**1. Intro.** This program finds all "perfect digital invariants" of order $m$ in the decimal system, namely all integers that satisfy $\pi_m x = x$, where $\pi_m$ takes an integer into the sum of the $m$th powers of its digits.

It can be shown without difficulty that such integers have at most $m+1$ digits. Indeed, if $10^p \le x < 10^{p+1}$ we have $\pi_m x < 10^p$ whenever $p > m$. (The proof follows from the fact that $(m+1)9^m < 10^{m+1}$.)

It's an interesting backtrack program, in which I successively choose the digits $9 \ge x_1 \ge x_2 \ge \cdots \ge x_{m+1} \ge 0$ that will be the digits of $x$ (in some order). Lower bounds and upper bounds on $x$ are sufficiently sharp to rule out lots of cases before very many of those digits have been specified. (And if $m$ is small, I could even run through *all* such sequences of digits, because there are only $\binom{m+10}{9}$ of them. That's about 2.5 billion when $m = 40$.)

The only high-precision arithmetic needed here is addition. I implement it with binary-coded decimal representation (15 digits per octabyte), using bitwise techniques as suggested in exercise 7.1.3–100.

Memory references (mems) are counted as if an optimizing compiler were doing things like inlining subroutines, and as if the distribution arrays were packed into a single octabyte. I actually keep the elements unpacked, to keep debugging simple.

```
#define maxm 1000
#define maxdigs (1 + (maxm/15))   /* octabytes per binary-coded decimal number */
#define o mems++
#define oo mems += 2

#include <stdio.h>
#include <stdlib.h>
  int m;   /* command-line parameter */
  typedef unsigned long long ull;
  ull mems;
  ull nodes;
  ull thresh = 10000000000;   /* reporting time */
  ull profile[maxm + 3];
  int count;
  int vbose;   /* level of verbosity */
  ⟨Global variables 8⟩;
  ⟨Subroutines 4⟩;
  main(int argc, char *argv[])
  {
    register int j, k, l, p, r, t, pd, alt, blt, xl, change;
    ⟨Process the command line 3⟩;
    ⟨Precompute the power tables 7⟩;
    ⟨Backtrack through all cases 11⟩;
    fprintf(stderr, "Altogether␣%d␣solutions␣for␣m=%d␣(%llu␣nodes,␣%llu␣mems).\n", count, m,
        nodes, mems);
    if (vbose) ⟨Print the profile 2⟩;
  }
```

**2.** ⟨Print the profile 2⟩ ≡

```
  {
    fprintf(stderr, "Profile:␣␣␣␣␣␣␣␣␣␣1\n");
    for (k = 2; k ≤ m + 2; k++) fprintf(stderr, "%19lld\n", profile[k]);
  }
```

This code is used in section 1.

**3.** ⟨Process the command line 3⟩ ≡

```
if (argc < 2 ∨ sscanf(argv[1], "%d", &m) ≠ 1) {
  fprintf(stderr, "Usage: %s m [profile] [verbose] [extraverbose]\n", argv[0]);
  exit(−1);
}
vbose = argc − 2;
if (m < 2 ∨ m > maxm) {
  fprintf(stderr, "Sorry, m should be between 2 and %d, not %d!\n", maxm, m);
  exit(−2);
}
mdigs = 1 + (m/15);
```

This code is used in section 1.

**4. Tricky arithmetic.** I've got to deal with biggish numbers and inspect their decimal digits. But I'm using a binary computer and I don't want to be repeatedly dividing by powers of 10. So I have an addition routine that computes (say) the sum of hexadecimal-coded numbers `#344159959` and `#271828043`, giving `#615988002` as if the numbers were decimal instead.

⟨ Subroutines 4 ⟩ ≡

```
void add(ull *p, ull *q, ull *r)
{   /* add p to q, giving r */
  register int k, c;
  register ull t, w, x, y;
  for (k = c = 0; k < mdigs; k++) ⟨Add c + *(p + k) to *(q + k), giving *(r + k) and carry c 5⟩;
  if (c) {   /* this shouldn't happen */
    fprintf(stderr, "Overflow!\n");
    exit(−999);
  }
}
```

See also sections 6 and 10.

This code is used in section 1.

**5.** It's interesting that I must add $c$ to $x$ here, *not* to $y$. Otherwise the nondecimal digit `a` might appear in the result.

⟨ Add $c + *(p + k)$ to $*(q + k)$, giving $*(r + k)$ and carry $c$ 5 ⟩ ≡

```
{
  o, x = *(p + k) + c;   /* x might have a nondecimal digit now */
  o, y = *(q + k) + #666666666666666;   /* no cross-digit carries occur */
  t = x + y;
  w = (t ⊕ x ⊕ y) & #1111111111111110;   /* this is where cross-digit carries happen */
  w = (w ⊕ #1111111111111110) ≫ 3;
  t −= w + (w ≪ 1);   /* subtract 6 where there were no carries */
  o, *(r + k) = t & #fffffffffffffff;
  c = t ≫ 60;
}
```

This code is used in section 4.

**6.** At the beginning of this program, I need a table of $0^m$, $1^m$, $2^m$, …, $9^m$. So why not compute it via addition?

⟨ Subroutines 4 ⟩ +≡

```
void kmult(int k, ull *a)
{   /* multiply a by k */
  switch (k) {
  case 8: add(a, a, a);
  case 4: add(a, a, a);
  case 2: add(a, a, a); break;
  case 6: add(a, a, a);
  case 3: add(a, a, z); add(a, z, a); break;
  case 5: add(a, a, z); add(z, z, z); add(a, z, a); break;
  case 9: add(a, a, z); add(z, z, z); add(z, z, z); add(a, z, a); break;
  case 7: add(a, a, z); add(a, z, z); add(z, z, z); add(a, z, a); break;
  case 0: case 1: break;
  }
}
```

**7.** ⟨ Precompute the power tables 7 ⟩ ≡

```
for (k = 1; k < 10; k++) {
  table[1][k][0] = k;
  for (j = 2; j ≤ m; j++) kmult(k, table[1][k]);   /* compute k^m */
  for (j = 2; j ≤ m + 1; j++) add(table[1][k], table[j − 1][k], table[j][k]);   /* compute j · k^m */
}
```

This code is used in section 1.

**8.** ⟨ Global variables 8 ⟩ ≡

```
int mdigs;   /* our multiprecision arithmetic routine uses this many octabytes */
ull table[maxm + 2][10][maxdigs];   /* precomputed tables of j · k^m */
ull z[maxdigs];   /* temporary buffer for bignums */
```

See also section 14.

This code is used in section 1.

**9.** Here's a macro that delivers a given digit (nybble) of a multibyte number.

```
#define nybb(a, p) (int)((a[p/15] ≫ (4 ∗ (p % 15))) & #f)
```

**10.** When debugging, or operating verbosely, I want to see all digits of a multiprecise number, with a vertical bar just before digit number $t$.

⟨ Subroutines 4 ⟩ +≡

```
void printnum(ull *a, int t)
{
  register int k;
  for (k = m; k ≥ 0; k−−) {
    if (t ≡ k) fprintf(stderr, "|");
    fprintf(stderr, "%d", nybb(a, k));
  }
}
```

**11. The algorithm.** This program has the overall structure of a typical backtrack program, with a few twists. One of those twists is the state parameter *pd*, which is nonzero when the move at level $l-1$ was forced. (Such cases are rare, but important.)

```
⟨ Backtrack through all cases 11 ⟩ ≡
b1: ⟨ Initialize the data structures 15 ⟩;
b2: profile[l]++, nodes++;
  ⟨ Report the current state, if mems ≥ thresh 12 ⟩;
  for (k = 0; k < 10; k++) {
    pdist[l][k] = pdist[l − 1][k];
    dist[l][k] = dist[l − 1][k] + (k ≡ xl ? 1 : 0);
  }
  oo, oo;    /* two mems to copy pdist and dist, which could have been packed */
  if (pd) ⟨ Absorb a forced move 22 ⟩
  else {
    if (r ≡ 0) goto b5;    /* we haven't room to accept a new digit xl */
    r−−, add(sig[l − 1], table[1][xl], sig[l]);
  }
  if (l > m + 1) ⟨ Print a solution and goto b5 17 ⟩;
b3: if (vbose > 1) fprintf(stderr, "Level␣%d,␣trying␣%d␣(%lld␣mems)\n", l, xl, mems);
  ⟨ If there's an easy way to prove that x_l can't be ≤ xl, goto b5 18 ⟩;
move: ⟨ Advance to the next level with x_l = xl and goto b2 16 ⟩;
b4: if (xl) {
    xl−−;
    o, pd = pdist[l][xl];    /* dist[l][xl] was zero */
    goto b3;
  }
b5: if (−−l) {
    o, pd = pdsave[l];
    if (pd) goto b5;
    ⟨ Restore the previous state at level l 21 ⟩;
    goto b4;
  }
```

This code is used in section 1.

**12.** ⟨ Report the current state, if *mems* ≥ *thresh* 12 ⟩ ≡

```
  if (mems ≥ thresh) {
    thresh += 10000000000;
    fprintf(stderr, "After␣%lld␣mems:", mems);
    for (k = 2; k ≤ l; k++) fprintf(stderr, "␣%lld", profile[k]);
    fprintf(stderr, "\n");
  }
```

This code is used in section 11.

**13.** The purpose of backtrack level $l$ is to compute the $l$th largest digit, $x_l$, of a solution $x$, assuming that $x_1, \ldots, x_{l-1}$ have already been specified.

The main idea is to compute bounds $a_l$ and $b_l$ such that $a_l \le x \le b_l$ must be valid, whenever $x_1, \ldots, x_{l-1}$ have the given values and $x_l$ is at most a given threshold value $xl$. Those bounds, like all of the multiprecise numbers in this computation, are $(m+1)$-digit numbers whose individual digits are $a_{lm} \ldots a_{l0}$ and $b_{lm} \ldots b_{l0}$. They share a common prefix $p_m \ldots p_{t+1}$ of length $m+1-t$; thus if $a_l < b_l$ we have $0 \le t \le m$ and $a_{lt} < b_{lt}$.

The main point is that each of the digits in the multiset $P = \{p_m, \ldots, p_{t+1}\}$ must appear in $x$, and so must each of the digits in the multiset $D = \{d_1, \ldots, d_{t-1}\}$. Therefore we know that each of the digits in $S = P \cup D$ must be present in any solution $x$. (Recall that if $d$ appears $a$ times in a multiset $A$ and $b$ times in a multiset $B$, then it appears $\max(a, b)$ times in $A \cup B$.)

The digit $d$ occurs $dist[l][d]$ times in $D$ and $pdist[l][d]$ times in $P$. If $d > xl$ we must have $pdist[l][d] \le dist[l][d]$. If $d = xl$ we set $pd = \max(0, pdist[l][d] - dist[l][d])$. Thus, if $xl$ occurs thrice in $D$ but only once in $P$, we have $pd = 0$; but if $xl$ occurs thrice in $P$ but only once in $D$, we have $pd = 2$. In the latter case we must choose $x_l = xl$ and also $x_{l+1} = xl$.

Let $r$ be the number of unknown digits of $x$. (When $pd = 0$, this is $m+1$ minus $|S|$, the number of known digits.) If $a_{lt} < b_{lt} < xl$, we know that $r > 0$ and that one of the unknown digits lies between $a_{lt}$ and $b_{lt}$, inclusive.

When $xl$ decreases, the bounds get tighter, hence the prefix can become longer. And that's good.

These are the key facts governing our bounds $a_l$ and $b_l$. In order to do the computations conveniently we maintain the sum of known digits, $sig[l] = \sum_{k=0}^{xl-1} pdist[l][k] \cdot k^m + \sum_{k=xl}^{9} dist[l][k] \cdot k^m + pd \cdot xl^m$.

**14.** ⟨ Global variables 8 ⟩ +≡

```
int dist[maxm + 1][16], pdist[maxm + 1][16];
ull a[maxm + 1][maxdigs], b[maxm + 1][maxdigs], sig[maxm + 1][maxdigs];
int x[maxm + 1], rsave[maxm + 1], tsave[maxm + 1], pdsave[maxm + 1];
```

**15.** ⟨ Initialize the data structures 15 ⟩ ≡

```
l = 1;
pd = pdsave[1] = 0;
alt = 0, blt = 9;
t = m, r = m + 1;
xl = 9;
profile[1] = 1;
goto b3;    /* I really don't want to do step b2 at root level! */
```

This code is used in section 11.

**16.** ⟨ Advance to the next level with $x_l = xl$ and **goto** $b2$ 16 ⟩ ≡

```
oo, tsave[l] = t, rsave[l] = r;
o, pdsave[l] = pd;
o, x[l++] = xl;
goto b2;
```

This code is used in section 11.

**17.** ⟨ Print a solution and **goto** *b5* 17 ⟩ ≡

```
{
  count++;
  printf("%d: ", count);
  for (k = 1; k <= m + 1; k++) printf("%d", x[k]);
  printf("->");
  for (k = m; k >= 0; k--) printf("%d", nybb(sig[l], k));
  printf("\n");
  goto b5;
}
```

This code is used in section 11.

**18.** When this code is performed, *sig*[*l*] and *dist*[*l*] and *pdist*[*l*] are supposed to be up to date, as well as *xl*, *t*, *r*, *alt*, and *blt*.

⟨ If there's an easy way to prove that $x_l$ can't be $\le$ *xl*, **goto** *b5* 18 ⟩ ≡

```
loop: if (t >= 0) {
    change = 0;
    ⟨ Recompute a_l and b_l 19 ⟩;
    if (vbose > 2) {
      fprintf(stderr, " a=");
      printnum(a[l], t);
      fprintf(stderr, ",b=");
      printnum(b[l], t);
      fprintf(stderr, "\n");
    }
    if (change) goto loop;      /* either a_l or b_l or both can be improved */
    while (alt == blt) ⟨ Increase the current prefix, or goto b5 20 ⟩;
    if (change) goto loop;
  }
```

This code is used in section 11.

**19.** The numbers *alt* and *blt* just past the prefix give important constraints on what the future can bring. If we can improve them, we can often improve them further yet, and possibly even extend the prefix.

⟨Recompute $a_l$ and $b_l$ 19⟩ ≡

```
if (blt < xl) {
  if (r ≡ 0) goto b5;
  add(sig[l], table[1][alt], a[l]);    /* a_l ← sig[l] + alt^m */
  add(sig[l], table[1][blt], b[l]);
  add(b[l], table[r − 1][xl], b[l]);    /* b_l ← sig[l] + blt^m + (r − 1) · xl^m */
} else {
  for (k = 0; k < mdigs; k++) oo, a[l][k] = sig[l][k];    /* a_l ← sig[l] */
  add(sig[l], table[r][xl], b[l]);    /* b_l ← sig[l] + r · xl^m */
}
if (o, alt ≠ nybb(a[l], t)) {
  if (alt > nybb(a[l], t)) {
    fprintf(stderr, "Confusion␣(a␣decreased)!\n");
    exit(−13);
  }
  alt = nybb(a[l], t);
  if (blt < xl) change = 1;
}
if (o, blt ≠ nybb(b[l], t)) {
  if (blt < nybb(b[l], t)) {
    fprintf(stderr, "Confusion␣(b␣increased)!\n");
    exit(−14);
  }
  blt = nybb(b[l], t);
  if (blt < xl) change = 1;
}
```

This code is used in section 18.

**20.** Here's the most delicate (and most important) part, as we've learned another digit of $x$.

Incidentally, here's an interesting example of a "flowchart" where a **goto** statement seems necessary without repeating code. Consider two conditions $A$ and $B$, and two actions $\alpha$ and $\beta$. If $A$ and $B$, we want to do $\alpha$ then $\beta$; if $A$ and not $B$, we want to do nothing; if not $A$, we want to do $\beta$. Without a **goto** I must either evaluate $A$ twice (as in 'if (not $A$) or $B$ then (if $A$ do $\alpha$; do $\beta$)') or code $\beta$ twice (as in 'if $A$ then (if $B$ do $\alpha$ and $\beta$) else do $\beta$').

⟨ Increase the current prefix, or **goto** *b5* 20 ⟩ ≡

```
{
  o, p = pdist[l][blt];
  if (blt ≥ xl) {
    if (o, p < dist[l][blt]) goto okay;     /* a "necessary" goto! */
    if (blt > xl) goto b5;     /* oops, we've already saturated that digit */
    pd = p + 1 − dist[l][blt];     /* pd becomes positive, if it wasn't already */
  }
  if (−−r < 0) goto b5;
  add(sig[l], table[1][blt], sig[l]);     /* newly known digit less than xl */
okay: o, pdist[l][blt] = p + 1;
  t−−, change = 1;
  if (t < 0) break;
  oo, alt = nybb(a[l], t), blt = nybb(b[l], t);
}
```

This code is used in section 18.

**21.** ⟨ Restore the previous state at level $l$ 21 ⟩ ≡

```
oo, t = tsave[l], r = rsave[l];
if (t ≥ 0) oo, alt = nybb(a[l], t), blt = nybb(b[l], t);
else alt = blt = 9;
o, xl = x[l];
```

This code is used in section 11.

**22.** When *dist* is "catching up" with *pdist*, we don't change *sig*, because a digit that occurred in the prefix was already accounted for; we knew that an *xl* would be coming, and it has finally arrived. (Also $t$ and $r$ remain unchanged.)

⟨ Absorb a forced move 22 ⟩ ≡

```
{
  if (vbose > 1) fprintf(stderr, "Level␣%d,␣that␣%d␣was␣forced\n", l, xl);
  for (k = 0; k < mdigs; k++) oo, sig[l][k] = sig[l − 1][k];
  if (−−pd) goto move;
}
```

This code is used in section 11.

**23. Index.**

⟨ Absorb a forced move 22 ⟩ Used in section 11.
⟨ Add $c + *(p + k)$ to $*(q + k)$, giving $*(r + k)$ and carry $c$ 5 ⟩ Used in section 4.
⟨ Advance to the next level with $x_l = xl$ and **goto** *b2* 16 ⟩ Used in section 11.
⟨ Backtrack through all cases 11 ⟩ Used in section 1.
⟨ Global variables 8, 14 ⟩ Used in section 1.
⟨ If there's an easy way to prove that $x_l$ can't be $\le xl$, **goto** *b5* 18 ⟩ Used in section 11.
⟨ Increase the current prefix, or **goto** *b5* 20 ⟩ Used in section 18.
⟨ Initialize the data structures 15 ⟩ Used in section 11.
⟨ Precompute the power tables 7 ⟩ Used in section 1.
⟨ Print a solution and **goto** *b5* 17 ⟩ Used in section 11.
⟨ Print the profile 2 ⟩ Used in section 1.
⟨ Process the command line 3 ⟩ Used in section 1.
⟨ Recompute $a_l$ and $b_l$ 19 ⟩ Used in section 18.
⟨ Report the current state, if $mems \ge thresh$ 12 ⟩ Used in section 11.
⟨ Restore the previous state at level $l$ 21 ⟩ Used in section 11.
⟨ Subroutines 4, 6, 10 ⟩ Used in section 1.

# BACK-PDI