알고리즘\_과제\_3

(정렬 알고리즘 실행시간 비교)



학부:컴퓨터학부

학번:20162518

출석번호 : 156번

이름 : 최승서

[소스코드 및 주석]

#include<stdio.h>

#include<windows.h>

#include<time.h>

#define N 5000 //n의 크기를 지정

LARGE\_INTEGER PreFreq; //QueryPerformanceFrequency 함수를 사용하기 위한 변수 지정

LARGE\_INTEGER s1, s2;

void merge\_sort(double a[], int i, int j);

void merge(double a[], int i1, int j1, int i2, int j2);

void quick\_sort(double[], int, int);

int q\_partition(double[], int, int);

void exchange\_sort(double a[]) {

int i, j;

double tmp;

for (i = 0; i < (N- 1); i++){ //a[i]를 증가시키며 비교

for (j = (i + 1); j < N; j++){ //a[i]값을 a[j]를 증가시키며 값을 비교

if (a[i] > a[j]){ // a[i]가 a[j]보다 크면 둘이 교환

tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

}

}

}

void insertion\_sort(double a[]) {

int i, j;

double tmp;

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

tmp = a[i]; //a[i]값을 tmp에 저장

for (j = i - 1; j >= 0 && a[j] > tmp; j--) //정렬된 배열이 i-1까지 이므로 역순으로 확인한다.

a[j + 1] = a[j]; //배열의 오른쪽 이동

a[j + 1] = tmp;

}

}

void merge\_sort(double a[], int start, int end) //합병정렬

{

int mid;

if (start < end){

mid = (start + end) / 2;

merge\_sort(a, start, mid); //왼쪽 절반 재귀호출

merge\_sort(a, mid + 1, end); // 오른쪽 절반 재귀호출

merge(a, start, mid, mid + 1, end); //정렬된 두개를 merge(합병)

}

}

void merge(double a[], int start, int mid, int midn, int end){

double tmp[N]; //합병을 위한 임시배열

int i, j, k;

i = start; //왼쪽 배열의 시작

j = midn; //오른쪽 배열의 시작

k = 0;

while (i <= mid && j <= end){ //양쪽 배열 끝까지 반복

if (a[i] <= a[j])

tmp[k++] = a[i++];

else

tmp[k++] = a[j++];

}

while (i <= mid) //첫번째 배열의 남아있는 값을 임시배열에 복사

tmp[k++] = a[i++];

while (j <= end) //두번째 배열의 남아있는 값을 임시배열에 복사

tmp[k++] = a[j++];

for (i = start, j = 0; i <= end; i++, j++) //임시배열에 있는 값을 원래 배열에 복사

a[i] = tmp[j];

}

void quick\_sort(double a[], int start, int end){

int j; //피벗의 위치

if (start < end){

j = q\_partition(a, start, end); //피벗을 기준으로 배열 분할

quick\_sort(a, start, j - 1); //피벗을 기준으로 왼쪽 배열 list(정렬)-conquer(정복)

quick\_sort(a, j + 1, end); //피벗을 기준으로 오른쪽 배열 list(정렬)-conquer(정복)

}

}

int q\_partition(double a[], int start, int end){

int i, j;

double pivot, temp;

pivot = a[start]; //정렬할 배열의 첫번째 데이터를 피벗으로 지정

i = start;

j = end + 1;

do {

do {

i++; //i가 start+1에서 시작한다

} while (a[i] < pivot && i <= end); //a[i]가 pivot보다 작으면 i증가

do {

j--; //j는 end에서 시작한다

} while (pivot < a[j]); //a[j]가 pivot보다 크면 j 감소

if (i < j){ //i와 j가 교차하지 않았으면 a[i]와 a[j]를 교환해준다.

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

} while (i < j); //i, j가 교차할때 까지 반복해준다.

//i ,j가 교차했으면 a[start]와 a[j]를 교환해준다.

a[start] = a[j];

a[j] = pivot;

return j; //pivot의 위치를 반환한다

}

void copy\_arry(double a[], double tmp[]) { //배열 복사함수

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

tmp[i] = a[i];

}

void print\_arry(double a[]) { //배열 출력함수

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

printf("%.3lf ", a[i]);

}

float randomRange() { // -1~1 사이의 실수값 난수를 생성

return (2 \* ((double)rand() / RAND\_MAX)) - 1;

}

int main(void) {

double ary[N] = { 0, }, tmp[N] = { 0 , };

double cost\_1, cost\_2, cost\_3, cost\_4; //각 sort별 실행시간 저장

srand((unsigned)time(NULL));

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

ary[i] = randomRange();

copy\_arry(ary, tmp); //각 정렬별로 똑같은 배열을 정렬하기 위해 임시 배열에 저장

printf("BEFORE SORTING : ");

print\_arry(ary);

QueryPerformanceFrequency(&PreFreq);

QueryPerformanceCounter(&s1);

exchange\_sort(ary);

QueryPerformanceCounter(&s2); //interative함수 수행시간측정을 위한 종단점

cost\_1 = (double)(s2.QuadPart - s1.QuadPart) / (double)PreFreq.QuadPart; //각각 시작점과 종단점에 의해 얻어진 값의 차이를 구해서 수행시간 판단

printf("\n\n-----------------------------------------------------");

printf("\nAFTER SORTING : ");

print\_arry(ary);

printf("\n\n-----------------------------------------------------");

copy\_arry(tmp, ary); //다시 처음 배열로 초기화

QueryPerformanceFrequency(&PreFreq); // sort 정렬시간 측정을 위한 함수

QueryPerformanceCounter(&s1); //측정시작

insertion\_sort(ary);

QueryPerformanceCounter(&s2); //측정 끝

cost\_2 = (double)(s2.QuadPart - s1.QuadPart) / (double)PreFreq.QuadPart; //각각 시작점과 종단점에 의해 얻어진 값의 차이를 구해서 수행시간 판단

copy\_arry(tmp, ary);

QueryPerformanceFrequency(&PreFreq);

QueryPerformanceCounter(&s1);

merge\_sort(ary, 0, N - 1);

QueryPerformanceCounter(&s2);

cost\_3 = (double)(s2.QuadPart - s1.QuadPart) / (double)PreFreq.QuadPart;

copy\_arry(tmp, ary);

QueryPerformanceFrequency(&PreFreq);

QueryPerformanceCounter(&s1);

quick\_sort(ary, 0, N - 1);

QueryPerformanceCounter(&s2);

cost\_4 = (double)(s2.QuadPart - s1.QuadPart) / (double)PreFreq.QuadPart;

printf("\n n = %d일때 측정값\n", N);

printf("\n exchange sort 측정시간: %f\n", cost\_1);

printf("\n insertion sort 측정시간: %f\n", cost\_2);

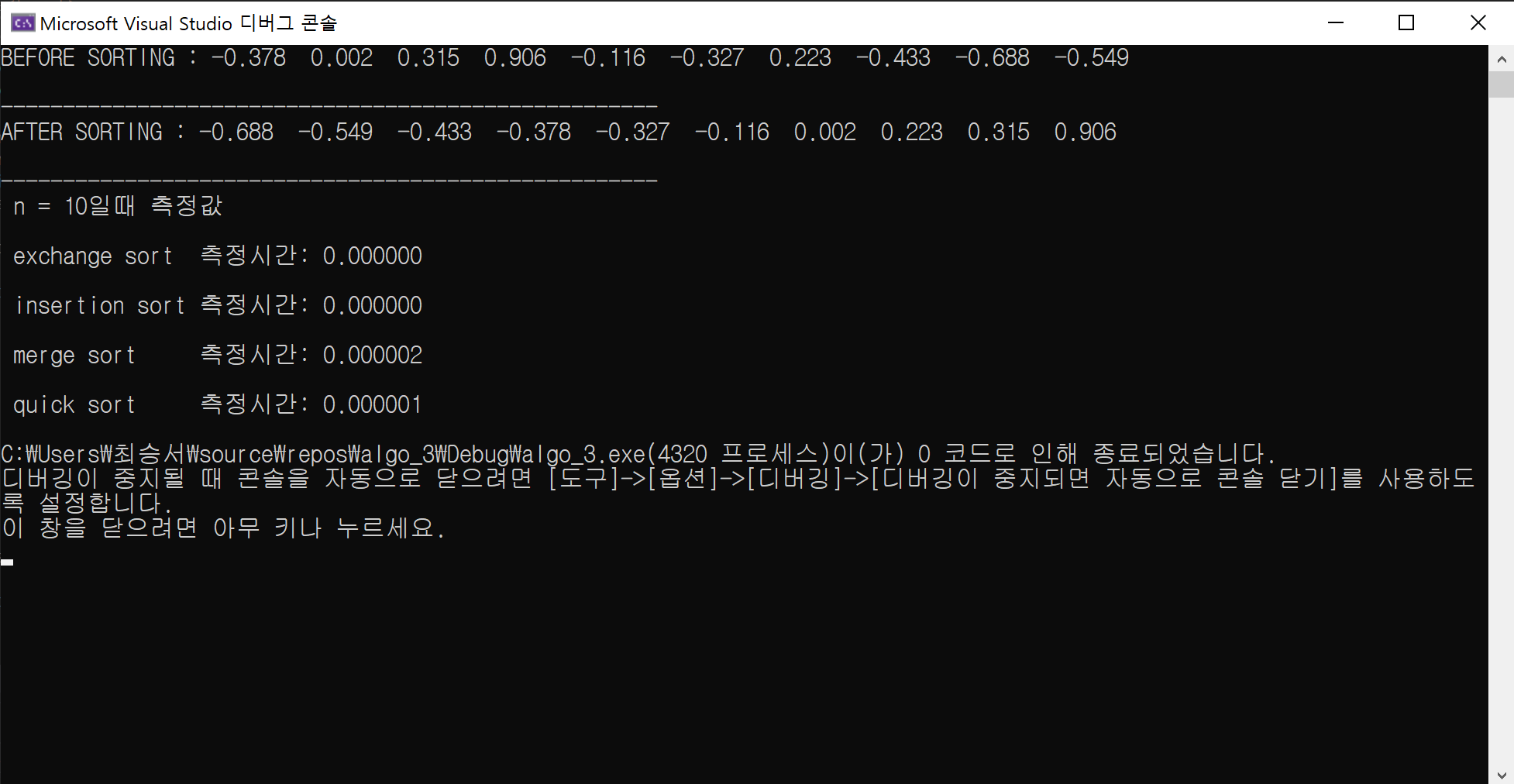
printf("\n merge sort 측정시간: %f\n", cost\_3);

printf("\n quick sort 측정시간: %f\n", cost\_4);

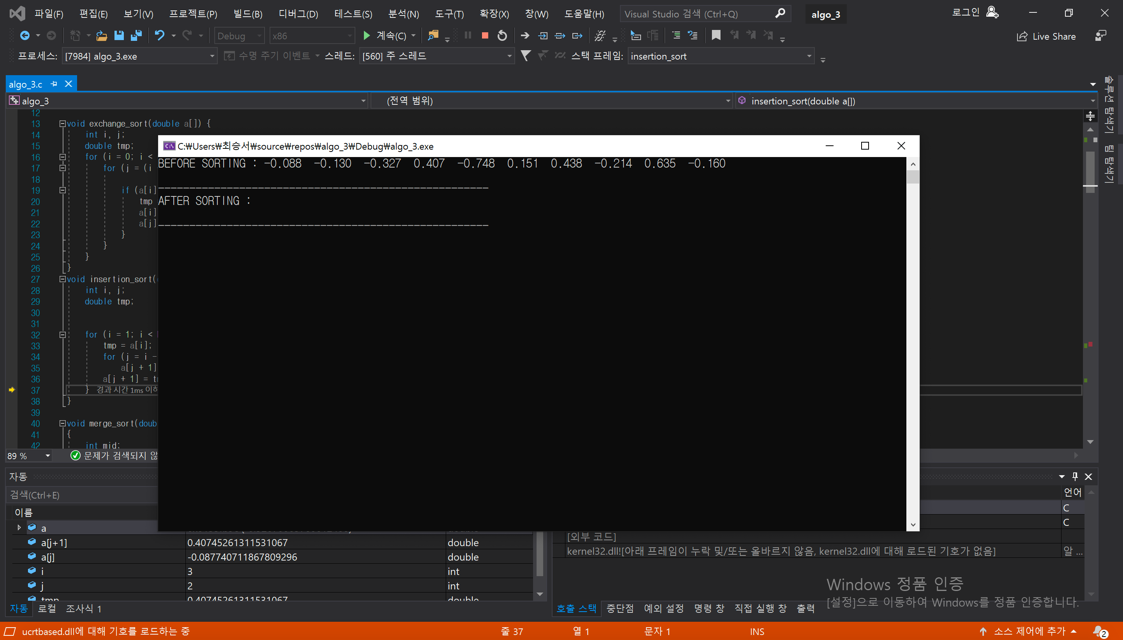
return 0;

}

**[실행 화면]**

****

**[디버거 사용화면]**

****

앞서 보여드린 코드를 통해 exchange, insertion, merge, quick sort를 n=10부터 n=5000 까지의 실행시간을 각각 비교하였습니다. 아래 그래프와 표를 보면 n이 1000 이하 일 때 까지는 서로 비슷한 시간이 소요되는데 그 이후부터는 각 정렬 알고리즘별로 차이가 발생하는 것을 볼 수 있습니다. 지금부터 가장 빠른 실행 속도를 보여준 정렬 알고리즘부터 설명하겠습니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | exchange | insertion | merge | quick |
| 10 | 0.000000 | 0.000000 | 0.000002 | 0.000001 |
| 100 | 0.000019 | 0.000007 | 0.000013 | 0.000008 |
| 200 | 0.000074 | 0.000024 | 0.000031 | 0.000018 |
| 300 | 0.000151 | 0.000099 | 0.000048 | 0.000028 |
| 400 | 0.000257 | 0.000101 | 0.000081 | 0.000037 |
| 500 | 0.000421 | 0.000499 | 0.000094 | 0.000047 |
| 750 | 0.000931 | 0.000453 | 0.000163 | 0.000111 |
| 1000 | 0.001616 | 0.000521 | 0.000399 | 0.000106 |
| 1500 | 0.006227 | 0.001535 | 0.000448 | 0.000153 |
| 2000 | 0.011956 | 0.002122 | 0.000764 | 0.000215 |
| 3000 | 0.018197 | 0.004879 | 0.001433 | 0.000322 |
| 4000 | 0.036969 | 0.008779 | 0.003557 | 0.000434 |
| 5000 | 0.065434 | 0.014444 | 0.004892 | 0.000553 |

먼저 가장 빠른 실행시간을 보여준 정렬 알고리즘은 Quick Sort입니다

**Quick Sort**는 분할 정복(Divide and Conquer)을 이용하여 정렬을 수행하는 알고리즘입니다. Pivot이라는 기준이 되는 값을 하나 설정하여, 이 값을 기준으로 작은 값은 왼쪽, 큰 값은 오른쪽으로 옮기는 방식으로 정렬을 진행합니다. 이런 과정을 반복하여 분할된 배열의 크기가 1이 되면 배열이 모두 정렬 된 것 입니다.

Quick Sort는 pivot을 기준으로 두 개의 부분 집합으로 나누어 자료의 위치를 교환하기 때문에 시간복잡도는 다음과 같습니다. **Ο()**

하지만 최악의 경우(worst case)가 존재하는데, 이미 배열이 정렬 되어있는 경우입니다. 이때의 시간복잡도는 Ο()입니다.

이때문에 Merge Sort보다 느린 알고리즘이라고 생각할 수 있지만, 발생하기 쉽지 않고, 보통 Quick Sort가 Merge Sort보다 20%정도 빠르다고 합니다.

두번째로 빠른 실행시간을 보여준 알고리즘은 Merge Sort입니다.

**Merge Sort**또한 분할 정복(Divide and Conquer)방식을 사용하는 알고리즘입니다. 하나의 배열을 두개의 배열로 계속 쪼개어 나간 뒤, 합치면서 정렬해 최후에는 하나의 정렬을 출력합니다.

정렬 과정은 두 배열 A,B를 정렬하기 때문에 A배열의 크기를 n1, B배열의 크기를 n2라고 했을 때

*)*가 됩니다. 이때 n=n1+n2이므로 *)* 라고 할 수 있습니다.

분할 과정은 만큼 반복되므로 Merge Sort의 시간복잡도는 **Ο()**이 됩니다.

그 다음으로 빠른 실행시간을 보여준 알고리즘은 Insertion Sort입니다.

**Insertion Sort**는 현재 위치에서 그 이하의 배열들을 비교하여 자신이 들어갈 위치를 찾아 그 위치에 삽입하는 정렬 알고리즘입니다.

최악의 경우(worst case)는 거꾸로 정렬이 되어 있을 때 인데, n-1개, n-2개,….,1개씩 반복 비교하여 시간복잡도는 상한을 기준으로 하는 Big-O는 최악의 경우를 기준으로 평가하므로 시간복잡도는 이 된다

마지막으로 가장 긴 실행시간을 보여준 알고리즘은 Exchange Sort입니다.

**Exchange Sort**는 처음 배열과 다음 배열을 비교하여 순서가 잘못되어 있으면 바꾸고, 그렇지 않으면 처음 배열과 그 다음 배열을 비교하는 방법으로 모든 배열에 대한 검색을 수행하는 가장 기본적인 정렬 알고리즘입니다. 모든 대상 항목(배열)을 비교하기 때문에 정렬 알고리즘중에서 가장 많은 시간이 소요됩니다.

한번 비교할 때 하나 이상의 역은 제거할 수 없으므로 시간복잡도가 최악의 경우(worst case) **)**, 평균적으로도 **)** 보다 좋을 수 없습니다.

이번 비교를 통해 4개의 정렬 알고리즘 중에 가장 빠른 알고리즘은 Quick Sort이고 그 다음으로 Merge Sort, Insertion Sort, 가장 느린 알고리즘은 Exchange Sort인 것을 확인했습니다.