

■ 配列と初期値、終値

繰返して処理することが最も多いのが、配列の要素です。スレッパーを要素番号にして、配列の要素を参照します。

例えば、次のように配列arrayを作成します。

【例】配列array

○整数型の配列: array ← {1, 2, 3, 4, 5}

配列の要素が1から始まる場合には、arrayの要素番号と値の対応は、次のようになります。

| 要素番号 | 1 | 2 | 3 | 4 | 5 |
|------|---|---|---|---|---|
| 要素の値 | 1 | 2 | 3 | 4 | 5 |

配列arrayの要素数は5なので、次のようなプログラムを記述すると、arrayの要素の内容をすべて出力できます。

【例】配列arrayの要素をすべて出力 (要素番号が1から始まる場合)

```
○整数型: i
for (i を 1 から arrayの要素数 まで 1 ずつ増やす)
    array[i] を出力
endfor
```

このプログラムでは、for文は5回実行され、以下のような出力となります。

実行結果

```
1
2
3
4
5
```

しかし、このプログラムの配列arrayの要素番号が0から始まる場合には、うまく動きません。配列の要素が0から始まる場合

うになります。ここで、配列の要素番号は0から始まります。

【例】選択ソートを行う関数 sort

```
○整数型の配列: sort(整数型の配列: data)
  整数型: i, j, min_i
  for (i を 0 から dataの要素数-2 まで 1 ずつ増やす)
    min_i ← i
    for (j を i+1 から dataの要素数-1 まで 1 ずつ増やす)
      if (data[min_i] > data[j])
        min_i ← j
      endif
    endfor
    data ← swap(data, min_i, i)
  endfor
  return data
```

④シェルソート

ある一定間隔おきに取り出した要素から成る部分列をそれぞれ整列させ、さらに間隔を狭めて同様の操作を繰り返し、最後に間隔を1にして完全に整列させるというアルゴリズムです。挿入ソートの発展形で、ざっくり整列させてから細かくしていくので効率が良くなります。間隔は、7, 3, 1……と、 $2^n - 1$ で n を1ずつ減らして狭めていくので、計算量は $O(n \log n)$ となります。

擬似言語でシェルソートを行う関数sortを記述すると、次のようになります。ここで、配列の要素番号は0から始まります。

【例】シェルソートを行う関数 sort

```

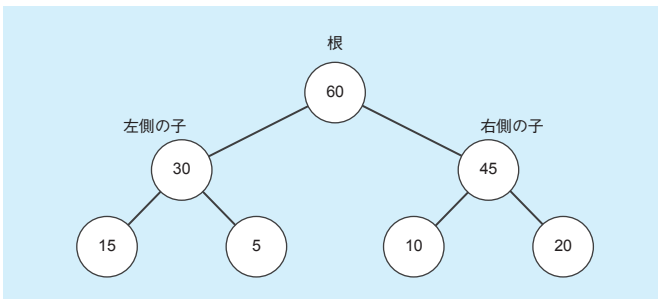
○整数型の配列: sort( 整数型の配列: data)
  整数型の配列: gaps ← {7, 3, 1} /* 間隔をあらかじめ設定 */
  整数型: start, gap, i, j, k
  for (k を 0 から gapsの要素数-1 まで 1 ずつ増やす)
    gap = gaps[k]
    for (start を 0 から gap-1 まで 1 ずつ増やす)
      for (i を start から dataの要素数-1 まで gap ずつ増やす)
        for (j を i-gap から 0 まで gap ずつ減らす)
          if (data[j] > data[j+gap])
            data ← swap(data, j, j+gap)
          else
            break /* 最も内側の for 文を終了 */
        endif
      endfor
    endfor
  endfor
  return data

```

⑤ヒープソート

ヒープとは、2分木の一種で、根から葉に向けてだけデータを整列させたデータ構造です。例えば、図1のような2分木は、根から葉までがすべて降順に整列されているのでヒープです。

●図1 ヒープの例



擬似言語でヒープソートを行う関数sortを記述すると、次のようになります。ヒープソートでは、先にヒープを作成する必要がありますので、ヒープを作成する関数makeHeapを作成します。これまでに定義したswap, rchild, lchild, parentの4つの関数を使用します。ここで、配列の要素番号は0から始まります。

【例】ヒープを作成する関数makeHeap

```
○整数型の配列: makeHeap(整数型の配列: data)
整数型の配列: heap ← {dataの要素数 個の未定義の配列}
整数型: i, k
for (i を 0 から dataの要素数-1 まで 1 ずつ増やす)
    heap[i] ← data[i]
    k ← i
    while (k > 0)
        if (heap[k] > heap[parent(k)])
            heap ← swap(heap, k, parent(k))
            k ← parent(k)
        else
            break /* 内側のwhile文を終了 */
    endif
endwhile
endfor
return heap
```

【例】ヒープソートを行う関数 sort

○整数型の配列: sort(整数型の配列: data)

整数型: last, hlast, n, tmp

data \leftarrow makeHeap(data)

for (last を dataの要素数-1 から 1 まで 1 ずつ減らす)

data \leftarrow swap(data, 0, last)n \leftarrow 0hlast \leftarrow last-1while(lchild(n) \leq hlast)tmp \leftarrow lchild(n)if (rchild(n) \leq hlast)if (data[tmp] \leq data[rchild(n)])tmp \leftarrow rchild(n)

endif

endif

if (data[tmp] > data[n])

data \leftarrow swap(data, n, tmp)

else

break /* 内側のwhile文を終了 */

endif

n \leftarrow tmp

endwhile

endfor

return data

⑥クイックソート

最初に中間的な**基準値**を決めて、それよりも大きな値を集めた部分列と小さな値を集めた部分列に要素を振り分けます。

その後、それぞれの部分列の中で基準値を決めて、同様の操作を繰り返すアルゴリズムです。ランダムなデータの場合には、計算量は $O(n \log n)$ となります。

クイックソートでは再帰を使うので、詳しくは「4-3-1 再帰のアルゴリズム」で説明します。

【例】再帰を用いてマージソートを行う関数sort

```

○整数型の配列: sort(整数型の配列: data)
  整数型の配列: left ← {}
  整数型の配列: right ← {}
  整数型の配列: merge ← {}
  整数型: l ← 0
  整数型: r ← 0
  整数型: mid ← dataの要素数 ÷ 2 の商
  if (dataの要素数 が 1 以下)
    return data
  endif
  /* 分割操作 */
  left ← dataの要素番号 0 から mid-1 までの部分列
  right ← dataの要素番号 mid から dataの要素数-1 までの部分列
  /* 再帰関数で、部分列を整列 */
  left ← sort(left)
  right ← sort(right)
  /* 結合操作 */
  while (l < leftの要素数 and r < rightの要素数)
    if (left[l] ≤ right[r])
      mergeの末尾に left[l] を追加する
      l ← l + 1
    else
      mergeの末尾に right[r] を追加する
      r ← r + 1
    endif
  endwhile
  if (l < leftの要素数)
    mergeの末尾に leftの要素番号l以降を結合
  elseif (r < rightの要素数)
    mergeの末尾に rightの要素番号r以降を結合
  endif
  data ← merge
  return data

```

【例】単純な照合方法による文字列探索の関数 search1

```
○整数型: search1(文字型の配列: text, 文字型の配列: pattern)
整数型: i, j
for (i を 0 から textの要素数-1 まで 1 ずつ増やす)
  for (j を 0 から patternの要素数-1 まで 1 ずつ増やす)
    if (text[i+j] = pattern[j])
      if j = patternの要素数 - 1
        return i
      endif
    else
      break /* 内側のforループを抜ける */
    endif
  endfor
endfor
return -1
```

② BM (Boyer Moore) 法

比較位置を1文字ずつではなく、なるべく多くずらすことで比較回数を減らすアルゴリズムです。パターンを末尾から検索していき、一致しなかった場合に、特定の文字数分だけ進めます。具体的には、パターンの末尾に対応する位置にあるテキストの文字(判定文字)により、次の条件で文字数分だけ進めます。

- ・判定文字がパターンに含まれていない場合には、パターンの文字数分だけ比較位置を進める
- ・判定文字がパターンの末尾以外に含まれている場合には、判定文字と一致するパターンの文字が、テキストの判定文字に対応する位置に来るように比較位置を進める

例えば、パターンの文字列が "ACAB" の場合、判定文字とスキップ数(進める文字数)の関係は次のようになります。

●文字列 "ACAB"

| 判定文字 | A | C | B | その他の文字 |
|-------|---|---|---|--------|
| スキップ数 | 1 | 2 | 4 | 4 |

スキップ数の計算も含めた、BM法での文字列探索の関数 `search2(text, pattern)` は、次のように記述できます。

【例】BM法による文字列探索の関数 `search2`

```
○整数型: search2(文字型の配列: text, 文字型の配列: pattern)
整数型: i, j, skip
真偽型: found
文字型の配列: skip_char ← {pattern[0]}
整数型の配列: skip_num ← {patternの要素数}
/* スキップ数を計算 */
for (i を 1 から patternの要素数-1 まで 1 ずつ増やす)
    found ← false
    for (j を 0 から skip_charの要素数-1 まで 1 ずつ増やす)
        if (pattern[i] = skip_char[j])
            skip_num[j] ← patternの要素数 - i
            found ← false
        endif
    endfor
    if (not found)
        skip_charの末尾に pattern[i] を追加
        skip_numの末尾に patternの要素数 - i を追加
    endif
endfor
/* 文字列の探索 */
i ← 0
while (i < (textの要素数 - patternの要素数))
    for (j を patternの要素数-1 から 0 まで 1 ずつ減らす)
        if (text[i+j] = pattern[j])
            if j = 0
                return i
            endif
        else
            break /* 内側のforループを抜ける */
        endif
    endfor
```



```

/* スキップ数の決定 */
skip ← patternの要素数
for (j を 0 から skip_charの要素数-1 まで 1 ずつ増やす)
    if (text[i+patternの要素数-1] = skip_char[j])
        skip ← skip_num[j]
        break /* 内側のforループを抜ける */
    endif
endfor
i ← i + skip
endwhile
return -1

```

この関数では、スキップ数について2つの配列 skip_char と skip_num で保持しています。

■ 最大公約数を求めるアルゴリズム

最大公約数 (GCD : Greatest Common Divisor) を求める定番アルゴリズムに、ユークリッドの互除法があります。2つの自然数 a , b で $a \geq b$ のとき、 $a \bmod b$ で求めた余りを r とします。 a と b の最大公約数は、 b と r の最大公約数と等しくなるという性質があります。これを利用し、今度は $b \bmod r$ を求めて余りを求める、ということを繰り返し、余りが0になるまで続けると、最大公約数を求めることができます。

ユークリッドの互除法で、引数で与えられた2つの正の整数 a と b の最大公約数を求める関数 gcd は、次のようになります。

【例】ユークリッドの互除法で最大公約数を求める関数 gcd

```

○整数型: gcd(整数型: a, 整数型: b)
整数型: r, tmp
if (a < b) /* a > b となるように入れ替え */
    tmp ← a
    a ← b
    b ← tmp
endif
r ← a mod b

```

4-3-4 演習問題

問1 再帰的に定義された関数

CHECK ☐☐☐

三つのスタック A, B, C のいずれの初期状態も {1, 2, 3} であるとき, 再帰的に定義された関数 f() を呼び出して終了した後の B の値はどれか。ここで, スタックは以下のクラス Stack を用いて定義することとする。

表 クラス Stack の説明

| メンバ変数 | 型 | 説明 |
|-----------------------|--------|---------------------------------------|
| data | 整数型の配列 | スタックの値 |
| コンストラクタ | | 説明 |
| Stack (整数型の配列: qData) | | 引数qDataでメンバ変数dataを初期化する |
| メソッド | 戻り値 | 説明 |
| push(整数型: value) | なし | dataの末尾にvalueを追加する |
| pop() | 整数型 | dataの末尾を取り出して返す。取り出したデータはdataから取り除かれる |
| empty() | 真偽型 | dataの要素数が0ならtrue, 0でないならfalseを返す |

[関数 f() のプログラム]

```
○関数: f( )
  if (A.empty())
    return
  else
    C.push(A.pop())
    f()
    B.push(C.pop())
  return
endif
```

[実行手順]

```
大域: 整数型の配列 A ← stack({1, 2, 3})
大域: 整数型の配列 B ← stack({1, 2, 3})
大域: 整数型の配列 C ← stack({1, 2, 3})
f()
```

同様に、iが3の3行目{0, 0, 0, 1, 3}を見ていくと、値が0でないのはjが4のときと5のときで、値は4のときが1, 5のときが3です。そのため、次のように値を追加します。

| sparseMatrix | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 3 | 3 |
| 2 | 1 | 2 | 3 | 4 | 5 |
| 3 | 3 | 2 | 2 | 1 | 3 |

同様に、iが4の4行目{0, 0, 0, 2, 0}, iが5の5行目{0, 0, 0, 0, 1}では、iが4, jが4のときのmatrix[4, 4]=2, iが5, jが5のときのmatrix[5, 5]=1が0以外の値となります。追加すると、次のようになります。

| sparseMatrix | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|--------------|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 3 | 3 | 4 | 5 |
| 2 | 1 | 2 | 3 | 4 | 5 | 4 | 5 |
| 3 | 3 | 2 | 2 | 1 | 3 | 2 | 1 |

以上より、空欄aはsparseMatrix[1]で{1, 2, 2, 3, 3, 4, 5}, 空欄bはsparseMatrix[2]で{1, 2, 3, 4, 5, 4, 5}, 空欄cはsparseMatrix[3]で{3, 2, 2, 1, 3, 2, 1}となります。したがって、組合せの正しいイが正解です。

問3 文字コード変換

《解答》ク

Unicodeの符号位置を、UTF-8の符号に変換する関数encodeに対する空欄穴埋め問題です。問題文の内容を、プログラムに当てはめて考えていきます。

〔プログラム〕の1行目で、関数encodeは「encode(整数型: codePoint)」となっているので、引数codePointにUnicodeの符号位置が渡されます。例えば、問題文にある、ひらがなの“あ”の符号位置は3042 (16)で、2進数で表すと11000001000010となります。10進数に変換すると、12354です。この値を、UTF-8の符号に変換していきます。

関数の最初の行「整数型の配列: utf8Bytes ← {224, 128, 128}」は、コメントに説明があるとおり、2進数を変換した値です。2進数で表すと、utf8Bytesの各要素番号の値は、次のようになります。

| 要素番号 | 1 | 2 | 3 |
|-------|----------|----------|----------|
| 2進数の値 | 11100000 | 10000000 | 10000000 |