【課題3-1】

Nを16以下の整数とする。N個の整数乱数(1以上100以下の整数)を発生させ、配列Merge[]に格納する。マージソートによりソートするプログラムを作成する。

【ソースプログラム】

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define N 8

int Merge[N], tmpmerge[N];

void printdata(int data[]); /\*ソートの途中経過や結果の出力\*/

void m\_sort(int first, int last); /\*関数名mergesortはstdlib.hの

\*mergesortと競合してしまうため

\*m\_sortにした

\*/

void merge\_two(int fr1, int to1, int fr2, int to2);

int main(void){

int i;

/\*1~100の乱数を発生させ、Merge[]に格納\*/

srand((unsigned int)time(0));

for(i = 0; i < N; i++){

Merge[i] = rand() % 100 + 1;

}

printf("Numbers: ");

printdata(Merge);

m\_sort(0, N-1);

return 0;

}

void m\_sort(int first, int last){

if(first < last){

m\_sort(first, (first + last)/2);

m\_sort(((first + last)/2) + 1, last);

merge\_two(first, (first + last)/2, ((first + last)/2) + 1, last);

}

}

void merge\_two(int fr1, int to1, int fr2, int to2){

int i = 0, sw1 = 0, sw2 = 0, tmp1 = fr1, tmp2 = fr2; /\*sw1, sw2:

\*真(1)か偽

\*(0)か

\*/

for(i = fr1; i <= to2; i++){

tmpmerge[i] = Merge[i];

}

i = fr1;

while(i <= to2){

if((sw1 == 0)&&(sw2 == 0)){

if(tmpmerge[tmp1] == tmpmerge[tmp2]){

Merge[i] = tmpmerge[tmp1];

++tmp1;

++i;

Merge[i] = tmpmerge[tmp2];

++tmp2;

++i;

if(tmp1 > to1){

sw1 = 1;

}

if(tmp2 > to2){

sw2 = 1;

}

}else if(tmpmerge[tmp1] < tmpmerge[tmp2]){

Merge[i] = tmpmerge[tmp1];

++tmp1;

++i;

if(tmp1 > to1){

sw1 = 1;

}

}else if(tmpmerge[tmp1] > tmpmerge[tmp2]){

Merge[i] = tmpmerge[tmp2];

++tmp2;

++i;

if(tmp2 > to2){

sw2 = 1;

}

}

}else if((sw1 == 1)&&(sw2 == 0)){

Merge[i] = tmpmerge[tmp2];

++tmp2;

++i;

if(tmp2 > to2){

sw2 = 1;

}

}else if((sw2 == 1)&&(sw1 == 0)){

Merge[i] = tmpmerge[tmp1];

++tmp1;

++i;

if(tmp1 > to1){

sw1 = 1;

}

}

if((sw1 == 1)&&(sw2 == 1)){

break;

}

}

printf("fr1=%d, to1=%d, fr2=%d, to2=%d\n", fr1, to1, fr2, to2);

printdata(Merge);

}

void printdata(int data[]){

int i;

for(i = 0; i < N; i++){

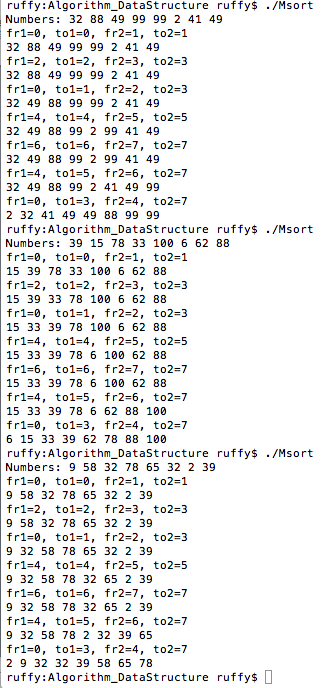
printf("%d ", data[i]);

}

printf("\n");

}

【実行結果】



【課題3-2】

N個の整数乱数(1以上10000以下の整数)を発生させ、配列Bubble[], Quick[], Merge[]に格納する。バブルソート、クイックソート、マージソートでそれぞれソートするときの実行時間を比較する。ソートに要する時間について考察する。

【ソースプログラム】

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define Nmax 10000000

int Bubble[Nmax], Quick[Nmax], Merge[Nmax], tmpmerge[Nmax], N;

void bubblesort();

void swap(int sw1, int sw2);

void quicksort(int from, int to);

void m\_sort(int first, int last); /\*関数名mergesortはstdlib.hの

\*mergesortと競合してしまうため

\*m\_sortにした

\*/

void merge\_two(int fr1, int to1, int fr2, int to2);

int main(void){

int i;

double t, x, y, z; /\*t: ソート処理前のclock, x: バブルソート実行時間,

\*y: クイックソート実行時間, z: マージソート実行時

\*間

\*/

printf("N? ");

scanf("%d", &N);

/\*set random numbers Bubble[], Quick[], Merge[]\*/

srand((unsigned int)time(0));

for(i = 0; i < N; i++){

Bubble[i] = Quick[i] = Merge[i] = (rand() % 10000) + 1;

}

t = clock();

bubblesort();

x = (clock() - t) / CLOCKS\_PER\_SEC;

printf("Bubble Sort %g sec \n", x);

t = clock();

quicksort(0, N - 1);

y = (clock() - t) / CLOCKS\_PER\_SEC;

printf("Quick Sort %g sec \n", y);

t = clock();

m\_sort(0, N - 1);

z = (clock() - t) / CLOCKS\_PER\_SEC;

printf("Merge Sort %g sec \n", z);

return 0;

}

/\*\*\*ここからバブルソート\*\*\*/

void bubblesort(){

int i, j, tmp;

for(i = 0; i < N-1; i++){

for(j = N-1; j > i; j--){

if(Bubble[j-1] > Bubble[j]){

tmp = Bubble[j];

Bubble[j] = Bubble[j-1];

Bubble[j-1] = tmp;

}

}

}

}

/\*\*\*ここまでバブルソート\*\*\*/

/\*\*\*ここからクイックソート\*\*\*/

void swap(int sw1, int sw2){

int tmp;

tmp = Quick[sw1];

Quick[sw1] = Quick[sw2];

Quick[sw2] = tmp;

}

void quicksort(int from, int to){

if(from < to){

int pivot = Quick[from];

int less = from, greater = from + 1, i;

for(i = greater; i <= to; i++){

if(Quick[i] < pivot){

less++;

swap(less, i);

}

}

Quick[from] = Quick[less];

Quick[less] = pivot;

quicksort(from, less - 1);

quicksort(less + 1, to);

}

}

/\*\*\*ここまでクイックソート\*\*\*/

/\*\*\*ここからマージソート\*\*\*/

void m\_sort(int first, int last){

if(first < last){

m\_sort(first, (first + last)/2);

m\_sort(((first + last)/2) + 1, last);

merge\_two(first, (first + last)/2, ((first + last)/2) + 1, last);

}

}

void merge\_two(int fr1, int to1, int fr2, int to2){

int i = 0, sw1 = 0, sw2 = 0, tmp1 = fr1, tmp2 = fr2; /\*sw1, sw2:

\*真(1)か偽

\*(0)か

\*/

for(i = fr1; i <= to2; i++){

tmpmerge[i] = Merge[i];

}

i = fr1;

while(i <= to2){

if((sw1 == 0)&&(sw2 == 0)){

if(tmpmerge[tmp1] == tmpmerge[tmp2]){

Merge[i] = tmpmerge[tmp1];

++tmp1;

++i;

Merge[i] = tmpmerge[tmp2];

++tmp2;

++i;

if(tmp1 > to1){

sw1 = 1;

}

if(tmp2 > to2){

sw2 = 1;

}

}else if(tmpmerge[tmp1] < tmpmerge[tmp2]){

Merge[i] = tmpmerge[tmp1];

++tmp1;

++i;

if(tmp1 > to1){

sw1 = 1;

}

}else if(tmpmerge[tmp1] > tmpmerge[tmp2]){

Merge[i] = tmpmerge[tmp2];

++tmp2;

++i;

if(tmp2 > to2){

sw2 = 1;

}

}

}else if((sw1 == 1)&&(sw2 == 0)){

Merge[i] = tmpmerge[tmp2];

++tmp2;

++i;

if(tmp2 > to2){

sw2 = 1;

}

}else if((sw2 == 1)&&(sw1 == 0)){

Merge[i] = tmpmerge[tmp1];

++tmp1;

++i;

if(tmp1 > to1){

sw1 = 1;

}

}

if((sw1 == 1)&&(sw2 == 1)){

break;

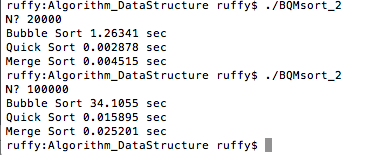
}

}

}

/\*\*\*ここまでマージソート\*\*\*/

【実行結果】



【結果】

　要素数を5000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000 と変化させて上記プログラムを実行したときのそれぞれのソートによるソート時間を表としてまとめたものを以下に示す。

表1. バブルソートの実行時間

|  |  |
| --- | --- |
| 要素数[個] | 時間[秒] |
| 5000 | 0.068182 |
| 10000 | 0.300709 |
| 20000 | 1.26431 |
| 30000 | 2.90318 |
| 40000 | 5.21242 |
| 50000 | 8.29179 |
| 要素数[個] | 時間[秒] |
| 60000 | 11.9646 |
| 70000 | 16.4597 |
| 80000 | 21.5993 |
| 90000 | 27.3849 |
| 100000 | 34.1055 |

表2. クイックソートの実行時間

|  |  |
| --- | --- |
| 要素数[個] | 時間[秒] |
| 5000 | 0.000662 |
| 10000 | 0.001361 |
| 20000 | 0.002878 |
| 30000 | 0.004439 |
| 40000 | 0.006296 |
| 50000 | 0.007628 |
| 60000 | 0.009575 |
| 70000 | 0.010825 |
| 80000 | 0.012342 |
| 90000 | 0.014054 |
| 100000 | 0.015895 |

表3. マージソートの実行時間

|  |  |
| --- | --- |
| 要素数[個] | 時間[秒] |
| 5000 | 0.001068 |
| 10000 | 0.002166 |
| 20000 | 0.004515 |
| 30000 | 0.007005 |
| 40000 | 0.009422 |
| 50000 | 0.011939 |
| 60000 | 0.014472 |
| 70000 | 0.017065 |
| 80000 | 0.019505 |
| 要素数[個] | 時間[秒] |
| 90000 | 0.022438 |
| 100000 | 0.025201 |

【考察】

　表1, 2, 3を比較すると、クイックソート, マージソート, バブルソートの順でソートにかかる実行時間が短く、理論通りであることを確認できるが、クイックソートとマージソートの間に生じた僅差は、今回はランダムな整数データを扱っているため、クイックソートの長所が生かされて最も実行時間が短くなっていると考える。これらより、クイックソートが最も高速であるとして現実では用いられていることが分かる。

　結果の項に記した表を参考に、要素数を変化させてそれぞれのソートアルゴリズムを適用したときの実行時間をグラフにしたものを図として以下に示す。なお、比較のためにNLogNとN2のグラフも示す。

図1. 各種ソートアルゴリズムによる実行時間の比較(縦軸のみ対数目盛)

図2. 各種ソートアルゴリズムによる実行時間の比較(縦軸, 横軸共に対数目盛)

　図1, 2より、各種ソートアルゴリズムの実行時間を比較すると、バブルソートのグラフのみN2のグラフと同様の形をし、また平行になっていることが分かる。これはつまりバブルソートがオーダー(n2)の計算量であることを示している。

　図1, 2より、クイックソートとマージソートのグラフはNLogNのグラフと同様の形をし、また平行になっていることが分かる。これはつまりクイックソートとマージソートがオーダー(nlogn)の計算量であることを示している。