ＦＦＴルーチンの概略と使い方

大浦拓哉

京都大学数理解析研究所

目次

ＦＦＴの概略

離散Fourier変換(DFT)：$\{a_0, a_1, \ldots, a_{N-1}\} \to \{A_0, A_1, \ldots, A_{N-1}\}$

$$A_k = \sum_{j=0}^{N-1} a_j W_N^{jk}, \quad W_N = \mathrm{e}^{-2\pi \mathrm{i}/N}$$

FFT アルゴリズム：DFT を高速に計算する方法で，DFT をより小さな DFT に分解することで成り立つ.

DFTの分解

DFT：

$$A_k = \sum_{j=0}^{N-1} a_j W_N^{jk}, \quad W_N = \mathrm{e}^{-2\pi \mathrm{i}/N}$$

仮定：$N = N_1 N_2$と因数分解できる

$$j_1 = 0, 1, 2, \ldots, N_1 - 1, \quad j_2 = 0, 1, 2, \ldots, N_2 - 1$$

写像の定義： j_1, j_2からjへの写像：

$$j \equiv (J_1 j_1 + J_2 j_2) \bmod N$$

添字の分解

写像：

$$j \equiv (J_1 j_1 + J_2 j_2) \bmod N$$

が１対１となるための必要十分条件：

- N_1 と N_2 が互いに素の場合：

  $$J_1 = pN_2, \quad J_2 = qN_1$$

  の少なくとも一方が満たされかつ

  $$\gcd(J_1, N_1) = \gcd(J_2, N_2) = 1$$

- N_1 と N_2 が互いに素でない場合：

  $$J_1 = pN_2 \text{ かつ } J_2 \neq qN_1 \text{ かつ}$$
  $$\gcd(p, N_1) = \gcd(J_2, N_2) = 1$$

  または

  $$J_1 \neq pN_2 \text{ かつ } J_2 = qN_1 \text{ かつ}$$
  $$\gcd(J_1, N_1) = \gcd(q, N_2) = 1$$

ＦＦＴの分解の種類

$$A_k = \sum_{j=0}^{N-1} a_j W_N^{jk}$$

$$\Downarrow \; j \equiv (J_1 j_1 + J_2 j_2) \bmod N, \;\; k \equiv (K_1 k_1 + K_2 k_2) \bmod N$$

$$A_{K_1 k_1 + K_2 k_2} = \sum_{j_2=0}^{N_2-1} \sum_{j_1=0}^{N_1-1} a_{J_1 j_1 + J_2 j_2} W_N^{J_1 K_1 j_1 k_1} W_N^{J_1 K_2 j_1 k_2} W_N^{J_2 K_1 j_2 k_1} W_N^{J_2 K_2 j_2 k_2}$$

分解のための十分条件:

$$J_1 K_2 \equiv 0 \bmod N \text{ または } J_2 K_1 \equiv 0 \bmod N$$

ＦＦＴの分解の種類

分解のための十分条件:

$$J_1 K_2 \equiv 0 \mod N \text{ または } J_2 K_1 \equiv 0 \mod N$$

これを満たす二種類の分解：

- N_1 と N_2 が互いに素の場合 (**素因数型**)

  $$J_1 = N_2, J_2 = N_1, \quad K_1 = N_2, K_2 = N_1$$

- N_1 と N_2 が任意の場合 (**Cooley-Tukey 型**)

  $$J_1 = N_2, J_2 = 1, \quad K_1 = 1, K_2 = N_1$$

  または

  $$J_1 = 1, J_2 = N_1, \quad K_1 = N_2, K_2 = 1$$

素因数型ＦＦＴ

分解１：

$$A_{N_1k_2+N_2k_1} = \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} a_{N_1j_2+N_2j_1} W_{N_1}^{N_2j_1k_1} W_{N_2}^{N_1j_2k_2}$$

$$\Downarrow \ 1 \equiv N_1t_1 + N_2t_2 \bmod N$$

分解２：

$$A_{N_1t_1k_2+N_2t_2k_1} = \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} a_{N_1j_2+N_2j_1} W_{N_1}^{j_1k_1} W_{N_2}^{j_2k_2}$$

分解３：

$$A_{N_1t_1k_2+N_2t_2k_1} = \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} a_{N_1t_1j_2+N_2t_2j_1} W_{N_1}^{t_2j_1k_1} W_{N_2}^{t_1j_2k_2}$$

素因数FFT（PFA）

仮定：

$$N = N_1 N_2 N_3 \cdots N_p$$

と小さな N_i に分解できる

素因数FFT：多次元DFT

$$A_{k_1,\cdots,k_p} = \sum_{j_1=0}^{N_1-1} \cdots \sum_{j_p=0}^{N_p-1} a_{j_1,\cdots,j_p} W_{N_1}^{j_1 k_1} \cdots W_{N_p}^{j_p k_p}$$

を単純な直積で計算

Cooley-Tukey FFT

仮定：

$$N = R^p$$

基数Rの周波数間引きの分解(一段目)：

$$A_{Rk_1+k_2} = \sum_{j_2=0}^{N/R-1} \left(\sum_{j_1=0}^{R-1} a_{Nj_1/R+j_2} W_R^{j_1 k_2} \right) W_N^{j_2 k_2} W_{N/R}^{j_2 k_1}$$

Cooley-Tukey FFT：この分解を再帰的に行う.

高次基数ＦＦＴ(演算量削減１)

基数RのCooley-Tukey ＦＦＴの演算量:

$$\frac{N \log N}{R \log R}((R-1\text{回の複素数乗算}) + (\text{長さ}\,R\,\text{のDFT}))$$

演算量の主要項$/(N \log_2 N)$:

基数R	乗算	加算	加算＋乗算
2	2	3	5
4	1.5	2.75	4.25
8	1.3333	2.75	4.0833
16	1.3125	2.71875	4.03125

Rを大きくとる$\Rightarrow$演算量削減

Split-Radix FFT(演算量削減２)

偶数項は基数２の分解：

$$A_{2k} = \sum_{j=0}^{N/2-1} (a_j + a_{N/2+j}) W_{N/2}^{jk}$$

奇数項は基数４の分解：

$$A_{4k+1} = \sum_{j=0}^{N/4-1} (a_j - a_{N/2+j} - \mathrm{i}a_{N/4+j} + \mathrm{i}a_{3N/4+j}) W_N^j W_{N/4}^{jk}$$

$$A_{4k+3} = \sum_{j=0}^{N/4-1} (a_j - a_{N/2+j} + \mathrm{i}a_{N/4+j} - \mathrm{i}a_{3N/4+j}) W_N^{3j} W_{N/4}^{jk}$$

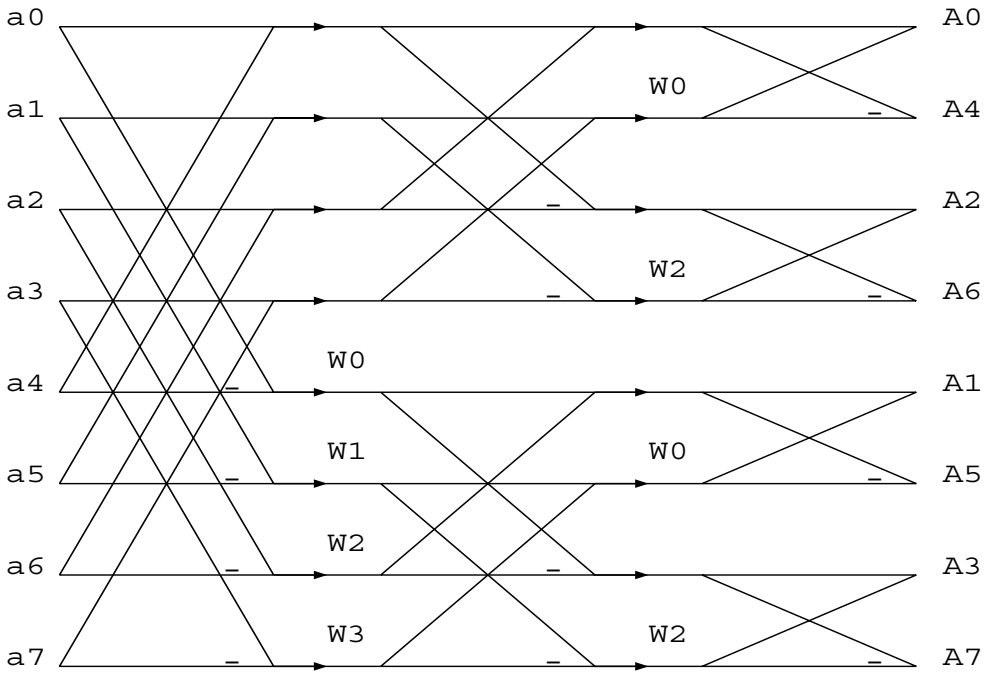
Split-Radix FFTの演算量

演算量の主要項 $/(N \log_2 N)$:

基数	乗算	加算	加算＋乗算
Split-Radix	1.33333	2.66666	4

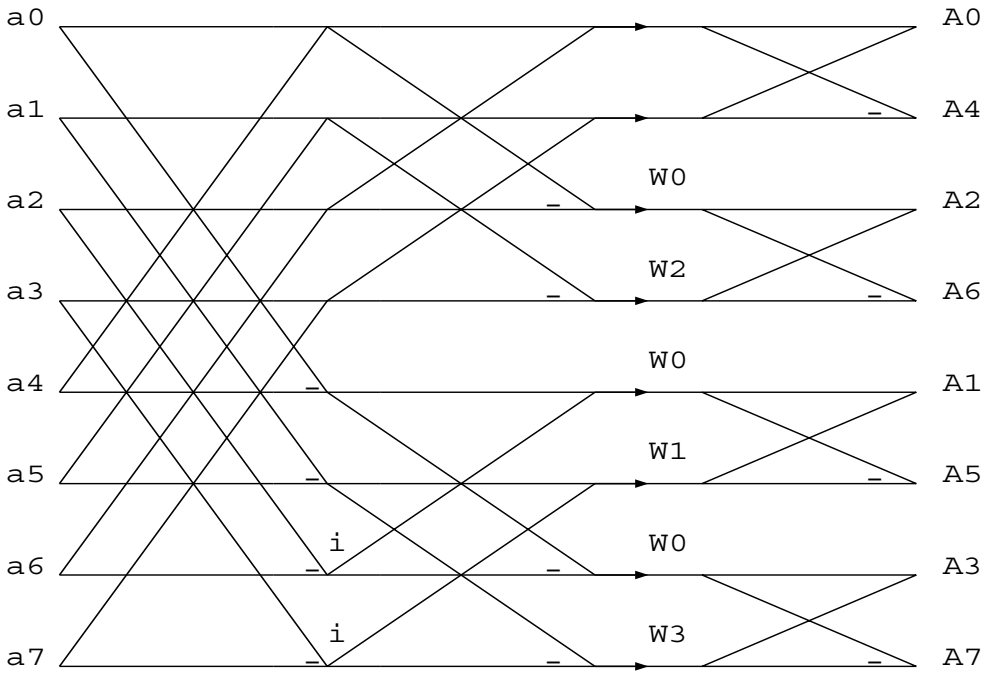
Split-Radix FFT：基数2,4,8,16のCooley-Tukey FFTよりも演算量は少ない.

基数２のＦＦＴのデータフロー



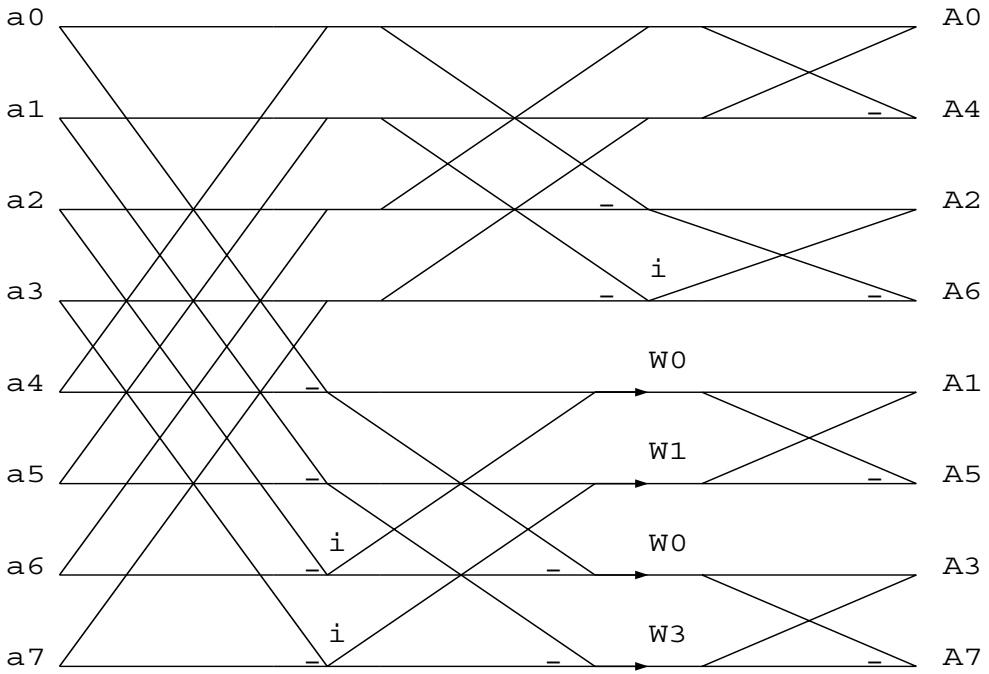
基数２周波数間引きＦＦＴのデータフロー

基数4,2のＦＦＴのデータフロー



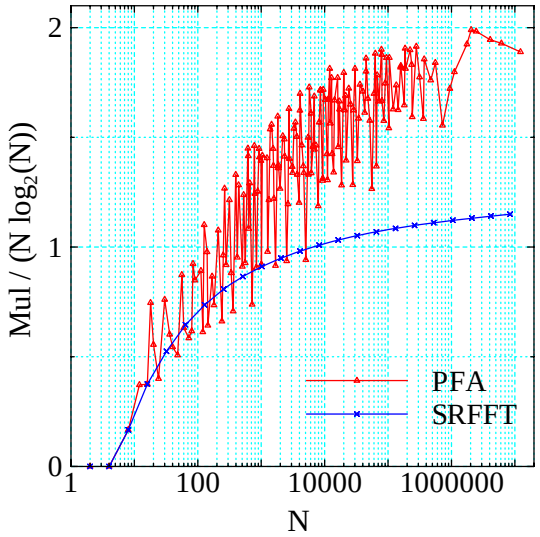
基数４周波数間引きＦＦＴのデータフロー

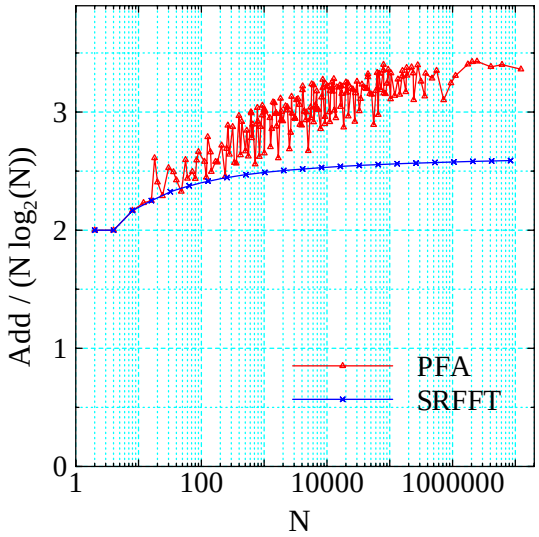
Split-Radix FFTのデータフロー

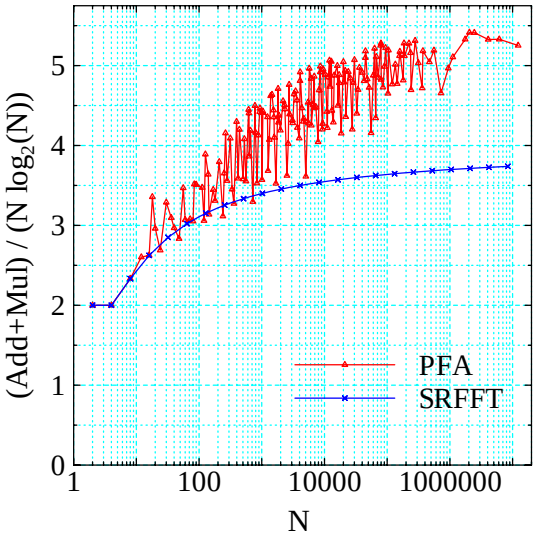


周波数間引きSplit-Radix FFTのデータフロー

素因数ＦＦＴとSplit-Radix ＦＦＴの比較







それぞれのＦＦＴの長所と短所

素因数ＦＦＴ：

- ループ構造がシンプルで素直な設計で高速になる
- 三角関数表のサイズがきわめて少ない
- 素数の長さのDFTモジュールが必要でインプリメントが重労働になる
- 計算できるFFTサイズに制限がある

Split-Radix FFT：

- 設計が比較的容易である
- バタフライ構造が複雑で素直な設計では高速にならない
- 高速計算には相当な工夫がいる

Recursive ＦＦＴ(ＣＰＵキャッシュ利用１)

ＦＦＴのバタフライの実行順序を，再帰的な構造のＦＦＴのバタフライの実行順序に変更する.

Recursive ＦＦＴの特徴：

- ＣＰＵのキャッシュに入りきらないサイズで有効.
- 演算量は変わらない.

4-Step FFT(CPUキャッシュ利用２)

データを$N = N_1 \times N_2, (N_1 \simeq N_2)$の行列とみなし，Cooley-Tukey型分解を行う：

1. それぞれの行に関してFFTを実行する.
2. 回転因子W_N^{jk}を掛ける.
3. 転置する.
4. それぞれの行に関してFFTを実行する.

4-Step FFTの特徴：

- CPUのキャッシュがより有効に利用される.
- 演算量は逆に増えることがある.

Split-Radix FFTの改良(メモリアクセス改善)

DFTの分解($m = 0, 1, 3$):

$$A_{4k+m} = \sum_{j=0}^{N/4-1} \left(\sum_{p=0}^{3} i^{-mp} a_{Np/4+j} \right) W_N^{mj} W_{N/4}^{jk}$$

$$A_{4k+2} = \sum_{j=0}^{N/4-1} \left(\sum_{p=0}^{3} (-1)^p a_{Np/4+j} \right) W_{N/4}^{j(k+1/2)}$$

ずらしDFTの導入:

$$B_k = \sum_{j=0}^{N-1} b_j W_N^{j(k+1/2)}$$

ずらしDFTの分解($m = 0, 1$)：

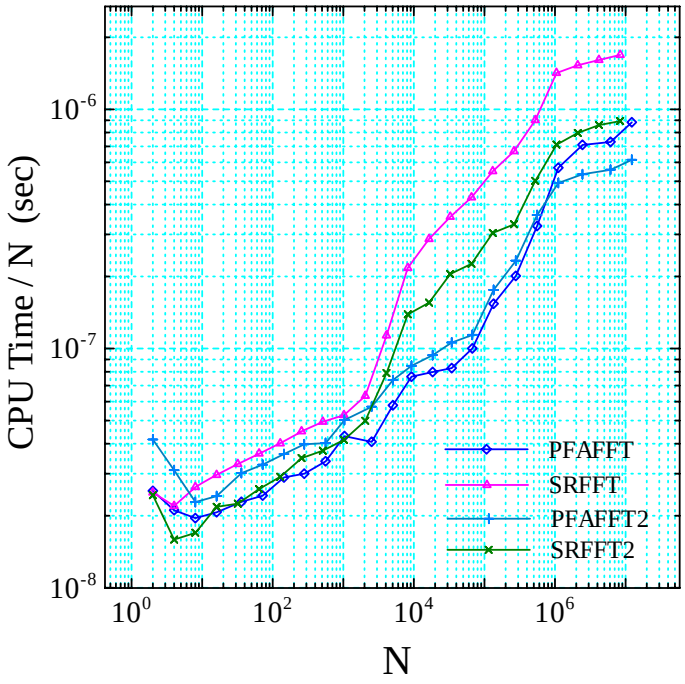
$$B_{4k+m} = \sum_{j=0}^{N/4-1} (c_j^{(m)} + c_{N/4+j}^{(m)}) W_{N/4}^{jk}$$

$$B_{4k+2+m} = \sum_{j=0}^{N/4-1} (c_j^{(m)} - c_{N/4+j}^{(m)}) W_{N/4}^{j(k+1/2)}$$

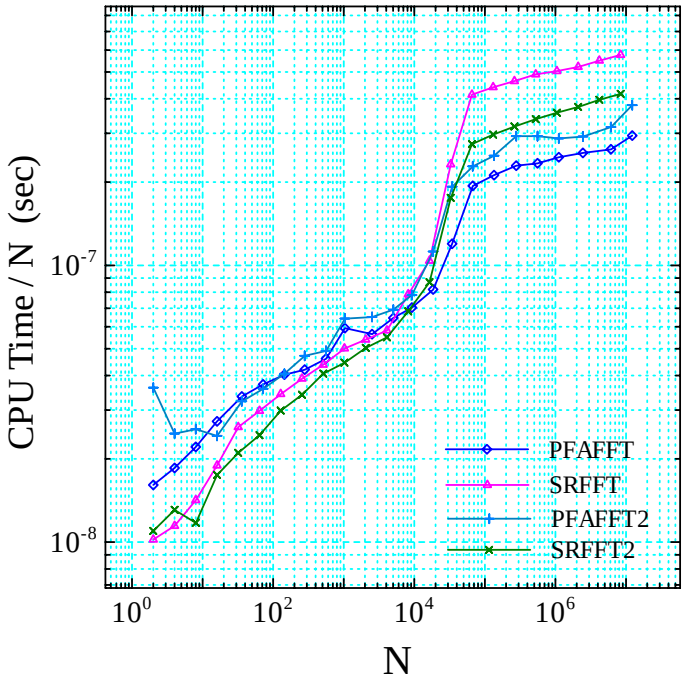
ただし,

$$c_j^{(m)} = (b_j - i(-1)^m b_{N/2+j}) W_N^{j(m+1/2)}$$

性能比較例



Alpha 21264C / 1GHz



Pentium 4 / 2.4GHz

高速なＦＦＴルーチンの入手

Ooura FFT: http://www.kurims.kyoto-u.ac.jp/~ooura/

- 改良版Split-Radix FFTを基本にする
- Recursive FFTを用いる
- メモリアクセスは連続にし，データは上書きする

FFTW: http://www.fftw.org/

- さまざまなFFTアルゴリズムを動的に組み合わせる
- 機種ごとにベンチマークをして最適なFFTの分解を行う

ＦＦＴの使い方
巡回畳み込みの計算法

巡回畳み込み：

$$
\begin{aligned}
c_k &= \sum_{j=0}^{N-1} a_j b_{k-j} \quad (b_{-j} = b_{N-j}) \\
&\Downarrow \mathrm{DFT} \\
C_k &= A_k B_k
\end{aligned}
$$

$\{A_k\}, \{B_k\}, \{C_k\}$：$\{a_j\}, \{b_j\}, \{c_j\}$ の DFT

非巡回畳み込みの計算法

方法１：ゼロをつめる

方法２：負の巡回畳み込み：

$$
\begin{aligned}
c_k &= \sum_{j=0}^{N-1} a_j b_{k-j} \quad (b_{-j} = -b_{N-j}) \\
&\Downarrow \text{ずらしDFT} \\
C'_k &= A'_k B'_k
\end{aligned}
$$

$\{A'_k\}, \{B'_k\}, \{C'_k\}$：$\{a_j\}, \{b_j\}, \{c_j\}$のずらしDFT：

$$
A'_k = \sum_{j=0}^{N-1} a_j W_N^{(j+1/2)k}, \quad W_N = \mathrm{e}^{-2\pi \mathrm{i}/N}
$$

素数の長さのＦＦＴ

Raderの方法：

$$
\begin{aligned}
A(k) &= \sum_{j=0}^{N-1} a(j) W_N(jk), \quad W_N(j) = \mathrm{e}^{-2\pi \mathrm{i} j/N} \\
&\Downarrow \textbf{添字の変換：} k \equiv g^q \ , \ \ j \equiv g^{-p} \quad (g：\text{原始根}) \\
A(g^q) &= a(0) + \sum_{p=0}^{N-2} a(g^{-p}) W_N(g^{q-p})
\end{aligned}
$$

長さNのＤＦＴ⇒長さ$N-1$の巡回畳み込み

ＦＦＴの利用例

1．周波数解析

2．離散畳み込みの計算

3．関数近似（スペクトル法など）

4．etc