November 24, 2020 at 13:23

1. Intro. This program is a transcription of the code that I wrote in 1969 to determine the length $l(n)$ of a shortest addition chain for n. My original program was written in IMP, an idiosyncratic language that was basically an assembly program for the Control Data 6600. It computed $l(n)$ for $n \le 18269$, and it consumed an unknown but probably nontrivial amount of background time on the computer over a period of several weeks. I decided to see how efficient that program really was, by recoding it for a modern machine.

Better techniques for that problem are known by now, of course. I think I can also make the original method go quite a bit faster, by changing the data structures. But I'll never know how much speedup is achieved by any of the newer approaches until I get the old algorithm running again.

The command line should have two parameters, which name an input file and an output file. Both files contain values of $l(1)$, $l(2)$, ..., with one byte per value, using visible ASCII characters by adding `'␣'` to each integer value. The numbers in the input file need not be exact, but they must be valid lower bounds; if the input file contains fewer than n bytes, this program uses the simple lower bound of Theorem 4.6.3C. The output file gets answers one byte at a time, and I expect to "kill" the program manually before it finishes.

By the way, it's fun to look at the output file with a text editor.

```
#define nmax 10000000
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
  char l[nmax];
  int a[129];
  struct {
    int lbp, ubp, lbq, ubq, savep;
  } 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, p, q, n, s, ubp, ubq, lbp, lbq;
    int lb, ub, timer = 0;

    ⟨Process the command line 2⟩;
    prime[0] = 2, pr = 1;
    a[0] = 1, a[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. All the above was necessary just to get warmed up and to set the groundwork for nontrivial cases. The method adopted is simple, but it has some subtleties that I discovered one by one in the 60s.

If $lb < ub$, we will try to build an addition chain of length $ub - 1$. If that succeeds, we decrease ub and try again. Finally we will have established the fact that $l(n) = ub$.

```
⟨ Backtrack until l(n) is known 7 ⟩ ≡
  while (lb < ub) {
    a[ub − 1] = n;
    for (i = 2; i < ub − 1; i++) a[i] = 0;
    ⟨ Try to fill in the rest of the chain; goto done if it's impossible 8 ⟩;
    l[n] = −−ub;
  }
```

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 $p \geq a[s]/2$, trying to make both p and q present.

Two key methods are used to prune down the number of possibilities explored. First, the number p can't be inserted into $a[t]$ when $t < l(p)$. Second, two consecutive nonzero entries of an addition chain must satisfy $a_t < a_{t+1} \leq 2a_t$.

Suppose, for example, that we have a partial solution with $a_{10} = 100$ and $a_{13} = 500$, but a_{11} and a_{12} still are zero (meaning that they haven't been filled in). Then we can't set $a_{11} = 124$, because that would force a_{12} to be at most 248, and a_{13} would be unsupportable. It follows that a_{11} must lie between 125 and 200.

Now suppose that $a_{10} = 100$ and $a_{13} = 200$, while a_{11} and a_{12} are still vacant. In this case a_{11} can be any value from 101 to 199, *including* 199 (because we cannot be sure that the chain will require a_{12}). That's what I meant by a subtlety.

In general, for each value of s, we have three nested loops: one on the value of p described above, one on the different places where p might go into the chain, and one on the different places where q might go into the chain.

The following program is organized outside-in. I could also have written it inside-out; but either way (as I tried to explain in my old paper on **goto** statements) there seems to be a need for jumping into a loop.

My program from 1969 didn't have the statement '**if** $(ubp \equiv s - 1 \land lbp < ubp)$ $lbp = ubp$;' but that improvement is easily justified. For if there's no solution with p in position $a[s-1]$, there won't be one with p placed even earlier.

[Hmm: My "subtle" argument above doesn't *really* need to allow $a_{11} = 199$.]

```
⟨Try to fill in the rest of the chain; goto done if it's impossible 8⟩ ≡
  for $(s = ub - 1;\ s > 1;\ s--)$
    if $(a[s])$ {
      for $(q = a[s] \gg 1, p = a[s] - q;\ q;\ p++, q--)$ {
        ⟨Find bounds lbp and ubp on where p can be inserted; goto tryq if p is already present 9⟩;
        if $(ubp \equiv s - 1 \land lbp < ubp)$ $lbp = ubp$;
        for ( ; $ubp \geq lbp$; $ubp--$) {
          $a[ubp] = p$;
        tryq: ⟨Find bounds lbq and ubq on where q can be inserted; goto happiness if q is already
              present 10⟩;
          for ( ; $ubq \geq lbq$; $ubq--$) {
            $a[ubq] = q$;
          happiness: $stack[s].savep = p$;
            $stack[s].lbp = lbp, stack[s].ubp = ubp$;
            $stack[s].lbq = lbq, stack[s].ubq = ubq$;
            goto onward;      /* now $a[s]$ is covered; try to fill in $a[s-1]$ */
          backup: $s++$;
            if $(s \equiv ub)$ goto done;
            if $(a[s] \equiv 0)$ goto backup;
            $lbq = stack[s].lbq, ubq = stack[s].ubq$;
            $lbp = stack[s].lbp, ubp = stack[s].ubp$;
            $p = stack[s].savep, q = a[s] - p$;
            $a[ubq] = 0$;
          }
          $a[ubp] = 0$;
        }
      }
      goto backup;
    onward: continue;
    }
```

This code is used in section 7.

9. The heart of the computation is the following routine, which decides where insertion is possible. A tedious case analysis seems necessary. We set *ubp* to a harmless value so that the subsequent statement $a[ubp] = 0$ doesn't remove p if p was already present.

#define *harmless* 128

⟨ Find bounds *lbp* and *ubp* on where p can be inserted; **goto** *tryq* if p is already present 9 ⟩ ≡

```
  lbp = l[p];
  if (lbp ≤ 1) goto p_ready;      /* if p is 1 or 2, it's already there */
  if (lbp ≥ ub) goto p_hopeless;
p_search: while (a[lbp] < p) {
    if (a[lbp] ≡ 0) goto p_empty_slot;
    lbp++;
  }
  if (a[lbp] ≡ p) goto p_ready;
p_hopeless: ubp = lbp − 1; goto p_done;      /* no way */
p_empty_slot: for (j = lbp − 1; a[j] ≡ 0; j−−) ;
  i = a[j];
  if (p < i) goto p_hopeless;
  for (i += i, j++; j < lbp; j++) i += i;
  while (p > i) {
    lbp++;
    if (a[lbp]) goto p_search;
    i += i;
  }
  for (j = lbp + 1; a[j] ≡ 0; j++) ;
  i = a[j];
  if (i ≤ p) {
    if (i < p) {
      lbp = j + 1; goto p_search;
    }
  p_ready: ubp = lbp = harmless;
    goto tryq;      /* we found p */
  }
  for (ubp = j − 1, i = (i + 1) ≫ 1; p < i; ubp−−) i = (i + 1) ≫ 1;
p_done:
```

This code is used in section 8.

10. The other case is essentially the same. So if I have a bug in one routine, it probably is present in the other one too.

⟨ Find bounds *lbq* and *ubq* on where *q* can be inserted; **goto** *happiness* if *q* is already present 10 ⟩ ≡

```
  lbq = l[q];
  if (lbq ≥ ub) goto q_hopeless;
  if (lbq ≤ 1) goto q_ready;     /* if q is 1 or 2, it's already there */
q_search: while (a[lbq] < q) {
    if (a[lbq] ≡ 0) goto q_empty_slot;
    lbq++;
  }
  if (a[lbq] ≡ q) goto q_ready;
q_hopeless: ubq = lbq − 1; goto q_done;     /* no way */
q_empty_slot: for (j = lbq − 1; a[j] ≡ 0; j−−) ;
  i = a[j];
  if (q < i) goto q_hopeless;
  for (i += i, j++; j < lbq; j++) i += i;
  while (q > i) {
    lbq++;
    if (a[lbq]) goto q_search;
    i += i;
  }
  for (j = lbq + 1; a[j] ≡ 0; j++) ;
  i = a[j];
  if (i ≤ q) {
    if (i < q) {
      lbq = j + 1; goto q_search;
    }
  q_ready: ubq = lbq = harmless;
    goto happiness;     /* we found p */
  }
  for (ubq = j − 1, i = (i + 1) ≫ 1; q < i; ubq−−) i = (i + 1) ≫ 1;
q_done:
```

This code is used in section 8.

11. Index.

ACHAIN0