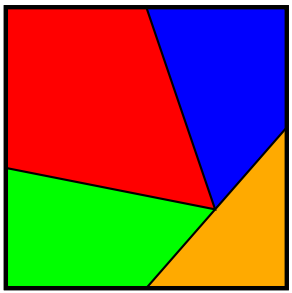
2012年夏のプログラミング・シンポジウム

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
技術研究所



http://www.iijlab.net/~ew/slide2j.pdf

世の中には感動させるプログラムがある

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

PC-1 の小数の十進二進変換

0.1+ を読み 36 ビットレジスタに $(0.00011001100...)_2$ を置きたい 小数点を読んだ時 レジスタにマジックナンバー

$1.193359375=(1.001100011)_2$ を置き

以後数字を読むたびにレジスタを 10 倍し 数字の数をレジスタの下位に足す

$+$ または $-$ を読んだら レジスタが正になるまで 10 倍を繰返し 10^{10} で割る

36 bit

```
1.001100011000000000000000000000000
1.111011110000000000000000000000001 0
1.010101100000000000000000000001010 1
1.010111000000000000000000001100100 2
1.100110000000000000000001111101000 3
1.111100000000000000100111000100000 4
1.011000000000000001100001101010000 5
1.111000000000000111010000100100000 6
1.100000000010011000100101101000000 7
1.000000010111110101111000100000000 8
0.000001110111001101011001010000000 9
```

赤字の部分がマジックナンバー

6桁のマジックナンバーの作り方

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

1958年頃のPC-1 左 後藤先生 右 高橋先生
情報処理学会コンピュータ博物館

横方向の和

EDSACの第2版

64ビット用のKnuthの説明 (TAOCP vol.4A p.143)

$\mu_0 = \ldots 01010101$ 1100101101011110

$\mu_1 = \ldots 00110011$ 1000011001011001

$\mu_2 = \ldots 00001111$ 0010001100100011

0001000100010001

When $x = (x_{63} \ldots x_1 x_0)_2$:

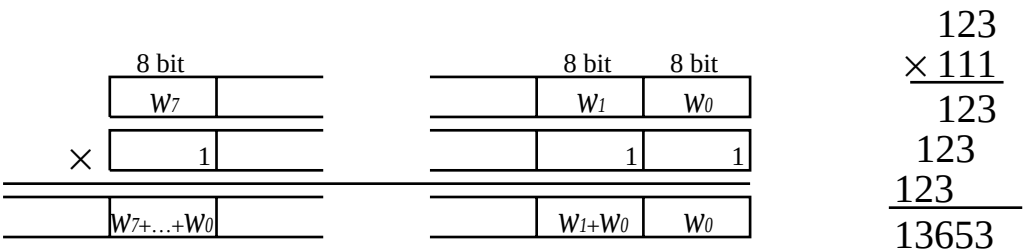
$y \leftarrow x - ((x \gg 1) \& \mu_0)$. 00→0, 01→1, 10→1, 11→10

(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 $y = (v_{15} \ldots v_1 v_0)_{16}$, $v_j = u_{2j+1} + u_{2j}$.)

$y \leftarrow (y + (y \gg 4))\&\mu_2.$
(Now $y = (w_7 \ldots w_1 w_0)_{256}$, $w_j = v_{2j+1} + v_{2j}$.)
$\nu \leftarrow ((a \cdot y) \mathrm{mod} 2^{64}) \gg 56,$
where $a = (11111111)_{256}$.



123
×111
123
123
123
13653

8ビットにしてから乗算する理由

w_i は高々8.
$8w_i$ は高々64だから8ビットに収まる
4ビットだと $4w_i$ は高々16で4ビットに収まらない

横方向の和

Hakcer's Delight (p.68 HAKMEM 169の変形)

```
int pop(unsigned x){
unsigned n;
n = (x >> 1) & 0x77777777;        // Count bits in
x = x - n;                        // exch 4-bit
n = (n >> 1) & 0x77777777;        // field.
x = x - n;
n = (n >> 1) & 0x77777777;
x = x - n;
x = (x + (x >> 4)) & 0x0F0F0F0F; // Get byte sums.
x = x * 0x01010101;               // Add the bytes.
return x >> 24;}
```

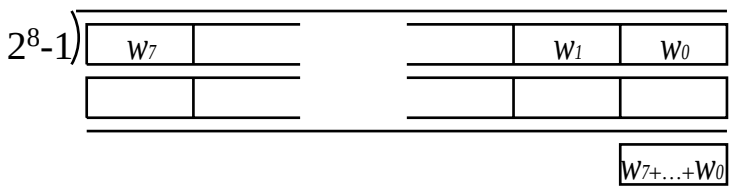
4ビットの横方向の和

x	n	x	n	x	n	x	x	n	x	n	x	n	x
0	0	0	0	0	0	0	8	4	4	2	2	1	1
1	0	1	0	1	0	1	9	4	5	2	3	1	2
2	1	1	0	1	0	1	10	5	5	2	3	1	2
3	1	2	0	2	0	2	11	5	6	2	4	1	3
4	2	2	1	1	0	1	12	6	6	3	3	1	2
5	2	3	1	2	0	2	13	6	7	3	4	1	3
6	3	3	1	2	0	2	14	7	7	3	4	1	3
7	3	4	1	3	0	3	15	7	8	3	5	1	4

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

横方向の和

除算によるもの (HAKMEM に多く見られる)




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

剰余の定理

$f(x) = w_7x^7 + w_6x^6 + \cdots + w_1x + w_0$

を $(x - \alpha)$ で割った剰余は $f(\alpha)$

$x = 2^8$, $\alpha = 1$ とすると

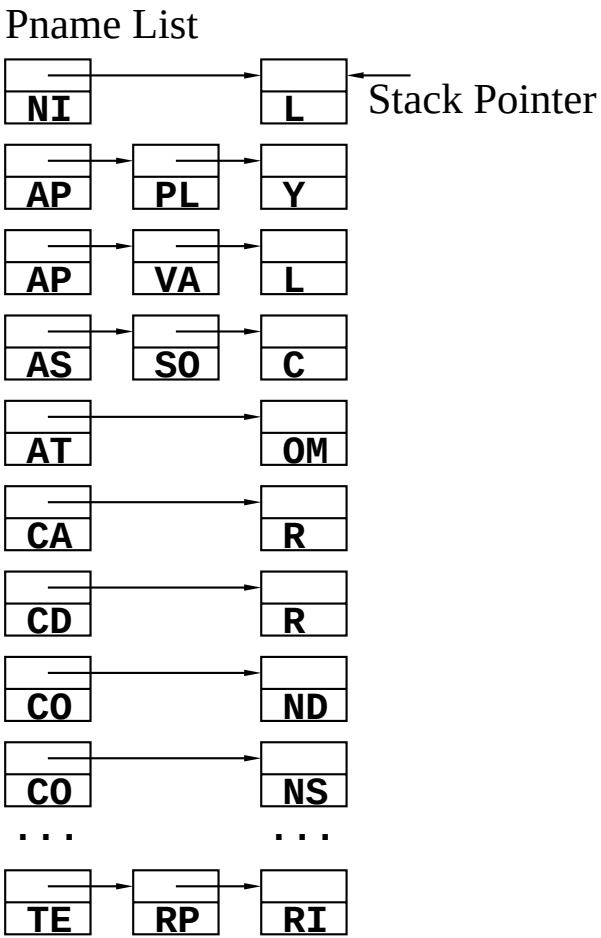
$f(2^8) = w_7 2^{56} + w_6 2^{48} + \cdots + w_1 2^8 + w_0$

を $(2^8 - 1)$ で割った剰余は $f(1)$

$f(1) = w_7 + w_6 + \cdots + w_1 + w_0$

Pname List を借用したスタック

van der Poel の PDP-8 の LISP では GC に使うスタックに Pname のリストの最後のセルを利用した

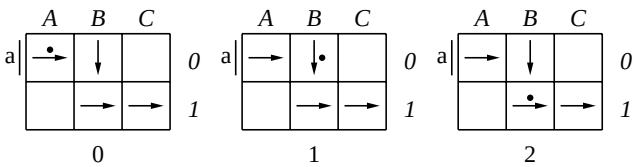


セルオートマトン

時刻$t+1$のセルの状態はtでのこのセルと周囲のセルの状態できまる

von Neumannの自己増殖機械

→や↑など29状態

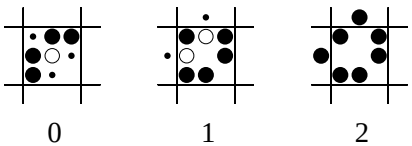


Conwayの Life Game

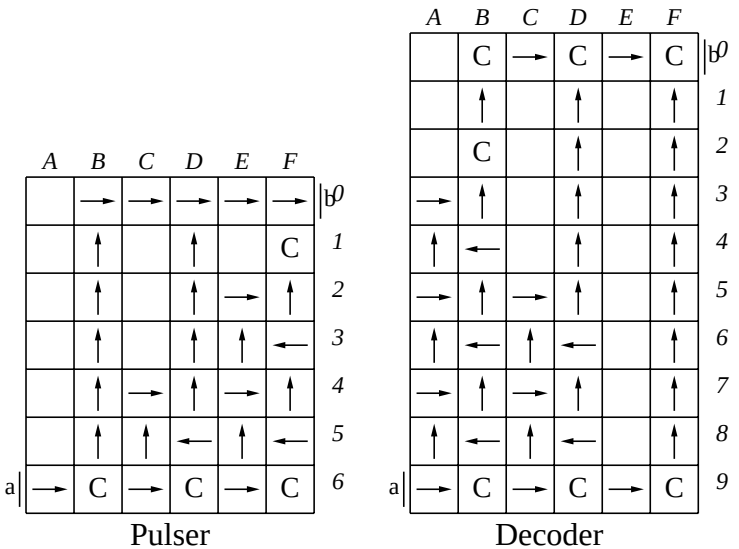
状態は0と1

0の時 8隣りの和が3なら1になる そうでないなら0のまま

1の時 和が2か3なら1のまま そうでないなら0になる



von Neumann 自己増殖モデルの C 素子



C 素子の機能

興奮を 1 クロック記憶する

興奮を分岐させる

複数の入力の興奮の AND をとる

一般伝達から特殊伝達へ興奮を伝える

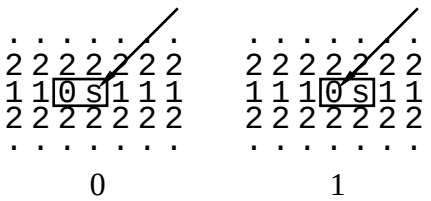
von Neumann モデルの C 素子の実装

(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))

```
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



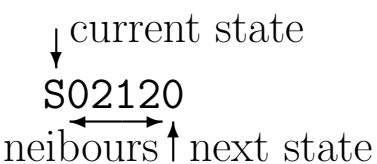
状態は 0 から 7 で表す

s は 4 から 7 を表す

. は 0 を表す

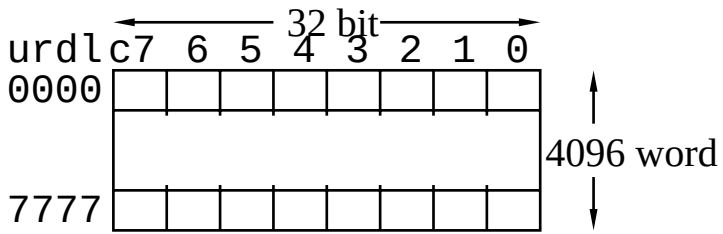
状態遷移は回転対称

4 隣りは辞書式順で最小に書く



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

状態遷移表の実装法




```
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,
...}
```

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

Life Gameの生死の判定

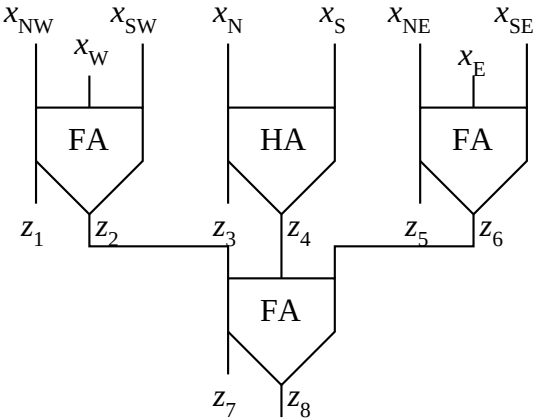
$f(x_{\mathrm{NW}}, \ldots, x, \ldots, x_{\mathrm{SE}}) =$

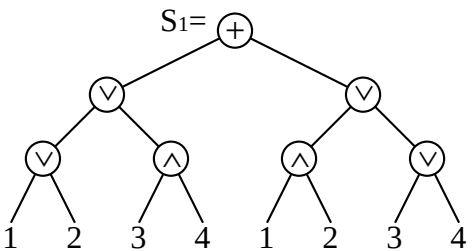
$[2 < x_{\mathrm{NW}} + x_{\mathrm{N}} + x_{\mathrm{NE}} + x_{\mathrm{W}} + \frac{1}{2}x + x_{\mathrm{E}} + x_{\mathrm{SW}} + x_{\mathrm{S}} + x_{\mathrm{SE}} < 4]$

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

$(z_1 z_2)_2 = x_{\mathrm{NW}} + x_{\mathrm{W}} + x_{\mathrm{SW}}$, $(z_3 z_4)_2 = x_{\mathrm{N}} + x_{\mathrm{S}}$, $(z_5 z_6)_2 = x_{\mathrm{NE}} + x_{\mathrm{E}} + x_{\mathrm{SE}}$, $(z_7 z_8)_2 = z_2 + z_4 + z_6$ の半加算器と3つの全加算器を作ることが出来る

すると $f = S_1(z_1, z_3, z_5, z_7) \wedge (x \vee z_8)$





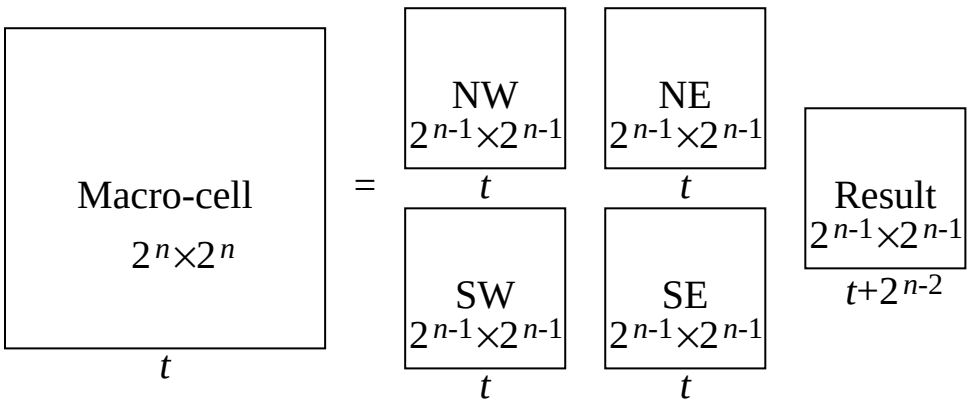
$x^- = X_{j-1}^{(t)}$, $x = X_j^{(t)}$ と $x^+ = X_{j+1}^{(t)}$ が与えられ, $a \leftarrow x^- \& x^+$ $(= z_3)$, $b \leftarrow x^- \oplus x^+$ $(= z_4)$, $c \leftarrow x \oplus b$, $d \leftarrow c \gg 1$ $(= z_6)$, $c \leftarrow c \ll 1$ $(= z_2)$, $e \leftarrow c \oplus d$, $c \leftarrow c \& d$, $f \leftarrow b \& e$, $f \leftarrow f \mid c$ $(= z_7)$, $e \leftarrow b \oplus e$ $(= z_8)$, $c \leftarrow x \& b$, $c \leftarrow c \mid a$, $b \leftarrow c \ll 1$ $(= z_5)$, $c \leftarrow c \gg 1$ $(= z_1)$, $d \leftarrow b \& c$, $c \leftarrow b \mid c$, $b \leftarrow a \& f$, $f \leftarrow a \mid f$, $f \leftarrow d \mid f$, $c \leftarrow b \mid c$, $f \leftarrow f \oplus c$ $(= S_1(z_1, z_3, z_5, z_7))$, $e \leftarrow e \mid x$, $f \leftarrow f \& e$ を計算する

MIT Schemeによる実装 bit-stringを使う

```
(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

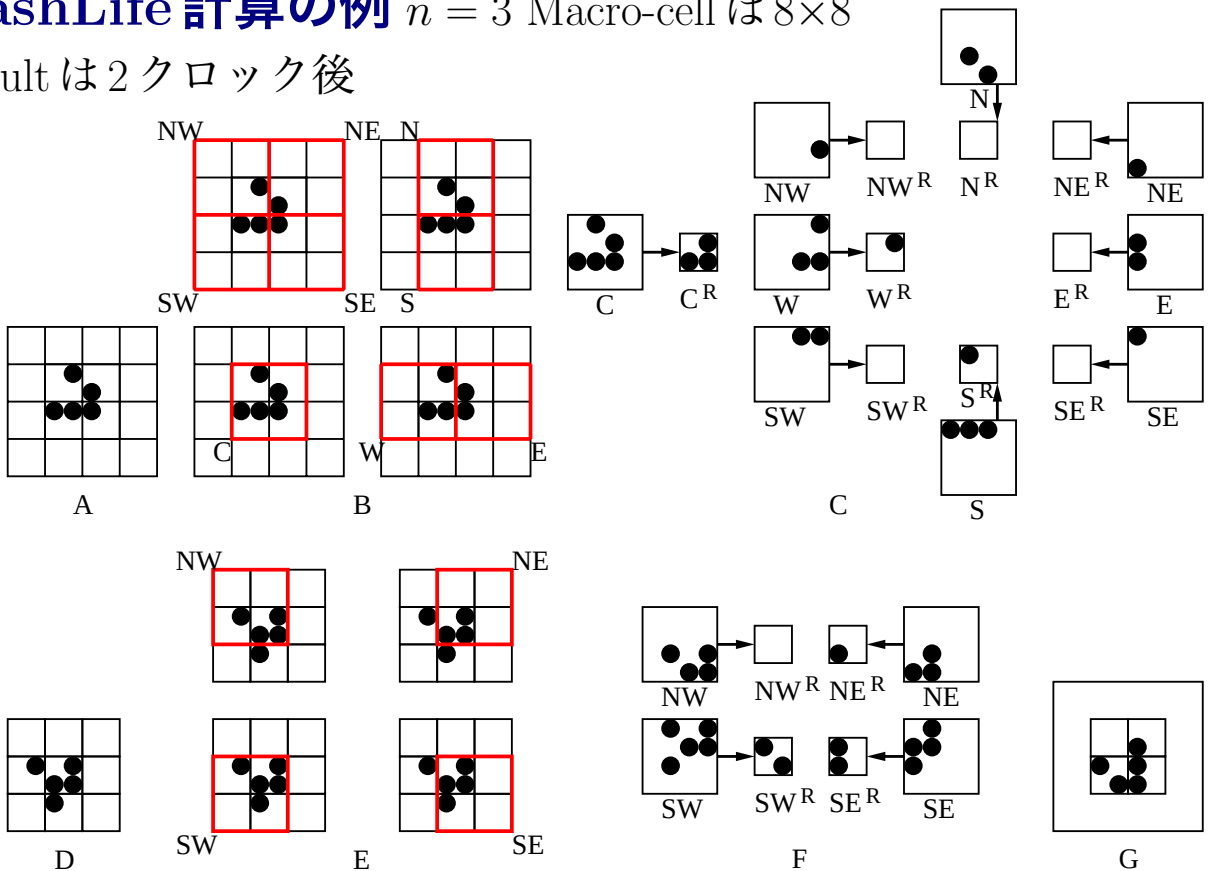
Gosper が考案した 遥か未来宇宙の計算法



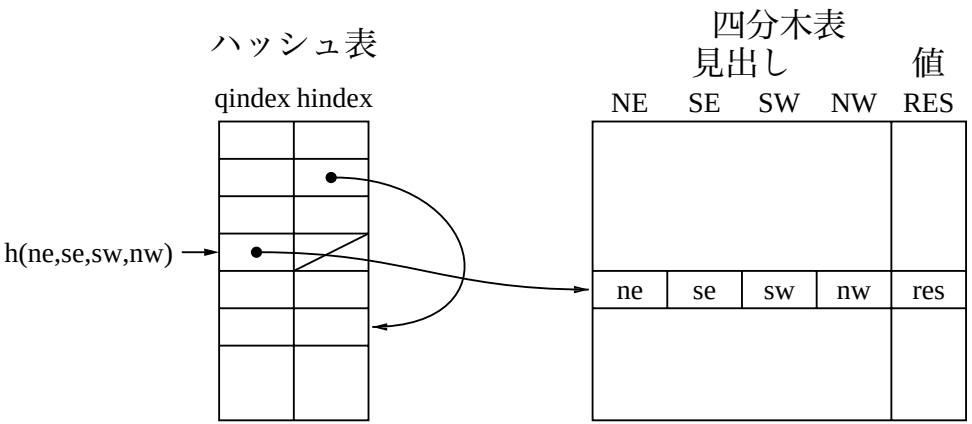
時刻 t サイズ $2^n \times 2^n$ の Macro-cell は
時刻 t サイズ $2^{n-1} \times 2^{n-1}$ の Macro-cell NE SE SW NW と
時刻 $t + 2^{n-2}$ サイズ $2^{n-1} \times 2^{n-1}$ の Macro-cell Result と
で構成される

HashLife 計算の例 $n = 3$ Macro-cell は 8×8

Result は 2 クロック後



ハッシュ表


ハッシュ表
qindex hindex
h(ne,se,sw,nw)
四分木表
見出し
値
NE
SE
SW
NW
RES
ne
se
sw
nw
res

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