/L)) &< (16m)^{200m}K\Omega^{1/m}(\log\Omega)^{1/m}\log B\Omega \\
&< (16m)^{200m}\left(K^m\prod_{1\le i\le m}\log x_i\right)^{1/m}(\log B\Omega)^2 \\
&\le (16C)^{200C}\left(\prod_{1\le i\le m}k_i\log x_i\right)^{1/m}(\log B\Omega)^2 \\
&\le (16C)^{200C}(\log U)(\log 6C^C + (2C+1)\log\log U)^2 \\
&\le K(-\log\log(U/L)).
\end{aligned}$$

Contradiction.

Hence $a_0/k_0 + a_1/k_1 + \cdots + a_m/k_m = 0$. But k_0 is coprime to $k_1,\ldots,k_m$, so k_0 must divide a_0.

Consider the column vector $V = (a_1, a_2, \ldots, a_m)$. Since $x_1^{a_1}x_2^{a_2}\cdots x_m^{a_m} = x_0^{-a_0}$ we have, for any prime q,

$$a_1\operatorname{ord}_q x_1 + a_2\operatorname{ord}_q x_2 + \cdots + a_m\operatorname{ord}_q x_m = -a_0\operatorname{ord}_q x_0.$$

In other words a_0 divides QV, so a_0 divides $(\operatorname{adj} Q)QV = (\det Q)V$, so a_0 divides $(\det Q)a_j$ for each j. But $\gcd\{a_1, a_2, \ldots, a_m\} = 1$, so a_0 must divide $\det Q$. Hence k_0 divides $\det Q$.

Step 4. We finish the proof as follows. For every pair $(x,k) \in S$, other than $(x_1,k_1),\ldots,(x_m,k_m)$, we have shown that k divides $\det Q \ne 0$. Different pairs have coprime k's, by hypothesis, so $\det Q$ is divisible by the product of all those k's. Each k is at least 2. Hence there are no more than $\lg|\det Q| \le \lg C! + C\lg\lg U$ pairs (x,k) other than $(x_1,k_1),\ldots,(x_m,k_m)$. Finally $m \le C$. □

Lemma 26.3. *Fix an interval $[L,U]$ with $U/e < L < U$ and $U \ge \exp\exp 1000$. Let S be the set of primes k such that there is a kth power in $[L,U]$. Then*

$$\#S < (\log\log U)^3\left(1 + \frac{\log U}{-\log\log(U/L)}\exp\left(30\sqrt{\log\log U\log(2.56\log\log U)}\right)\right).$$

Proof. Define $u = \log\log U$, $T = (1/10)\sqrt{u/\log 2.56u}$, and $C = \lfloor T\rfloor$.

We apply each piece of Lemma 25.2. First $T > 1$ so $C \ge 1$ so $T < C + 1 \le 2C$. Hence

$$\begin{aligned}(16C)^{200C}(\log U)^{1/C} &< (16T)^{200T}(\log U)^{2/T}\\ &= \exp\left(200T\log 16T + \frac{2u}{T}\right) < \exp(30\sqrt{u\log 2.56u}).\end{aligned}$$

Furthermore $6+C+\lg C!+C\lg\lg U \le C(7+\lg C+\lg\lg U) \le T(7+\lg T+\lg\lg U) = T(7+\lg T+u/\log 2-\lg\log 2) < u^3$.

Set

$$\begin{aligned}K &= 4 + (16C)^{200C}\frac{\log U}{-\log\log(U/L)}(\log U)^{1/C}u^3\\ &< 4 + \frac{\log U}{-\log\log(U/L)}u^3\exp(30\sqrt{u\log 2.56u}).\end{aligned}$$

We have $(C+1)u < (2T+1)u < u^{3/2}$ so

$$K > (16C)^{200C}\frac{\log U}{-\log\log(U/L)}(\log U)^{1/C}((C+1)u)^2.$$

Furthermore,

$$\log 6C^C \le C\log 6C < (2C+1)\log 6T < (2C+1)u \le (2T+1)u < (1/2)u^{3/2},$$

so

$$\begin{aligned}K &> (16C)^{200C}\frac{\log U}{-\log\log(U/L)}(\log U)^{1/C}(\log 6C^C + (2C+1)u)^2\\ &\ge (16C)^{200C}\frac{\log U}{-\log\log(U/L)}(\log 6C^C + (2C+1)u)^2.\end{aligned}$$

Finally we count the primes $k \in S$. There are at most $K-1$ primes $k < K$. For each $k \ge K$ select an integer x such that $x^k \in [L,U]$. Consider the pairs (x,k). By Lemma 26.2, there are at most $C + \lg C! + C\lg\lg U$ pairs with $x \ge 4$. Since $U/L < 3$, there is at most one power of 3 in $[L,U]$, and at most two powers of 2. Hence

$$\begin{aligned}\#S &\le K + 2 + C + \lg C! + C\lg\lg U\\ &< u^3\left(1 + \frac{\log U}{-\log\log(U/L)}\exp(30\sqrt{\log\log U\log 2.56u})\right)\end{aligned}$$

as desired. □

Lemma 26.4. *Fix $n \ge \exp\exp 1000$. Set $u = \log\log 2n$. Fix v with $1 \le v \le \log_5 n$. Let S be the set of primes k such that there is a kth power in the interval $[n - n^{1-1/v}, n + n^{1-1/v}]$. Then*

$$\#S < 3vu^3\exp(30\sqrt{u\log 2.56u}).$$

Proof. Set $L = n - n^{1-1/v}$ and $U = n + n^{1-1/v} < 2n$. By Lemma 25.3, $\log U < -2v\log\log(U/L)$. Also $U > n \ge \exp\exp 1000$, and $U/L \le (1+1/5)/(1-1/5) < e$, so by Lemma 26.3

$$\#S < (\log\log U)^3\left(1 + 2v\exp\left(30\sqrt{\log\log U\log(2.56\log\log U)}\right)\right).$$

Finally $\log\log U < u$. □

Corollary 26.5. *The interval* $[n, n+\sqrt{n}]$ *contains fewer than*

$$\exp(40\sqrt{\log\log n \log\log\log n})$$

perfect powers for $n \geq 16$.

Proof. Let S be the set of primes p such that there is a pth power in $I = [n, n+\sqrt{n}]$. Each perfect power in I is a pth power for some prime p. On the other hand, I is too short to contain two pth powers: if $x^p \geq n$ then $(x+1)^p \geq x^p + px^{p-1} \geq n + pn^{1-1/p} > n + \sqrt{n}$. Hence the number of perfect powers in I is at most the size of S.

Write $t = \log\log n$. We will show that $\#S < \exp(40\sqrt{t \log t})$. For $t < 1000$ this is easy. If $p \in S$ then $p \leq \lg(n+\sqrt{n}) < \lg 2n$. So $\#S < \lg n = \exp(t - \log\log 2) < \exp(40\sqrt{t\log t})$ by Lemma 25.4.

For $t \geq 1000$ we apply Lemma 26.4 with $v = 2$. Set $u = \log\log 2n$. Then $\#S < 6u^3 \exp\left(30\sqrt{u \log 2.56u}\right)$. Finally, by Lemma 25.5, $\#S < \exp(40\sqrt{t\log t})$. $\square$

Notes. Corollary 26.5 was stated in [19, Theorem 1]. There is a gap in the proof in [19]: it incorrectly assumes that, in our notation, $a_0/k_0 + a_1/k_1 + \cdots + a_m/k_m \neq 0$. (Note that "$-a_{m+1}b_j/b_{m+1}$" in [19] was a typo for "$+a_{m+1}b_j/b_{m+1}$.")

John Loxton graciously supplied a corrected proof to this author. The idea of the correction is expressed above in Step 2 and Step 4 of Lemma 26.2. Other than this, the approach here is the same as the approach of [19, Theorem 1], modified slightly to handle more general intervals $[L, U]$.

The conclusion of Lemma 26.2 could easily be improved. Each column of our matrix Q has sum at most $(\lg U)/K$. From this one can prove with Hadamard's inequality [14, exercise 4.6.1–15] or with Gershgorin's inequality—see [12, problem 6.1–3]—that the determinant of Q is at most $((\lg U)/K)^m$.

In general the bounds in this section are very far from best possible. A more careful study would produce many quantitative improvements and perhaps some qualitative improvements.

Let S be the set of exponents k such that there is a kth power in $[L, U]$. We could prove a bound on the size of S as follows. Lemma 26.3 gives us a bound—call it m—on the number of primes in S. Every $k \in S$ is built up from those primes. Hence the size of S is at most the number of products $\leq \lg U$ of those primes, which is at most the number of products $\leq \lg U$ of the first m prime numbers, which in turn can be estimated by analytic techniques. See [27] and [9].

27. Final $F(n)$ analysis

In this section we use Lemma 26.4 to prove an upper bound for the function $F(n) = \sum_{2 \leq p \leq \lg n} (\lg p) \max\left\{1, \lg n - d(n, (\operatorname{round} n^{1/p})^p)\right\}$ introduced in section 16.

Lemma 27.1. *Fix* $n \geq \exp\exp 1000$. *Set* $u = \log\log 2n$. *Then*

$$F(n) < (\lg n \lg\lg n)\left(1 + 3u^3 \exp(30\sqrt{u\log 2.56u}) \lg(4\lg n)\right).$$

This upper bound is in $(\lg n)^{1+o(1)}$.

Proof. Write $g(p) = (\operatorname{round} n^{1/p})^p$. Also abbreviate $K = 3u^3 \exp(30\sqrt{u \log 2.56u})$.

The critical idea here is to sort our primes p by $d(n, g(p))$. Let $c < \lg n$ be the number of primes between 2 and $\lg n$. Let $p_1, p_2, \ldots, p_c$ be the primes, in such an order that $d(n, g(p_j))$ is a nondecreasing function of j.

Now $F(n) = \sum_{1 \le j \le c} (\lg p_j) \max\{1, \lg n - d(n, g(p_j))\}$. We estimate this sum in two pieces: first where $1 \le j < K$, second where $K \le j \le c$.

There are fewer than K terms in the first piece, and each term is less than $\lg n \lg \lg n$, so the sum of the terms in the first piece is less than $K \lg n \lg \lg n$.

In the second piece, set $v = j/K \ge 1$. We have $j \le c < \lg n$ and $K > 3$ so $v < \log_5 n$.

Suppose that $\lg n - d(n, g(p_j)) > 1 + (1/v) \lg n$. Then

$$|n - g(p_i)| < 2^{d(n,g(p_i))+1} \le 2^{d(n,g(p_j))+1} < 2^{(1-1/v)\lg n} = n^{1-1/v}$$

for all $i \le j$. So there is a p_ith power within $n^{1-1/v}$ of n for $1 \le i \le j$. But that is impossible, since by Lemma 26.4 there are fewer than $Kv = j$ primes p with a pth power so close to n.

Hence $\lg n - d(n, g(p_j)) \le 1 + (1/v) \lg n$. So the sum of this piece is at most $\sum_{K \le j \le c} (\lg \lg n)(1 + (K/j) \lg n) < (\lg \lg n) \sum_{1 \le j \le c} (1 + (K/j) \lg n) < c \lg \lg n + K \lg n \lg \lg n \lg 2c$. $\square$

Notes. Various constants here can of course be improved.

Part VI. Practical improvements

28. Summary of Part VI

In section 30 we present 2-adic methods to compute a tentative kth root x of n, and to check whether $x^k = n$. Section 29 explains how the 2-adic methods differ from the methods in Part II and Part III.

We may compute $n \bmod q$ (and $x \bmod q$) for one or more primes q. In section 31 we discuss several ways to take advantage of this information.

The word "improvements" in the title of Part VI should be understood to refer to *some* of the techniques here, not necessarily *all* of them.

29. Advertisement for the 2-adic variant

In the next section we describe a 2-*adic variant* of Algorithm X. With this variant, we can work with integers rather than floating-point numbers; we no longer need guard bits; we can jump directly into Newton's method without a preliminary binary search; and a proper error analysis takes a few lines rather than several pages. We do not use any 2-adic machinery in our presentation.

Let's review how we check if n is a kth power. First we compute a tentative kth root of n—an integer x such that no integer other than x can possibly be the kth root of n. Then we test whether $x^k = n$.

To find a tentative kth root we find a number that is *close* to a kth root of n in the real numbers $\mathbf{R}$. Here we measure closeness with $\mathbf{R}$'s usual metric.

We use the real metric again when we test whether $x^k = n$. We compute x^k in low precision and see whether it is close to n; we increase the precision and repeat the test until we are sure about the sign of $x^k - n$.

Nothing in the original problem demands that we use the metric obtained from $\mathbf{R}$. There are other metrics on the integers. In particular, the q-adic integers $\mathbf{Z}_q$ supply a metric where i and j are close if $i-j$ is divisible by a high power of q.

In the next section we explain (1) how to use the 2-adic metric in constructing a tentative kth root of n and (2) how to use the 2-adic metric in checking a tentative kth root of n. We avoid further references to $\mathbf{Z}_2$ or the 2-adic metric; using the 2-adic metric comes down to working modulo 2^m for an increasing sequence of m.

30. The 2-adic variant

In this section we describe our 2-adic variant of Algorithm X. We refer back to Part II and Part III for detailed explanations of our approach; this variant works along the same lines, with the modifications explained in section 29.

It will be convenient to restrict attention to odd n. See section 31 for a method to handle even n.

Notation. In this section we deviate from the notation of Parts I, II, and III: we let r, y, and z denote odd integers rather than positive floating-point numbers.

We write $i \bmod j$ for $i - j\lfloor i/j \rfloor$, the remainder when i is divided by j. When j is a power of 2 we may "read off" $i \bmod j$ as the bottom $\lg j$ bits of i's binary expansion.

We write $i \equiv i' \pmod j$ when $i \bmod j = i' \bmod j$.

Lemma 30.1. *If* $2i \equiv 2j \pmod{2^{b+1}}$ *and* $b \ge 1$ *then* $i^2 \equiv j^2 \pmod{2^{b+1}}$.

Proof. Either $i \equiv j$ or $i \equiv j + 2^b$. In the first case $i^2 \equiv j^2$. In the second case $i^2 \equiv (j+2^b)^2 = j^2 + 2^{b+1}j + 2^{2b} \equiv j^2$. $\square$

2-adic approximate powers. Fix positive integers k and b. For any integer m define $\operatorname{pow}_{2,b}(m,k) = m^k \bmod 2^b$. See section 7 for methods of computing $\operatorname{pow}_{2,b}(m,k)$ without many multiplications.

Note that, as we compute $\operatorname{pow}_{2,b}(m,k)$, we may keep track of an "overflow bit" and thus figure out whether $m^k \bmod 2^b = m^k$.

Checking tentative kth roots. Here is a straightforward algorithm for checking tentative kth roots.

Algorithm C2. Given positive integers n, x, k, we see if $n = x^k$. In advance set $f = \lfloor \lg 2n \rfloor$.
1. If $x = 1$: Print 0 if $n = 1$, 2 otherwise. Stop.
2. Set $b \leftarrow 1$.
3. Compute $r \leftarrow \operatorname{pow}_{2,b}(x,k)$. Simultaneously figure out if $r = x^k$.
4. If $n \bmod 2^b \ne r$, print 2 and stop.
5. If $b \ge f$: Print 0 if $r = x^k$, 2 otherwise. Stop.
6. Set $b \leftarrow \min\{2b, f\}$. Go back to step 3.

Lemma 30.2. *Algorithm C2 prints* 0 *if and only if* $n = x^k$.

Proof. If $n = x^k$ then $r = \operatorname{pow}_{2,b}(x,k) = x^k \bmod 2^b = n \bmod 2^b$ so we never stop in step 4. Hence we stop in step 5. When we do, $b \ge f$, so $r = n \bmod 2^f = n = x^k$. Thus we print 0.

Conversely, say we print 0. Then $x^k = r = n \bmod 2^f = n$. $\square$

2-adic approximate multiplication and division. Fix $b \ge 1$. If m is an integer and k is a positive integer we write $\mathrm{mul}_{2,b}(m,k) = km \bmod 2^b$.

If k is odd we write $\mathrm{div}_{2,b}(m,k)$ for the unique integer between 0 inclusive and 2^b exclusive such that $m \equiv k\,\mathrm{div}_{2,b}(m,k) \pmod{2^b}$.

Finding 2-adic approximate kth roots. Fix an odd integer y and a positive odd integer k. We compute an approximate negative kth root of y by Newton's method. For motivation see section 11. (Question: Why do we insist that k be odd? Answer: Square roots introduce a bit of difficulty. See Algorithm S2 below.)

For each $b \ge 1$ we define and construct an odd integer $\mathrm{nroot}_{2,b}(y,k)$, between 0 and 2^b, by the following algorithm:

Algorithm N2. Given an odd integer y and positive integers b, k with k odd, we compute $\mathrm{nroot}_{2,b}(y,k)$. In advance set $b' = \lceil b/2 \rceil$.
1. If $b = 1$: $\mathrm{nroot}_{2,b}(y,k) = 1$. Stop.
2. Compute $z \leftarrow \mathrm{nroot}_{2,b'}(y,k)$ by Algorithm N2.
3. Set $r_2 \leftarrow \mathrm{mul}_{2,b}(z, k+1)$.
4. Set $r_3 \leftarrow y\,\mathrm{pow}_{2,b}(z, k+1) \bmod 2^b$.
5. Set $r_4 \leftarrow \mathrm{div}_{2,b}(r_2 - r_3, k)$. Now $\mathrm{nroot}_{2,b}(y,k) = r_4$.

Lemma 30.3. *If k is odd and $r = \mathrm{nroot}_{2,b}(y,k)$ then $r^k y \bmod 2^b = 1$.*

Proof. For $b = 1$ we have $r = 1$ and $y \bmod 2 = 1$.

If $b \ge 2$ then r shows up as r_4 in Algorithm N2. Note that $b' < b$. By induction $z^k y \bmod 2^{b'} = 1$. So $z^k y = 1 + 2^{b'} j$ for some integer j.

Note that $2^{2b'} \equiv 0 \pmod{2^b}$. So $(k - 2^{b'} j)^k \equiv k^k - k 2^{b'} j k^{k-1} = k^k(1 - 2^{b'} j)$ by the binomial theorem.

By construction $r_2 \equiv (k+1)z$, $r_3 \equiv z^{k+1} y$, and $k r_4 \equiv r_2 - r_3$. Hence $k r_4 \equiv z(k + 1 - z^k y) = z(k - 2^{b'} j)$. So

$$k^k r_4^k y \equiv z^k y (k - 2^{b'} j)^k \equiv (1 + 2^{b'} j) k^k (1 - 2^{b'} j) \equiv k^k (1 - 2^{2b'} j^2) \equiv k^k.$$

But k^k is odd, so $r^k y = r_4^k y \equiv 1 \pmod{2^b}$ as claimed. $\square$

Finding 2-adic approximate square roots. Again fix an odd integer y. For each $b \ge 1$ we define and construct $\mathrm{nroot}_{2,b}(y,2)$ by the following algorithm:

Algorithm S2. Given an odd integer y and a positive integer b, we compute an integer $\mathrm{nroot}_{2,b}(y,2)$. In advance set $b' = \lceil (b+1)/2 \rceil$.
1. If $b = 1$: $\mathrm{nroot}_{2,b}(y,2)$ is 1 if $y \bmod 4 = 1$, 0 otherwise. Stop.
2. If $b = 2$: $\mathrm{nroot}_{2,b}(y,2)$ is 1 if $y \bmod 8 = 1$, 0 otherwise. Stop.
3. Compute $z \leftarrow \mathrm{nroot}_{2,b'}(y,2)$ by Algorithm S2.
4. If $z = 0$: $\mathrm{nroot}_{2,b}(y,2) = 0$. Stop.
5. Set $r_2 \leftarrow \mathrm{mul}_{2,b+1}(z,3)$.
6. Set $r_3 \leftarrow y\,\mathrm{pow}_{2,b+1}(z,3) \bmod 2^{b+1}$.
7. Set $r_4 \leftarrow (r_2 - r_3)/2 \bmod 2^b$. Now $\mathrm{nroot}_{2,b}(y,2) = r_4$.

Lemma 30.4. *Set $r = \mathrm{nroot}_{2,b}(y,2)$. If $i^2 y \bmod 2^{b+1} = 1$ for some odd integer i then $r \ne 0$. If $r \ne 0$ then $r^2 y \bmod 2^{b+1} = 1$.*

Proof. First consider $b = 1$. If $y \bmod 4 = 1$ then $r = 1$ so $r^2 y \bmod 4 = 1$. If $y \bmod 4 = 3$ then $r = 0$ and $i^2 y \bmod 4 = 3$ for any i.

Next consider $b = 2$. If $y \bmod 8 = 1$ then $r = 1$ so $r^2 y \bmod 8 = 1$. If $y \bmod 8 \neq 1$ then $r = 0$ and $i^2 y \bmod 8 = y \bmod 8 \neq 1$ for any i.

If $b \geq 3$ then r shows up as r_4 in Algorithm S2. Note that $b' < b$. If $z = 0$ then by induction $i^2 y \bmod 2^{b'+1}$ is never 1, so $i^2 y \bmod 2^{b+1}$ is never 1; and we set $r = 0$.

If $z \neq 0$ then by induction $z^2 y \bmod 2^{b'+1} = 1$. So $z^2 y = 1 + 2^{b'+1} j$ for some integer j. Note that $(1 - 2^{b'} j)^2 = 1 - 2^{b'+1} j + 2^{2b'} j^2 \equiv 1 - 2^{b'+1} j \pmod{2^{b+1}}$ since $2b' \geq b + 1$.

By construction $r_2 \equiv 3z \pmod{2^{b+1}}$, $r_3 \equiv z^3 y$, and $2r_4 \equiv r_2 - r_3$. Hence $2r_4 \equiv z(3 - z^2 y) = z(2 - 2^{b'+1} j)$. By Lemma 30.1, $r_4^2 \equiv z^2 (1 - 2^{b'} j)^2$. Thus

$$r^2 y = r_4^2 y \equiv z^2 y (1 - 2^{b'} j)^2 \equiv (1 + 2^{b'+1} j)(1 - 2^{b'+1} j) = 1 - 2^{2b'+2} j^2 \equiv 1$$

as claimed. $\square$

Perfect-power decomposition. We imitate Algorithm K from section 14: to see if n is a kth power, we compute and then check a tentative kth root.

Algorithm K2. Given an positive odd integer n, an integer $k \geq 2$ such that either $k = 2$ or k is odd, and an odd integer y (see Lemma 30.5), we see if n is a kth power. In advance set $f = \lfloor \lg 2n \rfloor$ and $b = \lceil f/k \rceil$.
1. Calculate $r \leftarrow \mathrm{nroot}_{2,b}(y, k)$.
2. If $k = 2$: If $r = 0$, print 0 and stop.
3. Check if $n = r^k$ with Algorithm C2. If so, print r and stop.
4. If $k = 2$: Check if $n = (2^b - r)^k$ with Algorithm C2. If so, print $2^b - r$ and stop.
5. Print 0 and stop.

Lemma 30.5. *Set $f = \lfloor \lg 2n \rfloor$ and $b = \lceil f/k \rceil$. Assume that $yn \bmod 2^{b+1} = 1$. If n is a kth power, Algorithm K2 prints $n^{1/k}$. If n is not a kth power, Algorithm K2 prints* 0.

Proof. We consider three cases.

Case 1: n is not a kth power. Then $n \neq r^k$ and $n \neq (2^b - r)^k$, so Algorithm K2 does not stop in steps 3 or 4. So it prints 0.

Case 2: $n = x^k$, and k is odd. By Lemma 30.3, $r^k y \bmod 2^b = 1$. Furthermore $yn \bmod 2^b = 1$ so $r^k \equiv n = x^k \pmod{2^b}$.

Put $c = r^{k-1} + r^{k-2} x + \cdots + x^{k-1}$; each term in this sum is odd, and there are k terms, so c is odd. But 2^b divides $r^k - x^k = (r - x)c$ so 2^b divides $r - x$. Both r and x are positive integers smaller than 2^b, so $r = x$. Hence $n = r^k$ and we print r in step 3.

Case 3: $n = x^k$, and $k = 2$. By Lemma 30.4, r is nonzero, since $yx^2 \bmod 2^{b+1} = 1$. By Lemma 30.4 again, $r^2 y \bmod 2^{b+1} = 1$. Hence $r^2 \equiv x^2 \pmod{2^{b+1}}$.

We have either $r \equiv x \pmod 4$ or $r \equiv -x \pmod 4$. If $r \equiv x \pmod 4$ then $r + x \equiv 2 \pmod 4$. Since 2^{b+1} divides $r^2 - x^2 = (r - x)(r + x)$, and only one power of 2 divides $r + x$, we must have $r \equiv x \pmod{2^b}$. Both r and x are positive integers smaller than 2^b, so $r = x$, so we print r in step 3.

If $r \equiv -x \pmod 4$ then $r \neq x$ so $r^2 \neq n$ so we do not print r in step 3. However, $2^b - r \equiv x \pmod 4$, and 2^{b+1} divides $(2^b - r)^2 - x^2$, so $2^b - r \equiv x \pmod{2^b}$ as above. Thus $2^b - r = x$ and we print $2^b - r$ in step 4. $\square$

Algorithm X2. Given an odd integer $n \geq 2$, we attempt to decompose n as a perfect power. In advance set $f = \lfloor \lg 2n \rfloor$.

1. Compute $y \leftarrow \mathrm{nroot}_{2,\lceil f/2\rceil+1}(n,1)$.
2. For each prime number $p < f$:
3. Apply Algorithm K2 to (n,p,y); let x be the result.
4. If $x > 0$, print (x,p) and stop.
5. Print $(n,1)$.

Lemma 30.6. *If n is a perfect power, Algorithm X2 prints a prime number p and a positive integer x such that $x^p = n$. If n is not a perfect power, Algorithm X2 prints $(n,1)$.*

Proof. By Lemma 30.3, $yn \bmod 2^{\lceil f/2\rceil+1} = 1$. If Algorithm X2 stops in step 4 then $x^p = n$ by Lemma 30.5. If Algorithm X2 never stops in step 4 then, by Lemma 30.5, n is not a pth power for any prime $p < f$, so n is not a perfect power. □

We could synthesize Algorithm X and Algorithm X2. For each k we can compute a tentative kth root x by either Algorithm N or Algorithm N2. We can then check whether $x^k = n$ by either Algorithm C or Algorithm C2. We could even run Algorithm C and Algorithm C2 in parallel, stopping as soon as either one sees that $n \neq x^k$. After Algorithm N it is probably best to try Algorithm C2 first; after Algorithm N2 it is probably best to try Algorithm C first. We have a great deal of flexibility here.

We could convert n into base q for $q > 2$, and then use the q-adics instead of the 2-adics. This is probably not worthwhile in practice, unless for some strange reason n is already known in base q. But it may be worthwhile to compute $n \bmod q$. See the next section for further discussion.

Notes. See [14, exercise 4.1–31] for an introduction to the 2-adic numbers.

Two q-adic applications of Newton's method are generally known as "Hensel's lemma." The first is the use of Newton's method to refine a q-adic root of a polynomial; see [29, page 14] or [11, page 84]. The second is the more general use of Newton's (multidimensional) method to refine a q-adic *factor* of a polynomial; see [14, exercise 4.6.2–22], [21, Theorem 8.3], or [24, page 40].

See [14, exercise 4.4–14] or [7] for a fast method of converting n into base q.

Our previous perfect-power run-time analysis does not apply if we use Algorithm C2 in place of Algorithm C. The results of Part IV would remain valid, but to prove that the resulting perfect-power detection algorithm runs in essentially linear time we would need q-adic versions of the theorems in Part V, and in particular of [19, Theorem 4].

31. Trial division

As usual fix $n \geq 2$. In this section we discuss several tricks based on computing $n \bmod q$ for one or more primes q.

If n has no small prime divisors, lower the exponent bound. If n is odd and $n = x^k$ then x is also odd. So $x \geq 3$ and $k \leq \log_3 n$. We have reduced the upper bound for exponents from $\log_2 n$ all the way down to $\log_3 n$ with a single test.

More generally we may compute $n \bmod q$ for all primes $q < T$. If $n \bmod q$ is always nonzero, we need to check exponents up to only $\log_T n$. In some applications, such as factoring, we may already have tested that n is not divisible by q for any q up to some large bound T. In other applications we could attempt to choose a smart cutoff T in advance so as to minimize our total run time. Another approach is to simultaneously (1) check $n \bmod q$ for larger and larger primes q and (2) check whether n is a kth power for larger and larger exponents k, until the two tests "meet in the middle."

If n has a prime divisor, find its order. What if we find that n is divisible by some prime q? We first compute the number $\operatorname{ord}_q n$ of factors q in n, together with $n/q^{\operatorname{ord}_q n}$. Then we check, for each p dividing $\operatorname{ord}_q n$, whether $n/q^{\operatorname{ord}_q n}$ is a pth power. Otherwise n cannot be a perfect power. (Note that $n/q^{\operatorname{ord}_q n}$ may be 1, in which case no testing is necessary.)

Recall that the 2-adic method in section 30 requires that n be odd. This is not a serious restriction. If n is even and we use the method here, we end up checking whether $n/2^{\operatorname{ord}_2 n}$ is a pth power, for various primes p; and $n/2^{\operatorname{ord}_2 n}$ is odd.

There are several plausible ways to compute the number of factors q in n.

If $q = 2$ then $\operatorname{ord}_q n$ is the number of 0 bits at the bottom of n's binary expansion.

If $q > 2$, we could do a binary search upon the number of factors. The idea is to compute $n \bmod q^c$ and $\lfloor n/q^c \rfloor$ for some integer $c \approx (\log_q n)/2$. If $n \bmod q^c \neq 0$ then $\operatorname{ord}_q n = \operatorname{ord}_q(n \bmod q^c)$; if $n \bmod q^c = 0$ then $\operatorname{ord}_q n = c + \operatorname{ord}_q(n/q^c)$. We chop c in half and repeat. This method takes essentially linear time if we use fast multiplication and the algorithms from section 11.

We could instead do a linear search; this amounts to always taking $c = 1$ in the above description. This will be faster than a binary search on average. We could compromise with a sequence of c that is at first optimistic but backs off quickly if necessary. For example, we may begin with $c = 1$, double c if $n \bmod q^c = 0$, and chop c in half if $n \bmod q^c \neq 0$.

Check the character of residues of n. If n is a kth power, and q is a prime with $q \bmod k = 1$, then $n^{(q-1)/k} \bmod q$ is either 0 or 1. A non-kth power has roughly a $1/k$ chance of passing this test.

Compute many $n \bmod q$ simultaneously. We can compute together $n \bmod q$ and $n \bmod q'$ almost as quickly as we can compute $n \bmod q$ alone: first we find $m = n \bmod qq'$; then we find $m \bmod q$ and $m \bmod q'$.

In general we may calculate $n \bmod q$ for every q in a set S by **binary splitting**: (1) calculate $m = n \bmod \prod_{q \in S} q$; (2) split S into two subsets, S' and $S - S'$; (3) recursively calculate $m \bmod q$ for every $q \in S'$; and (4) recursively calculate $m \bmod q$ for every $q \in S - S'$.

Check the residues of tentative roots. If $n = x^k$ then $n \bmod q = x^k \bmod q$. So if we know $n \bmod q$ we can try to weed out a tentative root x by calculating the kth power modulo q of $x \bmod q$. In practice this test is quite powerful: if $n \neq x^k$ then very few primes q divide $n - x^k$.

In this test q need not be prime. It might be convenient to check, for example, whether n agrees with x^k modulo 2^{32}, although this is redundant if we already use the 2-adic methods described in the previous section.

We could develop a fast randomized power-testing algorithm along these lines. Start from a tentative root x. First check if $x^k \leq 2n$. Then check if $n \bmod q$ equals $x^k \bmod q$ for a set of "random" primes q with product larger than n. This test will succeed if and only if $n = x^k$. If $n \neq x^k$ then we expect to test very few q's.

Check for small divisors of tentative roots. If n is not divisible by any primes $q < T$, and $n = x^k$, then x is not divisible by any primes $q < T$. So we may throw away any tentative root x that has prime factors smaller than T. This is much weaker than testing whether $n \bmod q = x^k \bmod q$ for each $q < T$, but it is also much faster.

Notes. See generally [3] for precedents. The approach of [3] is, for each k, to precompute a database of primes q with $q \bmod k = 1$, and then to systematically compute the characters $n^{(q-1)/k} \bmod q$ for each q in the database. [3] also suggests (1) checking whether n is divisible by small primes, (2) lowering the exponent bound if n is not divisible by any small primes, and (3) finding $\operatorname{ord}_q n$ (with a linear search) if q divides n.

Our binary search method for computing $\operatorname{ord}_q n$ is a straightforward optimization of the following procedure: first apply [14, exercise 4.4–14] to write n in base q; then see how many of the low "$q$its" are zero.

The binary splitting method described above appears in, e.g., [1, page 291].

See [18] for an overview of practical and theoretical methods for checking whether x (or n) has a prime factor smaller than T.

In some applications we may know $n \bmod q$, or a representation of n from which $n \bmod q$ is easy to derive. Victor Miller points out that if n is represented in the factorial base [14, equation 4.1–10] then it is easy to compute $n \bmod q$ for small primes q.

We have many options here. Each subset of options poses a new optimization problem—e.g., if we use characters as in [3] but with fast arithmetic and binary splitting, how much trial division should we do?—for which an exact answer will depend heavily on characteristics of the computer at hand. Having not yet solved all such problems, the author does not feel competent to declare one algorithm the "winner."

32. Table of notation

$\#S$	the number of elements of S	
$j!$	j factorial; $j! = (j-1)!j$	
$\lfloor t \rfloor$	the largest integer less than or equal to t	
$\lceil t \rceil$	the smallest integer greater than or equal to t	
$\|t\|$	the absolute value of t; t if $t \ge 0$, $-t$ if $t < 0$	
$[L, U]$	the closed interval $\{t : L \le t \le U\}$	
a	an integer; the exponent for a floating-point number	
A_j	a real number; $A_j \ge 4$; upper bound for $H(\alpha_j)$	§24
α	a rational number	
$\operatorname{adj} Q$	the adjoint of Q; the matrix of cofactors of Q	
b	a positive integer; bits of precision	
B	bits used in computing $\operatorname{nroot}_b(y, k)$; $B = \lceil \lg(66(2k+1)) \rceil$ for Algorithm B; for Algorithm N, $B = b+5$ if $b - \lceil \lg k \rceil$ is odd, $B = b+6$ otherwise	§11
B	a real number; $B \ge 4$; upper bound for $H(\beta_{ij})$	§24
β	a rational number	
c	a positive integer	
C	a positive integer	
δ	a positive floating-point number; power of 2	
$d(i, j)$	logarithmic distance from i to j; $\|i - j\| < 2^{d(i,j)+1}$	§13
$\det Q$	the determinant of Q	

$\operatorname{div}_b(r,k)$	a floating-point approximation to r/k; $1 \le r/k \operatorname{div}_b(r,k) < 1+2^{1-b}$	§6
$\operatorname{div}_{2,b}(r,k)$	a 2-adic approximation to r/k; $r \equiv k \operatorname{div}_{2,b}(r,k) \pmod{2^b}$	§30
e	the base for natural logarithms; $e = \sum_{j\ge 0} 1/j! \approx 2.71828$	
ϵ	a real number	
$\exp t$	the exponential of t; $\exp t = e^t$	
f	a positive integer; $2^{f-1} \le n < 2^f$; $f = \lfloor \lg 2n \rfloor$	
$F(n)$	difficulty of distinguishing n from a perfect power	§16
g	an integer	
$\gcd S$	the greatest common divisor of the elements of S	
$H(\alpha)$	height of α	§24
$\vartheta(t)$	$\vartheta(t) = \sum_{p\le t} \log p \approx t$	§19
$\vartheta_2(t)$	$\vartheta_2(t) = \sum_{p\le t} \log^2 p \approx t \log t - t$	§19
Θ	a matrix	§26
i	an integer	
I	an interval	
j	an integer	
k	a positive integer; exponent	
K	a real number	
κ	a real number	
$\ell(t)$	$\ell(t) = \sum_{p\le t} (1/p)\log p \approx \log t$	§19
L	a real number	
Λ	a real number; linear form in logarithms	§24
$\lg t$	$\lg t = \log_2 t = (\log t)/(\log 2)$	
$\log t$	the natural logarithm of t	
m	an integer	
$M(b)$	an upper bound on the time used to find the product of two b-bit integers	§4
$\mu(b)$	a nondecreasing upper bound for $M(b)/b$	§4
$\max S$	the largest element of S	
$\min S$	the smallest element of S	
$i \bmod j$	for $j > 0$, the remainder when i is divided by j; $i = j\lfloor i/j \rfloor + (i \bmod j)$	§30
$\operatorname{mul}(r,k)$	$\operatorname{mul}(r,k) = kr$	§5
$\operatorname{mul}_{2,b}(m,k)$	$\operatorname{mul}_{2,b}(m,k) = km \bmod 2^b$	§30
n	a positive integer	
$\operatorname{nroot}_b(y,k)$	negative kth root; a floating-point approximation to $y^{-1/k}$; $1 - 2^{-b} < y^{-1/k}/\operatorname{nroot}_b(y,k) < 1 + 2^{-b}$	§11
$\operatorname{nroot}_{2,b}(y,k)$	negative kth root; an odd integer z such that $z^k y \bmod 2^b = 1$	§30
$o(1)$	the set of functions ϵ such that $\lim_{t\to\infty} \epsilon(t) = 0$	
$\operatorname{ord}_q x$	the number of powers of q in x	§31
p	a prime number	
$P(k)$	number of multiplications involved in computing a kth power; $P(k) \le 2\lfloor \lg k \rfloor$	§7
$\operatorname{pow}_b(r,k)$	a floating-point approximation to r^k; $1 \le r^k/\operatorname{pow}_b(r,k) < (1 + 2^{1-b})^{2k-1}$	§7
$\operatorname{pow}_{2,b}(x,k)$	a 2-adic approximation to x^k; $\operatorname{pow}_{2,b}(x,k) = x^k \bmod 2^b$	§30

q	a prime number	
Q	a matrix	§26
r	a positive floating-point number	
$\operatorname{round} t$	the nearest integer to t; integer i such that $\lvert i-t\rvert < 1/2$	§14
s	a positive floating-point number	
S	a set	
t	a real number	
T	a real number	
$\operatorname{trunc}_b r$	a b-bit floating-point approximation to r; $1 \le r/\operatorname{trunc}_b r < 1+2^{1-b}$	§6
u	a real number	
U	a real number	
v	a real number	
V	a vector	
w	a positive floating-point number	
x	a positive integer; base	
y	a positive floating-point number	
z	a positive floating-point number	
Ω	a real number	§24

References

1. A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.
2. R. S. Anderssen and Richard P. Brent (editors), *The Complexity of Computational Problem Solving*, University of Queensland Press, Queensland, 1976.
3. Eric Bach, Jonathan Sorenson, *Sieve Algorithms for Perfect Power Testing*, Algorithmica **9** (1993), 313–328.
4. Alan Baker, *The theory of linear forms in logarithms*, in [5], 1–27.
5. Alan Baker, D. W. Masser (editors), *Transcendence theory: advances and applications*, Academic Press, 1977.
6. Jonathan M. Borwein, Peter B. Borwein, *Pi and the AGM*, Wiley, New York, 1987.
7. Richard P. Brent, *The complexity of multiple-precision arithmetic*, in [2], 126–165.
8. Richard P. Brent, *Fast Multiple-Precision Evaluation of Elementary Functions*, Journal of the Association for Computing Machinery **23** (1976), 242–251.
9. E. R. Canfield, Pal Erdős, Carl Pomerance, *On a problem of Oppenheim concerning "factorisatio numerorum"*, Journal of Number Theory **17** (1983), 1–28.
10. Henri Cohen, *A Course in Computational Algebraic Number Theory*, Springer-Verlag, Berlin, 1993.
11. A. Frölich, M. J. Taylor, *Algebraic number theory*, Cambridge University Press, Cambridge, 1991.
12. Roger A. Horn, Charles A. Johnson, *Matrix Analysis*, Cambridge University Press, Cambridge, 1985.
13. Donald E. Knuth, *The Art of Computer Programming, volume 1: Fundamental Algorithms*, 2nd edition, Addison-Wesley, Reading, Massachusetts, 1973.
14. Donald E. Knuth, *The Art of Computer Programming, volume 2: Seminumerical Algorithms*, 2nd edition, Addison-Wesley, Reading, Massachusetts, 1981.
15. Arjen K. Lenstra, Hendrik W. Lenstra, Jr. (editors), *The development of the number field sieve*, Lecture Notes in Mathematics 1554, Springer-Verlag, Berlin, 1993.
16. Arjen K. Lenstra, Hendrik W. Lenstra, Jr., Mark S. Manasse, John M. Pollard, *The number field sieve*, in [15], 11–40.
17. Hendrik W. Lenstra, Jr., private communication.
18. Hendrik W. Lenstra, Jr., Jonathan Pila, Carl Pomerance, *A hyperelliptic smoothness test. I*, Philosophical Transactions of the Royal Society of London, Series A **345** (1993), 397–408.

19. John H. Loxton, *Some problems involving powers of integers*, Acta Arithmetica **46** (1986), 113–123.
20. John H. Loxton, Alfred J. van der Poorten, *Multiplicative dependence in number fields*, Acta Arithmetica **42** (1983), 291–302.
21. Hideyuki Matsumura, *Commutative ring theory*, Cambridge University Press, Cambridge, 1986.
22. James R. Munkres, *Elements of Algebraic Topology*, Addison-Wesley, Reading, Massachusetts, 1984.
23. H. Nussbaumer, *Fast Polynomial Transform Algorithms for Digital Convolution*, IEEE Transactions on Acoustics, Speech, and Signal Processing **28** (1980), 205–215.
24. Michael E. Pohst, *Computational Algebraic Number Theory*, Birkhäuser, Basel, 1993.
25. William H. Press, Brian P. Flannery, Saul P. Teukolsky, William P. Vetterling, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge, 1986.
26. Paul Pritchard, *Fast compact prime number sieves (among others)*, J. Algorithms **4** (1983), 332–344.
27. J. B. Rosser, L. Schoenfeld, *Approximate formulas for some functions of prime numbers*, Illinois Journal of Mathematics **6** (1962), 64–94.
28. A. Schönhage, V. Strassen, *Schnelle Multiplikation großer Zahlen*, Computing **7** (1971), 281–292.
29. Jean-Pierre Serre, *A Course in Arithmetic*, Springer-Verlag, New York, 1973.
30. E. T. Whittaker, G. N. Watson, *A Course of Modern Analysis*, 4th edition, Cambridge University Press, 1927.

5 Brewster Lane, Bellport, NY 11713