問 3　データ圧縮の前処理として用いられる Block-sorting に関する次の記述を読んで，設問 1～4 に答えよ。

Block-sorting は，文字列に対する可逆変換の一種である。変換後の文字列は，変換前の文字列と比較して同じ文字が多く続く傾向があるので，その後に行う圧縮処理において圧縮率を向上させることができる。

Block-sorting は，変換処理と復元処理の二つの処理で構成される。変換処理は，入力文字列を受け取って，変換結果の文字列と，入力文字列がソート後のブロックで何行目にあるか（以下，入力文字列の行番号という）を出力する。一方，復元処理は，変換結果の文字列と入力文字列の行番号を受け取って入力文字列を出力する。

データ圧縮における Block-sorting の使用方法を図 1 に示す。

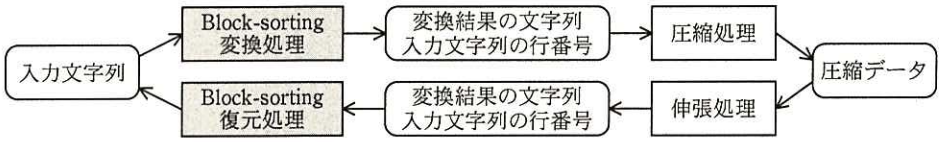
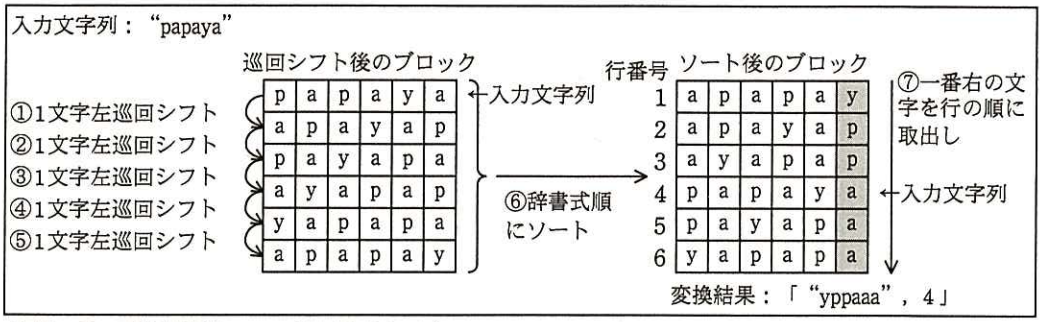


図 1　データ圧縮における Block-sorting の使用方法

〔Block-sorting の変換処理〕

例として“papaya”を入力文字列としたときの変換処理を図 2 に示す。図 2 では，入力文字列を 1 文字左に巡回シフトすること（①）で文字列“apayap”となる。さらに，もう 1 文字左に巡回シフトすること（②）で文字列“payapa”となる。同様に 1 文字ずつ左に巡回シフトした（③～⑤）結果の文字列を縦に並べて正方形のブロック（巡回シフト後のブロック）を作成する。

次に，このブロックを行単位で辞書式順にソートし（⑥），ソート後のブロックを得る。ソート後のブロックの各行の文字列から一番右の文字を行の順に取り出して並べた文字列と，ソート後のブロックにおいて入力文字列に一致する行の行番号を変換結果とする（⑦）。



注記　①～⑦は処理順，1～6 は行番号を示す。

図 2　Block-sorting の変換処理

〔Block-sorting の復元処理〕

図 2 の変換結果「“yppaaa”，4」を復元する手順を表 1 に示す。

表 1　Block-sorting の復元手順

手順	処理	内容
1	変換結果の文字列に対して，各文字に 1 から順に添字を付ける。	“yppaaa” → “y(1),p(2),p(3),a(4),a(5),a(6)”
2	文字をソートする。同じ文字の場合は添字の順に並べる。	“y(1),p(2),p(3),a(4),a(5),a(6)” → “a(4),a(5),a(6),p(2),p(3),y(1)”
3	手順 2 でソートした文字を次の手順で並べる。 ・変換結果の行番号“4”から，ソート後の文字列“a(4),a(5),a(6),p(2),p(3),y(1)”の 4 番目の要素“p(2)”を取り出して並べる。 ・“p(2)”の添字が 2 であることから，2 番目の要素“a(5)”を取り出して並べる。 ・“a(5)”の添字が 5 であることから 5 番目の要素の“p(3)”を取り出して並べる。以降，並べた要素の個数が変換結果の文字列の長さと同じになるまで，要素を取り出して並べることを繰り返す。	→ “p(2)” → “p(2),a(5)” → “p(2),a(5),p(3)” → “p(2),a(5),p(3),a(6)” → “p(2),a(5),p(3),a(6),y(1)” → “p(2),a(5),p(3),a(6),y(1),a(4)”
4	手順 3 の結果から添字を取り除く。	“p(2),a(5),p(3),a(6),y(1),a(4)” → “papaya”

〔Block-sorting の実装〕

Block-sorting のプログラムを作成するために使用する配列，関数及び変数を，表 2 に示す。

表２　使用する配列，関数及び変数

名称	種類	内容
EncodeArray[n]	配列	巡回シフト後のブロックを格納する。ブロックの１行を文字列として，配列の一つの要素に格納する。配列の添字は１から始まる。 例 “papaya” “apayap” “payapa” “ayapap” “yapapa” “apapay”
DecodeArray[2][n]	配列	復元用の文字と添字の組を格納する。配列の添字は１から始まる。 例 “y” “p” “p” “a” “a” “a” 1 2 3 4 5 6
sort1(Array[])	関数	１次元配列 Array[ ]の要素を辞書式順にソートする。
sort2(Array[][])	関数	２次元配列 Array[ ][ ]を，Array[1]の要素をキーにしてソートする。 例 “y” “p” “p” “a” “a” “a” → “a” “a” “a” “p” “p” “y” 1 2 3 4 5 6 → 4 5 6 2 3 1
rotation(String)	関数	文字列 String を１文字左に巡回シフトした結果を返す。
InputString	変数	入力文字列。この文字列の長さを“InputString の長さ”とする。他の文字列変数についても，長さを同様に表す。
BlockSortString	変数	変換結果の文字列。
Line	変数	ソート後のブロックでの入力文字列の行番号。
OutputString	変数	復元処理の出力文字列。

注記　n は入力文字列の長さを表す。

〔変換処理関数 encode〕

変換処理を実装した関数 encode のプログラムを図３に示す。

```
function encode(InputString)
  rString ← InputString
  for( i を [ア] から [イ] まで１ずつ増やす )
    EncodeArray[i] ← rString
    rString ← rotation(rString)
  endfor
  sort1(EncodeArray)
  BlockSortString を空文字列に初期化する
  for( k を [ア] から [イ] まで１ずつ増やす )
    BlockSortString の末尾に EncodeArray[k]の末尾の１文字を追加する
    if( [ウ] )
      Line ← k
    endif
  endfor
endfunction
```

図３　関数 encode のプログラム

〔復元処理関数 decode〕

復元処理を実装した関数 decode のプログラムを図 4 に示す。

```
function decode(BlockSortString, Line)
  for( i を 1 から BlockSortString の長さまで 1 ずつ増やす )
    DecodeArray[1][i] ← BlockSortString の i 文字目
    DecodeArray[2][i] ← i
  endfor
  sort2(DecodeArray)
  OutputString を空文字列に初期化する
  OutputString の末尾に [ エ ] に格納されている 1 文字を追加する
  n ← [ オ ]
  while( [ カ ] )
    OutputString の末尾に DecodeArray[1][n]に格納されている 1 文字を追加する ◀ (α)
    n ← DecodeArray[2][n]
  endwhile
endfunction
```

図 4　関数 decode のプログラム

〔関数 sort2(Array[ ][ ])の実装〕

関数 decode の処理時間は，使用する関数 sort2(Array[ ][ ])の計算量に大きく依存する。処理時間を短くするためには，sort2(Array[ ][ ])の内部で計算量が少ないソートのアルゴリズムを使用して実装する必要がある。

処理時間の違いを確認するために複数のソートアルゴリズムを使用して関数 sort2(Array[ ][ ])を実装したところ，Array[1]の要素をキーにしてクイックソート（不安定なソート）を使用した場合には復元処理の結果が入力文字列と一致しなかった。

この場合，sort2(Array[ ][ ])が表 1 の手順 2 を正しく実装できていないので，(β)ソートアルゴリズム，ソートキーのいずれかを見直す必要がある。

設問 1　文字列“kiseki”に対して Block-sorting を適用して変換した結果を答えよ。変換結果は図 2 の記法に合わせて記述すること。

設問 2　図 3 中の [ア] ～ [ウ] に入れる適切な字句を答えよ。

設問 3　〔復元処理関数 decode〕について，(1)，(2)に答えよ。

(1)　図 4 中の [エ] ～ [カ] に入れる適切な字句を答えよ。

(2)　BlockSortString の長さが p のとき，図 4 中の下線(α)の処理の実行回数を答えよ。

設問4　本文中の下線(β)について，ソートアルゴリズムを見直す場合とソートキーを見直す場合のそれぞれについて，どのように見直せばよいかを 30 字以内で述べよ。