1. Introduction. This program finds the length of the shortest strong addition chain that leads to a given number n. A strong addition chain — aka a Lucas chain or a Chebyshev chain — is a sequence of integers $1 = a_0 < a_1 < \cdots < a_m = n$ with the property that each element a_k, for $1 \le k \le m$, is the sum $a_i + a_j$ of prior elements (as in an ordinary addition chain); furthermore, the difference $|a_i - a_j|$ is either zero or appears in the chain. The minimum possible value of m is called $L(n)$.

This program calculates $L(1)$, $L(2)$, ..., getting as far as it can until time runs out. As usual, I wrote it in a big hurry. My main goal today is to get more experience writing backtrack algorithms, as I rev up to write the early sections of Volume 4 in earnest.

```
#define maxn 100000 /* size of arrays; I don't really expect to get this far */
#define maxm 40 /* actually 2 lg maxn will suffice */
#include <stdio.h>
  int L[maxn]; /* the results */
  int ub[maxm], lb[maxm]; /* upper and lower bounds on a_k */
  int choice[4 * maxm]; /* current choices at each level */
  int bound[4 * maxm]; /* maximum possible choices at each level */
  struct {
    int *ptr;
    int val;
  } assigned[8 * maxm * maxm]; /* for undoing */
  int undo_ptr; /* pointer to the assigned stack */
  int save[4 * maxm]; /* information for undoing at each level */
  int hint[4 * maxm]; /* additional information at each level */
  int verbose = 0; /* set nonzero when debugging */
  int record = 0; /* largest L(n) seen so far */
⟨ Subroutines 7 ⟩
main()
{
  register int a, k, l, m, n, t, work;
  int special = 0, ap, app;
  ub[0] = lb[0] = 1;
  n = 1; m = 0; work = 0;
  while (1) {
    L[n] = m;
    printf("L(%d)=%d:", n, m);
    if (m > record) {
      record = m;
      printf("*");
    }
    if (special) ⟨ Print special reason 21 ⟩
    else
      for (k = 0; k ≤ m; k++) printf("␣%d", ub[k]);
    printf("␣[%d]\n", work);
    n++; work = 0;
    ⟨ Find a shortest strong chain for n 2 ⟩;
  }
}
```

2. Backtracking. The choices to be made on levels 0, 1, 2, ... can be grouped in fours, so that the actions at level l depend on l mod 4. If $l \bmod 4 = 0$, we're given a number a and we want to decide how to write it as a sum $a' + a''$. If $l \bmod 4! = 0$, we're given a number b and we want to place it in the chain if it isn't already present. The cases $l \bmod 4 = 1, 2, 3$ correspond respectively to $b = a'$, $b = a''$, and $b = |a' - a''|$ in the previous choice $a = a' + a''$.

We keep the current state of the chain in arrays lb and ub. If $lb[k] = ub[k]$, their common value is a_k; otherwise a_k has not yet been specified, but its eventual value will satisfy $lb[k] \le a_k \le ub[k]$. These bounds are maintained in such a way that

$$ub[k] < ub[k+1] \le 2 * ub[k] \qquad and \qquad lb[k] < lb[k+1] \le 2 * lb[k].$$

As a consequence, setting the value of a_k might automatically fix the value of some of its neighbors.

Variable t records the state of our progress, in the sense that all elements a_k are known to satisfy the strong chain condition for $m \ge k \ge t$.

Variable l is the current level. We don't find all solutions, since one strong chain is enough to establish the value of $L(n)$.

```
⟨ Find a shortest strong chain for n 2 ⟩ ≡
  ⟨ Set m to the obvious lower bound 3 ⟩;
  while (1) {
    ⟨ Check for known upper bound to simplify the work 18 ⟩;
    ⟨ Initialize to try for a chain of length m 4 ⟩;
    ⟨ Backtrack until finding a solution or exhausting all possibilities 5 ⟩;
  not_found: m++;
  }
  found:
```

This code is used in section 1.

3. The obvious lower bound is $\lceil \lg n \rceil$.

```
⟨ Set m to the obvious lower bound 3 ⟩ ≡
  for (k = (n − 1) ≫ 1, m = 1; k; m++) k ≫= 1;
```

This code is used in section 2.

4. A slightly subtle point arises here: We make $lb[k]$ and $ub[k]$ infinite for $k > m$, because some of our routines below will look in positions $\ge L(a)$ when trying to insert an element a.

```
⟨ Initialize to try for a chain of length m 4 ⟩ ≡
  ub[m] = lb[m] = n;
  for (k = m − 1; k; k−−) {
    lb[k] = (lb[k + 1] + 1) ≫ 1;
    if (lb[k] ≤ k) lb[k] = k + 1;
  }
  for (k = 1; k < m; k++) {
    ub[k] = ub[k − 1] ≪ 1;
    if (ub[k] > n − (m − k)) ub[k] = n − (m − k);
  }
  l = 0;
  t = m + 1;
  for (k = t; k ≤ record; k++) lb[k] = ub[k] = maxn;
  undo_ptr = 0;
```

This code is used in section 2.

5. At each level l we make all choices that lie between $choice[l]$ and $bound[l]$, inclusive. If successful, we advance l and go to *start_level*; otherwise we go to *backup*.

```
⟨ Backtrack until finding a solution or exhausting all possibilities 5 ⟩ ≡
start_level: work++;
  if (verbose) ⟨ Print diagnostic info 6 ⟩;
  if (l & 3) ⟨ Place hint[l] 14 ⟩
  else ⟨ Decrease t and vet another entry, or go to found 8 ⟩;
  ⟨ Additional code reached by goto statements 15 ⟩;
```

This code is used in section 2.

6.

```
⟨ Print diagnostic info 6 ⟩ ≡
  {
    printf("Entering␣level␣%d:\n", l);
    for (k = 1; k < t; k++) printf("␣%d..%d", lb[k], ub[k]);
    printf("␣|");
    for ( ; k ≤ m; k++) printf("␣%d..%d", lb[k], ub[k]);
    printf("\n");
    for (k = 0; k < l; k++) printf("%c%d..%d", (k & 3 ? ',' : '␣'), choice[k], bound[k]);
    printf("\n");
  }
```

This code is used in section 5.

7. Choosing the summands. When a is in the chain and we want to express it as $a' + a''$, we can assume that $a' \geq a''$. Naturally we want to look first to see if suitable values of a' and a'' are already present.

```
⟨ Subroutines 7 ⟩ ≡
  int lookup(int x)      /* is x already in the chain? */
  {
    register int k;
    if (x ≤ 2) return 1;
    for (k = L[x]; x > ub[k]; k++) ;
    return x ≡ ub[k] ∧ x ≡ lb[k];
  }
```

See also sections 9, 12, 13, and 20.

This code is used in section 1.

8. The values of a_1 and a_2 can never be a problem.

```
⟨ Decrease t and vet another entry, or go to found 8 ⟩ ≡
  save[l] = t;      /* remember the current value of t, in case we fail */
decr_t: t−−;
  if (t ≤ 2) goto found;
  if (ub[t] > lb[t]) goto restore_t_and_backup;
  a = ub[t];
  for (k = t − 1; ; k−−)
    if (ub[k] ≡ lb[k]) {
      ap = ub[k], app = a − ap;
      if (app > ap) break;
      if (lookup(app) ∧ lookup(ap − app)) goto decr_t;      /* yes, it's OK already */
    }
  choice[l] = (a + 1) ≫ 1;      /* the minimum choice is a' = ⌈a/2⌉ */
  bound[l] = a − 1;      /* and the maximum choice is a' = a − 1 */
vet_it: ⟨ Put a', a'', and a' − a'' into hint[l + 1], hint[l + 2], and hint[l + 3] 10 ⟩;
advance: l++; goto start_level;
```

This code is used in section 5.

9. The *impossible* subroutine determines rapidly when there is no "hole" in which an element can be placed in the current chain.

```
⟨ Subroutines 7 ⟩ +≡
  int impossible(int x)      /* is there obviously no way to put x in? */
  {
    register int k;
    if (x ≤ 2) return 0;
    for (k = L[x]; x > ub[k]; k++) ;
    return x < lb[k];
  }
```

10. The impossibility test here is redundant, since we would discover in any case that placement fails. But the test makes this program run about twice as fast.

```
⟨ Put a', a'', and a' − a'' into hint[l + 1], hint[l + 2], and hint[l + 3] 10 ⟩ ≡
  ap = choice[l]; app = a − ap;
  if (impossible(ap) ∨ impossible(app) ∨ impossible(ap − app)) goto next_choice;
  hint[l + 1] = ap; hint[l + 2] = app; hint[l + 3] = ap − app;
```

This code is used in section 8.

11. Placing the summands. Any change to the *ub* and *lb* table needs to be recorded in the *assigned* array, because we may need to undo it.

```
#define assign(x, y) assigned[undo_ptr].ptr = x, assigned[undo_ptr++].val = *x, *x = y
```

12. The algorithm for adjusting upper and lower bounds is probably the most interesting part of this whole program. I suppose I should prove it correct.

(Since this subroutine is called only in one place, I might want to try experimenting to see how much faster this program runs when subroutine-call overhead is avoided by converting to inline code. Subroutining might actually turn out to be a win because of the limited number of registers on x86-like computers.)

```
⟨ Subroutines 7 ⟩ +≡
  place(int x, int k)      /* set a_k = x */

  {
    register int j = k, y = x;
    if (ub[j] ≡ y ∧ lb[j] ≡ y) return 0;
    while (ub[j] > y) {
      assign(&ub[j], y);      /* the upper bound decreases */
      j−−, y−−;
    }
    j = k + 1, y = x + x;
    while (ub[j] > y) {
      assign(&ub[j], y);      /* the upper bound decreases */
      j++, y += y;
    }
    j = k, y = x;
    while (lb[j] < y) {
      assign(&lb[j], y);      /* the lower bound increases */
      j−−, y = (y + 1) ≫ 1;
    }
    j = k + 1, y = x + 1;
    while (lb[j] < y) {
      assign(&lb[j], y);      /* the lower bound increases */
      j++, y++;
    }
  }
```

13. We need a subroutine that does a bit more than just plain *lookup*; *choice_lookup* returns zero if the entry is ≤ 2, otherwise it returns the least index where the entry might possibly be found based on the *ub* table.

```
⟨ Subroutines 7 ⟩ +≡
  int choice_lookup(int x)      /* find the smallest viable place for x */
  {
    register int k;
    if (x ≤ 2) return 0;
    for (k = L[x]; x > ub[k]; k++) ;
    return k;
  }
```

14. In the special case that the entry to be placed is already present, we avoid unnecessary computation by setting $bound[l]$ to zero.

(Note: I thought this would be a good idea, but it didn't actually decrease the observed running time.)

```
⟨ Place hint[l] 14 ⟩ ≡
  {
    a = hint[l];
    save[l] = undo_ptr;
    k = choice[l] = choice_lookup(a);
    if (k ≡ 0 ∨ (a ≡ ub[k] ∧ a ≡ lb[k])) {
      bound[l] = 0;
      goto advance;
    }
    else {
      while (a ≥ lb[k]) k++;
      bound[l] = k − 1;
    }
    goto next_place;
  }
```

This code is used in section 5.

15. ⟨ Additional code reached by **goto** statements 15 ⟩ ≡

```
unplace:
  if (¬bound[l]) goto backup;
  while (undo_ptr > save[l]) {
    −−undo_ptr;
    ∗assigned[undo_ptr].ptr = assigned[undo_ptr].val;
  }
  choice[l]++;
  a = hint[l];
next_place: if (choice[l] > bound[l]) goto backup;
  place(a, choice[l]);
  goto advance;
```

See also section 17.

This code is used in section 5.

16. Finally, when we run out of steam on the current level, we reconsider previous choices as follows.

17. ⟨ Additional code reached by **goto** statements 15 ⟩ +≡

```
restore_t_and_backup: t = save[l];
backup:
  if (l ≡ 0) goto not_found;
  −−l;
  if (l & 3) goto unplace;      /∗ l mod 4 = 1, 2, or 3 ∗/
  a = ub[t];      /∗ l mod 4 = 0 ∗/
next_choice: choice[l]++;
  if (choice[l] ≤ bound[l]) goto vet_it;
  goto restore_t_and_backup;
```

18. Simple upper bounds. We can often save a lot of work by using the fact that $L(mn) \le L(m)+L(n)$.

```
⟨ Check for known upper bound to simplify the work 18 ⟩ ≡
  {
    for (k = 2, a = n/k; k ≤ a; k++, a = n/k)
      if (n % k ≡ 0 ∧ m ≡ L[k] + L[a]) {
        special = k;
        goto found;
      }
    ⟨ Check for binary method 19 ⟩;
  }
```

This code is used in section 2.

19. Another simple upper bound, $L(n) \le \lfloor \lg n \rfloor + \lfloor \lg \frac{2}{3} n \rfloor$, follows from the fact that a strong chain ending with $(a, a+1)$ can be extended by appending either $(2a, 2a+1)$ or $(2a+1, 2a+2)$.

I programmed it here just to see how often it helps, but I doubt if it will be very effective. (Indeed, experience showed that it was the method of choice only for $n = 2$, 3, 5, 7, 11, and 23; probably not for any larger n.)

Incidentally, the somewhat plausible inequality $L(2n+1) \le L(n)+2$ is *not* true, although the analogous inequality $l(2n+1) \le l(n)+2$ obviously holds for ordinary addition chains. Indeed, $L(17) = 6$ and $L(8) = 3$.

```
⟨ Check for binary method 19 ⟩ ≡
  if (m ≡ lg(n) + lg((n + n)/3)) special = 1;
```

This code is used in section 18.

20.

```
⟨ Subroutines 7 ⟩ +≡
  int lg(int n)
  {
    register int m, x;
    for (x = n ≫ 1, m = 0; x; m++) x ≫= 1;
    return m;
  }
```

21.

```
⟨ Print special reason 21 ⟩ ≡
  {
    if (special ≡ 1) printf("␣Binary␣method");
    else printf("␣Factor␣method␣%d␣x␣%d", special, n/special);
    special = 0;
  }
```

This code is used in section 1.

22. Experience showed that the factor method often gives an optimum result, at least for small n. Indeed, the factor method was optimum for all nonprime $n < 1219$. (The first exception, 1219, is 23×53, the product of two primes that have worse-than-normal L values.) Yet the factoring shortcut reduced the total running time by only about 4%, because it didn't help with the hard cases — the cases that keep the computer working hardest. (These timing statistics are based only on the calculations for $n \le 1000$; larger values of n may well be a different story. But I think most of the running time goes into proving that shorter chains are impossible.)

23. Index.

⟨ Additional code reached by **goto** statements 15, 17 ⟩ Used in section 5.
⟨ Backtrack until finding a solution or exhausting all possibilities 5 ⟩ Used in section 2.
⟨ Check for binary method 19 ⟩ Used in section 18.
⟨ Check for known upper bound to simplify the work 18 ⟩ Used in section 2.
⟨ Decrease t and vet another entry, or go to *found* 8 ⟩ Used in section 5.
⟨ Find a shortest strong chain for n 2 ⟩ Used in section 1.
⟨ Initialize to try for a chain of length m 4 ⟩ Used in section 2.
⟨ Place $hint[l]$ 14 ⟩ Used in section 5.
⟨ Print diagnostic info 6 ⟩ Used in section 5.
⟨ Print special reason 21 ⟩ Used in section 1.
⟨ Put a', a'', and $a' - a''$ into $hint[l+1]$, $hint[l+2]$, and $hint[l+3]$ 10 ⟩ Used in section 8.
⟨ Set m to the obvious lower bound 3 ⟩ Used in section 2.
⟨ Subroutines 7, 9, 12, 13, 20 ⟩ Used in section 1.

STRONGCHAIN