HOW TO FIND SMALL FACTORS OF INTERVALS

DANIEL J. BERNSTEIN

ABSTRACT. This paper presents an algorithm that, given a set of positive integers, finds all the prime factors $\leq y$ of each integer. If there are $y/(\lg y)^{O(1)}$ integers, each with $(\lg y)^{O(1)}$ bits, then the algorithm takes time $(\lg y)^{O(1)}$ per integer, using fast multiplication of numbers with $y/(\lg y)^{O(1)}$ bits. This paper also presents a comprehensive survey of previous methods and a survey of applications. The new algorithm is useful in congruence-combination methods to compute large factors, discrete logarithms, class groups, etc.; in particular, it indirectly speeds up the number field sieve.

1. INTRODUCTION

Fix an integer $y \geq 2$. A prime number is small if it is at most $y$.

Consider a positive integer $n$. What are all the small prime divisors of $n$? Is $n$ smooth, i.e., are all its prime divisors small?

(Readers who wish to experiment with examples may focus on the following special case: $y$ is $10^6$, while $n$ ranges up to $10^{60}$. More generally, in the applications discussed in Section 3, log $n$ log log $n$ typically ranges up to roughly $2(\log y)^2$.)

This paper presents an algorithm that answers these questions for many integers $n$ simultaneously. If there are $y/(\lg y)^{O(1)}$ integers, each with $(\lg y)^{O(1)}$ bits, then the algorithm takes total time only $y(\lg y)^{O(1)}$. Here $\lg = \log_2$. The time per integer is $(\lg y)^{O(1)}$, just as if there were a polynomial-time algorithm to handle a single $n$.

The algorithm is presented in a bottom-up fashion in Sections 4, 5, 6, and 7. The reader who wishes to understand the central idea as quickly as possible may skip to Algorithm 7.1.

The algorithm manipulates integers with as many as $y(\lg y)^{O(1)}$ bits. The first step—see Algorithm 7.1—is to multiply together all the integers $n$ that we want to factor! To achieve the time bound $(\lg y)^{O(1)}$ stated above, one needs to multiply integers with $b$ bits in time $b(\lg b)^{O(1)}$ for various $b$.

The fact that one can quickly find all small prime divisors of many integers is a special case of the result proved in my recent paper [21]: given any finite subset $S$ of any free commutative monoid, one can very quickly factor $S$ into coprimes, if there are fast algorithms for multiplication, exact division, and gcd. The algorithm in this paper is simply a streamlined version of the algorithm in the last section of [21].

Section 2 of this paper presents a comprehensive survey of previous smoothness-testing methods. Section 3 presents a survey of applications. The reader can find older surveys of factorization in the books [37], [99], [159], [97], and [63].

Date: 20020923.

1991 Mathematics Subject Classification. Primary 11Y05; Secondary 11Y16.

The author was supported by the National Science Foundation under grant DMS-9974009.
Heavily tuned implementation results for the new algorithm, including various improvements in subroutines as described in [23] and [24], will be presented in a future paper.

Acknowledgments. Thanks to Carl Pomerance for drawing my attention to the unsieveable integers in [55]. Thanks to Christine Swart for her comments. Thanks to two anonymous referees for their comments.

2. Previous algorithms

The most obvious method to find small prime divisors of \( n \) is trial division: divide \( n \) by 2, 3, 5, etc. This takes time \( y^{1+o(1)} \) if \( n \) has \( y^o(1) \) bits.

The early-abort method in [142] and [154] is a modification to trial division. The idea is to check, after each division, how many factors of \( n \) have been discovered, and give up if the unfactored part of \( n \) is uncomfortably large. Pomerance showed in [142] that, for uniform random \( n \) and for a particular definition of “uncomfortably large,” early-abort trial division takes average time only \( y^o(1) \), while it has a \( y^{-1/2+o(1)} \) chance of recognizing \( n \) if \( n \) is smooth. In the applications discussed in Section 3, the speedup outweighs the loss of effectiveness.

Pollard’s fast-factorial method in [138] achieves the same result as trial division in time only \( y^{1/2+o(1)} \). The \( o(1) \) can be reduced by Schönhage’s technique in [164]. Pomerance showed in [142] that early-abort fast factorial takes average time only \( y^o(1) \), while it has a \( y^{-1/4+o(1)} \) chance of recognizing \( n \) if \( n \) is smooth.

Pollard’s \( \rho \) method in [139] seems to achieve the same result as trial division in time \( y^{1/2+o(1)} \), with the \( o(1) \) not quite as large as in the fast-factorial method. See [32] and [36] for improvements, and [14] for analysis of a randomized version of the method.

Pollard’s \( p-1 \) method in [138] finds certain primes \( p \) quickly: in particular, it seems to find at least one out of every \( z \) primes in time \( z^{1+o(1)} \) if \( n \) has \( z^o(1) \) bits, where \( 2(\log z)^2 = \log y \log \log y \). The same comment applies to Williams’s \( p+1 \) method in [179] and Lenstra’s elliptic-curve method in [110]. A uniform random choice of \( z^{1+o(1)} \) elliptic curves seems to find every prime \( \leq y \) in total time \( z^{2+o(1)} = \exp(2+o(1)) \log y \log \log y \) with negligible error probability. For further discussion see [33], [121], [90], [34], [122], [10], [169], [153], [30], and [35].

The other \( \Psi_{k}(p) \) methods in [15], and the hyperelliptic-curve method in [111], seem slower than the \( p-1 \) method. The hyperelliptic-curve method has the virtue of provably finding every prime \( \leq y \) in subexponential time with negligible error probability.

Impact of the new algorithm. The algorithm introduced in this paper is faster than any of the above methods, when \( y \) is reasonably large and when there are many \( n \)'s to test for smoothness. Furthermore, this algorithm is easy to prove correct, and has no chance of error.

The previous methods remain useful in two situations. First, if there are only a few \( n \)'s to test for smoothness, the new algorithm is not much faster than trial division. Second, in the context of special-purpose hardware, low-memory methods such as the early-abort elliptic-curve method are more cost-effective than high-memory methods such as the new algorithm; see [20] for further discussion.
**Sieving.** In some applications, the integers \( n \) are successive values of an integer polynomial: \( f(0), f(1), f(2), \ldots \). Sieving is a well-known method of factoring many such \( n \)'s simultaneously: build an array of, say, \( A \) successive values of \( n \); for each prime \( p \), mark \( p \) at each position in the array where \( n \) is divisible by \( p \). The set of these positions is a union of arithmetic progressions mod \( p \).

One can use an early abort with sieving. Sieve all primes \( p \) up through, say, \( B \); throw away the \( n \)'s whose unfactored part is uncomfortably large; then apply some other method to the \( n \)'s that remain. The sieving time per number is \( B^{1+o(1)}/A + r(A)(\log \log B + O(1)) \), where \( r(A) \) is the random-access time for an array of length \( A \), i.e., the time needed to make a single mark. On typical computers, \( r(A) \) increases in sudden steps: it jumps by an order of magnitude as \( A \) increases past “level-1 cache size,” then another factor of 2 or 3 as \( A \) increases past “level-2 cache size,” then several orders of magnitude as \( A \) increases past “DRAM size.”

**Impact of the new algorithm upon sieving.** The speed of sieving is indirectly affected by the speed of other factorization methods. A faster method of handling the \( n \)'s that remain after sieving means that one can afford to look at more \( n \)'s; so sieving can do a less precise job of identifying the interesting \( n \)'s; so one can reduce \( B \). If the overhead \( B^{1+o(1)}/A \) is large then reducing \( B \) improves sieve time; otherwise one can reduce \( A \), hopefully enough to reduce \( r(A) \), which again improves sieve time.

**The function-field case.** Many more methods are available in the function-field case. It is already well known that univariate polynomials over finite fields can be factored into irreducible polynomials quickly. The Kaltofen-Shoup polynomial-factorization method in [94] could be faster or slower than the algorithm described here; a careful comparison would account for the sizes of \( n \) and \( y \) and for many implementation details. There are several ways to merge the ideas into a single algorithm; this will be discussed in a future paper.

## 3. Applications

Consider the problem of finding all factors (not just small factors!) of a positive integer \( D \).

**The continued-fraction factorization method.** The Lehmer-Powers-Brillhart-Morrison continued-fraction method produces a set of integers \( n \), each \( n \) having a known square root mod \( D \), and finds nonempty subsets with square product. To find such subsets, it looks for smooth \( n \)'s, factors each \( n \) as a product of powers of \(-1, 2, 3, 5, \ldots \), and then finds linear relations among the exponent vectors mod 2. See [103], [126], [182], [142], [131], [154], [170], [183], [180], [174], [148], and [149]. See also [151], [101], [178], [57], [56], [109], [124], [69], and [16] for relevant linear-algebra algorithms.

The integers \( n \) in the continued-fraction method are bounded in absolute value by \( x \) for some \( x \in D^{1/2+o(1)} \); one chooses \( y \) with \((\log y)^2 \in (1/2+o(1))\log x \log \log x \). It seems that the first \( y^{2+o(1)} \) values of \( n \) always suffice to produce \( y^{1+o(1)} \) smooth integers, many square products, and the complete factorization of \( D \). The total time is \( y^{2+o(1)} = \exp \sqrt{(1+o(1))\log D \log \log D} \) if one can recognize the smooth \( n \)'s in time \( y^{o(1)} \) per number. The algorithm in this paper is a very fast way to recognize smooth \( n \)'s.
Rigorous factoring bounds. The methods of [70], [144], and [173] seem slower than the continued-fraction method, but they have the virtue of provably finding the complete factorization of every $D$ in subexponential average time. The Schnorr-Seysen-Lenstra-Lenstra-Pomerance class-group method developed in [163], [166], [104], and [112] is more complicated but provably factors every composite $D$ in average time $y^{2+o(1)}$ with $y$ as above.

Each of these methods has the same outline as the continued-fraction method. It is crucial to provably recognize smooth $n's$ quickly. One can do this with the elliptic-curve method or the hyperelliptic-curve method, but the algorithm in this paper is faster, simpler, and much easier to prove.

The linear sieve and its descendants. Schroeppel in 1977 introduced the idea of generating $n's$ as successive values of various polynomials so that many $n's$ could be factored simultaneously by sieving. Pomerance’s quadratic sieve is a simplification of Schroeppel’s linear sieve. Each method seems to always succeed in time $y^{2+o(1)}$ with $y$ as above. See [142], [81], [170], [64], [143], [67], [65], [167], [47], [152], [66], [157], [108], [13], [146], [158], [168], [135], [68], [9], [11], [28], and [50].

As explained in Section 2, the algorithm in this paper can be used to indirectly speed up sieving. Furthermore, a reduction in the sieve array size allows a reduction in the size of $n$; see, e.g., [53].

Pollard’s number-field sieve, as generalized by Buhler, Lenstra, and Pomerance, seems to always succeed in time $\exp((64/9 + o(1))^{1/3}(\log D)^{1/3}(\log \log D)^{2/3})$. See [140], [106], [107], [3], [45], [141], [61], [25], [42], [147], [123], [84], [19], [71], [150], [74], [75], [76], [78], [62], [77], [125], [128], [132], and [129]. The algorithm in this paper can again be used to indirectly speed up sieving and reduce the size of $n$.

Coppersmith’s number-field-sieve variant in [55] seems asymptotically faster, with $64/9$ reduced slightly. Coppersmith’s method factors many numbers with a sieve, and then factors not quite as many unsieveable numbers. The algorithm in this paper directly speeds up the handling of the unsieveable numbers; it may make Coppersmith’s variant worthwhile for current sizes of $D$.

Other applications. The ideas behind these integer-factorization methods are also used in the index-calculus method of computing discrete logarithms in finite fields. See [177], [119], [2], [91], [27], [73], [17], [60], [102], [5], and [165] for the basic index-calculus method; [160], [85], [161], [134], [162], [175], and [176] for an index-calculus application of the number-field sieve; and [54], [59], [133], [117], and [7] for a function-field analogue.

The same ideas are also used to compute class groups and regulators of number fields. See [89], [39], [40], [92], and [41].

4. Multiplication

One can compute $xz$, given nonnegative integers $x$ and $z$, in time at most $b(\lg b)^{O(1)}$ if $b$ is a positive integer with $xz < 2^b$. See, e.g., [22].

Starting from this bound $b(\lg b)^{O(1)}$, one could prove similar bounds for the amount of time spent on multiplications in Algorithms 5.1, 5.3, 6.1, 6.3, and 7.1.

However, it is easier, more illuminating, and more precise to start from a general bound $b\mu(b)$. Here $\mu$ is any nondecreasing positive function.

Time spent on multiplications is called $\mu$-time. The reader may check that $\mu$-time dominates the run time of the algorithms.
5. Division

Algorithms 5.1 and 5.3 are standard examples of Hensel’s method, i.e., 2-adic applications of Newton’s method.

**Algorithm 5.1.** Given a positive integer $b$ and an odd positive integer $u$, to print a nonnegative integer $v < 2^b$ such that $1 + uv \equiv 0 \pmod{2^b}$:
1. If $b = 1$: Print 1. Stop.
2. Set $c \leftarrow \lceil b/2 \rceil$.
3. Find $v_0 < 2^c$ such that $1 + u v_0 \equiv 0 \pmod{2^c}$ by Algorithm 5.1.
4. Set $u_0 \leftarrow u \mod{2^c}$ and $u_1 \leftarrow [u/2^c] \mod{2^c}$. (Now $u \equiv u_0 + 2^c u_1 \pmod{2^{2c}}$; and $1 + u_0 v_0 \equiv 0 \pmod{2^c}$.)
5. Set $z \leftarrow ((1 + u v_0)/2^c + u_1 v_0) \mod{2^c}$. (Now $1 + u v_0 \equiv 0 \pmod{2^c}$.)
6. Set $v \leftarrow v_0 + 2^c z v_0 \mod{2^b}$. (Now $1 + u v \equiv 1 + u v_0 + 2^c z u v_0 \equiv 2^c z + 2^c z u v_0 \equiv 2^c z + 2^c z \equiv 0 \pmod{2^b}$.)
7. Print $v$.

**Theorem 5.2.** Algorithm 5.1 uses $\mu$-time at most $6(b + \lceil \log b \rceil - 1)\mu(b + 1)$.

**Proof.** For $b = 1$: Algorithm 5.1 uses no $\mu$-time, and $b + \lceil \log b \rceil - 1 = 0$.

For $b \geq 2$: By induction, step 3 uses $\mu$-time at most $6(c + \lceil \log c \rceil - 1)\mu(c + 1) \leq 6((b + 1)/2 + \lceil \log b \rceil - 2)\mu(b + 1)$. Steps 5 and 6 use $\mu$-time at most $3(b + 1)\mu(b + 1)$ to compute the products $u_0 v_0, u_1 v_0,$ and $z v_0$, each of which is below $2^{2c} \leq 2^{b+1}$. The total $\mu$-time is at most $6\mu(b + 1)$ times $(b + 1)/2 + (b + 1)/2 + \lceil \log b \rceil - 2 = b + \lceil \log b \rceil - 1$.

**Algorithm 5.3.** Given positive integers $b$ and $c$, an odd positive integer $u < 2^c$, and a nonnegative integer $x < 2^{c+b}$, to print a nonnegative integer $r < 2^{c+1}$ such that $2^b r \equiv x \pmod{u}$:
1. Find $v < 2^b$ such that $1 + u v \equiv 0 \pmod{2^b}$ by Algorithm 5.1.
2. Set $x_0 \leftarrow x \mod{2^b}$ and $x_1 \leftarrow [x/2^b]$. (Now $x = 2^b x_1 + x_0$.)
3. Set $q \leftarrow u x_0 \mod{2^b}$. (Now $x_0 + w q \equiv x_0 + u v x_0 \equiv 0 \pmod{2^b}$.)
4. Set $r \leftarrow x_1 + (x_0 + u q)/2^b$. (Now $2^b r = x + u q \equiv x \pmod{u}$; and $r < 2^{c+1}$ since $x + u q < 2^{c+b} + 2^{c+b} = 2^{c+b+1}$.)
5. Print $r$.

**Theorem 5.4.** Algorithm 5.3 uses $\mu$-time at most $12(b + c)\mu(2(b + c))$.

**Proof.** Step 1 uses $\mu$-time at most $6(b + \lceil \log b \rceil - 1)\mu(b + 1) \leq 9b\mu(b + 1)$ by Theorem 5.2. Step 3 uses $\mu$-time at most $2b\mu(2b)$ to compute $x_0$. Step 4 uses $\mu$-time at most $(b + c)\mu(b + c)$ to compute $u q$. The total is at most $\mu(2b + 2c)$ times $9b + 2b + b + c < 12b + 12c$. Note that these bounds are rather crude. □

**Notes.** Algorithms 5.1 and 5.3 have some redundancy that can be removed; see [23]. The techniques of [23] also apply to Algorithms 6.3 and 7.1.

When $b$ is larger than $c$, one can save time in Algorithm 5.3 by handling $x$ in chunks. See [97, Exercise 4.3.3–13] and [97, Algorithm 4.3.1–D].

One could use real division instead of 2-adic division in the subsequent sections, but 2-adic division is easier to implement.

6. Multipoint Evaluation

Algorithm 6.3 is a standard example of the Borodin-Moenck method in [29], which reduces a large integer modulo many small integers in essentially linear time.
Let \( m \) be a positive integer. Let \( P = (p_1, p_2, \ldots, p_m) \) be a sequence of positive integers. The product tree of \( P \) is a binary tree of positive integers defined as follows. The root of the tree is \( p_1p_2 \cdots p_m \). If \( m = 1 \), the root has no children. If \( m \geq 2 \), the root has the product tree of \( p_1, p_2, \ldots, p_{\lfloor m/2 \rfloor} \) as its left subtree, and the product tree of \( p_{\lfloor m/2 \rfloor + 1}, \ldots, p_m \) as its right subtree. Observe that each non-leaf vertex in the product tree is the product of its two children.

For example, here is the product tree of \((2, 3, 5, 7, 11, 13, 17)\):

```
      510510
     /     /
   30    17017
  / 5    / 5
 3 15  11 13
```

To reduce an integer \( x \) modulo \( 2, 3, 5, 7, 11, 13, 17 \), Algorithm 6.3 first reduces \( x \) modulo 510510, then reduces the result modulo 30 and 17017, etc.

Define bits \( \text{bits}(p_1, \ldots, p_m) = [\lfloor \log(p_1 + 1) \rfloor] + \cdots + [\lfloor \log(p_m + 1) \rfloor] \). The root of the product tree of \( P \) is smaller than \( 2^{\text{bits}}P \).

**Algorithm 6.1.** Given positive integers \( m, p_1, p_2, \ldots, p_m \), to print the product tree of \((p_1, p_2, \ldots, p_m)\):

1. If \( m = 1 \): Print \( p_1 \). Stop.
2. Print the product tree \( T \) of \((p_1, p_2, \ldots, p_{\lfloor m/2 \rfloor})\) by Algorithm 6.1.
3. Print the product tree \( U \) of \((p_{\lfloor m/2 \rfloor + 1}, \ldots, p_m)\) by Algorithm 6.1.
4. Print the product of the roots of \( T \) and \( U \).

**Theorem 6.2.** If \( b = \text{bits}(p_1, p_2, \ldots, p_m) \), \( m \leq 2^k \), and \( k \geq 0 \) then Algorithm 6.1 uses \( \mu \)-time at most \( k\mu(b) \).

**Proof.** If \( m \leq 1 \) then Algorithm 6.1 uses no \( \mu \)-time. So assume \( m \geq 2 \); then \( k \geq 1 \). By induction on \( k \), step 2 uses \( \mu \)-time at most \((k - 1)a\) times \( \mu(a) \leq \mu(b) \), where \( a = \text{bits}(p_1, p_2, \ldots, p_{\lfloor m/2 \rfloor}) \); and step 3 uses \( \mu \)-time at most \((k - 1)(b - a)\) times \( \mu(b - a) \leq \mu(b) \). Step 4 uses \( \mu \)-time at most \( b\mu(b) \). The total \( \mu \)-time is at most \( b\mu(b) \) times \((k - 1)a + (k - 1)(b - a) + b = kb \). \( \square \)

**Algorithm 6.3.** Given a nonnegative integer \( x \), and given the product tree \( T \) of a nonempty sequence \( P \) of odd positive integers, to print \( \{p \in P : x \mod p = 0\} \):

1. Set \( u \leftarrow \text{the root of } T \).
2. Set \( c \leftarrow [\log(u + 1)] \) and \( d \leftarrow [\log(x + 1)] \). (Now \( 1 \leq 2^{c-1} \leq u < 2^c \).)
3. If \( d > c + 1 \): Apply Algorithm 5.3 to \((d - c, c, u, x)\) to find a nonnegative integer \( r < 2^{c+1} \) such that \( 2^{d-c}r \equiv x \mod u \).
4. If \( d \leq c + 1 \): Set \( r \leftarrow x \).
5. (Now \( 0 \leq r < 4u \) and \( 2^kr \equiv x \mod u \) for some \( k \).) If the root of \( T \) has no children: Print \( u \) if \( r \in \{0, u, u + u, u + u + u\} \). Stop.
6. Apply Algorithm 6.3 to \( r \) and the left subtree of \( T \).
7. Apply Algorithm 6.3 to \( r \) and the right subtree of \( T \).

**Theorem 6.4.** If \( b = \text{bits } P \), \#\( P \leq 2^k \), \( k \geq 0 \), \( x < 2^e \), and \( e \geq 0 \) then Algorithm 6.3 uses \( \mu \)-time at most \( e + 2kb + 2^{k+1} - 2 \times 12\mu(2\max\{e, b + 1\}) \).
Proof. First $u < 2^b$ so $c \leq b$; also $d \leq e$. Step 3 uses $\mu$-time at most $12\mu(2e)$ by Theorem 5.4, whether or not $d > c + 1$.

For $k = 0$: There is no other $\mu$-time; and $e + 2kb + 2^{k+1} - 2 = e$.

For $k \geq 1$: Write $a = \text{bits } Q$ where $Q$ is the left half of $P$. By induction on $k$, step 6 uses $\mu$-time at most $(c + 1) + 2(k - 1)a + 2^k - 2 \leq b + 2(k - 1)a + 2^k - 1$ times $12\mu(2\max\{c + 1, a + 1\}) \leq 12\mu(2b + 1)$. Similarly, step 7 uses $\mu$-time at most $b + 2(k - 1)(b - a) + 2^k - 1$ times $12\mu(2b + 1))$. The total is at most $e + 2b + 2(k - 1)b + 2^{k+1} - 2 = e + 2kb + 2^{k+1} - 2$ times $12\mu(2\max\{e, b + 1\})$. □

Notes. The product tree for $P$ takes substantially more memory than $P$ does. One can save memory by discarding portions of the product tree and recomputing them on demand.

Algorithm 5.1 can be sped up in the context of Algorithm 6.3. Say one wants to divide by $pp'$, then by $p$, then by $p'$. Algorithm 5.1 finds an approximate reciprocal of $pp'$ by Newton iteration starting from 1. It is better to start from the product of approximate reciprocals of $p$ and $p'$.

Strassen in [172] suggested multiplying elements of $P$ in a different order: replace the two smallest elements of $P$ by their product, then repeat. One can use a heap to rapidly identify the smallest elements of $P$ at each step; see [181], [79], [98, Exercise 5.2.3–18], and [98, Exercise 5.2.3–28]. This saves time in Algorithms 6.1 and 6.3 when the elements of $P$ have wildly varying sizes.

7. Factorization

Algorithm 7.1. Given a sequence $P = (p_1, p_2, \ldots, p_m)$ of odd primes, and a nonempty finite multiset $N$ of positive integers, to print $(n, \{p \in P : n \mod p = 0\})$ for each $n \in N$:

1. If $m = 0$: Print $(n, \{\})$ for each $n \in N$. Stop.
2. Compute $x \leftarrow \prod_{n \in N} n$ by Algorithm 6.1.
3. Compute the product tree $T$ of $P$ by Algorithm 6.1.
4. Compute $P' \leftarrow \{p \in P : x \mod p = 0\}$ by Algorithm 6.3. (The elements of $P'$ are exactly the primes relevant to factorizations of elements of $N$.)
5. If $\#N = 1$: Find $n \in N$. Print $(n, P')$. Stop.
6. Select $M \subseteq N$ with $\#M = \lfloor \#N/2 \rfloor$.
7. Apply Algorithm 7.1 to $(M, P')$.
8. Apply Algorithm 7.1 to $(N - M, P')$.

For example, given $P = (3, 5, 7, 11, 13, 17, 19)$ and $N = \{492, 2567, 3135, 5889\}$, Algorithm 6.3 computes $x = 492 \cdot 2567 \cdot 3135 \cdot 5889$ and $P' = \{3, 5, 11, 13, 17, 19\}$. It recursively factors $M = \{492, 2567\}$ and $N - M = \{3135, 5889\}$ using $P'$. The following picture shows the subsequent levels of recursion:
Theorem 7.2. If $\#N \leq 2^j$, $j \geq 0$, $b = \text{bits } P$, $\#P \leq 2^k$, and $k \geq 0$ then Algorithm 7.1 uses $\mu$-time at most $(100jk + 108j + (j + 1)/2 + 12)$ bits $N + 25kb + 24 \cdot 2^k$ times $\mu(2 \max \{\text{bits } N, b + 1\})$.

Proof. Step 4 uses $\mu$-time at most $12(\text{bits } N + 2kb + 2^{k+1}) \mu(2 \max \{\text{bits } N, b + 1\})$ by Theorem 6.4. Steps 2 and 3 use $\mu$-time at most $j(\text{bits } N) \mu(\text{bits } N + kb\mu(b))$ by Theorem 6.2.

For $j = 0$: The total is at most $(12(\text{bits } N + 2kb + 2^{k+1}) \mu(2 \max \{\text{bits } N, b + 1\})$.

For $j \geq 1$: The point is that $P'$ cannot be much larger than $N$. Each element of $P'$ divides $x$, so $\prod_{p \in P'} p$ divides $x$, so $\sum_{p \in P'} \lg p \leq \lg x < \text{bits } N$. The crude bound $[\lg(p+1)] \leq 2\lg p$ then implies that $\text{bits } P' \leq 2\text{bits } N$. Also, $\#P' < \text{bits } N$, so $2^k < 2\text{bits } N$ if $k'$ is the least nonnegative integer with $\#P' \leq 2^{k'}$.

Therefore, by induction on $j$, step 7 uses $\mu$-time at most

$$(100(j - 1)k' + 108(j - 1) + (j - 1)j/2 + 12) \text{ bits } M + 25k' \text{ bits } P' + 24 \cdot 2^k$$

$$(\text{times } \mu(2 \max \{\text{bits } M, \text{bits } P' + 1\})) \leq \mu(2 \max \{\text{bits } N, b + 1\})$$. Similarly, step 8 uses $\mu$-time at most $(100(j - 1)k + 108(j - 1) + (j - 1)j/2 + 12) \text{ bits } (N - M) + (50k + 48) \text{ bits } N$ times $\mu(2 \max \{\text{bits } M, \text{bits } P' + 1\})$.

The total is $\mu(2 \max \{\text{bits } N, b + 1\})$ times $12 \text{ bits } N + 24kb + 12 \cdot 2^{k+1} + j \text{ bits } N + kb + (100(j - 1)k + 108(j - 1) + (j - 1)j/2 + 12) \text{ bits } N + (100k + 96) \text{ bits } N = (100jk + 108j + (j + 1)/2 + 12) \text{ bits } N + 25kb + 12 \cdot 2^{k+1}$ as claimed.

Notes. Before feeding $n$ to Algorithm 7.1, one should trial-divide $n$ by 2, and perhaps by a few more primes. The unfactored portion of $n$ often takes slightly less space than $n$, speeding up Algorithm 7.1. The speedup should be balanced against the time taken by trial division.

In step 6 of Algorithm 7.1, rather than continuing the recursion, one can trial-divide each element of $N$ by $P'$. The best cutoff for the size of $N$ depends on the relative speeds of trial division and Algorithm 6.3.

In step 5 of Algorithm 7.1, if one wants to know whether $n$ is smooth, one can simply trial-divide $n$ by $P'$. At this point $P'$ has very few elements. See [21] for asymptotically faster algorithms.

One can save some time in Algorithm 7.1 by recording the product tree for $N$ in step 2, then reusing the tree in steps 7 and 8.

In practice, $P'$ is rarely as large as $N$. One can profitably split $N$ into more than two pieces at the end of Algorithm 7.1.

References


Department of Mathematics, Statistics, and Computer Science (M/C 249), The University of Illinois at Chicago, Chicago, IL 60607-7045

E-mail address: djb@cr.yp.to