알고리즘 및 실습 과제 (정렬 보고서)

20011619

보고서 목적

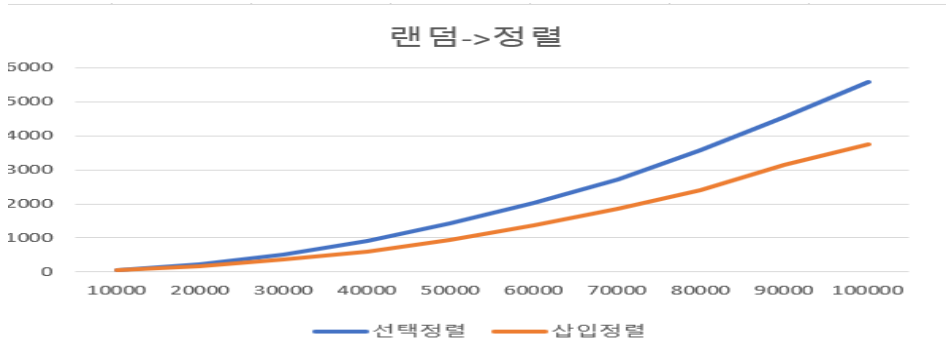
이 보고서의 목적은 선택 정렬과 삽입 정렬이 다양한 초기 배열 상태(랜덤, 정렬된 상태, 역순 상태)에서 배열 크기 n을 증가시키면서 실행되는 과정에서 실행 시간의 변화를 분석하고 성능을 비교하는 것입니다.

실험 결과

#A 각 정렬의 입력으로 정렬이 안 된 랜덤 데이터가 주어지는 경우

A	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
선택정렬	59.83	227.79	507.2	901.95	1415.12	2032.79	2719.24	3562.78	4551.61	5586.09
삽입정렬	44.32	159.97	360.16	607.86	941.15	1364.02	1852.43	2398.18	3155.41	3751.66

표

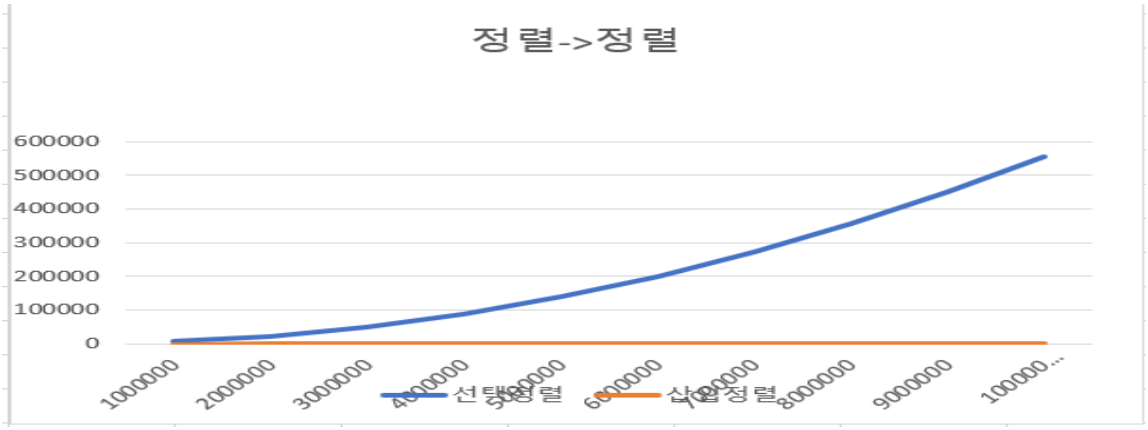


그래프

#B 각 정렬의 입력으로 정렬된 데이터가 주어지는 경우

B	1000000	2000000	3000000	4000000	5000000	6000000	7000000	8000000	9000000	10000000
선택정렬	5440.52	22168.69	49333.24	88742.83	138786.8	199959.7	272344.9	357151.9	450783.7	556468.4
삽입정렬	2.01	5.67	8.92	11.32	17.34	14.63	32.17	28.12	25.74	22.12

표

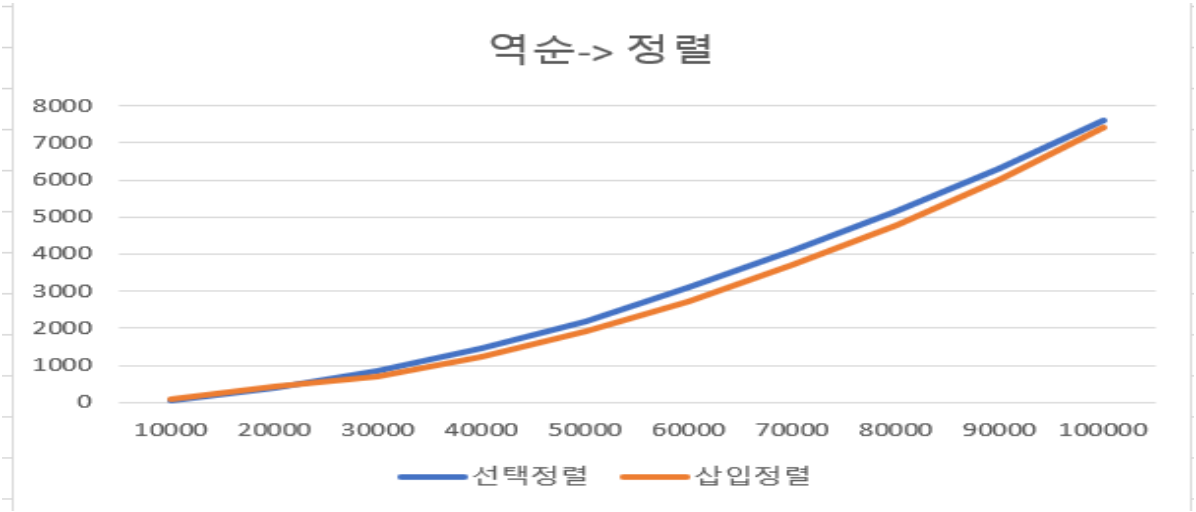


그래프

#C 각 정렬의 입력으로 역순으로 정렬된 데이터가 주어지는 경우

C	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
선택정렬	63.1	401.56	842.8	1442.45	2184.43	3097.84	4099.43	5143.57	6318.53	7602.08
삽입정렬	80.86	431.25	683.26	1232.93	1907.81	2702.69	3701.31	4792	6016.47	7417.56

표

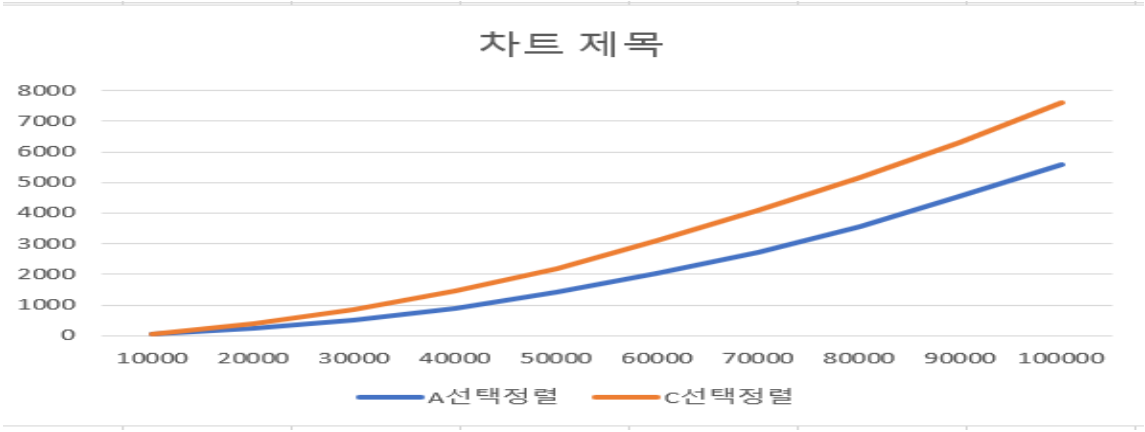


그래프

#A-C 선택정렬

A-C 선택	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
A선택정렬	59.83	227.79	507.2	901.95	1415.12	2032.79	2719.24	3562.78	4551.61	5586.09
C선택정렬	63.1	401.56	842.8	1442.45	2184.43	3097.84	4099.43	5143.57	6318.53	7602.08

표

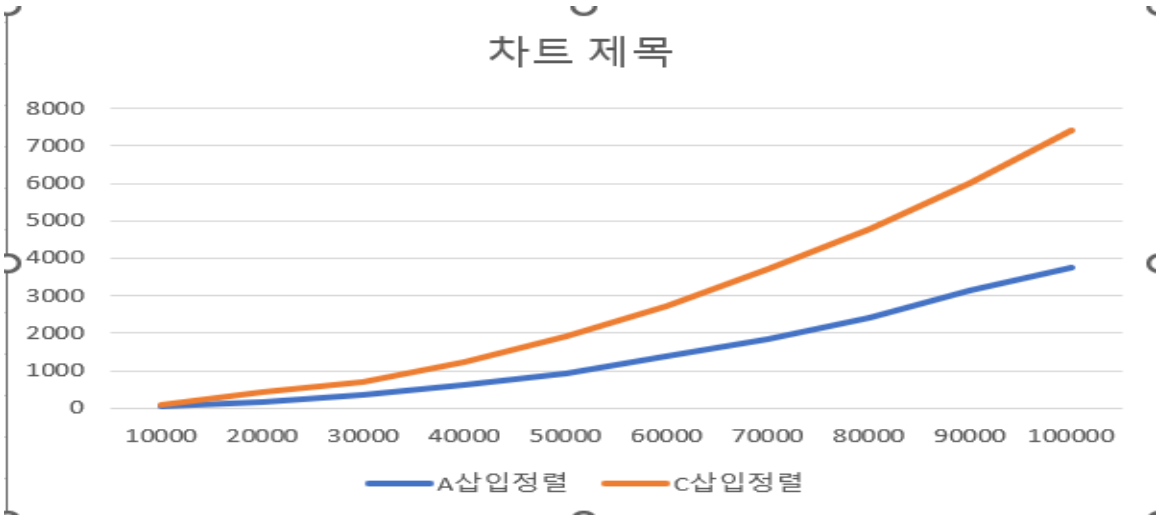


그래프

#A-C 삽입정렬

A-C 삽입	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
A삽입정렬	44.32	159.97	360.16	607.86	941.15	1364.02	1852.43	2398.18	3155.41	3751.66
C삽입정렬	80.86	431.25	683.26	1232.93	1907.81	2702.69	3701.31	4792	6016.47	7417.56

표



그래프

분석

-그래프 및 표 분석

1. A,B,C 결과를 통하여 선택정렬은 O(n^2)의 시간 복잡도를 가진다고 분석 할 수 있다.

2. A,B,C 결과를 통하여 삽입정렬은 입력 상태에 따라 다른 시간 복잡도를 가지며 정렬 된 상태인 경우(B) O(n), 정렬 되어 있지 않은 경우(A,C) O(n^2)의 시간 복잡도를 가진다고 분석할 수 있다.

3. A-C 선택정렬, A-C 삽입정렬 표와 그래프를 통하여 두 정렬 모두 C의 경우 보다 A의 경우 실행시간이 더 빠르다고 분석할 수 있다.

2-1선택 정렬과 삽입 정렬이 B의 경우 달라지는 이유 분석

삽입 정렬은 배열을 순회하면서 현재 원소를 정렬된 부분 배열에 삽입하는 원리를 가진다. 이미 정렬된 배열의 경우, 대부분의 원소는 이미 정렬된 상태이며 뒤로 이동할 필요가 없습니다. 그러나 선택정렬은 항상 배열에서 가장 작은(또는 큰)원소를 찾아 정렬된 부분 배열의 맨 앞에 삽입합니다. 이미 정렬된 배열에서도 선택 정렬은 모든 원소를 순회하면서 선택 연산을 수행하므로 정렬된 배열에서도 시간 복잡도가 항상 O(n^2)으로 유지된다.

따라서 이미 정렬된 배열의 경우 삽입 정렬이 선택 정렬보다 더 빠르게 동작한다.

3-1 A의 경우 더 빠른 이유

삽입 정렬

삽입 정렬은 역순으로 정렬된 배열에서는 현재 원소를 삽입하기 위하여 뒤로 많은 이동 연산을 수행해야한다. 삽입 정렬은 현재 원소를 정렬된 부분 배열의 적절한 위치에 삽입하기 위해 뒤로 이동해야 하는 횟수가 역순 배열에서 최대가 된다. 이로 인해 이동 연산의 수가 많아지고 실행 시간이 증가한다.

선택 정렬

역순으로 정렬된 배열에서는 최솟값을 찾는 과정이 가장 복잡하다. 이 배열에서는 최솟값이 맨 뒤에 있기 때문에 모든 반목문의 모든 인덱스마다 최솟값을 초기화 해주어야 하므로 선택 연산의 수가 많아져 실행 시간이 증가한다.

실험에 사용한 코드

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include<time.h>
#include <Windows.h>
void reverse_Insertion_Sort(int arr[], int n) {//배열을 역순으로 정렬
	int i, j, key;

	for (i = 1; i < n; i++) {
		key = arr[i];
		j = i - 1;

		while (j >= 0 && arr[j] < key) {
			arr[j + 1] = arr[j];
			j = j - 1;
		}
		arr[j + 1] = key;
	}
}
void insertion_sort(int arr[], int n) {//삽입 정렬
	for (int pass = 1; pass <= n - 1; pass++) {
		int save = arr[pass];
		int j = pass - 1;
		while ((j >= 0) && (arr[j] > save)) {
			arr[j + 1] = arr[j];
			j = j - 1;
		}
		arr[j + 1] = save;
	}
}
void selection_sort(int arr[], int n) {//선택정렬
	for (int pass = 0; pass <= n - 2; pass++) {
		int minLoc = pass;
		for (int j = pass + 1; j <= n - 1; j++) {
			if (arr[j] < arr[minLoc])
				minLoc = j;
		}
		int tmp = arr[pass];
		arr[pass] = arr[minLoc];
		arr[minLoc] = tmp;
	}
}
int main() {

	int n, a;

	srand(time(NULL));
	LARGE_INTEGER ticksPerSec;//시간 측정을 위한 함수
	LARGE_INTEGER start, end, diff;
	QueryPerformanceFrequency(&ticksPerSec);
```

```
double elapsedTimeA;
double elapsedTimeB;
//기본
printf("A의 n값 : ");
scanf("%d", &n);
int* A = (int*)malloc(sizeof(int) * n);
int* B = (int*)malloc(sizeof(int) * n);
for (int i = 0; i < n; i++) {
	a = rand() % 10001;
	*(A + i) = a;
	*(B + i) = a;
}
printf("A기본\n");
QueryPerformanceCounter(&start);
selection_sort(A, n);
QueryPerformanceCounter(&end);

diff.QuadPart = end.QuadPart - start.QuadPart;
double elapsedTimeA = (double)diff.QuadPart / (double)ticksPerSec.QuadPart * 1000.0;
printf("선택정렬 : %.2lf ms\n", elapsedTimeA);

QueryPerformanceCounter(&start);
insertion_sort(B, n);
QueryPerformanceCounter(&end);

diff.QuadPart = end.QuadPart - start.QuadPart;
double elapsedTimeB = (double)diff.QuadPart / (double)ticksPerSec.QuadPart * 1000.0;
printf("삽입정렬 : %.2lf ms\n", elapsedTimeB);

//B 실험

printf("B의 n값 : ");
scanf("%d", &n);
int* A = (int*)malloc(sizeof(int) * n);
int* B = (int*)malloc(sizeof(int) * n);
for (int i = 0; i < n; i++) {
	a = rand() % 10001;
	*(A + i) = a;
	*(B + i) = a;
}
printf("fin");

printf("이미 정렬된 데이터B\n");
insertion_sort(A, n);//실험 전 정렬
insertion_sort(B, n);

QueryPerformanceCounter(&start);
selection_sort(A, n);
QueryPerformanceCounter(&end);

diff.QuadPart = end.QuadPart - start.QuadPart;
```

```
    elapsedTimeA = (double)diff.QuadPart / (double)ticksPerSec.QuadPart * 1000.0;
    printf("선택정렬 : %.2lf ms\n", elapsedTimeA);

    QueryPerformanceCounter(&start);
    insertion_sort(B, n);
    QueryPerformanceCounter(&end);

    diff.QuadPart = end.QuadPart - start.QuadPart;
    elapsedTimeB = (double)diff.QuadPart / (double)ticksPerSec.QuadPart * 1000.0;
    printf("삽입정렬 : %.2lf ms\n", elapsedTimeB);


    //C 실험

    printf("C의 n값 : ");
    scanf("%d", &n);
    int* A = (int*)malloc(sizeof(int) * n);
    int* B = (int*)malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++) {
        a = rand() % 10001;
        *(A + i) = a;
        *(B + i) = a;
    }
    printf("역순정렬 C\n");
    reverse_Insertion_Sort(A, n);//실험 전 역 정렬
    reverse_Insertion_Sort(B, n);

    QueryPerformanceCounter(&start);
    selection_sort(A, n);
    QueryPerformanceCounter(&end);

    diff.QuadPart = end.QuadPart - start.QuadPart;
    elapsedTimeA = (double)diff.QuadPart / (double)ticksPerSec.QuadPart * 1000.0;
    printf("선택정렬 : %.2lf ms\n", elapsedTimeA);

    QueryPerformanceCounter(&start);
    insertion_sort(B, n);
    QueryPerformanceCounter(&end);

    diff.QuadPart = end.QuadPart - start.QuadPart;
    elapsedTimeB = (double)diff.QuadPart / (double)ticksPerSec.QuadPart * 1000.0;
    printf("삽입정렬 : %.2lf ms\n", elapsedTimeB);

    return 0;
}
```