- 프로그램 소스

#include <stdio.h>

#include <stdlib.h>

#include <Windows.h>

#include <time.h>

#define N 1000000 // 리스트의 크기

#define Limit 50 // 거의 정렬된, 거의 역정렬된 리스트를 위해 선언한 변수

typedef struct Arrays // 리스트 6개를 전달하기 위한 구조체 선언

{

int \*A, \*B, \*C, \*A1, \*B1, \*C1; // 총 6개의 리스트

}arrays;

typedef struct Quick // 파티션에게서 중복값의 처음과 끝 위치를 받기위한 구조체

{

int a, b; // 중복원소가 모여있는 곳의 첫과 마지막 위치값

}quick;

void mergeSort(int \* A, int l, int r);

/\*

function name : mergeSort

pre condition(전제 조건) : 함수를 호출하면서 리스트의 처음, 끝 인덱스값과 리스트를 전달받는다

function(기능) : 합병 정렬을 수행한다

\*/

void merge(int \* A, int l, int m, int r);

/\*

function name : merge

pre condition(전제 조건) : 함수를 호출하면서 리스트의 처음, 중간, 끝 인덱스값과 리스트를 전달받는다

function(기능) : 정렬되있는 것을 통합한다

\*/

void inPlaceQuickSort(int \* L, int l, int r);

/\*

function name : inPlaceQuickSort

pre condition(전제 조건) : 함수를 호출하면서 리스트의 처음, 끝 인덱스값과 리스트를 전달받는다

function(기능) : 제자리 퀵정렬을 수행한다

\*/

quick inPlacePartition(int \*A, int l, int r, int k);

/\*

function name : inPlacePartition

pre condition(전제 조건) : 함수를 호출하면서 리스트의 처음, 끝 인덱스값과 리스트와 기준이 될 원소 위치를 전달받는다

function(기능) : 기준값을 기준으로 파티션 작업 수행 -> 역정렬

\*/

void arraySwap(int \*A, int n1, int n2);

/\*

function name : arraySwap

pre condition(전제 조건) : 리스트와 바꾸고자 하는 두 값의 위치 값

function(기능) : A[n1]과 A[n2]의 값을 바꿔준다

\*/

arrays createArrays();

/\*

function name : createArrays

pre condition(전제 조건) : none

function(기능) : 6개의 리스트를 생성하고, 전달

\*/

void a\_s\_inPlaceQuickSort(int \* L, int l, int r);

/\*

function name : a\_s\_inPlaceQuickSort

pre condition(전제 조건) : 함수를 호출하면서 리스트의 처음, 끝 인덱스값과 리스트를 전달받는다

function(기능) : 리스트를 거의 정렬한다

\*/

void a\_i\_s\_inPlaceQuickSort(int \* L, int l, int r);

/\*

function name : a\_i\_s\_inPlaceQuickSort

pre condition(전제 조건) : 함수를 호출하면서 리스트의 처음, 끝 인덱스값과 리스트를 전달받는다

function(기능) : 리스트를 거의 역정렬한다

\*/

quick inverse\_inPlacePartition(int \*A, int l, int r, int k);

/\*

function name : inverse\_inPlaceParition

pre condition(전제 조건) : 함수를 호출하면서 리스트의 처음, 끝 인덱스값과 리스트와 기준이 될 원소 위치를 전달받는다

function(기능) : 기준값을 기준으로 파티션 작업 수행 -> 역정렬

\*/

int main()

{

arrays L; // 리스트 6개를 받는 구조체 arrays 변수

LARGE\_INTEGER Start, End, Freq; // 실행시간을 재는 변수들.

double runtime; // 실행시간을 저장하는 변수

L = createArrays(); // 리스트 6개를 만들고 구조체로 넘겨받는다

QueryPerformanceFrequency(&Freq);

QueryPerformanceCounter(&Start); // 실행시간 시작시간

inPlaceQuickSort(L.A, 0, N-1); // {deterministic version} 제자리 결정적 퀵 정렬 실행

QueryPerformanceCounter(&End); // 실행시간 종료시간

runtime = (double)(End.QuadPart - Start.QuadPart) / (double)Freq.QuadPart; // 실행시간 계산

printf("퀵정렬(랜덤) CPU time = %0.10lf\n",runtime); // 실행시간 출력

QueryPerformanceFrequency(&Freq);

QueryPerformanceCounter(&Start); // 실행시간 시작시간

mergeSort(L.A1, 0, N-1);// 합병정렬 실행

QueryPerformanceCounter(&End); // 실행시간 종료시간

runtime = (double)(End.QuadPart - Start.QuadPart) / (double)Freq.QuadPart; // 실행시간 계산

printf("합병정렬(랜덤) CPU time = %0.10lf\n",runtime); // 실행시간 출력

printf("\n");

// 랜덤 리스트

QueryPerformanceFrequency(&Freq);

QueryPerformanceCounter(&Start); // 실행시간 시작시간

inPlaceQuickSort(L.B, 0, N-1); // {deterministic version} 제자리 결정적 퀵 정렬 실행

QueryPerformanceCounter(&End); // 실행시간 종료시간

runtime = (double)(End.QuadPart - Start.QuadPart) / (double)Freq.QuadPart; // 실행시간 계산

printf("퀵정렬(거의 정렬된) CPU time = %0.10lf\n",runtime); // 실행시간 출력

QueryPerformanceFrequency(&Freq);

QueryPerformanceCounter(&Start); // 실행시간 시작시간

mergeSort(L.B1, 0, N-1);// 합병정렬 실행

QueryPerformanceCounter(&End); // 실행시간 종료시간

runtime = (double)(End.QuadPart - Start.QuadPart) / (double)Freq.QuadPart; // 실행시간 계산

printf("합병정렬(거의 정렬된) CPU time = %0.10lf\n",runtime); // 실행시간 출력

printf("\n");

// 거의 정렬된 리스트

QueryPerformanceFrequency(&Freq);

QueryPerformanceCounter(&Start); // 실행시간 시작시간

inPlaceQuickSort(L.C, 0, N-1);// {deterministic version} 제자리 결정적 퀵 정렬 실행

QueryPerformanceCounter(&End); // 실행시간 종료시간

runtime = (double)(End.QuadPart - Start.QuadPart) / (double)Freq.QuadPart; // 실행시간 계산

printf("퀵정렬(거의 역정렬된) CPU time = %0.10lf\n",runtime); // 실행시간 출력

QueryPerformanceFrequency(&Freq);

QueryPerformanceCounter(&Start); // 실행시간 시작시간

mergeSort(L.C1, 0, N-1); // 합병정렬 실행

QueryPerformanceCounter(&End); // 실행시간 종료시간

runtime = (double)(End.QuadPart - Start.QuadPart) / (double)Freq.QuadPart; // 실행시간 계산

printf("합병정렬(거의 역정렬된) CPU time = %0.10lf\n",runtime); // 실행시간 출력

// 거의 역정렬된 리스트

return 0;

}

void mergeSort(int \* A, int l, int r) {

int m;

if (l < r) {

m = (l + r)/2; // 리스트 반으로 분할

mergeSort(A, l, m); //왼쪽 블록 분할

mergeSort(A, m+1, r); //오른쪽 블록 분할

merge(A, l, m, r); //분할된 블록 병합

}

}

void merge(int \* A, int l, int m, int r) {

int i, j, k;

int B[N]; // N크기의 배열 선언

i = l;

j = m + 1;

k = l; //결과 배열의 인덱스

//i부터 m 까지의 블록과 j부터 r까지의 블록을 서로 비교하는 부분

while (i <= m && j <= r) {

if (A[i] <= A[j]) //i 값이 j 값보다 작거나 같으면 i 값을 결과 복사

B[k++] = A[i++];

else //아니라면 j 값을 복사

B[k++] = A[j++];

}

while (i <= m)

B[k++] = A[i++];

while (j <= r)

B[k++] = A[j++];

for (k = l; k <= r; k++)

A[k] = B[k]; // 복사했던 값 돌려줌

}

void inPlaceQuickSort(int \* L, int l, int r) {

int k;

quick q;

if (l >= r)

return;

k = r; // 결정적 퀵정렬이기 때문에 기준원소는 가장 마지막에 위치한 값이다

q = inPlacePartition(L, l, r, k); // 중복원소 가능한 파티션 호출

inPlaceQuickSort(L, l, (q.a)-1); // 기준값 기준으로 왼쪽 분할 퀵정렬

inPlaceQuickSort(L, (q.b)+1, r); // 기준값 기준으로 오른쪽 분할 퀵정렬

}

quick inPlacePartition(int \*A, int l, int r, int k){

quick q;

int i, j;

int a, b;

int temp\_loc;

int p;

p = A[k]; // k의 위치에 있는 값이 기준값

i = l; // i는 l을 시작점으로 한다

j = r-1; // j는 r-1을 시작점으로 한다

a = l; // 중복된 원소를 배열 가장 앞부터 모아둔다

b = r; // 중복된 원소를 배열 가장 뒤부터 모아둔다

while(i <= j){

while(i <= j && A[i] <= p){ // i은 l를 시작점으로 p보다 작거나 같은값을 찾음

if(A[i] == p){ // 같은값을 찾을 경우, 해당범위내 배열 왼쪽에 모아둠

arraySwap(A,a,i); // 현재 i의 값과 a의 값을 바꾼다

a++; // a가 다음 인덱스값으로 이동한다

}

i++;

}

while(i <= j && A[j] >= p){ // j은 r-1를 시작점으로 p보다 크거나 같은값을 찾음

if(A[j] == p){ // 같은값을 찾을 경우, 해당범위내 배열 오른쪽에 모아둠

b--;

arraySwap(A,j,b);

}

j--;

}

if(i < j){

arraySwap(A,i,j); // i < j일 경우 i과 j의 값을 바꿈

}

}

// 해당 범위내의 배열의 왼쪽과 오른쪽에 모아둔 p과 중복되는 값을 배열 중앙으로 보낸다

temp\_loc = i-1; // a에게 i-1의 위치를 전달하기 위해 임시로 저장

while(a > i){

arraySwap(A,a,temp\_loc);

a--;

temp\_loc--;

} // a가 i보다 클 때 중복된 값을 하나씩 중앙으로 이동한다

a = temp\_loc+1; // a는 현재 중복된 원소 모아놓은 것 의 가장 첫번째 위치를 가진다

temp\_loc = i; // a에게 i의 위치를 전달하기 위해 임시로 저장

while(b < r+1){

arraySwap(A,b, temp\_loc);

b++;

temp\_loc++;

} // b가 r+1보다 작을 때 중복된 값을 하나씩 중앙으로 이동한다

b = temp\_loc-1; // b는 현재 중복된 원소 모아놓은 것 의 가장 마지막 위치를 가진다

q.a = a;

q.b = b; // 구조체 이용해서 a,b값 전달

return q;

}

void arraySwap(int \* A, int n1, int n2) {

int temp; // 임시로 값을 저장할 변수

temp = A[n1];

A[n1] = A[n2];

A[n2] = temp;

// A[n1]의 값과 A[n2]의 값을 swap한다

}

arrays createArrays() {

arrays L;

int i;

L.A = (int\*)malloc(sizeof(int) \* N);

L.B = (int\*)malloc(sizeof(int) \* N);

L.C = (int\*)malloc(sizeof(int) \* N);

L.A1 = (int\*)malloc(sizeof(int) \* N);

L.B1 = (int\*)malloc(sizeof(int) \* N);

L.C1 = (int\*)malloc(sizeof(int) \* N);

// N의 크기를 가진 리스트를 6개 생성한다

srand((unsigned)time(NULL));

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

L.A[i] = rand()%N+1;

} // 1~N사이의 난수로 이루어진 L.A 리스트

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

L.B[i]=L.A[i];

L.C[i]=L.A[i];

} // copyArray 수행, B와 C를 A와 동일한 리스트로 생성

a\_s\_inPlaceQuickSort(L.B, 0, N-1);

a\_i\_s\_inPlaceQuickSort(L.C, 0, N-1);

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

L.A1[i]=L.A[i]; // A1을 A와 동일한 리스트로 생성

L.B1[i]=L.B[i]; // B1을 B와 동일한 리스트로 생성

L.C1[i]=L.C[i]; // C1을 C와 동일한 리스트로 생성

} // copyArray 수행

return L;

}

void a\_s\_inPlaceQuickSort(int \* A, int l, int r) {

quick q;

if ((r-l) > Limit) { // 퀵정렬을 어느 정도의 시점까지만 실행

q = inPlacePartition(A,l,r,0);

a\_s\_inPlaceQuickSort(A, l, q.a-1);

a\_s\_inPlaceQuickSort(A, q.b+1, r);

}

// 거의 정렬된 리스트가 된다

}

void a\_i\_s\_inPlaceQuickSort(int \* A, int l, int r){

quick q;

if ((r-l) > Limit) { // 퀵정렬(역)을 어느 정도의 시점까지만 실행

q = inverse\_inPlacePartition(A, l, r, 0);

a\_i\_s\_inPlaceQuickSort(A, l, q.a-1);

a\_i\_s\_inPlaceQuickSort(A, q.b+1, r);

}

// 거의 역정렬된 리스트가 된다

}

quick inverse\_inPlacePartition(int \*A, int l, int r, int k){

quick q;

int i, j;

int a, b;

int temp\_loc;

int p;

p = A[k];

i = l;

j = r-1;

a = l;

b = r;

// 주석 달린 부분 제외하곤 inPlacePartition과 동일

while(i <= j){

while(i <= j && A[i] >= p){ // 역정렬을 위해 inPlacePartition과는 다르게 i은 l를 시작점으로 p보다 크거나 같은값을 찾음

if(A[i] == p){

arraySwap(A,a,i);

a++;

}

i++;

}

while(i <= j && A[j] <= p){ // 역정렬을 위해 inPlacePartition과는 다르게 j은 r-1를 시작점으로 p보다 작거나 같은값을 찾음

if(A[j] == p){

b--;

arraySwap(A,j,b);

}

j--;

}

if(i < j){

arraySwap(A,i,j);

}

}

temp\_loc = i-1;

while(a > i){

arraySwap(A,a,temp\_loc);

a--;

temp\_loc--;

}

a = temp\_loc+1;

temp\_loc = i;

while(b < r+1){

arraySwap(A,b, temp\_loc);

b++;

temp\_loc++;

}

b = temp\_loc-1;

q.a = a;

q.b = b;

return q;

}

- 사용자 입력과 프로그램 출력을 보여주는 이미지

