1. Intro. This program is a revision of ACHAIN0, 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) Instead of a simple array a containing the addition chain, I have two arrays a and b, which contain lower and upper bounds on the current status. This idea should speed up the process of placing tentative entries, because the logic in the previous program was rather tortuous. (ii) I try first to find a chain satisfying the lower bound, and work upward until succeeding, instead of trying first to beat the upper bound and continue until failing. This idea, analogous to "iterative deepening," gives me a chance to improve the bounds, because empty slots cannot occur. (iii) I consider ranges for placing both p and q before trying either one of them, because it's often possible to anticipate failure sooner that way. (iv) When trying possibilities for $a[s] = p + q$ and $p \neq q$, we needn't consider any cases with $q > b[s-2]$ or $p > b[s-1]$. This observation makes a huge improvement; shame on me for not thinking of it in the 60s. (v) And it gets even better: Suppose $a[k] = b[k]$ for $k \geq r$. Then as soon as $p > b[r-1]$, we have at most $s - r$ possibilities for p. (My old program ran through $a[s] - a[r]$ possibilities!)

```
#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, p, r, ptrp, ptrq;
  } stack[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 */
  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. If $lb < ub$, we will try to build an addition chain of length lb. If that fails, we increase lb and try again. Finally we will have established the fact that $l(n) = lb$.

Throughout the computation we'll have $a[j] < a[j+1] \le 2a[j]$ and $b[j] < b[j+1] \le 2b[j]$. Element $a[j]$ of the chain is "fixed" if and only if $a[j] = b[j]$. Those local conditions have the nice property that, if $a[j] < b[j]$, we can increase $a[j]$ and/or decrease $b[j]$ to any value in between, causing a corresponding increase and/or decrease in neighboring values but always in a way that preserves the local inequalities without forcing $a[i] > b[i]$ anywhere. (Go figure.)

```
⟨ Backtrack until l(n) is known 7 ⟩ ≡
  l[n] = lb;
  while (lb < ub) {
    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 8 ⟩;
    l[n] = ++lb;
  }
```

This code is used in section 1.

8. 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 8 ⟩ ≡
  ptr = 0;      /* clear the undo stack */
  for (r = s = lb; s > 2; s--) {      /* a[k] = b[k] for all k ≥ r */
    for ( ; r > 1 ∧ a[r − 1] ≡ b[r − 1]; r--) ;
    for (q = a[s] ≫ 1, p = a[s] − q; p ≤ b[s − 1]; ) {
      if (p > b[r − 1]) {
        while (p > a[r]) r++;      /* this step keeps r < s */
        p = a[r], q = a[s] − p, r++;
      }
      ⟨ 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 9 ⟩;
      ptrp = ptr;
      for ( ; ubp ≥ lbp; ubp--) {
        ⟨ Put p into the chain at location ubp; goto failp if there's a problem 11 ⟩;
        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 12 ⟩;
        happiness: stack[s].p = p, stack[s].r = r;
          stack[s].lbp = lbp, stack[s].ubp = ubp;
          stack[s].lbq = lbq, stack[s].ubq = ubq;
          stack[s].ptrp = ptrp, stack[s].ptrq = ptrq;
          goto onward;      /* now a[s] is covered; try to fill in a[s − 1] */
        backup: s++;
          if (s > lb) goto impossible;
          ptrq = stack[s].ptrq, ptrp = stack[s].ptrp;
          lbq = stack[s].lbq, ubq = stack[s].ubq;
          lbp = stack[s].lbp, ubp = stack[s].ubp;
          p = stack[s].p, q = a[s] − p, r = stack[s].r;
          if (p ≡ q) goto failp;
        failq: while (ptr > ptrq) ⟨ Undo a change 10 ⟩;
        }
      failp: while (ptr > ptrp) ⟨ Undo a change 10 ⟩;
      }
    failpq: if (p ≡ q ∧ q > b[s − 2]) q = b[s − 2], p = a[s] − q; else p++, q--;
    }
    goto backup;
  onward: continue;
  }
  goto done;
impossible:
```

This code is used in section 7.

9. 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 9 ⟩ ≡
  lbp = l[p];
  if (lbp ≥ lb) 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 8.

10. 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 10 ⟩ ≡
  {
    i = undo[−−ptr];
    if (i ≥ 0) a[i ≫ 24] = i & #ffffff;
    else b[(i & #3fffffff) ≫ 24] = i & #ffffff;
  }
```

This code is used in section 8.

11. 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 11 ⟩ ≡

```
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);
  }
}
```

This code is used in section 8.

12. 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 12 ⟩ ≡
  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);
    }
  }
```

This code is used in section 8.

13. Index.

a: 1.
argc: 1, 2.
argv: 1, 2.
b: 1.
backup: 8.
clock: 1.
CLOCKS_PER_SEC: 1.
done: 1, 5, 8.
exit: 2.
failp: 8, 11.
failpq: 8, 9.
failq: 8, 12.
fflush: 3.
fgetc: 4.
fopen: 2.
fprintf: 2, 3.
happiness: 8.
i: 1.
impossible: 8.
infile: 1, 2, 4.
j: 1.
l: 1.
lb: 1, 4, 7, 8, 9.
lbp: 1, 8, 9.
lbq: 1, 8, 9, 12.
main: 1.
n: 1.
newa: 10, 11, 12.
newb: 10, 11, 12.
nmax: 1.
onward: 8.
outfile: 1, 2, 3.
p: 1.
pr: 1, 6.
prime: 1, 6.
printf: 1.
ptr: 1, 8, 10.
ptrp: 1, 8.
ptrq: 1, 8.
q: 1.
r: 1.
s: 1.
stack: 1, 8.
stderr: 2.
timer: 1.
ub: 1, 5, 6, 7.
ubp: 1, 8, 9, 11.
ubq: 1, 8, 9, 12.
undo: 1, 8, 10.
x: 1.

⟨ Backtrack until $l(n)$ is known 7 ⟩ Used in section 1.
⟨ Find an upper bound; or in simple cases, set $l(n)$ and **goto** *done* 5 ⟩ Used in section 1.
⟨ 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 9 ⟩ Used in section 8.
⟨ Input the next lower bound, *lb* 4 ⟩ Used in section 1.
⟨ Output the value of $l(n)$ 3 ⟩ Used in section 1.
⟨ Process the command line 2 ⟩ Used in section 1.
⟨ Put p into the chain at location *ubp*; **goto** *failp* if there's a problem 11 ⟩ Used in section 8.
⟨ Put q into the chain at location *ubq*; **goto** *failq* if there's a problem 12 ⟩ Used in section 8.
⟨ Try reducing *ub* with the factor method 6 ⟩ Used in section 5.
⟨ Try to fix the rest of the chain; **goto** *done* if it's possible 8 ⟩ Used in section 7.
⟨ Undo a change 10 ⟩ Used in section 8.

ACHAIN1

	Section	Page
Intro	1	1
The interesting part	7	4
Index	13	9