알고리즘 과제 2. 정렬 알고리즘 비교

제출일 : 2017/03/31

경영학과 201101328　박성환

목차 :

1. 문제 정의

알고리즘 수업에서 선택, 버블, 삽입, 쉘, 병합, 퀵 정렬을 학습했다. 그 정렬들을 실제로 구현하고 정렬간 성능을 비교하는 테스트를 진행한다.

1.1. 학습한 "선택, 버블, 삽입, 쉘, 병합, 퀵" 정렬을 구현하라.

: 학습한 정렬을 C로 구현한다. 구체적인 구현은 학습한 ppt를 충실히 재현하는 데 초점을 맞춘다. 테스트를 디자인하면서 제기된 세부 문제는 다음과 같다.

1.1.1 개별 알고리즘과 실제 실험 파일은 각각 따로 작성할 것인가, 같이 작성할 것인가?

1.1.2. 실제 실험 파일에서 개별 알고리즘을 어떻게 끌어다 사용할 것인가?

1.1.3. 추가 변수 저장 공간이 필요한 병합 정렬에서 어떻게 공간을 만들고 사용할 것인가?

1.2. "선택, 버블, 삽입, 쉘, 병합, 퀵" 알고리즘의 성능을 판단하라

: 알고리즘들의 성능을 판단한다. 과제 내용대로 횟수를 늘려가며 알고리즘들 내에서 계산이 얼마나 이루어지는지 계산한다. 성능을 판단하는 데 제기된 세부 문제는 다음과 같다.

1.2.1. 성능을 판단하는 변수는 무엇으로 설정하는가?

1.2.2. 알고리즘의 성능은 어떻게 판단할 것인가?

1.2.3. 재귀 알고리즘에서의 변수 초기화 문제 이슈

* 구체적인 알고리즘의 구현은 배운 내용 그대로 공부하고 구현하면 온전히 작동하므로 그것에 대한 해설이나 설명은 하지 않는다.

2. 해결방안

2.1.1 개별 알고리즘과 실제 실험 파일은 각각 따로 작성할 것인가, 같이 작성할 것인가?

-> 구현 실험과 성능 실험을 구분해서 작성하고, 각 실험은 6가지 알고리즘을 모두 모은 알고리즘 파일과 실험 진행 파일로 나누어 작성한다.
여러 방안이 있었다. 한 파일에 알고리즘과 실험 진행 코드를 모두 담을 수도 있고, 알고리즘마다 파일을 만들어 실험 진행 파일에서 6개의 파일을 가져다 쓸 수도 있다. 결국은 하나의 헤더파일에 알고리즘을 모두 담고 진행 파일에서 그 헤더 파일을 가져다 쓰는 것이 도움이 될 것이라고 판단했다.

2.1.2. 실제 실험 파일에서 개별 알고리즘을 어떻게 끌어다 사용할 것인가?

-> 함수 포인터 배열을 사용해 6개의 정렬을 반복해서 실험을 진행한다.
6개의 정렬을 실험 진행 파일에서 개별적으로 사용해서 실험을 진행하면 코드가 불필요하게 길어질 것이라 생각했다. 확인 결과 C에는 함수 포인터 변수를 생성할 수 있었고 함수 포인터 배열을 반복문과 같이 사용해 코드를 간결하게 사용할 수 있다고 판단했다.

```
int main(void)
{
    // 실험 환경 세팅
    srand(time(NULL));
    int original_arr[LENGTH];
    int temp_arr[LENGTH];
    int sort_size = 6;
    void (*sort_algorithms[])(int[], int, int) = {selection_sort, bubble_sort,
                                                  insertion_sort, shell_sort,
                                                  merge_sort, quick_sort};

    char *sort_way[] = {"선택정렬", "버블정렬", "삽입정렬", "쉘정렬", "병합정렬", "퀵정렬"};
```

위 사진과 같이 'sort_algorithms'라는 함수 포인터 배열을 설정하고 그 안을 6개의 정렬 함수로 초기화했다. C의 함수 포인터 배열에는 약점이 있었는데, 그것은 각 원소들이 모두 같은 개수, 타입의 인자를 받고 리턴형도 모두 동일해야 한다는 것이었다. 그런데 삽입, 버블, 선택 정렬은 정렬시 배열, 배열의 길이 이렇게 2개의 인자만 있으면 정렬을 구현할 수 있었던 반면, 병합, 퀵은 세 개의 원소가 필요했다. 즉, 원소들의 인자 개수가 달라 어떻게 해야할지 고민했고 결국은 세 개의 정렬에 쓰이지 않는 offset이라는 변수를 추가해 6개 정렬의 모든 입력인자 개수를 통일해서 함수 포인터 배열을 문제없이 사용할 수 있었다.

```
    // 각 실험 진행
    for (int i = 0; i < sort_size; i++) {  // 각 정렬 횟수만큼 반복

        for (int j = 0; j < LENGTH; j++) { // 실험 배열 초기화
            temp_arr[j] = original_arr[j];
        }
        printf("%s\n", sort_way[i]);

        for (int k = 0; k < LENGTH; k++) {
            printf("%d ", original_arr[k]);
        }
        printf("\n");

        sort_algorithms[i](temp_arr, 0, LENGTH - 1);

        for (int k = 0; k < LENGTH; k++) {
            printf("%d ", temp_arr[k]);
        }
        printf("\n\n");
    }

    return 0;
}
```

결과 실제 실험 환경에서 배열의 원소를 반복문으로 바꿔가면서 효율적으로 알고리즘을 교체해가며 실험을 진행할 수 있었다. 이 방식은 성능 실험에도 똑같이 사용되었다.

2.1.3. 추가 변수 저장 공간이 필요한 병합 정렬에서 어떻게 공간을 만들고 사용할 것인가?

-> malloc을 사용해 동적 할당으로 새 배열을 만든다. divide된 배열들을 combine할 때마다 정렬결과를 새 배열에 저장하고 함수 종료 전 값을 원래 배열에 뒤집어 씌운다.

```
void helper_merge(int a[], int left, int mid, int right) {

    int length = right - left + 1;
    int *temp_array = (int*)malloc(sizeof(int) * length);
    int temp_index = 0;
    int low_index = left;
    int high_index = mid + 1;
```

분할 정복 알고리즘인 병합 정렬에서 정복 부분을 담당하는 ‘helper_merge’ 함수이다. 이 함수는 특성상 정렬시 결과를 저장할 임시 공간이 필요한데 이것을 'temp_array'라는 동적할당 포인터로 정의했다.

```
// temp_array를 원래 a에 덮어씌움
for(int i = 0; i < length; i++) {
    a[i + left] = temp_array[i];
}

free(temp_array);
```

값을 정렬한 이후에는 'temp_array'의 값들을 원래 배열 'a'에 뒤집어 씌운다. 마지막에 동적 할당을 해제하는 것도 잊지 말아야 한다.

2.2.1. 성능을 판단하는 변수는 무엇으로 설정하는가?

-> 정렬 알고리즘마다 if문, while문과 같은 반복문에서의 비교시에는 compare 변수를, 값 swap, 변수 대입시에는 move라는 변수를 증가시키고 최종 결과를 배열로 전달한다.

```
int compare = 0;
int move = 0;
int compare_and_move[2]; // 0번째 인덱스에 비교 횟수, 1번째에 대입 횟수를 기록
```

compare과 move 변수는 알고리즘 파일의 글로벌 변수로 정의했다. 처음에는 알고리즘 함수마다 변수를 정의했는데 그때에 문제점이 있었다. 지역변수로 설정된 경우에는 함수 종료시에 스택 프레임이 없어지면서 값도 같이 사라지게 되는 것이었다. 그래서 전역 변수로 정의해서 함수가 종료되어도 변수가 사라지지 않아 값을 온전히 받아올 수 있었다.

그리고 'compare_and_move' 변수는 compare와 move의 값을 저장한 배열로, 함수 리턴시에 이 주소를 반환하고자 했다.

```
//// shell sort implementation
int* shell_sort(int a[], int offset, int end) {

    int i, h;
    compare = 2;
    move = 2;

    for( h = (end + 1) / 2; h >= 1; h /= 2 ) {
        for( i = 0; i < h; i++ ) {
            insertion_for_sort(a, i, end, h, &compare, &move);
            move++;    // i 값 증가
            compare++; // for문 검사
        }
        move++;    // h값 변화
        compare++; // for문 검사
    }

    compare_and_move[0] = compare;
    compare_and_move[1] = move;

    return compare_and_move;
}
```

위 사진은 쉘 함수를 구현한 사진이다. 함수 진행 시에 비교, 대입 연산 시행 횟수를 측정하고 함수 마지막에 그 값들을 배열에 담아 주소를 반환한다. 원래 의도는, C에서는 함수에서 변수를 두 개를 동시에 리턴할 수 없다고 알고 있어서 배열에 담아서 리턴하자는 의도였다. 이후 지인에게 피드백을 구한 결과, 어차피 compare, move는 전역 변수로 정의되었기 때문에 배열에 값을 담아 배열 주소를 전달할 필요 없이 변수 그 자체에 접근해도 상관 없었을 것이라는 의견을 들었다. C의 프로그래밍을 정확히 몰라 발생한 문제로 다음 작성시에는 참고해야겠다고 생각했다.

2.2.2. 알고리즘의 성능은 어떻게 판단할 것인가?

-> 과제 파일에 언급되어 있는 대로 값 비교, 대입의 횟수가 더 적은 알고리즘이 성능이 더 좋은 알고리즘이라고 판단할 수 있다.

반복문에서의 비교는 compare 변수를 증가시키고, 값 swap, 변수 대입의 경우에는 move 변수를 증가시켰다.

이때 swap의 경우에는 move를 2 증가시켰다. for 반복문의 경우에는 블록이 n번 실행되면 조건식과 값 대입(값 증가)는 n + 1번이 실행되기 때문에 이것이 반영할 수 있도록 했다. 그리고 재귀 함수를 사용하는 정렬의 경우, copy by value로 값이 복사되기 때문에 그것도 move에 반영할까 고민했지만, 교수님이 원하시는 케이스인지 확신이 없어 일단 반영하지는 않았다.

알고리즘의 성능은 결국 비교, 대입 횟수가 더 적은 정렬이 우수한 정렬이라고 판단할 수 있을 것이다.

```
// 선택정렬
int* selection_sort(int a[], int offset, int end) {
    // offset : 인자를 3개로 맞춰주기 위한 변수. 의미 없음.

    int i, j, index;
    int temp;
    compare = 2; // 두 번의 for문의 마지막 조건식 연산
    move = 2;    // 두 번의 for문의 마지막 변수 증가 연산

    for (i = 0; i <= end; i++) {
        index = i;
        move++;    // index 대입

        for (j = i + 1; j <= end; j++) {
            if ( a[index] > a[j] ) {
                index = j;
                move++; // index 대입
            }
            compare++; // if문 비교

            move++;    // j 증가
            compare++; // for문 비교
        }

        temp = a[i];
        a[i] = a[index];
        a[index] = temp;
        move += 2; // swap

        move++;    // i 증가
        compare++; // for 문 비교
    }

    compare_and_move[0] = compare;
    compare_and_move[1] = move;

    return compare_and_move;
}
```

compare, move가 각각 2로 시작하는 것은 반복문에서의 추가 횟수를 반영한 것이다.

2.2.3. 재귀 알고리즘에서의 변수 초기화 문제 이슈

-> 퀵, 병합과 같은 재귀 알고리즘의 경우 wrapper로 감싸 초기화 문제를 해결했다.
compare, move 변수는 전역 변수로 각 알고리즘 실행할 때마다 0으로 다시 맞춰 놓고 값을 증가시키는 방법을 사용했다. 일반 정렬에서는 문제 없이 작동했는데 퀵, 병합과 같은 재귀 알고리즘의 경우는 문제가 있었다. 자신을 호출할 때마다 compare, move가 0으로 다시 초기화되는 문제가 발생했고, 어떻게 해야 하나 고민을 많이 했다. 결론은 quick, merge 정렬을 wrapper로 감싸서

wrapper에 compare, move 값을 초기화하고, wrapper 안의 본 함수는 값을 증가만 시키도록 함으로써 문제를 해결했다.

```
//// 2. Merge Sort implementation
int* _merge_sort(int a[], int left, int right, int *compare, int *move) {

    int mid = (left + right) / 2;
    (*move)++;  // mid 변수값 설정

    (*compare)++; // 뒷 조건식 시행
    if( left >= right ) return NULL;

    _merge_sort(a, left, mid, compare, move);
    _merge_sort(a, mid + 1, right, compare, move);
    helper_merge(a, left, mid, right, compare, move);
}

//// 3. Wrapper for merge. 재귀로 인한 변수 재초기화를 막기 위해 wrapper로 감쌈.
int* merge_sort(int a[], int left, int right) {
    compare = 0;
    move = 0;  // 0으로 초기화

    _merge_sort(a, left, right, &compare, &move);

    compare_and_move[0] = compare;
    compare_and_move[1] = move;

    return compare_and_move;
}
```

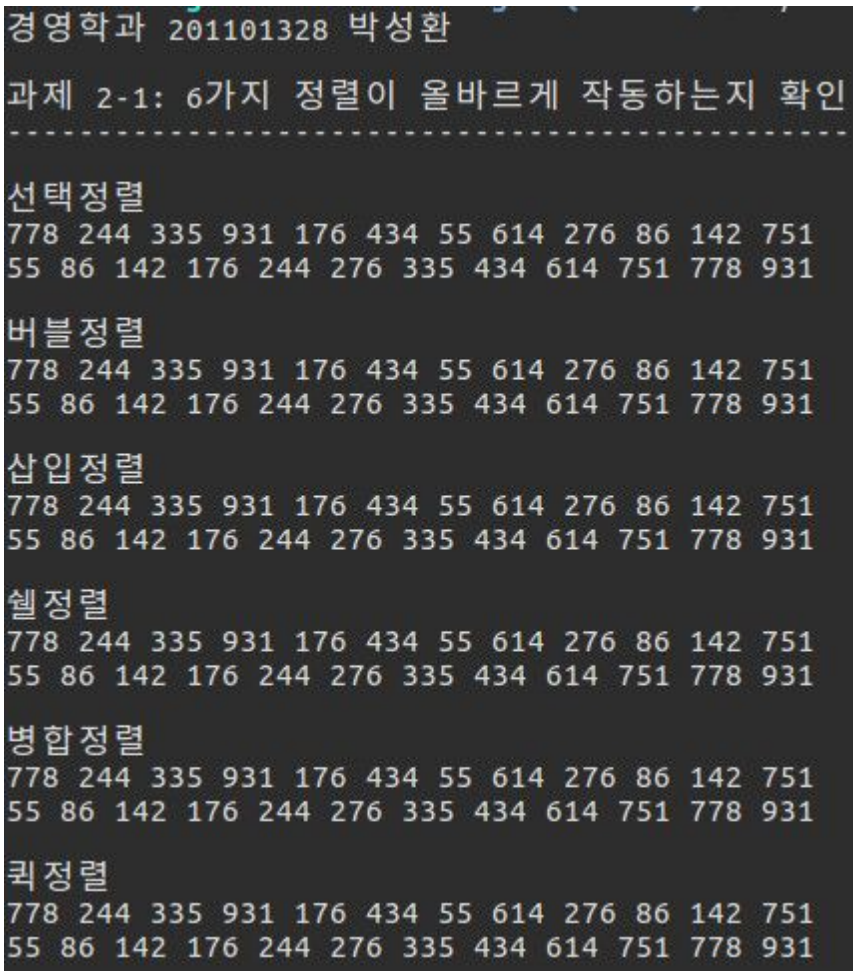
```
//// 2. Quick sort implementation
int* _quick_sort(int a[], int start, int end, int* compare, int* move) {

    int pivot;

    (*compare)++; // 뒷 if문 실행
    if (start < end) {
        pivot = get_pivot(a, start, end, compare, move);
        _quick_sort(a, start, pivot - 1, compare, move);
        _quick_sort(a, pivot + 1, end, compare, move);
    }
}

//// 3. Quick sort wrapper 재귀식의 영향으로 변수 재초기화를 막기 위해 wrapper로 감쌈.
int* quick_sort(int a[], int start, int end) {

    compare = 0;
    move = 0;  // 0으로 초기화

    _quick_sort(a, start, end, &compare, &move);

    compare_and_move[0] = compare;
    compare_and_move[1] = move;

    return compare_and_move;
}
```

merge, quick 정렬에서 실제 정렬을 실행하는 건 _merge_sort, _quick_sort이지만 merge_sort, quick_sort로 감싸서 값은 문제 없이 증가할 수 있도록 했다.

3. 테스트

3.1. 학습한 "선택, 버블, 삽입, 쉘, 병합, 퀵" 정렬을 구현하라.

: 정렬은 온전히 작동한다.



12개의 원소를 갖는 랜덤 배열을 만들고 각 정렬을 시행한 결과 모두 정확히 값을 출력했다.

3.2. 6가지 알고리즘의 성능을 판단하라

: 병합, 퀵, 쉘정렬이 선택, 삽입, 버블 정렬에 비해 압도적으로 우수하다.

```
과제 2-2: 6가지 정렬이 실행 성능을 비교.
---------------------------------------------------

********** 선택정렬 **********

size :  1000, compare :     1000002, move :      508890
size :  2000, compare :     4000002, move :     2018837
size :  3000, compare :     9000002, move :     4528146
size :  4000, compare :    16000002, move :     8038691
size :  5000, compare :    25000002, move :    12547777
size :  6000, compare :    36000002, move :    18058410
size :  7000, compare :    49000002, move :    24568747
size :  8000, compare :    64000002, move :    32077377
size :  9000, compare :    81000002, move :    40589048
size : 10000, compare :   100000002, move :    50097320


********** 버블정렬 **********

size :  1000, compare :      999669, move :     1242762
size :  2000, compare :     3999649, move :     4940862
size :  3000, compare :     8996839, move :    11243143
size :  4000, compare :    15984541, move :    19842931
size :  5000, compare :    25001809, move :    31433551
size :  6000, compare :    36000149, move :    45234779
size :  7000, compare :    49002841, move :    60980290
size :  8000, compare :    64002301, move :    80573199
size :  9000, compare :    80920495, move :   101452294
size : 10000, compare :   100007839, move :   124536748
```

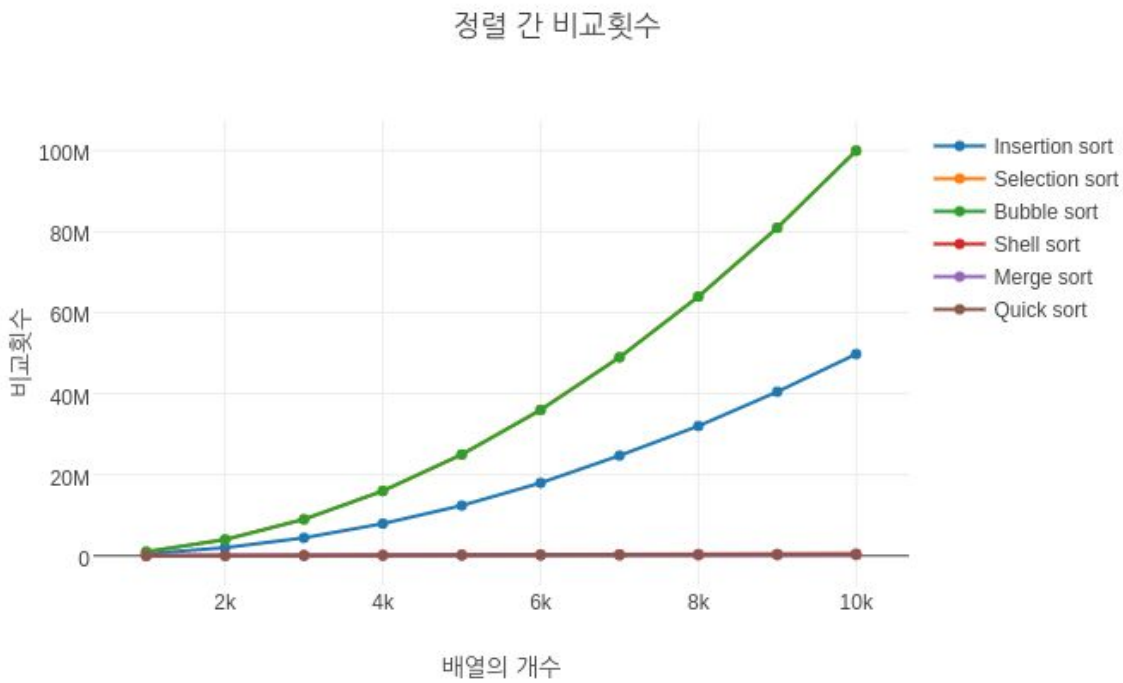
위 사진은 두 번째 테스트 결과 중 일부이다. 랜덤 배열의 사이즈를 1,000에서 10,000까지 1,000씩 증가시켜가며 비교와 대입이 몇 번씩 실행되는지 측정했다. 정렬간 보다 편하게 비교할 수 있도록 표와 그림을 첨부한다.

알고리즘 성능 비교

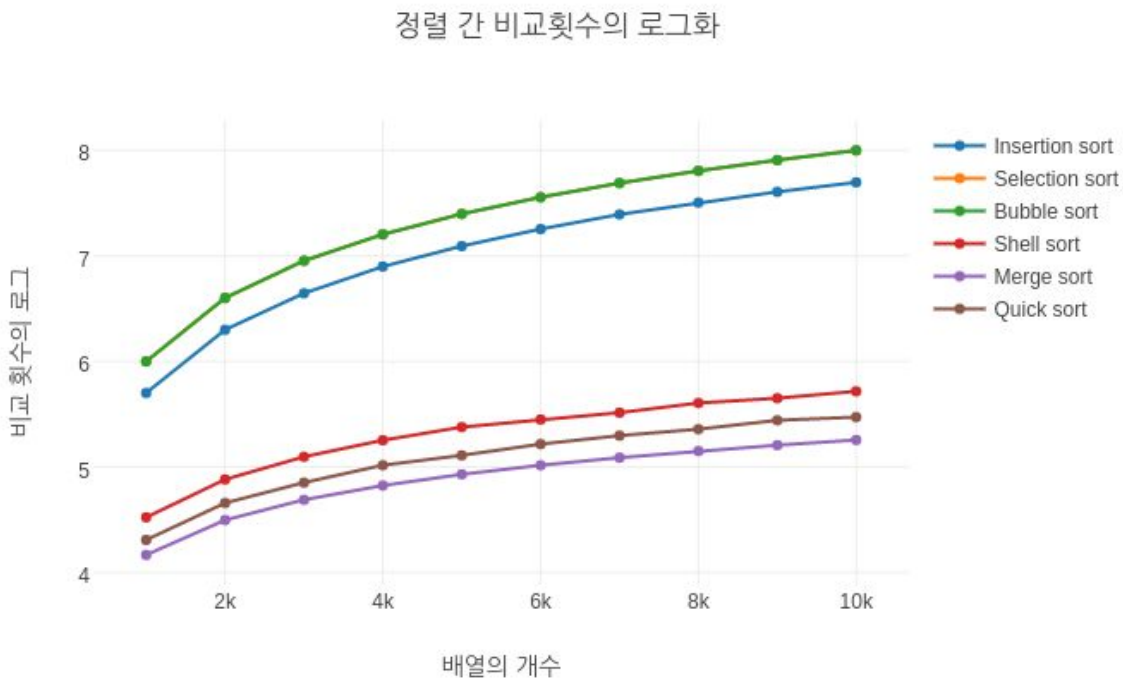
비교 횟수										
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
선택	1,000,002	4,000,002	9,000,002	16,000,002	25,000,002	36,000,002	49,000,002	64,000,002	81,000,002	100,000,002
버블	1,000,791	3,996,889	9,001,519	15,995,445	24,991,429	35,990,501	48,979,279	63,985,949	80,946,251	99,986,131
삽입	503,401	1,997,565	4,426,564	7,935,305	12,398,008	17,985,540	24,732,858	32,030,453	40,489,803	49,792,921
쉘	33,323	76,222	124,969	179,314	238,822	281,905	327,190	403,909	448,547	521,224
병합	14,681	31,422	48,924	66,722	85,165	103,892	122,632	141,618	161,012	180,360
퀵	20,364	45,590	71,154	103,978	128,992	164,806	198,295	228,213	276,892	296,967

교환 횟수										
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
선택	508,506	2,018,074	4,528,550	8,038,480	12,548,663	18,058,550	24,568,883	32,078,346	40,588,605	50,098,756
버블	1,237,216	5,073,898	11,143,070	19,803,046	31,247,119	44,970,389	60,717,175	79,887,421	100,948,169	124,758,475
삽입	753,110	2,992,353	6,633,862	11,894,972	18,587,028	26,966,317	37,085,297	48,029,688	60,716,719	74,669,390
쉘	33,232	76,827	125,282	185,969	244,568	286,413	331,670	423,900	450,212	534,350
병합	46,607	101,276	159,634	218,432	280,587	343,314	406,054	469,040	534,858	601,206
퀵	17,944	40,280	62,666	92,508	114,184	146,749	177,068	203,396	249,100	265,447

표로 봤을 때 위의 3개 정렬과 밑의 3개 정렬의 반복, 대입 횟수가 현격히 차이남을 확인할 수 있다. 위의 세 정렬인 삽입, 버블, 선택정렬은 시간복잡도가 n ^ 2이다. 배열의 크기가 10,000일 때 비교횟수가 5천만 ~ 1억번에 달하고, 대입횟수가 5천만 ~ 1억 2천만번에 달한다. 반대로 쉘 정렬의 경우 시간복잡도가 n ^ 1.5, 병합과 퀵 정렬은 n * logn로서 비교횟수와 교환횟수 모두 20~50만번 정도로 앞의 세 정렬에 비해 획기적으로 적다. 병합, 퀵 정렬을 적극적으로 써야할 것 같다고 생각했다. 그래프로 다시 한 번 확인해보기로 했다.



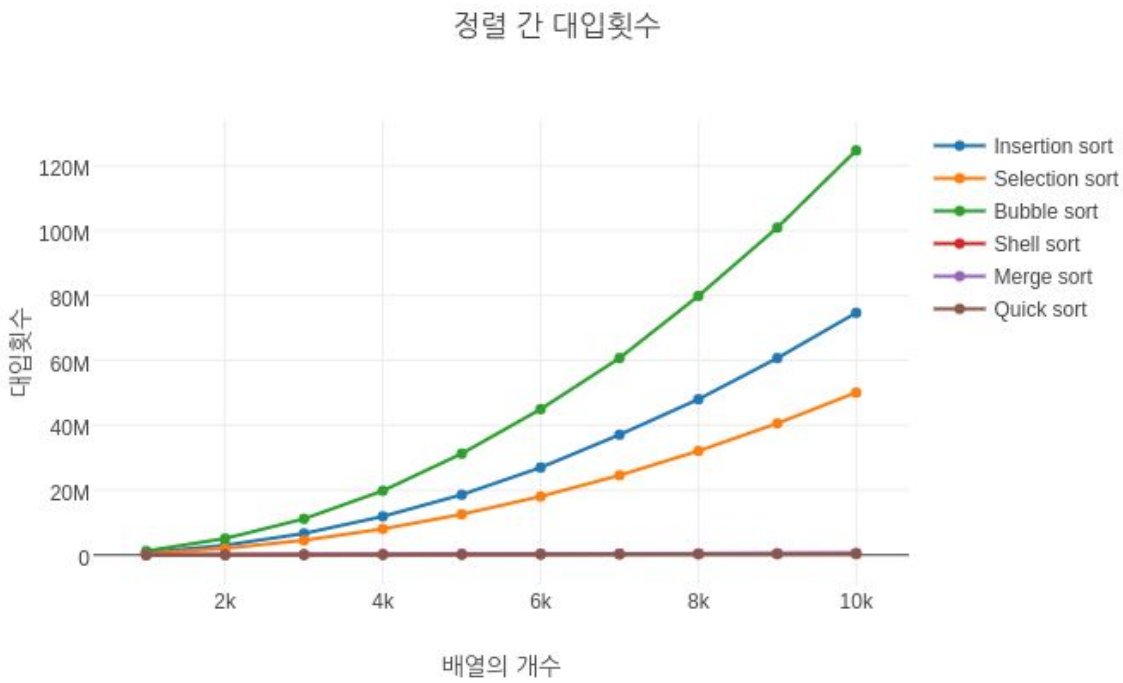
위 그래프는 6가지 정렬의 비교횟수를 선 그래프로 나타낸 것이다. 그래프를 보면 선이 세 개만 있는 것처럼 보이는데, 먼저 선택정렬은 버블정렬과 횟수가 거의 비슷해서 선이 겹쳐서 보이지 않는다. 그리고 쉘, 병합, 퀵 정렬의 경우 위의 세 정렬의 단위가 너무 커서 어느 정도 차이가 있는 세 정렬이 하나의 선으로 겹쳐 보이고 있다. 단위가 너무 차이가 나서 이 선들을 상용로그로 크기를 줄여 다시 보기로 했다.



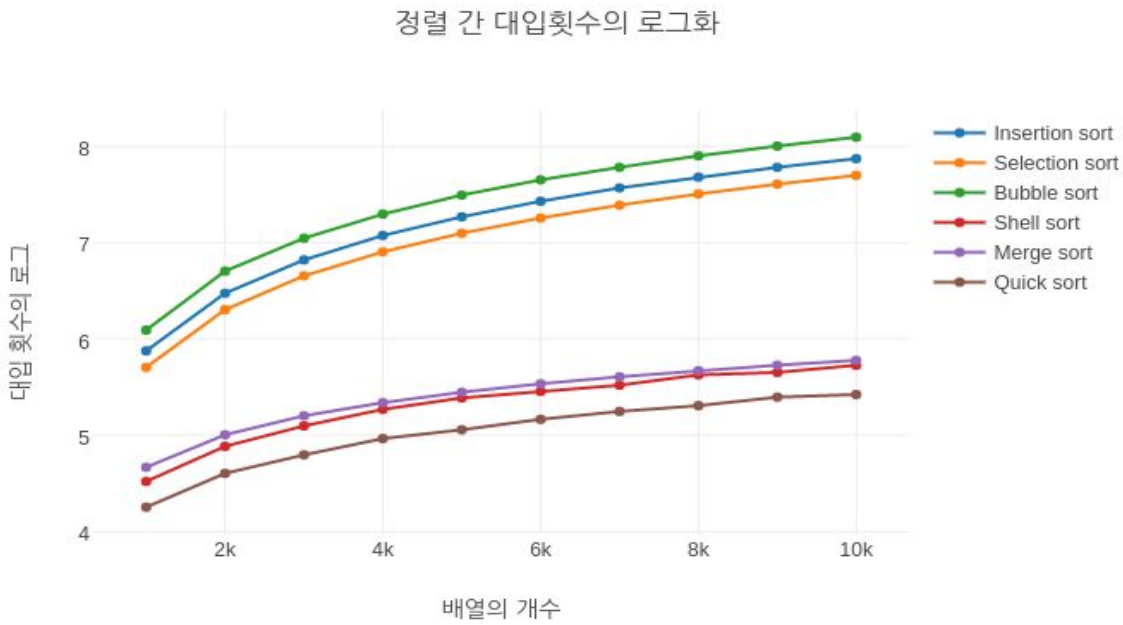
맨 위의 그래프를 로그로 치환하니 조금 더 비교가 수월해졌다. 버블정렬과 선택정렬은 횟수가 너무 비슷해서 여기서도 선이 겹쳐 보인다. 그렇지만 밑의 세 정렬은 대소관계를 파악할 수 있게 되었다. 비교에서는 횟수가 쉘 > 퀵 > 병합의 순서로 더 많았다. 결론적으로 비교 횟수가 더 적을수록 더 좋은 알고리즘이라고 할 때 알고리즘의 성능은

선택 < 버블 < 삽입 < 쉘 < 퀵 < 병합

순으로 더 좋다고 생각할 수 있다.



다음은 대입횟수이다. 횟수가 적은 밑의 세 정렬은 여전히 선이 겹쳐 대소관계를 파악하기 힘들지만 대입과 달리 버블과 선택정렬의 횟수가 확연히 구분되어 있다. 대입에서는 삽입정렬보다도 더 적은 위치에 있음을 확인할 수 있다. 밑의 세 정렬의 성능을 파악하기 위해 다시 로그화하였다.



병합정렬이 제일 횟수가 많고, 쉘정렬이 다음, 퀵정렬이 제일 적음을 확인할 수 있다. 정리하면 변수의 대입에 있어서 대입 횟수가 적을수록 성능이 좋다고 가정할 때 알고리즘의 성능은

버블 < 삽입 < 선택 < 병합 < 쉘 < 퀵

순으로 우수하다고 평가할 수 있다. 이때 주목할 것은 선택, 버블, 삽입보다 쉘, 퀵, 병합정렬이 압도적으로 우수하다는 것이다. 비교와 대입 모두에서 나타난 현상으로 버블, 삽입, 선택간의 순위는 크게 중요하지 않았다. 병합, 쉘, 퀵 사이에서의 순위도 마찬가지이다.

4. 결론

병합, 퀵, 쉘 정렬은 삽입, 선택, 버블정렬보다 압도적으로 우수하며 거의 대부분의 상황에서 더 좋은 정렬이라고 판단된다. 측정치가 10,000일 때 대입과 비교 모두에서 병합, 퀵, 쉘정렬의 횟수는 삽입, 선택, 버블정렬의 약 0.1% 수준으로 측정되었다. 또한 특이한 점은 정렬하는 배열의 크기가 커질수록 성능 상위 3개 정렬과 하위 3개 정렬의 성능차는 더욱 벌어진다. 배열의 크기가 커질수록 성능 좋은 알고리즘의 필요성을 더욱 절감하게 될 것 같다. 그리고 상위 3개의 알고리즘간, 하위 3개의 알고리즘간 성능 차이를 이 자료만으로 판단하기는 힘들 것이라 판단된다. 대입과 비교에서의 순위가 조금씩 다르고, 무엇보다 대입과 비교 중 어떤 지표가 알고리즘의 성능에 더 큰 영향을 미치는지 단언할 수 없기 때문이다. 추가적인 공부가 필요할 것 같다.

개인적으로는 처음 빅오 시간복잡도 계산법을 배웠을 때 최고차항의 계수를 떼어낸다는 것에 반감이 있었는데 이번 실험을 통해서 납득하게 되었다. 계수보다는 최고차항 자체가 훨씬 중요하다는 것을 알게 되어서 앞으로 알고리즘 공부할 때 도움이 많이 될 것 같다. 또한 C 헤더 파일을 처음 만들어보고 함수 포인터를 써보는 등 C 공부를 많이 할 수 있어서 좋았다. 2년 전 C를 한 달 동안 독학하다 포기했었는데 알고리즘 수업을 통해 이제는 어느 정도 익숙해진 것 같다. 더 공부해도 재미있겠다.
마지막으로 데이터의 시각화가 의미 있다는 것을 다시 느꼈다. 표를 작성하고 나서 데이터가 제대로 보이지 않는다는 느낌을 받았는데 표로 작성하니 뚜렷하게 차이를 볼 수 있었다. 또한 데이터 크기가 압도적으로 차이가 나서 log화를 시켰는데 처음 해보는 시도였는데 데이터를 좀 더 잘 살펴 볼 수 있었다. 앞으로도 자주 사용할 수 있는 방법이라고 생각한다.