The Seven-Eleven Problem

Paul Pritchard and David Gries[1]

TR 83-574
September 1983

Department of Computer Science
Cornell University
Ithaca, New York 14853

[1]Supported by NSF grant MCS-81-03605.

0. Introduction

Throughout the United States there are small grocery stores that, for convenience, are open at all hours of the day and night. In the south these are called 711 stores, because originally they were open from 7 a.m. until 11 p.m. One day a customer bought four items at a 711 store. The cashier bagged them and said "That will be $7.11, please." The customer asked "Is it $7.11 because this is a 711 store?" "No," replied the cashier, "I multiplied the prices together and got $7.11." "But you're supposed to *add* them, not multiply them." said the customer. "Oh, you're right!" exclaimed the cashier. "Let me recalculate ... that will be $7.11."

What were the prices of the four items?

From this puzzle we extract a problem: Design an algorithm that, given two natural numbers N and M, finds, if they exist, four integers b, c, d, e satisfying

(0) (a) $1 \leq b, c, d, e \leq N$,
(b) $b + c + d + e = N$,
(c) $b * c * d * e = M$.

Call such a 4-tuple (b, c, d, e) a *solution*.

For the original puzzle, use $N = 711$ and $M = 711000000$, and if (b, c, d, e) is a solution the prices in dollars are $b/100$, $c/100$, $d/100$ and $e/100$.

We develop an algorithm by starting with a very general algorithm called *Saddleback Search* and then using properties of the objects being manipulated to speed it up. Our final algorithm requires only $O(N^{\epsilon})$ arithmetic operations for any $\epsilon > 0$, given a certain partial factorization of M. If the Continued Fraction factoring method is used to produce the latter, and the assumptions on which its analysis in [5] is predicated are true, then the total cost of our algorithm is $O(N^{\epsilon})$ arithmetic operations for any $\epsilon > 0$.

We note that there is no solution if $(N/4)^4 < M$ and therefore treat only the case $M \leq (N/4)^4$ throughout the paper.

1. A naive quartic algorithm

A naive solution to the problem is to check all 4-tuples (b, c, d, e) satisfying $1 \leq b, c, d, e \leq N$ to see whether (0b,0c) are satisfied. This approach has the virtue of being obviously correct. The algorithm requires $O(N^4)$ arithmetic operations.

2. A cubic algorithm

Any solution to (0) can be arranged so that

$$1 \leq b \leq c \leq d \leq e \leq N.$$

So we restrict our search to the space of *ordered* 4-tuples (b, c, d, e).

It is now apparent that $1 \leq b \leq N \div 4$, since $b > N \div 4$ implies $b + c + d + e > N$. Similarly, if b is given, then $b \leq c \leq (N - b) \div 3$. Also, if b, c are both given then

$c \le d \le (N-b-c) \div 2$.

Finally, if b, c, d are given, $e = N-b-c-d$. At this point we can see an algorithm that requires only $O(N^3)$ operations. It is the naive algorithm for the following problem. Find all 3-tuples (b, c, d) with

(1) (a) $b*c*d*(N-b-c-d) = M$,
(b) $1 \le b \le N \div 4$,
(c) $b \le c \le (N-b) \div 3$,
(d) $c \le d \le (N-b-c) \div 2$.

There is an obvious 1-1 correspondence between solutions to (1) and ordered solutions to (0).

3. A quadratic algorithm

Incorporating constraint (0b) into the search reduced by 1 the exponent of the complexity. A further reduction can be obtained by paying attention to constraint (1a). Consider a fixed pair b, c satisfying (1b,1c). By rearranging (1a), we see that (b, c, d) is a solution of (1) if and only if the following quadratic equation in d has an integer solution satisfying (1d):

(2) $b*c*d^2 - b*c*(N-b-c)*d + M = 0$.

Then $(b, c, d, N-b-c-d)$ is an ordered solution to (0).

It is now easy to write an algorithm that searches through pairs (b, c) looking for a solution d to (2). This algorithm involves finding the square root of a rather large integer at each of its $O(N^2)$ steps.

4. A neater quadratic algorithm

Let us reconsider the naive cubic algorithm for problem (1), with b varying from 1 to $N \div 4$ (in the outermost loop). At any stage, variables b, c, d contain integers, and it is known that no 3-tuple with first component $< b$ is a solution. The problem, given fixed b, is to search the space of ordered pairs (c, d) efficiently. Let us briefly digress to look at Saddleback Search, for reasons that will become apparent.

Saddleback Search [1, p. 215]. Given is an array $f[0{:}m, 0{:}n]$. The elements in each row and each column are in non-decreasing order. A value x lies in f; the problem is to determine its row and column number. (If x is in f more than once, any one of its positions will do.) The following algorithm solves the problem in $O(m+n)$ operations. It begins by identifying a rectangular section of f in which x appears, then iteratively reduces the size of this rectangle, always maintaining the fact that x lies in it. In a rougher sense, it begins looking for x at the upper right element $f[0,n]$ and proceeds towards the lower left element $f[m,0]$; the fact that the rows and columns are ordered allows the search to proceed efficiently.

$i,j := 0,n;$
$\{\text{invariant: } 0 \le i \le m \wedge 0 \le j \le n \wedge x \in f[i{:}m, 0{:}j]\}$
$\{\text{bound function: } m-i+j\}$
do $f[i,j] > x \rightarrow j := j-1$
 $[]\ f[i,j] < x \rightarrow i := i+1$
od
$\{x = f[i,j]\}$

Now let us return to problem (1). For fixed b we must search the space of pairs (c, d), with c, d in the range defined by (1c,1d), for a pair satisfying (1a). Consider a two-dimensional array $f[b{:}(N-b)\div 3, b{:}(N-2*b)\div 2]$. The value $f[c,d]$ is $b*c*d*(N-b-c-d)$, the product of the components of the 4-tuple $(b, c, d, N-b-c-d)$. If each row and column of f is ordered, we can adapt Saddleback Search (to handle the case that $M \notin f$) to determine whether M is in f. This would reduce the time to search the pairs (c, d) for fixed b to $O(N)$, thus reducing the time for the complete algorithm to $O(N^2)$. Further, since each array element is a function of its subscripts, there is no need to maintain the array itself.

Array f does not have the desired property (of rows and columns being ordered), but the part of f corresponding to the search space does. To see this, first consider a fixed c in the range (1c). Then d must lie in the range (1d), over which the quadratic function $d*(N-b-c-d)$ in d is monotonically increasing, so the row-sections in the search space are ordered. Now consider a fixed d in the range $b \le d \le (N-2*b)\div 2$. Then c must lie in the range $b \le c \le \min\{d, (N-b-d)\div 2\}$. Since the quadratic function $c*(N-b-c-d)$ of c is monotonically increasing for $c \le (N-b-d)\div 2$, the column-sections in the search space are also ordered. Figure 0 below gives a rough picture of the search-space for a given b.

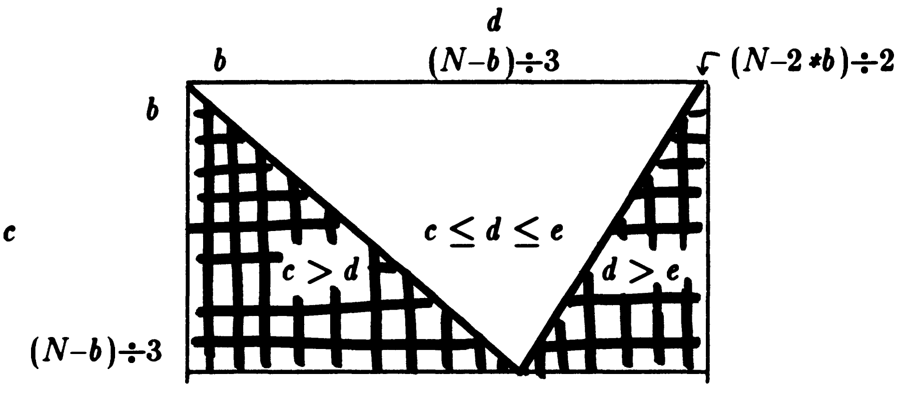

Figure 0: the search-space for a given b.

It is now simple to adapt Saddleback Search to the task at hand. We need only define the elements of virtual array f that are outside the search-space defined by (1c,1d) to be ∞. (These elements are represented by the shaded part of Fig. 0.) Then the full rectangular array has the property needed for Saddleback Search. The new quadratic algorithm is given below.

Algorithm 0:

```
{1 ≤ N ∧ 1 ≤ M ≤ (N/4)^4}
b, m:= 0, 0;
{invariant: P0 (see below)}
{bound function: N÷4 − b}
do m ≠ M ∧ b < N÷4 →
    b:= b+1;
    c, d:= b, (N−2*b)÷2;
    e:= N−b−c−d; m:= b*c*d*e;
    {invariant: P}
    {bound function: d−c}
    do m ≠ M ∧ c < d →
        if m > M ∨ d = e → d, e:= d−1, e+1
        [] m < M ∧ d < e → c, e:= c+1, e−1
        fi;
        m:= b*c*d*e
    od
od
{if m = M then (b, c, d, e) is an ordered solution, otherwise there is no solution}
```

Each iteration of the main loop searches the 4-tuples with first component b for a solution. Variable m is used to contain the value $b*c*d*e$. Invariant $P0$ of the main loop is

$$P0\colon 0 \le b \le N \div 4 \ \wedge \ m = M \Rightarrow (b,\ c,\ d,\ e) \text{ is an ordered solution } \wedge$$
$$m \ne M \Rightarrow \text{no ordered solution with first component } \le b \text{ exists.}$$

The bound function of the main loop is $N \div 4 - b$. Initialization $b, m := 0, 0$ establishes $P0$, and it remains only to show that the body of the main loop maintains it.

The inner loop searches for a solution among all 4-tuples with first component b using a modified Saddleback Search as discussed above. The modifications are quite straightforward: the guard of the **do** -statement causes the search to stop if M is found or if nothing remains because $c = d$; the extra conjuncts in the guards in the **if**-statement ensure that the search does not stray into the region $d > e$. Each iteration increases c or decreases d, at the same time changing e, m to maintain the invariant P:

P: (1b) $\wedge$ (1c) $\wedge$ (1d) $\wedge$ $e = N-b-c-d$ $\wedge$ $m = b*c*d*e$ $\wedge$
no solution with first component $< b$ exists $\wedge$
any ordered solution $(b, c', d', N-b-c'-d')$ satisfies $c \le c' \le d' \le d$.

5. Direct Searching

Saddleback Search is a useful algorithmic paradigm because of its generality—it is applicable to any matrix with ordered rows and columns. However we know much more about our virtual matrix f, and we now try to exploit this information.

In the original Saddleback Search, j is repeatedly decreased by 1 until $f[i,j] \le x$, and i is treated similarly. It would seem worthwhile to rewrite the algorithm using a binary search in each instance. This is not done because in the worst case the change is always by 1, so that binary searching would *increase* the worst-case complexity by a logarithmic factor.

But in the present context we can do better: because of the form of matrix f, we can compute the required row or column directly. Suppose $m > M$. Then there is an integer d' satisfying $0 \le d' < d$ and

$$b*c*d' *(N-b-c-d') \le M < b*c*(d'+1)*(N-b-c-(d'+1)).$$

(The graph of function $b*c*d*(N-b-c-d)$ is given in Fig.1.) Therefore d may be set to $\max(d', c)$, where d' is the integer part of the smaller (real) root of (2), since the larger root is $> (N-b-c) \div 2$.

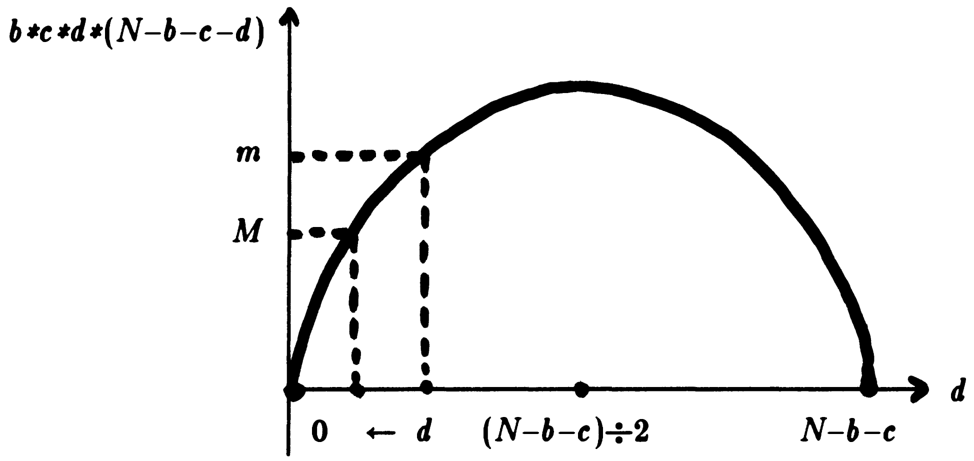

Figure 1: decreasing d when $m > M$.

If $m < M$, there are two cases to consider. The first is when $d = e$. In this case, increasing c would falsify the invariant, so the original code, which decreases d, is retained.

The other case is when $m < M$ and $d < e$. Then invariant P can be maintained by setting c to the smallest integer c' that satisfies one of the following:

(3) (a) $b*(c'-1)*d*(N-b-(c'-1)-d) < M \leq b*c'*d*(N-b-c'-d)$,
(b) $c' = d$,
(c) $d = N-b-c'-d$.

(3b) and (3c) ensure that the new c satisfies $c \leq d$ and $d \leq e$ respectively. If the discriminant δ of the quadratic

(4) $b*d*c^2 - b*d*(N-b-d)*c + M$

is negative, the quadratic has no real solution and the desired value c' is $\min(d,\ N-b-2*d)$. If the discriminant δ is non-negative, then the c' that satisfies (3a) is ceil(smaller root of (4)) (see Fig. 2).

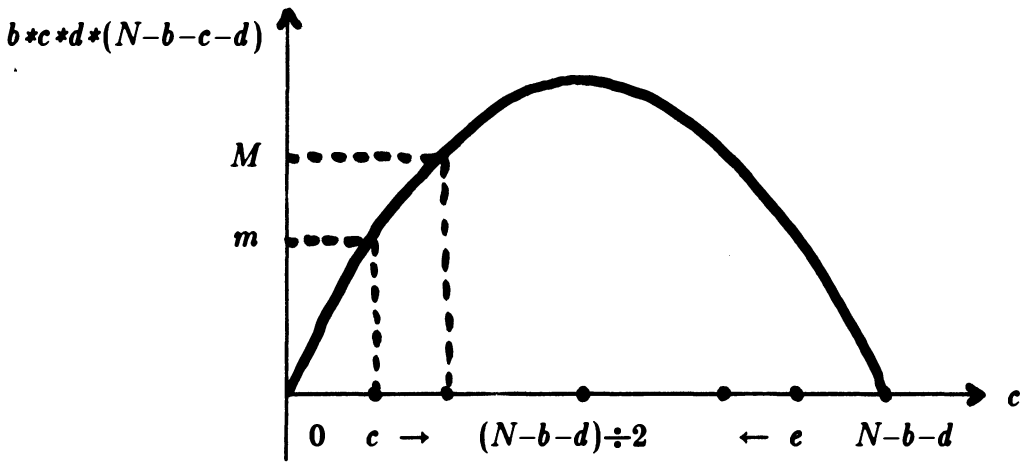

Figure 2: increasing c when $m < M \wedge d < e$.

The resulting algorithm 1 is given below. Although it is an improvement over algorithm 0 in many cases, it still has worst-case complexity $O(N^2)$. Part of the problem is that for many values of b there is no solution because $m < M$ (or $m > M$) for all values in the search space. In the next section we tackle the problem of ruling out such useless values of b before the search proper gets underway.

Algorithm 1:

Same as algorithm 0, but the body of the inner loop is replaced by

$$
\begin{array}{l}
\textbf{if } m > M \rightarrow \delta := \mathrm{sqr}(b*c*(N-b-c)) - 4*b*c*M; \\
\qquad\qquad d := (b*c*(N-b-c) - \mathrm{sqrt}(\delta)) \div (2*b*c);\ d := \max\{d, c\} \\
[]\ m < M \wedge d = e \rightarrow d := d-1 \\
[]\ m < M \wedge d < e \rightarrow \delta := \mathrm{sqr}(b*d*(N-b-d)) - 4*b*d*M; \\
\qquad\qquad \textbf{if } \delta \geq 0 \rightarrow c := \mathrm{ceil}((b*d*(N-b-d) - \mathrm{sqrt}(\delta)) / (2*b*d)); \\
\qquad\qquad\qquad\qquad c := \min\{c, d, N-b-2*d\} \\
\qquad\qquad []\ \delta < 0 \rightarrow c := \min\{d, N-b-2*d\} \\
\qquad\qquad \textbf{fi} \\
\textbf{fi}; \\
e := N-b-c-d;\ m := b*c*d*e
\end{array}
$$

6. Restricting the range of b

Let b be in the range (1b). Then the minimum value of m consistent with invariant P is $b^3(N-3*b)$ and the maximum such value is at most $b((N-b)/3)^3$ (it may be less because all variables are integers). Fig. 3 shows the graphs of the corresponding real functions of real variable b. Both functions have a turning point at $b = N/4$. The maximum function is concave; the minimum function is convex for $0 \leq b \leq N/6$ and concave for $N/6 \leq b \leq N/4$. The b-coordinates of the intersection of the line $y = M$ with these functions establish limits $b_{\min}$ and $b_{\max}$ for b.

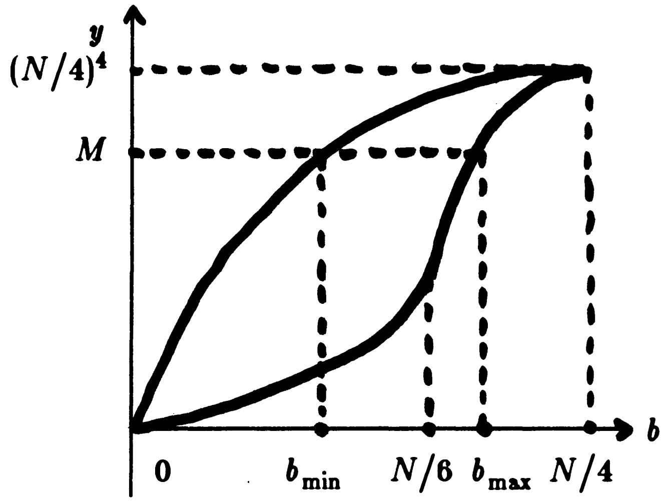

Figure 3: plots of minimum and maximum products against b.

To find the points $b_{\min}$ and $b_{\max}$, two quartic equations need to be solved. A suitable approach is to use Newton's method. Geometric arguments show that 0 is a safe starting point when finding $b_{\min}$, as is $N/6$ when finding $b_{\max}$ (under our assumption that $M \leq (N/4)^4$).

Requiring b to be between b_{min} and b_{max} dispenses with a great number of cases previously having quadratic complexity, but not with them all. We shall content ourselves with a rough argument for this fact. First note that there are values of M with $b_{max}-b_{min}=\Theta(N)$. Fig. 3 suggests this is the case for most $M \le (N/4)^4$, but it suffices to note that for $b_{min}=N/7$ we have $M=8N^4/2401$, but with $b=N/6$ we have $b^3(N-3b)=N^4/432$ and hence $b_{max}>N/6$.

Now consider a typical search for a given b in $[b_{min}, b_{max}]$. Fig. 4 below shows the search-space, with "contour lines" showing equal values of $m = b*c*d*(N-b-c-d)$ when c, d are regarded as real variables. The m-values increase along the line $c=d$ as c increases.

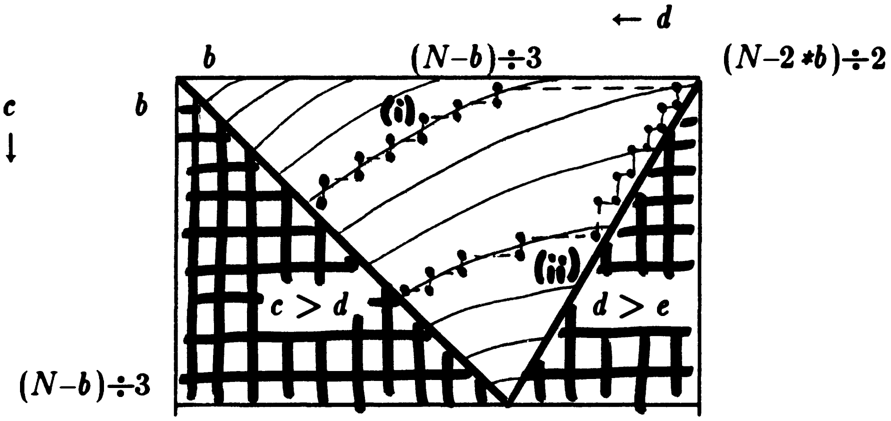

Figure 4: contour lines for the (c, d)-search-space.

To see that these contour lines give a true picture, consider points $P_1=(c, d)$, $P_2=(c-1, d)$ and $P_3=(c-1, d+1)$ in the search-space. The decrease in m in moving from $P1$ to $P2$ is $b*d*(e+1-c)$. The increase in m in moving from $P2$ to $P3$ is $b*(c-1)*(e-d)$. It follows that the contour lines are almost perpendicular to the boundary $c=d$ and gradually flatten to be almost horizontal near the boundary $d=e$.

Now it is apparent how a typical search for a given b proceeds. The test point (c, d) first moves from the upper right corner to the closest contour line, either directly by decreasing d (see search (i) in Fig. 4), or by gradually following the boundary $d=e$ (see search (ii) in Fig. 4). Then the test point moves along the contour, on alternating sides, until $m=M$ or $c=d$. Early steps may be large, but as the test point approaches the line $c=d$ the step-size approaches 1. So the typical search for a given b requires $\Theta(N)$ operations, and since $\Theta(N)$ values of b may need to be considered, the worst-case complexity is quadratic.

The above investigations show that direct search as in algorithm 1 is probably worthwhile for the first few moves, but that the payoff decreases rapidly thereafter, so that Saddleback Search may as well be used (since direct search does more work per move). We note that it is possible to move more quickly to a contour line that meets the boundary $d=e$ by using binary search along that boundary, because m increases along it.

7. Restricting the search to divisors

Thus far we have taken no account of a strong constraint, viz. that in any solution b, c, d and e are divisors of M. Sections 7 and 8 exploit this constraint.

We want to avoid as much as possible values of b, c, d, e that do not divide M. During a search with fixed b, $b_{\min} \le b \le c \le d \le e$ and $b+c+d+e=N$. Let

$$\begin{aligned} minb &= \text{ceil}(b_{\min}), \\ maxb &= \text{floor}(b_{\max}), \\ maxd &= (N-2*minb)\div 2. \end{aligned}$$

Then, during a search, all values of b, c and d are in the range $minb \mathinner{..} maxd$. Let $D(m,n)$ denote the number of divisors of m in the range $1 \mathinner{..} n$. We postulate the existence of array $v[0{:}k]$ defined as follows:

$$\begin{aligned} P_v\text{:}\ k &= D(M, maxd) - D(M, minb-1) \ \wedge \\ v[0] &= minb-1 \ \wedge \\ v[1{:}k] &= \text{the divisors of } M \text{ in the range } minb..maxd\text{, in order } \wedge \\ v[k+1] &= maxd+1. \end{aligned}$$

We will discuss generating array v later; for now, assume it exists. Then, instead of having b, c and d take on all integer values in a certain range, we need let them range only over the values in v. Below, we give a direct transformation of algorithm 0, using this idea. The algorithm does not use variables b, c and d; instead it uses variables ib, ic and id that are always in the subscript range of array v, and $b=v[ib]$, etc. The invariant $P0$ of the outer loop is

$$\begin{aligned} P0\text{: } & 0 \le ib \le k \ \wedge \\ & m=M \Rightarrow (v[ib], v[ic], v[id], e) \text{ is an ordered solution } \wedge \\ & m \ne M \Rightarrow \text{no ordered solution with first component} \le v[ib] \text{ exists,} \end{aligned}$$

and the invariant $P1$ of the inner loop is

$$\begin{aligned} P1\text{: } & 0 \le ib \le ic \le id \le k \ \wedge \\ & e = N-v[ib]-v[ic]-v[id] \ \wedge \ m = v[ib]*v[ic]*v[id]*e \ \wedge \\ & \text{no solution with first component} < v[ib] \text{ exists } \wedge \\ & \text{any ordered solution } (v[ib], c', d', N-v[ib]-c'-d') \text{ satisfies } v[ic] \le c' \le d' \le v[id]. \end{aligned}$$

Algorithm 2:

$\{1 \leq N \wedge 1 \leq M \leq (N/4)^4\}$
$minb, maxb := \text{ceil}(b_{\min}), \text{floor}(b_{\max})$;
$maxd := (N-2*minb) \div 2$;
Set v and k to establish P_v;
$ib, m := 0, 0$;
{invariant: $P0$}
{bound function: $k-ib$}
do $m \neq M \wedge v[ib+1] \leq maxb \rightarrow$
    $ib := ib+1$;
    $ic, id := ib, k$;
    $e := N-v[ib]-v[ic]-v[id]$;
    $m := v[ib]*v[ic]*v[id]*e$;
    {invariant: $P1$}
    {bound function: $id-ic$}
    do $(m \neq M \vee e < v[id]) \wedge ic < id \rightarrow$
        if $m > M \vee v[id] \geq e \rightarrow id := id-1$
        $[]$ $m < M \wedge v[id] < e \rightarrow ic := ic+1$
        fi;
        $e := N-v[ib]-v[ic]-v[id]$;
        $m := v[ib]*b[ic]*v[id]*e$
    od
od
{if $m = M$ then $(v[ib], v[ic], v[id], e)$ is an ordered solution, otherwise there is no solution}

The correctness proof for algorithm 2 is mainly a straightforward adaption of that for algorithm 0. But there is a fine point: the inequality $v[id] \leq e$ is not always true, and it must be shown to hold if $m = M$ on termination (so that the solution is ordered). To do this, first note that the inner loop of algorithm 2 terminates with

$$(m = M \wedge e \geq v[id]) \vee ic = id.$$

Therefore if the program terminates with $m = M$, either $e \geq v[id]$ or $e < v[id] = v[ic]$. But the latter condition cannot hold. For although $v[ic] \leq e$ might not be true before execution of the inner loop, once it becomes true it stays true thereafter. And it must become true because there can be no ordered solution with first component $< v[ib]$.

The above argument reveals a way to modify algorithm 2 to simplify the correctness proof. For $v[ic] \leq e$ will indeed be invariant if the initialization of id ensures $v[id] \leq e$ initially. To do this, the assignment of k to id need only be followed by

Establish $v[id] \leq (N-2*v[ib]) \div 2$:
  do $v[id] > (N-2*v[ib]) \div 2 \rightarrow id := id-1$ **od**;

(Such a modification is needed should the algorithm be required to generate *all* ordered solutions. It is omitted from our algorithm only to highlight the similar forms of algorithms 0 and 2.) The time required to initialize *id* can be reduced by using a binary search, and still further by noting that the search for the initial value of *id* can start from the (remembered) previous initial value.

Disregarding the first two statements, execution of algorithm 2 takes $O(D(M, N)^2)$ arithmetic operations. In [4] we find that

$$D(M, M) = O(M^{(1+\epsilon)\log 2/\log\log M})$$

for any $\epsilon > 0$, so that $D(M, M) = O(M^{\epsilon})$ for any $\epsilon > 0$. Since $M \le (N/4)^4$, algorithm 2 uses $O(N^{\epsilon})$ operations after initialization, for any $\epsilon > 0$.

8. Determining the divisors of M that are $\le N$

Array v can be initialized simply by sequencing through the values in the range $minb..maxd$ and storing in v those that divide M. This takes $O(maxd - minb) = O(N)$ operations and dominates the complexity of algorithm 2, so let us look for faster ways of initializing v.

Now, $v[1{:}k]$ consists of the divisors of M in a certain interval. These divisors can be efficiently generated from a partial prime factorization of M. The following algorithm finds such a factorization of M, generates the divisors, and finally sorts them. Arrays $p[0{:}r]$ and $n[0{:}r]$ define the partial factorization:

$$\begin{aligned} Q0\colon\ & 0 \le r \wedge p[r] = 1 \wedge n[r] = 1 \wedge \{(p[i], r[i]) \mid 0 \le i < r\} = \\ & \quad \{(x, y) \mid \mathrm{prime}(x) \wedge 1 \le y \wedge M \textbf{ mod } x^y = 0 \wedge x^y \le maxd \wedge \\ & \qquad (maxd < x^{y+1} \vee M \textbf{ mod } x^{y+1} \ne 0)\} \wedge \\ & \quad \text{the elements of } p[0{:}r-1] \text{ are distinct.} \end{aligned}$$

The algorithm searches through possible combinations of exponents of the primes $p[0{:}r-1]$, putting the corresponding products that are in range into v. During the search, variables f and $s[0{:}r]$ are defined as follows:

$$\begin{aligned} Q1\colon\ & (\forall i\colon 0 \le i \le r\colon s[i] \le n[i]) \wedge f = (\Pi\, i\colon 0 \le i < r\colon p[i]^{s[i]}) \wedge \\ & \quad (0, 0, \ldots, 0) \le \mathrm{reverse}(s[0{:}r]) \le (1, 0, \ldots, 0), \end{aligned}$$

where $<$ on tuples denotes lexicographic order. Thus, $s[0{:}r-1]$ represents a divisor f of M. The search through exponents given by $s[0{:}r-1]$ is done in increasing order of reverse($s[0{:}r]$). The invariant $Q2$ of the loop is

$$\begin{aligned} Q2\colon\ & Q0 \wedge Q1 \wedge \{v[i] \mid 1 \le i \le k\} = \{x \mid M \textbf{ mod } x = 0 \wedge \\ & \qquad (\exists t\colon t \text{ an } (r+1)\text{-tuple: } (0, \ldots, 0) \le \mathrm{reverse}(t) < \mathrm{reverse}(s[0{:}r]) \wedge \\ & \qquad\quad minb \le (\Pi\, i\colon 0 \le i < r\colon p[i]^{t[i]}) \le maxd)\} \wedge \\ & \quad \text{the elements of } v[1{:}k] \text{ are distinct.} \end{aligned}$$

Set v and k to establish P_v:
    Partially factorize M and set $p[r], n[r]$ to 1, thus establishing $Q0$;
    $\{Q0\}$
    $(\forall i: 0 \leq i \leq r: s[i] := 0)$;
    $f, k := 1, 1$;
    {invariant: $Q2$}
    {bound function: $(\mathbf{N}t: t$ an $(r+1)$-tuple $\wedge$ $(\Pi i: 0 \leq i \leq r: p[i]^{t[i]}) \leq maxd$ $\wedge$
        $(\forall i: 0 \leq i \leq r: t[i] \leq n[i])$ $\wedge$ reverse$(s[0:r]) \leq t < (1,0, \ldots, 0))$}
    do $s[r] \neq 1 \rightarrow$
        if $minb \leq f \rightarrow v[k+1], k := f, k+1$
        $[]$ $minb > f \rightarrow skip$
        fi;
        $j := 1$; **do** $s[j] = n[j] \vee f * p[j] > maxd \rightarrow s[j], f, j := 0, f * p[j]^{-s[j]}, j+1$ **od**;
        $s[j], f := s[j]+1, f * p[j]$
    od;
    sort $v[1:k]$;
    $v[0], v[k+1] := minb-1, maxd+1$
    $\{P_v\}$

The cost of generating the divisors is $O(D(M, maxd) * r) = O(D(M, N) * \log M)$ arithmetic operations, because $O(r)$ operations are sufficient to generate the next divisor $\leq maxd$. Note that the exponentiation in the innermost loop can be avoided by maintaining an array of the powers $p[i]^{s[i]}$, $0 \leq i < r$. The cost of the sort need only be $O(k * \log k) = O(D(M, N) * \log D(M, N))$ arithmetic operations.

It remains to refine the statement "Partially factorize M ..." and determine its complexity. Factorization is a difficult, classical problem. A good choice for the programmer who does not wish to go to extremes is the Monte Carlo method of Pollard as improved by Brent [0]. It would be expected to find all prime factors of M not greater than N in $O(N^{1/2+\epsilon})$ operations for any $\epsilon > 0$, and thus would dominate the complexity of algorithm 2. Unfortunately, however, as with many factoring methods, its complexity has not been rigorously determined. The fastest deterministic factoring algorithm whose complexity has been fully proved is the Pollard-Strassen method (see [5]). In our context, where only prime factors not greater than N are needed, it requires $O(N^{1/2+\epsilon})$ operations for any $\epsilon > 0$, so the complexity of algorithm 2 need be no greater than this. The Continued Fraction factoring method and its variants are shown in [5] to have complexities of the form $O(M^{\epsilon(M)})$, where $\epsilon(M) = c\sqrt{\log\log M / \log M}$ for some constant c, provided certain "reasonable" conjectures are true. If this is the case, then algorithm 2 can be implemented so as to have complexity $O(N^{\epsilon})$ for any $\epsilon > 0$, as claimed in the introduction. Moreover, Dixon's probabilistic factoring algorithm is proven without assumptions to have an expected (with overwhelmingly high probability) complexity of this form. Since $M \leq (N/4)^4$, it seems reasonable to claim that the complexity of our final algorithm is $O(N^{\epsilon})$ for any $\epsilon > 0$, a gigantic improvement on the naive algorithm.

Acknowledgements and history. Don Edwards of USNSW, Dahlgren, Virginia, brought the problem to our attention. The first two $O(N^2)$ solutions given in this paper previously appeared in [2]. A variant of the second solution that dispensed with all multiplications was added in [3]. By the way, the unique ordered solution to the original seven-eleven problem is (\$ 1.20, \$ 1.25, \$ 1.50, \$ 3.16).

References

[0] Brent, R.P. An improved Monte Carlo factorization algorithm. *B.I.T.* **20** (1980), 176-184.

[1] Gries, D. *The Science of Programming.* Springer Verlag, New York, 1981.

[2] Gries, D. The 711 problem. TR 82-493, Department of Computer Science, Cornell University, Ithaca, New York, May 1982.

[3] Gries, D., and J. Misra. The seven-eleven problem. Unpublished, undated MS, circa July 1982.

[4] Hardy, G.H., and E.M. Wright. *An Introduction to the Theory of Numbers.* 5th ed., Oxford University Press, Oxford, England, 1979.

[5] Pomerance, C. Analysis and comparison of some integer factoring algorithms. In *Computational Methods in Number Theory* (eds. Lenstra and Tijdeman), Math. Centrum, Amsterdam, 1983.