第３章

探　索


●線形探索
●番兵法
●２分探索
●計算量
●Arrays.binarySearch による探索
●コンパレータの定義
●ハッシュ法
●チェイン法
●オープンアドレス法

3–1 探索アルゴリズム

本章では、データの集合から、目的とする値をもった要素を探し出す探索アルゴリズムを学習します。

探索とキー

住所録からの探索（*searching*）を考えましょう。ひとことで《探索》といっても、以下に示すように、さまざまな探し方があります。

- 国籍が日本である人を探す。
- 年齢が 21 歳以上 27 歳未満の人を探す。
- ある語句と最も発音が似ている名前の人を探す。

どの探索も、**何らかの項目**に着目する点が共通です。着目する項目のことを**キー**（*key*）と呼びます。国籍での探索を行う場合は国籍がキーであり、年齢で探索する場合は年齢がキーです。

多くの場合、**キーはデータの“一部”です**。もっとも、データが単なる整数値であれば、データの値がそのままキー値となります。

さて、上記の探索は、キー値に関して、次のような指定を行ったものでした。

- キー値と**一致**することを指定する。
- キー値の**区間**で指定する。
- キー値の**近接**として指定する。

もちろん、これらの条件を単独に指定するのではなく、論理積や論理和を用いて複合的に指定することもあります。

とはいえ、ある値と一致するキー値をもつデータを探すのが、単純であるとともに一般的です。他の条件による探索は、その応用と考えられます。

配列からの探索

これまでに、数多くの探索手法が考案されています。

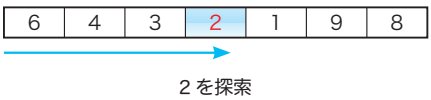
Fig.3-1 に示すのが、探索の例です。

これらの中には、データの格納先のデータ構造に依存するものがあります。図**b**の線形リストからの探索は第 9 章で学習し、図**c**の 2 分探索木からの探索は第 10 章で学習します。

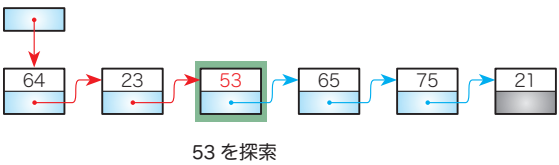
また、ここには示していませんが、文字列の中の一部として存在する文字列の探索については第 8 章で学習します。

探索とは、ある条件を満たすデータを探し出すこと。

a 配列からの探索



b 線形リストからの探索



c 2分探索木からの探索

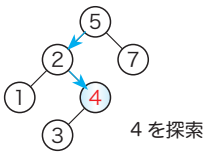


Fig.3-1 探索の例

本章で学習するのは、図aに示す《配列からの探索》です。具体的には、以下に示すアルゴリズムです。

- **線形探索**：ランダムに並んだデータの集まりからの探索を行う。
- **2分探索**：一定の規則で並んだデータの集まりからの高速な探索を行う。
- **ハッシュ法**：追加・削除が高速に行えるデータの集まりからの高速な探索を行う。
  - **チェイン法**：同一ハッシュ値のデータを線形リストでつなぐ手法。
  - **オープンアドレス法**：衝突時に再ハッシュを行う手法。

データの集合から『探索さえ行えればよい』のであれば、探索に要する計算時間が短いアルゴリズムを選択することになります。

しかし、データの集合に対して、探索だけでなく、データの追加や削除などを頻繁に行う場合は、探索以外の操作に要するコストなども含めて総合的に評価してアルゴリズムを選択する必要があります。たとえば、データの追加を頻繁に行うのであれば、たとえ探索が速くても、追加のコストが高くつくようなアルゴリズムは避けるべきです。

ある目的に対して複数のアルゴリズムが存在する場合は、用途や目的・実行速度・対象となるデータ構造などを考慮してアルゴリズムを選択します。

3–2 線形探索

配列からの探索として最も基本的なアルゴリズムが、本節で学習する線形探索です。このアルゴリズムは、後の章でも利用しますので、しっかりと学習しましょう。

線形探索

要素が直線状に並んだ配列からの探索は、目的とするキー値をもつ要素に出会うまで先頭から順に要素を走査する（なぞる）ことで実現できます。

これが、**線形探索**（*linear search*）あるいは**逐次探索**（*sequential search*）と呼ばれるアルゴリズムです。

具体的な手順を、次に示すデータの並びを例に考えていきましょう。

0	1	2	3	4	5	6
6	4	3	2	1	3	8

この配列から値が 2 である要素を線形探索する様子を示したのが **Fig.3-2** です。

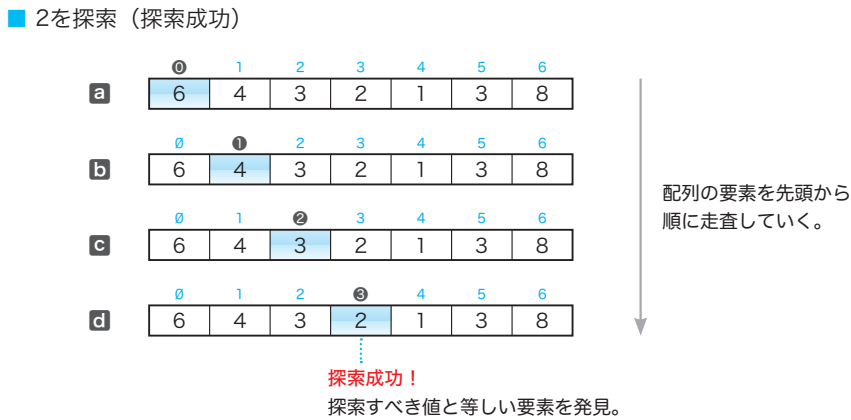


Fig.3-2　線形探索の一例（2を探索：探索成功）

図中、●内に示している値は、配列を走査する過程で着目する要素のインデックスです。探索は次のように行われます。

- **a**　1 番目の要素 6 に着目します。目的とする値ではありません。
- **b**　2 番目の要素 4 に着目します。目的とする値ではありません。
- **c**　3 番目の要素 3 に着目します。目的とする値ではありません。
- **d**　4 番目の要素 2 に着目します。目的とする値ですから、**探索に成功です**。

探索に成功する例を考えました。もっとも、キー値と同じ値の要素が配列中に存在するとは限りません。たとえば、同じ配列から 5 を探索すると失敗します。

その探索の様子を示したのが **Fig.3-3** です。図 a から図 h まで、配列の要素を先頭から順に走査していきます。キー値と同じ値の要素に出会うことは、最後までありません。

■ 5を探索（探索失敗）

Fig.3-3　線形探索の一例（5を探索：探索失敗）

成功例と失敗例とから、配列の走査の終了条件が二つあることが分かります。次に示す条件の**いずれか一方でも成立すれば、走査は終了です。**

> ① 探索すべき値が見つからず末端を通り越した（あるいは通り越しそうになった）。
> ② 探索すべき値と等しい要素を見つけた。

もちろん、条件①の成立時は**探索失敗**で、条件②の成立時は**探索成功**です。要素数が n であれば、これらの条件を判断する回数は、いずれも平均 n / 2 回です。

▶ 配列中に目的とする値が存在しないときは、①と②の判定は、それぞれ n + 1 回と n 回行われます。

＊

ここまでの考え方をもとに線形探索を実現したプログラムを **List 3-1**（次ページ）に示します。

List 3-1 Chap03/SeqSearch.java

```
// 線形探索

import java.util.Scanner;

class SeqSearch {

   //--- 配列aの先頭n個の要素からkeyと一致する要素を線形探索 ---//
   static int seqSearch(int[] a, int n, int key) {
      int i = 0;

      while (true) {
         if (i == n)
            return -1;      // 探索失敗（-1を返却）          ①
         if (a[i] == key)
            return i;       // 探索成功（インデックスを返却）  ②
         i++;
      }
   }

   public static void main(String[] args) {
      Scanner stdIn = new Scanner(System.in);

      System.out.print("要素数：");
      int num = stdIn.nextInt();
      int[] x = new int[num];     // 要素数numの配列

      for (int i = 0; i < num; i++) {
         System.out.print("x[" + i + "]：");
         x[i] = stdIn.nextInt();
      }

      System.out.print("探す値：");     // キー値の読込み
      int ky = stdIn.nextInt();

      int idx = seqSearch(x, num, ky); // 配列xから値がkyの要素を探索

      if (idx == -1)
         System.out.println("その値の要素は存在しません。");
      else
         System.out.println("その値はx[" + idx + "]にあります。");
   }
}
```

実行例

```
要素数：7↵
x[0]：22↵
x[1]：8↵
x[2]：55↵
x[3]：32↵
x[4]：120↵
x[5]：55↵
x[6]：70↵
探す値：55↵
その値はx[2]にあります。
```

メソッド *seqSearch* は、配列 a の先頭 *n* 個の要素を対象に、値が *key* の要素を線形探索します。返却するのは、見つけた要素のインデックスです。

なお、値が *key* の要素が複数個存在する場合に返却するのは、走査の過程で最初に見つけた要素（すなわち最も先頭側の要素）のインデックスです。また、値が *key* の要素が存在しない場合には -1 を返却します。

▶ 探索失敗時に返却する -1 は、配列のインデックスとしてはあり得ない値です。したがって、メソッドを呼び出す側では、探索に成功したかどうかを確実に判断できます。

なお、実行例に示しているのは、55 を探索する例です。この値は、*x*[2] と *x*[5] の両方に存在しますが、先頭側のものを見つけて 2 を返却します。

配列の走査時に着目する要素のインデックスを表すのが変数 *i* です（前ページの図で●内に示した値に相当します）。宣言時に 0 で初期化しておき、要素を一つなぞるたびに、**while** 文が制御するループ本体の末尾でインクリメントしていきます。

while 文を抜け出るのは、p.77 に示した終了条件①と②のいずれかが成立したときであり、それぞれ以下のように対応しています。

1. *i* == *n* が成立した（終了条件①）。
2. a[*i*] == *key* が成立した（終了条件②）。

▶ なお、**while** 文による繰返しを続けるかどうかの判定では、黒網部の制御式 **true** が評価されます。そのため、繰返しのたびに行われる条件判定は、厳密には 2 回ではなく 3 回です。

配列の走査を **while** 文ではなく **for** 文で実現すると、プログラムは短く簡潔になります。**List 3-2** に示すのが、そのプログラムです。

List 3-2 Chap03/SeqSearchFor.java

```
//--- 配列aの先頭n個の要素からkeyと一致する要素を線形探索 ---//
static int seqSearch(int[] a, int n, int key) {
    for (int i = 0; i < n; i++)
        if (a[i] == key)
            return i;          // 探索成功（インデックスを返却）
    return -1;                 // 探索失敗（-1を返却）
}
```

▶ 本書では、本プログラムのように、メソッドのみを示すことがあります。メソッドだけではプログラムは実行できません。メソッドを呼び出す **main** メソッドを含むプログラムは、自分で作りましょう。本プログラムの場合は、**List 3-1** を参考にすれば簡単に作れます。

なお、**main** メソッドなどを含む完全なプログラムは、ホームページからダウンロードできるファイルに含まれています（p.iv）。

先頭から順に要素を走査する線形探索は、ランダムな並びの配列から探索を行うための唯一の方法です。

Column 3-1 無限ループの実現

List 3-1 の **while** 文は、《無限ループ》の形をしています。“無限”といっても、**break** 文を使えばループから抜け出せますし、**return** 文を使えばループを含んだメソッドから抜け出せます。

さて、その無限ループは、以下のように実現できます（**for** 文では、繰返しの継続を判定するための制御式を省略すると、**true** が指定されたものとみなされます）。

```
while (true) {
    // 中略
}
```

```
for ( ; ; ) {
    // 中略
}
```

```
do {
    // 中略
} while (true);
```

私たちは通常、ソースプログラムを上から下へと眺めていきます。そのため、**while** 文と **for** 文は、最初の 1 行を読むだけで無限ループと分かります。

最後まで読まないと無限ループと分からない **do** 文による実現は、お勧めできません。

番兵法

線形探索では、繰返しのたびに二つの終了条件①と②（p.77）の両方をチェックします。単純な判定とはいえ、“塵も積もれば山となる”のですから、そのコストは決して無視できません。

このコストを半分に抑えるのが、ここで学習する**番兵法**（*sentinel method*）です。**Fig.3-2** と **Fig.3-3** に示した探索を、番兵法によって行う様子を示したのが **Fig.3-4** です。この図を見ながら理解していきましょう。

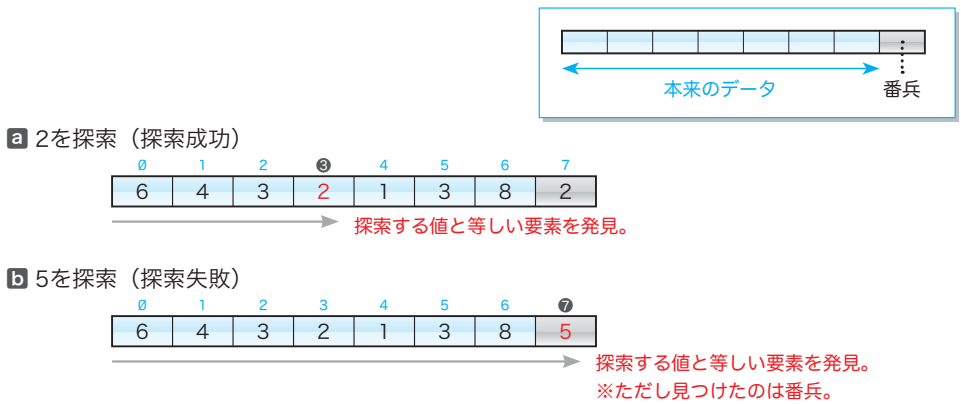


Fig.3-4　番兵法を用いた線形探索

この図において、配列中の a[0] ～ a[6] の要素は本来のデータです。

探索の前準備として、**探索するキー値と同じ値を末尾要素 a[7] に格納します**。このときに格納するデータが**番兵**（*sentinel*）です。

- 図a：2 を探索する準備として、番兵として a[7] に 2 を格納する。
- 図b：5 を探索する準備として、番兵として a[7] に 5 を格納する。

そうすると、図bのように、目的とする値が本来のデータ内に存在しなくても、番兵である a[7] まで走査した段階で終了条件②が成立します。そのため、条件①の判定が**不要**となります。**番兵は、繰返しの終了判定を削減する役割をもちます。**

*

番兵法を導入して **List 3-1** を書きかえたプログラムが **List 3-3** です。

黒網部では、キーボードから読み込んだ要素数に 1 を加えた要素数の配列を生成します（たとえば、要素数として 7 が入力されると、要素数 8 の配列を生成します）。

▶　本来のデータに加えて、その後ろに番兵を格納するためです。

*

メソッド *seqSearchSen* の中を理解していきましょう。

List 3-3 Chap03/SeqSearchSen.java

```
// 線形探索（番兵法）

import java.util.Scanner;

class SeqSearchSen {

    //--- 配列aの先頭n個の要素からkeyと一致する要素を線形探索（番兵法）---//
    static int seqSearchSen(int[] a, int n, int key) {
        int i = 0;

        a[n] = key;                  // 番兵を追加 ←1

        while (true) {
            if (a[i] == key)         // 探索成功
                break;
            i++;
        }
        return i == n ? -1 : i;      ←2 ←3
    }

    public static void main(String[] args) {
        Scanner stdIn = new Scanner(System.in);

        System.out.print("要素数：");
        int num = stdIn.nextInt();
        int[] x = new int[num + 1];          // 要素数num + 1の配列

        for (int i = 0; i < num; i++) {
            System.out.print("x[" + i + "]：");
            x[i] = stdIn.nextInt();
        }

        System.out.print("探す値：");          // キー値の読込み
        int ky = stdIn.nextInt();

        int idx = seqSearchSen(x, num, ky);  // 配列xから値がkyの要素を探索

        if (idx == -1)
            System.out.println("その値の要素は存在しません。");
        else
            System.out.println("その値はx[" + idx + "]にあります。");
    }
}
```

実行例

```
要素数：7
x[0]：22
x[1]：8
x[2]：55
x[3]：32
x[4]：120
x[5]：55
x[6]：70
探す値：120
その値はx[4]にあります。
```

1 探索する値 *key* を番兵として a[*n*] に代入します。

2 配列の要素を走査します。**List 3-1** の **while** 文には **if** 文が二つありました。

```
if (i == n)          // 終了条件1
if (a[i] == key)     // 終了条件2
```

本プログラムでは、前者が不要となったため、**if** 文は一つだけです。そのため、繰返し終了のための**判定回数は実質的に半分となります。**

3 **while** 文による繰返しが終了すると、**見つけたのが、配列内の本来のデータなのか、それとも番兵なのかの判定が必要です。**変数 *i* の値が *n* になっていれば、見つけたのは番兵ですから、探索に失敗したことを表す **-1** を返します。

3–3 2分探索

本節では2分探索法を学習します。このアルゴリズムの適用は、データがキー値でソートずみの場合に限定されるのですが、線形探索よりも高速に探索を行えます。

2分探索

2分探索（*binary search*）は、要素がキーの昇順または降順にソート（整列）されている配列から効率よく探索を行うアルゴリズムです。

▶ ソートアルゴリズムは第6章で学習します。

下図に示す、昇順にソートされた（小さいほうから順に並んだ）データの並びからの39の探索を考えましょう。まず、配列の中央に位置する要素a[5]すなわち31に着目します。

0	1	2	3	4	❺	6	7	8	9	10
5	7	15	28	29	31	39	58	68	70	95

目的とする39は、この要素よりも末尾側に存在するはずです。そこで、探索の対象を末尾側の5個すなわちa[6]～a[10]に絞り込みます。

引き続き、更新された対象範囲の中央要素であるa[8]すなわち68に着目します。

0	1	2	3	4	5	6	7	❽	9	10
5	7	15	28	29	31	39	58	68	70	95

目的とする値は、この要素より先頭側に存在するはずですから、探索の対象を先頭側の2個すなわちa[6]～a[7]に絞り込みます。

二つの要素の中央要素は、先頭側の39と末尾側の58のどちらでも構いませんが、ここでは、先頭側の値である39に着目します（整数どうしの除算では小数点以下が切り捨てられて、二つのインデックス6と7の中央値（6 + 7）/ 2が6となるからです）。

0	1	2	3	4	5	❻	7	8	9	10
5	7	15	28	29	31	39	58	68	70	95

着目した39は、目的とするキー値と一致しますので、**探索成功**です。

*

*n*個の要素が昇順に並んでいる配列aから*key*を探索するとして、このアルゴリズムを一般的に表現しましょう。

探索範囲の先頭、末尾、中央のインデックスを*pl*、*pr*、*pc*とします。探索開始時の*pl*は0、*pr*は*n* - 1、*pc*は（*n* - 1）/ 2です。これが**Fig.3-5** a の状態です。

探索の対象範囲は青い□内の要素で、探索の対象から外れた範囲は黒い□内の要素です。探索範囲は、比較のたびに（ほぼ）半分に絞り込まれていきます。また、一つずつ着目要素をずらす線形探索とは異なり、●で示す着目要素は**一気に移動します**。

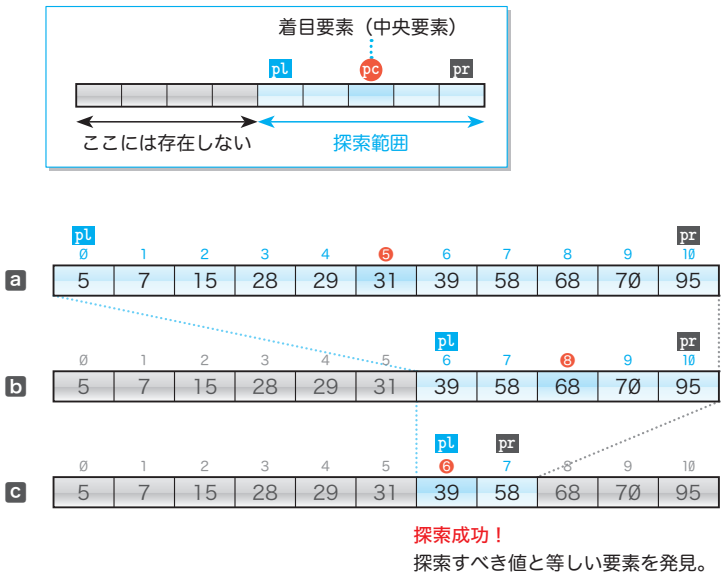


Fig.3-5　2分探索の一例（39を探索：探索成功）

図cのように、a[*pc*]と*key*を比較して等しければ**探索成功**ですが、そうでない場合は、次のように探索範囲を絞り込みます。

▪ a[*pc*] < *key* のとき

a[*pl*]～a[*pc*]は、*key*よりも小さいことが明らかであって探索対象から外せます。
探索範囲は、中央要素a[*pc*]より後方のa[*pc* + 1] ～ a[*pr*]に絞り込めます。
そこで、*pl*の値を*pc* + 1に更新します（図a⇨図b）。

▪ a[*pc*] > *key* のとき

a[*pc*]～a[*pr*]は、*key*よりも大きいことが明らかであって探索対象から外せます。
探索範囲は、中央要素a[*pc*]より前方のa[*pl*]～a[*pc* - 1]に絞り込めます。
そこで、*pr*の値を*pc* - 1に更新します（図b⇨図c）。

アルゴリズムの終了条件は、以下の条件①と②のいずれか一方が成立することです。

① a[*pc*]と*key*が一致した。
② 探索範囲がなくなった。

図に示したのは、条件①が成立して、探索に成功する例でした。
条件②が成立して、探索に失敗する具体例も考えてみましょう。同じ配列から6を探索する様子を**Fig.3-6**（次ページ）に示します。

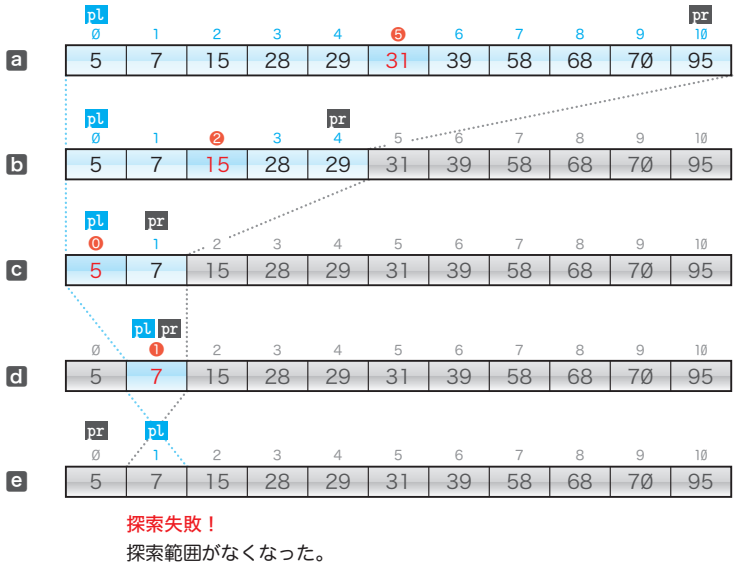


Fig.3-6　2分探索の失敗例（6を探索）

a　探索すべき範囲は配列全体すなわち a[0] ～ a[10] であり、中央要素 a[5] の値は 31 です。これは *key* の値 6 より大きいため、探索する範囲を先頭から a[5] の直前の要素まで、すなわち a[0] ～ a[4] に絞り込みます。

b　縮小された範囲の中央要素 a[2] の値は 15 です。これは *key* の値 6 より大きいため、探索すべき範囲を a[2] の直前の要素まで、すなわち a[0] ～ a[1] に絞り込みます。

c　縮小された範囲の中央要素 a[0] の値は 5 です。これは *key* の値 6 より小さいため、*pl* を *pc* + 1 すなわち 1 に更新します。そうすると、*pl* と *pr* は 1 になります。

d　縮小された範囲の中央要素 a[1] の値は 7 です。これは *key* の値 6 より大きいため、*pr* を *pc* - 1 すなわち 0 に更新します。そうすると、*pl* が *pr* よりも大きくなって**探索範囲がなくなります。終了条件②が成立しますので、探索失敗です。**

2分探索を行うプログラムを **List 3-4** に示します。

▶　2分探索アルゴリズムでは、探索の対象となる配列がソートされている必要があります。本プログラムの網かけ部では、各要素の値を読み込む際に一つ前に読み込んだ要素よりも小さい値が入力された場合は、再入力させるようにしています。

繰返しのたびに探索範囲が半分になりますから、必要となる比較回数の平均は $\log n$ です。なお、探索に失敗した場合は $\lceil \log(n+1) \rceil$ 回、探索に成功した場合は約 $\log n - 1$ 回となります。

List 3-4 Chap03/BinSearch.java

```
// ２分探索

import java.util.Scanner;

class BinSearch {

   //--- 配列aの先頭n個の要素からkeyと一致する要素を２分探索 ---//
   static int binSearch(int[] a, int n, int key) {
      int pl = 0;          // 探索範囲先頭のインデックス
      int pr = n - 1;      //     〃     末尾のインデックス

      do {
         int pc = (pl + pr) / 2; // 中央要素のインデックス
         if (a[pc] == key)
            return pc;            // 探索成功
         else if (a[pc] < key)
            pl = pc + 1;          // 探索範囲を後半に絞り込む
         else
            pr = pc - 1;          // 探索範囲を前半に絞り込む
      } while (pl <= pr);

      return -1;                  // 探索失敗
   }

   public static void main(String[] args) {
      Scanner stdIn = new Scanner(System.in);

      System.out.print("要素数：");
      int num = stdIn.nextInt();
      int[] x = new int[num];     // 要素数numの配列

      System.out.println("昇順に入力してください。");

      System.out.print("x[0]：");       // 先頭要素の読込み
      x[0] = stdIn.nextInt();

      for (int i = 1; i < num; i++) {
         do {
            System.out.print("x[" + i + "]：");
            x[i] = stdIn.nextInt();
         } while (x[i] < x[i - 1]); // 一つ前の要素より小さければ再入力
      }

      System.out.print("探す値：");     // キー値の読込み
      int ky = stdIn.nextInt();

      int idx = binSearch(x, num, ky); // 配列xから値がkyの要素を探索

      if (idx == -1)
         System.out.println("その値の要素は存在しません。");
      else
         System.out.println("その値はx[" + idx + "]にあります。");
   }
}
```

実行例

```
要素数：7
昇順に入力してください。
x[0]：15
x[1]：27
x[2]：39
x[3]：77
x[4]：92
x[5]：108
x[6]：121
探す値：39
その値はx[2]にあります。
```

▶ ⌈x⌉ は、x の**天井関数**（*ceiling*）であり、x 以上の最小の整数を表します。たとえば ⌈3.5⌉ は 4 です。

計算量

プログラムの実行速度や実行に要する時間は、それを動作させるハードウェアやコンパイラなどの条件に依存します。アルゴリズムの性能を客観的に評価するための尺度として用いられるのが、**計算量**（*complexity*）です。

計算量は、以下の二つに大別されます。

- **時間計算量**（*time complexity*）
  実行に要する時間を評価したもの。
- **領域計算量**（*space complexity*）
  どのくらいの記憶域やファイル域が必要であるかを評価したもの。

前章で学習した《素数》のプログラム例（第１版・第２版・第３版）は、アルゴリズム選択の際に、二つの計算量のバランスを考える必要性を示しています。

ここでは、線形探索と２分探索の時間計算量を考察します。

線形探索の時間計算量

以下に示す線形探索のメソッドをもとに、時間計算量を考えていきましょう。

```
   static int seqSearch(int[] a, int n, int key) {
1     int i = 0;

2     while (i < n) {
3        if (a[i] == key)
4           return i;          // 探索成功
5        i++;
      }
6     return -1;               // 探索失敗
   }
```

▶ このプログラムは、**List 3-1** のメソッド *seqSearch* を改変したものです。

1〜6の各ステップが何回実行されるかをまとめたのが **Table 3-1** です。

＊

変数 i に 0 を代入する1が行われるのは 1 回限りであり、データ数 n とは無関係です。このような計算量を O(1) と表します。

もちろん、メソッドから値を返すための4と6なども同様に O(1) です。

配列の末尾に到達したかを判断する2や、着目要素と探索すべき値との等価性を判定するための3が行われる平均回数は n / 2 です。このように、n に比例した回数だけ実行される計算量は O(n) と表します。

計算量の表記で利用している O は order の頭文字です。O(n) は、**“n のオーダー”** あるいは **“オーダー n”** と呼ばれます。

Table 3-1　線形探索における各ステップの実行回数と計算量

ステップ	実行回数	計算量
1	1	O(1)
2	n / 2	O(n)
3	n / 2	O(n)
4	1	O(1)
5	n / 2	O(n)
6	1	O(1)

さて、n をどんどん大きくしていくと、O(n) に要する計算時間は、n に比例して長くなります。一方、O(1) に要する計算時間が変化することはありません。

このことからも推測できるように、一般に、O(f(n)) と O(g(n)) の操作を連続した場合の計算量は、次のようになります。

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

▶ max(a, b) は a と b の大きいほうを表します。

すなわち、二つの計算から構成されるアルゴリズムの計算量は、**より大きいほうの計算量に支配されます**。二つの計算でなく、三つ以上の計算から構成されるアルゴリズムも同様です。全体の計算量は、**最も大きい計算量に支配されます**。

このことから、線形探索のアルゴリズムの計算量を求めると、以下に示すように O(n) となります。

$$\begin{aligned} &O(1) + O(n) + O(n) + O(1) + O(n) + O(1) \\ &\quad = O(\max(1, n, n, 1, n, 1)) \\ &\quad = O(n) \end{aligned}$$

演習 3–1

List 3-3（p.81）のメソッド *seqSearchSen* を、**while** 文ではなく **for** 文を用いて書きかえたプログラムを作成せよ。

演習 3–2

右のように、線形探索の走査過程を詳細に表示するプログラムを作成せよ。

各行の左端に着目要素のインデックスを表示するとともに、着目中の要素の上に、アステリスク記号 '*' を表示すること。

```
   |  0  1  2  3  4  5  6
---+-------------------------
   |  *
 0 |  6  4  3  2  1  9  8
   |
   |     *
 1 |  6  4  3  2  1  9  8
   |
   |        *
 2 |  6  4  3  2  1  9  8

3はx[2]に存在します。
```

2分探索の時間計算量

2分探索法では、着目する要素の範囲が半分ずつに減っていきます。プログラム中の各ステップの実行回数と計算量は、**Table 3-2** のようになります。

```
   //--- 2分探索 ---//
   static int binSearch(int[] a, int n, int key) {
1    int pl = 0;             // 探索範囲先頭のインデックス
2    int pr = n - 1;         //     〃     末尾のインデックス

     do {
3      int pc = (pl + pr) / 2; // 中央要素のインデックス
4      if (a[pc] == key)
5        return pc;            // 探索成功
6      else if (a[pc] < key)
7        pl = pc + 1;          // 探索範囲を後半に絞り込む
       else
8        pr = pc - 1;          // 探索範囲を前半に絞り込む
9    } while (pl <= pr);

10   return -1;                // 探索失敗
   }
```

Table 3-2　2分探索における各ステップの実行回数と計算量

ステップ	実行回数	計算量
1	1	O(1)
2	1	O(1)
3	log n	O(log n)
4	log n	O(log n)
5	1	O(1)
6	log n	O(log n)
7	log n	O(log n)
8	log n	O(log n)
9	log n	O(log n)
10	1	O(1)

2分探索アルゴリズムの計算量を求めると、以下のように O(log n) が得られます。

$$O(1) + O(1) + O(\log n) + O(\log n) + O(1) + O(\log n) + \cdots + O(1)$$
$$= O(\log n)$$

さて、O(n) や O(log n) が O(1) より大きいのは当然です。これらを含めて、計算量の大小関係を示したのが **Fig.3-7** です。

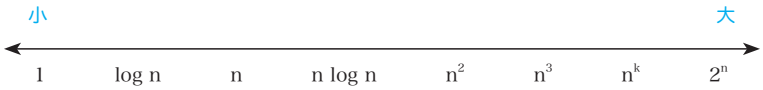


Fig.3-7　計算量と増加率

演習 3–3

要素数が *n* である配列 a から *key* と一致する全要素のインデックスを、配列 *idx* の先頭から順に格納し、一致した要素数を返す以下のメソッドを作成せよ。

```
static int searchIdx(int[] a, int n, int key, int[] idx)
```

たとえば、要素数 8 の配列 a の要素が {1，9，2，9，4，6，7，9} であって、*key* が 9 であれば、配列 *idx* に {1，3，7} を格納するとともに 3 を返却する。

演習 3–4

右のように、２分探索の過程を詳細に表示するプログラムを作成せよ。

各行の左端に中央要素（現在着目している要素）のインデックスを表示するとともに、探索範囲の先頭要素の上に "<-" を、末尾要素の上に "->" を、着目している中央要素の上に "+" を表示すること。

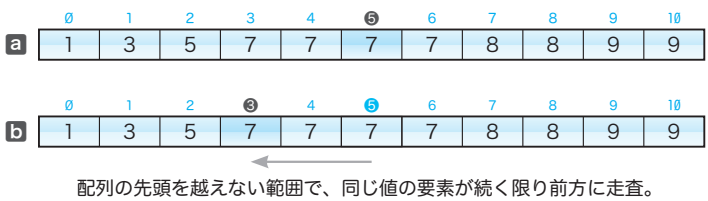
```
   |  0  1  2  3  4  5  6
---+-----------------------
   | <-           +     ->
 3|  1  2  3  5  6  8  9
   |
   | <-  +  ->
 1|  1  2  3  5  6  8  9

2はx[1]にあります。
```

演習 3–5

２分探索アルゴリズムでは、探索すべきキー値と同じ値をもつ要素が複数存在する場合、それらの要素の先頭要素を見つけるとは限らない。たとえば、下図に示す配列から 7 を探索すると、中央要素のインデックスである 5 を見つける。

２分探索アルゴリズムによって探索に成功した場合（下図ⓐ）、その位置から先頭側へ一つずつ走査すれば（下図ⓑ）、複数の要素が一致する場合でも、最も先頭側に位置する要素のインデックスを見つけられる。



配列の先頭を越えない範囲で、同じ値の要素が続く限り前方に走査。

そのように改良したメソッドを作成せよ。

```
static int binSearchX(int[] a, int n, int key)
```