1. 問題

　データの並べ替えを行うアルゴリズムを実装し、実データを用いて比較検討する。

データの種類は以下の2種を扱う。

・負の数を含まないint型整数

・負の数を含まないfloat型実数

これらをそれぞれ疑似乱数で100,500,1000,5000,10000個発生させ、並べ替えを行った際のデータの比較回数、代入回数および処理にかかった時間を計測する。

並べ替えを行うアルゴリズムは、既存のソート法3つと独自のアルゴリズムを1つ使用する。既存のソート法は以下の3つを採用した。

・バブルソート

・選択法

・クイックソート

　また、独自のアルゴリズムは選択法を改良したものとなっている。

　乱数の発生にはrand関数を使用し、乱数の初期化ではsrand関数に引数として自分の学籍番号を指定したものを使用する。

1. プログラム構成図

　　プログラムファイルは図1に示すものを作成した。

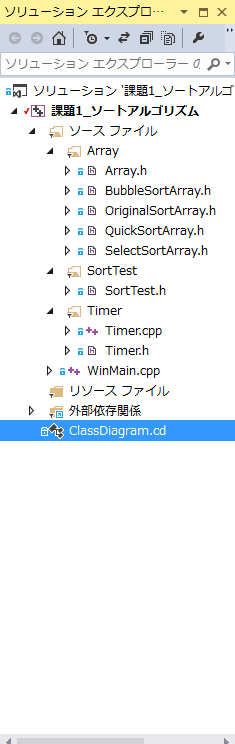


図1：ソリューションエクスプローラー

具体的なソートを実行する配列を扱うクラスを定義したBubbleSortArray.h、OriginalSortArray.h、QuickSortArray.h、SelectSortArray.hの４つが基になるArray.hを継承しており、それら4つのクラスと時間計測を行うクラスを定義したTimerクラスを、SortTestが持っている構造になっている。

SortTestがソートの時間計測を行い、内部でコンソール表示させる仕組みである。

なお、配列のクラスとそれらを扱うSortTestの一部の関数は複数の型を扱いやすくするためテンプレートにしており、宣言と定義をヘッダーファイルにまとめてある。

1. ソースプログラム

作成したソースプログラムを以下に示す。

// WinMain.cpp

// メイン関数を記述

#include <iostream>

#include "BubbleSortArray.h"

#include "SelectSortArray.h"

#include "QuickSortArray.h"

#include "OriginalSortArray.h"

#include "SortTest.h"

using namespace std;

int main()

{

// テスターを宣言

CSortTest tester;

cout << "########バブルソート<int>########" << endl;

tester.BubbleSortTest<int>( 100 );

tester.BubbleSortTest<int>( 500 );

tester.BubbleSortTest<int>( 1000 );

tester.BubbleSortTest<int>( 5000 );

tester.BubbleSortTest<int>( 10000 );

cout << "########バブルソート<float>########" << endl;

tester.BubbleSortTest<float>( 100 );

tester.BubbleSortTest<float>( 500 );

tester.BubbleSortTest<float>( 1000 );

tester.BubbleSortTest<float>( 5000 );

tester.BubbleSortTest<float>( 10000 );

cout << "########選択法<int>########" << endl;

tester.SelectSortTest<int>( 100 );

tester.SelectSortTest<int>( 500 );

tester.SelectSortTest<int>( 1000 );

tester.SelectSortTest<int>( 5000 );

tester.SelectSortTest<int>( 10000 );

cout << "########選択法<float>########" << endl;

tester.SelectSortTest<float>( 100 );

tester.SelectSortTest<float>( 500 );

tester.SelectSortTest<float>( 1000 );

tester.SelectSortTest<float>( 5000 );

tester.SelectSortTest<float>( 10000 );

cout << "########クイックソート<int>########" << endl;

tester.QuickSortTest<int>( 100 );

tester.QuickSortTest<int>( 500 );

tester.QuickSortTest<int>( 1000 );

tester.QuickSortTest<int>( 5000 );

tester.QuickSortTest<int>( 10000 );

cout << "########クイックソート<float>########" << endl;

tester.QuickSortTest<float>( 100 );

tester.QuickSortTest<float>( 500 );

tester.QuickSortTest<float>( 1000 );

tester.QuickSortTest<float>( 5000 );

tester.QuickSortTest<float>( 10000 );

cout << "########オリジナルソート<int>########" << endl;

tester.OriginalSortTest<int>( 100 );

tester.OriginalSortTest<int>( 500 );

tester.OriginalSortTest<int>( 1000 );

tester.OriginalSortTest<int>( 5000 );

tester.OriginalSortTest<int>( 10000 );

cout << "########オリジナルソート<float>########" << endl;

tester.OriginalSortTest<float>( 100 );

tester.OriginalSortTest<float>( 500 );

tester.OriginalSortTest<float>( 1000 );

tester.OriginalSortTest<float>( 5000 );

tester.OriginalSortTest<float>( 10000 );

}

// SortTest.h

// ソート検証を行うクラス

#ifndef \_SORTTEST\_H

#define \_SORTTEST\_H

#include "Array.h"

#include "Timer.h"

#include <fstream>

class CSortTest

{

private:

CTimer timer; // タイマー

LONGLONG count; // 平均カウント数

LONGLONG time; // 平均時間

LONGLONG minCount, maxCount; // 最大カウント数、最小カウント数

LONGLONG minTime, maxTime; // 最大時間、最小時間

// テスト実行

// ary：ソートを行う配列

// arrayNum：配列の要素数

// testNum：試行回数

template <class TYPE>

void Test( CArray<TYPE>\* ary, const int arrayNum, const int testNum )

{

ary -> CompSort(); // 比較回数と代入回数を計測

for( int i = 0; i < testNum; i++ ){

ary -> Init( arrayNum ); // 配列初期化

timer.Begin(); // 計測開始

ary -> Sort(); // ソート実行

timer.End(); // 計測終了

// このあと何度もtimerの変数を使うので一時変数に保存しておく

LONGLONG tCount = timer.GetCount();

LONGLONG tTime = timer.GetTime();

if( i == 0 ){

// 1回目の結果を最大・最小の値として保存しておく

maxCount = minCount = tCount;

maxTime = minTime = tTime;

}else{

// 2回目以降は最大か最小かを検証して、変数を更新

if( maxCount < tCount ) maxCount = tCount;

else if( minCount > tCount ) minCount = tCount;

if( maxTime < tTime ) maxTime = tTime;

else if( minTime > tTime ) minTime = tTime;

}

count += tCount; // カウント数を加算

time += tTime; // 時間を加算

}

count /= testNum; // 平均カウント数を計算

time /= testNum; // 平均時間を計算

// 結果表示

cout << "============結果============" << endl;

cout << arrayNum << "個の要素数で" << testNum << "回試行しました。" << endl;

cout << "平均時間：" << time << "ミリ秒" << endl;

cout << "平均カウント数：" << count << "カウント" << endl;

cout << "比較回数：" << ary -> GetCompareCount() << "回" << endl;

cout << "代入回数：" << ary -> GetSubstituteCount() << "回" << endl;

cout << "--------------------------------" << endl;

cout << "最大時間：" << maxTime << "ミリ秒" << endl;

cout << "最小時間：" << minTime << "ミリ秒" << endl;

cout << "最大カウント数：" << maxCount << "カウント" << endl;

cout << "最小カウント数：" << minCount << "カウント" << endl << endl;

}

public:

// コンストラクタ

CSortTest(){

// 変数を0で初期化

time = 0; count = 0;

minCount = 0; maxCount = 0;

minTime = 0; maxTime = 0;

}

// 平均カウント数取得

LONGLONG GetCount(){ return count; }

// 平均時間取得

LONGLONG GetTime(){ return time; }

// main関数で呼び出す関数

// ここで試行回数を100回に固定

// バブルソート

template <class TYPE>

void BubbleSortTest( const int arrayNum )

{

// ソートを行う配列クラスを宣言

CArray<TYPE>\* ary = new CBubbleSortArray<TYPE>( arrayNum );

Test( ary, arrayNum, 100 ); // テスト実行

delete[] ary; // 後片付け

}

// 選択法ソート

template <class TYPE>

void SelectSortTest( const int arrayNum )

{

// ソートを行う配列クラスを宣言

CArray<TYPE>\* ary = new CSelectSortArray<TYPE>( arrayNum );

Test( ary, arrayNum, 100 ); // テスト実行

delete[] ary; // 後片付け

}

// クイックソート

template <class TYPE>

void QuickSortTest( const int arrayNum )

{

// ソートを行う配列クラスを宣言

CArray<TYPE>\* ary = new CQuickSortArray<TYPE>( arrayNum );

Test( ary, arrayNum, 100 ); // テスト実行

delete[] ary; // 後片付け

}

// オリジナルソート

template <class TYPE>

void OriginalSortTest( const int arrayNum )

{

// ソートを行う配列クラスを宣言

CArray<TYPE>\* ary = new COriginalSortArray<TYPE>( arrayNum );

Test( ary, arrayNum, 100 ); // テスト実行

delete[] ary; // 後片付け

}

};

#endif

// Timer.h

// 時間計測を行うクラス

#ifndef \_TIMER\_H

#define \_TIMER\_H

#include <Windows.h>

#include <stdlib.h>

class CTimer{

private:

LARGE\_INTEGER m\_freq; // 周波数

LARGE\_INTEGER m\_before; // 計測開始時刻

LARGE\_INTEGER m\_after; // 計測終了時刻

LONGLONG m\_time; // 結果(ミリ秒単位)

LONGLONG m\_count; // 結果(カウント数)

public:

// コンストラクタ

CTimer();

// 計測開始

void Begin(){ QueryPerformanceCounter( &m\_before ); }

// 計測終了

void End(){

QueryPerformanceCounter( &m\_after );

// 経過時間を計算

m\_count = ( m\_after.QuadPart - m\_before.QuadPart );

m\_time = m\_count \* 1000 / m\_freq.QuadPart ;

}

// カウント数取得

LONGLONG GetCount(){ return m\_count; }

// 時間結果取得

LONGLONG GetTime(){ return m\_time; }

// コンソールに表示

void Disp();

};

#endif

// Timer.cpp

// 時間計測を行うクラス

#include "Timer.h"

#include <iostream>

using namespace std;

// コンストラクタ

CTimer::CTimer()

{

// 変数を初期化

memset( &m\_freq, 0x00, sizeof( LARGE\_INTEGER ) );

memset( &m\_before, 0x00, sizeof( LARGE\_INTEGER ) );

memset( &m\_after, 0x00, sizeof( LARGE\_INTEGER ) );

m\_time = 0;

m\_count = 0;

// CPUの周波数を取得

QueryPerformanceFrequency( &m\_freq );

}

// コンソールに表示

void CTimer::Disp(){

cout << "================================" << endl;

cout << "経過時間" << endl;

cout << "================================" << endl;

cout << m\_count << "カウント" << endl;

cout << m\_time << "ミリ秒" << endl;

}

// Array.h

// さまざまな型の配列を扱うクラス宣言

#ifndef \_ARRAY\_H

#define \_ARRAY\_H

#include <iostream>

#include <stdlib.h>

template <class TYPE>

class CArray

{

protected:

TYPE\* m\_Array; // 配列

int m\_iArrayNum; // 配列の要素数

int compareCount; // 比較回数

int substituteCount; // 代入回数

private:

// 初期化

void Init();

// 解放

void Release();

// インデックスチェック

void CheckIndex( const int index ) const;

public:

// コンストラクタ

explicit CArray( const int num );

// コピーコンストラクタ

CArray( const CArray <TYPE> & rother );

// デストラクタ

virtual ~CArray();

// 配列に数値をランダムで入れる初期化

// val : 配列の要素数

void Init( const int val );

// ソート実行

virtual void Sort(){};

// 比較回数、代入回数を測るためのソート

virtual void CompSort(){};

// 要素へのアクセス

TYPE Get( const int index ) const;

void Set( const int index, const TYPE value );

// メンバの参照

TYPE& operator []( unsigned int index );

TYPE& operator []( int index );

TYPE operator []( unsigned int index ) const;

TYPE operator []( int index ) const;

// 配列の直接参照

operator const TYPE\* () const;

// 配列が有効かどうか

bool IsValid() const;

// 配列の要素数を取得

int ArrayNum() const;

// 配列のサイズを取得

int ArraySize() const;

// 比較回数を取得

int GetCompareCount() const;

// 代入回数を取得

int GetSubstituteCount() const;

// 配列のコピー

bool Copy( const CArray <TYPE> & rother );

// =演算子のオーバーロード

CArray <TYPE> & operator =( const CArray <TYPE> & rother );

// 要素の入れ替え

void Swap( TYPE& val1, TYPE& val2 );

// 要素を表示

void Disp();

};

// コンストラクタ

template <class TYPE>

CArray <TYPE> ::CArray( const int num )

{

compareCount = 0;

substituteCount = 0;

m\_Array = new TYPE[num];

if( m\_Array == NULL )

m\_iArrayNum = 0;

else

m\_iArrayNum = num;

}

CArray <int> ::CArray( const int num )

{

compareCount = 0;

substituteCount = 0;

m\_Array = new int[num];

if( m\_Array == NULL )

m\_iArrayNum = 0;

else

m\_iArrayNum = num;

srand( 1423056 );

for( int i = 0; i < m\_iArrayNum; i++)

m\_Array[i] = rand();

}

CArray <float> ::CArray( const int num )

{

compareCount = 0;

substituteCount = 0;

m\_Array = new float[num];

if( m\_Array == NULL )

m\_iArrayNum = 0;

else

m\_iArrayNum = num;

srand( 1423056 );

for( int i = 0; i < m\_iArrayNum; i++ )

m\_Array[i] = (float)rand() / RAND\_MAX;

}

// コピーコンストラクタ

template <class TYPE>

CArray <TYPE> ::CArray( const CArray <TYPE> & rother )

{

Init();

Copy( rother );

}

// デストラクタ

template <class TYPE>

CArray <TYPE> ::~CArray()

{

Release();

}

// 配列にランダムに数値を入れる初期化

template <>

void CArray <int> ::Init( const int val )

{

srand( 1423056 );

for( int i = 0; i < val; i++)

m\_Array[i] = rand();

}

template <>

void CArray <float> ::Init( const int val )

{

srand( 1423056 );

for( int i = 0; i < val; i++ )

m\_Array[i] = (float)rand() / RAND\_MAX;

}

// メンバの参照

template <class TYPE>

TYPE& CArray <TYPE> ::operator []( unsigned int index )

{

CheckIndex( index );

return m\_Array[index];

}

template <class TYPE>

inline TYPE& CArray <TYPE> ::operator []( int index )

{

CheckIndex( index );

return (\*this)[(unsigned int)index];

}

template <class TYPE>

TYPE CArray <TYPE> ::operator []( unsigned int index ) const

{

CheckIndex( index );

return m\_Array[index];

}

template <class TYPE>

inline TYPE CArray <TYPE> ::operator []( int index ) const

{

CheckIndex( index );

return (\*this)[(unsigned int)index];

}

// 配列の直接参照

template <class TYPE>

inline CArray <TYPE> ::operator const TYPE\*() const

{

return m\_Array;

}

// 要素を取得

template <class TYPE>

TYPE CArray <TYPE> ::Get( const int index ) const

{

CheckIndex( index );

return m\_Array[index];

}

// 要素に数を代入

template <class TYPE>

void CArray <TYPE> ::Set( const int index, const TYPE value )

{

CheckIndex( index );

m\_Array[index] = value;

}

// 配列のコピー

template <class TYPE>

bool CArray <TYPE> ::Copy( const CArray& rother )

{

// 自身はコピーしない

if( this == &rother )

return true;

Release(); // 配列が確保されているときはメモリ解放

if( rother.IsValid() == true ){

m\_Array = new TYPE[rother.m\_iArrayNum];

if( m\_Array == NULL ){

m\_iArrayNum = 0;

return false;

}

m\_iArrayNum = rother.m\_iArrayNum;

for( int i = 0; i < m\_iArrayNum; i++)

m\_Array[i] = rother [i];

}

return true;

}

// ＝演算子のオーバーロード

template <class TYPE>

CArray <TYPE> & CArray <TYPE> ::operator =( const CArray& rother )

{

Copy( rother );

return \*this;

}

// インデックスチェック

template <class TYPE>

void CArray <TYPE> ::CheckIndex( const int index ) const

{

if( (unsigned int)index < (unsigned int)m\_iArrayNum )

return;

cout << "インデックスが不正です" << endl

<< "値：" << index << endl;

exit( 1 );

}

// 配列が有効かどうか

template <class TYPE>

inline bool CArray <TYPE> ::IsValid() const

{

return m\_Array != NULL;

}

// 配列の要素数を取得

template <class TYPE>

inline int CArray <TYPE> ::ArrayNum() const

{

return m\_iArrayNum;

}

// 配列のサイズを取得

template <class TYPE>

inline int CArray <TYPE> ::ArraySize() const

{

return m\_iArrayNum \* sizeof \*m\_Array;

}

// 比較回数を取得

template <class TYPE>

inline int CArray <TYPE> ::GetCompareCount() const

{

return compareCount;

}

// 代入回数を取得

template <class TYPE>

inline int CArray <TYPE> ::GetSubstituteCount() const

{

return substituteCount;

}

// 初期化

template <class TYPE>

void CArray <TYPE> ::Init()

{

m\_Array = NULL;

m\_iArrayNum = 0;

compareCount = 0;

substituteCount = 0;

}

// 解放

template <class TYPE>

void CArray <TYPE> ::Release()

{

if( IsValid() == true ){

delete[] m\_Array;

Init();

}

};

// 要素の入れ替え

template <class TYPE>

void CArray <TYPE> ::Swap( TYPE& val1, TYPE& val2 )

{

TYPE temp = val1;

val1 = val2;

val2 = temp;

}

// 要素を表示

template <class TYPE>

void CArray <TYPE> ::Disp()

{

for( int i = 0; i < m\_iArrayNum; i++ ){

cout << i << " : " << m\_Array[i] << endl;

}

}

#endif

// BubbleSortArray.h

// バブルソートを実行する配列を扱うクラス

#ifndef \_BUBBLESORT\_ARRAY\_H

#define \_BUBBLESORT\_ARRAY\_H

#include "Array.h"

template <class TYPE>

class CBubbleSortArray : public CArray <TYPE>

{

public:

// コンストラクタ

CBubbleSortArray( const int num );

// バブルソート実行

void Sort();

// 比較回数、代入回数を測るためのソート

void CompSort();

};

// コンストラクタ

template <class TYPE>

CBubbleSortArray <TYPE> ::CBubbleSortArray( const int val ) : CArray( val )

{}

// バブルソート実行

template <class TYPE>

void CBubbleSortArray <TYPE> ::Sort()

{

for( int i = 0; i < m\_iArrayNum - 1; i++ ){

for( int j = m\_iArrayNum - 1; j > 0; j-- ){ // 下から順に検索

if( m\_Array[j] < m\_Array[j - 1] ) // １つ前のほうが大きければ

Swap( m\_Array[j], m\_Array[j - 1] ); // 入れ替え

}

}

}

template <class TYPE>

void CBubbleSortArray <TYPE> ::CompSort()

{

for( int i = 0; i < m\_iArrayNum - 1; i++ ){

for( int j = m\_iArrayNum - 1; j > 0; j-- ){ // 下から順に検索

if( m\_Array[j] < m\_Array[j - 1] ){ // １つ前のほうが大きければ

Swap( m\_Array[j], m\_Array[j - 1] ); // 入れ替え

substituteCount++; // 代入回数＋1

}

compareCount++; // 比較回数＋1

}

}

}

#endif

// SelectSortArray.h

// 選択法を行う配列を扱うクラス

#ifndef \_SELECTSORT\_ARRAY\_H

#define \_SELECTSORT\_ARRAY\_H

#include "Array.h"

template <class TYPE>

class CSelectSortArray : public CArray <TYPE>

{

private:

int iMin; // 最小の要素のインデックス

TYPE nMin; // 現在の最小データ

public:

// コンストラクタ

CSelectSortArray( const int val );

// ソート実行

void Sort();

// 比較回数と代入回数を測るためのソート

void CompSort();

};

// コンストラクタ

template <class TYPE>

CSelectSortArray <TYPE> ::CSelectSortArray( const int val ) : CArray( val )

{

iMin = 0;

nMin = 0;

}

// 選択法ソート実行

template <class TYPE>

void CSelectSortArray <TYPE> ::Sort()

{

// 要素数-1回ループ

for( int i = 0; i < m\_iArrayNum - 1; i++ ){

// 初期値代入

iMin = i;

nMin = m\_Array[i];

// 最小要素を検索

for( int j = i + 1; j < m\_iArrayNum; j++ ){

if( m\_Array[j] < nMin ){ // より小さい要素が見つかったら

iMin = j; // 要素番号を覚えておく

nMin = m\_Array[j]; // 最小データ更新

}

}

// 入れ替え

Swap( m\_Array[iMin], m\_Array[i] );

}

}

template <class TYPE>

void CSelectSortArray <TYPE> ::CompSort()

{

// 要素数-1回ループ

for( int i = 0; i < m\_iArrayNum - 1; i++ ){

// 初期値代入

iMin = i;

nMin = m\_Array[i];

// 最小要素を検索

for( int j = i + 1; j < m\_iArrayNum; j++ ){

if( m\_Array[j] < nMin ){ // より小さい要素が見つかったら

iMin = j; // 要素番号を覚えておく

nMin = m\_Array[j]; // 最小データ更新

}

compareCount++; // 比較回数＋1

}

// 入れ替え

Swap( m\_Array[iMin], m\_Array[i] );

substituteCount++; // 代入回数＋１

}

}

#endif

// QuickSortArray.h

// クイックソートを行う配列を扱うクラス

#ifndef \_QUICKSORT\_ARRAY\_H

#define \_QUICKSORT\_ARRAY\_H

#include "Array.h"

template <class TYPE>

class CQuickSortArray : public CArray <TYPE>

{

private:

// 再帰関数

void SortSub( const int start, const int end );

// 比較回数と代入回数を測るための再帰関数

void CompSortSub( const int start, const int end );

public:

// コンストラクタ

CQuickSortArray( const int val );

// クイックソート実行

void Sort();

// 比較回数と代入回数を測るためのソート

void CompSort();

};

// コンストラクタ

template <class TYPE>

CQuickSortArray <TYPE> ::CQuickSortArray( const int num ) : CArray( num )

{}

// クイックソート実行

template <class TYPE>

void CQuickSortArray <TYPE> ::Sort()

{

SortSub( 0, m\_iArrayNum - 1 );

}

template <class TYPE>

void CQuickSortArray <TYPE> ::CompSort()

{

CompSortSub( 0, m\_iArrayNum - 1 );

}

// 再帰関数

template <class TYPE>

void CQuickSortArray <TYPE> ::SortSub( const int start, const int end )

{

if( start >= end ) return;

int iLess = start; // 前半のインデックス

int iGreater = end; // 後半のインデックス

TYPE temp = ( m\_Array[iLess] + m\_Array[iGreater] ) / 2; // 基準値は両端の平均

// 交換ループ

while( iLess < iGreater )

{

// tempより小さい要素をスルー

while( m\_Array[iLess] < temp ) iLess++;

// tempより大きい要素をスルー

while( m\_Array[iGreater] > temp ) iGreater--;

// 検索位置が交わったら交換し、iLessとiGreaterを進める

if( iLess < iGreater )

Swap( m\_Array[iLess++], m\_Array[iGreater--] );

}

// iLess == iGreater で終わった時は

// iGreater < iLessとなるようにする

if( iLess == iGreater )

{

if( m\_Array[iLess] < temp )

iLess++;

else if( m\_Array[iLess] > temp )

iGreater--;

}

// 次のブロックのサイズが2以上ならば再帰

if( iLess >= start + 2 )

SortSub( start, iLess - 1);

if( iGreater <= end + 2 )

SortSub( iGreater + 1, end );

}

template <class TYPE>

void CQuickSortArray <TYPE> ::CompSortSub( const int start, const int end )

{

if( start >= end ) return;

int iLess = start; // 前半のインデックス

int iGreater = end; // 後半のインデックス

TYPE temp = ( m\_Array[iLess] + m\_Array[iGreater] ) / 2; // 基準値は両端の平均

// 交換ループ

while( iLess < iGreater )

{

// tempより小さい要素をスルー

while( m\_Array[iLess] < temp ){

iLess++;

compareCount++;

}

// tempより大きい要素をスルー

while( m\_Array[iGreater] > temp ){

iGreater--;

compareCount++;

}

// 検索位置が交わったら交換し、iLessとiGreaterを進める

if( iLess < iGreater ){

Swap( m\_Array[iLess++], m\_Array[iGreater--] );

substituteCount++; // 代入回数＋1

}

}

// iLess == iGreater で終わった時は

// iGreater < iLessとなるようにする

if( iLess == iGreater )

{

if( m\_Array[iLess] < temp )

iLess++;

else if( m\_Array[iLess] > temp )

iGreater--;

}

// 次のブロックのサイズが2以上ならば再帰

if( iLess >= start + 2 )

SortSub( start, iLess - 1);

if( iGreater <= end + 2 )

SortSub( iGreater + 1, end );

}

#endif

// OriginalSort.h

// オリジナルのソートを行う配列を扱うクラス

#ifndef \_ORIGINALSORT\_ARRAY\_H

#define \_ORIGINALSORT\_ARRAY\_H

#include "Array.h"

template <class TYPE>

class COriginalSortArray : public CArray <TYPE>

{

private:

int iMin, iMax; // 最小の要素のインデックス

TYPE nMin, nMax; // 現在の最小のデータ

public:

// コンストラクタ

COriginalSortArray( const int val );

// ソート実行

void Sort();

// 比較回数と代入回数を測るためのソート

void CompSort();

};

// コンストラクタ

template <class TYPE>

COriginalSortArray <TYPE> ::COriginalSortArray( const int val ) : CArray( val )

{

iMin = 0;

nMin = 0;

iMax = 0;

nMax = 0;

}

// ソート実行

template <class TYPE>

void COriginalSortArray <TYPE> ::Sort()

{

int h = m\_iArrayNum - 1; // 一番後ろの要素のインデックス

// 要素数 - 1 回ループ

for( int i = 0; i < h; i++ ){

// 初期値代入

iMin = iMax = i;

nMin = m\_Array[i];

nMax = m\_Array[h];

// 最小要素と最大要素を検索

for( int j = i + 1; j < h; j++ ){

if( m\_Array[j] < nMin ){ // より小さい要素が見つかったら

iMin = j;

nMin = m\_Array[j];

}

else if( m\_Array[j] > nMax ){ // より大きい要素が見つかったら

iMax = j;

nMax = m\_Array[j];

}

}

// 入れ替え

Swap( m\_Array[iMin], m\_Array[i] );

Swap( m\_Array[iMax], m\_Array[h] );

h--;

}

}

template <class TYPE>

void COriginalSortArray <TYPE> ::CompSort()

{

int h = m\_iArrayNum - 1; // 一番後ろの要素のインデックス

// 要素数 - 1 回ループ

for( int i = 0; i < h; i++ ){

// 初期値代入

iMin = iMax = i;

nMin = m\_Array[i];

nMax = m\_Array[h];

// 最小要素と最大要素を検索

for( int j = i + 1; j < h; j++ ){

if( m\_Array[j] < nMin ){ // より小さい要素が見つかったら

iMin = j;

nMin = m\_Array[j];

compareCount++; // 比較回数＋1

}

else if( m\_Array[j] > nMax ){ // より大きい要素が見つかったら

iMax = j;

nMax = m\_Array[j];

compareCount++; // 比較回数＋１

}

}

// 入れ替え

Swap( m\_Array[iMin], m\_Array[i] ); substituteCount++; // 代入回数＋1

Swap( m\_Array[iMax], m\_Array[h] ); substituteCount++; // 代入回数＋1

h--;

}

}

#endif

1. 実行結果

それぞれのソートを実行して計測した結果を以下に示す。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 要素数 | 100 | 500 | 1000 | 5000 | 10000 |
|  |  |  |  |  |  |
| 表1．バブルソート（int） | | | | | |
| 平均時間(ミリ秒) | 0 | 2 | 10 | 326 | 1100 |
| 平均カウント数 | 262 | 6388 | 25163 | 763918 | 2575039 |
| 比較回数(回) | 9801 | 249001 | 998001 | 24990001 | 99980001 |
| 代入回数(回) | 2425 | 64111 | 250041 | 6175493 | 25050290 |
| 最大時間(ミリ秒) | 0 | 4 | 16 | 457 | 1223 |
| 最小時間(ミリ秒) | 0 | 2 | 9 | 259 | 1056 |
| 最大カウント数 | 388 | 10026 | 38333 | 1070500 | 2860163 |
| 最小カウント数 | 240 | 5656 | 22774 | 605978 | 2470054 |
|  |  |  |  |  |  |
| 表2．バブルソート（float） | | | | | |
| 平均時間(ミリ秒) | 11 | 2 | 9 | 266 | 1113 |
| 平均カウント数 | 25990 | 6127 | 23467 | 623371 | 2605169 |
| 比較回数(回) | 9801 | 249001 | 998001 | 24990001 | 99980001 |
| 代入回数(回) | 2425 | 64111 | 250041 | 6175493 | 25050290 |
| 最大時間(ミリ秒) | 0 | 2 | 11 | 287 | 1267 |
| 最小時間(ミリ秒) | 0 | 2 | 9 | 258 | 1071 |
| 最大カウント数 | 481 | 6510 | 27954 | 672832 | 2965026 |
| 最小カウント数 | 218 | 5668 | 22757 | 605178 | 2504860 |
|  |  |  |  |  |  |
| 表3．選択法（int） | | | | | |
| 平均時間(ミリ秒) | 0 | 0 | 1 | 36 | 157 |
| 平均カウント数 | 48 | 962 | 3549 | 87414 | 368360 |
| 比較回数(回) | 4950 | 124750 | 499500 | 12497500 | 49995000 |
| 代入回数(回) | 99 | 499 | 999 | 4999 | 9999 |
| 最大時間(ミリ秒) | 0 | 0 | 2 | 54 | 172 |
| 最小時間(ミリ秒) | 0 | 0 | 1 | 34 | 150 |
| 最大カウント数 | 80 | 1503 | 4988 | 127776 | 403789 |
| 最小カウント数 | 41 | 879 | 3336 | 79724 | 351820 |
|  |  |  |  |  |  |
| 表4．選択法（float） | | | | | |
| 平均時間(ミリ秒) | 1 | 0 | 1 | 38 | 150 |
| 平均カウント数 | 3746 | 1069 | 3730 | 91382 | 354003 |
| 比較回数(回) | 4950 | 124750 | 499500 | 12497500 | 49995000 |
| 代入回数(回) | 99 | 499 | 999 | 4999 | 9999 |
| 最大時間(ミリ秒) | 0 | 0 | 2 | 52 | 186 |
| 最小時間(ミリ秒) | 0 | 0 | 1 | 33 | 122 |
| 最大カウント数 | 72 | 1247 | 4791 | 122446 | 435050 |
| 最小カウント数 | 59 | 897 | 2265 | 79369 | 285614 |
|  |  |  |  |  |  |
| 表5．クイックソート（int） | | | | | |
| 平均時間(ミリ秒) | 0 | 0 | 0 | 1 | 2 |
| 平均カウント数 | 32 | 217 | 476 | 2727 | 5792 |
| 比較回数(回) | 66 | 236 | 509 | 2519 | 6790 |
| 代入回数(回) | 17 | 132 | 245 | 1240 | 1605 |
| 最大時間(ミリ秒) | 0 | 0 | 0 | 1 | 3 |
| 最小時間(ミリ秒) | 0 | 0 | 0 | 1 | 2 |
| 最大カウント数 | 76 | 1193 | 1741 | 3233 | 8504 |
| 最小カウント数 | 26 | 176 | 424 | 2652 | 5416 |
|  |  |  |  |  |  |
| 表6．クイックソート（float） | | | | | |
| 平均時間(ミリ秒) | 0 | 0 | 0 | 1 | 2 |
| 平均カウント数 | 86 | 249 | 601 | 3192 | 6713 |
| 比較回数(回) | 66 | 236 | 509 | 2519 | 6790 |
| 代入回数(回) | 17 | 132 | 245 | 1240 | 1605 |
| 最大時間(ミリ秒) | 0 | 0 | 0 | 2 | 3 |
| 最小時間(ミリ秒) | 0 | 0 | 0 | 1 | 2 |
| 最大カウント数 | 48 | 355 | 764 | 4983 | 9142 |
| 最小カウント数 | 26 | 207 | 492 | 3029 | 5679 |
|  |  |  |  |  |  |
| 表7．オリジナルソート（int） | | | | | |
| 平均時間(ミリ秒) | 0 | 0 | 0 | 21 | 84 |
| 平均カウント数 | 44 | 577 | 2402 | 51727 | 198165 |
| 比較回数(回) | 310 | 2315 | 5194 | 34214 | 76031 |
| 代入回数(回) | 100 | 500 | 1000 | 5000 | 10000 |
| 最大時間(ミリ秒) | 0 | 0 | 1 | 30 | 110 |
| 最小時間(ミリ秒) | 0 | 0 | 0 | 19 | 78 |
| 最大カウント数 | 74 | 704 | 3157 | 70706 | 257269 |
| 最小カウント数 | 40 | 534 | 2220 | 46393 | 183584 |
|  |  |  |  |  |  |
| 表8．オリジナルソート（float） | | | | | |
| 平均時間(ミリ秒) | 0 | 0 | 0 | 21 | 84 |
| 平均カウント数 | 2016 | 591 | 2102 | 52052 | 198684 |
| 比較回数(回) | 310 | 2315 | 5194 | 34214 | 76031 |
| 代入回数(回) | 100 | 500 | 1000 | 5000 | 10000 |
| 最大時間(ミリ秒) | 0 | 0 | 1 | 30 | 101 |
| 最小時間(ミリ秒) | 0 | 0 | 0 | 20 | 80 |
| 最大カウント数 | 62 | 708 | 2835 | 70609 | 236205 |
| 最小カウント数 | 30 | 550 | 2061 | 47720 | 188632 |

以下に、ソート前の配列の内容を表示させた結果の一部と、ソート後の配列の内容を表示させた結果の一部を示す。

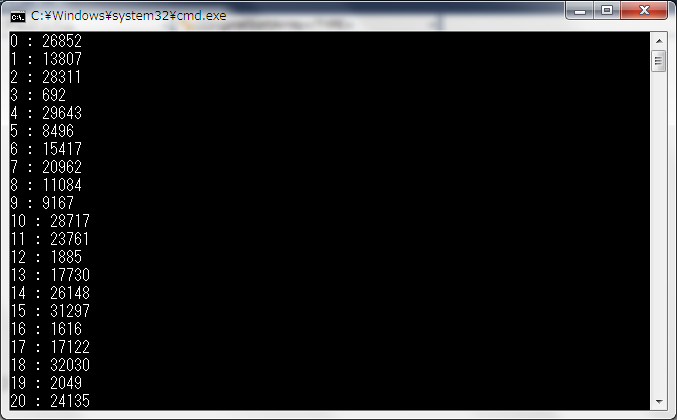


図2：ソート前の配列

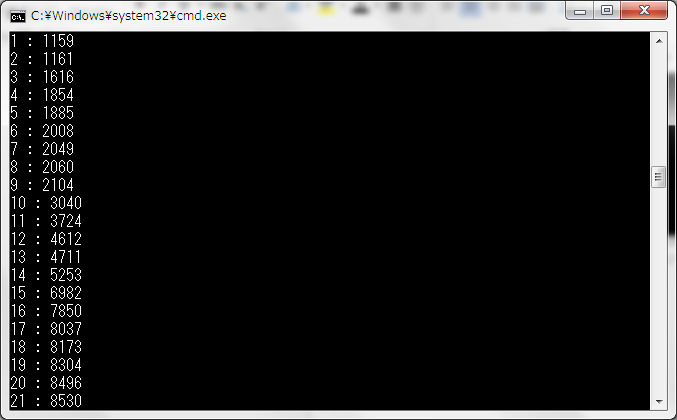


図3：ソート後の配列

1. 考察

要素数が100や500程度の少ない配列の場合ではかかった時間が短すぎるためミリ秒では表現できない数値となってしまった。クイックソートに関してはグラフの通りほとんど経過時間に変化が見られなかった。クイックソートならば、より大きな要素数の配列でも実用的な速度でソートできると思われる。

独自のソートアルゴリズムは選択法の1ループ内で、最も小さいものと最も大きいものをそれぞれ検出し、最も小さいものを配列の1番最初の位置に、最も大きいものを配列の1番最後の位置に入れ替える処理を行っている。比較回数を減らして選択法の効率化を目指したものだが、結果の通りかかった時間が短くなった。

プログラムを作成しながらソートを何度も試したが、パソコンはシステムの動作に必要なソフトウェアなども実行させているせいか結果があまり安定しなかった。今回の課題では100回の試行回数で検証したがより多くの回数を重ねた方が正確な値が計算できると考えられる。

参考文献

基本的なソートアルゴリズム

<http://alctail.sakura.ne.jp/tip/iroiro_kannrenn/sort_algorithm/>

参照日　5月17日

ロベールのC++教室 第3部

<http://www7b.biglobe.ne.jp/~robe/cpphtml/>

参照日 5月18日