알고리즘 및 실습 과제 #1

19011077 황건하

1. 선택 정렬과 삽입 정렬

이 실험은 제자리 선택 정렬과 제자리 삽입 정렬의 데이터에 따른 시간 복잡도를 측정하고 분석하기 위해 진행되었다. 먼저 제자리 정렬이란 리스트 자체를 위한 공간 이외에 O(1) 공간만을 사용하는 정렬을 뜻한다. 따라서 제자리 선택 정렬과 제자리 삽입 정렬은 외부의 데이터구조를 사용하지 않고 제자리에서 수행된다.

2. 입력되는 데이터에 따른 선택 정렬과 삽입 정렬의 시간 복잡도

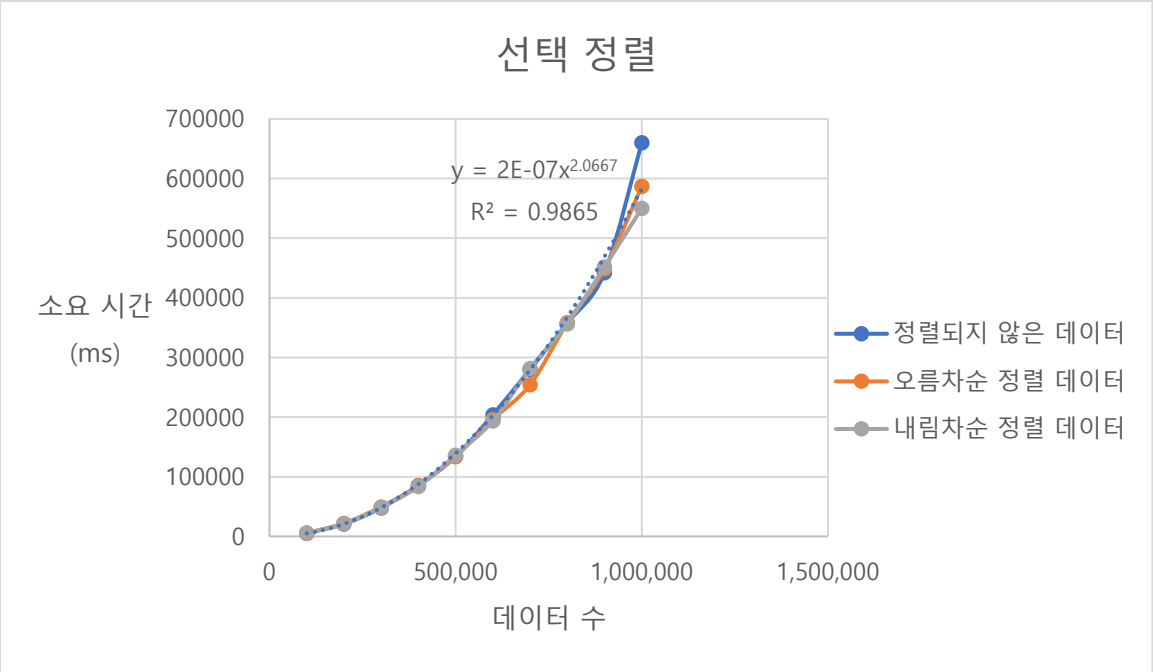
위의 두 정렬 방법은 모두 대체적으로 $O(n^2)$의 시간 복잡도를 가지고 있다. 하지만 대략적인 시간 복잡도는 같지만, 두 방식의 구현은 다르다. 따라서 입력되는 데이터에 따른 차이를 알아보려고 한다. 먼저 입력되는 데이터는 3가지로 구분된다. 첫 번째는 무작위로 입력된다. 즉, 아예 정렬되어 있지 않은 데이터이다. 두 번째는 오름차순으로 정렬된 데이터, 세 번째는 내림차순으로 정렬된 데이터이다. 이러한 데이터들이 입력으로 들어왔을 때, 같은 데이터를 기준으로 제자리 선택 정렬과 제자리 삽입 정렬을 수행했을 때, 소요되는 시간을 알아보고, 차이가 발생했다면 어떠한 이유에서 차이가 발생했는지 알아보자. 각각의 정렬은 오름차순으로 데이터를 정렬시키고, 다음은 실험에 사용된 데이터와 소요된 시간이다.

데이터에 따른 선택 정렬과 삽입 정렬의 소요 시간

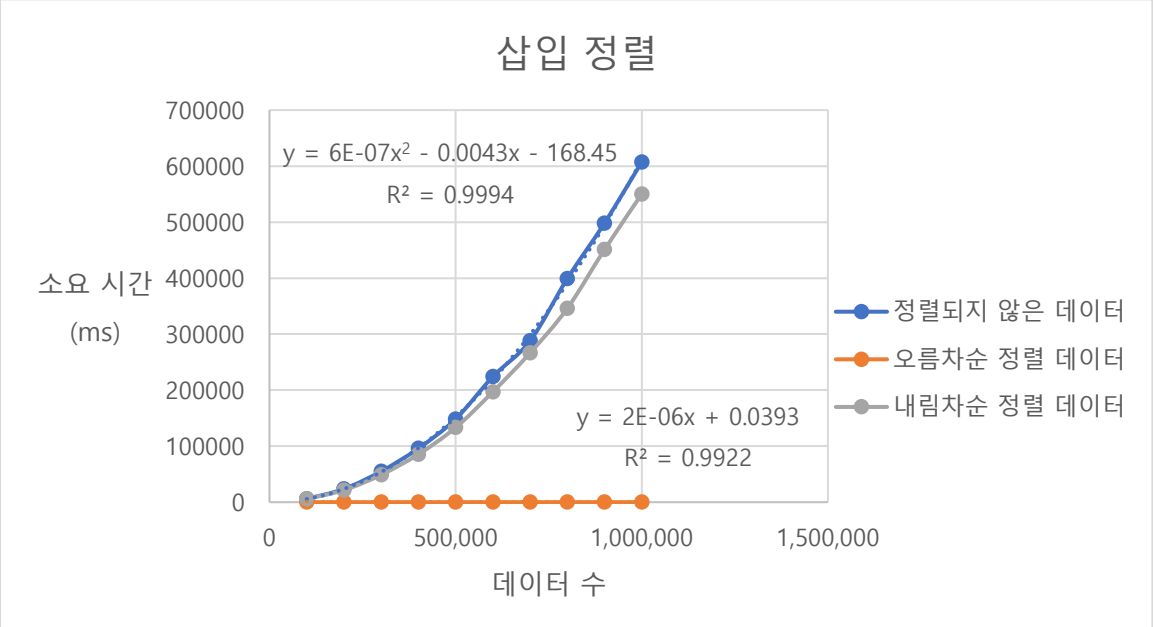
데이터	정렬 방식	정렬되지 않은 데이터(ms)	오름차순 정렬 데이터(ms)	내림차순 정렬 데이터(ms)
100,000	선택 정렬	5178.01100	5418.58800	5640.03900
	삽입 정렬	5884.63200	0.17400	5727.49600
200,000	선택 정렬	20894.57800	22027.48700	21391.47600
	삽입 정렬	23578.81900	0.37700	21567.06900
300,000	선택 정렬	47828.18700	49512.02500	48575.10600
	삽입 정렬	55081.65100	0.57799	48853.07800
400,000	선택 정렬	85310.41000	85312.24100	83510.74000
	삽입 정렬	95994.32700	0.89000	85516.82000
500,000	선택 정렬	133943.22200	134331.16400	136420.37800
	삽입 정렬	148442.65700	0.93500	133188.15900
600,000	선택 정렬	203806.60000	196247.00000	193677.95300
	삽입 정렬	224207.71400	1.13400	197032.57700
700,000	선택 정렬	280213.04700	254332.00200	281688.30100
	삽입 정렬	288036.86400	1.26500	266467.15600

800,000	선택 정렬	357749.29000	358260.11600	356636.70800
	삽입 정렬	399206.37100	1.46200	346060.62300
900,000	선택 정렬	442142.05800	448658.70300	451861.16200
	삽입 정렬	498271.63100	1.66400	451033.21800
1,000,000	선택 정렬	660227.73600	587363.24300	550140.07000
	삽입 정렬	607352.53900	1.84500	550222.55500

데이터에 따른 선택 정렬의 소요 시간 그래프

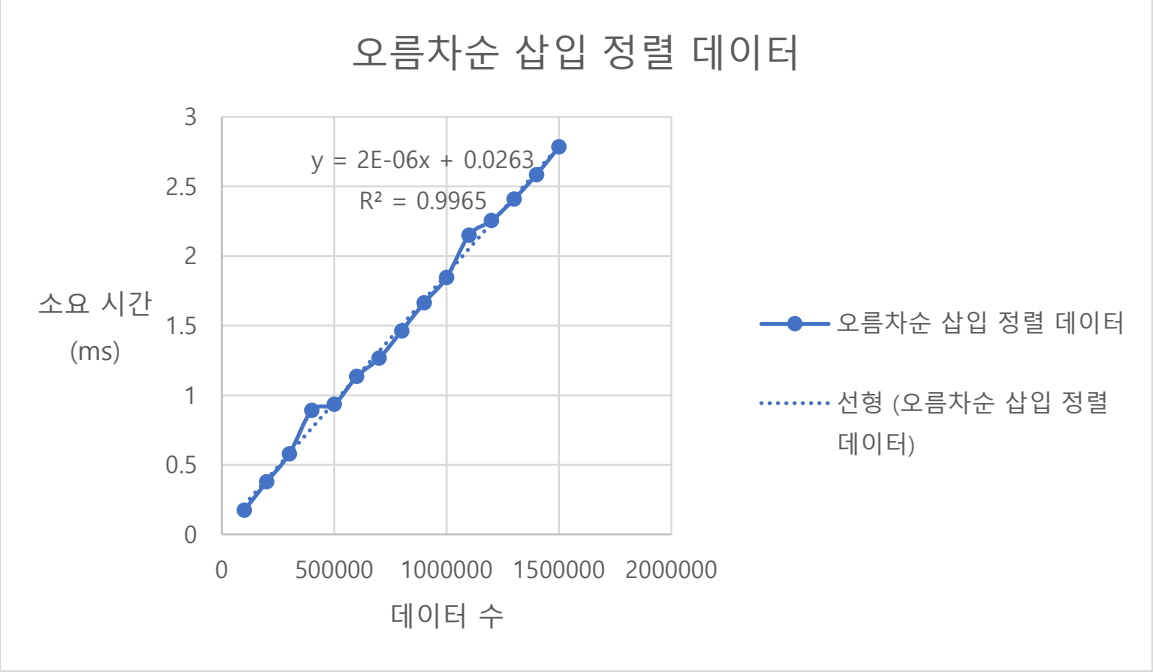


데이터에 따른 삽입 정렬의 소요시간 그래프



위 그래프를 살펴보면, 정렬된 데이터가 입력값인 삽입 정렬을 제외하고는 비슷한 추이를 가진다. 해당 경우를 더 자세히 보기 위해 데이터를 추가하고 그래프를 따로 작성하였다.

오름차순으로 정렬된 데이터가 들어올 때 삽입 정렬의 소요 시간



먼저 각각의 데이터에 따른 결과를 분석해보자. 선택 정렬은 대체로 모든 경우가 비슷한 추세를 가진다. 선택 정렬의 추세선 함수를 보면 데이터 수의 제곱과 비슷하게 증가한다. 따라서 선택 정렬의 시간 복잡도는 $O(n^2)$으로 추정할 수 있다. 삽입 정렬을 살펴보면, 오름차순 정렬 데이터를 제외하고는 선택 정렬과 비슷하게 $O(n^2)$의 시간 복잡도를 가진다. 하지만 오름차순 정렬 데이터는 선형으로 증가한다. 따라서 정렬된 데이터는 $O(n)$의 시간 복잡도를 가진다.

정렬 방식	정렬되지 않은 데이터	오름차순 정렬 데이터	내림차순 정렬 데이터
선택 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$
삽입 정렬	$O(n^2)$	$O(n)$	$O(n^2)$

3. 선택 정렬과 삽입 정렬에서 차이가 발생하는 이유

이러한 차이가 나는 이유는 무엇일까? 먼저 두 방식은 이름부터 차이가 있듯이, 정렬하는 과정에서도 차이가 존재한다. 두 정렬 모두 $O(n^2)$의 시간 복잡도를 가지는 이유는 내부 반복문과 외부 반복문이 각각 $O(n)$의 시간 복잡도를 가져서 총 $O(n^2)$의 시간 복잡도를 가진다.

하지만 예외가 되는 오름차순 정렬데이터가 삽입 정렬의 입력값으로 들어올 때, $O(n)$의 시간이 소요된다. 그 이유를 알기 위해서는 삽입 정렬이 어떠한 방식으로 정렬하는지 알아야 한다. 삽입 정렬은 매 순서마다 해당 원소를 배열의 정렬된 부분에서 삽입할 수 있는 위치를 찾아서 해당 위치

에 넣는다. 즉, 정렬된 부분의 뒤에서부터 자신보다 크다면 계속 작은 쪽으로 향하면서 위치를 찾아가다가 자신보다 작은 원소가 나오면 반복을 멈춘다. 이러한 정렬방식에서 이미 정렬된 데이터가 들어온다면, 이미 모든 데이터가 정렬되어 있기 때문에 자신의 위치를 찾을 필요가 없어진다. 따라서 내부 반복문의 시간 복잡도는 O(1)이 되어서 전체적으로 O(n)의 시간 복잡도를 가지게 된다.

4. 마무리

결론적으로, 선택 정렬과 삽입 정렬은 대체적으로 $O(n^2)$의 시간 복잡도를 가진다. 하지만 삽입 정렬은 정렬된 데이터가 들어올 때, 내부 반복문의 시간 복잡도가 O(1)이 되므로 O(n)의 시간 복잡도를 가진다. 따라서 어느정도 정렬된 데이터가 들어온다면, 삽입 정렬을 사용하는 것이 시간 소모를 줄일 수 있는 방법이다. 하지만 평균적으로 두 정렬 모두 다른 정렬 알고리즘에 비해 시간 소모가 큰 편이다. 따라서 적은 데이터에 사용하는 것이 적합하며, 비교적 많은 양의 데이터가 입력으로 들어올 때에는, 병합 정렬과 같은 O(n*logn) 시간 복잡도를 가지는 정렬 알고리즘을 사용하는 것이 적합하다.

5. 프로그램 소스 코드

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

// 배열의 두 데이터를 SWAP 하는 데에 편리함을 위함.
#define SWAP(x, y, tmp) ((tmp) = (x), (x) = (y), (y) = (tmp))

int *arr_insertion; // 삽입정렬을 수행하는데 사용되는 배열.
int *arr_selection; // 선택정렬을 수행하는데 사용되는 배열.
int *tmp;           // 병합정렬을 사용하는데 사용되는 temp 배열.

// 정렬된 데이터를 입력하기 위해 병합 정렬을 사용했다.
void merge(int list[], int left, int mid, int right)
{
    int i, j, k, l;
    i = left;
    j = mid + 1;
    k = left;

    while (i <= mid && j <= right)
    {
        if (list[i] <= list[j])
            tmp[k++] = list[i++];
        else
            tmp[k++] = list[j++];
    }
    while (i <= mid)
        tmp[k++] = list[i++];
    while (j <= right)
        tmp[k++] = list[j++];
    for (l = left; l <= right; l++)
    {
        list[l] = tmp[l];
    }
}

void merge_sort(int list[], int left, int right)
{
    int mid;

    if (left < right)
    {
        mid = (left + right) / 2;
        merge_sort(list, left, mid);
        merge_sort(list, mid + 1, right);
        merge(list, left, mid, right);
    }
}

void insertion_sort(int n) // 삽입 정렬
{
    int i, j, idx, tmp;
```

```
    for (i = 1; i < n; i++)
    {
        idx = i;

        /*
        j 와 j-1 을 비교하므로 j >= 1 까지 수행하고,
        정렬된 배열로 들어갈 인자가 앞으로 가야한다면,
        for 문을 반복해서 수행한다.
        */
        for (j = i; j >= 1 && arr_insertion[j] < arr_insertion[j - 1]; j--)
            SWAP(arr_insertion[j], arr_insertion[j - 1], tmp);
    }
}

void selection_sort(int n) // 선택 정렬
{
    int idx;
    int tmp;
    int i, j;

    for (i = n - 1; i > 0; i--)
    {
        idx = 0;
        for (j = 1; j <= i; j++)
        {
            // 최대값이 있는 인덱스를 idx 에 저장한다.
            if (arr_selection[idx] < arr_selection[j])
                idx = j;
        }
        SWAP(arr_selection[idx], arr_selection[i], tmp);
    }
}

// 오름차순으로 정렬된 데이터를 내림차순으로 바꾼다.
void reverse_arr(int arr[], int n)
{
    int i, tmp;

    for (i = 0; i < n / 2; i++)
        SWAP(arr[i], arr[n - 1 - i], tmp);
}

int main()
{
    int n, i;
    double start, end;

    srand(time(NULL)); // 난수 설정을 위함.
    scanf("%d", &n);
    arr_insertion = (int *)malloc(sizeof(int) * n);
    arr_selection = (int *)malloc(sizeof(int) * n);
    tmp = (int *)malloc(sizeof(int) * n);

    for (i = 0; i < n; i++)
    {
        // 배열에 임의의 수를 입력하고, 두 배열의 데이터를 같게 만든다.
```

```
        arr_insertion[i] = rand() % 1000;
        arr_selection[i] = arr_insertion[i];
    }

    // 정렬 안되어있을 때
    printf("--------------\n");
    printf("정렬이 안 된 데이터\n");
    start = (double)clock() / CLOCKS_PER_SEC;
    selection_sort(n);
    end = (double)clock() / CLOCKS_PER_SEC;
    printf("정렬 안됐을 때 선택정렬 : %.5lfms\n", (end - start) * 1000);
    start = (double)clock() / CLOCKS_PER_SEC;
    insertion_sort(n);
    end = (double)clock() / CLOCKS_PER_SEC;
    printf("정렬 안됐을 때 삽입정렬 : %.5lfms\n", (end - start) * 1000);
    printf("--------------\n");
    printf("\n");

    // 오름차순으로 정렬되어있을 때
    merge_sort(arr_insertion, 0, n - 1); // 배열을 오름차순으로 만든다.
    merge_sort(arr_selection, 0, n - 1); // 배열을 오름차순으로 만든다.
    printf("--------------\n");
    printf("오름차순으로 정렬된 데이터\n");
    start = (double)clock() / CLOCKS_PER_SEC;
    selection_sort(n);
    end = (double)clock() / CLOCKS_PER_SEC;
    printf("오름차순으로 정렬되어있을 때 선택정렬 : %.5lfms\n", (end - start) * 1000);
    start = (double)clock() / CLOCKS_PER_SEC;
    insertion_sort(n);
    end = (double)clock() / CLOCKS_PER_SEC;
    printf("오름차순으로 정렬되어있을 때 삽입정렬 : %.5lfms\n", (end - start) * 1000);
    printf("--------------\n");
    printf("\n");

    // 내림차순으로 정렬되어있을 때
    reverse_arr(arr_insertion, n); // 오름차순인 배열을 내림차순으로 바꾼다.
    reverse_arr(arr_selection, n); // 오름차순인 배열을 내림차순으로 바꾼다.
    printf("--------------\n");
    printf("내림차순으로 정렬된 데이터\n");
    start = (double)clock() / CLOCKS_PER_SEC;
    selection_sort(n);
    end = (double)clock() / CLOCKS_PER_SEC;
    printf("내림차순으로 정렬되어있을 때 선택정렬 : %.5lfms\n", (end - start) * 1000);
    start = (double)clock() / CLOCKS_PER_SEC;
    selection_sort(n);
    end = (double)clock() / CLOCKS_PER_SEC;
    printf("내림차순으로 정렬되어있을 때 선택정렬 : %.5lfms\n", (end - start) * 1000);
    printf("--------------\n");

    free(arr_insertion);
    free(arr_selection);
    free(tmp);
}
```