情報処理概論
(Basic Theory of Information Processing)

第７回：配列

概要

- 配列
- 配列変数の宣言
- 領域確保
- 多次元変数

10　配列 (p.178)

- **ベクトル・行列のようなもの．**
- 数多くのデータを，**整数のインデクスを使って**扱うことができる．
- Javaでは配列はクラスとして実現されている．

例：

```
a[0] = 1;                ⇐ 1次元配列
a[4] = 7;
i = 2;
j = 8;
a[i] = j;
b[0][0] = 1;             ⇐ 2次元配列
b[2][3] = 10;
b[i][j] = a[2] * 3;
x = a[2];
a[2] = b[i][3] * x;
```

のように使うことができる．

例：

```
public class Array0 {
  public static void main(String[] args) {
    int iArray[] = new int[5];
    iArray[0] = 3;
    iArray[1] = 1;
    iArray[2] = 4;
    iArray[3] = 1;
    iArray[4] = 5;
    for (int i = 0 ; i < 5 ; ++i) {
      System.out.println("iArray[" + i + "] = " + iArray[i]);
    }
  }
}
```

あたりまえであるが,

```
    int a0, a1, a2;
```

と宣言して，i = 1;として，aiを参照しても，a1を意味しない。単に，識別子がaiという変数として参照される。

10.1　配列変数の宣言

- 配列を使うためには，配列変数を宣言する．
- [] を使って，配列変数であることを示す．

例：以下の配列変数を宣言する．

- int型1次元配列の配列変数a，e，f
- int型2次元配列の配列変数b
- int型3次元配列の配列変数c
- double型1次元配列の配列変数x
- double型2次元配列の配列変数y

```
int      a[], b[][], c[][][];
int[]    e, f;
double   x[], y[][];
```

右上の宣言で，２行めは，

```
int      e[], f[];
```

と同じ．

10.2　領域の確保

- 配列変数には，配列のデータそのものではなく，配列のデータが格納されている場所を示す値(メモリーのアドレスみたいなもの)が格納されている．⇐ **参照型変数**
- 実際にデータを格納する領域をメモリー上に確保する必要がある．
  ⇒ new キーワードを使う．
- 通常，配列のデータはその領域の中に順番に格納される．

領域を確保するプログラム例：

```
a = new int[10];
b = new int[20][30];
x = new double[40][50];
```

- はじめの例は，10個分の int の領域を確保して，配列変数 a がその領域を使うことができるようにしている．a[0] から a[9] まで使うことができる．
- 次の例は，20×30 個分の int 型の領域を確保して，配列変数 b に割り当てている．b[0][0] から b[19][29] まで使うことができる．
- 3つめの例は，40×50 個分の double 型の領域を確保して，配列変数 x に割り当てている．x[0][0] から x[39][49] まで使うことができる．
- (２次元配列のデータ構造は１次元配列を組み合わせてできている．)

- 宣言時に領域を確保して初期化することもできる.

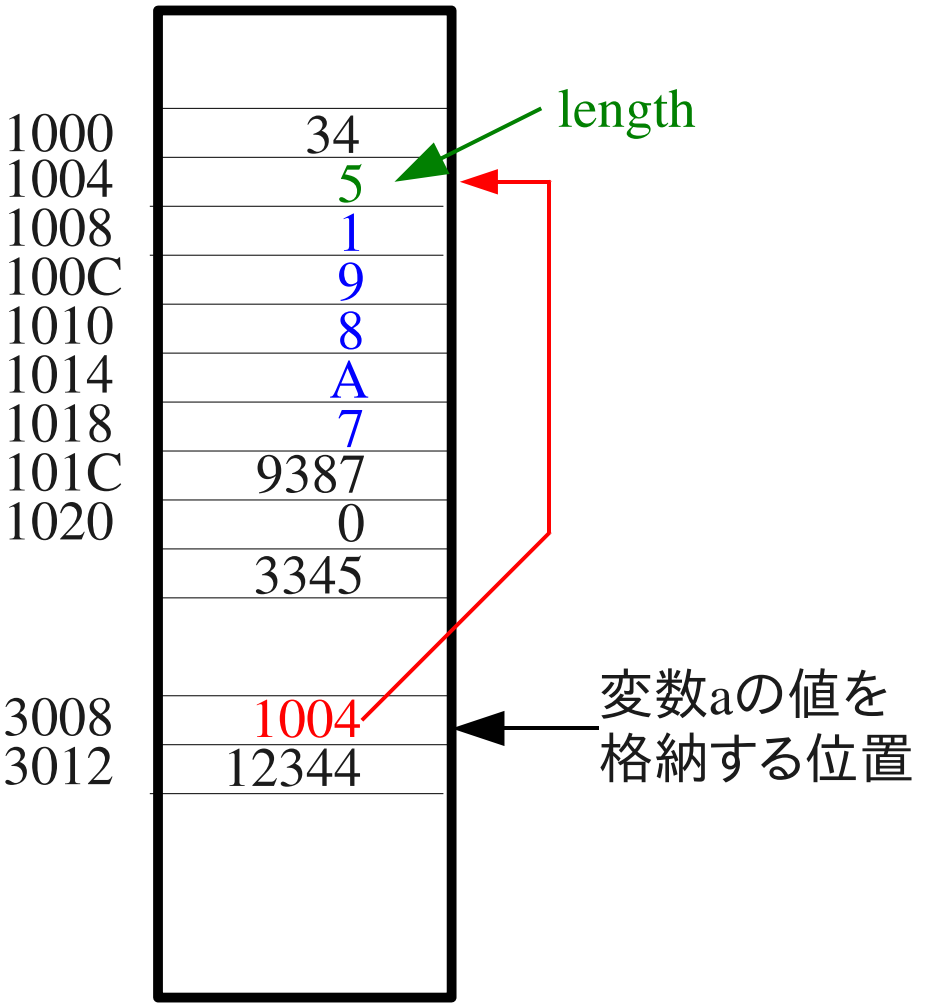
```
int[] a = {1,9,8,10,7};
```

上のプログラムは,

```
int a[];
a = new int[5];
a[0] = 1;
a[1] = 9;
a[2] = 8;
a[3] = 10;
a[4] = 7;
```

と書くことと等しい.

右の場合のメモリーのイメージ図



注意：Javaの参照は，バーチャルマシンの関係でもう少し複雑である。

10.3 長さ

- 1次元配列の大きさは，フィールドlengthによってわかる．
- プログラム例

```
class ArrayLength {
  public static void main(String[] args) {
    int[] a, b;
    a = new int[2];
    b = new int[4];
    int[] c = {1, 2, 3, 4, 5};
    System.out.println("a.length = " + a.length);
    System.out.println("b.length = " + b.length);
    System.out.println("c.length = " + c.length);
  }
}
```

- 出力結果

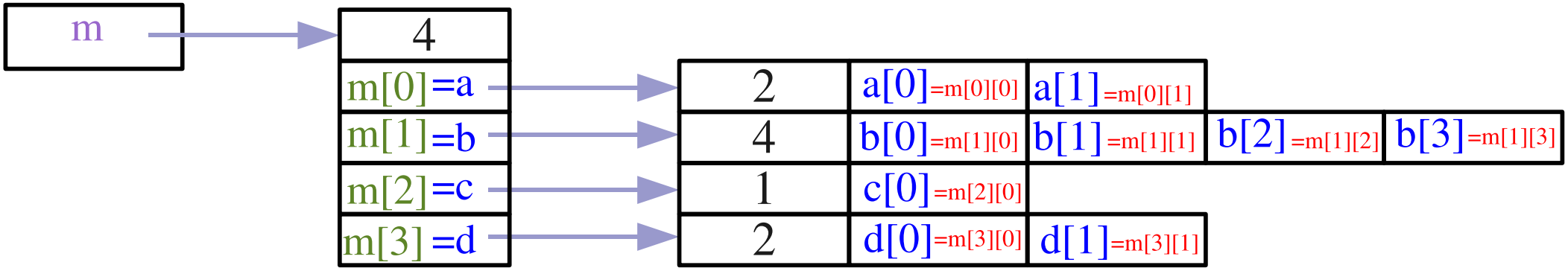
```
a.length = 2
b.length = 4
c.length = 5
```

10.4　多次元配列

- 多次元配列は，１次元配列を組み合わせて作られている．
- 2次元配列は，1次元配列が複数と，1次元配列の配列から構成されている．

```
int[][] m;              ⇐ 2次元配列変数 m
int[] a, b, c, d;       ⇐ 1次元配列変数 a, b, c, d
m = new int[4][];
a = new int[2]; b = new int[4]; c = new int[1]; d = new int[2];
m[0] = a; m[1] = b; m[2] = c; m[3] = d; ⇐ mは1次元配列の配列
```

データ構造のイメージ図



プログラム例

```
class TwoArray1 {
 public static void main(String[] args) {
   int[][] m;              ⇐ 2次元配列変数 m
   int[] a, b, c, d;       ⇐ 1次元配列変数 a, b, c, d
   m = new int[4][];
   a = new int[2]; b = new int[4]; c = new int[1]; d = new int[2];
   m[0] = a; m[1] = b; m[2] = c; m[3] = d; ⇐ mは1次元配列の配列
   System.out.println("m.length = " + m.length);
   for (int i = 0 ; i < 4 ; ++i) {
     System.out.println("m[" + i + "].length = " + m[i].length);
   }
 }
}
```

結果

```
m.length = 4
m[0].length = 2
m[1].length = 4
m[2].length = 1
m[3].length = 2
```

```
class TwoArray2 {
  public static void main(String[] args) {
    int[][] m;
    int[] a, b, c, d;
    m = new int[4][];
    a = new int[2]; b = new int[4]; c = new int[1]; d = new int[2];
    System.out.println("d[1] = " + d[1]);
    m[0] = a; m[1] = b; m[2] = c; m[3] = d; b[3] = 6; m[3][1] = 5;
    for (int i = 0 ; i < 4 ; ++i) {
      for (int j = 0 ; j < m[i].length ; ++j) {
        System.out.println("m[" + i + "][" + j + "] = " + m[i][j]);
    } }
  System.out.println("");
  System.out.println("d[1] = " + d[1]);
} }
```

結果

```
d[1] = 0
m[0][0] = 0
m[0][1] = 0
m[1][0] = 0
m[1][1] = 0
m[1][2] = 0
m[1][3] = 6
m[2][0] = 0
m[3][0] = 0
m[3][1] = 5

d[1] = 5
```

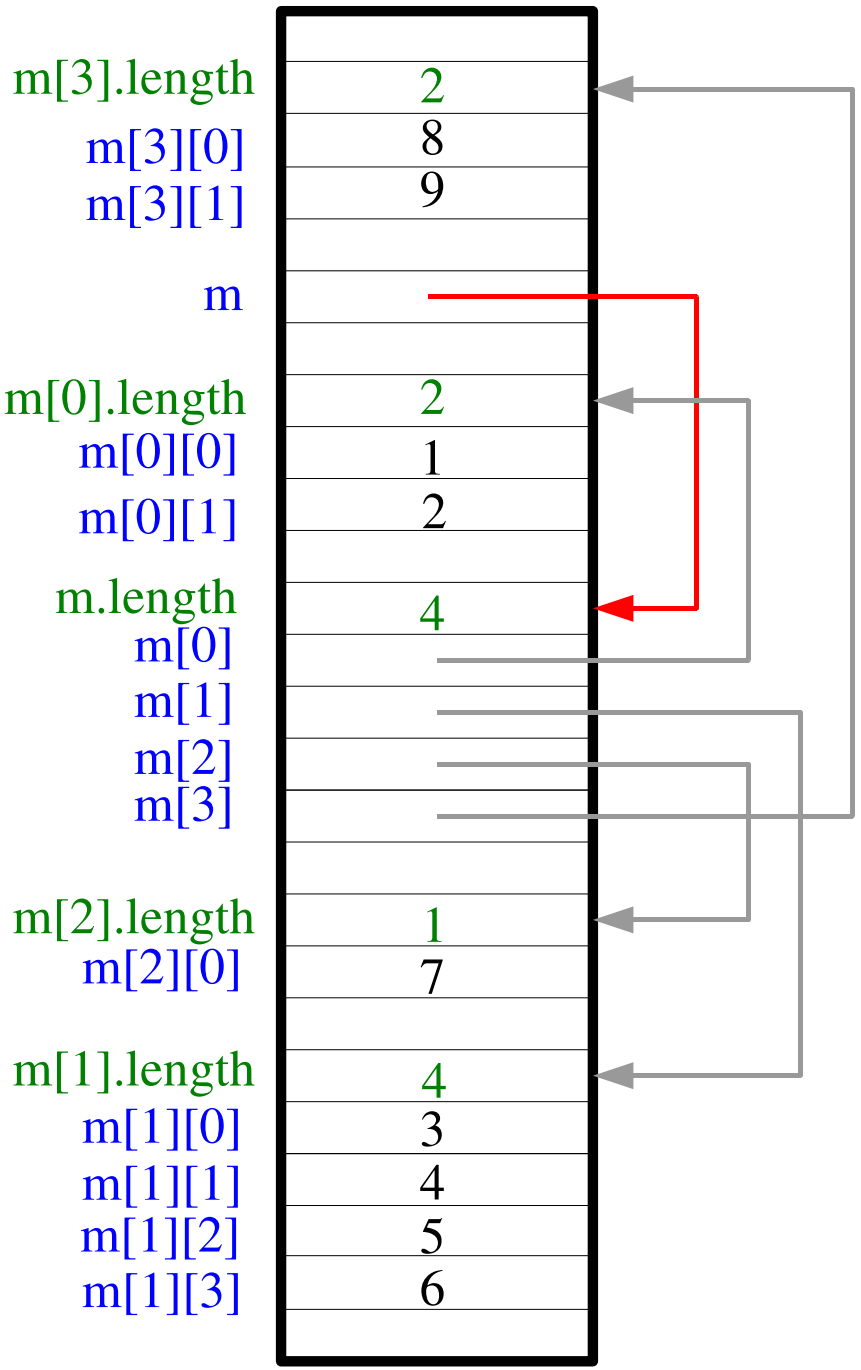
初期化

```
class TwoArray3 {
  public static void main(String[] args) {
    int[][] m = {{1, 2}, {3, 4, 5, 6}, {7}, {8, 9}};
    for (int i = 0 ; i < 4 ; ++i) {
      System.out.println("m[" + i + "].length = " + m[i].length);
      for (int j = 0 ; j < m[i].length ; ++j) {
        System.out.println("m[" + i + "][" + j + "] = " + m[i][j]);
      }
    }
  }
}
```

結果

```
m[0].length = 2
m[0][0] = 1
m[0][1] = 2
m[1].length = 4
m[1][0] = 3
m[1][1] = 4
m[1][2] = 5
m[1][3] = 6
m[2].length = 1
m[2][0] = 7
m[3].length = 2
m[3][0] = 8
m[3][1] = 9
```

メモリーにおけるイメージ図



11　配列の応用

- 行列の計算
- 素数の計算
- ソート
  - – 選択ソート
  - △ バブルソート
  - △ クイックソート
  - △ マージソート
  - △ ヒープソート

  (△印は，概略だけ説明する。)

11.1 ベクトル・行列計算

- **科学技術計算の大部分は，線型代数の計算である．**
  - 構造解析，流体解析
  - 電磁界解析，回路解析
  - 画像処理
- ベクトル計算が速くなるようになっている．
  - ベクトルプロセッサ (スーパーコンピュータ)
  - ベクトル計算命令 (SSE: Streaming Single instruction multiple data Extensions, AVX: Advanced Vector eXtensions)
  - GPU (Graphics Processing Unit)
- 線形代数の表現
  - １次元配列：ベクトル
  - ２次元配列：行列
  - ３次元配列：テンソル
- 高速化のために，行列の掛け算といえどもそのためのプログラムは肥大化の一途．
  ⇒ ライブラリーの利用も重要である。
  BLAS (Basic Linear Algebra Subprograms)
  LAPACK (Linner Algebra Package)

11.1.1　ベクトルの加算

N 次元ベクトルの加算：

$$\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$$

成分で書けば，$i = 1, 2, \ldots, N$ に対して，

$$z_i = x_i + y_i$$

```
public class AddVect { // AddVect.java
  public static void main(String[] args) {
    int   N = 4;
    int[] a = {1, 3, 4, 5}, b = {3, 9, 4, 5};
    int[] c = new int[4];
    for (int i = 0 ; i < N ; ++i) { // ベクトルの加算
      c[i] = a[i] + b[i];
    }
    for(int i = 0 ; i < N ; ++i) {
      System.out.println("c[" + i + "] = " + c[i]);
    }
  }
}
```

11.1.2　ベクトルの内積

N次元ベクトルの加算：

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle$$

成分で書けば,

$$\sum_{i=1}^{N} x_i y_i$$

```
public class InPro { // InPro.java
  public static void main(String[] args) {
    int   N = 4;
    int[] a = {1, 3, 4, 5}, b = {3, 9, 4, 5};
    int   inPro = 0;
    for(int i = 0 ; i < N ; ++i) {
      inPro += a[i] * b[i];
    }
    System.out.println("inPro = " + inPro);
  }
}
```

11.1.3　行列の加算

$M \times N$ 行列の加算：

$$C = A + B$$

成分で書けば，$i = 1, 2, \ldots, M$，$j = 1, 2, \ldots, N$ に対して，

$$C_{ij} = A_{ij} + B_{ij}$$

```
public class AddMat { // AddMat.java
  public static void main(String[] args) {
    int     M = 3, N = 4;
    int[][] A = {{1, 3, 2, 7}, {4, 5, 1, -3}, {7, 4, -3, -2}};
    int[][] B ={{3, 5, 2, 7}, {3, 2, 5, -7}, {3, 2, -4, 2}};
    int[][] C = new int[M][N];
```

```
    for(int i = 0 ; i < M ; ++i) {
      for(int j = 0 ; j < N ; ++j) {
        C[i][j] = A[i][j] + B[i][j];
      }
    }
    // 表示
    for(int i = 0 ; i < M ; ++i) {
      for(int j = 0 ; j < N ; ++j) {
        System.out.println("C[" + i + "][" + j + "] = " + C[i][j]);
      }
    }
  }
}
```

11.1.4　行列の転置

$M \times N$ 行列の転置：

$$C = A^T$$

成分で書けば，$i = 1, 2, \ldots, N$，$j = 1, 2, \ldots, MN$ に対して，

$$C_{ij} = A_{ji}$$

```
public class TransMat { // Transposition.java
  public static void main(String[] args) {
    int     M = 3, N = 4;
    int[][] A = {{1, 3, 2, 7}, {4, 5, 1, -3}, {7, 4, -3, -2}};
    int[][] B = new int[N][M];
    for(int i = 0 ; i < N ; ++i) {
      for(int j = 0 ; j < M ; ++j) {
        B[i][j] = A[j][i];
      }
    }
    for(int i = 0 ; i < N ; ++i) {
      for(int j = 0 ; j < M ; ++j) {
        System.out.println("B[" + i + "][" + j + "] = " + B[i][j]);
      }
    } } }
```

11.1.5　行列とベクトルの積

$M \times N$ 行列と N 次元ベクトルの積：

$$\boldsymbol{y} = A\boldsymbol{x}$$

成分で書けば,

$$y_i = \sum_{j=1}^{N} A_{ij} x_j$$

```
public class MulMatVect { // MulMatVect.java
  public static void main(String[] args) {
    int     M = 3, N = 2;
    int[][] A = {{1, 3}, {4, 5}, {7, 4}};
    int[] x = {3, 9};
    int[] y = new int[M];
    for(int i = 0 ; i < M ; ++i) {
      int sum = 0;
      for(int j = 0 ; j < N ; ++j) {
        sum += A[i][j] * x[j];
      }
      y[i] = sum;
    }
    for(int i = 0 ; i < M ; ++i) {
      System.out.println("y[" + i + "] = " + y[i]);
    }
  }
}
```

11.1.6　行列と行列の積

$M \times K$ 行列と $K \times N$ 行列の積：

$$C = AB$$

成分で書けば,

$$C_{ij} = \sum_{k=1}^{K} A_{ik} B_{kj}$$

```
public class MulMat { // MulMatMat.java
  public static void main(String[] args) {
    int     M = 3, K = 2, N = 4;
    int[][] A = {{1, 3}, {4, 5}, {7, 4}};
    int[][] B = {{3, 9, 4, 5}, {3, 7, 1, 2}} ;
    int[][] C = new int[M][N];
```

```
    for(int i = 0 ; i < M ; ++i) {
      for(int j = 0 ; j < N ; ++j) {
        int sum = 0;
        for(int k = 0 ; k < K ; ++k) {
          sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
      }
    }
    for(int i = 0 ; i < M ; ++i) {
      for(int j = 0 ; j < N ; ++j) {
        System.out.println("C[" + i + "][" + j + "] = " + C[i][j]);
      }
    }
  }
}
```

11.2　配列を使って，素数を求めるプログラム

- 以前に作成した素数を判定するプログラムでは，1からその数までの間の全ての数で割りきれないことを確かめていた。
- しかしながら，素数であるかどうか判定するためには，全ての数でなくその数までの素数で割りきれないことを確かめれば良い。
- **今まで求めた素数を配列に格納し**，配列から取り出して割り切れないことを確かめる。
- 次回の実習の課題とするので，プログラムを考えておくこと。

11.3　ソート

目的

- 配列に格納されている番号などを昇順または降順に並び換える.
- １つのデータが複数の項目から構成されて，その中の項目のいくつかをキーにして，それ以外の項目もいっしょに並び替える場合もある.
  例：学籍番号，名前，点数からなるデータに対して，点数をキーにして並び替える.

代表的手法

- **バカソート (通称)**
- **選択ソート**
- **バブルソート**
- **クイックソート**
- **マージソート**
- **ヒープソート**

以下，昇順に並び換えるアルゴリズムを紹介する.
並び替えるデータは，`int[] A = new int[N]` 全体とする.

11.3.1　バカソート

アルゴリズム

1. 配列の全ての並び換えを全て生成する.
2. その中で昇順になっているものを選び出す.

計算量

$N!$のオーダーの計算量が必要となる.

11.3.2　選択ソート

アルゴリズム

1. `l = 0`とする.
2. `A[l]`から`A[N-1]`の中から，最小の要素を探し出す(それを`A[m]`とする).
3. `A[l]`と`A[m]`を入れ替える.
4. `l = l + 1`とする.
5. `l == N - 1`ならば処理を終了する．そうでなければ，2へ戻る.

計算量

N^2のオーダーの計算量が必要となる.

選択ソートの実施例

- 与えられた配列：

4	7	3	5	8	2	6	1

- 最小の要素を選び出す．(1 = 0)

4	7	3	5	8	2	6	**1**

- 先頭の要素とその最小の要素を入れ替える．

1	7	3	5	8	2	6	**4**

- 2番目から最後までの最小の要素を選び出す．(1 = 1)

1	7	3	5	8	**2**	6	4

- 2番目の要素とその最小の要素を入れ替える．

1	**2**	3	5	8	**7**	6	4

- 3番目から最後までの最小の要素を選び出す．(1 = 2)

1	**2**	**3**	5	8	7	6	4

- 3番目の要素とその最小の要素を入れ替える．

1	**2**	**3**	5	8	7	6	4

- 4番目から最後までの最小の要素を選び出す．(1 = 3)

1	2	3	5	8	7	6	4

- 4番目の要素とその最小の要素を入れ替える．

1	2	3	4	8	7	6	5

- 5番目から最後までの最小の要素を選び出す．(1 = 4)

1	2	3	4	8	7	6	5

- 5番目の要素とその最小の要素を入れ替える．

1	2	3	4	5	7	6	8

- 6番目から最後までの最小の要素を選び出す．(1 = 5)

1	2	3	4	5	7	6	8

- 6番目の要素とその最小の要素を入れ替える．

1	2	3	4	5	6	7	8

- 7番目から最後までの最小の要素を選び出す．(1 = 6)

1	2	3	4	5	6	7	8

- 7番目の要素とその最小の要素を入れ替える．

1	2	3	4	5	6	7	8

11.3.3　選択ソートのプログラム

- 整数を昇順に並び替える。

```
public class SortAsc {
  public static void main(String[] args) {
    int data[] = {1, 3, 5, -2, -3, 6, -9};
    for (int k = 0 ; k < data.length - 1 ; ++k) {
      int minInd = k;
      for (int l = k + 1 ; l < data.length ; ++l) {
        if (data[minInd] > data[l]) minInd = l;
      }
      int min = data[minInd];
      data[minInd] = data[k];
      data[k] = min;
    }
    for (int k = 0 ; k < data.length ; ++k) {
      System.out.println((k + 1) + "番目に小さな値は," + data[k] + "です。");
    }
  }
}
```

- 整数を**降順 (大きいもの順)** に並び替えるプログラムは，次回の演習で作成する。

11.3.4　バブルソート

1. `l = 0;　m = 1;` とする.

2. `l` が負ならば, `l = m;　m = m + 1;` とする.
3. `l == N - 1` ならばプログラムを終了する. そうでないならば4.に進む。
4. `A[l] <= A[l + 1]` ならば, `l = m; m = m + 1;` とする.
   そうでないならば,
   - `A[l]` と `A[l + 1]` を入れ換える.
   - `l = l - 1` とする.
5. 2にもどる.

計算量

N^2 のオーダーの計算量が必要となる.

- 与えられた配列：(l = 0, m = 1)

4	7	3	5	8	2	6	1

- 1番目と2番目を比べる．(l = 0, m = 1)

4	**7**	3	5	8	2	6	1

- 順序は合っているので，2番目と3番目を比べる．(l = 1, m = 2)

4	**7**	**3**	5	8	2	6	1

- 順序が反対なので入れ替える．(結果：l = 0, m = 2)

4	**3**	**7**	5	8	2	6	1

- 1番目と2番目を比べる．(l = 0, m = 2)

4	**3**	**7**	5	8	2	6	1

- 順序が反対なので入れ替える．(結果：l = -1, m = 2)

3	**4**	**7**	5	8	2	6	1

- 先頭まできた(l = -1)ので，入れ替えをはじめた(l = 1)次のところ(l = 2)に戻り，3番目と4番目を比べる．(l = 2, m = 3)

3	4	**7**	**5**	8	2	6	1

- 順序が反対なので入れ替える．(結果：l = 1, m = 3)

3	4	**5**	**7**	8	2	6	1

- 2番目と3番目を比べる．(l = 1, m = 3)

3	4	5	7	8	2	6	1

- 順序があっているので，入れ替えをはじめた(l = 2)次のところ(l = 3)に戻り，4番目と5番目を比べる．(結果：l = 3, m = 4)

3	4	5	7	8	2	6	1

- 順序があっているので，次に進み，5番目と6番目を比べる．(l = 4, m = 5)

3	4	5	7	8	2	6	1

- 順序が反対なので入れ替える．(結果：l = 3, m = 5)

3	4	5	7	2	8	6	1

- 4番目と5番目を比べる．(l = 3, m = 5)

3	4	5	7	2	8	6	1

- 順序が反対なので入れ替える．(結果：l = 2, m = 5)

3	4	5	2	7	8	6	1

- 3番目と4番目を比べる．(l = 2, m = 5)

3	4	5	2	7	8	6	1

- 順序が反対なので入れ替える．(結果：l = 1, m = 5)

3	4	2	5	7	8	6	1

以下，実行結果だけ示す．

3	4	2	5	7	8	6	1
3	2	4	5	7	8	6	1
3	2	4	5	7	8	6	1
2	3	4	5	7	8	6	1
2	3	4	5	7	8	6	1
2	3	4	5	7	6	8	1
2	3	4	5	7	6	8	1
2	3	4	5	6	7	8	1
2	3	4	5	6	7	8	1
2	3	4	5	6	7	8	1

2	3	4	5	6	7	1	8
2	3	4	5	6	7	1	8
2	3	4	5	6	1	7	8
2	3	4	5	6	1	7	8
2	3	4	5	1	6	7	8
2	3	4	5	1	6	7	8
2	3	4	1	5	6	7	8
2	3	4	1	5	6	7	8
2	3	1	4	5	6	7	8
2	3	1	4	5	6	7	8
2	1	3	4	5	6	7	8
2	1	3	4	5	6	7	8
1	2	3	4	5	6	7	8

11.3.5　クイックソート

アルゴリズム

1. はじめに，配列全部を分割対象要素列する．
2. 分割対象要素列の要素数が１以下ならばreturnする．
3. ある値をピボットとして決める (要素列の中央値だと最も効率的になる)。
4. 分割対象要素列の配列に，ピボットよりも小さい要素を前方から，ピボットより大きな要素を後方から詰めて行く．また，ピボットと同じ大きさのものは，後からと前から交互に詰めていく。(その結果，その要素の値がピボット以下の列と，ピボット以上の列に分けられいることになる．)
5. 分割によってできた2つの要素列に対して，それぞれ2からの処理を実行する．
6. 上の処理が終わったらreturnする．

このアルゴリムからreturnされると配列がソートされていることになる．

計算量

$N \log N$ から N^2 のオーダーの計算量が必要となる．

- 次の例では，単に**分割対象要素列の先頭要素をピボットにしている。**

- 与えられた配列：

4	7	3	5	8	2	6	1

- 先頭の４をピボットとする．

4		4	7	3	5	8	2	6	1

- 並び替えのため，先頭の４を取り出す．

4	4		7	3	5	8	2	6	1

- ４は４と等しく，その１番目なので，配列の一番最後の１を取り出し，４を置く．

4	1		7	3	5	8	2	6	4

- １は４より小さいので，配列の先頭におき，その次の７を取り出す．

4	7	1		3	5	8	2	6	4

- ７は４より大きいので，配列の最後から２番めの６を取り出し，７を置く．

4	6	1		3	5	8	2	7	4

- ６は４より大きいので，配列の最後から３番めの２を取り出し，６を置く．

4	2	1		3	5	8	6	7	4

- ２は４より小さいので，配列の先頭から２番めにおき，その次の３を取り出す．

4	3	1	2		5	8	4	7	4

- ３は４より小さいので，配列の先頭から３番めにおき，その次の５を取り出す．

4	5	1	2	3		8	6	7	4

- ５は４より大きいので，配列の最後から４番めの８を取り出し，５を置く．

4	8	1	2	3		5	6	7	4

- 最後に，８は４より大きいので，配列の最後から５番めに８を置く．

4		1	2	3	8	5	6	7	4

これで，前の方が４以下で，後の方が４以上と２つのグループに分かれた．これを，それぞれに繰り返す．以後，結果だけを記す．

1	1		2	3	8	5	6	7	4
1	3		2	1	8	5	6	7	4
1	2		3	1	8	5	6	7	4
1		2	3	1	8	5	6	7	4

分割は生じなかった．

- 次に，ピボットを２で分割

2	2		3	1	8	5	6	7	4
2	1		3	2	8	5	6	7	4
2	3	1		2	8	5	6	7	4
2		1	3	2	8	5	6	7	4

分割された前方のグループの要素数が１になったので，そのグループの分割を終了する．

- ２と３の分割に移る．

3	3	1		2	8	5	6	7	4
3	2	1		3	8	5	6	7	4
3		1	2	3	8	5	6	7	4

２と３が要素１つだけになったので，分割を終了する．

- 先の後半部分に戻る．

8	8	1	2	3		5	6	7	4
8	4	1	2	3		5	6	7	8
8	5	1	2	3	4		6	7	8
8	6	1	2	3	4	5		7	8
8	7	1	2	3	4	5	6		8
8		1	2	3	4	5	6	7	8
4	4	1	2	3		5	6	7	8
4	7	1	2	3		5	6	4	8
4	6	1	2	3		5	7	4	8
4	5	1	2	3		6	7	4	8
4		1	2	3	5	6	7	4	8

5	5	1	2	3		6	7	4	8
5	4	1	2	3		6	7	5	8
5	6	1	2	3	4		7	5	8
5	7	1	2	3	4		6	5	8
5		1	2	3	4	7	6	5	8
7	7	1	2	3	4		6	5	8
7	5	1	2	3	4		6	7	8
7	6	1	2	3	4	5		7	8
7		1	2	3	4	5	6	7	8
5	5	1	2	3	4		6	7	8
5	6	1	2	3	4		5	7	8
5		1	2	3	4	6	5	7	8
6	6	1	2	3	4		5	7	8
6	5	1	2	3	4		6	7	8
6		1	2	3	4	5	6	7	8
		1	2	3	4	5	6	7	8

- いつも配列の前の方から処理をしている．**深さ優先探索**．

11.3.6　マージソート

アルゴリズム

- n = 1とする.
- ソートされている長さnの要素列2つずつとり，2つを併せたソートされた長さ$2n$要素列を作る.
- 配列全部が1つの要素列になったところで処理を終了する.
- ソートされた２つの要素列を併せて，１つのソートされた要素列を作るための計算量は要素数のオーダである．**両方の配列の先頭だけ見れば良い．**
- 例：次の２つを１つの配列にマージする.

1	4	6	8
2	3	5	7

⇒

- 先頭の１と２を比べる.

1	4	6	8
2	3	5	7

⇒

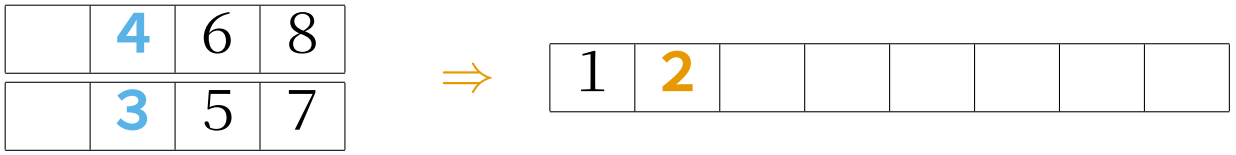
- １の方が小さいので，１を出力する配列に格納する．つぎに，４と２を比べる.

	4	6	8
2	3	5	7

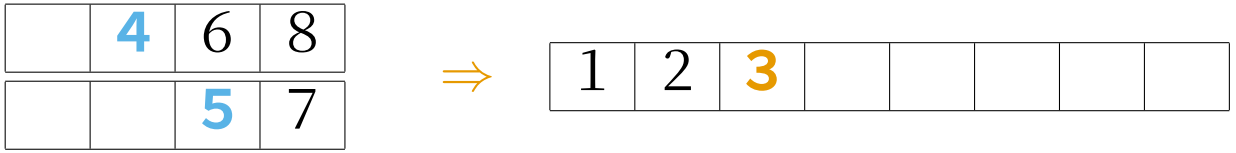
⇒

1							

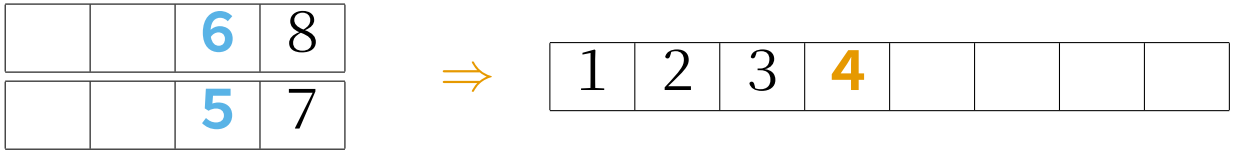
- ２の方が小さいので出力する配列に格納する．つぎに，４と３を比べる．



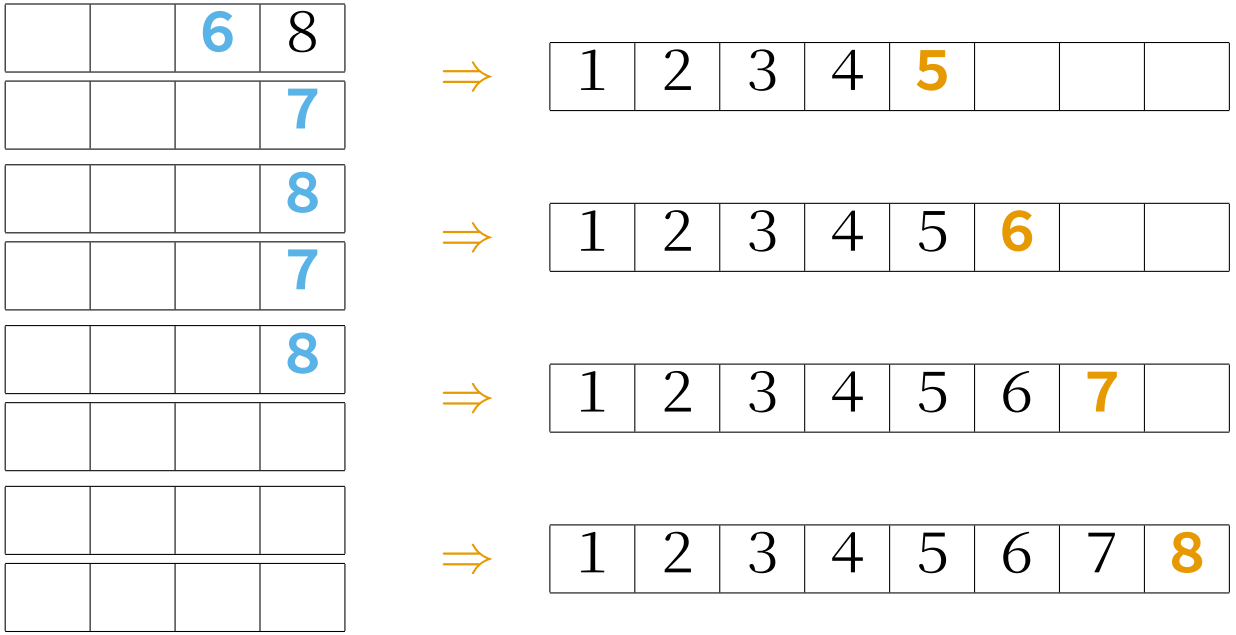
- ３の方が小さいので出力する配列に格納する．つぎに，４と５を比べる．



- ４の方が小さいので出力する配列に格納する．つぎに，６と５を比べる．



以下，説明は省略し結果だけ記す．



計算量

$N \log N$ のオーダーの計算量が必要となる (作業用配列が必要).

- 与えられた配列：

4	7	3	5	8	2	6	1

- 要素１つの集まり２つを組にする．(４組になる)

4	**7**	**3**	**5**	**8**	**2**	**6**	**1**

- その組においてソートされた配列のマージを実行する．

4	7	3	5	2	8	1	6

- できたもの２つずつを組にする．(２組になる)

4	**7**	**3**	**5**	**2**	**8**	**1**	**6**

- その組においてソートされた配列のマージを実行する．

3	4	5	7	1	2	6	8

- できたもの２つずつを組にする．(１組になる)

3	**4**	**5**	**7**	**1**	**2**	**6**	**8**

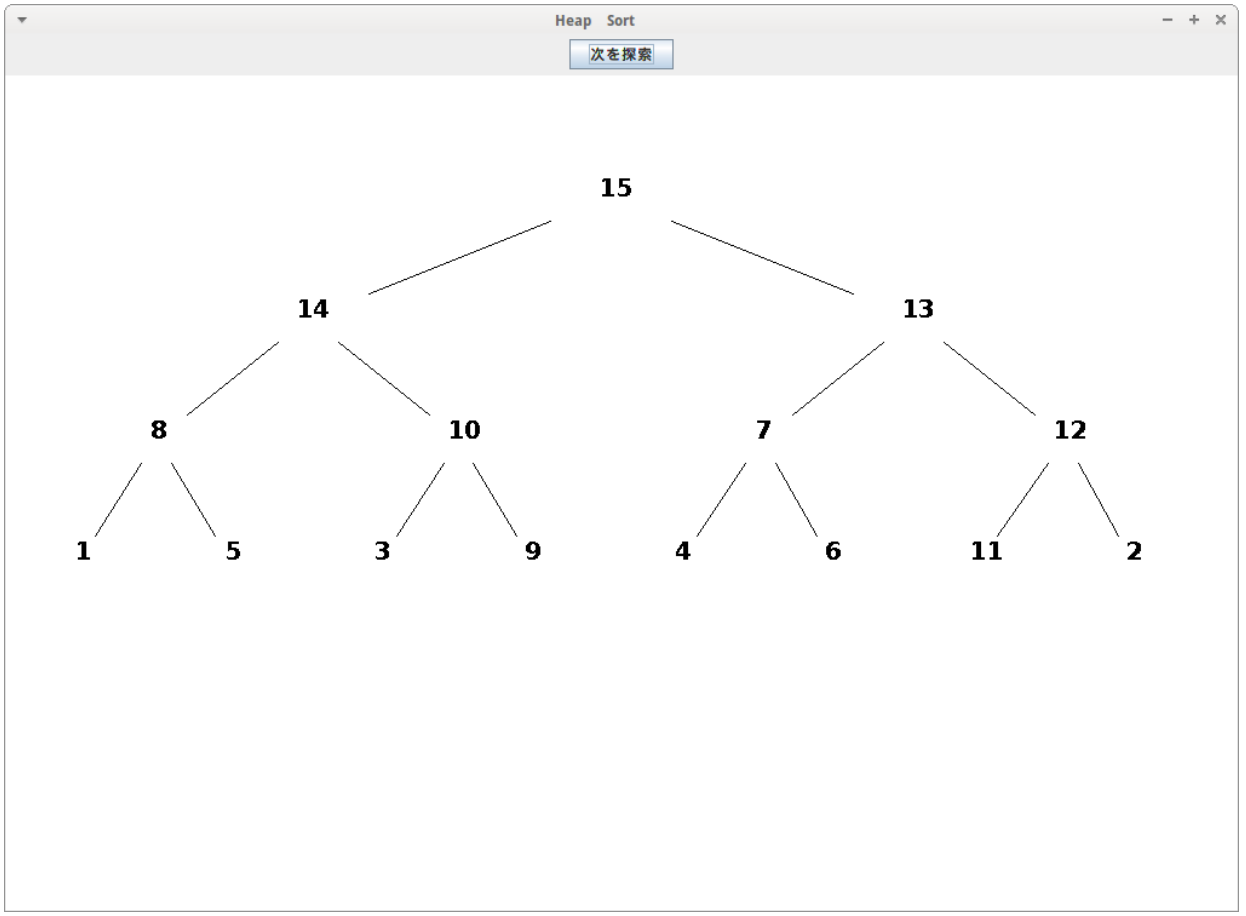
- その組においてソートされた配列のマージを実行する．

1	2	3	4	5	6	7	8

11.3.7　ヒープソート

アルゴリズム

- 1つずつ加えヒープを作成し，１つずつ取り出していく.
- ヒープ：常に親の節点の方が子の節点の方が値よりも小さい(大きい)木構造.
- $N \log N$ のオーダーの計算量が必要となる.
- デモプログラム (ShowHeapSort.java) を使って説明する。



11.4 まとめ

- 配列を理解することは極めて重要。

```
int    i[];
int[] j;
double d[][];
i = new int[5];
j = new int[10];
d = new int[8][8];
```

- **制御構造と配列を駆使すれば，どんなプログラムでも作成することが可能**。
- プログラムが複雑になってくるので，しっかりと理解する努力が必要。