November 24, 2020 at 13:23

1. Intro. This program is a revision of ACHAIN2 (*not* ACHAIN3), which you should read first. After I found an unpublished preprint by D. Bleichenbacher and A. Flammenkamp (1997) on the web, I realized that several changes would speed that program up significantly.

The main changes here are: (i) Links are maintained so that it's easy to skip past cases with large $l[p]$. (ii) When an odd number p is inserted into the chain at position j, we make sure that $p \le \min(b[j-2]+b[j-1], b[j])$. Previously the weaker test $p \le b[j]$ was used. (iii) Whenever we find a good chain for n, we update the upper bounds for larger numbers, in case this chain implies a better way to compute them than was previously known. (The factor method, used previously, is just a special case of this technique.)

One change I intentionally did *not* make: When trying to make $a[s]$ be equal to $p+q$ for some previous values p and q, Flammenkamp and Bleichenbacher check to see whether appropriate p and q are already present; if so, they accept $a[s]$ and move on. Very plausible. But they don't implement a strong equivalence test with canonical chains, as I do; and I have not been able to verify that their "move on" heuristic is justifiable together with the strong cutoffs in my canonical approach, because of subtle ambiguities that arise in special cases.

```
#define nmax 10000000 /* should be less than 2^24 on a 32-bit machine */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
  unsigned char l[nmax + 2];
  int a[128], b[128];
  unsigned int undo[128 * 128];
  int ptr; /* this many items of the undo stack are in use */
  struct {
    int lbp, ubp, lbq, ubq, r, ptrp, ptrq;
  } stack[128];
  int tail[128], outdeg[128], outsum[128], limit[128];
  FILE *infile, *outfile;
  int down[nmax]; /* a navigation aid discussed below */
  int link[nmax]; /* stack links when propagating upper bounds */
  int top; /* top element of the stack (or 1 when empty) */
  int main(int argc, char *argv[])
  {
    register int i, j, n, p, q, r, s, ubp, ubq = 0, lbp, lbq, ptrp, ptrq;
    int lb, ub, timer = 0;
    ⟨ Process the command line 2 ⟩;
    a[0] = b[0] = 1, a[1] = b[1] = 2; /* an addition chain always begins like this */
    ⟨ Initialize the l and down tables 5 ⟩;
    for (n = 1; n < nmax; n++) {
      ⟨ Input the next lower bound, lb 4 ⟩;
      ⟨ Backtrack until l(n) is known 8 ⟩;
      if (n % 1000 ≡ 0) {
        j = clock( );
        printf("%d..%d done in %.5g minutes\n", n - 999, n,
            (double)(j - timer)/(60 * CLOCKS_PER_SEC));
        timer = j;
      }
    done: ⟨ Update the down links 11 ⟩;
      ⟨ Output the value of l(n) 3 ⟩;
    }
  }
```

2. ⟨ Process the command line 2 ⟩ ≡

```
if (argc ≠ 3) {
  fprintf(stderr, "Usage: %s infile outfile\n", argv[0]);
  exit(−1);
}
infile = fopen(argv[1], "r");
if (¬infile) {
  fprintf(stderr, "I couldn't open `%s' for reading!\n", argv[1]);
  exit(−2);
}
outfile = fopen(argv[2], "w");
if (¬outfile) {
  fprintf(stderr, "I couldn't open `%s' for writing!\n", argv[2]);
  exit(−3);
}
```

This code is used in section 1.

3. ⟨ Output the value of $l(n)$ 3 ⟩ ≡

```
fprintf(outfile, "%c", l[n] + ' ');
fflush(outfile);      /* make sure the result is viewable immediately */
```

This code is used in section 1.

4. At this point I compute the known lower bound $\lfloor \lg n \rfloor + \lceil \lg \min(\nu n, 16) \rceil$.

(With a change file, I could make the program set $l[n] = lb$ and **goto** *done* if the input value is ' ' or more. This change would amount to believing that the input file has true values, thereby essentially restarting a computation that was only partly finished. Hmm, wait: Actually I should also use the factor method to update upper bounds, before going to *done*, if $l[n]$ has changed.)

⟨ Input the next lower bound, lb 4 ⟩ ≡

```
for (q = n, i = −1, j = 0; q; q >>= 1, i++) j += q & 1;      /* now i = ⌊lg n⌋ and j = νn */
if (j > 16) j = 16;
for (j−−; j; j >>= 1) i++;
lb = fgetc(infile) − ' ';      /* fgetc will return a negative value after EOF */
if (lb < i) lb = i;
```

This code is used in section 1.

5. Initially we want to fill the l table with fairly decent upper bounds. Here I use Theorem 4.6.3F, which includes the binary method as a special case; namely, $l(2^A+xy)\le A+\nu x+\nu y-1$, when $A\ge\lfloor\lg x\rfloor+\lfloor\lg y\rfloor$. Then I also apply the factor method.

The *down* table temporarily holds νn values here. But then, as explained later, we want $down[n]=n-1$ initially.

```
⟨ Initialize the l and down tables 5 ⟩ ≡
  for (n = 2, down[1] = 1; n < nmax; n++) down[n] = down[n ≫ 1] + (n & 1), l[n] = 127;
  for (i = 1, p = 0; i * i < nmax; i++, p = (i ≡ 1 ≪ (p + 1) ? p + 1 : p)) {      /* p = ⌊i⌋ */
    for (j = i, q = p; i * j < nmax; j++, q = (j ≡ 1 ≪ (q + 1) ? q + 1 : q)) {      /* q = ⌊j⌋ */
      for (r = p + q; (1 ≪ r) + i * j < nmax; r++)
        if (l[(1 ≪ r) + i * j] ≥ r + down[i] + down[j]) l[(1 ≪ r) + i * j] = r + down[i] + down[j] − 1;
            /* Hansen's method */
    }
  }
  for (n = 1; n < nmax; n++) down[n] = n − 1;
  ⟨ Apply the factor method 6 ⟩;
```

This code is used in section 1.

6. Whenever we learn a better upper bound, we might as well broadcast all of its consequences, using the factor method. (The total number of times this happens for a particular number n is at most $\lg n$, and the time to propagate is proportional to $nmax/n$, so the total time for all these updates is at most $O(\log n)^2$ times $nmax$.)

A linked stack is used to handle these updates. An element p is on the stack if and only if $link[p]\neq 0$, if and only $l[p]$ has decreased since the last time we checked all of its multiples.

```
#define upbound(p, x)
        if (l[p] > x) {
          l[p] = x;
          if (link[p] ≡ 0) link[p] = top, top = p;
        }
⟨ Apply the factor method 6 ⟩ ≡
  top = 1;      /* start with empty stack */
  for (i = 4; i < nmax; i++) {
    upbound(i, l[i − 2] + 1);
    upbound(i, l[i − 1] + 1);
  }
  for (i = 2; i * i < nmax; i++)
    for (j = i; i * j < nmax; j++) upbound(i * j, l[i] + l[j]);
  while (top > 1) {
    p = top, top = link[p], link[p] = 0;
    upbound(p + 1, l[p] + 1);
    upbound(p + 2, l[p] + 2);
    for (i = 2; i * p < nmax; i++) upbound(i * p, l[i] + l[p]);
  }
```

This code is used in section 5.

7. Later, whenever we've come up with an addition chain $a[0], \ldots, a[lb]$ that beats the known upper bound, we use the factor method again to send out the good news. And we also might as well use an extension of the factor method found in equation (4.3) in the Bleichenbacher/Flammenkamp paper.

At the beginning of this step, $top = 1$. We've allocated space for $l[nmax]$ and $l[nmax + 1]$ in order to avoid making a special test here.

```
⟨ Update the upper bounds based on the chain found 7 ⟩ ≡
  for (j = 1; j * n < nmax; j++) upbound(j * n, l[j] + l[n]);
  for (i = 0; i < lb; i++)
    for (j = 1; j * n + a[i] < nmax; j++) upbound(j * n + a[i], l[j] + l[n] + 1);      /* B and F's method */
  while (top > 1) {
    p = top, top = link[p], link[p] = 0;
    upbound(p + 1, l[p] + 1);
    upbound(p + 2, l[p] + 2);
    for (i = 2; i * p < nmax; i++) upbound(i * p, l[i] + l[p]);
  }
```

This code is used in section 12.

8. The interesting part. Nontrivial changes begin to occur when we get into the guts of the backtracking structure carried over from the previous versions of this program, but the controlling loop at the outer level remains intact.

```
⟨ Backtrack until l(n) is known 8 ⟩ ≡
  ub = l[n], l[n] = lb;
  while (lb < ub) {
    for (i = 0; i ≤ lb; i++) outdeg[i] = outsum[i] = 0;
    a[lb] = b[lb] = n;
    for (i = 2; i < lb; i++) a[i] = a[i − 1] + 1, b[i] = b[i − 1] ≪ 1;
    for (i = lb − 1; i ≥ 2; i−−) {
      if ((a[i] ≪ 1) < a[i + 1]) a[i] = (a[i + 1] + 1) ≫ 1;
      if (b[i] ≥ b[i + 1]) b[i] = b[i + 1] − 1;
    }
    ⟨ Try to fix the rest of the chain; goto backtrackdone if it's possible 12 ⟩;
    l[n] = ++lb;
  }
  backtrackdone:
```

This code is used in section 1.

9. One of the key operations we need is to increase p to the smallest element $p' > p$ that has $l[p'] < s$, given that $l[p] < s$. Since $l[p+1] \le l[p]+1$, we can do this quickly by first setting $p \leftarrow p+1$; then, if $l[p] = s$, we set $p \leftarrow down[p]$, where $down[p]$ is the smallest $p' > p$ that has $l[p'] < l[p]$.

The links $down[p]$ can be prepared as we go, starting them off at ∞ and updating them whenever we learn a new value of $l[n]$.

Instead of using infinite links, however, we can save space by temporarily letting $down[p] = p''$ in such cases, where p'' is the largest element *less than* p whose *down* link is effectively infinite. These temporary links tell us exactly what we need to know during the updating process. And we can distinguish them from "real" *down* links by pretending that $down[p] = \infty$ whenever $down[p] \le p$.

```
⟨ Given that l[p] < s, increase p to the next such element 9 ⟩ ≡
  {
    p++;
    if (l[p] ≡ s) p = (down[p] > p ? down[p] : nmax);
  }
```

This code is used in section 14.

10. ⟨ Given that $l[p] \ge s$, increase p to the next element with $l[p] < s$ 10 ⟩ ≡

```
  do {
    if (down[p] > p) p = down[p];
    else {
      p = nmax; break;
    }
  } while (l[p] ≥ s);
```

This code is used in sections 13 and 14.

11. I can't help exclaiming that this little algorithm is quite pretty.

```
⟨ Update the down links 11 ⟩ ≡
  if (l[n] < l[n − 1]) {
    for (p = down[n]; l[p] > l[n]; p = q) q = down[p], down[p] = n;
    down[n] = p;
  }
```

This code is used in section 1.

12. We maintain a stack of subproblems, and a stack for undoing, as in ACHAIN2 and its predecessors.

```
⟨ Try to fix the rest of the chain; goto backtrackdone if it's possible 12 ⟩ ≡
  ptr = 0;      /* clear the undo stack */
  for (r = s = lb; s > 2; s--) {
    if (outdeg[s] ≡ 1) limit[s] = a[s] − tail[outsum[s]]; else limit[s] = a[s] − 1;
        /* the max feasible p */
    if (limit[s] > b[s − 1]) limit[s] = b[s − 1];
    ⟨ Set p to its smallest feasible value, and q = a[s] − p 13 ⟩;
    while (p ≤ limit[s]) {
      ⟨ Find bounds (lbp, ubp) and (lbq, ubq) on where p and q can be inserted; but go to failpq if they
          can't both be accommodated 17 ⟩;
      ptrp = ptr;
      for ( ; ubp ≥ lbp; ubp--) {
        ⟨ Put p into the chain at location ubp; goto failp if there's a problem 19 ⟩;
        if (p ≡ q) goto happiness;
        if (ubq ≥ ubp) ubq = ubp − 1;
        ptrq = ptr;
        for ( ; ubq ≥ lbq; ubq--) {
          ⟨ Put q into the chain at location ubq; goto failq if there's a problem 21 ⟩;
        happiness: ⟨ Put local variables on the stack and update outdegrees 15 ⟩;
          goto onward;      /* now a[s] is covered; try to cover a[s − 1] */
        backup: s++;
          if (s > lb) goto impossible;
          ⟨ Restore local variables from the stack and downdate outdegrees 16 ⟩;
          if (p ≡ q) goto failp;
        failq: while (ptr > ptrq) ⟨ Undo a change 18 ⟩;
        }      /* end loop on ubq */
      failp: while (ptr > ptrp) ⟨ Undo a change 18 ⟩;
      }      /* end loop on ubp */
    failpq: ⟨ Advance p to the next smallest feasible value, and set q = a[s] − p 14 ⟩;
    }      /* end loop on p */
    goto backtrackdone;
  onward: continue;
  }      /* end loop on s */
  ⟨ Update the upper bounds based on the chain found 7 ⟩;
  goto backtrackdone;
impossible:
```

This code is used in section 8.

13. At this point we have $a[k] = b[k]$ for all $r \le k \le lb$.

```
⟨ Set p to its smallest feasible value, and q = a[s] − p 13 ⟩ ≡
  if (a[s] & 1) {      /* necessarily p ≠ q */
  unequal: if (outdeg[s − 1] ≡ 0) q = a[s]/3; else q = a[s] ≫ 1;
    if (q > b[s − 2]) q = b[s − 2];
    p = a[s] − q;
    if (l[p] ≥ s) {
      ⟨ Given that l[p] ≥ s, increase p to the next element with l[p] < s 10 ⟩;
      q = a[s] − p;
    }
  } else {
    p = q = a[s] ≫ 1;
    if (l[p] ≥ s) goto unequal;      /* a rare case like l[191] = l[382] */
  }
  if (p > limit[s]) goto backup;
  for ( ; r > 2 ∧ a[r − 1] ≡ b[r − 1]; r−−) ;
  if (p > b[r − 1]) {      /* now r < s, since p ≤ b[s − 1] */
    while (p > a[r]) r++;      /* this step keeps r < s, since a[s − 1] = b[s − 1] */
    p = a[r], q = a[s] − p;
  } else if (q < p ∧ q > b[r − 2]) {
    if (a[r] ≤ a[s] − b[r − 2]) p = a[r], q = b[s] − p;
    else q = b[r − 2], p = a[s] − q;
  }
```

This code is used in section 12.

14. ⟨ Advance p to the next smallest feasible value, and set $q = a[s] - p$ 14 ⟩ ≡

```
if (p ≡ q) {
  if (outdeg[s − 1] ≡ 0) q = (a[s]/3) + 1;      /* will be decreased momentarily */
  if (q > b[s − 2]) q = b[s − 2]; else q−−;
  p = a[s] − q;
  if (l[p] ≥ s) {
    ⟨ Given that l[p] ≥ s, increase p to the next element with l[p] < s 10 ⟩;
    q = a[s] − p;
  }
} else {
  ⟨ Given that l[p] < s, increase p to the next such element 9 ⟩;
  q = a[s] − p;
}
if (q > 2) {
  if (a[s − 1] ≡ b[s − 1]) {      /* maybe p has to be present already */
  doublecheck: while (p < a[r] ∧ a[r − 1] ≡ b[r − 1]) r−−;
    if (p > b[r − 1]) {
      while (p > a[r]) r++;
      p = a[r], q = a[s] − p;      /* possibly r = s now */
    } else if (q > b[r − 2]) {
      if (a[r] ≤ a[s] − b[r − 2]) p = a[r], q = b[s] − p;
      else q = b[r − 2], p = a[s] − q;
    }
  }
  if (ubq ≥ s) ubq = s − 1;
  while (q ≥ a[ubq + 1]) ubq++;
  while (q < a[ubq]) ubq−−;
  if (q > b[ubq]) {
    q = b[ubq], p = a[s] − q;
    if (a[s − 1] ≡ b[s − 1]) goto doublecheck;
  }
}
```

This code is used in section 12.

15. ⟨ Put local variables on the stack and update outdegrees 15 ⟩ ≡

```
tail[s] = q, stack[s].r = r;
outdeg[ubp]++, outsum[ubp] += s;
outdeg[ubq]++, outsum[ubq] += s;
stack[s].lbp = lbp, stack[s].ubp = ubp;
stack[s].lbq = lbq, stack[s].ubq = ubq;
stack[s].ptrp = ptrp, stack[s].ptrq = ptrq;
```

This code is used in section 12.

16. ⟨ Restore local variables from the stack and downdate outdegrees 16 ⟩ ≡

```
ptrq = stack[s].ptrq, ptrp = stack[s].ptrp;
lbq = stack[s].lbq, ubq = stack[s].ubq;
lbp = stack[s].lbp, ubp = stack[s].ubp;
outdeg[ubq]−−, outsum[ubq] −= s;
outdeg[ubp]−−, outsum[ubp] −= s;
q = tail[s], p = a[s] − q, r = stack[s].r;
```

This code is used in section 12.

17. After the test in this step is passed, we'll have $ubp > ubq$ and $lbp > lbq$.

```
⟨ Find bounds (lbp, ubp) and (lbq, ubq) on where p and q can be inserted; but go to failpq if they can't both
      be accommodated 17 ⟩ ≡
  if (l[p] ≥ s) goto failpq;
  lbp = l[p];
  while (b[lbp] < p) lbp++;
  if ((p & 1) ∧ p > b[lbp − 2] + b[lbp − 1]) {
    if (++lbp ≥ s) goto failpq;
  }
  if (a[lbp] > p) goto failpq;
  for (ubp = lbp; a[ubp + 1] ≤ p; ubp++) ;
  if (ubp ≡ s − 1) lbp = ubp;
  if (p ≡ q) lbq = lbp, ubq = ubp;
  else {
    lbq = l[q];
    if (lbq ≥ ubp) goto failpq;
    while (b[lbq] < q) lbq++;
    if (a[lbq] < b[lbq]) {
      if ((q & 1) ∧ q > b[lbq − 2] + b[lbq − 1]) lbq++;
      if (lbq ≥ ubp) goto failpq;
      if (a[lbq] > q) goto failpq;
      if (lbp ≤ lbq) lbp = lbq + 1;
      while ((q ≪ (lbp − lbq)) < p)
        if (++lbp > ubp) goto failpq;
    }
    for (ubq = lbq; a[ubq + 1] ≤ q ∧ (q ≪ (ubp − ubq − 1)) ≥ p; ubq++) ;
  }
```

This code is used in section 12.

18. The undoing mechanism is very simple: When changing $a[j]$, we put $(j \ll 24) + x$ on the *undo* stack, where x was the former value. Similarly, when changing $b[j]$, we stack the value $(1 \ll 31) + (j \ll 24) + x$.

```
#define newa(j, y) undo[ptr++] = (j ≪ 24) + a[j], a[j] = y
#define newb(j, y) undo[ptr++] = (1 ≪ 31) + (j ≪ 24) + b[j], b[j] = y
⟨ Undo a change 18 ⟩ ≡
  {
    i = undo[−−ptr];
    if (i ≥ 0) a[i ≫ 24] = i & #ffffff;
    else b[(i & #3fffffff) ≫ 24] = i & #ffffff;
  }
```

This code is used in section 12.

19. At this point we know that $a[ubp] \le p \le b[ubp]$.

```
⟨ Put p into the chain at location ubp; goto failp if there's a problem 19 ⟩ ≡
  if (a[ubp] ≠ p) {
    newa(ubp, p);
    for (j = ubp − 1; (a[j] ≪ 1) < a[j + 1]; j−−) {
      i = (a[j + 1] + 1) ≫ 1;
      if (i > b[j]) goto failp;
      newa(j, i);
    }
    for (j = ubp + 1; a[j] ≤ a[j − 1]; j++) {
      i = a[j − 1] + 1;
      if (i > b[j]) goto failp;
      newa(j, i);
    }
  }
  if (b[ubp] ≠ p) {
    newb(ubp, p);
    for (j = ubp − 1; b[j] ≥ b[j + 1]; j−−) {
      i = b[j + 1] − 1;
      if (i < a[j]) goto failp;
      newb(j, i);
    }
    for (j = ubp + 1; b[j] > b[j − 1] ≪ 1; j++) {
      i = b[j − 1] ≪ 1;
      if (i < a[j]) goto failp;
      newb(j, i);
    }
  }
  ⟨ Make forced moves if p has a special form 20 ⟩;
```

This code is used in section 12.

20. If, say, we've just set $a[8] = b[8] = 132$, special considerations apply, because the only addition chains of length 8 for 132 are

$$1, 2, 4, 8, 16, 32, 64, 128, 132;$$
$$1, 2, 4, 8, 16, 32, 64, 68, 132;$$
$$1, 2, 4, 8, 16, 32, 64, 66, 132;$$
$$1, 2, 4, 8, 16, 32, 34, 66, 132;$$
$$1, 2, 4, 8, 16, 32, 33, 66, 132;$$
$$1, 2, 4, 8, 16, 17, 33, 66, 132.$$

The values of $a[4]$ and $b[4]$ must therefore be 16; and then, of course, we also must have $a[3] = b[3] = 8$, etc. Similar reasoning applies whenever we set $a[j] = b[j] = 2^j + 2^k$ for $k \le j - 4$.

Such cases may seem extremely special. But my hunch is that they are important, because efficient chains need such values. When we try to prove that no efficient chain exists, we want to show that such values can't be present. Numbers with small $l[p]$ are harder to rule out, so it should be helpful to penalize them.

```
⟨ Make forced moves if p has a special form 20 ⟩ ≡
  i = p − (1 ≪ (ubp − 1));
  if (i ∧ ((i & (i − 1)) ≡ 0) ∧ (i ≪ 4) < p) {
    for (j = ubp − 2; (i & 1) ≡ 0; i ≫= 1, j−−) ;
    if (b[j] < (1 ≪ j)) goto failp;
    for ( ; a[j] < (1 ≪ j); j−−) newa(j, 1 ≪ j);
  }
```

This code is used in section 19.

21. At this point we had better not assume that $a[ubq] \le q \le b[ubq]$, because p has just been inserted. That insertion can mess up the bounds that we looked at when lbq and ubq were computed.

```
⟨ Put q into the chain at location ubq; goto failq if there's a problem 21 ⟩ ≡
  if (a[ubq] ≠ q) {
    if (a[ubq] > q) goto failq;
    newa(ubq, q);
    for (j = ubq − 1; (a[j] ≪ 1) < a[j + 1]; j−−) {
      i = (a[j + 1] + 1) ≫ 1;
      if (i > b[j]) goto failq;
      newa(j, i);
    }
    for (j = ubq + 1; a[j] ≤ a[j − 1]; j++) {
      i = a[j − 1] + 1;
      if (i > b[j]) goto failq;
      newa(j, i);
    }
  }
  if (b[ubq] ≠ q) {
    if (b[ubq] < q) goto failq;
    newb(ubq, q);
    for (j = ubq − 1; b[j] ≥ b[j + 1]; j−−) {
      i = b[j + 1] − 1;
      if (i < a[j]) goto failq;
      newb(j, i);
    }
    for (j = ubq + 1; b[j] > b[j − 1] ≪ 1; j++) {
      i = b[j − 1] ≪ 1;
      if (i < a[j]) goto failq;
      newb(j, i);
    }
  }
  ⟨ Make forced moves if q has a special form 22 ⟩;
```

This code is used in section 12.

22.

```
⟨ Make forced moves if q has a special form 22 ⟩ ≡
  i = q − (1 ≪ (ubq − 1));
  if (i ∧ ((i & (i − 1)) ≡ 0) ∧ (i ≪ 4) < q) {
    for (j = ubq − 2; (i & 1) ≡ 0; i ≫= 1, j−−) ;
    if (b[j] < (1 ≪ j)) goto failq;
    for ( ; a[j] < (1 ≪ j); j−−) newa(j, 1 ≪ j);
  }
```

This code is used in section 21.

23. Index.

ACHAIN4