(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Intro. This program is a revision of ACHAIN1, which you should read first. I'm thinking that a few changes will speed that program up, but as usual the proof of the pudding is in the eating.

The main changes here are: (i) If $a[j] = b[j] = 2^j + 2^k$, where $k < j - 3$, one can prove that $a[j-k-2]$ and $b[j-k-2]$ must both be 2^{j-k-2}. This additional constraint makes more chain values "exist" at early stages. (ii) Every addition chain corresponds to a directed graph, and many different addition chains can correspond to the same "reduced" digraph, as explained at the end of Section 4.6.3. Therefore I've worked out a scheme by which only one chain of each equivalence class is explored.

These changes, and the various changes that distinguish ACHAIN1 from ACHAIN0, are fairly independent. If I had time, I could therefore experiment with various subsets, in order to see which of them are really worth the effort. Who knows, maybe some of them actually cause a slowdown, taken individually.

```
#define nmax 10000000   /* should be less than 2^24 on a 32-bit machine */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
  char l[nmax];
  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 prime[1000];   /* 1000 primes will take us past 60 million */
  int pr;   /* the number of primes known so far */
  char x[64];   /* exponents of the binary representation of n, less 1 */
  int main(int argc, char *argv[])
  {
    register int i, j, n, p, q, r, s, ubp, ubq, lbp, lbq, ptrp, ptrq;
    int lb, ub, timer = 0;
    ⟨Process the command line 2⟩;
    prime[0] = 2, pr = 1;
    a[0] = b[0] = 1, a[1] = b[1] = 2;   /* an addition chain always begins like this */
    for (n = 1; n < nmax; n++) {
      ⟨Input the next lower bound, lb 4⟩;
      ⟨Find an upper bound; or in simple cases, set l(n) and goto done 5⟩;
      ⟨Backtrack until l(n) is known 7⟩;
    done: ⟨Output the value of l(n) 3⟩;
      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;
      }
    }
  }
```

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 "lower bound" $\lfloor \lg n \rfloor + 3$, which is valid if $\nu n > 4$. Simple cases where $\nu n \le 4$ will be handled separately below.

⟨ Input the next lower bound, *lb* 4 ⟩ ≡

```
for (q = n, i = −1, j = 0; q; q >>= 1, i++)
  if (q & 1) x[j++] = i;   /* now i = ⌊lg n⌋ and j = νn */
lb = fgetc(infile) − '␣';   /* fgetc will return a negative value after EOF */
if (lb < i + 3) lb = i + 3;
```

This code is used in section 1.

5. Three elementary and well-known upper bounds are considered: (i) $l(n) \le \lfloor \lg n \rfloor + \nu n - 1$; (ii) $l(n) \le l(n-1)+1$; (iii) $l(n) \le l(p) + l(q)$ if $n = pq$.

Furthermore, there are four special cases when Theorem 4.6.3C tells us we can save a step. In this regard, I had to learn (the hard way) to avoid a curious bug: Three of the four cases in Theorem 4.6.3C arise when we factor n, so I thought I needed to test only the other case here. But I got a surprise when $n = 165$: Then $n = 3 \cdot 55$, so the factor method gave the upper bound $l(3)+l(55) = 10$; but another factorization, $n = 5 \cdot 33$, gives the better bound $l(5) + l(33) = 9$.

⟨ Find an upper bound; or in simple cases, set $l(n)$ and **goto** *done* 5 ⟩ ≡

```
ub = i + j − 1;
if (ub > l[n − 1] + 1) ub = l[n − 1] + 1;
⟨ Try reducing ub with the factor method 6 ⟩;
l[n] = ub;
if (j ≤ 3) goto done;
if (j ≡ 4) {
  p = x[3] − x[2], q = x[1] − x[0];
  if (p ≡ q ∨ p ≡ q + 1 ∨ (q ≡ 1 ∧ (p ≡ 3 ∨ (p ≡ 5 ∧ x[2] ≡ x[1] + 1)))) l[n] = i + 2;
  goto done;
}
```

This code is used in section 1.

6. It's important to try the factor method even when $j \le 4$, because of the way prime numbers are recognized here: We would miss the prime 3, for example.

On the other hand, we don't need to remember large primes that will never arise as factors of any future n.

⟨ Try reducing ub with the factor method 6 ⟩ ≡

```
if (n > 2)
  for (s = 0; ; s++) {
    p = prime[s];
    q = n/p;
    if (n ≡ p * q) {
      if (l[p] + l[q] < ub) ub = l[p] + l[q];
      break;
    }
    if (q ≤ p) {      /* n is prime */
      if (pr < 1000) prime[pr++] = n;
      break;
    }
  }
```

This code is used in section 5.

7. 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 7 ⟩ ≡

```
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 done if it's possible 9 ⟩;
  l[n] = ++lb;
}
```

This code is used in section 1.

8. The main new idea implemented below is related to the reduced digraph representation of an addition chain, in which each element a_s of the chain is effectively expressed as a sum of two or more previous elements. For example, we might have $a_s = a_i + a_j + a_k + a_l$ where $i \le j \le k \le l$, represented by four arrows coming in to node a_s from nodes a_i, a_j, a_k, and a_l. The colexicographically largest chain with this reduced digraph has elements $a_k + a_l$, $a_j + a_k + a_l$, and $a_i + a_j + a_k + a_l$, and we want to avoid redundant work by restricting consideration to such chains.

Fortunately there's an easy way to do this: For each chain element x we remember its "tail," which is the smallest node that points to x; and we also keep track of the outdegree of each fixed element. If $a_s = p + q$ with $p \ge q$, the tail of a_s is q. And if a_s has outdegree 1 because it is used only as an input to a_t, we insist that q be at least as large as the tail of a_t.

Array element *outdeg*$[j]$ is the current number of elements a_{s+1}, a_{s+2}, …, that make use of a_j, and array element *outsum*$[j]$ is the sum of all their subscripts. These two arrays are easily updated as we move forward and backward while backtracking; and *outsum*$[j]$ turns out to be exactly the value t that we need to know when *outdeg*$[j] = 1$. (Has anybody seen this idea before? At my age I thought I had seen all such simple tricks long ago, but maybe there still are many more waiting to be discovered.)

(Note added 08 Oct 2005: I just found this idea attributed to David Wood, by Eric Bach and Marcos Kiwi in *Theoretical Computer Science* **235** (2000), 5. Bach and Kiwi go on to generalize the idea so that more than one element can be identified, using power sums.)

One consequence: If $\mathit{outdeg}[s-1] \equiv 0$, we must either have $a[s-1] = \frac{1}{2}a[s]$ or $a[s-1] \ge \frac{2}{3}a[s]$. Because after setting $a[s]$ to a value other than $\frac{1}{2}a[s]$ we will have $\mathit{outdeg}[s-1] = 1$, and $\frac{1}{2}a[s-1] \ge \mathit{tail}[s-1] \ge \mathit{tail}[s] = a[s] - a[s-1]$.

9. We maintain a stack of subproblems, as usual when backtracking. Suppose $a[t]$ is the sum of two items already present, for all $t > s$; we want to make sure that $a[s]$ is legitimate too. For this purpose we try all combinations $a[s] = p + q$ where $p \ge a[s]/2$, trying to make both p and q present. (By the nature of the algorithm, we'll have $a[s] = b[s]$ at the time we choose p and q, because shorter addition chains have been ruled out.)

As elements of a and b are changed, we record their previous values on the *undo* stack, so that we can easily restore them later. Pointers *ptrp* and *ptrq* contain the limiting indexes for undo information.

```
⟨Try to fix the rest of the chain; goto done if it's possible 9⟩ ≡
  ptr = 0;    /* clear the undo stack */
  for (r = s = lb; s > 2; s−−) {
    if (outdeg[s] ≡ 1) limit[s] = tail[outsum[s]]; else limit[s] = 1;
    for ( ; r > 1 ∧ a[r − 1] ≡ b[r − 1]; r−−) ;
    if (outdeg[s − 1] ≡ 0 ∧ (a[s] & 1)) q = a[s]/3; else q = a[s] ≫ 1;
    for (p = a[s] − q; p ≤ b[s − 1]; ) {
      if (p > b[r − 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, r++;
      }
      if (q < limit[s]) goto backup;
      ⟨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 12⟩;
      ptrp = ptr;
      for ( ; ubp ≥ lbp; ubp−−) {
        ⟨Put p into the chain at location ubp; goto failp if there's a problem 14⟩;
        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 16⟩;
        happiness: ⟨Put local variables on the stack and update outdegrees 10⟩;
          goto onward;    /* now a[s] is covered; try to fill in a[s − 1] */
        backup: s++;
          if (s > lb) goto impossible;
          ⟨Restore local variables from the stack and downdate outdegrees 11⟩;
          if (p ≡ q) goto failp;
        failq: while (ptr > ptrq) ⟨Undo a change 13⟩;
        }
      failp: while (ptr > ptrp) ⟨Undo a change 13⟩;
      }
    failpq: 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;
      } else p++, q−−;
    }
    goto backup;
  onward: continue;
  }
  goto done;
impossible:
```

This code is used in section 7.

10. ⟨ Put local variables on the stack and update outdegrees 10 ⟩ ≡

```
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 9.

11. ⟨ Restore local variables from the stack and downdate outdegrees 11 ⟩ ≡

```
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 9.

12. 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 12 ⟩ ≡

```
lbp = l[p];
if (lbp ≥ s) goto failpq;
while (b[lbp] < p) lbp++;
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 (lbq ≥ ubp) goto failpq;
  if (a[lbq] > q) goto failpq;
  for (ubq = lbq; a[ubq + 1] ≤ q ∧ ubq + 1 < ubp; ubq++) ;
  if (lbp ≡ lbq) lbp++;
}
```

This code is used in section 9.

13. 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 13 ⟩ ≡

```
{
  i = undo[−−ptr];
  if (i ≥ 0) a[i ≫ 24] = i & #ffffff;
  else b[(i & #3fffffff) ≫ 24] = i & #ffffff;
}
```

This code is used in section 9.

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

$\langle$ Put p into the chain at location ubp; **goto** *failp* if there's a problem 14 $\rangle \equiv$

```
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 15 ⟩;
```

This code is used in section 9.

15. 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 15 ⟩ ≡

```
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 14.

16. 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 16 ⟩ ≡

```
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 17 ⟩;
```

This code is used in section 9.

17. ⟨ Make forced moves if q has a special form 17 ⟩ ≡

```
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 16.

18. Index.

ACHAIN2