データ構造とアルゴリズム実験レポート
課題：<課題名>

<学籍番号><クラス名><氏名>

締切日：<正式な締切日>
提出日：<実際の提出日>

1　必須課題

この課題では，教科書リスト 1-3 (p.4) の「線形探索による配列データの探索」に基づいた Java プログラム LinearSearch.java，教科書リスト 1-6 (p.8) の「2 分探索による配列データの探索」に基づいた Java プログラム BinarySearch.java を作成し，作成したプログラムのリストおよび実行結果を示した．

1.1　LinearSearch.java の作成

1.1.1　実装の方針

まず，`LinearSearch` クラスを定義し，線形探索による配列データの探索を実行するための機能を `linear_search` メソッドとして，線形探索が実行されているかを確認するための機能を `main` メソッドとして，クラス内にそれぞれ実装した．このように，機能単位でメソッドを分割することによって，コードのメンテナンス性と可読性が向上する．また，`main` メソッドは，配列の長さ `n` と探索の対象となる整数型データ `d` をコマンドライン引数で渡すことによって動作する．このため，配列の長さと探索データに応じた探索時間の変化を調べるための実験が容易にできるようになった．配列の初期化では，`linear_search` メソッドのデバッガビリティを考慮して，A[0] に 1，A[1] に 2，A[2] に 3，...，A[n-1] に `n` を格納し，値が 1 ずつ離れた要素が昇順にソートされているようにした．

1.1.2　実装コードおよびコードの説明

図 1 に，LinearSearch.java のソースコードを示す．1.1.1 節で述べた，`linear_search` メソッドと `main` メソッドは，それぞれ 8〜13 行目，16〜48 行目の部分に相当する．

`linear_search` メソッドは，探索データ `d` を線形探索アルゴリズムで探索し，配列 `A` 内に `d` が存在するならば `true`，存在しないならば `false` を boolean 型の値で返す．8 行目で，探索を実行するための配列 `A`，探索データ `d`，配列の長さ `n` を受け取るための仮引数を宣言し，9〜11 行目の `for` ループで，データ `d` を A[0] から順に探索し，配列 `A` に `d` が存在するかどうかを調べている．そして，データを発見すれば (10 行目の `if` 文の条件に合致すれば)，boolean 型の `true` を返し，データを発見できなければ (`for` ループが `i == n` まで回って終了するならば)，11 行目で boolean 型の `false` を返す．

`main` メソッドは，`linear_search` メソッドのテストケースとして機能する．配列の長さ `n` および探索デー

```java
////////////////////////////////////////////////////////////////////////////////
// A Java program to find data with Linear Search                    <Name> <Date>
////////////////////////////////////////////////////////////////////////////////
import java.util.Objects;

public class LinearSearch {
  // Find d from A
  static boolean linear_search(int[] A, int d, int n) {
    for (int i = 0; i < n; i++) {
      if (A[i] == d) return true;  // Data d exists
    }
    return false;  // Data d does not exist
  }


  public static void main(String[] args) {
    // Process arguments
    if (args.length < 2 || args.length >= 4) {
      System.err.println("Usage: java LinearSearch <numdata> <target data> <option (v)>");
      System.exit(1);
    }
    int n = Integer.parseInt(args[0]);
    int d = Integer.parseInt(args[1]);

    int[] A = new int[n];

    // Array initialized
    for (int i = 0; i < n; i++) A[i] = i + 1;

    // Find data
    boolean rslt = linear_search(A, d, n);

    // Show result
    if (rslt == true) System.out.println("Data found: " + d);
    else              System.out.println("Sorry. Data not found: " + d);

    // Show array contents if v option is set
    if (args.length == 3) {
      if (Objects.equals(args[2], "v")) {
        for (int i = 0; i < n; i++) System.out.print(A[i] + " ");
        System.out.println();
      } else {
        System.err.println("Invalid option: " + args[2]);
        System.exit(1);
      }
    }

  }
}
```

図 1　LinearSearch.java のソースコード

タ d は，コマンドライン引数によってセットされ，ソースコードの 22，23 行目がそれに相当する．セットされた配列の長さ n に基づいて，配列 A を 25 行目で宣言し，28 行目で配列を初期化している．その後，31 行目で linear_search メソッドを呼び出し，その際に配列 A，探索データ d，配列の長さ n を linear_search メソッドに渡すことによって線形探索を実行し，データが配列内に存在するかどうかを示す結果を boolean 型の変数 rslt に格納している．そして，rslt の値に基づいて，34 行目の if 文の条件に合致する (データが存在する) ならば，34 行目の println() メソッドを実行し，データが存在しないならば，35 行目の println() メソッドを実行する．なお，配列の中身を全て表示する機能も main メソッドに実装されており，38〜46 行目の部分がそれに相当する．第 3 コマンドライン引数を v とセットすると，40 行目の for ループによって配列の全要素を標準出力し，v 以外がセットされると，44 行目の exit(1) を実行して LinearSearch.java をエラー終了させる．第 3 コマンドライン引数をセットしない場合は，データが存在する，もしくは存在しないことを標準出力 (34，35 行目) し，38〜46 行目の部分をスキップする．このため，main メソッドに渡せるコマ

ンドライン引数の数は 2 個以上 4 個未満とし，そのスクリーニングを 18 行目で実行している．

1.1.3 実行結果

まず，LinearSearch.java を以下のコマンドでコンパイルする．

```
--------------------------------------
$ javac -g -verbose LinearSearch.java
--------------------------------------
```

ここで，-g はデバッグ情報の付加させるためのオプション，-verbose はコンパイルの詳細な出力を表示させるためのオプションである．コンパイルが成功すると LinearSearch.class が生成され，その後以下のコマンドを入力することによって，プログラムを実行できる．

```
--------------------------------------
$ java LinearSearch 10 5 v
Data found: 5
1 2 3 4 5 6 7 8 9 10
--------------------------------------
```

第 1 コマンドライン引数の 10，第 2 コマンドライン引数の 5 は，それぞれ配列の長さ n，探索データ d に相当する．すなわち，このプログラムは長さ 10 の配列 A の中から，5 を線形探索によって探索する処理を実行した結果，その値が配列内に存在することを標準出力している．1.1.2 節で述べたように，v が第 3 コマンドライン引数にセットされているため，一番最後の行にて配列の全要素が標準出力されている．これにより，データの 5 が配列内に存在することが視覚的に分かる．

次は，配列の長さを 10 のままにし，探索データを変化させてみる．ここでは，配列内に存在しないデータである 11 をコマンドライン引数にセットし，プログラムの挙動を観察する．

```
--------------------------------------
$ java LinearSearch 10 11 v
Sorry. Data not found: 11
1 2 3 4 5 6 7 8 9 10
--------------------------------------
```

今回は，長さ 10 の配列 A の中から 11 を線形探索したが，データが配列内に存在せず (図 1 の 9 行目の for ループが回りきった)，その結果を標準出力している．この例でも，v が第 3 コマンドライン引数にセットされているため，一番最後の行にて配列の全要素が標準出力されている．これにより，データの 11 が配列内に存在しないことが視覚的に分かり，「線形探索による配列データの探索」に基づいた Java プログラム LinearSearch.java が正しく実装されていることを確認できる．最後に，線形探索により配列内からデータを発見する・発見できなかった場合のプログラムの挙動を図 2 に示す．

1.1.4 考察

今回の実装では，探索対象のデータ型は整数型としたが，float 型，double 型といった整数型以外の型にも，線形探索アルゴリズムを適用することが可能である．それは，for ループを回して配列の中身をシーケンシャ

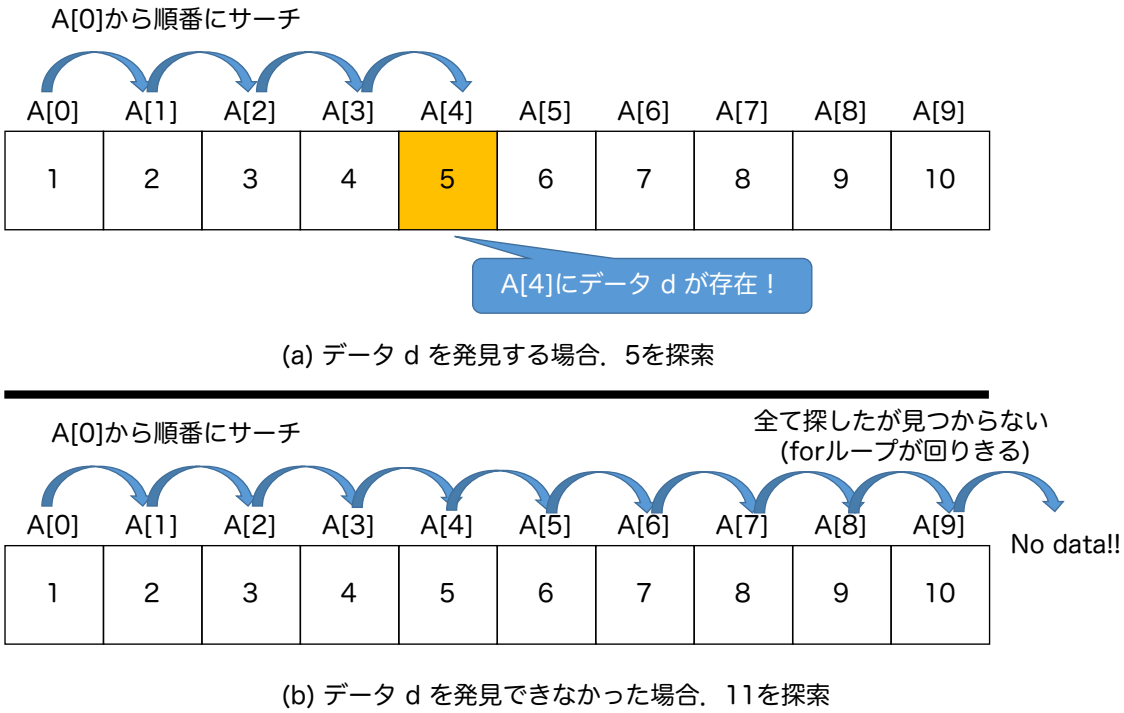

図 2　LinearSearch.java の挙動の図式化．(a) 線形探索により配列内からデータを発見する，(b) 発見できなかった場合のプログラムの挙動を示している．

ルに見ながら対象データを探すというアルゴリズムであることから，要素の型と対象データの型を揃えておけば，たとえ String 型であっても探索することが可能であるためである．また，A[i] の探索と A[i-1] の探索は互いに独立なため，OpenMP を用いた並列化による処理速度の改善も期待できる．そして，非常にシンプルなアルゴリズムであるため，アルゴリズム自体を論理回路としてハードウェア的に実装するのも容易であると考えられる．

1.2　BinarySearch.java の作成

1.2.1　実装の方針

実装の方針は，LinearSearch.java と同様である．`BinarySearch` クラスを定義し，2 分探索による配列データの探索を実行するための機能を `binary_search` メソッドとして，2 分探索が実行されているかを確認するための機能を `main` メソッドとして，クラス内にそれぞれ実装した．すなわち，`linear_search` メソッドを改変するだけで，BinarySearch.java は作成可能である．これは，LinearSearch.java の実装において，機能単位でメソッドを分割することによってコードのメンテナンス性を向上させたことによる恩恵を受けている．また，LinearSearch.java と同様に，`main` メソッドは，配列の長さ `n` と探索対象とする整数型データ `d` をコマンドライン引数で渡すことによって動作し，配列の初期化では，A[0] に 1，A[1] に 2，A[2] に 3，...，A[n-1] に `n` を格納した．これにより，BinarySearch.java でも LinearSearch.java と同様の実験および探索アルゴリズムのデバッグができるようになった．また，2 分探索はデータが昇順 (あるいは降順) にソートされた配列を用いる探索アルゴリズムであるため，配列の初期値をそのようにしている．ソートを必要とする理由は後述

```java
////////////////////////////////////////////////////////////////////////////////
// A Java program to find data with Binary Search                    <Name> <Date>
////////////////////////////////////////////////////////////////////////////////
import java.util.Objects;

public class BinarySearch {
  // Find d from A
  static boolean binary_search(int[] A, int d, int n) {
    int l = 0;
    int h = n - 1;
    while (l < h) {
      int m = (l + h) / 2;
      if (A[m] == d) return true;  // Data d exists

      if (d < A[m]) h = m - 1;
      else          l = m + 1;
    }
    if (A[l] == d) return true;    // Data d exists
    else           return false;   // Data d does not exist
  }


  public static void main(String[] args) {
    // Process arguments
    if (args.length < 2 || args.length >= 4) {
      System.err.println("Usage: java BinarySearch <numdata> <target data> <option (v)>");
      System.exit(1);
    }
    int n = Integer.parseInt(args[0]);
    int d = Integer.parseInt(args[1]);

    int[] A = new int[n];

    // Array initialized
    for (int i = 0; i < n; i++) A[i] = i + 1;

    // Find data
    boolean rslt = binary_search(A, d, n);

    // Show result
    if (rslt == true) System.out.println("Data found: " + d);
    else              System.out.println("Sorry. Data not found: " + d);

    // Show array contents if v option is set
    if (args.length == 3) {
      if (Objects.equals(args[2], "v")) {
        for (int i = 0; i < n; i++) System.out.print(A[i] + " ");
        System.out.println();
      } else {
        System.err.println("Invalid option: " + args[2]);
        System.exit(1);
      }
    }

  }
}
```

図 3　BinarySearch.java のソースコード

する．

1.2.2　実装コードおよびコードの説明

図 3 に，BinarySearch.java のソースコードを示す．1.2.1 節で述べた，`binary_search` メソッドと `main` メソッドは，それぞれ 8〜20 行目，23〜55 行の部分に相当する．

`binary_search` メソッドは，探索データ `d` を 2 分探索アルゴリズムで探索し，配列 `A` 内に `d` が存在するならば `true`，存在しないならば `false` を boolean 型の値で返す．8 行目で，配列 `A`，探索データ `d`，配列の長

さ n を受け取るための仮引数を宣言しており，この部分は linear_search メソッドと同様である．9 行目と 10 行目にて，配列の 0 番目のインデックスと n-1 番目のインデックスを，それぞれ変数 l と変数 h に格納している．この変数 l と変数 h は，探索範囲を示す境界として機能する (変数 l に格納されているインデックスから変数 h に格納されているインデックスまでを探索範囲とする)．12 行目にて，探索範囲の中間地点を示すインデックスを計算し，変数 m にその結果を格納している．そして，13 行目の if 文の条件に合致する (探索範囲の中間地点のデータが探索データと一致する) 場合，boolean 型の true を返す．合致しない場合は，15 行目の if 文の条件によって，探索範囲が変更される．すなわち，探索範囲の中間地点のデータ A[m] と探索データ d の大小関係を比較し，d が A[m] よりも小さければ，d は探索範囲の中間地点を示すインデックス m よりもインデックスが小さくなる範囲に存在するということになるため，探索範囲の右端として機能している変数 h に m-1 を代入して，探索範囲を l から m-1 に狭める (15 行目)．逆に，d が A[m] よりも大きければ，d は探索範囲の中間地点を示すインデックス m よりもインデックスが大きくなる範囲に存在するということになる．したがって，探索範囲の左端として機能している変数 l に m+1 を代入して，探索範囲を m+1 から h に狭める (16 行目)．このように，12∼16 行目の処理を繰り返すことによって，探索範囲を $n/2$，$n/4$，$n/8$，と狭めていき，狭める段階で探索データを発見 (13 行目)，もしくは狭め終えた時にデータを発見 (18 行目) する．狭め終えた時にデータを発見できなければ，配列内に探索データは存在しないということになるので，19 行目で boolean 型の false を返す．この大小関係の比較結果に基づいた探索範囲の縮小処理のために，配列 A のデータは探索処理の前にソート (この場合は昇順ソート) されていることが前提となる．

main メソッドについては，1.1.2 節で述べた main メソッドと同様の機能であるため，それについての説明は割愛する．

1.2.3 実行結果

1.1.3 節で述べたコンパイル方法と同様に，BinarySearch.java を javac コマンドでコンパイルし，BinarySearch.class を生成する．その後，以下のコマンドを入力することによって，BinarySearch.java のプログラムを実行する．

```
--------------------------------------
$ java BinarySearch 10 8 v
Data found: 8
1 2 3 4 5 6 7 8 9 10
--------------------------------------
```

1.1.3 節と同様に，第 1 コマンドライン引数の 10，第 2 コマンドライン引数の 8 は，第 3 コマンドライン引数の v は，それぞれ配列の長さ n，探索データ d，配列の全要素を標準出力させるためのオプションに相当する．すなわち，長さ 10 の配列 A の中から，8 を 2 分探索アルゴリズムで探索し，データが配列内に存在していることを標準出力している．また，v オプションによって，一番最後の行にて配列の全要素が標準出力されている．これにより，データの 8 が配列内に存在しており，それが 2 分探索アルゴリズムによって発見されたことが視覚的に分かる．

次は，配列の長さを 10 のままにし，探索データを変化させてみる．ここでは，配列内に存在しないデータである 11 をコマンドライン引数にセットし，プログラムの挙動を観察する．

```
-------------------------------------
$ java BinarySearch 10 11 v
Sorry. Data not found: 11
1 2 3 4 5 6 7 8 9 10
-------------------------------------
```

今回は，長さ 10 の配列 A の中から 11 を 2 分探索したが，データが配列内に存在せず (探索範囲を狭め終えた後の if 文の条件 (18 行目) に合致しなかった)，その結果を標準出力している．この例でも，v が第 3 コマンドライン引数にセットされているため，一番最後の行にて配列の全要素が標準出力されている．これにより，データの 11 が配列内に存在しないことが視覚的に分かり，「2 分探索による配列データの探索」に基づいた Java プログラム BinarySearch.java が正しく実装されていることを確認できる．最後に，2 分探索により配列内からデータを発見する・発見できなかった場合のプログラムの挙動を図 4 に示す．

1.2.4 考察

2 分探索は線形探索と異なり，探索前に配列内のデータが昇順 (もしくは降順) にソートされている必要があるため，String 型に適用するためには工夫が必要である．その一例としては，String 型のデータを，文字コードを利用して一旦整数型に変換し，探索後に元に戻すというのが考えられる．また，線形探索では OpenMP ディレクティブを一行挿入 (#pragma omp parallel for) するだけで，容易に並列化を施すことが可能だが，2 分探索の場合は，配列をスレッド毎に分割し，その際にスレッド毎の変数 l と h を計算する必要があるため，コードを多少変更する必要がある．そのため，並列化の実装コストは線形探索と比べて，やや高いと考えられる．アルゴリズムのハードウェア化についても，1. m を計算，2. A[m] が d と一致しているかチェック，3. 一致していないなら，A[m] と d で大小比較を行って探索範囲を変更するという，1~3 の状態を制御するためのロジック (ステートマシン) を実装する必要があるため，線形探索と比べて実装コストが高く，かつハードウェアリソースを多く使用することが考えられる．このため，時間計算量については 2 分探索は線形探索より優れているが，その他の観点を考慮に入れた場合は，必ずしも 2 分探索は最適解であると言い切れない．線形探索と 2 分探索の時間計算量は発展課題にて述べる．

2 発展課題

この課題では，必須課題で作成した LinearSearch.java，BinarySearch.java について，それぞれの時間計算量を議論した．

一般的に，時間計算量はループにおける繰り返し回数によって左右されると言っても良い．LinearSearch.java では，探索データが配列内に存在しない場合，配列データの個数 n に対して，ループの繰り返し回数は n 回となるため，計算量は $O(n)$ である．データが配列内に存在する場合のループの繰り返し回数は，最小で 1 回，最大で n 回となり，平均は約 $n/2$ 回である．これも n がパラメータであることに変わりはないので，計算量は $O(n)$ となる．すなわち，n を大きくするにつれて，探索時間は線形に増加する．

一方，BinarySearch.java では，図 3 における 11 行目の while ループを 1 回実行する毎に，探索範囲が $n/2$，$n/4$，$n/8$，... のように縮小していく．すなわち，ループの繰り返し回数は約 $\log_2 n$ 回となるので，計算量は $O(\log n)$ となる．ただし，前述したように 2 分探索では探索前にソート処理が実行されていなければならないのに対し，線形探索では配列の要素を順にサーチするアルゴリズムであるため，特別な前処理を必要

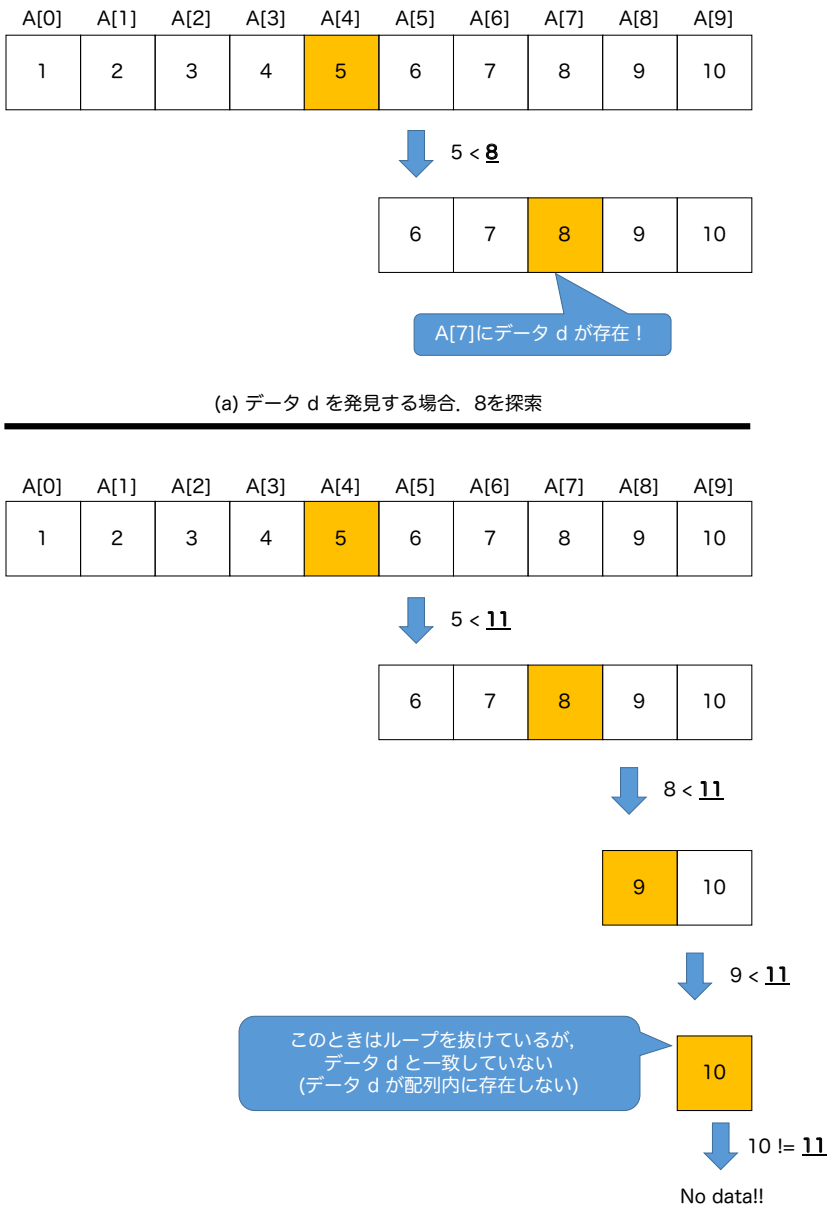

図 4　BinarySearch.java の挙動の図式化．(a) 2 分探索により配列内からデータを発見する，(b) 発見できなかった場合のプログラムの挙動を示している．

としない．つまり，ソート処理を含めた場合，ソート処理のアルゴリズムが最適化されていないと，探索自体の時間計算量は 2 分探索の方が線形探索と比べて優れているのにもかかわらず，線形探索の方が結果として高速であるというケースが起こりうる．このため，アルゴリズムの計算量の良し悪しだけではなく，アルゴリズムの前提条件や特性，そしてアルゴリズムを実行するハードウェアの性能をよく吟味して，最適解を選択することが重要であると主張し，この議論をまとめる．