알고리즘설계와분석.HW3

Sorting 방법의 효율적인 구현

이름 : 이예준

학번 : 20212022

1. 목적

다양한 Sorting 알고리즘(Insertion, Heap, Quick, Merge 등)을 구현하고,

각 알고리즘의 성능을 비교 및 분석하였다. 이를 통해 각 Sorting 방법의 시간 복잡도와 실제 실행 시간이 어떻게 연관되는지 확인하고, 최적화 기법의 효과를 분석하였다.

2. 구현 방법

- Heap Sort

```
#define SWAP(a,b) {RECORD tmp; tmp = a; a = b; b = tmp;}
```

```
void max_adjust(RECORD records[], int i, int j) {
    int child, rootkey;
    RECORD tmp;
    tmp = records[i];
    rootkey = records[i].key;
    child = 2 * i + 1;

    while (child <= j) {
        if (child < j && (records[child].key < records[child + 1].key))
            child++;
        if (rootkey > records[child].key)
            break;
        else {
            records[(child - 1) / 2] = records[child];
            child = 2 * child + 1;
        }
    }
    records[(child - 1) / 2] = tmp;
}
```

```
void RECORDS::sort_records_heap(int start_index, int end_index) {
    // Classic heap sort
    for (int i = (end_index-1) / 2; i >= 0; i--)
        max_adjust(records, i, end_index);
    for (int i = end_index; i >= 0; i--) {
        SWAP(records[0], records[i]);
        max_adjust(records,0, i-1);
    }
}
```

- 먼저 records 구조체 배열을 Max Heap의 구조로 만들어주기 위해

  (end_index-1)/2 인덱스부터 root 노드 인덱스인 0까지 순회하면서 노드들의 위치를

  Max Heap으로 조정해주는 max_adjust함수를 호출했다.

(end_index-1)/2 인덱스부터 시작하는 이유는 child가 있는 노드들 중 가장 Depth가 깊은 노드의 인덱스이기 때문이다. 즉, 다시 말해 이 인덱스 밑으로는 child가 없는 노드들이기 때문에 위치 조정이 필요 없다.

- max_adjust함수에서는 현재 노드에서 자식 노드로 계속 내려가면서 자식 노드를 비교하여 만약 자식 노드가 더 크다면 현재 노드와 자식 노드의 위치를 맞바꾼다.
  현재 노드가 자식 노드보다 더 크면 Max Heap 조건을 만족하므로 함수를 종료한다.

- 위에서 records 구조체 배열을 Max Heap의 구조로 변환했다면 이제 정렬단계를 거친다.
  Max Heap 구조를 가지고 있기 때문에 배열의 첫 인덱스에는 최댓값이 저장되어 있다.
  따라서 배열의 첫 인덱스에 있는 구조체를 배열의 마지막 인덱스로 이동시킨다.
  그 다음 Heap의 크기를 줄여서 이동시킨 노드를 포함하지 않도록 한 다음에 max_adjust함수를 호출한다. 그러면 크기를 줄인 Heap에서의 최댓값이 다시 배열의 첫 인덱스로 이동할 것이다. 이렇게 Heap의 root 노드를 배열의 끝으로 이동시키고, Heap의 크기를 줄인 뒤 max_adjust함수를 호출하는 과정을 반복하면 마지막에는 정렬된 records 구조체 배열이 남아있게 된다.

- Weird Sort (먼저 Min Heap을 구성 후에 Insertion Sort를 적용)

```
void min_adjust(RECORD records[], int i, int j) {
    int child, rootkey;
    RECORD tmp;
    tmp = records[i];
    rootkey = records[i].key;
    child = 2 * i + 1;

    while (child <= j) {
        if (child < j && (records[child].key > records[child + 1].key))
            child++;
        if (rootkey < records[child].key)
            break;
        else {
            records[(child - 1) / 2] = records[child];
            child = 2 * child + 1;
        }
    }
    records[(child - 1) / 2] = tmp;
}
```

```
void RECORDS::sort_records_weird(int start_index, int end_index) {
    // A weird sort with a make-heap operation followed by insertion sort
    for (int i = (end_index-1) / 2; i >= start_index; i--)
        min_adjust(records, i, end_index-1);

    sort_records_insertion(start_index, end_index);
}
```

- Heap Sort처럼 맨 처음에 구조체 배열을 Heap 구조로 만들어 준다.

  단, 여기서는 Max Heap이 아니라 Min Heap 구조로 조정해주기 위해 min_adjsut함수를 호출한다. 조정을 마치면 그 배열을 가지고 insertion sort 함수를 이용해서 정렬해준다.

- Insertion Sort는 배열을 순회하면서 각 인덱스에 있는 값을 적절한 위치로 이동시키면서 정렬을 해주는 정렬 방법인데, 먼저 Min Heap 구조로 만들어 특정한 순서를 가지게 한 뒤 Insertion Sort를 진행하여 Insertion Sort의 성능을 향상시키려는 목적을 가지고 구현했다.

- Quick Sort

```
void RECORDS::sort_records_quick_classic(int start_index, int end_index) {
    // Classic quick sort without any optimization techniques applied
    int pivot;

    if (start_index < end_index) {
        pivot = start_index;
        for (int i = start_index; i < end_index; i++) {
            if (compare_keys((const void*)(&records[i]), (const void*)(&records[end_index])) < 0) {
                SWAP(records[i], records[pivot]);
                pivot++;
            }
        }
        SWAP(records[end_index],records[pivot]);

        sort_records_quick_classic(start_index, pivot - 1);
        sort_records_quick_classic(pivot + 1,end_index);
    }
}
```

- 구조체 배열에서 하나의 요소를 pivot으로 선택한다음 pivot을 기준으로 작은 값들로 이루어진 부분 배열과 큰 값들로 이루어진 부분 배열로 나눈다. 분할된 두 배열은 다시 재귀적으로 sort_records_quick_classic 함수를 호출하며 분할된 모든 배열이 정렬되면 이들을 합쳐 최종적으로 전체 배열이 정렬되게 한다.

- Intro Sort

```
void RECORDS::sort_records_intro(int start_index, int end_index) {
    // Introsort described in https://en.wikipedia.org/wiki/Introsort
    int maxDepth = 2 * (log2(get_size()));

    auto introsort = [&](int n, int depth, int left, int right, auto& intro_sort_ref) -> void {
        if (n < 16) {
            sort_records_insertion(left, right);
        }
        else if(!depth){
            sort_records_heap(left, right);
        }
        else {
            int pivot = left;
            for (int i = left; i < right; i++) {
                if (compare_keys((const void*)(&records[i]), (const void*)(&records[right])) < 0) {
                    SWAP(records[i], records[pivot]);
                    pivot++;
                }
            }
            SWAP(records[right], records[pivot]);

            intro_sort_ref(pivot - left, depth - 1, left, pivot - 1, intro_sort_ref);
            intro_sort_ref(right - pivot, depth - 1, pivot + 1, right, intro_sort_ref);
        }
    };

    introsort(get_size(),maxDepth,start_index,end_index,introsort);
}
```

- Intro sort는 Quick Sort의 평균적인 성능과 Heap Sort의 최악의 성능 보장을 결합한 하이브리드 정렬 알고리즘이다. 또한 작은 배열에 대해서는 Insertion Sort를 사용하여 Cache 효율성을 높이는 최적화도 포함하고 있다.

- 최대 재귀 깊이를 $2 * \log_2(n)$로 설정하여 재귀 호출의 깊이를 제한한다. 재귀 깊이가 최대치를 넘으면 Heap Sort로 전환하여 최악의 경우에도 Heap Sort의 시간 복잡도를 보장한다.

- 함수 내부에는 람다 함수를 정의하여 재귀적으로 정렬을 수행한다. if문을 통해서 상황에 따라 Sort 방법을 바꿔준다. 만약 부분 배열의 크기가 16미만이면 Insertion Sort를 수행한다. 또는 만약 재귀 깊이가 최대치를 넘으면 Heap Sort를 수행한다. 그 외의 경우에는 pivot을 이용한 Quick Sort를 수행하여 배열들을 분할한다.

- ***Merge-Insertion Sort***

```
void merge(RECORD records[], int left, int right) {
    RECORD* list = new RECORD[right - left + 1];

    int mid = (left + right) / 2;
    int i = left;
    int j = mid + 1;
    int k = 0;

    while (i <= mid && j <= right) {
        if (records[i].key < records[j].key) {
            list[k++] = records[i++];
        }
        else {
            list[k++] = records[j++];
        }
    }

    while (i <= mid) {
        list[k++] = records[i++];
    }
    while (j <= right) {
        list[k++] = records[j++];
    }

    for (int l = left; l <= right; l++) {
        records[l] = list[l - left];
    }

    delete[] list;
}
```

```
void RECORDS::sort_records_merge_with_insertion(int start_index, int end_index) {
    // Merge sort optimized by insertion sort only
    if (end_index - start_index < 16) {
        sort_records_insertion(start_index, end_index);
        return;
    }

    if (start_index < end_index) {
        int mid = (start_index + end_index) / 2;
        sort_records_merge_with_insertion(start_index, mid);
        sort_records_merge_with_insertion(mid + 1, end_index);

        merge(records, start_index, end_index);
    }
}
```

- Merge Sort를 기반으로 하여 작은 부분 배열에 대해서는 Insertion Sort를 수행하여 성능을 최적화한 함수이다. 원래의 Merge Sort는 배열을 두개의 부분 배열을 분할하고, 이 과정을 부분 배열의 크기가 1이 될 때까지 반복하는데 구현한 함수에서는 부분 배열의 크기가 16미만이 되면 Insertion Sort를 수행한다.

- Insertion Sort로 정렬된 두 부분 배열을 merge함수를 통해 합치게 되며, 이 과정은 재귀적으로 거꾸로 올라가면서 수행하여 전체 배열을 정렬한다.
  merge 함수는 정렬된 두 부분배열을 하나의 배열로 합쳐주는 함수로 Two Pointer를 이용해서 각 부분배열의 요소를 비교하여 새로 선언한 배열에 오름차순으로 요소들을 저장한다. 하나의 배열의 모든 요소를 저장했다면 남아 있는 배열의 요소들은 모두 새로 선언한 배열의 마지막 요소보다 크다는 뜻이므로 별다른 비교없이 저장해주면 된다.
  마지막으로 병합된 배열을 원본 배열 records의 해당 위치에 복사한다.

3. 적용시킨 최적화 기법

- Merge Sort와 Intro Sort에서 부분 배열의 크기가 16미만일 경우 Insertion Sort를 수행하게 하여 Cache 효율성을 높였다.
- Intro Sort에서 재귀 높이를 제한하여 최악의 경우 Heap Sort로 전환하여 최대한 최적의 시간 복잡도를 보장하였다.

4. 실험 방법

- 입력 데이터 생성

무작위로 생성된 정수 키를 가진 RECORD 구조체 배열을 사용했다.

동일한 입력 데이터로 모든 정렬 알고리즘을 테스트하여 공정한 비교를 수행했다.

- 입력 크기 설정

2^{14} ~ 2^{20} 까지 다양한 입력 크기(n)를 선정하여 성능 변화를 관찰했다.

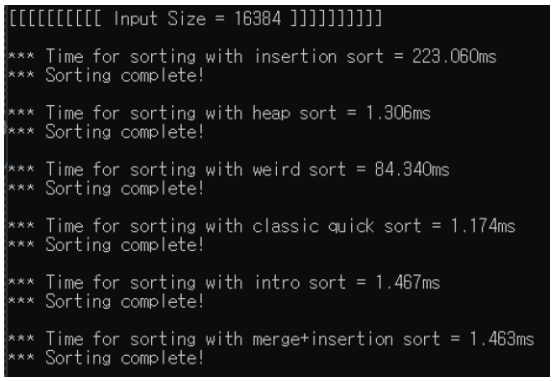
- 시간 측정 방법

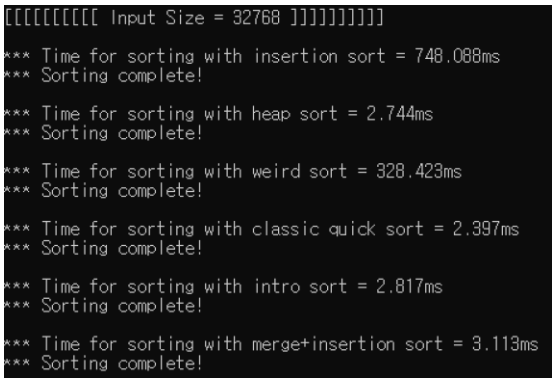
각 정렬 함수 호출 전후로 시간을 기록하여 실행 시간(ms)을 측정했고,

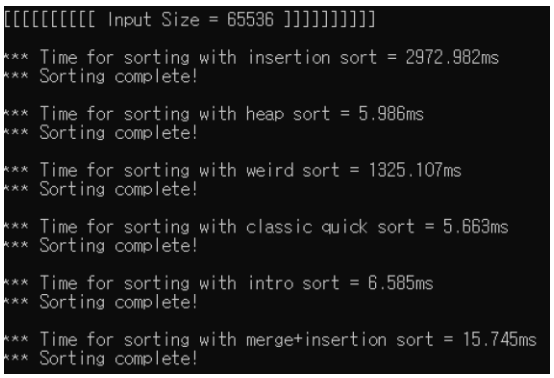
평균값의 신뢰성을 높이기 위해 각 입력 크기와 정렬 방법에 대해 여러 번의 실험을 수행하고 평균을 계산했다.

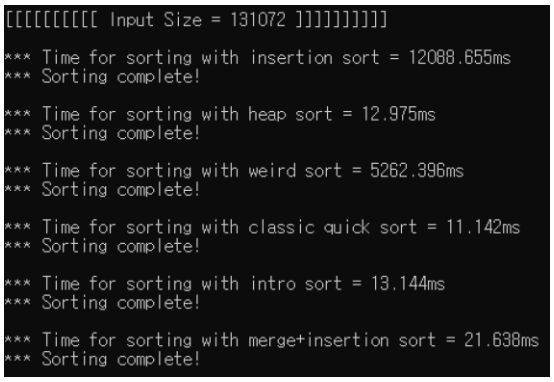
5. 실험 결과

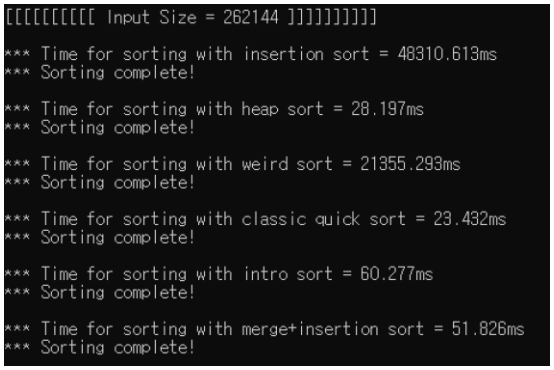
- 각 Sorting 함수에 대한 시간 측정 캡쳐

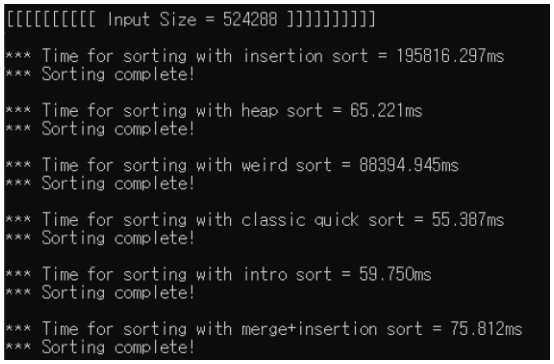


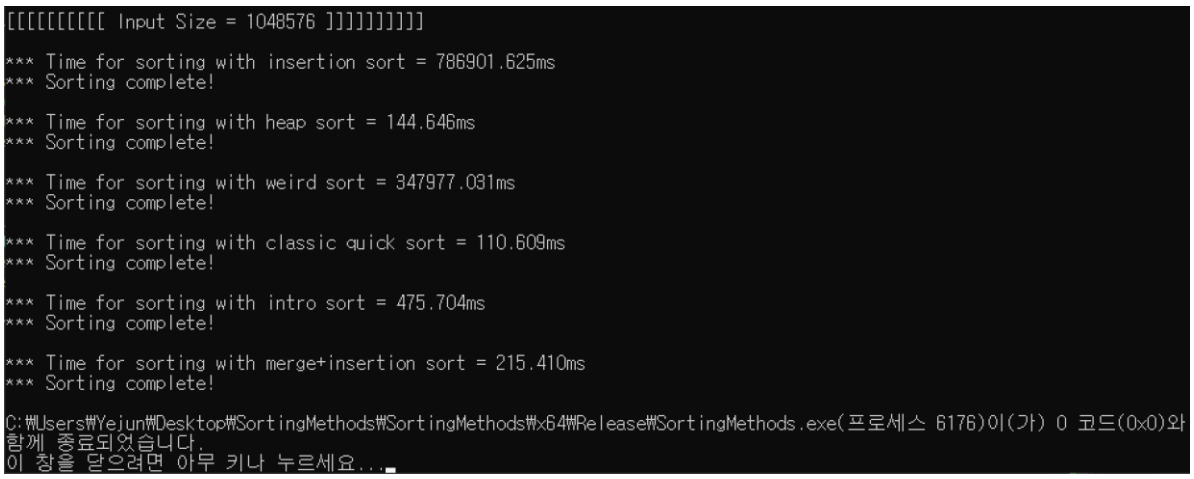












-시간 측정 결과표 (시간 단위 : ms)

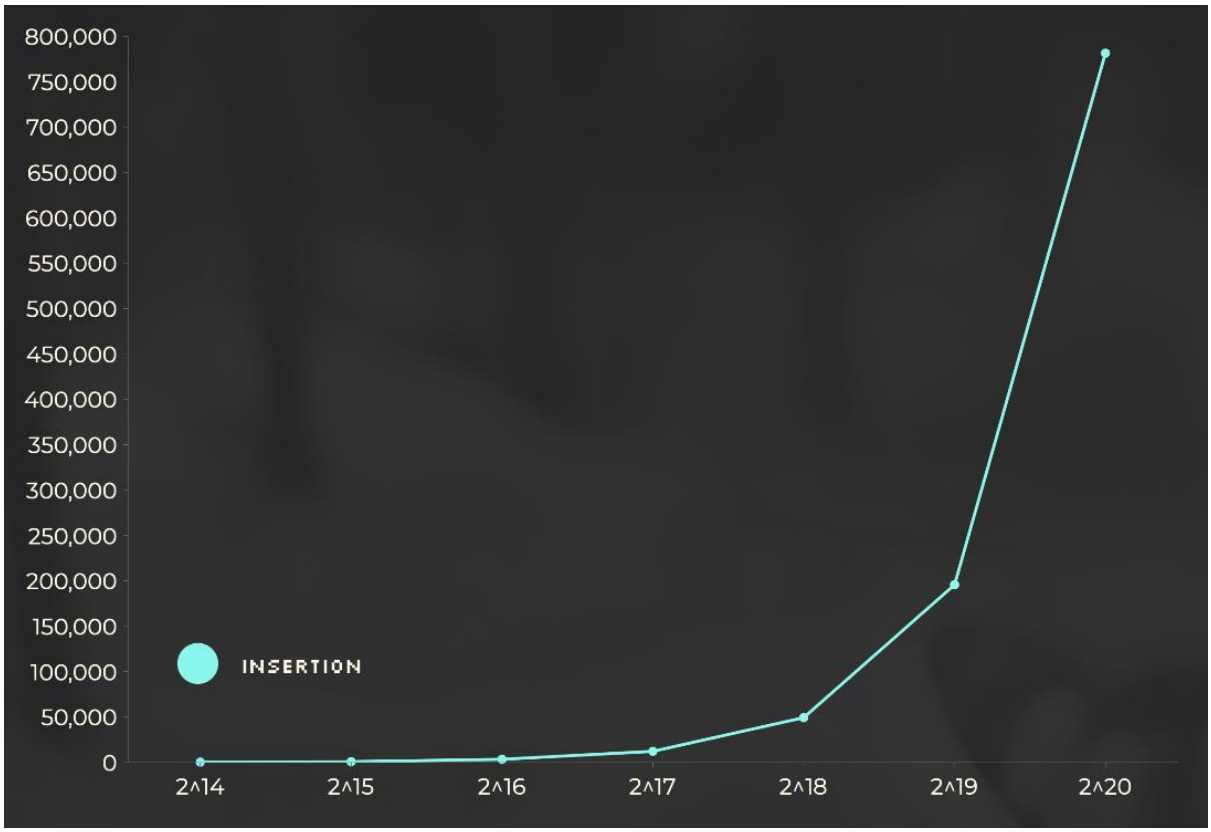
Sorting	Input (n)	Case			Average
		1st	2nd	3rd	
Insertion	2^{14}	216.70	196.50	195.40	**202.80**
	2^{15}	747.90	746.70	766.30	**753.60**
	2^{16}	3550.0	3201.0	3265.0	**3338.0**
	2^{17}	12128	12092	12148	**12122**
	2^{18}	49573	48139	49956	**49222**
	2^{19}	195816	199219	191862	**195632**
	2^{20}	786901	784918	771842	**781220**
Heap	2^{14}	1.5620	1.2930	1.3460	**1.4000**
	2^{15}	3.8930	2.8160	3.0010	**3.2360**
	2^{16}	7.7790	6.3300	9.1070	**7.7380**
	2^{17}	14.180	13.076	13.017	**13.424**
	2^{18}	30.220	28.130	31.682	**30.010**
	2^{19}	65.221	62.194	60.814	**62.743**
	2^{20}	144.646	139.184	140.673	**141.501**
Weird	2^{14}	86.080	86.220	85.260	**85.850**
	2^{15}	326.30	336.30	335.90	**332.83**
	2^{16}	1393.0	1392.0	1374.0	**1386.3**
	2^{17}	5407.4	5363.4	5305.0	**5358.6**
	2^{18}	21943	20184	21671	**21266**
	2^{19}	88394	88842	87472	**88236**
	2^{20}	347977	346828	341593	**345466**
Quick	2^{14}	1.2750	1.2300	1.1680	**1.2243**
	2^{15}	2.474	2.581	2.534	**2.5296**
	2^{16}	5.6380	5.7580	5.5000	**5.6320**
	2^{17}	11.346	11.589	12.011	**11.648**
	2^{18}	23.988	24.182	23.715	**23.961**
	2^{19}	55.387	55.105	54.926	**55.139**
	2^{20}	110.60	112.19	109.48	**110.75**
Intro	2^{14}	1.4990	1.3950	1.4690	**1.4540**
	2^{15}	2.9530	3.0110	2.9330	**2.9650**
	2^{16}	6.8210	6.7030	7.2890	**6.9376**

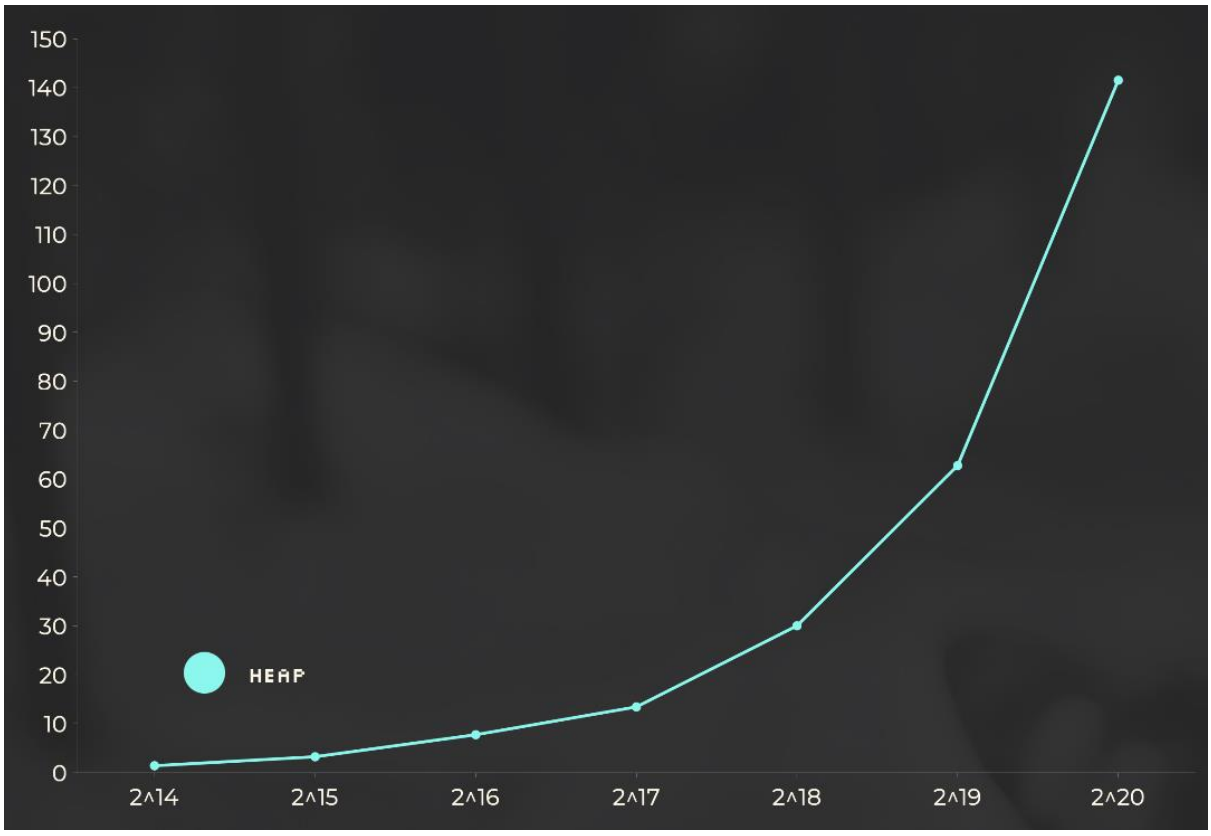
	2^{17}	13.385	13.211	14.259	**13.618**
	2^{18}	64.756	61.184	60.582	**62.174**
	2^{19}	61.478	61.416	61.518	**61.470**
	2^{20}	484.535	486.952	505.732	**492.406**
Merge - Insertion	2^{14}	1.5380	1.2030	1.3180	**1.3530**
	2^{15}	2.7210	2.7940	2.3340	**2.6160**
	2^{16}	7.5440	10.210	12.610	**10.121**
	2^{17}	26.038	18.032	28.577	**24.215**
	2^{18}	31.345	35.017	34.381	**33.581**
	2^{19}	96.362	108.962	118.379	**107.901**
	2^{20}	169.054	194.946	371.520	**245.173**

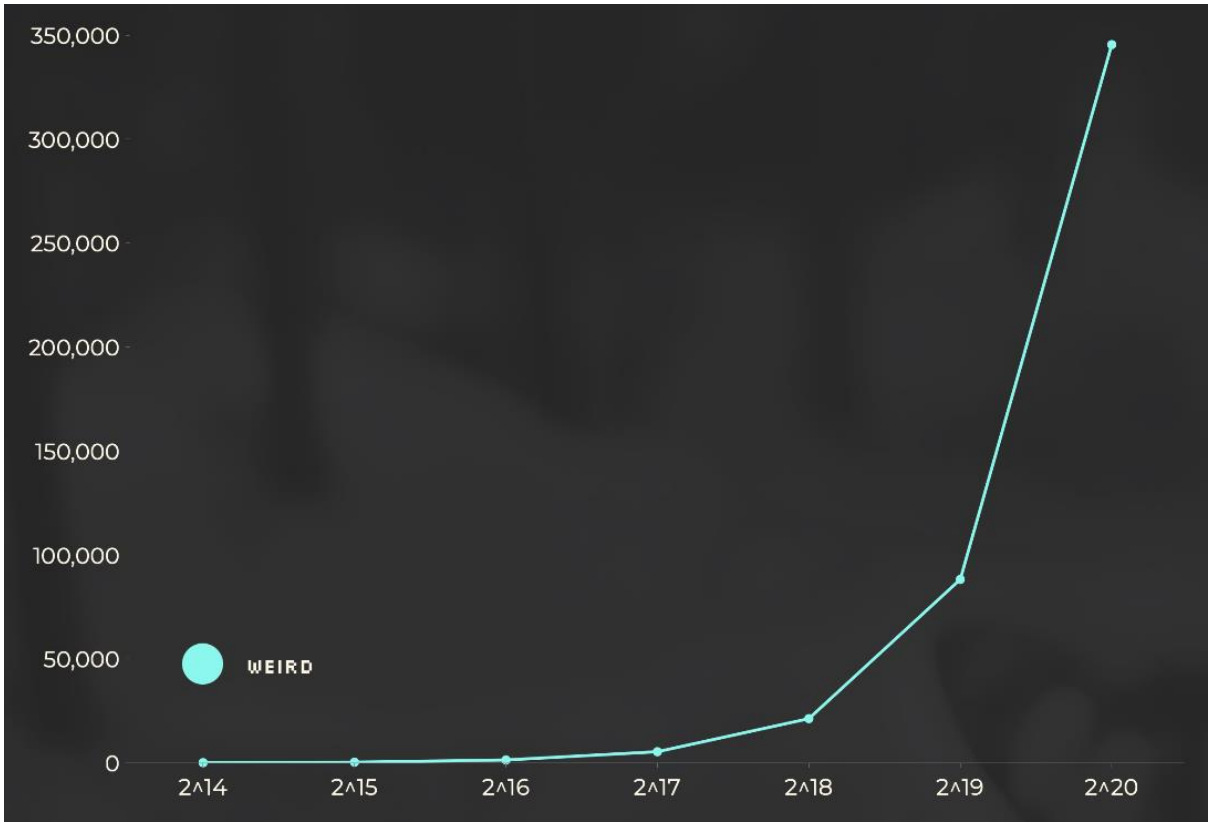
-평균 시간 측정 결과표 (시간 단위 : ms)

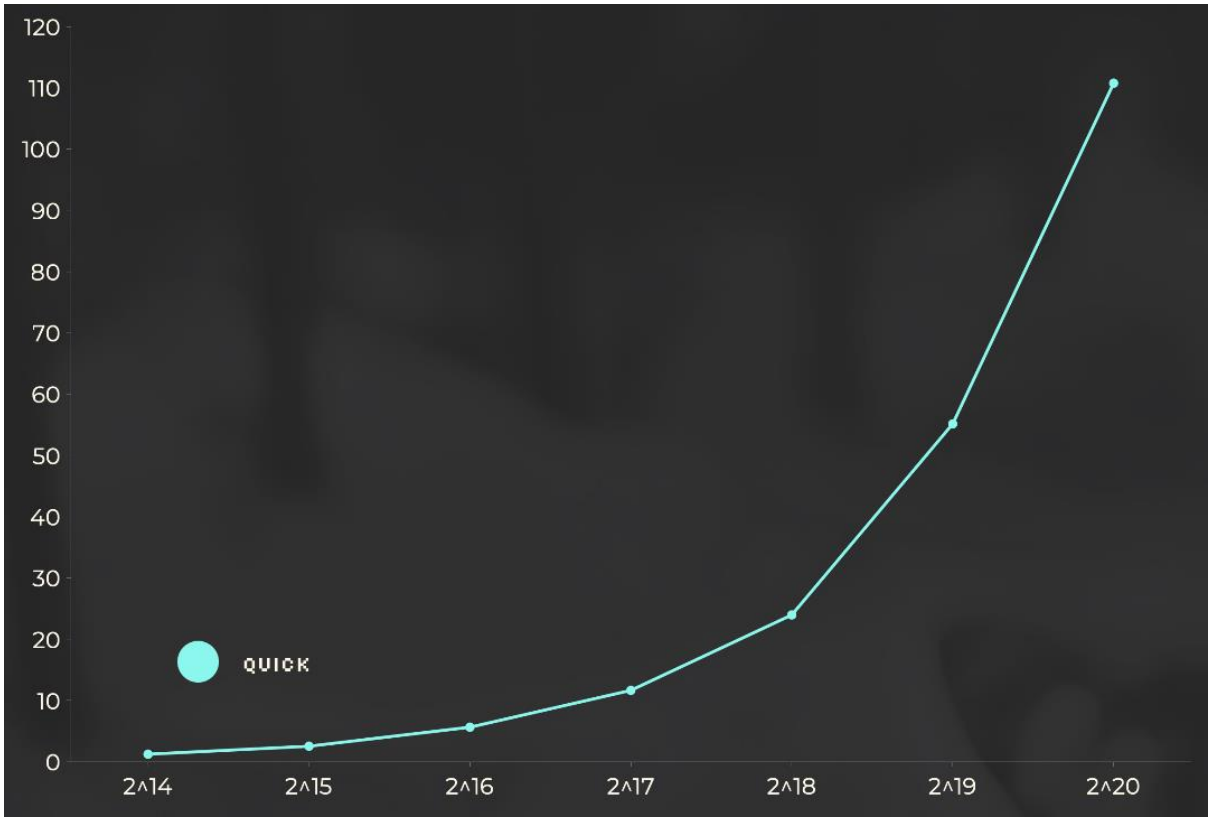
Input(n)	Insertion	Heap	Weird	Quick	Intro	Merge - Insertion
2^{14}	**202.80**	**1.4000**	**85.850**	**1.2243**	**1.4540**	**1.3530**
2^{15}	**753.60**	**3.2360**	**332.83**	**2.5296**	**2.9650**	**2.6160**
2^{16}	**3338.0**	**7.7380**	**1386.3**	**5.6320**	**6.9376**	**10.121**
2^{17}	**12122**	**13.424**	**5358.6**	**11.648**	**13.618**	**24.215**
2^{18}	**49222**	**30.010**	**21266**	**23.961**	**62.174**	**33.581**
2^{19}	**195632**	**62.743**	**88236**	**55.139**	**61.470**	**107.901**
2^{20}	**781220**	**141.501**	**345466**	**110.75**	**492.406**	**245.173**

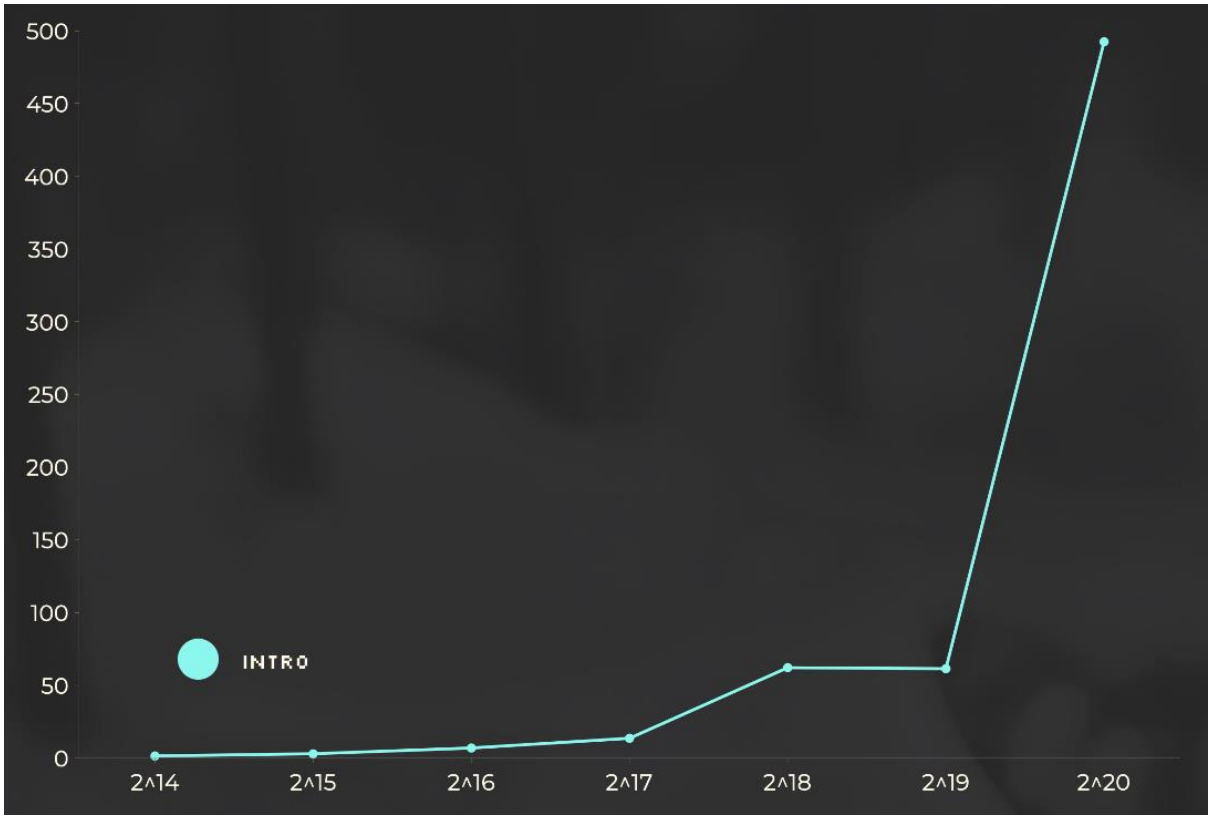
-평균 시간 측정 결과 그래프 (시간 단위 : ms)

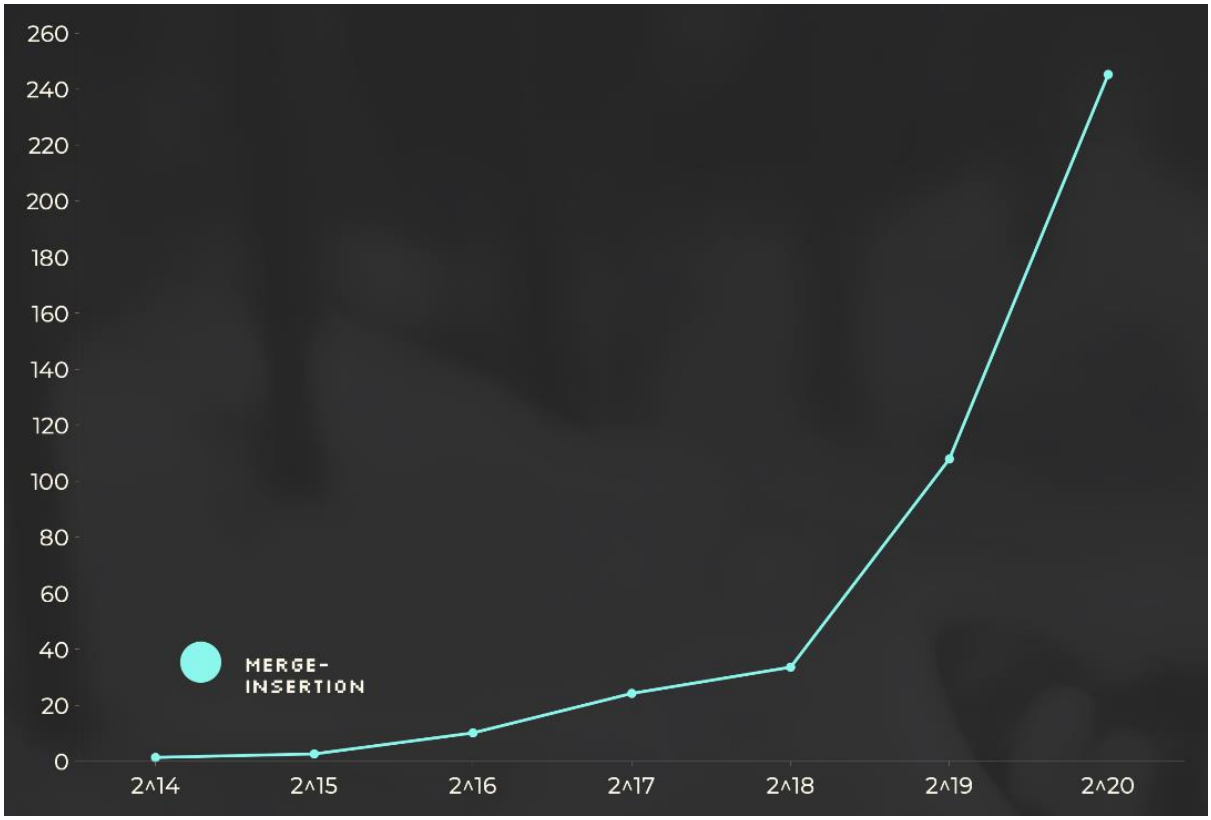





350,000
300,000
250,000
200,000
150,000
100,000
50,000
0
WEIRD
2^14
2^15
2^16
2^17
2^18
2^19
2^20


120
110
100
90
80
70
60
50
40
30
20
10
0
QUICK
2^14
2^15
2^16
2^17
2^18
2^19
2^20


500
450
400
350
300
250
200
150
100
50
0
INTRO
2^14
2^15
2^16
2^17
2^18
2^19
2^20


260
240
220
200
180
160
140
120
100
80
60
40
20
0
MERGE-
INSERTION
2^14
2^15
2^16
2^17
2^18
2^19
2^20

6. 이론적인 시간 복잡도와 실행 시간 비교

- Insertion Sort: **$O(n^2)$**

입력 크기가 증가할수록 실행 시간이 급격히 증가하며,

이론적인 시간 복잡도 $O(n^2)$과 일치하는 패턴을 보였다.

큰 입력 크기에서는 실행 시간이 매우 길어 효율성이 떨어진다.

- Heap Sort: **O(n log n)**

모든 입력 크기에서 일정하게 O(n log n)의 성능을 보였으며, 큰 입력에서도 효율적이었다.

이론적 시간 복잡도와 실제 실행 시간이 일치하는 것을 확인했다.

- Weird Sort: **$O(n^2)$**

Insertion Sort보다는 약간 더 빠른 실행 시간이 나왔지만 최적화가 제대로 이루어지지 않아

비효율적인 성능을 보였다.

이 알고리즘의 특성상, 이론적 분석과 일치하는 비효율적 결과가 나왔다.

- Quick Sort: **O(n log n) (Average)**

평균적으로 O(n log n)의 성능을 보였으나, 분할 방식으로 인해 일부 입력에서

성능 저하가 발생할 가능성이 있다.

그러나 전반적으로 이론적 시간 복잡도와 일치하는 성능을 보였다.

- Intro Sort: **O(n log n)**

재귀 깊이가 일정 수준을 초과할 경우 Heap Sort로 전환하여 O(n log n)을 보장하므로,

큰 입력에서도 안정적이었다.

이는 Quick Sort의 성능을 유지하면서도 안정성을 제공하여, 최적화가 효과적으로

이루어졌다.

** Input 2^{18}과 2^{19}에 대해 비슷한 실행시간이 나왔는데 그 이유에 대해서*

Quick Sort 전환으로 인해 비교적 일정한 성능을 유지하거나

메모리 접근 패턴이 일정하게 유지했기 때문이라고 예상할 수 있다.

** intro sort는 이론적으로 평균, 최악일 때 시간복잡도가 O(n log n)이고, quick sort는 평균일 때 O(n log n), 최악일 때 $O(n^2)$ 이다. 따라서 실제로 시간을 측정할 때 일반적으로 intro sort가 더 빨라야 하지만 결과를 보면 그렇지 않다는 것을 볼 수 있다. 그래서 따로 시간측정을 여러번 더 반복한 결과 intro sort가 더 빠른 경우도 나오는 것을 보아 이론적인 시간복잡도와 일치하기에는 Case의 수가 부족한 것이 원인이 아닐까라는 결론에 도달했다.*

- Merge-Insertion Sort: **O(n log n)**

Merge Sort의 시간 복잡도인 O(n log n)을 유지하였으며, 작은 배열에 대한 Insertion Sort 적용으로 효율성이 향상되었다.

큰 입력에서도 안정적이지만, 병합 시 추가 메모리 사용으로 인해 Heap Sort이나 Quick Sort보다 다소 느린 성능을 보인다.

7. 최적화 기법의 효과 분석

- Merge-Insertion Sort와 Intro Sort에서 작은 부분 배열에 대한 Insertion Sort적용은 확실한 성능 향상을 가져왔다. 작은 배열에 대해 Insertion Sort가 높은 Cache hit rate와 낮은 Overhead를 가져와, 전반적인 실행 시간을 줄이는 데 기여했다.

- Intro Sort에서 재귀 깊이 제한을 통해 Quick Sort의 최악 성능을 방지하고, 안정적인 성능을 보장했다. 큰 입력에서도 성능이 저하되지 않도록 Heap Sort로 전환하는 방식은 효과적인 최적화 기법으로 평가할 수 있을 것 같다.

- 반면 Weird Sort는 Min Heap을 만든 후 Insertion Sort를 수행하는 비효율적인 구조로 인해, 오히려 성능이 저하되었다. 이 알고리즘은 학습 목적 외에 실제 Sorting 알고리즘으로 활용하기에는 적합하지 않음을 확인했다.

5. Setting

-실험환경

OS: macOS Sonoma 14.3.1

CPU: Apple M1

RAM: 8.00GB

-가상머신 실험환경

OS: Windows 11 Home

CPU: virt-7.2 1GHz

RAM: 4.00GB

Compiler: Visual Studio 22 Release Mode/ARM64 Platform