November 24, 2020 at 13:23

1. Intro. This program, hacked from ACHAIN4, finds all canonical addition chains of minimum length for a given integer.

There are two command-line parameters. First is a file that contains values of $l(n)$, as output by the previous program. Then comes the desired integer n.

```
#define nmax 10000000   /* should be less than 2^24 on a 32-bit machine */
#include <stdio.h>
  unsigned 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];
  int down[nmax];   /* a navigation aid discussed below */
  FILE *infile;

  main(int argc, char *argv[])
  {
    register int i, j, n, p, q, r, s, ubp, ubq = 0, lbp, lbq, ptrp, ptrq;
    int lb, nn;
    ⟨Process the command line 2⟩;
    ⟨Initialize the down table 7⟩;
    for (n = 1; n ≤ nn; n++) {
      ⟨Input the next value, l[n] 3⟩;
      ⟨Update the down links 8⟩;
    }
    ⟨Backtrack through all solutions 4⟩;
  }
```

2. ⟨Process the command line 2⟩ ≡

```
  if (argc ≠ 3) {
    fprintf(stderr, "Usage: %s infile n\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);
  }
  if (sscanf(argv[2], "%d", &nn) ≠ 1 ∨ nn < 3 ∨ nn ≥ nmax) {
    fprintf(stderr, "The number '%s' was supposed to be between 3 and %d!\n", argv[2], nmax − 1);
    exit(−3);
  }
```

This code is used in section 1.

3. ⟨ Input the next value, $l[n]$ 3 ⟩ ≡

```
lb = fgetc(infile) - '␣';   /* fgetc will return a negative value after EOF */
if (lb < 0 ∨ (n > 1 ∧ lb > l[n - 1] + 1)) {
  fprintf(stderr, "Input␣file␣has␣the␣wrong␣value␣(%d)␣for␣l[%d]!\n", lb, n);
  exit(-4);
}
l[n] = lb;
```

This code is used in section 1.

4. The interesting part.

$\langle$ Backtrack through all solutions 4 $\rangle \equiv$

```
a[0] = b[0] = 1, a[1] = b[1] = 2;
n = nn, lb = l[n];
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, and output all the solutions 9 ⟩;
```

This code is used in section 1.

5. 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 $down[p] \le p$.

$\langle$ Given that $l[p] < s$, increase p to the next such element 5 $\rangle \equiv$

```
{
  p++;
  if (l[p] ≡ s) p = (down[p] > p ? down[p] : nmax);
}
```

This code is used in section 11.

6. $\langle$ Given that $l[p] \ge s$, increase p to the next element with $l[p] < s$ 6 $\rangle \equiv$

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

This code is used in sections 10 and 11.

7. $\langle$ Initialize the *down* table 7 $\rangle \equiv$

```
for (n = 1; n ≤ nn; n++) down[n] = n − 1;
```

This code is used in section 1.

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

$\langle$ Update the *down* links 8 $\rangle \equiv$

```
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.

9. ⟨ Try to fix the rest of the chain, and output all the solutions 9 ⟩ ≡

```
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 10 ⟩;
  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 14 ⟩;
    ptrp = ptr;
    for ( ; ubp ≥ lbp; ubp--) {
      ⟨ Put p into the chain at location ubp; goto failp if there's a problem 16 ⟩;
      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 18 ⟩;
      happiness: ⟨ Put local variables on the stack and update outdegrees 12 ⟩;
        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 13 ⟩;
        if (p ≡ q) goto failp;
      failq: while (ptr > ptrq) ⟨ Undo a change 15 ⟩;
      }      /* end loop on ubq */
    failp: while (ptr > ptrp) ⟨ Undo a change 15 ⟩;
    }      /* end loop on ubp */
  failpq: ⟨ Advance p to the next smallest feasible value, and set q = a[s] − p 11 ⟩;
  }      /* end loop on p */
  goto backup;
onward: continue;
}      /* end loop on s */
⟨ Print a solution 20 ⟩;
goto backup;
impossible:
```

This code is used in section 4.

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

⟨ Set p to its smallest feasible value, and $q = a[s] - p$ 10 ⟩ ≡

```
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 6 ⟩;
    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 9.

11. ⟨ Advance p to the next smallest feasible value, and set $q = a[s] - p$ 11 ⟩ ≡

```
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 6 ⟩;
    q = a[s] − p;
  }
} else {
  ⟨ Given that l[p] < s, increase p to the next such element 5 ⟩;
  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 9.

12. ⟨ Put local variables on the stack and update outdegrees 12 ⟩ ≡

```
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.

13. ⟨ Restore local variables from the stack and downdate outdegrees 13 ⟩ ≡

```
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.

14. 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 14⟩ ≡

```
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 9.

15. 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 15⟩ ≡

```
{
  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.

16. 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 16 $\rangle \equiv$

```
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 17 ⟩;
```

This code is used in section 9.

17. 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 17 ⟩ ≡

```
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 16.

18. 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.

$\langle$ Put q into the chain at location ubq; **goto** $failq$ if there's a problem 18 $\rangle \equiv$

```
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 19 ⟩;
```

This code is used in section 9.

19. $\langle$ Make forced moves if q has a special form 19 $\rangle \equiv$

```
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 18.

20. $\langle$ Print a solution 20 $\rangle \equiv$

```
for (j = 0; j ≤ lb; j++) printf("␣%d", a[j]);
printf("\n");
```

This code is used in section 9.

21. Index.

ACHAIN-ALL