**1. Intro.** This program finds all odd $n$-bit palindromes $x$ that are perfect squares, using roughly $2^{n/4}$ steps of computation. Thus I hope to use it for $n$ well over 100. The idea is to try all $2^t$ combinations of the rightmost and leftmost $t+3$ bits, for $t \approx n/4$, and to use number theory to rule out the bad cases rather quickly.

(When $n = 100$ I'll be using $t = 22$. This program is a big improvement over the one I wrote in 2013; that one used $t = 31$ when $n = 100$, and $t \approx n/3$ in general. Michael Coriand surprised me last week by claiming that he had a method using only about $n/4$. At first I was mystified, baffled, stumped. But aha, I woke up this morning with a good guess about what he'd discovered! He asked me to doublecheck his results; and I can't resist, even though I've got more than enough other things to do, because it's fun to write useless code like this.)

I haven't optimized this program for computational speed. My main goal was to get it right, with my personal time minimized. On the other hand I could easily have made it run a lot slower: I didn't pass up some "obvious" ways to avoid redundant computations.

```
#define maxn 180   /* I could go a little higher, but there won't be time */
#include <stdio.h>
#include <stdlib.h>
  int n;   /* the length of palindromes sought */
  unsigned long long y[maxn/2], r[maxn/2];   /* table of partial square roots */
  unsigned long long q[maxn/4], qq[maxn/4];   /* table of partial modular sqrts */
  unsigned long long pretrial[2], trial[3], acc[6];   /* multiplication workspace */
  main(int argc, char *argv[])
  {
    register unsigned long long prod, sqrtxl, a, bit;
    register int j, k, t, p, jj, kk;
    ⟨Process the command line 2⟩;
    printf("Binary palindromic squares with %d bits:\n", n);
    ⟨Choose t and initialize the tables 12⟩;
    for (a = 0; a < 1LL << t; a++) ⟨See if case a leads to any square palindromes 13⟩;
  }
```

**2.** ⟨Process the command line 2⟩ ≡

```
  if (argc ≠ 2 ∨ sscanf(argv[1], "%d", &n) ≠ 1) {
    fprintf(stderr, "Usage: %s n\n", argv[0]);
    exit(−1);
  }
  if (n < 15 ∨ n > maxn) {
    fprintf(stderr, "Sorry: n should be between 15 and %d.\n", maxn);
    exit(−2);
  }
```

This code is used in section 1.

**3.** Here's the theory: Let $a_1a_2\ldots a_t$ be a binary string, and suppose

$$x \;=\; 2^{n-1}+2^{n-4}a_1+2^{n-5}a_2+\cdots+2^{n-3-t}a_t+\cdots+2^{t+2}a_t+\cdots+2^4a_2+2^3a_1+1 \;=\; y^2.$$

Let $x_l=2^{n-1}+2^{n-4}a_1+2^{n-5}a_2+\cdots+2^{n-3-t}a_t$ and $x_u=x_l+2^{n-3-t}$.

It's easy to prove by induction on $t$ that there's a unique integer $q$ between 0 and $2^{t+2}$ such that $q \bmod 4=1$ and $q^2 \bmod 2^{t+3}=2^{t+2}a_t+\cdots+2^4a_2+2^3a_1+1$, whenever $t>0$. Hence the lower bits of the square root, $y \bmod 2^{t+2}$, must be either $q$ or $2^{t+2}-q$.

On the other hand $x_l<x<x_u$; hence $\sqrt{x_l}<y<\sqrt{x_u}$. This tells us about the upper bits: We have $x_u=x_l(1+\delta)$, where $\delta=2^{n-3-t}/x_l$; hence $\sqrt{x_u}-\sqrt{x_l}=\sqrt{x_l}\bigl(\sqrt{1+\delta}-1\bigr)<\sqrt{x_l}\,\delta/2=2^{n-4-t}/\sqrt{x_l}\le 2^{n-4-t}/2^{(n-1)/2}=2^{n/2-7/2-t}$. The integers between $\sqrt{x_l}$ and $\sqrt{x_u}$ will therefore be distinct, modulo $2^{t+2}$, if we have $n/2-7/2-t\le t+2$.

It follows that we need only check two potential values of $y$, for each of the $2^t$ choices of $a_1a_2\ldots a_t$, when $t\ge(n-11)/4$. Furthermore, this estimate is somewhat crude; there won't be many cases to try even if $t$ is a bit smaller.

For example, let's consider the case $n=25$ and $t=3$. In the first case $a_1a_2a_3=000$, we have $x_l=2^{24}$ and $x_u=2^{24}+2^{19}$; so $y$ lies between $(1000000000000)_2$ and $(1000000111111)_2$. Furthermore $y \bmod 2^5$ must be 1 or 31. So the only possible square roots of a binary palindrome having the form $(100000\ldots000001)_2$ are

$$(1000000000001)_2,\quad (1000000011111)_2,\quad (1000000100001)_2,\quad (1000000111111)_2.$$

In the last case $a_1a_2a_3=111$, we have $x_l=2^{24}+2^{22}-2^{19}$ and $x_u=2^{24}+2^{22}$; so $y$ lies between $(1000110101001)_2$ and $(1000111100011)_2$. Furthermore $y \bmod 2^5$ must be 21 or 11. Again there are only four $y$'s to try:

$$(1000110101011)_2,\quad (1000110110101)_2,\quad (1000111001011)_2,\quad (1000111010101)_2.$$

(And the first of these actually works! Its square is $(1001110000010100000111001)_2$.)

The program below actually finds it convenient to try a few cases that could have been ruled out by the arguments above. For example, it will try also $(1000111101011)_2$ and $(1000111110101)_2$ in the previous example.

**4.** We'll have to compute $2^t$ "modular square roots" $q$. Let $q[j]$ be the square root of $2^{j+2}a_j+\cdots+2^3a_1+1$ (modulo $2^{j+3}$), and let $qq[j]$ be the rightmost $t+2$ bits of its square. If $j<t$, $q[j+1]$ will be either $q[j]$ or $q[j]+2^{j+1}$, depending on the $(j+2)$nd bit of $qq[j]$.

When $a_1\ldots a_t=0\ldots0$, we have $q[j]=qq[j]=1$ for all $j$. And when moving from any $a_1\ldots a_t$ to its successor, we need only recompute a few of the entries — $q[t]$ always, $q[t-1]$ half the time, $q[t-2]$ one-fourth of the time, etc.

⟨Initialize the $q$ and $qq$ tables 4⟩ ≡

```
for (j = 1; j ≤ t; j++) q[j] = qq[j] = 1;
```

This code is used in section 12.

**5.** ⟨Update $q$ and $qq$ when $a_p$ changes from 0 to 1 5⟩ ≡

```
q[p] ⊕= 1LL ≪ (p + 1);
qq[p] = q[p] * q[p];
for (j = p + 1; j ≤ t; j++) {
  if (qq[j - 1] & (1LL ≪ (j + 2))) q[j] = q[j - 1] ⊕ (1LL ≪ (j + 1));
  else q[j] = q[j - 1];
  qq[j] = q[j] * q[j];
}
```

This code is used in section 13.

**6.** Similarly, we'll have to compute $2^t$ approximate square roots for the leading bits of $y$. Let $y[j]$ be bits $m$ through $j$ of $\sqrt{x_l}$, where $m = \lfloor n/2 \rfloor - 1$ is the index of the leading bit. The classical algorithm for square root extraction tells us how to go from $y[j]$ to $y[j-1]$: We have a "remainder" $r[j]$ representing the difference from the leading bits of $x_l$ and $y[j]^2$, where $r[j] \le 2y[j]$. To preserve this invariant when $a_1 \ldots a_t = 0 \ldots 0$, we set $y[j-1] = 2y[j]$ and $r[j-1] = 4r[j]$; if then $r[j-1] > 2y[j-1]$ we subtract $2y[j-1]+1$ from $r[j-1]$ and increase $y[j-1]$ by 1. To preserve the invariant for other values of $a_1 \ldots a_t$, the same steps apply except that $r[j-1] = 4r[j] + 2a_i + a_{i+1}$ for an appropriate value of $i$. The bits of the square root need only be computed for $j \ge t+2$; therefore all computations fit easily into a single **long long** register.

Once again it's easy to prime the pump when $a_1 \ldots a_t = 0 \ldots 0$, and to move to the successor by updating fewer than two entries on average (plus roughly $n/8$ entries "in the middle" where $x_l$ has roughly $n/4$ zeros).

⟨ Initialize the $y$ and $r$ tables 6 ⟩ ≡

```
if (n & 1) {
  y[(n-3)/2] = 2, r[(n-3)/2] = 0;
  for (j = (n-5)/2; j >= t+2; j--) y[j] = 2*y[j+1], r[j] = 0;
} else {
  y[n/2-1] = 1, r[n/2-1] = 1;
  for (j = n/2-2; j >= t+2; j--) {
    y[j] = 2*y[j+1], r[j] = 4*r[j+1];
    if (r[j] > 2*y[j]) r[j] -= 2*y[j]+1, y[j]++;
  }
}
```

This code is used in section 12.

**7.** ⟨ Update $y$ and $r$ when $a_p$ changes from 0 to 1 7 ⟩ ≡

```
j = (n-3-p)/2;
if ((n+p) & 1) r[j] += 1;
else r[j] = 4*r[j+1]+2, y[j] = 2*y[j+1];
if (r[j] > 2*y[j]) r[j] -= 2*y[j]+1, y[j]++;
for (j--; j >= t+2; j--) {
  y[j] = 2*y[j+1], r[j] = 4*r[j+1];
  if (r[j] > 2*y[j]) r[j] -= 2*y[j]+1, y[j]++;
}
```

This code is used in section 13.

**8.** Now comes the boring stuff. I hope I don't mess up here. To make the final test, I'll need to square a number of up to 90 bits. I simply treat it as three 32-bit chunks, and multiply by the textbook method.

```
#define m32 #ffffffff    /* 32-bit mask */
⟨ Square the contents of trial 8 ⟩ ≡
  for (j = 0; j < 3; j++) {
    prod = trial[j] * trial[0];
    if (j) prod += acc[j];
    acc[j] = prod & m32;
    prod >>= 32;
    prod += trial[j] * trial[1];
    if (j) prod += acc[j + 1];
    acc[j + 1] = prod & m32;
    prod >>= 32;
    prod += trial[j] * trial[2];
    if (j) prod += acc[j + 2];
    acc[j + 2] = prod & m32;
    acc[j + 3] = prod >> 32;
  }
```

This code is used in section 14.

**9.** To manufacture the *trial*, I need to shift the leading digits appropriately and combine them with the trailing digits. First, I put the leading digits into *pretrial* and *trial*. (This can be tricky: If $n = 129$ or 130, so that $t = 29$, there are 34 leading digits; one of them will go into *trial*[0], 32 into *trial*[1], and one into *trial*[2].)

```
⟨ Shift the leading digits 9 ⟩ ≡
  if (t + 2 < 32) pretrial[0] = (sqrtxl << (t + 2)) & m32, pretrial[1] = (sqrtxl >> (30 − t)) & m32;
  else pretrial[0] = 0, pretrial[1] = (sqrtxl << (t − 30)) & m32;
  trial[2] = sqrtxl >> (62 − t);
```

This code is used in section 13.

**10.**
```
⟨ Add q[t] to the trial 10 ⟩ ≡
  if (t + 2 ≤ 32) trial[0] = pretrial[0] + q[t], trial[1] = pretrial[1];
  else trial[0] = q[t] & m32, trial[1] = pretrial[1] + (q[t] >> 32);
```

This code is used in section 13.

**11.**
```
#define comp(x) ((1LL << (t + 2)) − (x))
⟨ Add the complement of q[t] to the trial 11 ⟩ ≡
  if (t + 2 ≤ 32) trial[0] = pretrial[0] + comp(q[t]), trial[1] = pretrial[1];
  else trial[0] = comp(q[t]) & m32, trial[1] = pretrial[1] + (comp(q[t]) >> 32);
```

This code is used in section 13.

**12.** I make $t = \lfloor (n - 11)/4 \rfloor$. (It will be between 1 and 42.)

```
⟨ Choose t and initialize the tables 12 ⟩ ≡
  t = (n − 11)/4;
  ⟨ Initialize the q and qq tables 4 ⟩;
  ⟨ Initialize the y and r tables 6 ⟩;
```

This code is used in section 1.

**13.** And now at last the denouement, where we put everything together.

⟨ See if case $a$ leads to any square palindromes 13 ⟩ ≡

```
{
  sqrtxl = y[t + 2];
  for (p = t, bit = 1; a & bit; p--, bit <<= 1) ;
  ⟨ Update y and r when a_p changes from 0 to 1 7 ⟩;
  if (y[t + 2] ≥ sqrtxl + 4) fprintf(stderr, "Something's wrong in case %llx!\n", a);
  for ( ; sqrtxl ≤ y[t + 2]; sqrtxl++) {
    ⟨ Shift the leading digits 9 ⟩;
    ⟨ Add q[t] to the trial 10 ⟩;
    ⟨ Check if trial is a solution 14 ⟩;
    ⟨ Add the complement of q[t] to the trial 11 ⟩;
    ⟨ Check if trial is a solution 14 ⟩;
  }
  ⟨ Update q and qq when a_p changes from 0 to 1 5 ⟩
}
```

This code is used in section 1.

**14.** ⟨ Check if *trial* is a solution 14 ⟩ ≡

```
⟨ Square the contents of trial 8 ⟩;
for (j = 0, k = n − 1; j < k; j++, k--) {
  jj = ((acc[j ≫ 5] & (1 ≪ (j & #1f))) ≠ 0);
  kk = ((acc[k ≫ 5] & (1 ≪ (k & #1f))) ≠ 0);
  if (jj ≠ kk) break;
}
if (j ≥ k)     /* solution! */
  printf("%08llx%08llx%08llx^2=%08llx%08llx%08llx%08llx%08llx%08llx\n", trial[2], trial[1],
      trial[0], acc[5], acc[4], acc[3], acc[2], acc[1], acc[0]);
```

This code is used in section 13.

**15. Index.**

# SQUAREPAL