알고리즘 및 실습 과제 #1

<선택 정렬과 삽입 정렬의 시간복잡도 실험>

컴퓨터공학과 18011575 정상헌
알고리즘 및 실습 [001분반]

1. 실험 목적

무작위 값이 주어진 배열, 정렬된 배열, 역순으로 정렬된 배열에 대하여 선택 정렬과 삽입 정렬의 시간을 측정하여 두 정렬의 성능을 비교하고 실험 결과를 분석하고자 한다.

2. 실험 원리와 이론

선택 정렬

만약 n마리의 동물들을 키가 작은 순으로 나열한다고 가정해보자.
가장 작은 동물을 조사하는데 소요되는 비용은 n
그 다음으로 작은 동물을 조사하는데 소요되는 비용은 n-1
...
마지막으로 작은 동물을 조사하는데 소요되는 비용은 1이다.
이 비용은 동물들이 키가 작은 순으로 서있거나, 키가 큰 순으로 서있거나, 아무렇게나 서있어도 똑같은 비용이 든다. 따라서 n이 같으면 동물이 어떻게 서있는지는 정렬의 시간에 영향을 끼치지 않는다. 이번 실험에서 이용할 선택 정렬은 제자리 선택 정렬로, 추가 배열을 이용하지 않고 정렬하는 방법이다.
선택 정렬의 시간복잡도 : $O(n^2)$

삽입 정렬

마찬가지로 n마리의 동물들을 키가 작은 순으로 나열한다고 가정해보자.
삽입 정렬은 동물을 한 마리씩 늘려가며 먼저 정렬한 배열 안에 삽입을 해주는 구조이다.
처음 동물을 삽입하는데 소요되는 비용은 1
다음 동물을 삽입하는데 소요되는 비용은 1~2 (최악의 경우 2)
그 다음 동물을 삽입하는데 소요되는 비용은 1~3 (최악의 경우 3)
...
마지막 동물을 삽입하는데 소요되는 비용은 1~n (최악의 경우 n)
비용이 가장 적게 드는 경우는 동물들이 키가 작은 순으로 서있는 상태에서 삽입 정렬을 수행하는 경우이다. 이때는 동물을 삽입하는데 소요되는 비용이 모두 1이므로, $O(n)$의 비용이 든다.
비용이 가장 많이 드는 경우는 동물들이 키가 큰 순으로 서있는 상태에서 삽입 정렬을 수행하

는 경우이다. 이때는 동물들을 삽입하는데 소요되는 비용이 순서대로 1, 2, 3, ...,n의 비용이 든다. 따라서 선택 정렬과 소요되는 비용이 같다.
이번 실험에서 이용할 삽입 정렬은 제자리 삽입 정렬로, 제자리 선택 정렬과 마찬가지로 추가 배열을 이용하지 않고 정렬하는 방법이다.
삽입 정렬의 시간복잡도 : $O(n^2)$

3. 실험 방법

실험 방법은 다음과 같다. 배열의 입력이 A, B, C 세 경우로 들어왔을 때,
A : 무작위 배열에 대하여 선택 정렬을 하여 걸린 시간, 삽입 정렬을 하여 걸린 시간을 각각 프로그래밍을 통해 측정하고, 측정값들을 표에 작성한다. 표를 토대로 엑셀을 이용해 차트를 형성해 변화 추이를 관찰한다.

B : 이번엔 정렬된 배열에 대하여 A와 마찬가지의 방법으로 측정을 한다. 정렬된 배열에 대하여 선택 정렬과 삽입 정렬의 성능은 차이가 심하므로 이 실험에 대해서는 n의 값을 다르게 하여 똑같은 n의 값에서 각각의 성능을 비교하는 것이 아니라 n의 변화함에 따라 변화량을 비교한다.

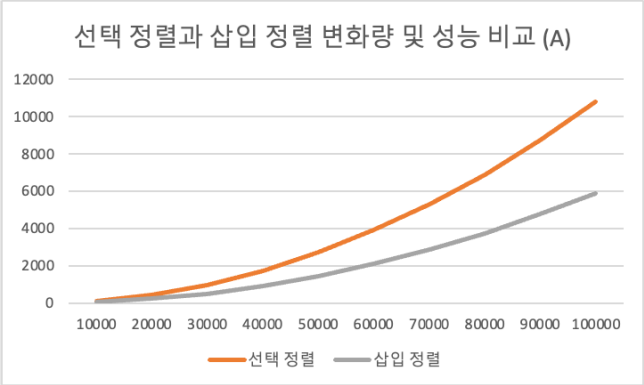
C : 이번엔 역으로 정렬된 배열에 대하여 A와 마찬가지의 방법으로 측정을 한다.

4. 측정값 및 실험 결과

A. 배열의 입력으로 정렬이 안 된 데이터가 주어지는 경우

n	10000	20000	30000	40000	50000
선택 정렬	108.555	447.917	977.092	1725.316	2721.096
삽입 정렬	58.247	234.641	517.012	930.017	1470.533

n	60000	70000	80000	90000	10000
선택 정렬	3914.832	5292.113	6907.123	8725.929	10824.720
삽입 정렬	2130.427	2867.227	3746.671	4789.018	5894.199



x축 : n의 값
y축 : 각 정렬의 시간 측정값(ms)

위 표와 그래프를 보면 똑같은 n의 값에서 측정 값은 약 두 배 차이가 나는 것을 확인 할 수 있다. 이는 무작위 배열에 대하여 삽입 정렬을 수행할 때, 내부 반복문에서 삽입할 배열의 위치가 최악의 삽입 케이스, 최선의 삽입 케이스의 평균적으로 이루어져 있기 때문이다.

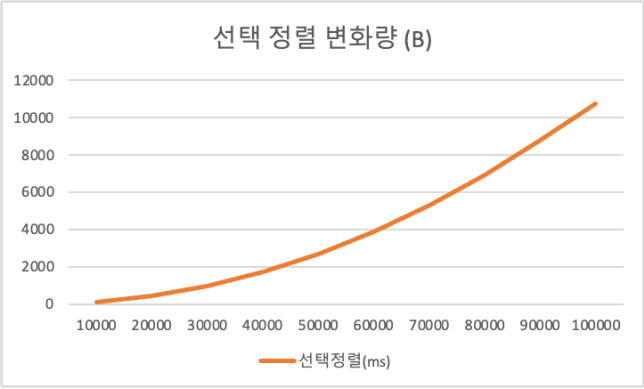
또한 두 그래프 모두 그래프모양이 곡선이고, 이는 두 방법의 정렬 모두 시간복잡도가 $O(n^2)$ 임을 나타낸다.

B. 각 정렬의 입력으로 정렬된 데이터가 주어지는 경우

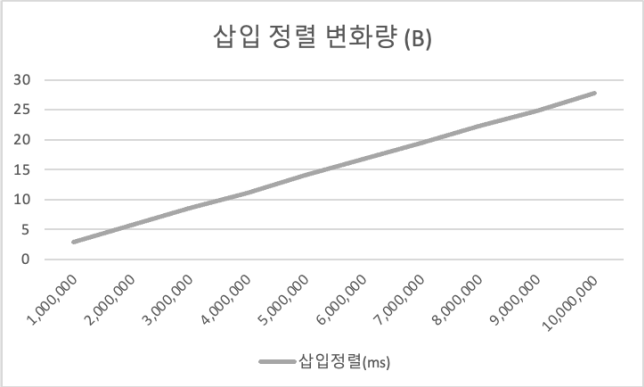
n	10,000	20,000	30,000	40,000	50,000
선택정렬(ms)	119.715	431.127	969.590	1721.297	2691.063
n	60,000	70,000	80,000	90,000	100,000
선택정렬(ms)	3893.882	5315.567	6941.497	8790.729	10770.137

n	1,000,000	2,000,000	3,000,000	4,000,000	5,000,000
삽입정렬(ms)	2.923	5.774	8.579	11.116	14.009
n	6,000,000	7,000,000	8,000,000	9,000,000	10,000,000
삽입정렬(ms)	16.726	19.409	22.262	24.850	27.789

각 표를 그래프로 표현하면 다음과 같다.



x축 : n의 값
y축 : 각 정렬의 시간 측정값(ms)

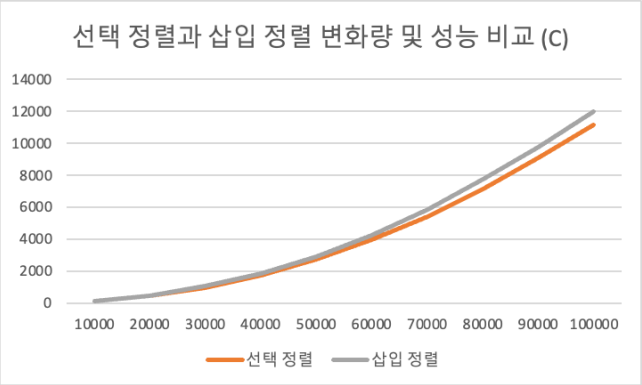


위 두 그래프를 보면 알 수 있듯이, 선택 정렬과 삽입 정렬의 그래프의 모양이 다르다.
선택 정렬은 케이스 A 에서의 그래프와 같은 모양을 띄고 있는 반면, 삽입 정렬의 그래프는 케이스 A에서의 그래프 모양과 달리 일직선이다. 그래프의 모양이 일직선이라는 것은 정비례 그래프라는 것이고, 배열의 크기 n의 값이 증가함에 따라 측정 시간이 정비례하게 늘어난다는 것이다. 이는 정렬된 데이터에 대한 삽입 정렬을 시행할 경우 $O(n)$의 시간복잡도를 갖는다는 것을 나타낸다.

C. 각 정렬의 입력으로 역순으로 데이터가 주어지는 경우

n	10000	20000	30000	40000	50000
선택 정렬	129.582	438.038	981.998	1747.618	2731.238
삽입 정렬	116.757	469.089	1053.613	1860.107	2937.542

n	60000	70000	80000	90000	100000
선택 정렬	3945.283	5445.193	7119.613	9067.077	11164.923
삽입 정렬	4274.210	5882.065	7740.869	9742.989	11980.371



x축 : n의 값
y축 : 각 정렬의 시간 측정값(ms)

역순으로 정렬된 데이터가 주어진 경우, 배열의 크기 n이 증가함에 따라 선택 정렬과 삽입 정렬의 변화 추이가 비슷한 것을 확인할 수 있다. 케이스 C에서는 두 방법의 정렬 모두 시간복잡도가 $O(n^2)$임을 확인할 수 있다.

5. 추가 분석

1. A, B, C 케이스에 대하여 선택 정렬 성능 비교

똑같은 n값에 대하여

A : 무작위 배열

B : 정렬된 배열

C : 역순으로 정렬된 배열

각 케이스에서 **선택 정렬**은 어떠한 성능을 보이는지 실험해보자.

n = 100000으로 놓고 각 케이스의 선택 정렬을 수행한 시간을 출력한 결과는 다음과 같다.
A : 11024.707ms
B : 11062.090ms
C : 11185.577ms
A, B, C경우의 측정값이 비슷하다. 이는 앞에서 예측한 것과 같다. 배열이 어떻게 정렬되어 있든 선택 정렬 알고리즘은 내부 반복문에서 최솟값 혹은 최댓값을 찾아야 하기 때문이다.

2. A, B, C 케이스에 대하여 삽입 정렬 성능 비교

마찬가지로 똑같은 n값에 대하여
A, B, C 각각의 케이스에서 **삽입 정렬**은 어떠한 성능을 보이는지 실험해보자.
n = 100000으로 놓고 각 케이스의 삽입 정렬을 수행한 시간을 출력한 결과는 다음과 같다.
A : 5899.195ms
B : 0.285ms
C : 11841.134ms
삽입 정렬은 선택 정렬과는 다르게, 정렬된 배열에 대하여 삽입 정렬을 수행한 케이스는 매우 작은 시간이 소요된 것을 볼 수 있다. 이는 앞에서 설명했듯이 정렬되어 있는 배열을 삽입 정렬할 경우, 1, 1, 1, 1, ..., 1 총 1의 비용이 n번 드므로 $O(n)$의 시간복잡도를 갖는다. 한편 삽입 정렬의 시간복잡도는 $O(n^2)$인데, $O(n)$과 $O(n^2)$의 시간 차이는 매우 크다. 그리고 무작위 배열인 A와 역순으로 정렬된 배열인 C를 정렬하는 데에 소요된 시간은 약 2배의 차이가 나는데, 이는 무작위로 정렬된 배열에 소요된 시간이 최악의 케이스와 최선의 케이스의 평균과 비슷하므로 두 배의 차이가 나는 것이다.

- 코드

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <stdint.h>
void swap(int *arr, int idx1, int idx2) {
//arr 배열의 idx1과 idx2자리에 있는 원소들을 swap
	int tmp;
	tmp = arr[idx1];
	arr[idx1] = arr[idx2];
	arr[idx2] = tmp;
}
void inPlaceSelectionSort(int *arr, int n) { // 제자리 선택 정렬
	int pass, j, minLoc ,tmp;
	for (pass=0;pass<n-1;pass++) {
		minLoc = pass;
		for (j=pass+1;j<n;j++) {
			if(arr[j] < arr[minLoc]) {
				minLoc = j;
			 }
		}
		swap(arr, pass, minLoc);
	}
}
void inPlaceInsertionSort(int *arr, int n) { // 제자리 삽입 정렬
	int pass, save, j;
	for (pass=1;pass<n;pass++) {
		save = arr[pass];
		j = pass-1;
		while(j>=0 && arr[j]>save) {
			arr[j+1] = arr[j];
			j=j-1;
		}
		arr[j+1] = save;
	 }
}
int main() {
int n, i, j;
	clock_t start_iSS, end_iSS, start_iIS, end_iIS;
	int *arr_by_iSS; // 선택 정렬할 배열
	int *arr_by_iIS; // 삽입 정렬할 ㅏ배열
	int *arr; // 정방향 배열
	int *arr_reverse; // 역방향 배열
	double elapsed_iSS, elapsed_iIS;
	// scanf("%d", &n);
	n=100000;
	arr_by_iSS = (int *)malloc(sizeof(int)*n); // 제자리 선택정렬할 배열
	arr_by_iIS = (int *)malloc(sizeof(int)*n); // 제자리 삽입정렬할 배열
	arr = (int *)malloc(sizeof(int)*n);
	arr_reverse = (int *)malloc(sizeof(int)*n);
	// for (j=0;j<10000000;j++) {
	// arr_by_iIS[j] = arr_by_iSS[j] = j+1;
	// }
	srand(time(NULL));
	for (j=0;j<n;j++) {
		arr_by_iSS[j] = arr_by_iIS[j] = rand();
		arr[j] = j+1;
		arr_reverse[j] = n-j;
	 } // arr_by_iTS배열과 arr_by_iIS배열 모두 난수가 n개의 똑같은 난수가 들어간 배열
```

```
// // 똑같은 n에 대한 실험
// start_iSS = clock(); // A
// inPlaceInsertionSort(arr_by_iSS, n);
// end_iSS = clock();
// elapsed_iSS = (double)(end_iSS - start_iSS)/CLOCKS_PER_SEC;
// printf("A : %.3fms\n", 1000*elapsed_iSS);
// start_iSS = clock(); // B
// inPlaceInsertionSort(arr, n);
// end_iSS = clock();
// elapsed_iSS = (double)(end_iSS - start_iSS)/CLOCKS_PER_SEC;
// printf("B : %.3fms\n", 1000*elapsed_iSS);
// start_iSS = clock(); // C
// inPlaceInsertionSort(arr_reverse, n);
// end_iSS = clock();
// elapsed_iSS = (double)(end_iSS - start_iSS)/CLOCKS_PER_SEC;
// printf("C : %.3fms\n", 1000*elapsed_iSS);


        for (i=1;i<=10;i++) {
                n= 10000*i;
                for (j=0;j<n;j++) {
                        arr_by_iSS[j] = arr_by_iIS[j] = rand();
                        arr[j] = j+1;
                        arr_reverse[j] = n-j;
                }
// arr_by_iTS배열과 arr_by_iIS배열 모두 난수가 n개의 똑같은 난수가 들어간 배열
                start_iSS = clock(); // 프로세스는 clock 단위로 일을 한다고 생각하자.
                inPlaceSelectionSort(arr, n);
                end_iSS = clock();
                elapsed_iSS = (double)(end_iSS - start_iSS)/CLOCKS_PER_SEC;
                // CLOCKS_PER_SEC == 1000000이다. 즉, 1초에 1000000CLOCKS
                for (j=0;j<n;j++) {
                        arr[j] = j+1;
                        arr_reverse[j] = n-j;
                 }
                start_iIS = clock();
                inPlaceInsertionSort(arr, n);
                end_iIS = clock();
                elapsed_iIS = (double)(end_iIS - start_iIS)/CLOCKS_PER_SEC;
                printf("\nn : %d00000\n", i);
                printf("선택 정렬 : %.3fms\n", 1000*elapsed_iSS);
                printf("삽입 정렬 : %.3fms\n", 1000*elapsed_iIS);
        }
        free(arr_by_iIS);
        free(arr_by_iSS);
        free(arr);
        free(arr_reverse);
        return 0;
}
```