

データ構造入門及び演習

9回目：整列処理（クイックソート）

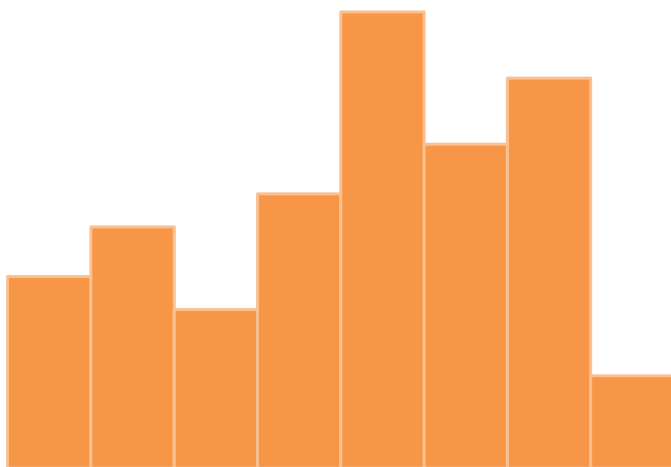
2014/06/13

担当：見越 大樹

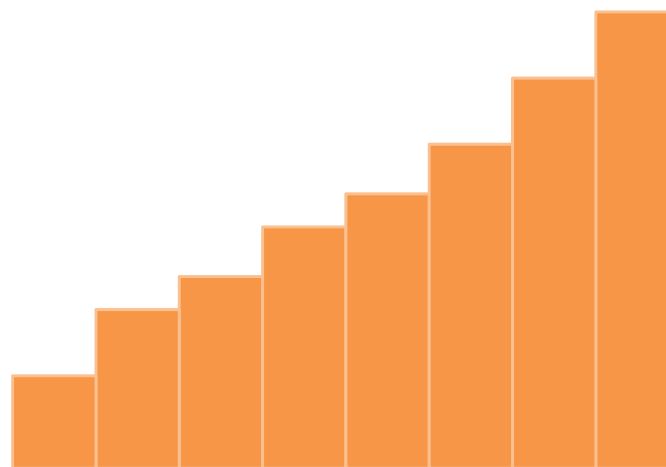
61号館304号室

ソート (sorting)

- ソートとは？
 - データの整列処理（並び替え）
 - データを一定の基準に従って並び替えること（大きい順, 小さい順, など）



整列前データ



整列後データ

整列処理の種類

- 整列処理を行う方法には、様々な種類がある
 - バブルソート(単純交換ソート) 本講義で学習(復習)
 - 挿入法(単純挿入法) 本講義で学習
 - クイックソート 本講義で学習
 - ヒープソート
- 整列の3つの基本処理:
 1. 比較
 2. 交換
 3. 挿入

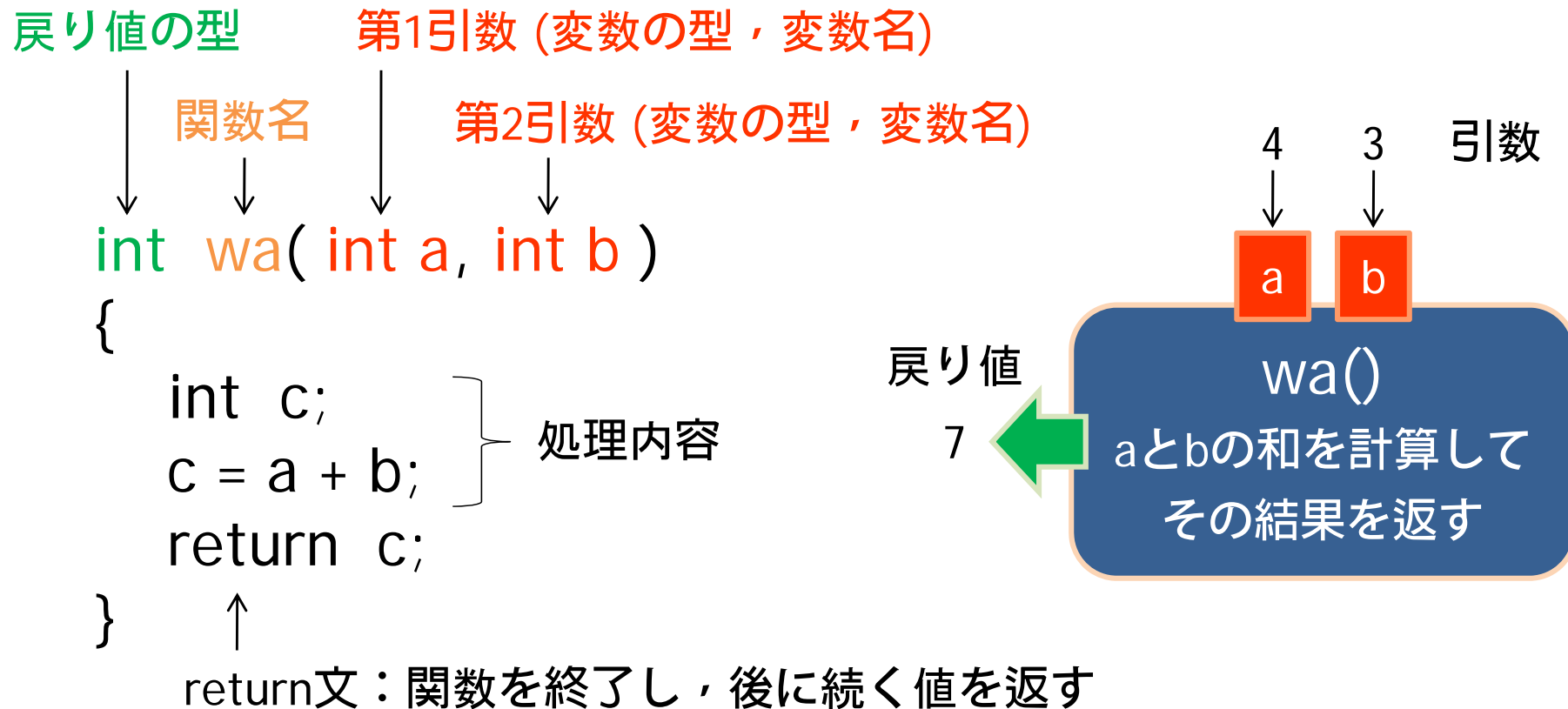
クイックソート

- より効果的な整列処理のために！
- 最も高速なソートアルゴリズムの1つ
- ソートされていない(バラバラな)データに有効
(平均的には最も速いソート法)
- 再帰処理を使用することによって、効率的で短いソースコードを実現している

関数(復習)

- 関数とは？

- 入力値を指示どおりに処理し、結果を返す処理のかたまり



再帰呼び出し (復習)

- 再帰呼び出しとは？
 - 関数が自分自身(同じ形の関数)を呼び出すこと
 - 再帰: リカーシブ(recursive)コール, 元に戻る, 繰り返し
 - 昔の言語(FORTRAN, BASIC等)では通常サポートされていない
- 効果:
 - 繰り返し処理の代用
 - プログラムサイズ(ソースコード)を格段に小さくできることもある

自然数の和を実現する関数例 (復習)

(再帰呼び出しを使用する場合)

```
#include <stdio.h>

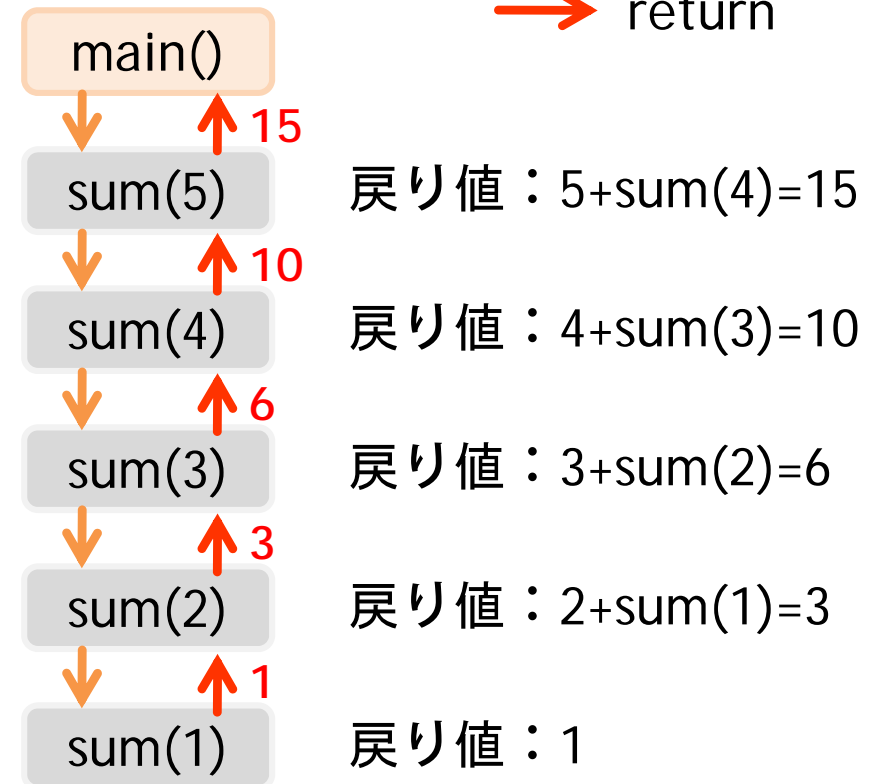
int sum( int );

int main()
{
    int n;
    n = sum(5);
    return 0;
}

int sum( int n )
{
    if (n==1) // 再帰の終了条件
        return 1;
    else
        return n+sum(n-1); // 再帰処理
}
```

例) n=5の場合

→ 呼び出し
→ return

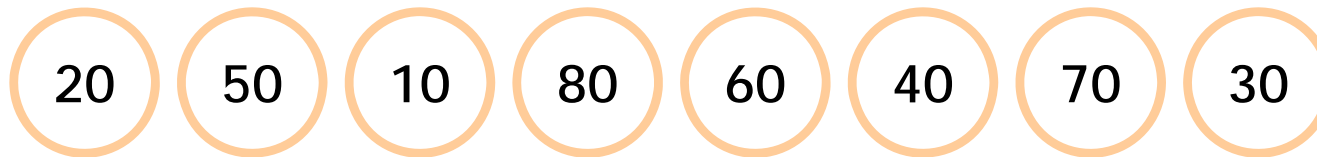


クイックソート

- 基準値を決めて、それより大きい数と小さい数のグループに分割する
- 分割されたグループに対しても同じ処理を繰り返す

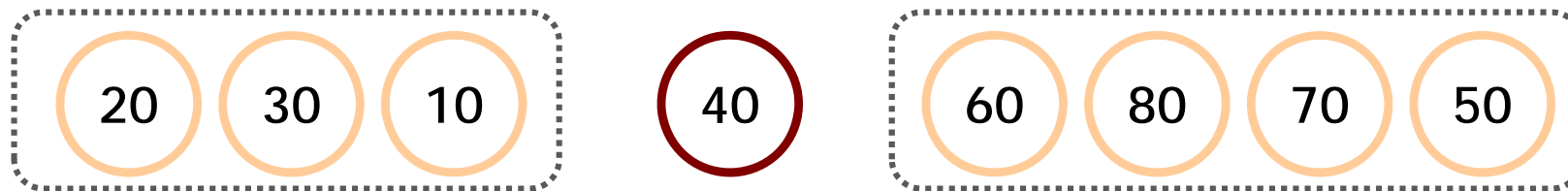
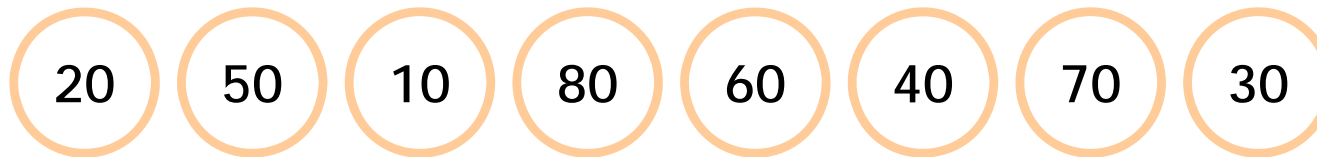
クイックソートのイメージ

問題: 8個の荷物を天秤を使って軽い順に並び替える



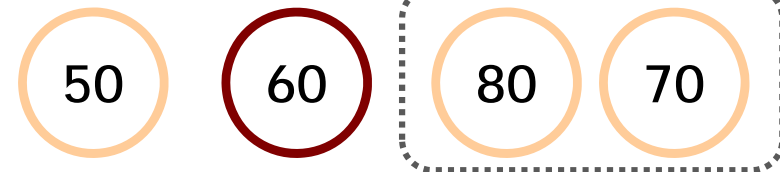
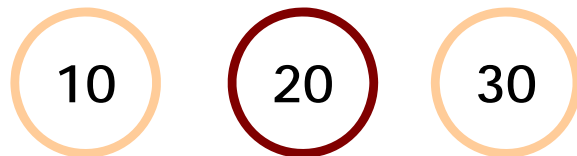
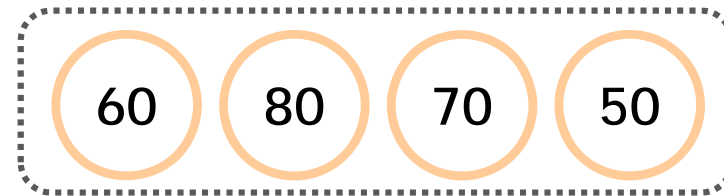
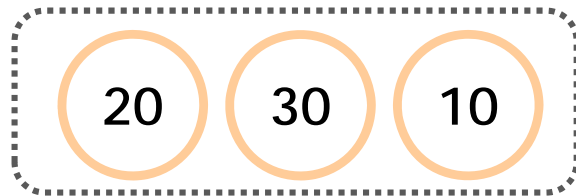
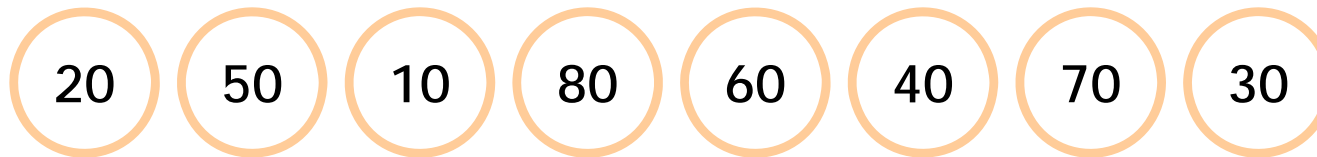
クイックソートのイメージ

問題: 8個の荷物を天秤を使って軽い順に並び替える



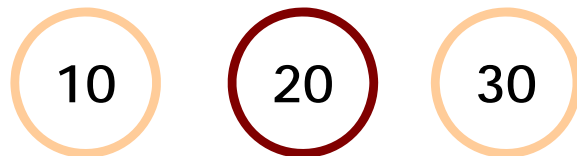
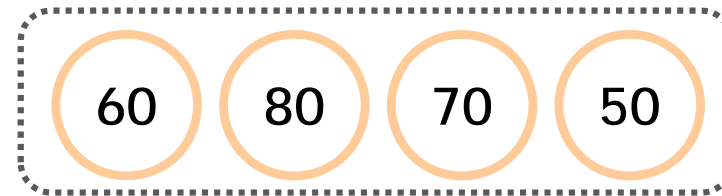
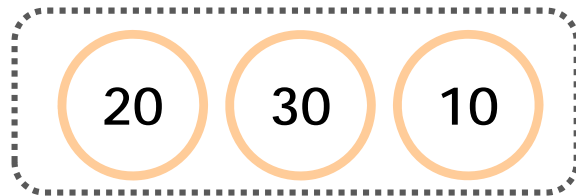
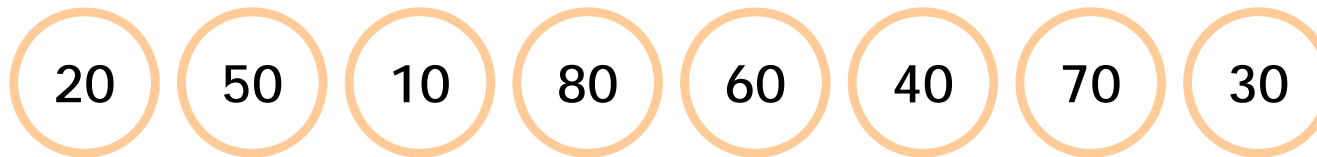
クイックソートのイメージ

問題: 8個の荷物を天秤を使って軽い順に並び替える

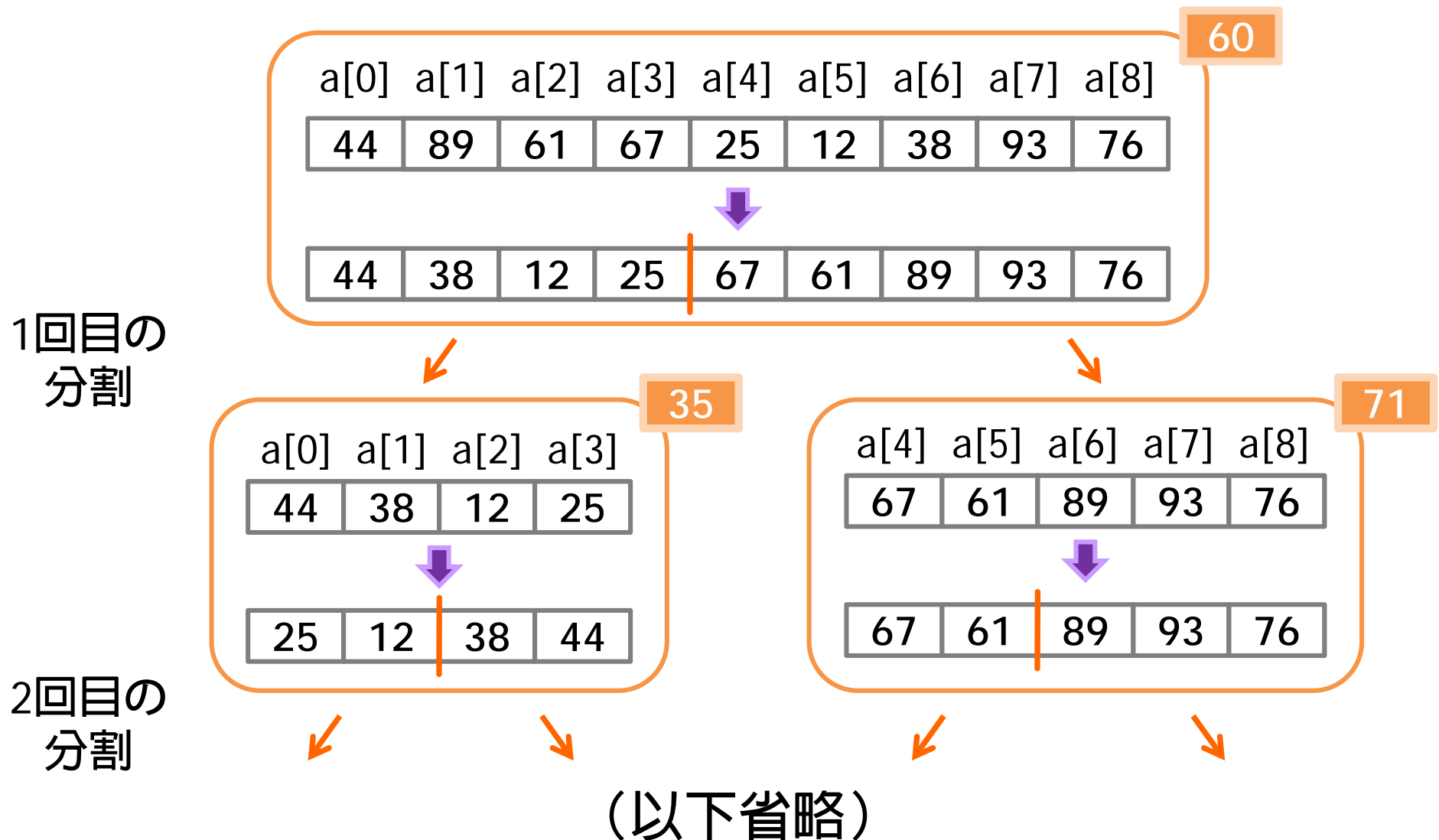


クイックソートのイメージ

問題: 8個の荷物を天秤を使って軽い順に並び替える



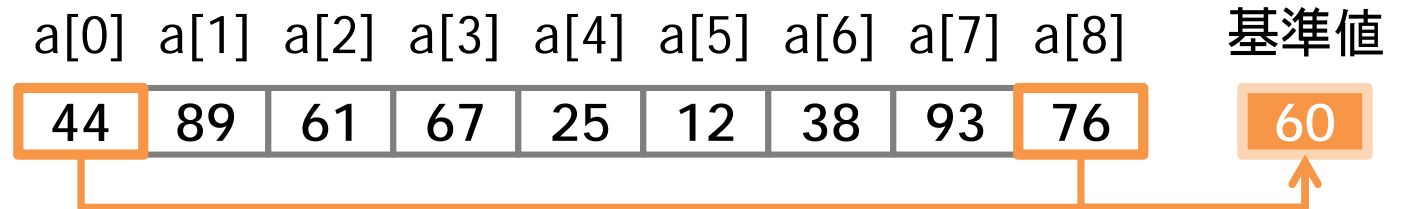
クイックソートの概要



クイックソートの手順 (1回の分割の手順)

1. 基準値の決定

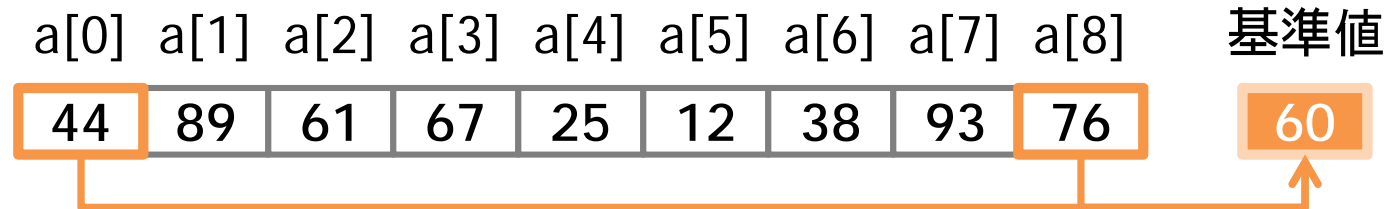
- 最初と最後の平均値など



クイックソートの手順 (1回の分割の手順)

1. 基準値の決定

- 最初と最後の平均値など



2. データの探索と入れ替え

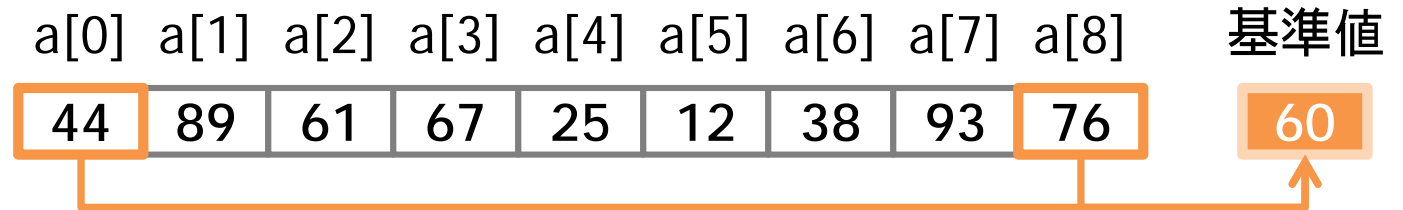
- 前方からは基準値より大きな数
- 後方からは基準値より小さな数
- 見つかったらデータの入れ替え



クイックソートの手順 (1回の分割の手順)

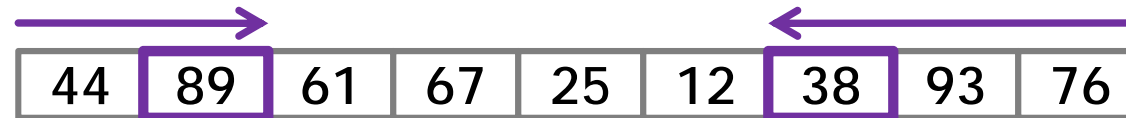
1. 基準値の決定

- 最初と最後の平均値など



2. データの探索と入れ替え

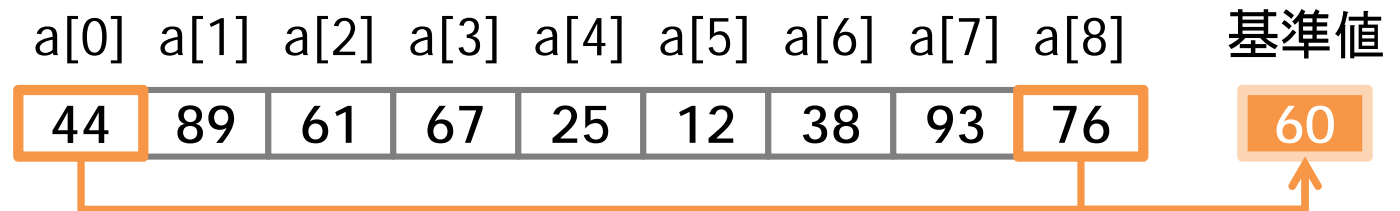
- 前方からは基準値より大きな数
- 後方からは基準値より小さな数
- 見つかったらデータの入れ替え



クイックソートの手順 (1回の分割の手順)

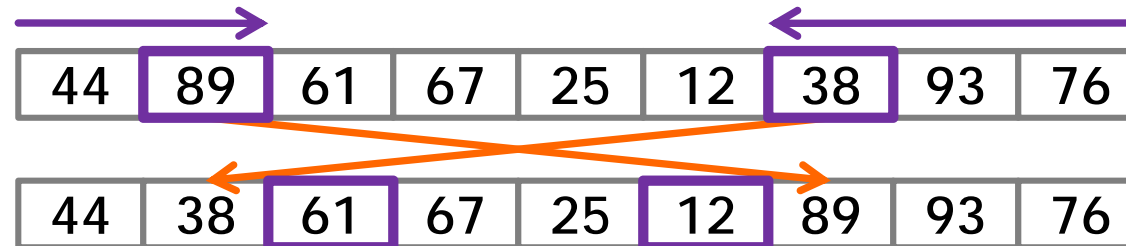
1. 基準値の決定

- 最初と最後の平均値など



2. データの探索と入れ替え

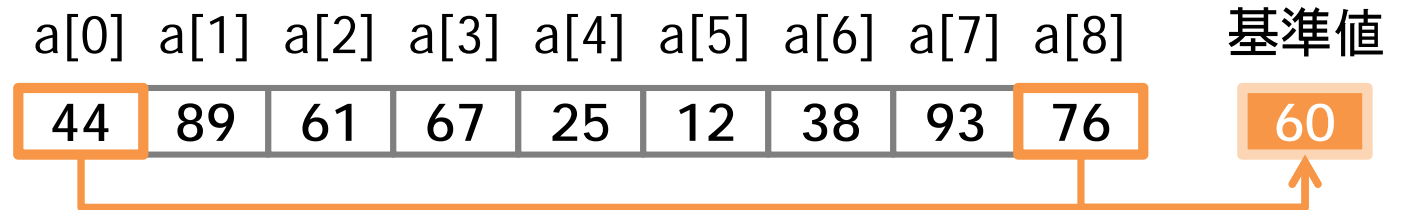
- 前方からは基準値より大きな数
- 後方からは基準値より小さな数
- 見つかったらデータの入れ替え



クイックソートの手順 (1回の分割の手順)

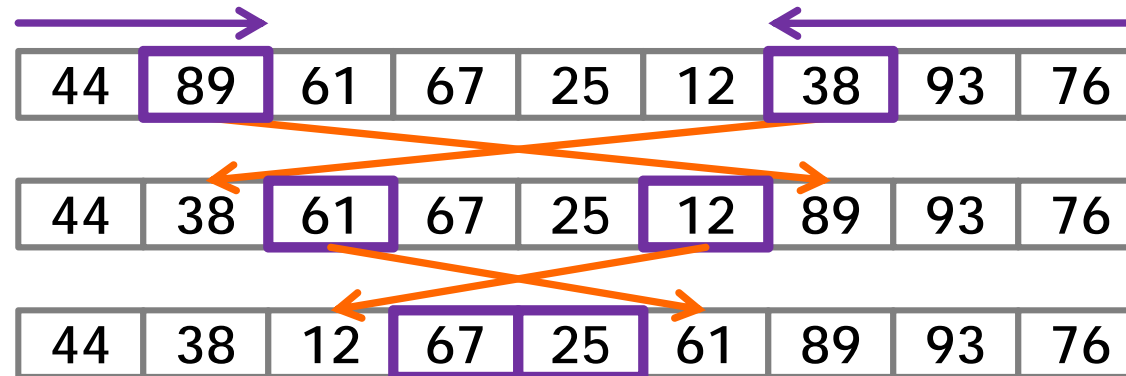
1. 基準値の決定

- 最初と最後の平均値など



2. データの探索と入れ替え

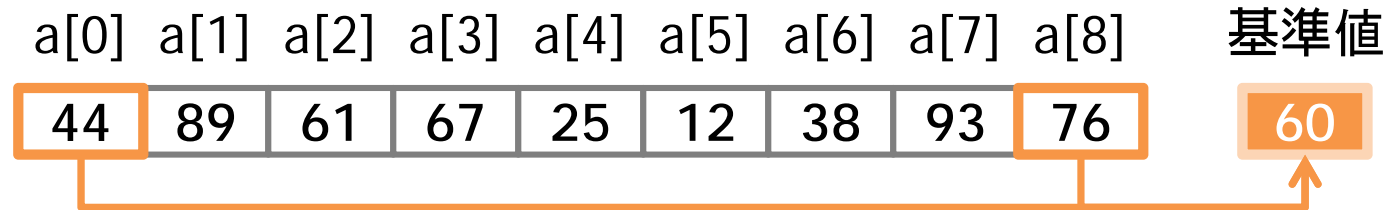
- 前方からは基準値より大きな数
- 後方からは基準値より小さな数
- 見つかったらデータの入れ替え



クイックソートの手順 (1回の分割の手順)

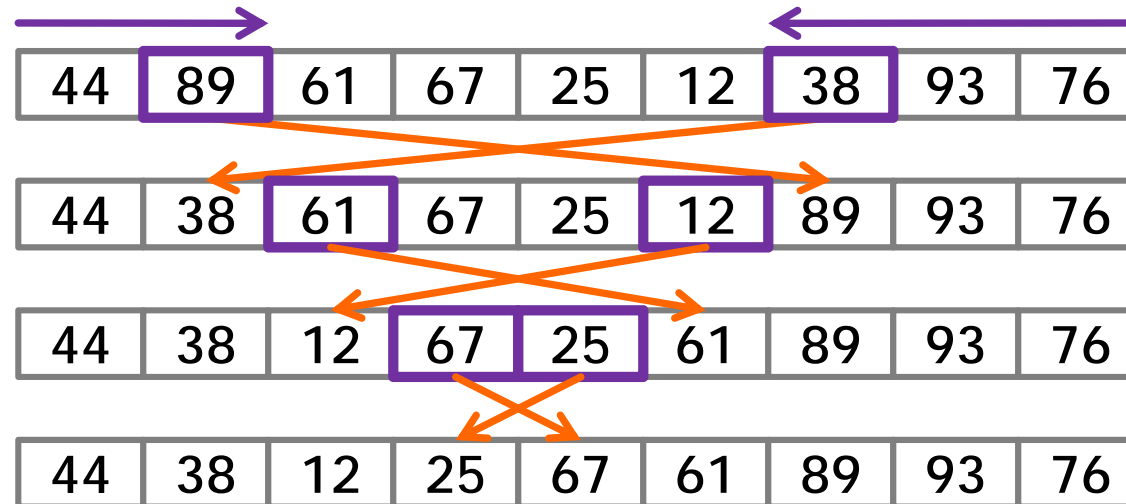
1. 基準値の決定

- 最初と最後の平均値など



2. データの探索と入れ替え

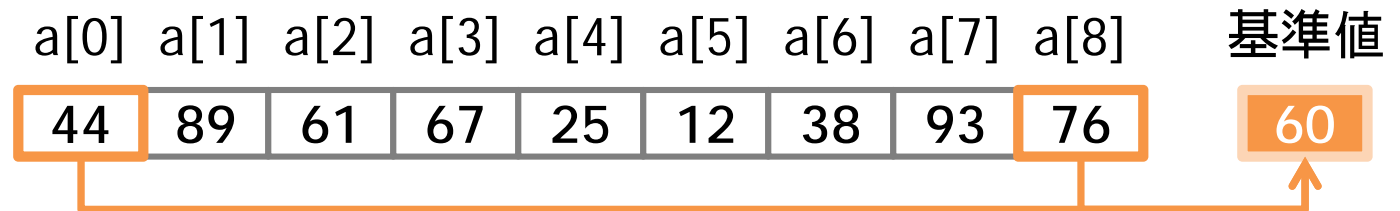
- 前方からは基準値より大きな数
- 後方からは基準値より小さな数
- 見つかったらデータの入れ替え
- ぶつかれば終了



クイックソートの手順 (1回の分割の手順)

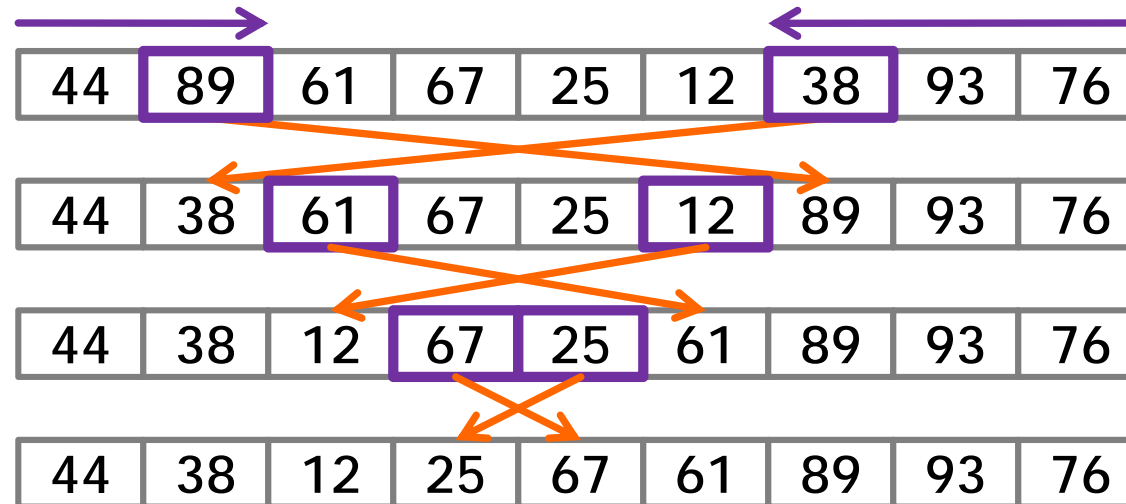
1. 基準値の決定

- 最初と最後の平均値など



2. データの探索と入れ替え

- 前方からは基準値より大きな数
- 後方からは基準値より小さな数
- 見つかったらデータの入れ替え
- ぶつかれば終了

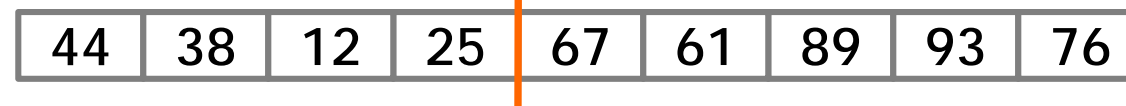


3. グループ分割

- 基準値より大きいグループ、小さいグループに分類
- グループごとに手順1, 2を繰り返す

基準値より小さいグループ

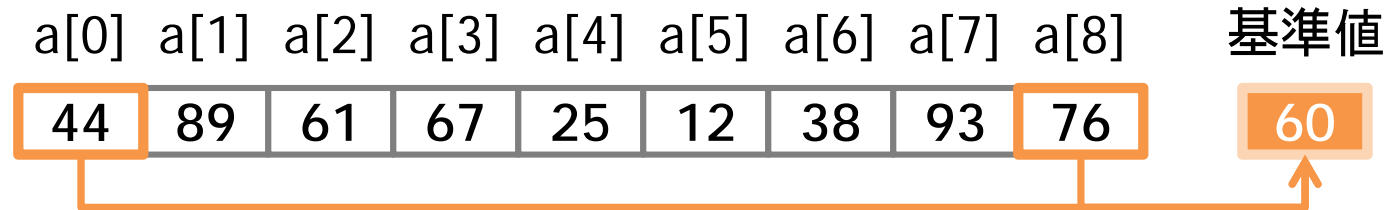
基準値より大きいグループ



クイックソートの手順 (1回の分割の手順)

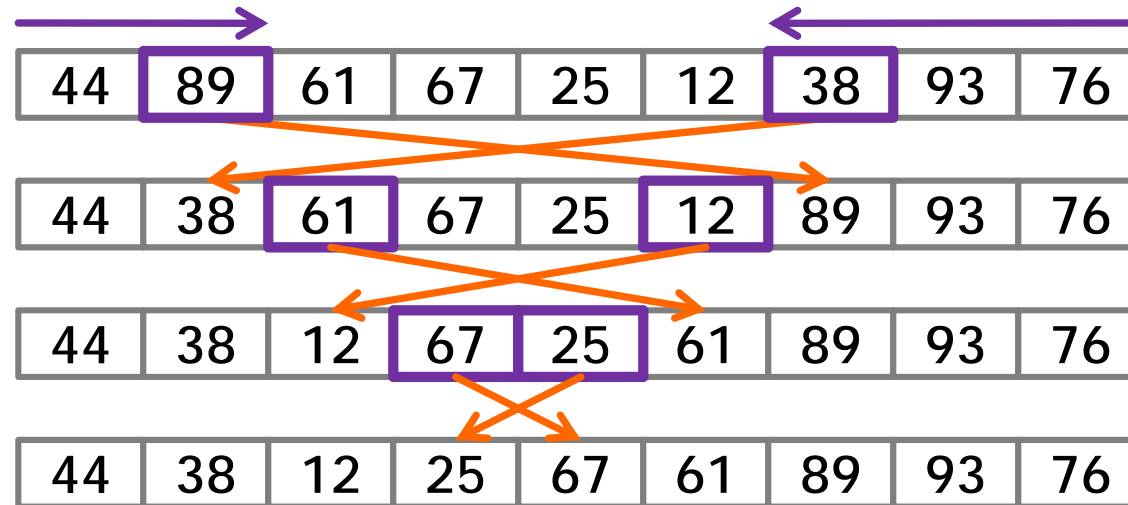
1. 基準値の決定

- 最初と最後の平均値など



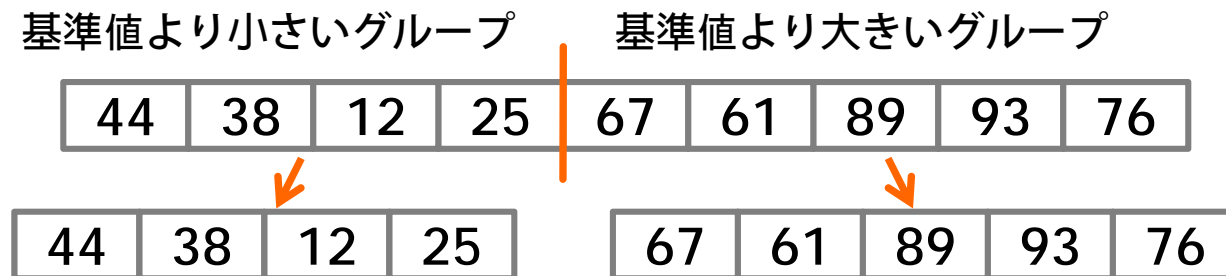
2. データの探索と入れ替え

- 前方からは基準値より大きな数
- 後方からは基準値より小さな数
- 見つかったらデータの入れ替え
- ぶつかれば終了

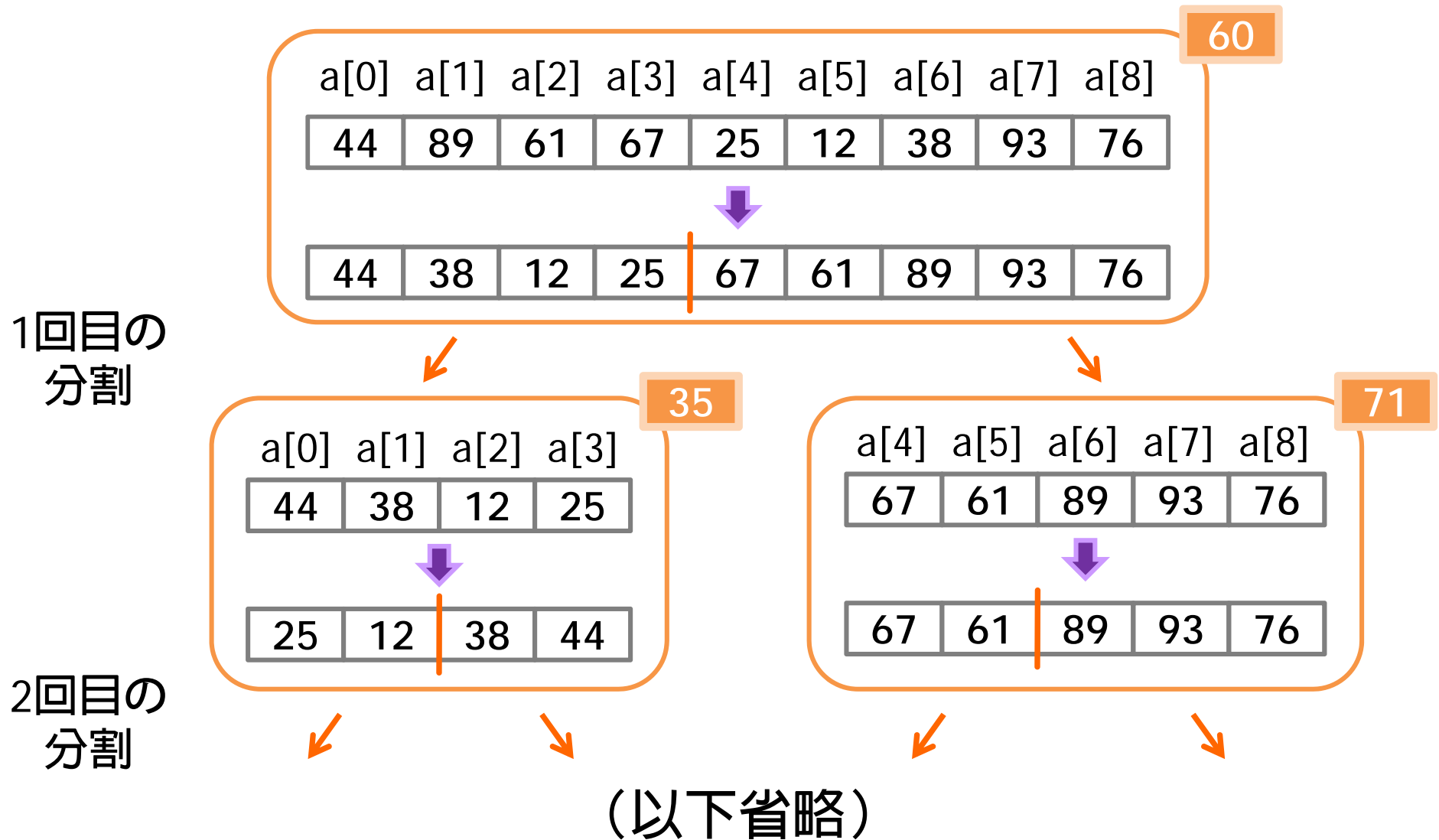


3. グループ分割

- 基準値より大きいグループ、小さいグループに分類
- グループごとに手順1, 2を繰り返す



クイックソート



プログラムに必要な変数

- 基準値 : pivot
- データ範囲 : left, right
- カーソル位置 : pl, pr

基準値

60

pivot

left

pl

pr

right

44	38	61	67	25	12	89	93	76
----	----	----	----	----	----	----	----	----

クイックソートプログラム

```
#include <stdio.h>
#define NUM 10    // データ数
void quick_sort( int a[], int left, int right );
void swap( int*, int* );

void main()
{
    // 入力データ
    int a[NUM] = { 44, 89, 61, ... , 93, 76 };

    // クイックソート
    //quick_sort( a, 0, NUM-1 );
}

void swap( int *a, int *b )
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
void quick_sort( int a[], int left, int right )
{
    int pl, pr, pivot;
    pl = left;
    pr = right;
    pivot = (a[pl]+a[pr])/2;    // 基準値

    do{
        while( a[pl] < pivot ){ pl++; }    // 左カーソル
        while( a[pr] > pivot ){ pr--; }    // 右カーソル
        if( pl <= pr ){
            swap( &a[pl], &a[pr] );
            pl++;
            pr--;
        }
    } while( pl <= pr );

    if( left < pr )
        quick_sort( a, left, pr );    // 再帰呼び出し
    if( pl < right )
        quick_sort( a, pl, right );    // 再帰呼び出し
}
```


基準値の選択方法

- クイックソートでは基準値の取り方によって比較回数が大きく異なる
- 理想的な基準値
 - ソート後に中央値(メディアン)になる値
 - 中央値を求める操作が必要 → 余分な計算が必要
- 中央値に近い値になる可能性の高い値を使用
 - 例1: データ列の最初と最後の平均値
 - 例2: データ列の中央の値
 - 例3: データ列の先頭, 中央, 末尾の値の中央値

乱数発生関数randの使用方法

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    int a,b,n,same,i;
    double ans;
    srand(time(NULL)); // 乱数発生初期化（同じ乱数が発生しないようにする）

    printf("Number of trials: ");
    scanf("%d", &n);

    same=0;
    for ( i=0;i<n;i++) {
        a=rand()%6+1; // 乱数発生1回目 1～6の値が出てくる
        b=rand()%6+1; // 乱数発生2回目
        if (a==b)
            same++; // 同じ目が出た場合sameを増やす
    }
    ans=(double)same/(double)n;
    printf("probability of same number = %f ¥n",ans);
    return(0);
}
```

処理時間計測

- NUM個の乱数を発生させ、それをソートするプログラムを複数回実行する
- バブルソートとクイックソートで計算時間を比較する

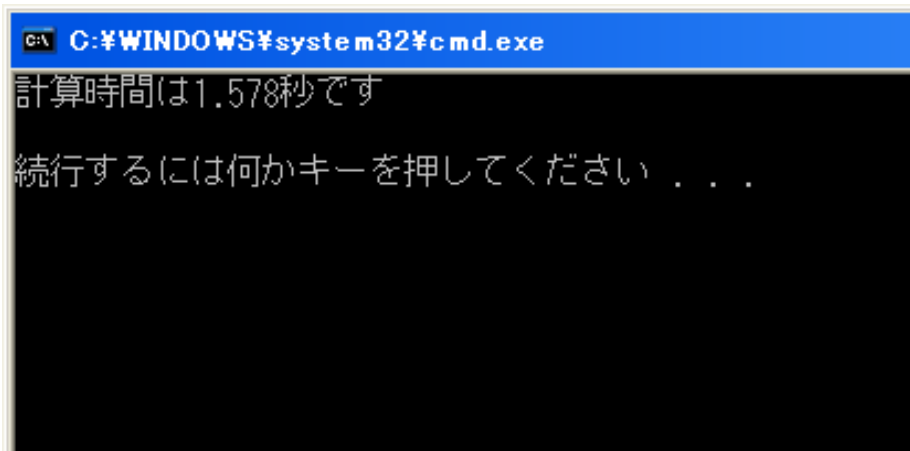
処理時間計測

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

#define NUM 100000 // データ数
#define iter 100    // ソートの繰り返し回数

void bubble_sort( int a[] );
void quick_sort( int a[], int left, int right );
void swap( int*, int* );
```

実行結果



The screenshot shows a Windows command prompt window with the title bar "C:\> C:\WINDOWS\system32\cmd.exe". The command prompt displays the following text:

```
計算時間は1.578秒です
続行するには何かキーを押してください . . .
```

```
void main()
{
    int i, j, a[NUM];
    time_t start, end;

    start = clock(); // 開始時間
    for( j=0; j<iter; j++ ){

        // 乱数発生 ( 0からRAND_MAXまで )
        for( i=0; i<NUM; i++ )
            a[i] = rand();

        // クイックソート
        quick_sort( a, 0, NUM-1 );
    }

    end = clock(); // 終了時間

    // 計算時間
    printf("計算時間は%.3f秒です\n\n",
        float(end-start)/CLOCKS_PER_SEC);
}
```

処理時間計測結果

- ・ 繰り返し回数はすべて100回

データ数	100	1,000	10,000	100,000
バブル ソート	0.000	0.296	26.046	2613.734
クイック ソート	0.000	0.015	0.156	1.578

(単位：秒)

整列処理(クイックソート)のまとめ

- 高速なソートアルゴリズムの一つ
 - 平均的には最も早いソート法
- 整列処理を行う範囲を次第に小さくする
 - ピボットの設定方法によって効率が異なる
- 再帰処理を使用することによって、効率的で短いソースコードを実現
- データ列の分割方法について理解すること
 - 左カーソル, 右カーソルの動き
 - ピボットを挟んだデータの交換方法
 - 分割後の領域範囲

バブルソートプログラム (参考資料)

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

```
#define NUM 5
```

```
void swap( int *a, int *b )
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
// バブルソート
```

```
int main(void)
{
    int i, j, k, count ;

    // データ入力
    int a[5] = { 60, 75, 70, 56, 52 };

    printf(“ 整列前データ:”);
    for( i=0;i<NUM;i++){
        printf( “%4d”, a[i] );
    }
}
```

```
// バブルソート
```

```
count = 0;
for( i=NUM-1;i>=0;i-- ){
    for( j=0;j<i;j++ ){
```

```
// 比較回数カウント
```

```
count ++;
```

```
// 条件を満たすならば交換
```

```
if( a[j] > a[j+1] ){
    swap( &a[j], &a[j+1] );
}
```

```
// 検証のための表示
```

```
printf( “%2d回目の処理後:”, count
);
for( k=0;k<NUM;k++){
    printf( “%4d”, a[k] );
}
}
```

挿入法プログラム (参考資料)

```
#include <stdio.h>

#define NUM 5

// 挿入法ソート
int main(void)
{
    int i, j, k, tmp;

    // データ入力
    int a[5] = { 60, 75, 70, 56, 52 };

    printf("  整列前データ:");
    for( i=0;i<NUM;i++){
        printf( "%4d", a[i] );
    }
}
```

```
// 挿入法ソート
for( i=1;i<NUM;i++){

    // 検証のための表示
    printf( "%2d回目の処理後:", i );
    for( k=0;k<i;k++){
        printf( "%4d", a[k] );
    }

    tmp = a[i];
    for( j=i-1;j>=0;j-- ){
        if( a[j]>tmp )
            a[j+1] = a[j];
        else
            break;
    }
    a[j+1] = tmp;
}
}
```