第3章

探　索

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**（*searching*）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》を考えることにします。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るでしょう。

- 福岡県出身の人を探す。
- 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。

いずれの探索においても、**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。出身県の探索を行う場合は出身県がキーですし、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、データ自体が単なる整数値であれば、そ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。

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。

もちろん、ある値と一致するキー値をもつデータを探すのが、単純であって、なおかつ一般的です。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

探索の手法としては、数多くのものが考案されています。

それらの中には、データの格納先のデータ構造に依存するアルゴリズムもあります。いくつかの探索例を **Fig.3-1** に示しています。

▶ 図**b**に示す線形リストからの探索は第 9 章で、図**c**に示す 2 分探索木からの探索は第 10 章で学習します。また、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。

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。

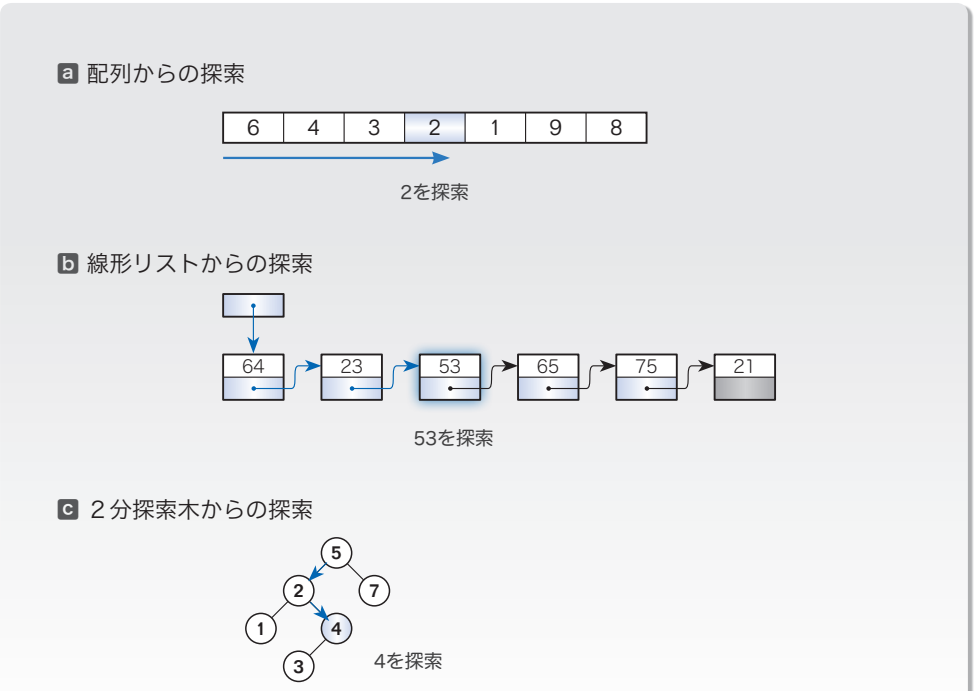


Fig.3-1 探索の例

- **線形探索**：ランダムなデータの並びからの探索を行う。
- **2分探索**：ソートされたデータの並びから高速に探索を行う。
- **ハッシュ法**：探索・追加・削除のいずれをも高速に行う。
  - **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。
  - **オープンアドレス法**：衝突時に再ハッシュを行う。

もしも、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することになります。

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）ことによって実現できます。

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	9	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。

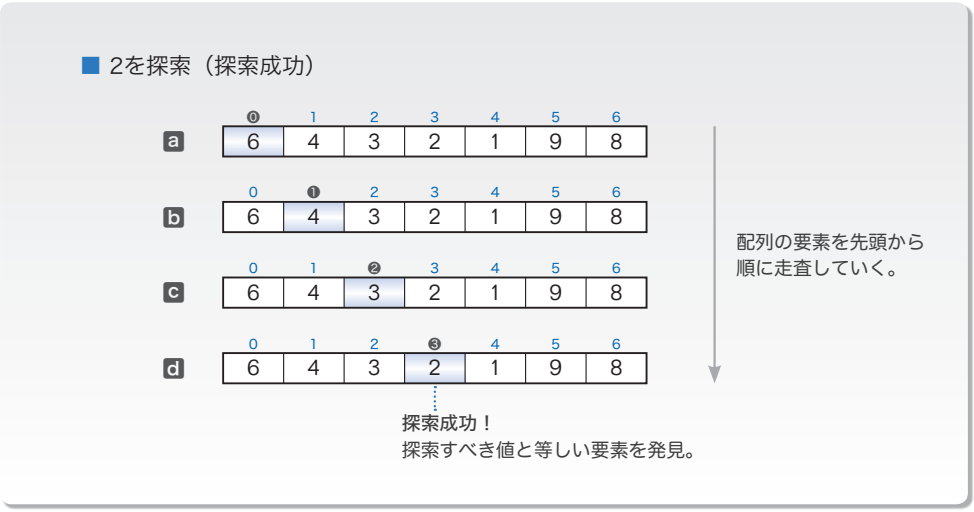


Fig.3-2 線形探索（探索成功）

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ありません。

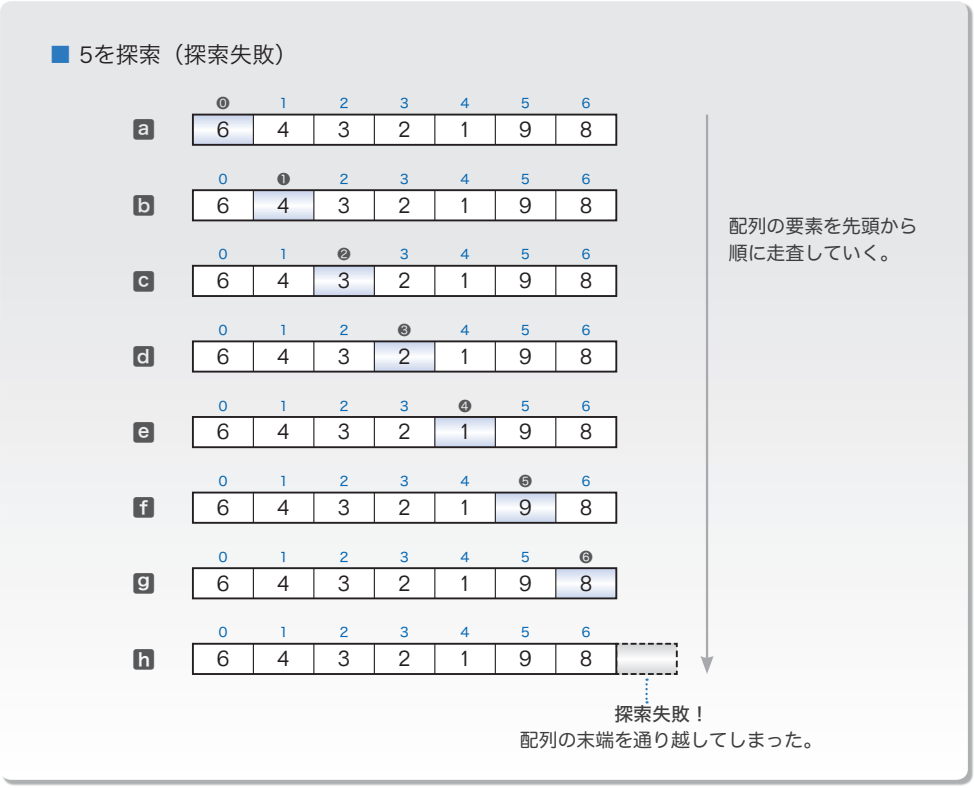


Fig.3-3 線形探索（探索失敗）

成功例と失敗例から、配列の走査の終了条件は、二つで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（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。

条件1が成立したときは**探索失敗**で、条件2が成立したときは**探索成功**です。

要素数を n とすると、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は $n + 1$ 回で、2は n 回となります。

*

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：120↵
その値はx[4]にあります。
```

メソッド *seqSearch* は、配列 *a* の先頭 *n* 個の要素から、*key* と同じ値をもつ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、*key* と同じ値をもつ要素が複数個存在する場合に返却するのは、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、*key* と同じ値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。

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。したがって、実際には繰返しのたびに三つの条件判定が行われます。

List 3-2 に示すのは、配列の走査を **for** 文で実現したプログラムです（こちらのほうが簡潔です）。

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、ソートされていない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であることが分かります。

最後まで読まないと無限ループであることが分からない **do** 文は、あまりお勧めできるものではありません。

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。単純な判定であるとはいっても、“塵も積もれば山となる”のですから、そのコストは決して小さくありません。

ここで学習する**番兵法**は、**このコストを半分に抑える手法です。Fig.3-2** と **Fig.3-3** に示したものと同じ探索を、番兵法によって行う例を **Fig.3-4** に示します。

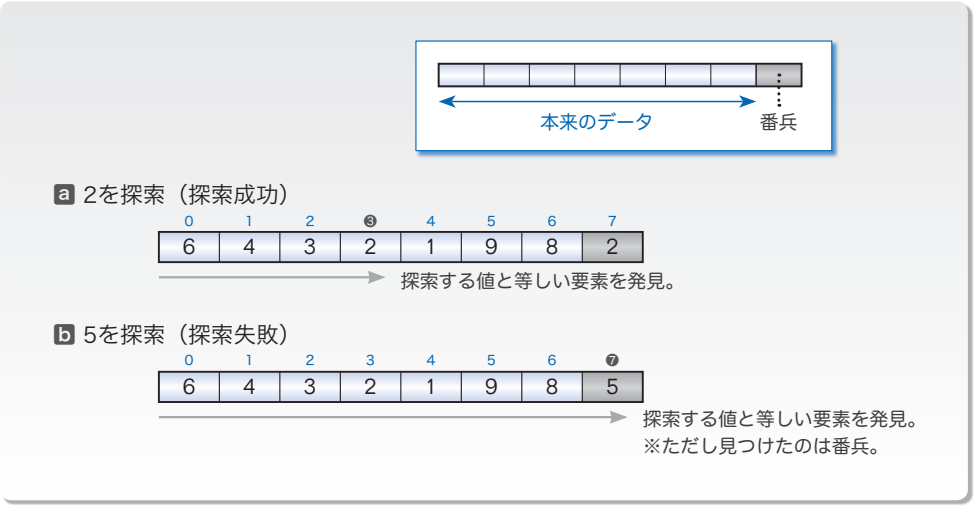


Fig.3-4 線形探索（番兵法）

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行うのは、末尾要素 a[7] に対して、**探索するキー値と同じ値を格納すること**です。このとき格納するデータが**番兵**（*sentinel*）です。

> 図a：2 を探索するために、番兵として a[7] に 2 を格納。
> 図b：5 を探索するために、番兵として a[7] に 5 を格納。

そうすると、図bのように、たとえ目的とする値が本来のデータ内に存在しなくても、a[7] まで走査した段階で終了条件2が必ず成立します。そのため、条件1の判定は**不要**となります。

番兵は、繰返しの終了判定を簡略化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。

このプログラムでは、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（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できるようにするためです。

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;                                ←2
         i++;
      }
      return i == n ? -1 : i;                     ←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。したがって、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、**配列内の本来のデータを見つけたのか、それとも番兵を見つけた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済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。

次に示す昇順にソートされたデータの並びから39を探索することを考えましょう。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。

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。

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。

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、探索の対象は先頭側の2個すなわちa[6]～a[7]に絞り込めます。

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、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るため、二つのインデックス6と7の中央値(6 + 7) / 2は6となります）。

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。

探索範囲の先頭インデックスを*pl*、末尾インデックスを*pr*、中央インデックスを*pc*と表すことにします。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。

□の要素が探索の対象範囲であり、■の要素は探索の対象から外れた範囲です。探索範囲は、比較のたびに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し

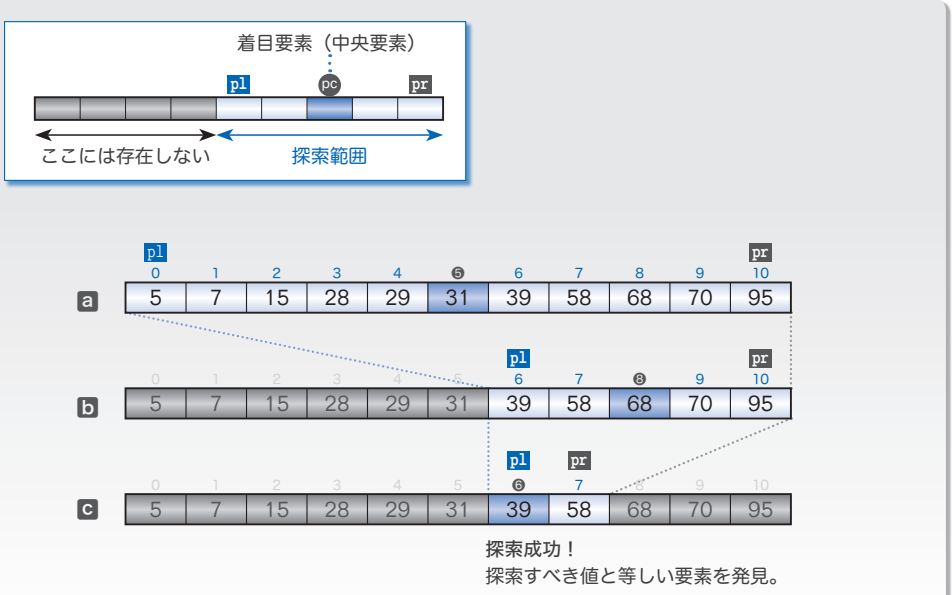


Fig.3-5　2分探索の成功例（39を探索）

ていく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**。

a[*pc*]と*key*を比較して、等しければ**探索成功**です（図c）が、そうでなければ探索範囲を次のように縮小します。

■ 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す。

探索範囲は中央要素より後方の a[*pc* + 1] ～ a[*pr*] に絞り込めます。

そのために、*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す。

探索範囲は中央要素より前方の a[*pl*] ～ a[*pc* - 1] に絞り込めます。

そのために、*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。

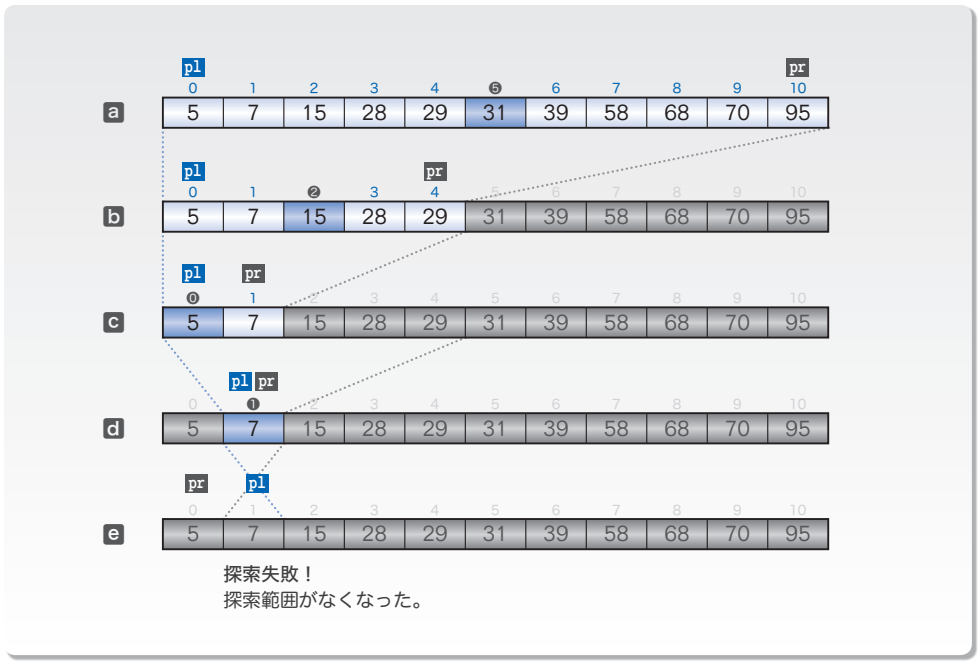


Fig.3-6 2分探索の失敗例（6を探索）

a 探索すべき範囲は配列全体すなわち a[0] ～ a[10] であり、中央要素である 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**探索範囲がなくなって終了条件②が成立します**ので、探索に失敗します。

2分探索を行うプログラムを **List 3-4** に示します。

▶ このアルゴリズムでは、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ので、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。そのため、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》の例からも分かるように、アルゴリズム選択の際は、二つの計算量のバランスを考える必要があります。

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メソッドをもとに、時間計算量を考えていきましょう。

```
//--- 線形探索 ---//
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
}
```

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が等しいかどうかを判断するための3が行われるのは、平均して n / 2 回です。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 の略です。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

	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る**のです。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 のメソッド *seqSearchSen* を、**while** 文ではなく **for** 文を用いて書きかえよ。

□ 演習 3-2

右のように、線形探索の走査の過程を詳細に表示するプログラムを作成せよ。

着目要素の上に '*' 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

その値はa[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。以下に示すプログラム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
8     else
9        pr = pc - 1;         // 探索範囲を前半に絞り込む
10 } while (pl <= pr);

11 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

	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	log n	O(log n)
11	1	O(1)

したがって、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) が O(1) より大きいのは当然ですね。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。

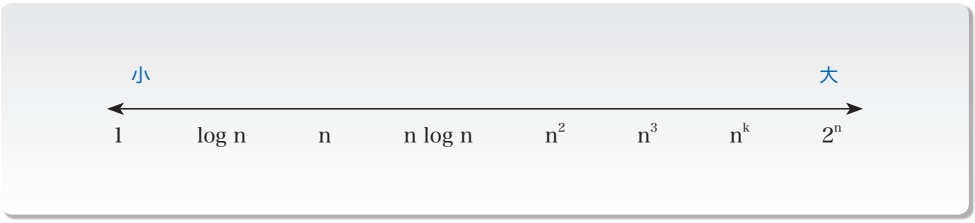


Fig.3-7 計算量と増加率

□ 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)
```

□ 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。

探索範囲の先頭要素の上に "<-" を、末尾要素の上に "->" を、中央要素（着目要素）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

その値はa[1]に存在します。
```

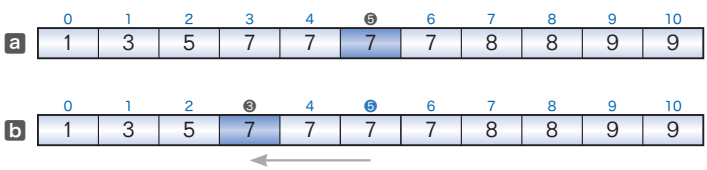
□ 演習 3-5

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 を見つけることになる。

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a）、その位置から先頭側へ走査することによって（下図b）、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。

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)
```



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。