알고리즘 과제 보고서

과제 3. 정렬 알고리즘 비교

제출일 : 2019년 4월 7일

학번 : 201501376

이름 : 박지수

목차

1.문제 정의

0. 사용할 데이터 개수와 배열, 데이터의 이동 비교회수를 세기 위한 변수 등 선언

    1) original[~~], a[~~]

    2) count, move

1. 선택 정렬

    1) 코드 작성

    2) SWAP 함수 구현

2. 버블 정렬

    1) 코드 작성

3. 삽입 정렬

    1) 코드 작성

4. 쉘 정렬

    1) insertionSort2 함수 구현

    2) 쉘 함수 코드 작성

5. 병합 정렬

    1) merge 함수 구현

    2) 병합 정렬 코드 작성

    3) 데이터를 저장할 또 하나의 배열

6. 퀵 정렬

1) partition 함수 구현

2) Loop 를 돌릴 때 사용할 반복문의 종류 선택

3) 퀵 정렬 코드 작성

7. 주어진 데이터 배열을 중복되는 데이터 없이 무작위로 섞기

1) Shuffle 함수 구현

8. 배열 한 개를 선언해서 6개 정렬을 한꺼번에 테스트할 수 있는 방법 - 배열 복사

1) 원본 데이터 배열과 Shuffle돌린 후의 배열

2) copyArr 함수 구현

9. 각 정렬 함수 들에서 move, compare 회수를 증가시켜야 하는 위치 찾기

10. move, compare 를 출력하는 방법

2.해결 방안

0. main 함수 위에 전역 변수 선언 공간에 배치

1) #define MAX 를 사용하여 입력할 원소의 개수를 정의

2) int original[MAX] : for 문을 사용하여 데이터가 순서대로 들어가 있는 원본 배열

3) int a[MAX] : Shuffle을 돌리고 난 후 무작위 배치 되어있는 정렬에 사용할 배열

1. 선택 정렬

1) 알고리즘을 보고 코드 그대로 작성

2) SWAP 함수는 여러 번 써야 하고 코드도 간단하니 전역함수로 선언

- 서로 바꿀 원소 2개와 저장할 원소 한 개 총 3개의 변수가 필요

2. 버블 정렬

1) 알고리즘을 보고 코드 그대로 작성

3. 삽입 정렬

1) 알고리즘을 보고 코드 그대로 작성,

i : 삽입을 하려고 하는 원소의 인덱스, pos : 맨 뒤부터 비교하는 원소의 인덱스

4. 쉘 정렬

1) 삽입 정렬의 insertionSort 를 참고하여 insertSort2 함수를 구현한다.

다만 부분집합의 기준이 되는 간격을 매개변수 h에 저장한다.

2) 단계가 수행 될 때마다 h를 반으로 감소시키고, h의 값이 1이 될 때까지 insertionSort2를 순환 호출한다.

5. 병합 정렬

1) merge - 데이터 배열의 가장 왼쪽에 있는 원소를 가리키는 변수 m, 가장 오른쪽에 있는 원소를 가리키는 변수 n, 가운데 원소를 가리키는 매개변수 mid를 선언

- 데이터를 비교할 때 사용할 변수 i, j, k, l 을 선언

- 앞에서부터 하나하나 비교해서 병합, 두 목록에서 작은 원소를 차례대로 빼낸다.

- 각 단계에서 병합하여 만든 부분집합을 저장할 또 하나의 배열이 필요 -> sorted[~] 배열을 전역 공간에 선언 << 문제 3)

2) mergeSort - 배열의 원소가 한 개 이상이면 두개의 부분집합으로 분할

- mid는 원소의 가운데 값으로 설정

- mergeSort 를 왼쪽 부분집합, 오른쪽 부분집합에서 각각 순환호출

- 두 mergeSort한 두 부분집합을 merge(병합)

6. 퀵 정렬

1), 2) partition – 강의자료에 나와있는 알고리즘만 보고 짰더니, 테스트 결과가 어느 경우에는 정상적으로 정렬되는 반면, 루프를 빠져나오지 못하는 경우도 생기고, 정렬이 완료되지 않는 경우도 발생하여 do - while 문으로 구현하였다

가장 왼쪽의 원소를 pivot 으로 지정, 값을 찾는 과정을 수행하고 새로운 pivot의 위치를 리턴

3) 조건을 확인하여 partition 함수로부터 리턴받은 새 pivot 을 기준으로 좌우 부분집합에 대해 퀵 정렬 알고리즘을 재귀 호출

7. Suffle 함수

1) 단순히 stdlib.h 헤더의 rand() 만 써서는, 난수를 생성하는 규칙이 같아서 항상 같은 배열이 나오게 된다

따라서 time.h 헤더를 사용, 현재 시각을 시드로 난수를 생성하는 방법을 택하였다 (srand)

8. 배열 한 개를 선언해서 6개 정렬을 한꺼번에 테스트할 수 있는 방법 - 배열 복사

1) 원본 데이터 배열을 만든 후 따로 저장한다

이후 각 정렬에서 사용할 배열은 그 복사본을 사용한다

2) copyArr 함수 구현

원본 배열을 실제 사용할 배열로 복사하는 함수

오류 수정 과정에서 원본 데이터 값에서 1씩 증가시킴

9. move와 compare의 위치

1) – if, while문의 경우 조건이 안 맞을 때 까지 실행되니 if, while 위에서 compare 수행,

- SWAP의 경우 move 2번, 위치는 데이터 이동 바로 직전에서 수행

- for 문의 index 비교는 제외

10. move 와 compare 출력

1) - 전역변수로 선언된 move와 compare를 각각 정렬이 실행되기 전 0으로 초기화

- 하나의 정렬이 끝나면 각각 출력하고 다음 정렬을 실행하기 전 다시 0으로 초기화

3. 구현

```
// 201501376 박지수

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

//배열 원소의 개수를 정의
#define MAX 10000
//SWAP함수
#define SWAP(x,y,t) ((t)=(x), (x)=(y), (y)=(t))
//원본 배열
int original[MAX];
//사용 데이터 배열
int a[MAX];
//데이터 개수를 받을 변수
int n;
//합병정렬에서 사용할 데이터를 저장할 배열
int sorted[MAX];
//데이터의 이동,비교회수를 세기 위한 변수
int move, compare;


//선택 정렬
void selectionSort(int a[], int n)
{
        int i, j, min, tmp;

        printf("₩n");
        printf("선택 정렬₩n");
        for (i = 0; i<n - 1; i++)
        {
                min = i;
                for (j = i + 1; j < n; j++) {
```

```
                        compare++;
                        if (a[j] < a[min]) {
                                min = j;
                        }
                }
                move += 2;
                SWAP(a[i], a[min], tmp);
        }
}


//버블 정렬
void bubbleSort(int a[], int n)
{
        int i, j, tmp;

        printf("버블 정렬₩n");
        for (i = n - 1; i>0; i--)
        {
                for (j = 0; j < i; j++) {
                        compare++;
                        if (a[j] > a[j + 1]) {
                                move += 2;
                                SWAP(a[j], a[j + 1], tmp);
                        }
                }
        }
}


//삽입 정렬
void insertionSort(int a[], int n)
{
        int i, pos, val;

        printf("삽입 정렬₩n");
        for (i = 1; i<n; i++)
        {
                val = a[i];
                for (pos = i; pos > 0 ; pos--) {
                        compare++;
                        if (val < a[pos - 1]) {
                                move++;
                                a[pos] = a[pos - 1];
                        }
                        else
                                break;
                }
                move++;
                a[pos] = val;
        }
}


//쉘 정렬 insertionSort2
```

```
insertionSort2(int a[], int first, int last, int h)
{
	int i, pos, val;

	for (i = first + h; i <= last; i = i + h) {
		val = a[i];
		for (pos = i; pos > first; pos -= h) {
			compare++;
			if (val < a[pos - h]) {
				move++;
				a[pos] = a[pos - h];
			}
			else
				break;
		}
		move++;
		a[pos] = val;
	}
}

//쉘 정렬
void shellSort(int a[], int n)
{
	int i, h;

	for (h = n / 2; h>0; h = h / 2) {
		for (i = 0; i < h; i++) {
			insertionSort2(a, i, n - 1, h);
		}
	}
}


 //병합 정렬 merge
void merge(int a[], int m, int mid, int n)
{

	int i, j, k, l;
	i = m, j = mid + 1; k = m;


	while (i <= mid && j <= n)
	{
		compare++;
		compare++;
		if (a[i] <= a[j]) {
			move++;
			sorted[k++] = a[i++];
		}
		else {
			move++;
			sorted[k++] = a[j++];
		}
	}
```

```
        compare++;
        if (i > mid) {
                for (l = j; l <= n; l++) {
                        sorted[k++] = a[l];
                }
        }
        else
                for (l = i; l <= n; l++) {
                        sorted[k++] = a[l];
                }

        for (l = m; l <= n; l++)
                a[l] = sorted[l];
}
//병합정렬
void mergeSort(int a[], int m, int n)
{
                int mid;

                compare++;
                if (m < n)
                {
                        move++;
                        mid = (m + n) / 2;
                        mergeSort(a, m, mid);
                        mergeSort(a, mid + 1, n);
                        merge(a, m, mid, n);
                }
}

//퀵정렬 피봇을 기준으로 분할하는 함수
int partition(int a[], int begin, int end)
{

        int pivot = a[begin], tmp, low = begin, high = end + 1;

        do {
                compare++;

                do {
                        compare++;
                        low++;
                }
                while (low <= end && a[low] < pivot);


                do {
                        compare++;
                        high--;
                }
                while (high >= begin && a[high]>pivot);

                compare++;
                if (low < high) {
                        move += 2;
```

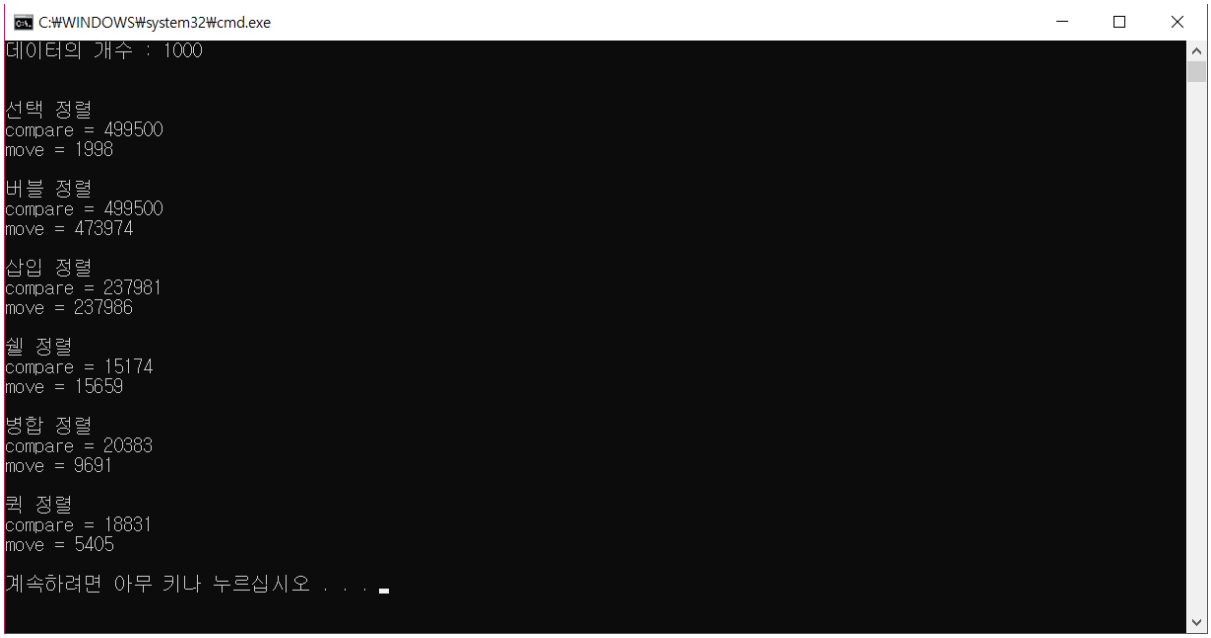
```
                        SWAP(a[low], a[high], tmp);
                }
        } while (low<high);

        move += 2;
        SWAP(a[begin], a[high], tmp);
        return high;
}

//퀵 정렬
void quickSort(int a[], int begin, int end)
{
        compare++;
        if (begin<end)
        {
                move++;
                int pivot = partition(a, begin, end);
                quickSort(a, begin, pivot - 1);
                quickSort(a, pivot + 1, end);
        }

}

//배열 원소를 섞는 함수 (셔플)
 void Shuffle(int *arr, int num) {

          srand(time(NULL));

          int temp; int rn;

          for (int i=0; i < num; i++) {
                   rn = rand() % (num - i) + i;
                   temp = arr[i];
                   arr[i] = arr[rn];
                   arr[rn] = temp; }
 }


//원본 배열을 복사하는 함수
void copyArr(void)
{
        int i;
        for (i = 0; i<n; i++)
                a[i] = original[i]+1;
}


//메인함수
void main(void)
{
        int i;
        srand((unsigned int)time(NULL));
        n = MAX;
        for (i = 0; i <n; i++)
                original[i] = i;
```

```
Shuffle(original, n);



printf("데이터의 개수 : %d₩n₩n", n);

copyArr();
move = 0;
compare = 0;
selectionSort(a, n);
printf("compare = %d₩n", compare);
printf("move = %d₩n", move);
printf("₩n");


copyArr();
move = 0;
compare = 0;
bubbleSort(a, n);
printf("compare = %d₩n", compare);
printf("move = %d₩n", move);
printf("₩n");


copyArr();
move = 0;
compare = 0;
insertionSort(a, n);
printf("compare = %d₩n", compare);
printf("move = %d₩n", move);
printf("₩n");


copyArr();
move = 0;
compare = 0;
printf("쉘 정렬₩n");
shellSort(a, n);
printf("compare = %d₩n", compare);
printf("move = %d₩n", move);
printf("₩n");



copyArr();
move = 0;
compare = 0;
printf("병합 정렬₩n");
mergeSort(a, 0, n);
printf("compare = %d₩n", compare);
printf("move = %d₩n", move);
printf("₩n");
```

```
        copyArr();
        printf("퀵 정렬₩n");
        move = 0;
        compare = 0;
        quickSort(a, 0, n);
        printf("compare = %d₩n", compare);
        printf("move = %d₩n", move);
        printf("₩n");

}
```

4.테스트

데이터 개수가 1000개 일 때 cmd 화면



비교

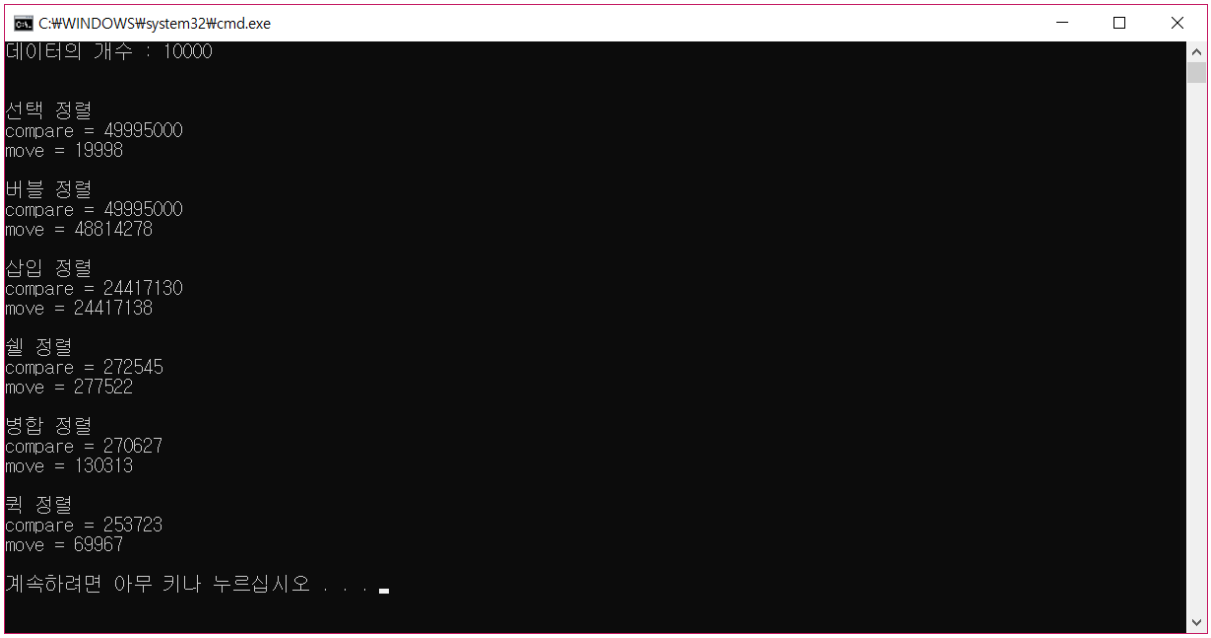
무작위데이터수	Selection	Bubble	Insertion	Shell	Merge	Quick
1000 개	499500	499500	247788	14524	20420	19360
2000 개	1999000	1999000	987894	35669	44987	41694
3000 개	4498500	4498500	2212023	56982	70827	66358

4000 개	7998000	7998000	3938087	84092	97821	95859
5000 개	12497500	12497500	6187340	115142	125541	121639
6000 개	17997000	17997000	8939794	142568	153703	143695
7000 개	24496500	24496500	12213601	157877	182267	170753
8000 개	31996000	31996000	15702154	202683	211329	198161
9000 개	40495500	40495500	19899609	224825	241039	247241
10000 개	49995000	49995000	24764272	270232	270805	255684

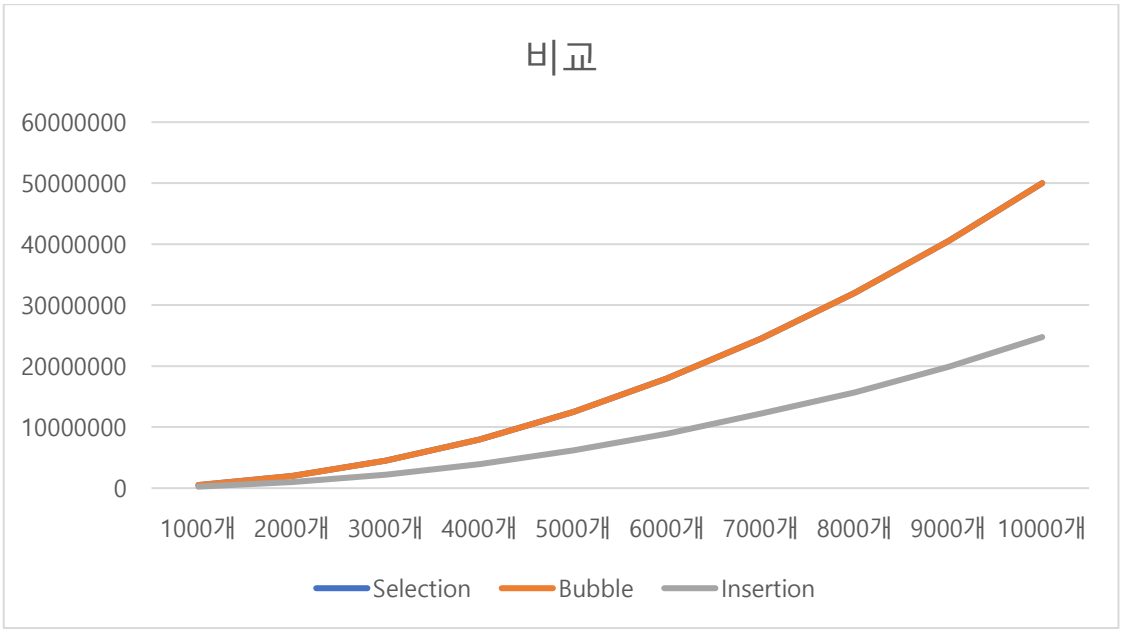
이동

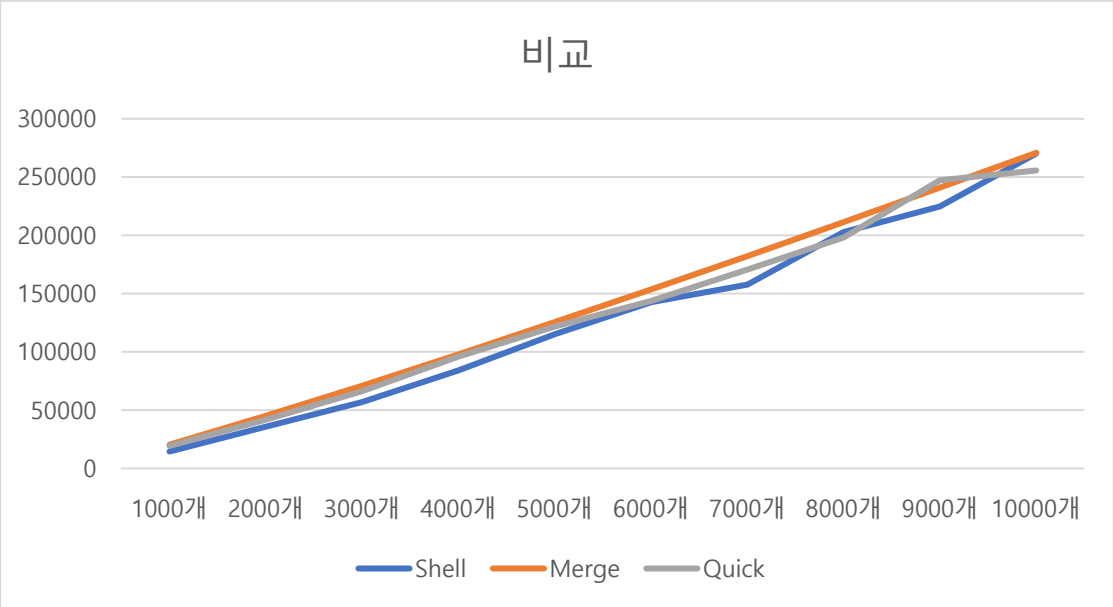
무작위데이터수	Selection	Bubble	Insertion	Shell	Merge	Quick
1000 개	1998	496246	249122	14875	9702	5458
2000 개	3998	1961376	982687	36674	21493	11728
3000 개	5998	4303408	2154703	58511	33913	18521
4000 개	7998	7862836	3935417	86058	46910	25202
5000 개	9998	12581592	6295795	117753	60270	32301
6000 개	11998	17805814	8908906	145618	73851	39639
7000 개	13998	24107618	12060808	161430	87633	47248
8000 개	15998	31388340	15702169	206680	101664	54480
9000 개	17998	39781248	19899623	229455	116019	61615
10000 개	19998	48578248	24299123	275237	130402	69505

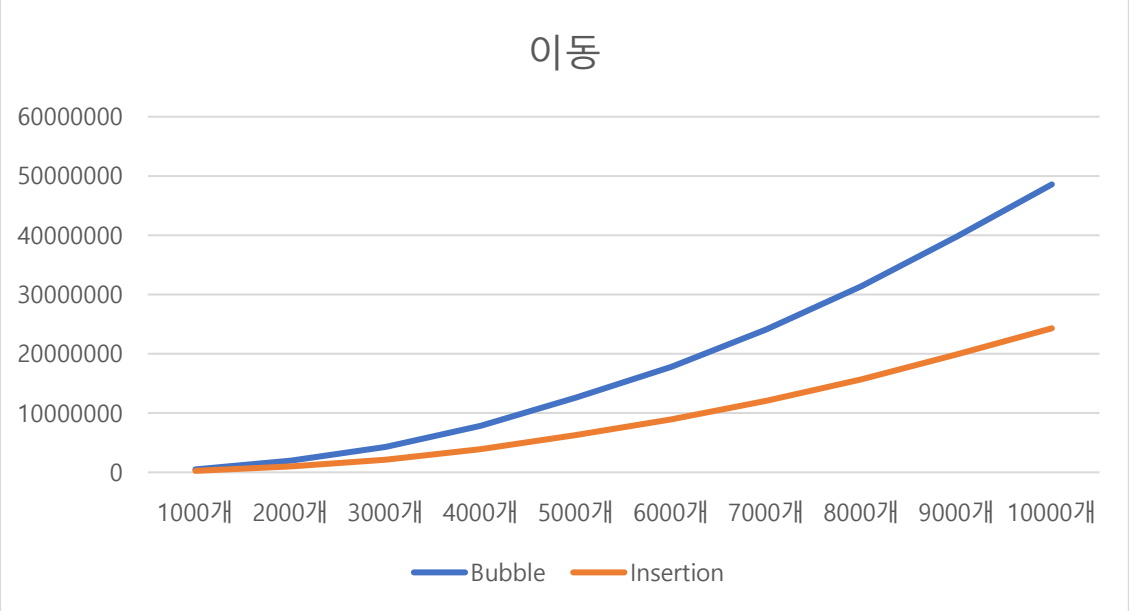
데이터 개수가 10000개 일 때 cmd 화면

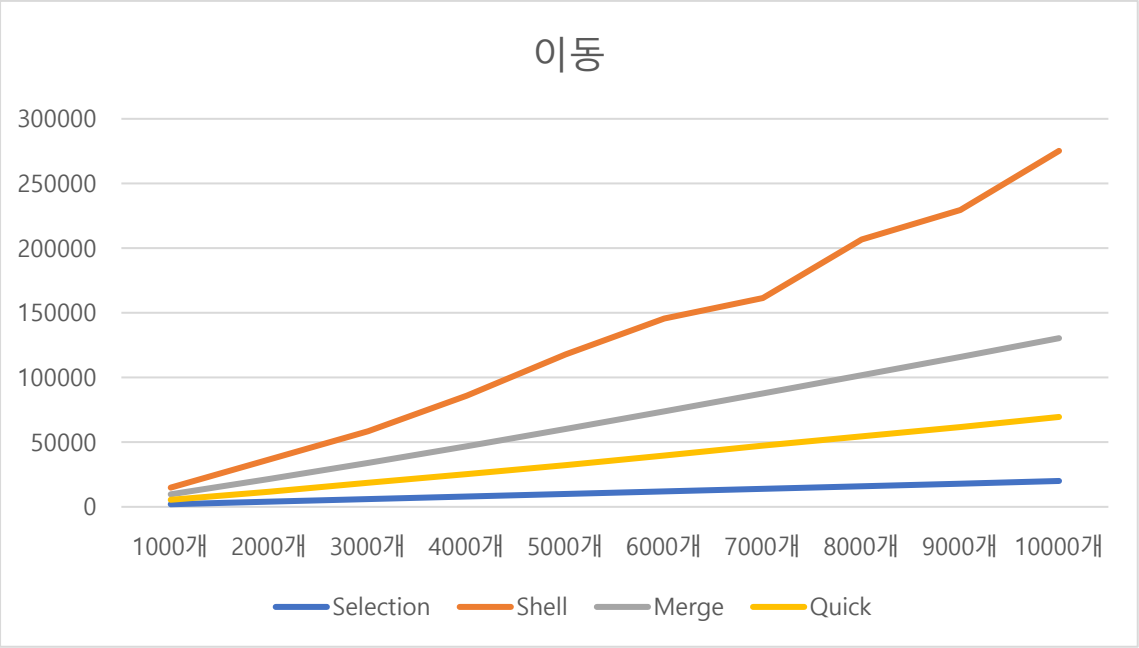


(선택과 버블 그래프가 동일)




비교
300000
250000
200000
150000
100000
50000
0
1000개
2000개
3000개
4000개
5000개
6000개
7000개
8000개
9000개
10000개
Shell
Merge
Quick


이동
60000000
50000000
40000000
30000000
20000000
10000000
0
1000개
2000개
3000개
4000개
5000개
6000개
7000개
8000개
9000개
10000개
Bubble
Insertion



5.결론

1) 실행 결과에 따른 결론

(1) - 선택, 버블, 삽입 정렬의 시간복잡도는 O(n^2) 이다.

실행 결과로 봤을 때도 다른 세 정렬과는 비교,이동 회수가 자릿수를 달리한다.

데이터 10000개 기준으로, 비교와 이동을 합한 연산회수는 선택, 버블, 삽입 정렬 각각 약 5천만회, 1억회, 5천만회 이다.

이는 10000^2 = 1억 임을 감안한다면 타당한 결과를 얻었다고 볼 수 있다.

데이터가 증가할수록 각 정렬 별 비교,이동 회수의 차이가 기하급수적으로 늘어나 동시에 같은 그래프에서 표현이 힘들다.

(2) - 쉘 정렬의 시간복잡도는 O(n^1.25) ~ O(n^1.5) 이다.

데이터 10000개 기준으로, 비교와 이동을 합한 연산회수는 약 560288회,

10000^1.25 ~ 10000^1.5 = 100000 ~ 1000000, 10만에서 100만이다.

따라서 실행 결과와 일치한다고 볼 수 있다.

(3) - 병합 정렬의 시간 복잡도는 O(nlogn) 이다 (log밑은 2)

데이터 10000개 기준으로, 비교와 이동을 합한 연산회수는 약 401330회,

10000log(10000) = 132877, 자리 수를 벗어나지 않으므로, 예상한 결과와 비슷한 값이 나왔다고 볼 수 있다.

(4) - 퀵 정렬의 시간 복잡도는 최악의 경우에는 O(n^2) 이지만, 수행 결과에서 확인할 수 있듯이 다른 정렬보다 move의 수행 회수가 적다 따라서 O(nlogn) 라고 할 수 있다.

데이터 10000개 기준으로, 비교와 이동을 합한 연산회수는 약 329187회, 시간복잡도는 병합 정렬과 같으나, 연산회수가 적다. 따라서 6가지 정렬 알고리즘 중에 가장 빠르다고 할 수 있다.

Move 회수만 보면 선택 정렬이 가장 우수하나, 선택 정렬은 compare 회수가 가장 크다.

2) 보고서 수행 과정에 대한 후기

일단 시간이 많이 걸렸다.

단순히 ppt에 나와있는 psedo code대로 코드를 짜면 되는 것 도 있고, 어떤 알고리즘은 제대로 실행되지 않았다. (퀵 정렬)

처음 프로그램의 구조를 설계할 때 많이 고민했고 하나를 고치면 멀쩡하던 다른 하나가 다시 어긋나는 경우가 있어서 시간이 많이 소요되었다.

실제 데이터 배열 [1,2,3,4,5,6,7,8,9,10] 을 넣어서 섞은 다음 제대로 정렬이 되는지는 직접 확인 하였지만, compare와 move를 추가하는 과정에서, 논리 오류가 발생했는지 검산은 하지 못하였다.

단적으로 삽입 정렬과 쉘 정렬의 move, compare의 결과가 조금 의심스럽다.

삽입 정렬의 경우, DATA SET이 정방향으로 이미 정렬되어 있는 경우는, compare는 하지만,

move는 일어나지 않는다. 이 경우 move의 회수는 0이 나와야 할 거 같은데, 이를 알아채고 검산을 하기에는 남은 시간이 부족하였다.

알고리즘별 연산 회수가 이렇게 차이가 나는지 직접 프로그램을 돌려보고 나서야 체감이 되었다. 실제로 한번 데이터 개수를 10만개를 입력했더니, i7 8th Gen 임에도 불구하고 시간복잡도가

O(n^2)인 정렬 알고리즘들은 결과가 나오기까지 수 초가 걸렸다. 반면 시간복잡도가 작은 알고리즘들은 1초도 걸리지 않았다.

이에 이렇게 단순하고 데이터 입력 개수도 적은 프로그램 임에도 불구하고 차이가 나는 것을 감안한다면, 프로그램이 커지고 처리하는 데이터 개수가 무한정 할 때 빠른 알고리즘의 중요성은 매우 크다는 것을 다시 한번 느꼈다.