データ構造とアルゴリズム

第2回　探索とソート

メディア情報コース
平居　悠（ひらい　ゆたか）

到達目標

- C言語を通じてデータ構造とアルゴリズムを理解する。
- 目的に応じて効率的なプログラムを自由に設計できる。

前回

第１回　構造体と共用体
第２回　探索とソート
第３回　メモリの動的確保と線形リスト
第４回　文字列の類似度計算、再帰、動的計画法
第５回　自由課題作成
第６回　自由課題作成
第７回　自由課題発表

前回の目標

構造体を定義してプログラムで使えるようになる。

前回学んだこと

1.C言語基本文法の復習
2.構造体の宣言
3.構造体へのポインタ
4.共用体

構造体 (structure)

異なるデータ型をひとまとめにして扱うことができる機能を持つデータ型の一種

構造体の宣言

```
struct 構造体タグ{
    メンバ;
    …;
};
```

メンバ：構造体が有する色々な型のデータ。

構造体のテンプレート

```
struct data{
    char name[32];
    int age;
    double bl;
    double bw;
};
```

テンプレート (templete, 鋳型)を
宣言しただけではまだ使えない。

構造体変数の宣言

struct data man;

構造体変数manはメンバに
name, age, bl, bwを持つ。

構造体メンバへのアクセス

構造体のメンバにアクセスするには、**構造体メンバ演算子 (.)**を使う。

構造体の配列

構造体変数の配列を作るには、

```
struct Man student;
```

としていたところを、

```
struct Man student[5];
```

などとすればよい。

構造体の配列

student[2]のnameメンバにアクセスするには、

student[2].name

とすればよい。

構造体へのポインタ

構造体変数を

struct Man *student;

のように宣言すると、studentはMan構造体へのポインタとなる。

構造体へのポインタが指す構造体メンバへのアクセス

アロー演算子 (->)を用いる

```
student->name;
```

新しい型の定義

typedefキーワードを使えば、データ型の名前を自由に定義できる。

typedef 既存の型名 新しい型名;

新しい型の定義

```
typedef struct Data {
    int age;
    double bl;
    double bw;
} MyDataType, *lpMyDataType;
```

とすれば、Data型の構造体変数を宣言するとき、

```
MyDataType data;
```

と書ける。

新しい型の定義

```
typedef struct Data {
    int age;
    double bl;
    double bw;
} MyDataType, *lpMyDataType;
```

この構造体へのポインタを宣言するときは、

```
lpMyDataType lpdata;
```

と書ける。

自己参照的構造体

```
struct data {
    char name[32];
    int age;
    struct data *nextdata;
};
```

のように、構造体メンバには自分自身と同じ型へのポインタを持たせることができる。

共用体 (union)とは

１つの記憶領域を確保して、そこに色々なデータ型のデータを格納できる機能を持つデータ型

共用体の宣言

```
union 共用体タグ{
    共用体メンバ;
    …;
} 共用体変数;
```

共用体の特徴

- 記憶できるのは１つのメンバだけ。後で別なメンバに値を代入すると、前のメンバの値は無効となる。
- 初期化はできない。

桑井康孝『猫でもわかるC言語プログラミング』
第３版、ソフトバンククリエイティブ

前回の問い

- 構造体を宣言すると何ができるのか？
  - 構造体は異なるデータ型をひとまとめに扱うことができる。
- 構造体ポインタが指す構造体のメンバにアクセスするには、どのようにすれば良いか？
  - アロー演算子 (->)を用いる。
- 共用体とは何か？
  - １つの記憶領域を確保して、そこに色々なデータを格納できる機能を持つデータ型。

今回

第１回　構造体と共用体
第２回　探索とソート
第３回　メモリの動的確保と線形リスト
第４回　文字列の類似度計算、再帰、動的計画法
第５回　自由課題作成
第６回　自由課題作成
第７回　自由課題発表

今回の目標

データを効率的に探索し、
ソートできるようになる。

今回学ぶこと

1.二分探索
2.クイックソート

今回の問い

- 二分探索とは何か？
- 配列のソートをするにはどのような関数を用いれば良いのか？

今回学ぶこと

1.二分探索

2.クイックソート

アルゴリズムとは

問題を解決する手順や方法

二分探索 (binary search)

要素が昇順（小さい順）または降順（大きい順）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

二分探索のアルゴリズム

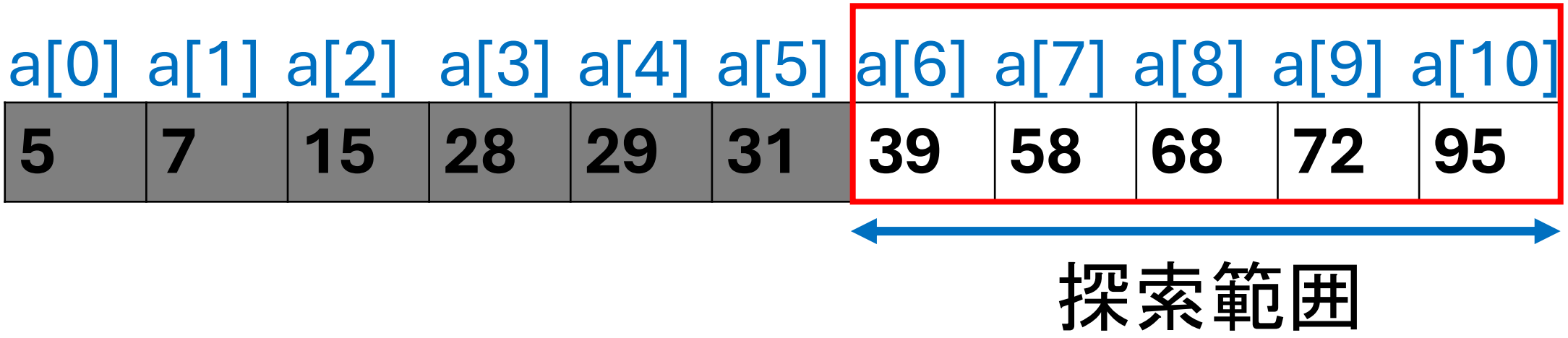
以下のようなデータが配列a[]に格納されているとする。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
5	**7**	**15**	**28**	**29**	**31**	**39**	**58**	**68**	**72**	**95**

値39をこのデータの中から探索する。

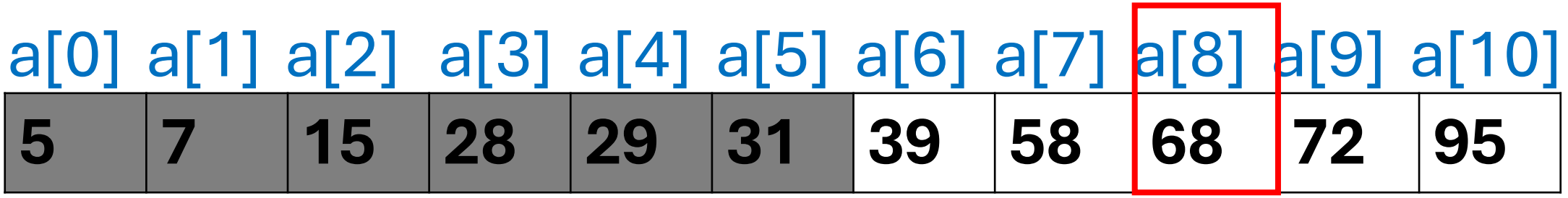
配列の中央に位置する要素a[5]に着目する。a[5] = 31 < 39なので、目的とする39はこの要素31よりも末尾側に存在するはず。

二分探索のアルゴリズム



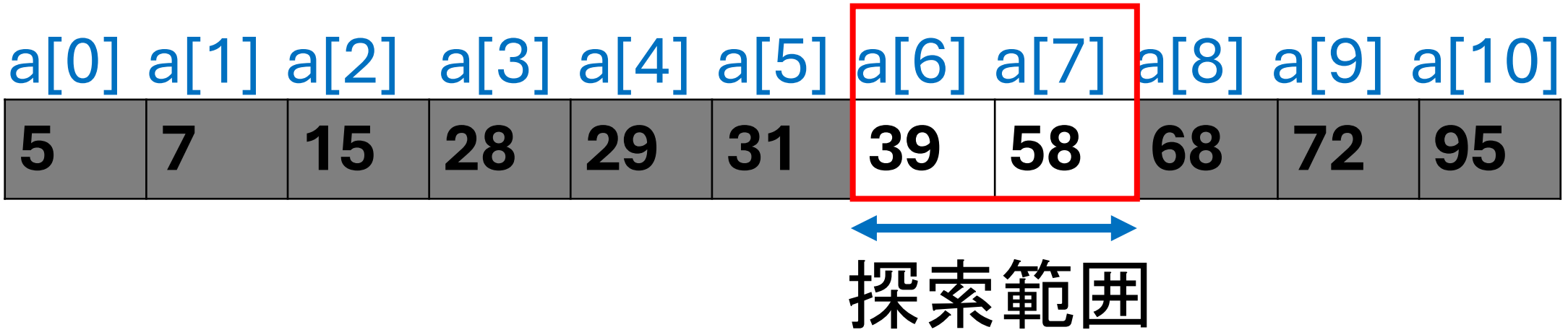
そこで、探索の対象を末尾側の
a[6]~a[10]の5個に絞り込む。

二分探索のアルゴリズム



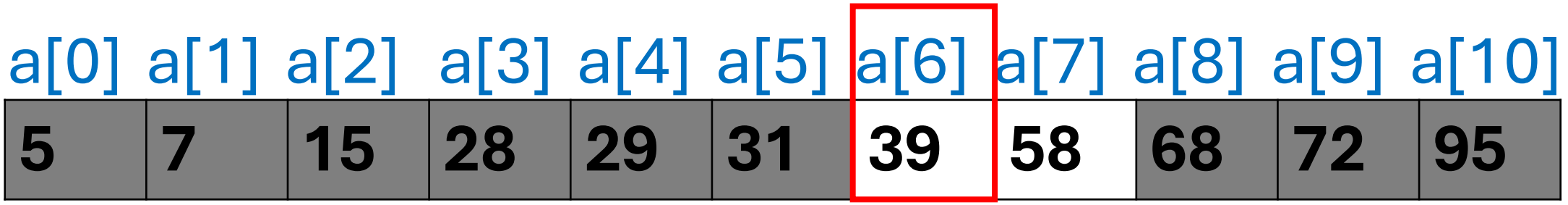
絞り込まれた対象範囲の中央要素であるa[8]に着目する。a[8] = 68 > 39より、目的とする値39は、この要素68より先頭側に存在するはず。

二分探索のアルゴリズム



そこで、探索の対象を先頭側のa[6], a[7]の2個に絞り込む。

二分探索のアルゴリズム

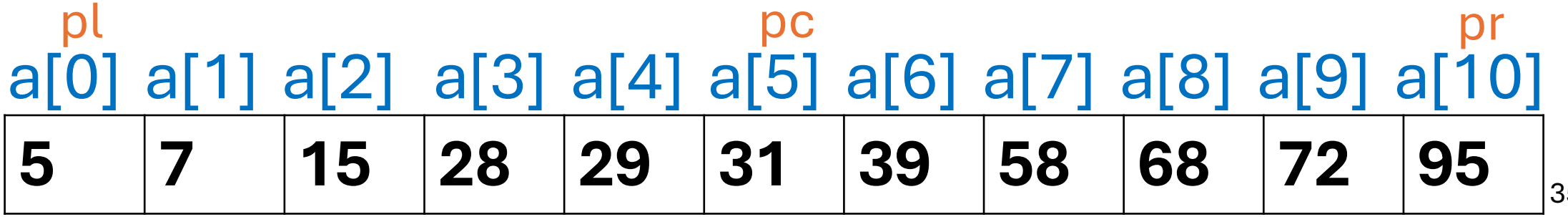


探索成功！

2つの要素の中央要素はどちらでも構わないが、先頭側の39に着目する。これは目的とする値なので、探索成功！

二分探索のアルゴリズム

- n個の要素が昇順に並んでいる配列aからある値keyを探索する。
- 探索範囲の先頭の添字をpl、末尾の添字をpr、中央の添字をpcとする。
- 探索開始時のplは0、prはn−1、pcは(n−1)/2である。
- a[pc]とkeyを比較して、等しければ探索成功、そうでなければ探索範囲を次ページのように縮小する。



二分探索のアルゴリズム

- **a[pc] < keyのとき**
  a[pl]からa[pc]はkeyより小さい。
  探索範囲を中央要素より後方のa[pc+1]からa[pr]に絞り込む。そのために、plの値をpc+1に更新する。
- **a[pc] > keyのとき**
  a[pc]からa[pr]はkeyより大きい。
  探索範囲は中央要素より前方のa[pl]からa[pc−1]に絞り込む。そのために、prの値をpc−1に更新する。

二分探索のアルゴリズム

- 探索範囲は比較のたびに（ほぼ）半分に絞り込まれる。
- 次に示す条件1、2のいずれか一方が成立すれば、アルゴリズムが終了する。
  1. **a[pc]とkeyが一致した（探索成功）。**
  2. **探索範囲が無くなった（探索失敗）。**

二分探索失敗の場合

値6を以下のデータの中から探索する。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
5	**7**	**15**	**28**	**29**	**31**	**39**	**58**	**68**	**72**	**95**

配列の中央に位置する要素a[5]に着目する。
a[5] = 31 > 6なので、探索する範囲はa[0]からa[4]に絞り込める。

二分探索失敗の場合

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
5	7	15	28	29	31	39	58	68	72	95

縮小された範囲の中央要素a[2]の値は15。
15＞6より、探索する範囲をa[0]とa[1]に絞り込む。

二分探索失敗の場合

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
5	7	15	28	29	31	39	58	68	72	95

縮小された範囲の中央要素a[0]の値は5。
5＜6より、探索すべき範囲はa[1]のみとなる。a[1]=7なので、探索失敗。

bsearch関数

ソート済み配列からの探索を行う。

ヘッダ：#include <stdlib.h>

形式：

```
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

bsearch関数の機能

```
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

baseが先頭要素を指している、

- 要素数：nmemb個
- 要素の大きさ：size

の配列からkeyが指すオブジェクトに一致する要素を探索する。

bsearch関数の返却値

- 配列中の一致する要素へのポインタを返す。
- 一致する要素がないときは、空ポインタを返す。
- 二つの要素が等しいとき、どちらの要素と一致するかは規定されない。

bsearch関数の特徴

- 探索対象の配列はソート済みでなければならない。
- 探索する値と同じ値をもつ要素が複数個存在する場合に、最も先頭の要素を見つけるとは限らない。

bsearch.cというファイルを作成し、次のコードを書いて実行してみよう。

```
#include <stdio.h>
#include <stdlib.h>

// 整数を比較する関数
int int_cmp(const int *a, const int *b)
{
   if (*a < *b)
      return -1;
   else if (*a > *b)
      return 1;
   else
      return 0;
}
```

```
int main()
{
   int nx, ky;
   printf(“要素数: ” );
   scanf(“%d”, &nx);
   int *x = calloc(nx, sizeof(int)); //要素数nxのint型配列xを生成
   printf(“昇順に入力せよ。\n”);
   printf(“x[0]: ”);
   scanf(“%d”, &x[0]);

   for(int i = 1; i < nx; i++) {
      do{
         printf(“x[%d]: “, i);
         scanf(“%d”, &x[i]);
      }while (x[i] < x[i-1]);  //一つ前の値よりも小さければ再入力
   }
   printf(“探す値: ”);
   scanf(“%d”, &ky);
   int *p = bsearch(
                  &ky, //探索値へのポインタ
                  x, //配列
                  nx, //要素数
                  sizeof(int), //要素の大きさ
                  (int (*)(const void *, const void *))int_cmp //比較関数
               );
   if (p == NULL)
      puts(“探索に失敗しました。”);
   else
      printf(“%dはx[%d]にあります。\n”, ky, (int)(p –x));
   free(x); //配列xを破棄
   return 0;
}
```

型修飾子const

受け取った変数や配列を関数内で書き換えられないようにする。

例：

const int

const double

bsearch関数

```
bsearch(
   &ky, //探索値へのポインタ
    x, //配列
    nx, //要素数
    sizeof(int), //要素の大きさ
    (int (*)(const void *, const void *))int_cmp //比較関数
);
```

第１引数 (&ky): キー値（探索すべき値）が格納されたオブジェクトへのポインタ。

bsearch関数

```
bsearch(
    &ky, //探索値へのポインタ
    x, //配列
    nx, //要素数
    sizeof(int), //要素の大きさ
    (int (*)(const void *, const void *))int_cmp //比較関数
);
```

第２引数 (x): 探索対象である配列の先頭要素へのポインタ。

bsearch関数

```
bsearch(
    &ky, //探索値へのポインタ
     x, //配列
     nx, //要素数
     sizeof(int), //要素の大きさ
     (int (*)(const void *, const void *))int_cmp //比較関数
);
```

第３引数 (nx): 配列の要素数。

bsearch関数

```
bsearch(
    &ky, //探索値へのポインタ
     x, //配列
     nx, //要素数
     sizeof(int), //要素の大きさ
     (int (*)(const void *, const void *))int_cmp //比較関数
);
```

第４引数 (sizeof(int)): 配列の要素の大きさ。

bsearch関数

```
bsearch(
    &ky, //探索値へのポインタ
     x, //配列
     nx, //要素数
     sizeof(int), //要素の大きさ
     (int (*)(const void *, const void *))int_cmp //比較関数
);
```

第５引数：比較関数へのポインタ。

比較関数

- 探索の過程では、探索するキー値と配列内の要素の値とを比較して、大小関係を判定する必要がある。
- ところが、大小関係の判定方法は要素型（整数、文字列など）によって異なる。
- そのため、二つの値を比較した結果を、次に示す値として返却する**比較関数**を利用者が用意して、その関数へのポインタをbsearch関数の第５引数として渡す必要がある。

比較関数

```
int int_cmp(const int *a, const int *b)
{
    if (*a < *b)
        return -1;
    else if (*a > *b)
        return 1;
    else
        return 0;
}
```

- 第１引数が指す値の方が**小さければ：負の値**
- 第１引数が指す値と第２引数が指す値が**等しければ：0**
- 第１引数が指す値の方が**大きければ：正の値**

比較関数（条件演算子?を用いた書き方）

```
int int_cmp(const int *a, const int *b)
{
    return *a < *b ? -1 : *a > *b ?1 : 0;
}
```

比較関数の誤った実装

```
int int_cmp(const int *a, const int *b)
{
    return *a -*b;
}
```

- 上記の実装は減算の演算結果がint型で表現できる値を超えてオーバーフローする可能性がある。
- 例えば、int型の表現範囲が-32768~32768のとき、30000から-10000を引くなどとするとint型の表現範囲に収まらない。

bsearch関数の呼び出し

```
bsearch(
    &ky, //探索値へのポインタ
    x, //配列
    nx, //要素数
    sizeof(int), //要素の大きさ
    (int (*)(const void *, const void *))int_cmp //比較関数
);
```

比較関数int_cmpが受け取る引数の型がconst int *なのに対し、bsearch関数が受け取る比較関数の引数はconst void *。両者の型が異なるため、bsearch関数の呼び出しには**キャスト**が必要。

呼び出し時のキャスト不要な比較関数

```
int int_cmp(const void *a, const void *b)
{
    if (*(int *)a < *(int *)b)
        return -1;
    else if (*(int *)a > *(int *)b)
        return 1;
    else
        return 0;
}
```

比較関数の中がキャストだらけになる。

今回の問い

- 二分探索とは何か？
  - 要素が昇順または降順にソートされている配列から効率良く探索を行うアルゴリズム
- 配列のソートをするにはどのような関数を用いれば良いのか？

今回学ぶこと

1.二分探索

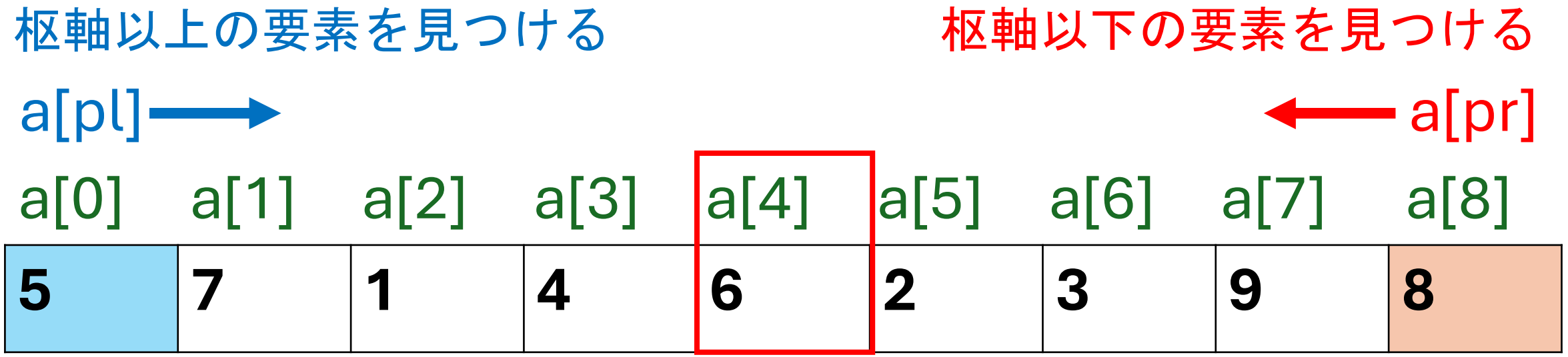
2.クイックソート

クイックソートのアルゴリズム

- 配列内のどれか一つの要素に着目。
- これを**枢軸**（すうじく、pivot）と呼ぶ。
- 枢軸以下のグループと枢軸以上のグループに分割する作業を繰り返すことによってソートを行う。

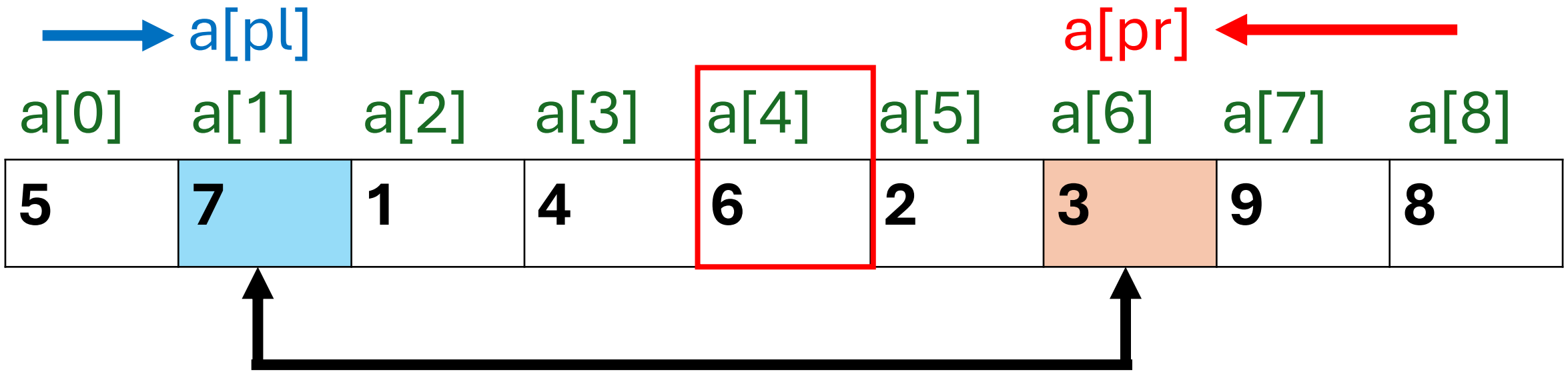
クイックソートのアルゴリズム

以下の要素数9の配列aから枢軸として6を選んで分割する。

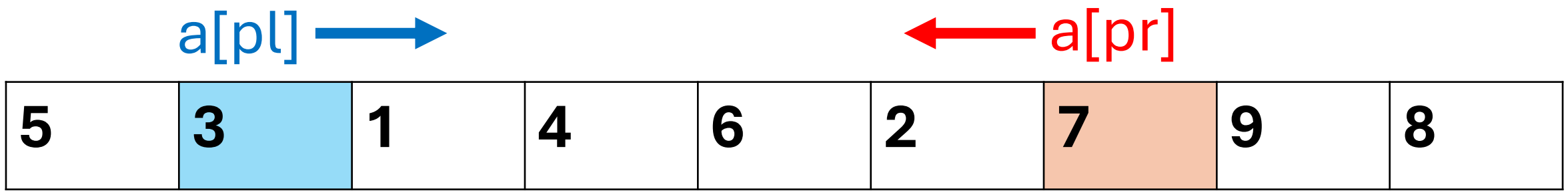


- a[pl] ≥ 6が成立する要素が見つかるまでplを右方向へ走査する。
- a[pr] ≤ 6が成立する要素が見つかるまでprを左方向へ走査する。

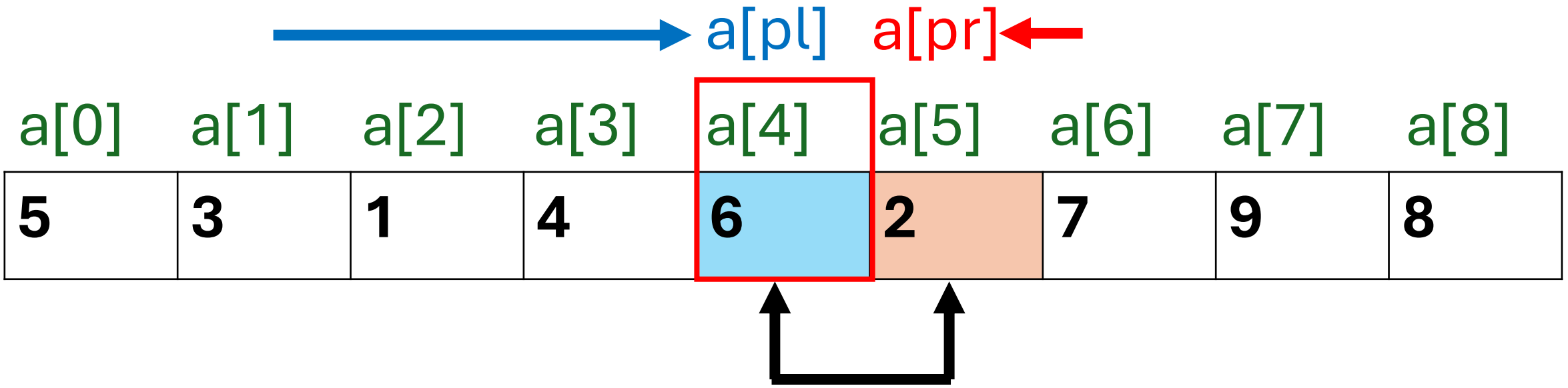
クイックソートのアルゴリズム



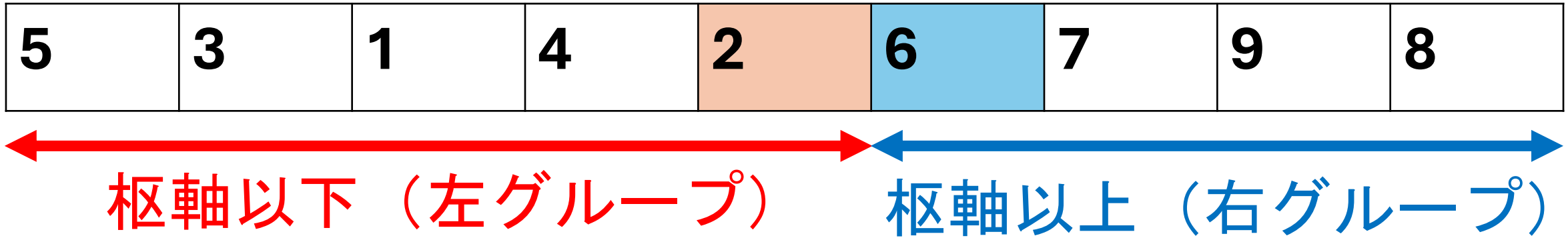
ここで、a[pl]とa[pr]の値を交換する。



クイックソートのアルゴリズム



ここで、a[pl]とa[pr]の値を交換する。



クイックソートのアルゴリズム

- 一旦配列の分割が完了すると、二つの部分に対して同じ手続きを適用して再分割する。
- 分割によって作られたグループの要素数が1になったら、それ以上分割しない。

1	2	3	4	5	6	7	8	9

qsort関数

配列のソートを行う。

ヘッダ：#include <stdlib.h>

形式：

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

qsort関数の機能

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

baseが先頭要素を指している、

- 要素数がnmemb個
- 要素の大きさがsize

の配列を、comparが指す比較関数に従って整列する。

qsort関数の特徴

- bsearch関数と同様に、int型やdouble型などの基本型の配列だけでなく、構造体型の配列など、あらゆる型の配列に適用できる。
- アルゴリズムは処理系まかせで、「クイックソート」が使われるとは限らない。

qsort.cというファイルを作成し、次のコードを書いて実行してみよう。

```
#include <stdio.h>
#include <stdlib.h>

// 整数を比較する関数
int int_cmp(const int *a, const int *b)
{
   if (*a < *b)
      return -1;
   else if (*a > *b)
      return 1;
   else
      return 0;
}
```

```
int main()
{
   int nx;

   printf(“qsortによるソート”);
   printf(“要素数: ”);
   scanf(“%d”, &nx);
   int *x = calloc(nx, sizeof(int)); //要素数nxのint型配列xを生成

   for(int i = 0; i < nx; i++) {
      printf(“x[%d]: “, i);
      scanf(“%d”, &x[i]);
   }

   qsort(x, //配列
         nx, //要素数
         sizeof(int), //要素の大きさ
         (int (*)(const void *, const void *))int_cmp //比較関数
        );
   puts(“昇順にソートしました。”);
   for(int i = 0; i < nx; i++)
      printf(“x[%d] = %d\n”, i, x[i]);
   free(x); //配列xを破棄

   return 0;
}
```

今回の問い

- 二分探索とは何か？
  - 要素が昇順または降順にソートされている配列から効率良く探索を行うアルゴリズム
- **配列のソートをするにはどのような関数を用いれば良いのか？**
  - **qsortを用いる。**

今回学んだこと

1.二分探索
2.クイックソート

今回の目標

データを効率的に探索し、
ソートできるようになる。

参考書

Textbook MEIKAI
by Dr.BohYoh Shibata

SB Creative

課題

ユーザーに要素数、ランダムな整数と探す値を入力させ、
その値を探索するプログラムalgorithm2.cを作成せよ。

提出先：yutaka.hirai@koeki-u.ac.jp

件名：データ構造とアルゴリズム第2回課題 [学籍番号]

提出物：algorithm2.c

締め切り：12月10日 (木)

次回

第１回　構造体と共用体
第２回　探索とソート
第３回　メモリの動的確保と線形リスト
第４回　文字列の類似度計算、再帰、動的計画法
第５回　自由課題作成
第６回　自由課題作成
第７回　自由課題発表