セルオートマトンのプログラムハック

和田英一
IIJ イノベーションインスティテュート
eiiti.wada@nifty.com

概要 これまで出会ったプログラムで, これは素晴らしいと思ったもの, うまく出来たと感じたものはいくつもある. そのうちで比較的簡単な小数の十進二進変換, 横方向の和のアルゴリズムなどの優れている例を述べた後, 特にセルオートマトンのプログラミング技法のいくつか, 中でも Life Game の生死判定, Hash による高速計算などを紹介する.

世の中には感動させるプログラムがある

それ古人の名算法を読むは, 碧梧桐の筆法をもってすれば, 迦陵頻伽(かりょうびんが) の声を聞くが如く, 菩薩の天冠を見るが如く, 茯苓(ぶくりょう) を噛むが如く, 檳榔(びんろう) を飲むが如しとか [1]. パラメトロン計算機 PC-1 の頃からプログラミングを始め, これまでに眺めた, あるいは読んだプログラムの中には, すばらしいと思うものも少なからず. 書いた本人も快哉を叫んだに違いない.

- プログラムとして正しい.
- それなりに速い.
- 新しい発想のアルゴリズムである.
- 同様なプログラムを書いた経験者なら, すごいと感じる.

これに通じるのがこのシンポジウムのサブタイトル “Programming should be fun. Programs should be beautiful.” で, この 2 文は Paul Graham が Lisp ハックの精神として *ANSI Common Lisp* の序文の最後に書いたものである [2].

パラメトロン計算機 PC-1 時代の素数を法とする正確計算や高速 Fourier 変換も左様な例だが, もっと簡単な例をいくつか挙げて見る.

小数の十進二進変換

PC-1 は当時の計算機の常識で固定小数点演算回路であった. つまり乗除算では再左端が符号ビットでその右に小数点があると仮定される. 例えば十進の `0.1+`を読み, $(0.00011001100...)_2$ にする. それには 0.1 を 1 としてレジスタの下に置き, 10 倍を 9 回繰り返して 1000000000=10^9 とし 10^{10} で割るのである. 小数点以下何桁読んだかにより 10 倍の回数が変わる.

後藤英一さんの発案で組み込んだのは, 小数点「.」を読むとレジスタの上にマジックナンバー 1.193359375 =$(1.001100011)_2$ を置く. その後の小数の各数字 d を読むと, レジスタを 10 倍して d を足す. 小数の終りを示す「+」か「-」を読むとレジスタが負である限り 10 倍を繰り返し, 正になったら 10^{10} で割る.

マジックナンバーは奇数が 10 個続くから 10 倍してレジスタからオーバーフローしても 2 を法として 1, 符号ビットは負であり続ける. 10 倍を 10 回するとマジックナンバーの最後の 1 までオーバーフローして影響はない. マジックナンバーの後 1 を読み 10 倍する様子は下の通り.

```
←――――――――36 bit――――――――→
1.001100011000000000000000000000000000
1.111011110000000000000000000000000001 0
1.010101100000000000000000000000001010 1
1.010111000000000000000000000001100100 2
1.100110000000000000000000001111101000 3
1.111100000000000000000010011100010000 4
1.011000000000000000110000110101000000 5
1.110000000000000011110100001001000000 6
1.100000000010011000100101101000000000 7
1.000000001011111010111100001000000000 8
0.000001110111001101011001010000000000 9
```

赤字がマジックナンバー

6 桁のマジックナンバーは以下のように作る.

```
1.0
0.5
0.25
0.125
0.0625
0.03125 +
1.59375
1.10011
```

1 から初めて, 順に 2 で割った数 2^{-k} を書く. 小数以下の各桁の和が奇数になるように, 下から 2^{-k} を足すべきか否かを判定していく.

横方向の和

EDSACの本[3]の第2版にあった横方向の和も面白かった．このプログラムについては，Knuthの TAOCP(vol 4A, p.143)[4], Warrenの Hacker's Delight(p. 68)[5] にも言及がある．

```
1100101101101110
1000011001011001
0010001100100011
0001000100010001
```

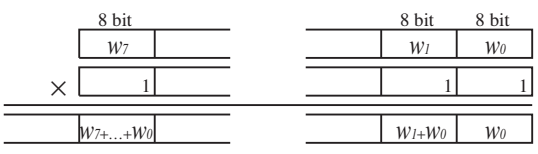
左の図の一番の 16 ビット上は奇数の素数の表で，右からの 1 のビットは順に 3,5,7,11,...,31 に対応する．この範囲の奇数の素数の個数を知るには，1 のビットを足せばよく，この操作を横方向の和とかポピュレーションとかいう．2 段目は 2 ビットごとに分け，その中の 1 のビットの数を示す．3 段目は隣同士合わせた 4 ビットの中での 1 のビットの数で，それを最下段の数と掛けると左端の 4 ビットに総和が出るのである．

EDSAC は 35 ビットなので TAOCP の 64 ビットの説明の方がよかろう．ここで μ_0, μ_1, μ_2 は

$\mu_0 = \ldots 01010101$
$\mu_1 = \ldots 00110011$
$\mu_2 = \ldots 00001111$

である．計算方法は TAOCP 流に書くと

When $x = (x_{63} \ldots x_1 x_0)_2$:
$y \leftarrow x - ((x \gg 1)\&\mu_0)$. { つまり 00→0, 01→1, 10→1, 11→10 のようになる.}
(Now $y = (u_{31} \ldots u_1 u_0)_4$, where $u_j = x_{2j+1} + x_{2j}$.)
$y \leftarrow (y\&\mu_1) + ((y \gg 2)\&\mu_1)$.
(Now $y = (v_{15} \ldots v_1 v_0)_{16}$, $v_j = u_{2j+1} + u_{2j}$.)
$y \leftarrow (y + (y \gg 4))\&\mu_2$. {2 ビットの時と形が違うことに注意．なぜか.}
(Now $y = (w_7 \ldots w_1 w_0)_{256}$, $w_j = v_{2j+1} + v_{2j}$.)
$\nu \leftarrow ((a \cdot y) \bmod 2^{64}) \gg 56$, where $a = (11111111)_{256}$.
図で示すと下のようで，右は十進法でも同様という例．




```
  123
×111
 123
123
123
13653
```

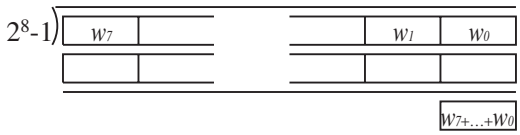
Hacker's Delight にも似た話がある．ただし，これは EDSAC ではなく，HAKMEM 169 の変形とする．

```
int pop(unsigned x){
unsigned n;
n = (x >> 1 ) & 0x77777777;
x = x - n;
n = (n >> 1) & 0x77777777;
x = x - n;
n = (n >>1) & 0x77777777;
x = x - n;
x = (x + (x >> 4) & 0x0F0F0F0F;
x = x * 0x01010101;
return x >> 24;
}
```

4 ビットで横方向の和をとるところが面白い．x を 1 ビット右シフトして n とし，さらに 2 回 1 ビット右シフトしてそれらを x から引く．

$$\begin{array}{rl} & x = 8a_3 + 4a_2 + 2a_1 + 1a_0 \\ & n = 4a_3 + 2a_2 + 1a_1 \\ & n = 2a_3 + 1a_2 \\ - & n = 1a_3 \\ \hline & a_3 + a_2 + a_1 + a_0 \end{array}$$

という計算をしたのだ．TAOCP の横方向の和の $x - ((x \gg 1)\&\mu_0)$ も同じ計算だ．
上記のように乗算で横方向の和が計算できるが，同様に除算によっても計算できる．HAKMEM にはそれを使った例が多い．




```
   13        34
9 ) 123   9 ) 312
  9         27
  33        42
  27        36
  6         6
```

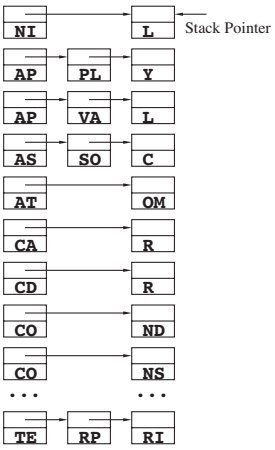
剰余の定理では $f(x) = w_7x^7 + w_6x^6 + \cdots + w_1x + w_0$ を $(x - \alpha)$ で割った剰余は $f(\alpha)$.
$x = 2^8$, $\alpha = 1$ とすると $f(2^8) = w_7 2^{56} + w_6 2^{48} + \cdots + w_1 2^8 + w_0$ を $(2^8 - 1)$ で割った剰余は $f(1)$.
$f(1) = w_7 + w_6 + \cdots + w_1 + w_0$.
図のように 8 ビットの枠に左から $w_7, w_6, \cdots, w_1, w_0$ と並んでいるとする. それを $2^8 - 1$ で割ると, 剰余の定理から分かるように, その剰余が $w_7 + w_6 + \cdots + w_1 + w_0$ になる. 右は十進法での例.

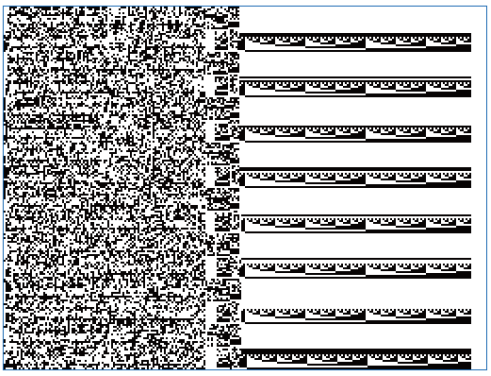
横方向の和は, MMIX 計算機にそのための命令 SA があるが, MMIX がペーパーマシンなので, 実装した CPU が実在するかどうかは知らない.

Pname list を借用したスタック

デルフト大学の van der Poel の書いた PDP-8 用の Lisp はアイディア満載だが, GC で使うスタックには驚く.

PDP-8 は 1 語 12 ビットで, 4096 語のミニコンであった. この Lisp は本体が 2K 語, ヒープ領域が 2K 語で, 記憶場所は節約して使わざるを得ない.
Lisp では予約語や変数名は表になっていて, 読み込んだ文字列がその表のいずれかであるか毎回比較する. PDP-8 では文字は 6 ビットで右の図のように 2 文字のセルがリストでつながっている. リストの最後は cdr 部を 0 にする.
リスト構造だから置く場所は自由であり, この処理系では最後のセルを一カ所に集めた. 従って 1 語置きに 0 の場所が出来る. ここを GC のスタックとして活用するのだ. GC 中は読込みがないからリストを辿ることはない. GC が終わればスタックは空になり, リストの最後は nil に戻る.





左は PDP8 のシミュレータに Lisp 処理系を読み込んだメモリーの図である. 右半分は Lisp のフリー領域. 左半分が Lisp 処理系で, 中央寄りに白い語のかたまりが見えるのが上で述べたスタック部分である.

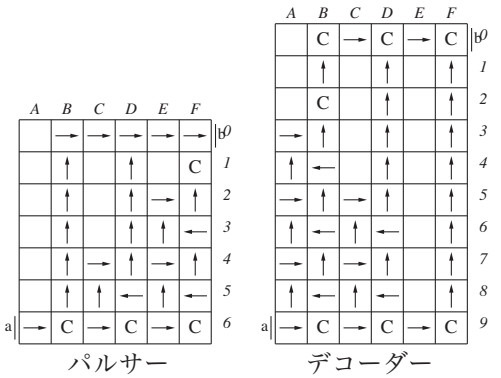
セルオーマトンのシミュレータ

計算機の元々の目的は数値計算であったから, 記号処理や画像処理などのプログラムにはそれなりの工夫が必要である. そういう工夫の必要な分野にセルオートマトンのシミュレータがある. そういうシミュレータで面白い技法を思い出してみた.

von Neumann モデルの C 素子

von Neumann の自己増殖モデル [6] は→や↑など 29 状態を持つセルで構成する. 方向性のある矢印で示す伝達状態は時々刻々と興奮状態を伝える. 伝達状態とは別の C の状態はこの方向を向く矢印のすべてが興

奮した時に興奮を受ける. つまり AND の機能を持つ. 興奮を 1 クロック記憶する. 興奮を分岐させる. 一般伝達から特殊伝達へ興奮を伝えるの機能を持つ.



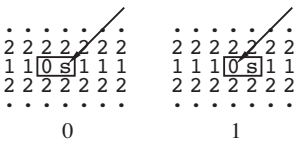
シミュレータでの実装は以下の通り. 判定はもっと複雑かと思ったが, それほどでもなかった.

(左隣りが平常の→でない)∧(下隣りが平常の↑でない)∧
(右隣りが平常の←でない)∧(上隣りが平常の↓でない)∧
((左隣りが興奮の→である)∨(下隣りが興奮の↑である)∨
(右隣りが興奮の←である)∨(上隣りが興奮の↓である))
プログラムは

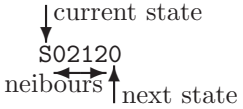
```
boolean rexcited(int r,int u,int l,int d)
{return((r==14)||(u==15)||(l==12)||(d==13));}
boolean aexcited(int r,int u,int l,int d)
{return((r!=10)&&(u!=11)&&(l!=8)&&(d!=9)&&
  (rexcited(r,u,l,d)));}

  case 4: if(sexcited(r,u,l,d)) {i=1;}
          else if(aexcited(r,u,l,d)){i=5;}
          else {i=4;} break;
```

Codd モデルの状態遷移表

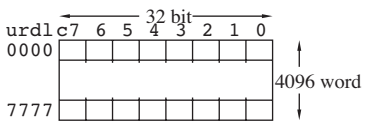
Codd のセルオートマトン [7] は 8 状態, Moore 近傍である. セルは回転対称なので, 中央のセルの状態 c の次に, 4 隣りのセルの状態 urdl(上右下左) を時計回りに書いたものの最小の値を書いて表す.



状態は 0 から 7 で表す. S は 4 から 7 を表す. 「.」は 0 を表す. 状態遷移は回転対称. 4 隣りは辞書式順で最小に書く.



あるセルの次の時刻の状態を得るのに, 時計回りを順に試みるわけには行かない. そこで urdl のそれぞれが 3bit なので, 32bit の 4096 語に



状態遷移規則の表を高速に引きたいから 4 回転分を前もって用意する.

```
int [] langtontab =
```

```
0x000000,0x000012,0x000020,0x000030,0x000050,0x000063,
0x000071,0x000112,0x000122,0x000132,0x000212,0x000220,
0x000230,0x000262,0x000272,0x000320,0x000525,0x000622,
0x000722,0x001022,0x001120,0x002020,0x002030,0x002050,
...
```

のような遷移規則を読み込み, 上の構造の表にしておく. プログラム

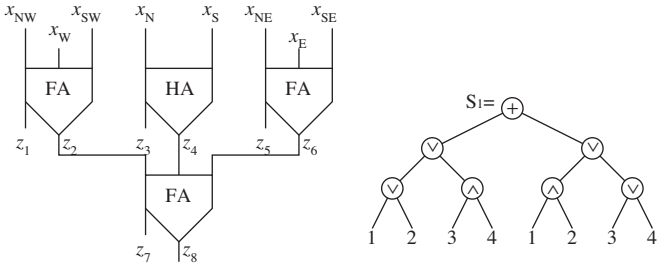
```
n=(tab[(u<<9)|(r<<6)|(d<<3)|l]>>(c*4))&7;
```

で次の状態を得る.

Life Game の生死の判定

Conway の考案による Life Game については, 私は Gardner が *Scientific American* に発表した直後から何度もプログラムを書いた [8]. 最近は Processing を使って動かしている.

その実装法で注目すべきは, TAOCP にある生死判定アルゴリズムだ. 演習問題 7.1.3-167 にある. W. F. Mann と D. Sleator の考えに基づく, 8 近傍の 8 ビットを足し, 1 列の生死を一斉に判定するアルゴリズムだ.



$x^- = X_{j-1}^{(t)}$, $x = X_j^{(t)}$ と $x^+ = X_{j+1}^{(t)}$ が与えられ, $a \leftarrow x^-\&x^+$ $(= z_3)$, $b \leftarrow x^- \oplus x^+$ $(= z_4)$, $c \leftarrow x \oplus b$, $d \leftarrow c \gg 1$ $(= z_6)$, $c \leftarrow c \ll 1$ $(= z_2)$, $e \leftarrow c \oplus d$, $c \leftarrow c\&d$, $f \leftarrow b\&e$, $f \leftarrow f \mid c$ $(= z_7)$, $e \leftarrow b \oplus e$ $(= z_8)$, $c \leftarrow x\&b$, $c \leftarrow c \mid a$, $b \leftarrow c \ll 1$ $(= z_5)$, $c \leftarrow c \gg 1$ $(= z_1)$, $d \leftarrow b\&c$, $c \leftarrow b \mid c$, $b \leftarrow a\&f$, $f \leftarrow a \mid f$, $f \leftarrow d \mid f$, $c \leftarrow b \mid c$, $f \leftarrow f \oplus c$ $(= S_1(z_1, z_3, z_5, z_7))$, $e \leftarrow e \mid x$, $f \leftarrow f\&e$ を計算する
これで f が計算出来る理由は次のようだ. $(z_1 z_2)_2 = x_{nw} + x_w + x_{sw}$ から z_2 は左の列のパリティである. z_1 は左の列の 1 が 2 個か 3 個あることを示す. 同様に z_4 は, 中の列の上下のセルのパリティ, z_3 は, 中の列に 1 が 2 個あることを示す. z_6 と z_5 についても同様である.

$(z_7 z_8)_2 = z_2 + z_4 + z_6$ から, z_8 は x を除いた全体のパリティである. また z_7 は, 各列のパリティの和が 2 か 3 かを示す.

ところで対称関数 S_1 は, 4 個の引数のうち, 1 が 1 個の時に限り 1 を返す. 従って $S_1(z_1, z_3, z_5, z_7)$ が 1 になるのは,
1. 左の列だけに 1 が 2 個か 3 個あるか,
2. 中の列に 1 が 2 個あるか,
3. 右の列に 1 が 2 個か 3 個あるか,
4. 列のパリティの和が 2 か 3 かのいずれかの時である.

1 の場合なら, 左に 1 が 2 個か 3 個あるだけで, 他は 0 である.
2 なら, 中に 1 が 2 個あるだけ, 3 なら右の列に 2 個か 3 個あるだけだ.
4 の場合はどうか. 他の列の 1 の数は 0 個か 1 個である. パリティの和が 2 以上だから, 1 が 1 個の列が 2 列か 3 列あるわけで, やはり 1 の数は全体で 2 個か 3 個である.

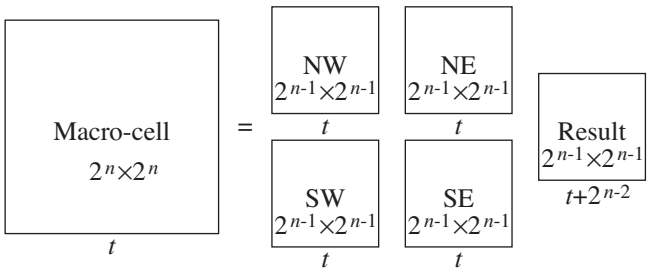
x が 1 なら, 周りに 2 個か 3 個の 1 があれば x は生きるからこれで OK だ. x が 0 なら, z_8 のパリティが 1 なら, 周りに 3 個の 1 があることになり, x は 1 になれる. これがこの計算のからくりであった.

MIT Scheme による実装を次に示す. bit-string を使う. or の縦棒が使えないので, or は!になっている. また 1 ビットシフトの<<と>>は, bit-string の関数で実現している.

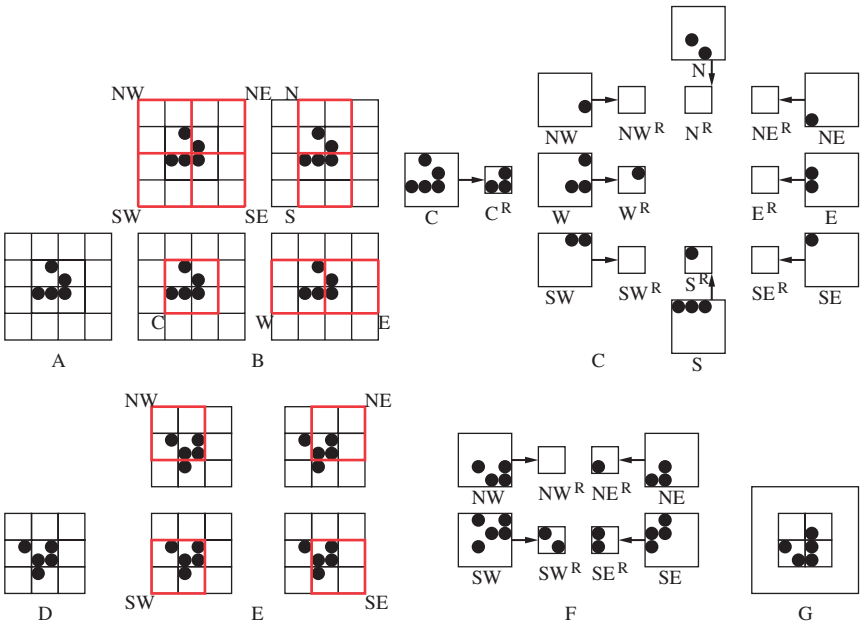
```
(define (b x- x x+)
 (let* (
   (<< (lambda (a) (bit-string-append (make-bit-string 1 #f)
    (bit-substring a 0 (- (bit-string-length a) 1)))))
   (>> (lambda (a) (bit-string-append (bit-substring a 1
    (bit-string-length a)) (make-bit-string 1 #f))))
   (& bit-string-and) (! bit-string-or) (^ bit-string-xor)
   (a0 (& x- x+)) (b0 (^ x- x+)) (c0 (^ x b0)) (d0 (>> c0))
   (c1 (<< c0)) (e0 (^ c1 d0)) (f1 (! (& b0 e0) (& c1 d0)))
   (c4 (! (& x b0) a0)) (b1 (<< c4)) (c5 (>> c4)))
  (& (^ (! (& b1 c5) (! a0 f1)) (! (& a0 f1) (! b1 c5)))
   (! (^ b0 e0) x))))
```

HashLife

Gosper のハッシュ表を使い, 遥か将来の宇宙を計算する Life Game の実装 (HashLife) も面白い [9]. Macro-cell は $n \geq 0$ について $2^n \times 2^n$ のセル空間の状態を表す. つまり, この空間を NE, SE, SW, NW の $2^{n-1} \times 2^{n-1}$ の 4 分木の Macro-cell と, この空間の 2^{n-2} クロック後の中央の $2^{n-1} \times 2^{n-1}$ の Macro-cell(Result という) への 5 個のポインタで構成される.



小さい n については, Result はない. $n = 0$ では 4 分木もなくなり, 生きているセル 1 と死んでいるセル 0 になる.



$n = 3$ の Macro-cell のグライダーの 2^{3-2} クロック後は次のように計算する. 図の A は $2^3 \times 2^3$ の Macro-cell で, Result は図の右下の G のように中央の $2^2 \times 2^2$ の Macro-cell である.

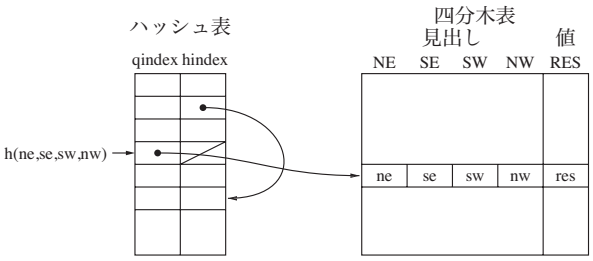
まず A の 4 つの 4 分木 NE,SE,SW,NW の Result NE^{R}, SE^{R}, SW^{R}, NW^{R} を得る. すでにあればそれを取り出し, なければ計算して記憶する.

次に NW の 4 分木 SW, SW の NE, SW の NE, NW の SE から中央に相当する Macro-cell C を作り, その Result C^R を得る. また NE の NW, NE の SW, NW の SE, NW の NE から Macro-cell N や, 同様にして E, S, W を作りそれらの Result を得る. この辺りを図の C に示す.

このようにして得られた 9 個の Result を合わせると, 図の D になる.

さらにこの D から 4 つの 4 分木, NE, SE, SW, NW を作り (図の E), 4 つの Result を得る (F). これを合わせたのが G で, A の Result である.

ハッシュ表は右のようになろう.



プログラムを美しく書くために

Beautiful Code への王道や万能薬は存在しない. 一旦動いたプログラムを何度も見直し 短く改良する余地がないか探す. 未完のプログラムを改良し始めてはいけない. 寝ても覚めても電車で立っていても考え続ける. 時にはまったく違う方向からの解法を試みる. プログラムすべき問題の理解が正しいか反省する. プログラムが複雑に見える時は必ず改良できる. 作文の時も 明瞭で簡潔に書くよう心がける. TAOCP や Hacker's Delight などのアルゴリズムを読みながら 自分ならどう書くか考える.

立教大学にいらした島内剛一先生が活躍されたのはアセンブリ言語の時代で, プログラミングに一家言を持っていた 1 人だ. 「プログラムを短くすると遅くなるという人がいるが, それは違う. 速くなるように短くしなければならない. その秘訣は IBM の標語の通り何度も THINK するしかない.」

竹内郁雄君が「ある人の書く文章とプログラミングには相関がある」という. 高橋秀俊先生: 「私は原稿を 2 度書く. 1 回目は材料を集めるためで, 2 回目はそれをまとめ直す.」 高橋先生は無駄のないプログラムを書かれた.

Edgar Dijkstra はこういった. 「プログラムを書くには, 1. 母国語が十分に駆使できる; 2. スタミナがある; 3. ユーモアのセンスがある; のが必要だ.」 ここでユーモアのセンスとは彼によると, 行き詰まった時に 1 歩下がり考え直すことだそうだ.

Reingold たちの Calendrical Calcuration では暦日計算の原点を, Gregorio 暦が昔からあったとして西暦 1 年 1 月 1 日にとる. その日を第 1 日とする.

その時, y 年 m 月 d 日の日数を計算するのに, 前年の終わりまでの日数として 365 に $y-1$ を掛ける. m 月に対して前月末までの日数の表を足し, d を足し, 閏日を補正する.

しかし, その年末までの日数から, m 月から年末までの日数を引くなら, y を掛けるから $y-1$ の減算が不要になる. プログラムを短く速くするのは, こういう改良の積み重ねである.

The Art of Computer Programming に復活祭の日を計算するアルゴリズムがある. その演算回数を減らすべく書き直したことがある. それを第 47 回プログラミング・シンポジウムの前書きから引用しよう.

Knuth による復活祭のアルゴリズム

元のアルゴリズム

$G \leftarrow (Y \bmod 19) + 1$ { 黄金数, 19 は Meton 周期 }
$C \leftarrow \lfloor (Y/100) \rfloor + 1$ { ほぼ世紀 }
$X \leftarrow \lfloor (3C/4) \rfloor - 12$ {Gregorian 暦のうるう補正 }
$Z \leftarrow \lfloor (8C+5)/25) \rfloor - 5$ { 長期の月 (moon) の補正 }
$D \leftarrow \lfloor 5Y/4 \rfloor - X - 10$ {3 月 $(-D) \bmod 7$ は日曜 }
$E \leftarrow (11G + 20 + Z - X) \bmod 30$ {Epact 歳首月齢 }
if $E = 25$ and $G > 11$ or $E = 24$ then $E \leftarrow E + 1$
$N \leftarrow 44 - E$
if $N < 21$ then $N \leftarrow N + 30$

$N \leftarrow N + 7 - ((D + N) \bmod 7)$
if $N > 31$ then $(N - 31)$ April else N March

ここから改良
$Gm1 \leftarrow Y \bmod 19$ {+1 は変数名で覚えて置く. $Gm1$ は $G-1$ のこと.}
$Cm1 \leftarrow \lfloor Y/100 \rfloor$ {+1 は変数名で覚えて置く. D の $-X-10$ を $Xp10$ とまとめる.}
$Xp10 \leftarrow X + 10 = \lfloor 3(Cm1 + 1)/4 \rfloor - 12 + 10 = \lfloor (3Cm1 - 5)/4 \rfloor$
$E \leftarrow (11(Gm1 + 1) + 20 + Z - (Xp10 - 10)) \bmod 30$ $\{41 = 11 \bmod 30\}$
$= (11Gm1 + 11 + Z - Xp10) \bmod 30$ $\{+11 + Z$ を $Zp11$ とまとめる.$\}$
$Zp11 \leftarrow \lfloor (8(Cm1 + 1) + 5)/25 \rfloor - 5 + 11 = \lfloor (8Cm1 + 163)/25 \rfloor$
$E \leftarrow (11Gm1 + Zp11 - Xp10) \bmod 30$
if $44 - E < 21$ that is if $E > 23$ then $E \leftarrow E - 30$
$Nm31 \leftarrow N + 7 - ((D + N) \bmod 7) - 31$ $\{Nm31$ で 4 月の日付けにする.$\}$
$= 44 - E + 7 - ((D + 44 - E) \bmod 7) - 31$
$= 20 - E - ((D + 49 - 5 - E) \bmod 7) = 20 - E - ((D - 5 - E) \bmod 7)$
$Dm5 \leftarrow \lfloor 5Y/4 \rfloor - (Xp10 + 5)$ $\{D - 5$ を $Dm5$ にし $Xp15$, $Zp16$ と修正.$\}$
$Xp15 \leftarrow \lfloor (3Cm1 - 5)/4 \rfloor + 5 = \lfloor (3Cm1 - 5 + 20)/4 \rfloor = \lfloor (3Cm1 + 15)/4 \rfloor$
$Zp16 \leftarrow \lfloor (8Cm1 + 163)/25 \rfloor + 5 = \lfloor (8Cm1 + 288)/25 \rfloor$
$Nm31 \leftarrow 20 - E - ((Dm5 - E) \bmod 7$

かくして元のアルゴリズムの 28 回の算術演算は 22 回まで減った.

参考文献

[1] 内藤鳴雪, 正岡子規, 高浜虚子, 河東碧悟桐ほか, 佐藤勝明校注, 蕪村句集講義 1, 東洋文庫 801, 平凡社 (2010)

[2] Paul Graham, *ANSI Common Lisp,* Prentice Hall, 1996

[3] Maurice Wilkes, David Wheeler, Stanley Gill, *The Preparation of Programs for an Electronic Digital Computer*, 2nd Edition, Addison-Wesley, 1957.

[4] Donald E. Knuth, *The Art of Computer Programming, Volume 4A Combinatorial Algorithms Part1,* Addison Wesley, 2011

[5] Henry S. Warren, Jr., *Dacker's Delight,* Addison Weslay, 2003

[6] John von Neumann, *Theory of Self-Reproducing Automata*, edited and completed by Authur W. Burks, University of Illinois Pross, 1966

[7] F. E. Codd, *Cellular Automata*, Academic Press, 1968

[8] Erwin R. Berkelamp, john H. Conway, Richard K. Guy, *Winning Ways for Your Mathematical Plays,* Second Edition, Volume 4, A K Peters, Ltd. 2004.

[9] R. Wm. Gosper, *Exploiting Regularities in Large Cellular Spaces,* Physica 10D (1984) pp. 75–80.