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

1. Intro. This program is a sequel to ACHAIN2, which you should read first. I'm experimenting with a brand-new way to find shortest addition chains. Maybe it will be good, maybe not; but in either case the results should be interesting (at least to me). At the end of this program I shall discuss the observed running time.

The new idea is to generalize the problem to $l_k(n)$, the minimum length of an addition chain for which $a_j = 2^j$ for $0 \le j \le k$, assuming that $n \ge 2^k$. Clearly $l_0(n) = l_1(n) = l(n)$ is the ordinary function, and we have $l_k(n) \le l_{k+1}(n)$. Furthermore the dual of the binary method (exercise 4.6.3–34) shows that $l_k(n) \le \lfloor \lg n \rfloor + \nu n - 1$. A slightly less obvious fact is the inequality $l_{k+1}(2n) \le l_k(n) + 1$; because if 1, 2, …, 2^k, a_{k+1}, …, n is an addition chain, so is 1, 2, …, 2^k, 2^{k+1}, $2a_{k+1}$, …, $2n$.

When I first thought of defining $l_k(n)$, I conjectured that $l_{k+1}(n) \le l_k(n) + 1$; but I'm tending to believe this less and less, the more I think about it. If it fails, we would have $l_{k+1}(n) > l_{k+1}(2n)$, by the previous inequality; but addition chains are full of surprises.

[Indeed, Neill Clift found the counterexample $l_7(142025) = 20$, $l_8(142025) = 22$ in 2022!]

Two parameters are given on the given line. If they are `foo` and `bar`, this program reads from files `foo-1`, `foo-2`, etc., and writes to files `bar-1`, `bar-2`, etc., with bytes of the kth file giving values of $f_k(n)$ for $n = 2^k$, $2^k + 1$, $2^k + 2$, etc. An input file that doesn't exist, or that is too short to contain information about the number n being worked on, is simply disregarded; but if data is present in an input file, it is believed to be true without further checking.

```
#define nmax (1 ≪ 20)    /* should be less than 2^24 on a 32-bit machine */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
  char l[20][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, lbq, ubq, r, ptrp, ptrq;
  } stack[128];
  int tail[128], outdeg[128], outsum[128], limit[128];
  FILE *infile[64], *outfile[64];
  char buf[100];

  int main(int argc, char *argv[])
  {
    register int i, j, n, p, q, r, s, ubq, lbp, lbq, ptrp, ptrq;
    int lg2n, kk, 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 */
    for (n = 2; n < nmax; n++) {
      ⟨Determine ⌊lg n⌋ and the binary upper bound 5⟩;
      for (kk = lg2n; kk; kk--) {
        ⟨Try to input l_k(n); goto done if successful 4⟩;
        ⟨Backtrack until l_k(n) is known 6⟩;
      done: ⟨Output the value of l_k(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␣foo␣bar\n", argv[0]);
  exit(−1);
}
```

This code is used in section 1.

3. ⟨ Output the value of $l_k(n)$ 3 ⟩ ≡

```
if (¬outfile[kk]) {
  sprintf(buf, "%s-%d", argv[2], kk);
  outfile[kk] = fopen(buf, "w");
  if (¬outfile[kk]) {
    fprintf(stderr, "Can't␣open␣file␣`%s'␣for␣writing!\n", buf);
    exit(−2);
  }
}
fprintf(outfile[kk], "%c", l[kk][n] + '␣');
fflush(outfile[kk]);    /* make sure the result is viewable immediately */
```

This code is used in section 1.

4. Note that the input file for $l_1(n)$ starts with $n = 2$, not $n = 1$ as in the previous programs.

⟨ Try to input $l_k(n)$; **goto** *done* if successful 4 ⟩ ≡

```
if (¬infile[kk]) {
  sprintf(buf, "%s-%d", argv[1], kk);
  infile[kk] = fopen(buf, "r");
  if (¬infile[kk]) infile[kk] = (FILE *) 1;
}
if (infile[kk] ≠ (FILE *) 1) {
  l[kk][n] = fgetc(infile[kk]) − '␣';
  if (l[kk][n] ≤ 0) infile[kk] = (FILE *) 1;    /* shut down input when something fails */
  goto done;    /* accept the input value unquestioningly */
}
```

This code is used in section 1.

5. ⟨ Determine $\lfloor \lg n \rfloor$ and the binary upper bound 5 ⟩ ≡

```
for (q = n, i = −1, j = 0; q; q >>= 1, i++) j += q & 1;
lg2n = i, ub = i + j − 1;
```

This code is used in section 1.

6. The interesting part. The canonical-chain reduction of ACHAIN2 works for $l_k(n)$ as well as for $l(n)$, because the first k steps of an l_k chain are always reduced in the digraph. So I've taken it over here without change.

Well, there is one change: In the former method, I started with a lower bound and worked upward until achieving success; now I'm going to start at an upper-bound-less-1 and continue until failing (as in ACHAIN0). This switch causes only minor modifications, in spite of what I believed when I wrote ACHAIN1.

At the top level, when $k = \lfloor \lg n \rfloor$, there's nothing to do, because *ub* clearly contains the optimal value. For smaller values of k, we start at $l_{k+1}(n) - 1$, and we also set $b[k+1] \leftarrow 2^{k+1}-$, because we know that the value 2^{k+1} has been ruled out.

```
⟨ Backtrack until $l_k(n)$ is known 6 ⟩ ≡
loop: $l[kk][n] = ub$;
  if ($kk \equiv lg2n$) goto done;
  $lb = ub - 1$;      /* lb isn't really a lower bound, it's just a holdover from ACHAIN2 */
  if ($lb \le kk + 1$) goto done;
  for ($i = 0$; $i \le lb$; $i$++) $outdeg[i] = outsum[i] = 0$;
  $a[lb] = b[lb] = n$;
  for ($i = 2$; $i \le kk$; $i$++) $a[i] = b[i] = 1 \ll i$;
  $a[i] = a[kk] + 1, b[i] = (1 \ll i) - 1$;
  for ($i$++; $i < lb$; $i$++) $a[i] = a[i-1] + 1, b[i] = b[i-1] \ll 1$;
  for ($i = lb - 1$; $i > kk$; $i$−−) {
    if (($(a[i] \ll 1) < a[i+1]$)) $a[i] = (a[i+1] + 1) \gg 1$;
    if ($b[i] \ge b[i+1]$) $b[i] = b[i+1] - 1$;
  }
  if ($a[lb-1] > b[lb-1]$) goto done;
  ⟨ Try to fix the rest of the chain; goto done if it's impossible 7 ⟩;
  $ub = lb$;
  goto loop;
```

This code is used in section 1.

7. The only change to this algorithm for ACHAIN2 occurs when we happen to encounter an empty slot (namely when $outdeg[s] \equiv 0$ and s isn't the top level). Then we simply reject the current solution. Reason: If it could be completed with the empty slot, that's great; but we'll discover the fact later. Meanwhile there certainly are canonical solutions with all slots nonempty, and they should be easy to find.

```
⟨Try to fix the rest of the chain; goto done if it's impossible 7⟩ ≡
  $ptr = 0$;      /* clear the undo stack */
  for ($r = s = lb$; $s > kk$; $s$−−) {
    if ($outdeg[s] \equiv 0 \land s < lb$) goto backup;
    if ($outdeg[s] \equiv 1$) $limit[s] = tail[outsum[s]]$; else $limit[s] = 1$;
    for ( ; $r > 1 \land a[r-1] \equiv b[r-1]$; $r$−−) ;
    if ($outdeg[s-1] \equiv 0 \land (a[s] \,\&\, 1)$) $q = a[s]/3$; else $q = a[s] \gg 1$;
    for ($p = a[s] - q$; $p \le b[s-1]$; ) {
      if ($p > b[r-1]$) {
        while ($p > a[r]$) $r$++;      /* this step keeps $r < s$ */
        $p = a[r], q = a[s] - p, r$++;
      }
      if ($q < limit[s]$) goto backup;
      ⟨Find bounds $(lbp, ubq)$ and $(lbq, ubq)$ on where $p$ and $q$ can be inserted; but go to failpq if they
          can't both be accommodated 10⟩;
      $ptrp = ptr$;
      for ( ; $ubq \ge lbp$; $ubq$−−) {
        ⟨Put $p$ into the chain at location $ubq$; goto failp if there's a problem 12⟩;
        if ($p \equiv q$) goto happiness;
        if ($ubq \ge ubq$) $ubq = ubq - 1$;
        $ptrq = ptr$;
        for ( ; $ubq \ge lbq$; $ubq$−−) {
          ⟨Put $q$ into the chain at location $ubq$; goto failq if there's a problem 14⟩;
        happiness: ⟨Put local variables on the stack and update outdegrees 8⟩;
          goto onward;      /* now $a[s]$ is covered; try to fill in $a[s-1]$ */
        backup: $s$++;
          if ($s > lb$) goto done;
          ⟨Restore local variables from the stack and downdate outdegrees 9⟩;
          if ($p \equiv q$) goto failp;
        failq: while ($ptr > ptrq$) ⟨Undo a change 11⟩;
        }
      failp: while ($ptr > ptrp$) ⟨Undo a change 11⟩;
      }
    failpq: if ($p \equiv q$) {
        if ($outdeg[s-1] \equiv 0$) $q = a[s]/3 + 1$;      /* 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;
  }
  possible:
```

This code is used in section 6.

8. ⟨Put local variables on the stack and update outdegrees 8⟩ ≡

```
$tail[s] = q, stack[s].r = r;$
$outdeg[ubq]{+}{+}, outsum[ubq] \mathrel{+}= s;$
$outdeg[ubq]{+}{+}, outsum[ubq] \mathrel{+}= s;$
$stack[s].lbp = lbp, stack[s].ubq = ubq;$
$stack[s].lbq = lbq, stack[s].ubq = ubq;$
$stack[s].ptrp = ptrp, stack[s].ptrq = ptrq;$
```

This code is used in section 7.

9. ⟨Restore local variables from the stack and downdate outdegrees 9⟩ ≡

```
$ptrq = stack[s].ptrq, ptrp = stack[s].ptrp;$
$lbq = stack[s].lbq, ubq = stack[s].ubq;$
$lbp = stack[s].lbp, ubq = stack[s].ubq;$
$outdeg[ubq]{-}{-}, outsum[ubq] \mathrel{-}= s;$
$outdeg[ubq]{-}{-}, outsum[ubq] \mathrel{-}= s;$
$q = tail[s], p = a[s] - q, r = stack[s].r;$
```

This code is used in section 7.

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

⟨Find bounds (lbp, ubq) and (lbq, ubq) on where p and q can be inserted; but go to *failpq* if they can't both be accommodated 10⟩ ≡

```
$lbp = l[kk][p];$
if $(lbp \ge lb)$ goto $failpq$;
while $(b[lbp] < p)$ $lbp{+}{+}$;
if $(a[lbp] > p)$ goto $failpq$;
for $(ubq = lbp;\ a[ubq + 1] \le p;\ ubq{+}{+})$ ;
if $(ubq \equiv s - 1)$ $lbp = ubq$;
if $(p \equiv q)$ $lbq = lbp, ubq = ubq$;
else {
  $lbq = l[kk][q];$
  if $(lbq \ge ubq)$ goto $failpq$;
  while $(b[lbq] < q)$ $lbq{+}{+}$;
  if $(lbq \ge ubq)$ goto $failpq$;
  if $(a[lbq] > q)$ goto $failpq$;
  for $(ubq = lbq;\ a[ubq + 1] \le q \land ubq + 1 < ubq;\ ubq{+}{+})$ ;
  if $(lbp \equiv lbq)$ $lbp{+}{+}$;
}
```

This code is used in section 7.

11. 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 $newa(j, y)$ $undo[ptr{+}{+}] = (j \ll 24) + a[j], a[j] = y$
#define $newb(j, y)$ $undo[ptr{+}{+}] = (1 \ll 31) + (j \ll 24) + b[j], b[j] = y$

⟨Undo a change 11⟩ ≡

```
{
  $i = undo[{-}{-}ptr];$
  if $(i \ge 0)$ $a[i \gg 24] = i \mathbin{\&} \texttt{#ffffff}$;
  else $b[(i \mathbin{\&} \texttt{#3fffffff}) \gg 24] = i \mathbin{\&} \texttt{#ffffff}$;
}
```

This code is used in section 7.

12. At this point we know that $a[ubq] \le p \le b[ubq]$.

```
⟨Put $p$ into the chain at location $ubq$; **goto** $failp$ if there's a problem 12⟩ ≡
  **if** $(a[ubq] \ne p)$ {
    $newa(ubq, p)$;
    **for** $(j = ubq - 1;\ (a[j] \ll 1) < a[j + 1];\ j--)$ {
      $i = (a[j + 1] + 1) \gg 1$;
      **if** $(i > b[j])$ **goto** $failp$;
      $newa(j, i)$;
    }
    **for** $(j = ubq + 1;\ a[j] \le a[j - 1];\ j++)$ {
      $i = a[j - 1] + 1$;
      **if** $(i > b[j])$ **goto** $failp$;
      $newa(j, i)$;
    }
  }
  **if** $(b[ubq] \ne p)$ {
    $newb(ubq, p)$;
    **for** $(j = ubq - 1;\ b[j] \ge b[j + 1];\ j--)$ {
      $i = b[j + 1] - 1$;
      **if** $(i < a[j])$ **goto** $failp$;
      $newb(j, i)$;
    }
    **for** $(j = ubq + 1;\ b[j] > b[j - 1] \ll 1;\ j++)$ {
      $i = b[j - 1] \ll 1$;
      **if** $(i < a[j])$ **goto** $failp$;
      $newb(j, i)$;
    }
  }
  ⟨Make forced moves if $p$ has a special form 13⟩;
```

This code is used in section 7.

13. 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 they are especially useful in ruling out cases that have no good $l_k(n)$.

```
⟨ Make forced moves if p has a special form 13 ⟩ ≡
  i = p − (1 ≪ (ubq − 1));
  if (i ∧ ((i & (i − 1)) ≡ 0) ∧ (i ≪ 4) < p) {
    for (j = ubq − 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 12.

14. 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 14⟩ ≡
  if $(a[ubq] \ne q)$ {
    if $(a[ubq] > q)$ goto $failq$;
    $newa(ubq, q)$;
    for $(j = ubq - 1;\ (a[j] \ll 1) < a[j+1];\ j{-}{-})$ {
      $i = (a[j+1] + 1) \gg 1$;
      if $(i > b[j])$ goto $failq$;
      $newa(j, i)$;
    }
    for $(j = ubq + 1;\ a[j] \le a[j-1];\ j{+}{+})$ {
      $i = a[j-1] + 1$;
      if $(i > b[j])$ goto $failq$;
      $newa(j, i)$;
    }
  }
  if $(b[ubq] \ne q)$ {
    if $(b[ubq] < q)$ goto $failq$;
    $newb(ubq, q)$;
    for $(j = ubq - 1;\ b[j] \ge b[j+1];\ j{-}{-})$ {
      $i = b[j+1] - 1$;
      if $(i < a[j])$ goto $failq$;
      $newb(j, i)$;
    }
    for $(j = ubq + 1;\ b[j] > b[j-1] \ll 1;\ j{+}{+})$ {
      $i = b[j-1] \ll 1$;
      if $(i < a[j])$ goto $failq$;
      $newb(j, i)$;
    }
  }
  ⟨Make forced moves if $q$ has a special form 15⟩;
```

This code is used in section 7.

15.
```
⟨Make forced moves if $q$ has a special form 15⟩ ≡
  $i = q - (1 \ll (ubq - 1))$;
  if $(i \land ((i \,\&\, (i-1)) \equiv 0) \land (i \ll 4) < q)$ {
    for $(j = ubq - 2;\ (i \,\&\, 1) \equiv 0;\ i \gg= 1, j{-}{-})$ ;
    if $(b[j] < (1 \ll j))$ goto $failq$;
    for ( ; $a[j] < (1 \ll j)$; $j{-}{-}$) $newa(j, 1 \ll j)$;
  }
```

This code is used in section 14.

16. The bottom line: Alas, this method turns out to be by far the slowest of all. But maybe somebody will find a use for it? The most interesting thing I noticed is that $l_1(n) = l_2(n)$ for $4 \le n < 14759$; in other words, when n is small there's always a way to get by without using '3' in the chain. But all four addition chains of length 17 for $n = 14759$ start with 1, 2, 3. For example, one of them is 1, 2, 3, 5, 10, 13, 23, 46, 92, 184, 368, 736, 1472, 2944, 2957, 5901, 8858, 14759.

(I learned subsequently that Schönhage had conjectured $l_1(n) = l_2(n)$ in 1975. Moreover, Bleichenbacher and Flammenkamp mentioned the first three counterexamples in an unpublished preprint of 1997. In fact, the counterexample $n = 38587$ had actually been found already by Tsai and Chin in 1992 [*Proc. Nat. Sci. Council* **A16** (Taiwan: 1992), 506–514].)

17. Index.

ACHAIN3