CSE3081-01 알고리즘 설계와 분석

[숙제 3]
Sorting 방법의
효율적인 구현

20231604 컴퓨터공학과 정유연

2024-09-30

<목차>

1. 컴퓨터 실험환경

- OS: Windows 11 Home
- CPU: 13th Gen Intel(R) Core(TM) i5-1340P 1.90 GHz
- RAM: 16.00GB
- Compiler: Visual Studio 22 Release Mode/x64 Platform

제출한 소스파일: HW3_S20231604.cpp, HW3_S20231604.h

2. 정렬 함수

Void sort_records_heap	Heapsort	O(nlogn)

교과서적인 heapsort함수로, adjust_max함수를 만들어 이용했다.

코드 시작 전 인덱스 0부터 시작하는 subarray인 tempRecord를 만들어 records array를 복사해서 이용한다. 따라서 위의 1. Make max heap에서 for문의 인덱스가 0까지 줄어들었고, 2. Extract에서 adjust함수도 인덱스 0을 넘겨주었다.

```
ll size = end_index - start_index;
RECORD* tempRecord = (RECORD*)malloc((size+ 1) * sizeof(RECORD));
if (tempRecord != nullptr) {
    memcpy(tempRecord, records + start_index, (size + 1) * sizeof(RECORD));
}
else printf("Allocation Error₩n");
```

먼저 (size+1)/n부터 0까지 adjust를 이용해 최대힙을 만들고, size부터 1까지 스왑한 다음 adjust함수를 적용한다.

```
// 1. Make a max heap
for (int i = (size + 1) / 2; i >= 0; i--) {
    adjust_max(tempRecord, i, size);
}

// 2. Extract items one by one.
for (int i = size; i > 0; i--) {
    temp = tempRecord[0];
    tempRecord[0] = tempRecord[i];
    tempRecord[i] = temp;
    adjust_max(tempRecord, 0, i - 1);
}
```

2. Extract에서 tempRecord[0]과 tempRecord[i]를 서로swap한다.

Adjust_max함수는 root, n을 입력받는다.

```
void RECORDS::adjust_max(RECORD arr[], int root, int n) {
    int child, rootkey;
    RECORD temp = arr[root];
    rootkey = arr[root].key;
    child = 2 * root + 1; //left child
    while (child <= n) {
        if ((child < n) && (arr[child].key < arr[child + 1].key))
            child++;
        if (rootkey > arr[child].key)
            break; // compare root and maxchild whether child is b
        else { // we need to keep going until child is bigger than
            arr[root] = arr[child]; //move to parent
            root = child;
            child = child * 2 + 1;
        }
    }
    arr[root] = temp;
}
```

부모 노드의 값이 자식 노드 보다 클 때까지 계속해서 내려가면서 검사한다.

이렇게 해서 가장 큰 값을 배열의 오른쪽에 쌓도록 한다.

Void sort_records_weird	Min heap을 만든 후 insertion	O(nlog n)

위의 heapsort에서는 최대힙을 만든 다음 맨 뒤와 처음 원소를 swap하여 adjust 함수를 적용했다. 이번 weird함수는 최소힙을 만든 다음, 원소를 하나씩 추출하는 것이 아니라 insertion함수를 이용했다.

최소힙을 만드므로 최대힙을 만드는 adjust_max와 다르게 adjust_min을 사용했다.

```
void RECORDS::sort_records_weird(int start_index, int end_index) {
    // A weird sort with a make min heap operation followed by insertion sort
    // 1. Make a min heap
    for (int i = (end_index - start_index + 1) / 2; i >= start_index; i--) {
        adjust_min(i, end_index);
    }
    // 2. Apply insertion sort
    sort_records_insertion(start_index, end_index);
```

Void sort_records_quick_classic	교과서적인 quick sort	O(n log n)

Partition 함수를 만들어 end_index가 start_index보다 큰 동안 분할 정복으로 정렬을 진행한다.

```
int RECORDS::partition(RECORD arr[], int left, int right)
{
    int j = left;
    int pivot = right;
    RECORD tmp;
    for (int i = left; i < right; i++) {
        if (arr[i].key < arr[pivot].key) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            j++;
        }
    }
    tmp = arr[pivot];
    arr[pivot] = arr[j];
    arr[j] = tmp;
    pivot = j;
    return pivot;
}
```

최적화 기법을 적용하지 않았기 때문에 pivot도 그냥 가장 끝 원소를 선택했다

I가 left부터 right까지 움직임에 따라 pivot과 i값을 비교하면서 j인덱스를 하나씩 옮긴다.

Void sort_records_intro	quick sort, heap sort, insertion sort를 합친 알고리즘	O(n log n)

위키피디아에 정의된 대로 구현한 introsort이다.

교수님께서 주어준 원형 함수에는 매개변수가 start_index와 end_index밖에 없어. 이에 추가해 Depth를 전달해주기 위해 real_intro(int start_index, int end_index, int

maxdepth) 를 새로 정의했다.

```
void RECORDS::real_intro(int start_index, int end_index, int maxdepth) {
    // Introsort described in https://en.wikipedia.org/wiki/Introsort
    int n = end_index - start_index + 1;

    if (n < 16) {
        sort_records_insertion(start_index, end_index);
    }
    else if (maxdepth == 0) {
        sort_records_heap(start_index, end_index);
    }
    else {
        int pivot = partition(records, start_index, end_index);
        real_intro(start_index, pivot - 1, maxdepth - 1);
        real_intro(pivot + 1, end_index, maxdepth - 1);
    }
}
```

N이 16보다 작을 때 insertion함수를 이용하고, depth가 0일 때는 앞서 만든 heap sort를 이용하고, 이외에는 quicksort를 이용한다.

Void sort_records_merge_with_insertion	mergesort다음에 insertion sort	O(n log n)

Mergesort를 이용하다가 N이 어떤 수보다 작을 때 insertion을 적용하는 함수다.

```
void RECORDS::sort_records_merge_with_insertion(int start_index, int end_index) {
    // Merge sort optimized by insertion sort only
    // Use insertion sort instead of combine
    if (end_index - start_index + 1 <= 16) {
        sort_records_insertion(start_index, end_index);
        return;
    }
    if (start_index < end_index) {
        int middle = (start_index + end_index) / 2;
        sort_records_merge_with_insertion(start_index, middle);
        sort_records_merge_with_insertion(middle + 1, end_index);
        merge(start_index, middle, end_index);
    }
}
```

- 실행 결과

실행 시간 분석은 4. 모든 정렬 함수에 대한 시간 비교에서 다룬다.

오른쪽 그림은 정렬이 올바르게 된 모습이다.

release mode일 때 2^14에서 2^20까지 모두 잘 출력되었다.

```
[[[[[[[[[[ Input Size = 16384 ]]]]]]]]]]

*** Time for sorting with insertion sort = 82.384ms
*** Sorting complete!

*** Time for sorting with heap sort = 1.206ms
*** Sorting complete!

*** Time for sorting with weird sort = 35.841ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 1.486ms
*** Sorting complete!

*** Time for sorting with intro sort = 1.185ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 1.746ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 32768 ]]]]]]]]]]

*** Time for sorting with insertion sort = 273.441ms
*** Sorting complete!

*** Time for sorting with heap sort = 3.316ms
*** Sorting complete!

*** Time for sorting with weird sort = 152.745ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 3.108ms
*** Sorting complete!

*** Time for sorting with intro sort = 2.722ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 23.843ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 65536 ]]]]]]]]]]

*** Time for sorting with insertion sort = 1087.166ms
*** Sorting complete!

*** Time for sorting with heap sort = 5.631ms
*** Sorting complete!

*** Time for sorting with weird sort = 477.325ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 6.021ms
*** Sorting complete!

*** Time for sorting with intro sort = 6.029ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 8.078ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 131072 ]]]]]]]]]]

*** Time for sorting with insertion sort = 4225.099ms
*** Sorting complete!

*** Time for sorting with heap sort = 13.909ms
*** Sorting complete!

*** Time for sorting with weird sort = 1925.515ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 12.668ms
*** Sorting complete!

*** Time for sorting with intro sort = 14.326ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 17.712ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 262144 ]]]]]]]]]]

*** Time for sorting with insertion sort = 17270.678ms
*** Sorting complete!

*** Time for sorting with heap sort = 47.803ms
*** Sorting complete!

*** Time for sorting with weird sort = 7637.349ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 31.066ms
*** Sorting complete!

*** Time for sorting with intro sort = 27.038ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 37.308ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 524288 ]]]]]]]]]]

*** Time for sorting with insertion sort = 71590.359ms
*** Sorting complete!

*** Time for sorting with heap sort = 110.604ms
*** Sorting complete!

*** Time for sorting with weird sort = 30851.428ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 58.930ms
*** Sorting complete!

*** Time for sorting with intro sort = 54.528ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 74.003ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 1048576 ]]]]]]]]]]

*** Time for sorting with insertion sort = 337688.938ms
*** Sorting complete!

*** Time for sorting with heap sort = 369.699ms
*** Sorting complete!

*** Time for sorting with weird sort = 159210.438ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 103.047ms
*** Sorting complete!

*** Time for sorting with intro sort = 129.651ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 151.167ms
*** Sorting complete!

C:\Users\mjddb\Desktop\HW3_S20231604\SortingMethods\x64\Release\Sort
이 창을 닫으려면 아무 키나 누르세요...
```

```
[[[[[[[[[[ Input Size = 16384 ]]]]]]]]]]

*** Time for sorting with insertion sort = 54.767ms
*** Sorting complete!

*** Time for sorting with heap sort = 1.154ms
*** Sorting complete!

*** Time for sorting with weird sort = 26.160ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 1.038ms
*** Sorting complete!

*** Time for sorting with intro sort = 0.932ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 1.588ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 32768 ]]]]]]]]]]

*** Time for sorting with insertion sort = 223.291ms
*** Sorting complete!

*** Time for sorting with heap sort = 2.313ms
*** Sorting complete!

*** Time for sorting with weird sort = 86.571ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 4.472ms
*** Sorting complete!

*** Time for sorting with intro sort = 4.357ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 5.442ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 65536 ]]]]]]]]]]

*** Time for sorting with insertion sort = 886.201ms
*** Sorting complete!

*** Time for sorting with heap sort = 8.487ms
*** Sorting complete!

*** Time for sorting with weird sort = 351.992ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 4.654ms
*** Sorting complete!

*** Time for sorting with intro sort = 4.929ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 6.602ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 131072 ]]]]]]]]]]

*** Time for sorting with insertion sort = 3483.285ms
*** Sorting complete!

*** Time for sorting with heap sort = 10.932ms
*** Sorting complete!

*** Time for sorting with weird sort = 1724.554ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 10.021ms
*** Sorting complete!

*** Time for sorting with intro sort = 9.120ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 14.345ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 262144 ]]]]]]]]]]

*** Time for sorting with insertion sort = 16284.822ms
*** Sorting complete!

*** Time for sorting with heap sort = 47.611ms
*** Sorting complete!

*** Time for sorting with weird sort = 6721.327ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 25.734ms
*** Sorting complete!

*** Time for sorting with intro sort = 35.275ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 26.836ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 524288 ]]]]]]]]]]

*** Time for sorting with insertion sort = 66411.855ms
*** Sorting complete!

*** Time for sorting with heap sort = 122.115ms
*** Sorting complete!

*** Time for sorting with weird sort = 28027.244ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 69.183ms
*** Sorting complete!

*** Time for sorting with intro sort = 68.919ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 78.471ms
*** Sorting complete!

[[[[[[[[[[ Input Size = 1048576 ]]]]]]]]]]

*** Time for sorting with insertion sort = 337497.969ms
*** Sorting complete!

*** Time for sorting with heap sort = 398.848ms
*** Sorting complete!

*** Time for sorting with weird sort = 123854.453ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 126.324ms
*** Sorting complete!

*** Time for sorting with intro sort = 102.984ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 148.166ms
*** Sorting complete!

C:\Users\mjddb\Desktop\HW3_S20231604\SortingMethods\x64\Release\Sort
이 창을 닫으려면 아무 키나 누르세요...
```

3. 최적화 방법

교과서적으로 구현한 heap sort와 quick sort에는 최적화 방법을 적용하지 않았다. Sort_records_weird는 min_heap을 구현하고 insertion을 하는 방식 이외에 더 최적화하지 않았다.

이외의 2개의 함수에 어떻게 최적화를 했는지 기술한다.

1) Sort_records_intro

① 먼저 wikipedia에 명시된 pseudo code대로 구현한 함수의 실행 결과를 3번 측정했다.

```
void RECORDS::real_intro(int start_index, int end_index, int maxdept
    // Introsort described in https://en.wikipedia.org/wiki/Introsor
    ll n = end_index - start_index + 1;
    if (n <= 16) {
        sort_records_insertion(start_index, end_index);
    }
    else if (maxdepth == 0) {
        sort_records_heap(start_index, end_index);
    }
    else {
        int pivot = partition_origin(start_index, end_index);
        real_intro(start_index, pivot - 1, maxdepth - 1);
        real_intro(pivot + 1, end_index, maxdepth - 1);
    }
}
```

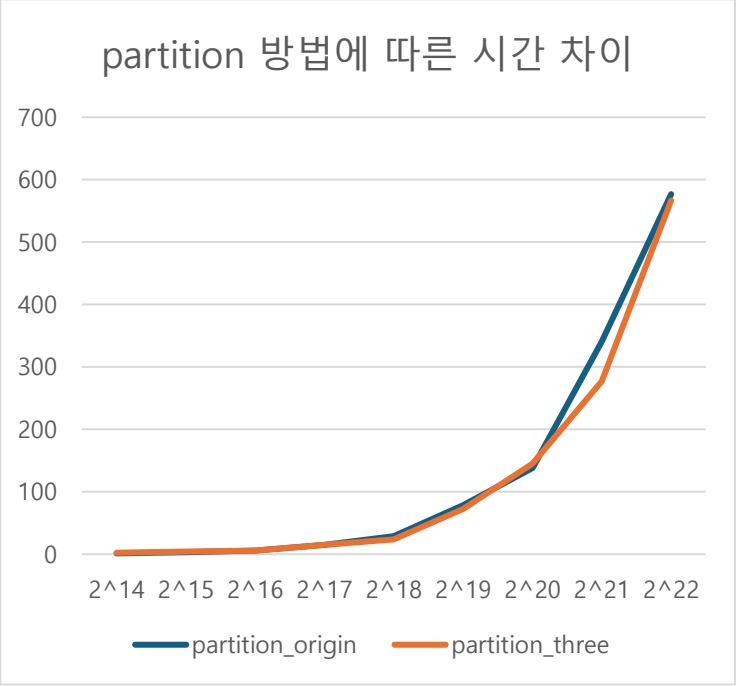
② 여기에 수업시간에 배운, pivot을 처음, 중간, 끝 인덱스 중 중앙값으로 정하는 방법(강의자료3 44p)을 이용해보았다. 3번 측정하였다.

```
void RECORDS::real_intro(int start_index, int end_index, int maxdep
    // Introsort described in https://en.wikipedia.org/wiki/Introso
    ll n = end_index - start_index + 1;
    if (n <= 16) {
        sort_records_insertion(start_index, end_index);
    }
    else if (maxdepth == 0) {
        sort_records_heap(start_index, end_index);
    }
    else {
        int pivot = partition_three(start_index, end_index);
        real_intro(start_index, pivot - 1, maxdepth - 1);
        real_intro(pivot + 1, end_index, maxdepth - 1);
    }
}
```

- Partition_origin과 partition_three의 실행 시간 평균 비교

inputdata	partition_origin			평균	partition_three			평균
2^14	1.24	1.591	1.288	1.373	2.113	3.603	1.549	2.421667
2^15	2.655	3.239	3.256	3.05	5.387	5.117	2.364	4.289333
2^16	5.248	5.939	6.736	5.974333	4.939	5.502	6.65	5.697
2^17	14.711	13.017	18	15.19567	12.396	12.943	19.153	14.83067
2^18	26.258	30.749	29.034	28.68033	23.436	23.238	23.943	23.539
2^19	61.734	69.909	104.378	78.67367	77.279	57.202	83.45	72.64367
2^20	143.354	143.699	127.974	138.3423	174.944	122.129	139.342	145.4717
2^21	372.835	281.923	367.318	340.692	242.953	320.841	267.062	276.952

inputdata	partition_origin	partition_three	**Three/origin**
2^14	1.373	2.421667	**1.763778**
2^15	3.05	4.289333	**1.406339**
2^16	5.974333	5.697	**0.953579**
2^17	15.19567	14.83067	**0.97598**
2^18	28.68033	23.539	**0.820737**
2^19	78.67367	72.64367	**0.923354**
2^20	138.3423	145.4717	**1.051534**
2^21	340.692	276.952	**0.81291**
2^22	576.856	566.606	**0.982231**



- 결론

입력 숫자가 2^14, 2^15일 때는 원본 partition함수가 확연히 빨랐지만, 그 이상의 크기에 대해서는 pivot을 중앙값으로 정한 함수의 실행시간이 2%~18%가량 빨랐다.

Input data가 2^14, 2^15일 때를 제외하고는 partition_three함수를 이용함이 효율적임을 확인했다.

③ Minimize the bookkeeping cost

앞서 partition_three함수가 효율적임을 확인했으므로, 이에 더하여 수업시간에 배운 재귀호출을 줄이는 코드를 적용해보았다. (강의자료3 46p)

```
void RECORDS::real_intro(int start_index, int end_index, int maxdepth) {
    // Introsort described in https://en.wikipedia.org/wiki/Introsort
    ll n = end_index - start_index + 1;
    if (n <= 16) {
        sort_records_insertion(start_index, end_index);
    }
    else if (maxdepth == 0) {
        sort_records_heap(start_index, end_index);
    }
    else {
        bookkeeping(start_index, end_index);
    }
}
```

bookkeeping함수의 quick sort에서 분할을 할 때 partition_three함수를 이용한다.

- bookkeeping함수

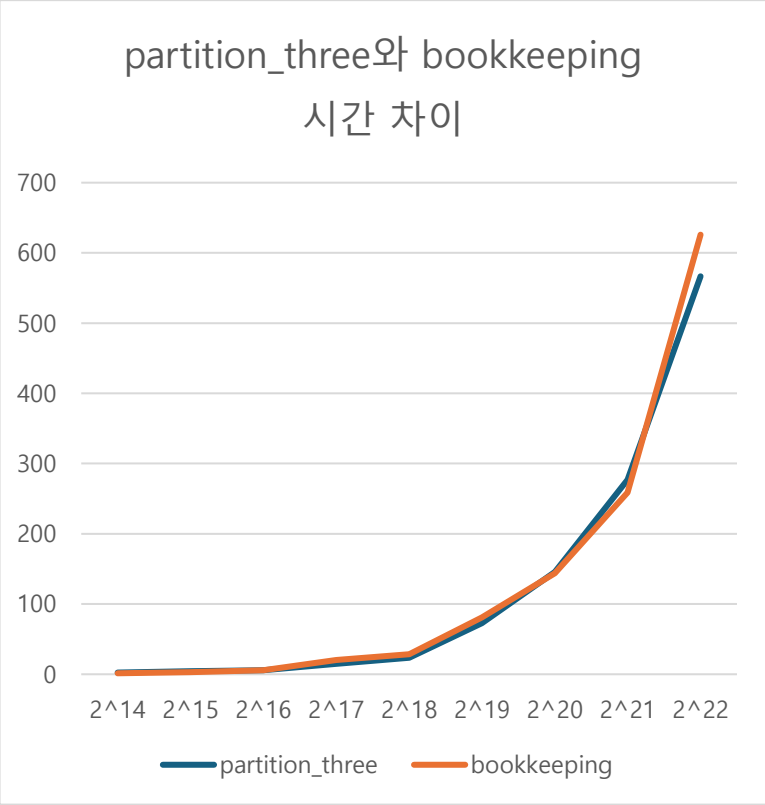
```
void RECORDS::bookkeeping(int first, int end) {
    int first1, first2, end1, end2;
    first2 = first; end2 = end;
    while (end2 - first2 > 0) {
        tmp = records[first];
        int pivot = partition_three(first2, end2);
        if (pivot < (first2 + end2) / 2) {
            first1 = first2; end1 = pivot - 1;
            first2 = pivot + 1; end2 = end2;
        }
        else {
            first1 = pivot + 1; end1 = end2;
            first2 = first2; end2 = pivot - 1;
        }
        bookkeeping(first1, end1);
    }
}
```

실행 예시

- Partition_three함수와 비교

3번 반복하여 측정한 실행시간의 평균을 비교하였다.

Input data	partition_three	bookkeeping	**Bookkeeping /three**
2^14	2.421667	1.384333	**0.571645**
2^15	4.289333	2.855333	**0.665682**
2^16	5.697	5.725333	**1.004973**
2^17	14.83067	20.52233	**1.383777**
2^18	23.539	28.67633	**1.218248**
2^19	72.64367	80.871	**1.113256**
2^20	145.4717	143.838	**0.98877**
2^21	276.952	258.8043	**0.934474**
2^22	566.606	625.9183	**1.10468**



- 결론

partition_three함수는 기존의 partition_origin을 적용한 intro_sort보다 2^14, 2^15에서 확연히 느렸던 단점이 있었다. bookkeeping함수는 이를 보완하여 2^14, 2^15에서도 빠르게 수행한다.

하지만 이후의 input에 대해서는 2~5%가량 빨라진 적도 있었던 반면 10~30%가량 더 느려진 적도 있다.

즉 Input data가 2^14, 2^15일 때에만 bookkeeping함수를 이용함이 효율적임을 확인했다.

- partition_three와 bookkeeping함수를 적용한 intro sort코드와 실행 예시

```
void RECORDS::real_intro(int start_index, int end_index, int maxdepth) {
    // Introsort described in https://en.wikipedia.org/wiki/Introsort
    ll n = end_index - start_index + 1;
    if (n <= 16) {
        sort_records_insertion(start_index, end_index);
    }
    else if (maxdepth == 0) {
        sort_records_heap(start_index, end_index);
    }
    else {
        if (end_index - start_index < 6e4)
            bookkeeping(start_index, end_index);
        else {
            int pivot = partition_three(start_index, end_index);
            real_intro(start_index, pivot - 1, maxdepth - 1);
            real_intro(pivot + 1, end_index, maxdepth - 1);
        }
    }
}
```

```
[[[[[[[[[ Input Size = 16384 ]]]]]]]]]

*** Time for sorting with intro sort = 1.991ms
*** Sorting complete!


[[[[[[[[[ Input Size = 32768 ]]]]]]]]]

*** Time for sorting with intro sort = 3.029ms
*** Sorting complete!


[[[[[[[[[ Input Size = 65536 ]]]]]]]]]

*** Time for sorting with intro sort = 5.338ms
*** Sorting complete!


[[[[[[[[[ Input Size = 131072 ]]]]]]]]]

*** Time for sorting with intro sort = 21.899ms
*** Sorting complete!


[[[[[[[[[ Input Size = 262144 ]]]]]]]]]

*** Time for sorting with intro sort = 35.439ms
*** Sorting complete!


[[[[[[[[[ Input Size = 524288 ]]]]]]]]]

*** Time for sorting with intro sort = 52.662ms
*** Sorting complete!


[[[[[[[[[ Input Size = 1048576 ]]]]]]]]]

*** Time for sorting with intro sort = 132.956ms
*** Sorting complete!


[[[[[[[[[ Input Size = 2097152 ]]]]]]]]]

*** Time for sorting with intro sort = 282.091ms
*** Sorting complete!


[[[[[[[[[ Input Size = 4194304 ]]]]]]]]]

*** Time for sorting with intro sort = 597.046ms
*** Sorting complete!

C:\Users\wjddb\Desktop\HW3_S20231604\SortingMethods\x64\Release
이 창을 닫으려면 아무 키나 누르세요...
```

④ N의 크기 설정

언제 insertion 함수를 적용하면 좋을지 실험해보았다. N=8일 때와 N=16일 때를 비교해보았다. Input data는 2^20, 2^21, 2^22로 실험했다.

- 실행결과 예시

```
[[[[[[[[[ Input Size = 1048576 ]]]]]]]]]

*** Time for sorting with intro sort(n=8) = 124.697ms
*** Sorting complete!

*** Time for sorting with intro sort(n=16) = 137.644ms
*** Sorting complete!


[[[[[[[[[ Input Size = 2097152 ]]]]]]]]]

*** Time for sorting with intro sort(n=8) = 228.380ms
*** Sorting complete!

*** Time for sorting with intro sort(n=16) = 303.360ms
*** Sorting complete!


[[[[[[[[[ Input Size = 4194304 ]]]]]]]]]

*** Time for sorting with intro sort(n=8) = 601.472ms
*** Sorting complete!

*** Time for sorting with intro sort(n=16) = 537.161ms
*** Sorting complete!

C:\Users\wjddb\Desktop\HW3_S20231604\SortingMethods\x64\Release\
다(코드: 0개).
```

- 3번 실행에서 평균 시간

	n=8	n=16	n=8	n=16	n=8	n=16	average	
2^20	112.203	141.358	108.793	187.08	124.697	137.644	115.231	155.3607
2^21	343.561	387.262	303.96	299.878	228.38	303.36	291.967	330.1667
2^22	659.94	668.475	678.073	689.455	601.472	537.161	646.495	631.697

	n=8	n=16	**N=8/n=16**
2^20	115.231	155.3607	**0.7417**
2^21	291.967	330.1667	**0.884302**
2^22	646.495	631.697	**1.023426**

- 결론

Input data가 2^22일 때는 n=8이 더 느릴 때도 있었지만 대체로 n=8이 n=16보다 20%가량 빠른 모습을 보여주었다. 즉 n=8일 때 insertion함수를 적용하기로 했다.

- **최종 intro sort 코드**

```
void RECORDS::real_intro(int start_index, int end_index, int maxdepth) {
    // Introsort described in https://en.wikipedia.org/wiki/Introsort
    ll n = end_index - start_index + 1;
    if (n <= 8) {
        sort_records_insertion(start_index, end_index);
    }
    else if (maxdepth == 0) {
        sort_records_heap(start_index, end_index);
    }
    else {
        if (end_index - start_index < 6e4)
            bookkeeping(start_index, end_index);
        else {
            int pivot = partition_three(start_index, end_index);
            real_intro(start_index, pivot - 1, maxdepth - 1);
            real_intro(pivot + 1, end_index, maxdepth - 1);
        }
    }
}
```

2) Sort_records_merge_with_insertion

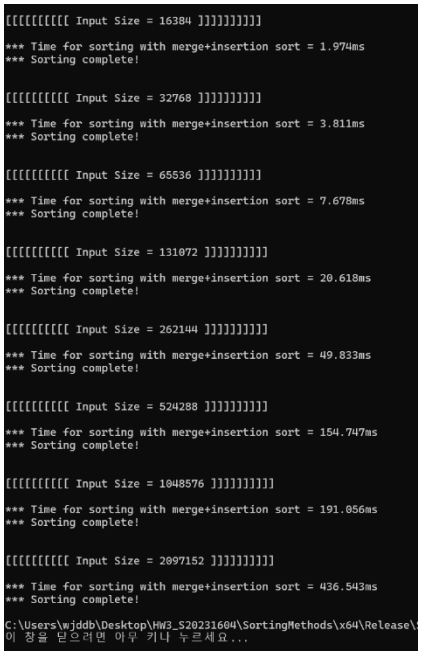
sort_records_merge_with_insertion 함수에서 언제 insertion 함수를 사용하면 좋을지 실험해보았다. N=0일 때, 5, 10, 16, 32일 때 insertion을 적용해보았다.

각 n의 크기마다 3번 실행하여 시간의 평균을 내었다.

- 실행 코드 예시

```
void RECORDS::sort_records_merge_with_insertion(int start_index, int end_index) {
    // Merge sort optimized by insertion sort only
    // Use insertion sort instead of combine
    if (end_index - start_index + 1 <= 32) {
        sort_records_insertion(start_index, end_index);
        return;
    }
    if (start_index < end_index) {
        int middle = (start_index + end_index) / 2;
        sort_records_merge_with_insertion(start_index, middle);
        sort_records_merge_with_insertion(middle + 1, end_index);
        merge(start_index, middle, end_index);
    }
}
```

- 실행 결과 예시



N=16일 때 insertion 1

N=32일 때 insertion 2

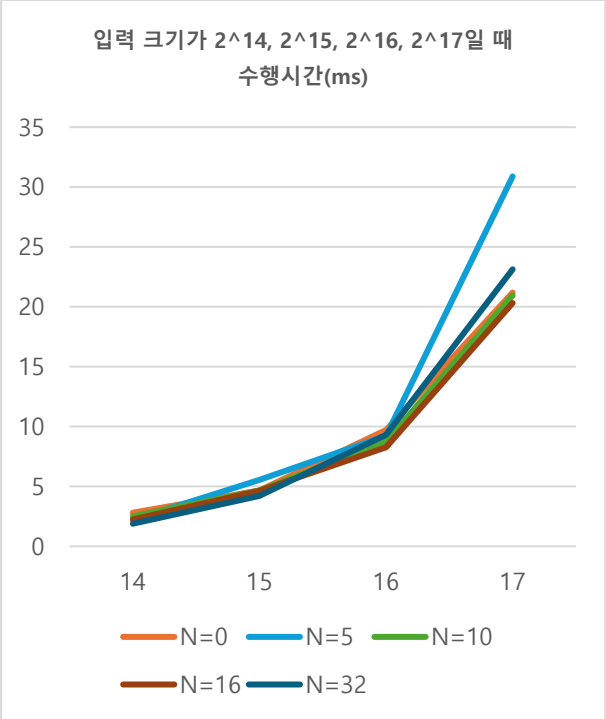
- N에 따른 정렬 시간 측정

input data	n=0			평균	n=5			평균	n=10			평균	n=16			평균	n=32			평균
2^14	2.843	2.215	3.378	2.812	2.014	2.083	2.538	2.211667	2.351	2.572	2.601	2.508	1.974	2.497	2.268	2.246333	2.136	1.745	1.804	1.895
2^15	4.48	4.672	4.804	4.652	4.275	5.36	7.039	5.558	3.753	5.465	4.737	4.651667	3.811	6.495	3.648	4.651333	4.687	3.636	4.299	4.207333
2^16	9.73	9.172	10.26	9.720667	10.488	8.321	8.888	9.232333	8.039	8.198	10.047	8.761333	7.678	8.713	8.403	8.264667	8.508	11.774	7.66	9.314
2^17	19.645	23.466	20.469	21.19333	23.196	26.016	43.449	30.887	25.324	19.345	18.182	20.95033	20.618	21.711	18.63	20.31967	23.683	23.772	21.938	23.131
2^18	53.973	44.239	63.006	53.73933	60.27	47.149	52.775	53.398	44.56	37.743	40.165	40.82267	49.833	47.991	43.923	47.249	42.421	42.747	57.787	47.65167
2^19	103.146	111.276	98.01	104.144	100.524	96.152	144.077	113.5843	95.612	86.8	92.065	91.49233	154.747	94.303	99.31	116.12	82.951	80.529	80.274	81.25133
2^20	204.485	227.782	233.055	221.774	189.603	222.435	225.335	212.4577	225.178	233.667	203.834	220.893	191.056	204.089	196.343	197.1627	242.195	180.438	210.273	210.9687
2^21	419.058	461.777	437.111	439.3153	420.258	468.999	421.477	436.9113	444.546	423.159	433.466	433.7237	436.543	408.522	420.823	421.9627	367.304	379.68	387.291	378.0917

- N의 크기에 따른 정렬 시간의 평균값

inputdata	N=0	N=5	N=10	N=16	N=32	가장 빠른 N	두번째로 빠른 N
2^14	2.812	2.211667	2.508	2.246333	1.895	N=32	N=5
2^15	4.652	5.558	4.651667	4.651333	4.207333	N=32	N=16
2^16	9.720667	9.232333	8.761333	8.264667	9.314	N=16	N=10
2^17	21.19333	30.887	20.95033	20.31967	23.131	N=16	N=10
2^18	53.73933	53.398	40.82267	47.249	47.65167	N=10	N=16
2^19	104.144	113.5843	91.49233	116.12	81.25133	N=32	N=10
2^20	221.774	212.4577	220.893	197.1627	210.9687	N=16	N=32
2^21	439.3153	436.9113	433.7237	421.9627	378.0917	N=32	N=16

- 입력 크기가 2^14, 2^15, 2^16, 2^17일 때



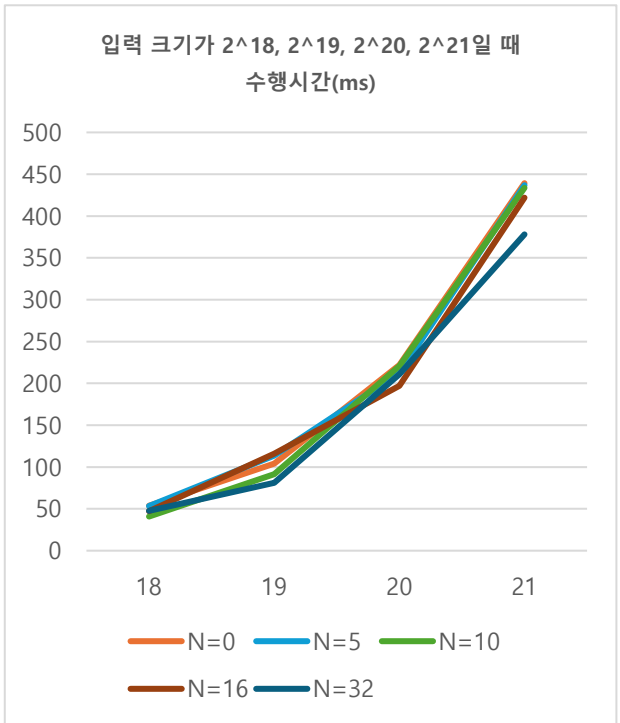
입력 크기	가장 빠른 N	두번째로 빠른 N
2^14	N=32	N=5
2^15	N=32	N=16
2^16	N=16	N=10
2^17	N=16	N=10

n=5일 때 대체로 시간이 가장 오래 걸렸고, n=16일 때 가장 적게 걸렸다.

그 다음으로는 n=10일 때가 빨랐다.

입력크기 2^15에서 2^17사이에서는 n=16일 때가 가장 빠르다.

- 입력 크기가 2^14, 2^15, 2^16, 2^17일 때



입력 크기	가장 빠른 N	두번째로 빠른 N
2^18	N=10	N=16
2^19	N=32	N=10
2^20	N=16	N=32
2^21	N=32	N=16

N=0일 때 대체로 가장 오래걸렸고, n=32일 때 가장 적게 걸렸다. 그 다음으로는 n=16일 때가 빨랐다.

입력크기 2^19에서 2^21사이에서는 n=32일 때가 가장 빠르다.

- **결론**

N=0일 때 대체로 큰 시간이 소요됐고, n=16, 32일 때 대체로 빠른 정렬을 보여주었다. 입력 크기를 모를 때(정해지지 않았을 때) N의 크기를 정한다면 **n=16**으로 정하는 것이 가장 합리적일 것이라 결론지었다.

4. 모든 정렬 함수에 대한 실행 시간 비교

1) 실행 속도 비교

6개의 함수에 대해 입력 크기 2^14부터 2^20까지 총 5회 실행하여 얻은 실행 시간의 평균을 구했다.

	insertionsort					평균
2^14	54.767	82.384	68.8991	98.4861	48.6022	70.62768
2^15	223.291	273.441	298.544	335.702	198.327	265.861
2^16	886.281	1087.166	1322.03	1201.44	926.307	1084.645
2^17	3483.285	4225.899	4809.35	4516.35	4206.99	4248.375
2^18	16284.82	17270.68	17454.9	24812.2	17321.5	18628.82
2^19	66411.86	71590.36	70601.9	122732	71944.8	80656.18
2^20	337498	337688.9	329818	555472	385689	389233.2

	heapsort					평균
2^14	1.154	1.286	1.4574	1.3303	0.9673	1.239
2^15	2.313	3.316	3.5747	2.5512	1.9055	2.73208
2^16	8.487	5.631	6.8976	6.4777	6.4289	6.78444
2^17	18.932	13.989	15.7779	15.7452	12.1822	15.32526
2^18	47.611	47.883	49.6127	82.1613	45.3883	54.53126
2^19	122.115	110.604	92.778	192.97	162.734	136.2402
2^20	398.848	369.699	367.702	462.276	520.879	423.8808

	weird sort					평균
2^14	26.168	35.841	26.4619	29.9088	22.8786	28.25166
2^15	86.571	152.745	146.146	159.953	84.1386	125.9107
2^16	351.992	477.325	464.195	562.677	391.629	449.5636
2^17	1724.554	1925.515	1883.96	2085.99	1805	1885.004
2^18	6721.327	7637.349	7409.97	13882.2	6823.81	8494.931
2^19	28827.24	30851.43	29826.1	51819.9	28898.8	34044.69
2^20	123854.5	159210.4	147833	222939	138760	158519.4

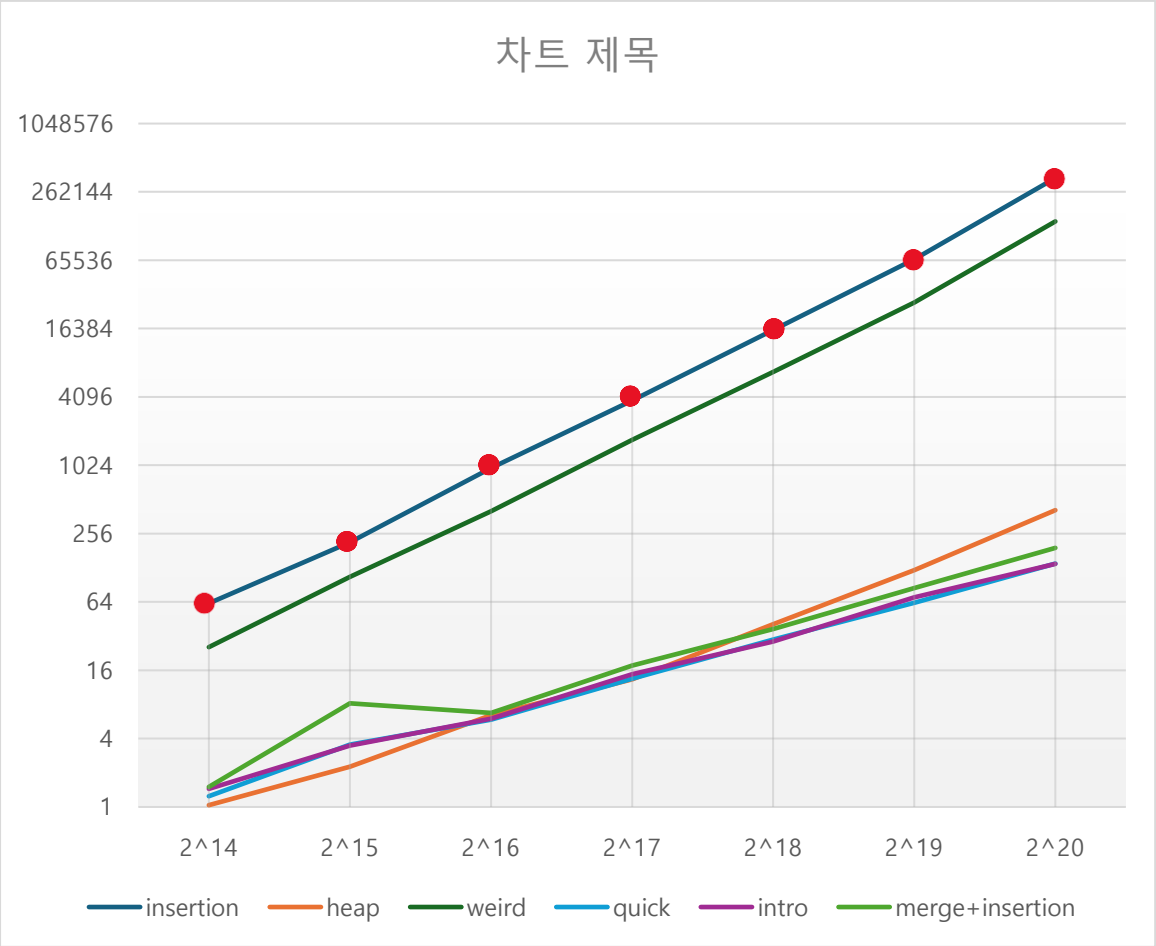
	Class quick sort					평균
2^14	1.038	1.486	1.5499	1.4613	1.4442	1.39588
2^15	4.472	3.188	3.3991	5.9389	2.6581	3.93122
2^16	4.654	6.021	7.7514	7.8841	5.3401	6.33012
2^17	18.821	12.668	15.1637	20.4705	13.9242	16.20948
2^18	25.734	31.066	33.9041	52.8029	47.355	38.1724
2^19	69.183	58.93	74.7739	114.028	55.0758	74.39814
2^20	126.324	103.047	250.306	232.642	160.419	174.5476

	Intro sort					평균
2^14	0.932	1.185	1.9724	1.8937	1.3323	1.46308
2^15	4.357	2.722	4.2769	5.8814	4.9158	4.43062
2^16	4.929	6.829	7.2584	10.1633	6.1913	7.0742
2^17	9.128	14.326	17.0918	21.6355	23.7859	17.19344
2^18	35.275	27.038	34.2893	60.4924	28.995	37.21794
2^19	68.919	54.528	82.8311	156.914	65.7304	85.7845
2^20	102.984	129.651	276.148	259.483	130.03	179.6592

	Merge+insertion sort					평균
2^14	1.588	1.746	2.4734	1.9813	1.4063	1.839
2^15	5.442	23.843	4.2121	4.7298	4.4059	8.52656
2^16	6.602	8.878	7.8871	10.0085	6.0682	7.88876
2^17	14.345	17.712	19.0487	17.093	17.0189	17.04352
2^18	26.836	37.308	50.5185	58.5144	63.6234	47.36006
2^19	78.471	74.003	79.6392	141.307	113.939	97.47184
2^20	148.166	151.167	298.173	278.269	242.538	223.6626

- 평균 시간 비교

	insertion	heap	weird	quick	intro	merge+insertion
2^14	62.24118	1.03732	25.5862	1.24382	1.44404	1.50246
2^15	214.6106	2.26384	106.5968	3.53178	3.47532	8.19004
2^16	970.0878	6.42654	404.0072	5.8734	6.00818	6.75664
2^17	3804.895	13.2986	1713.626	13.49164	14.79476	17.56404
2^18	15890.2	40.837	6775.145	29.89964	28.66062	36.85914
2^19	66722.82	122.3676	27857.59	62.9953	70.33718	84.67204
2^20	347345	411.7834	144057.6	139.3598	138.5184	191.6112



Insertion함수가 가장 오래 걸렸고, 그 다음으로 weird함수가 오래 걸렸다.

특히 두 함수 모두 y축이 log스케일일 때 함수가 선형적으로 증가하는 모습을 보였다. Inputdata가 2배 증가할 때 실행시간이 4배 증가한 것으로 보아 O(n^2)이 성립함을 알 수 있었다.

Input 크기가 작을 때는 merge+insertion 함수가 오래 걸리고, heap이 가장 빨랐지만, 크기가 커질수록 heap은 느려지고 intro sort가 빨라졌다.

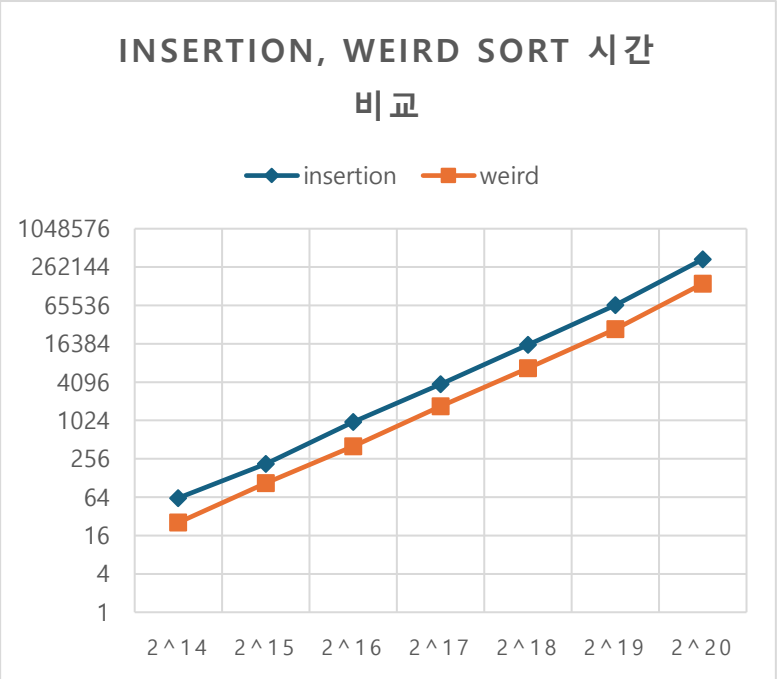
2) 시간 복잡도 확인

입력크기가 2배 증가할 때 실행시간이 몇 배 증가하는지 확인했다.

① Insertion/weird sort

앞서 확인했듯이 Inputdata가 2배 증가할 때 실행시간이 4배 증가하는 것을 알 수 있었다.

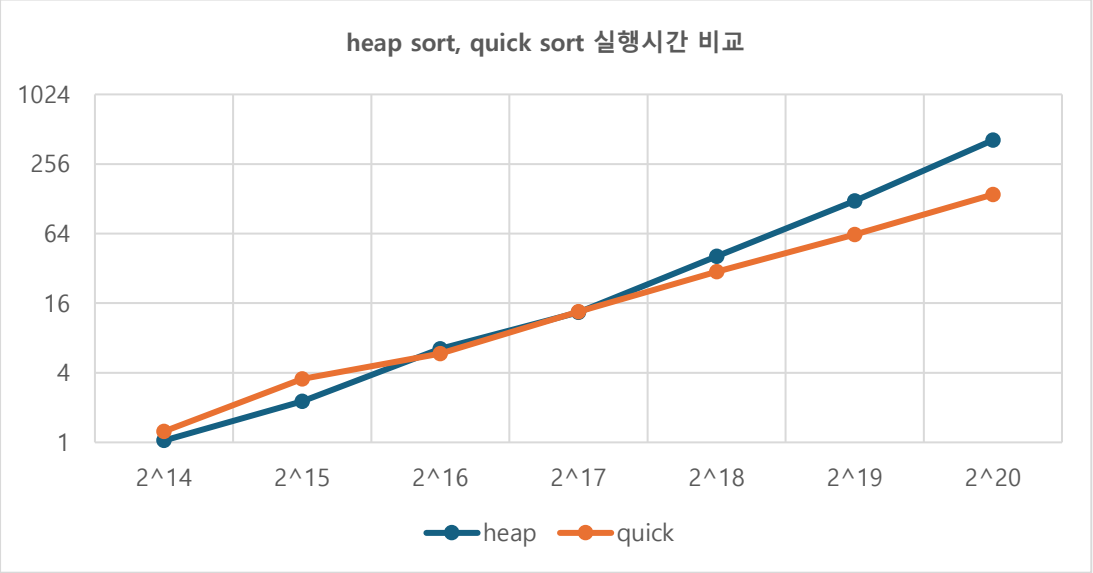
Input data	insertion	weird	Insertion /weird
2^14	62.24118	25.5862	2.432607
2^15	214.6106	106.5968	2.013293
2^16	970.0878	404.0072	2.401165
2^17	3804.895	1713.626	2.220377
2^18	15890.2	6775.145	2.345367
2^19	66722.82	27857.59	2.395139
2^20	347345	144057.6	2.411154



이를 통해 이론적 시간복잡도 **O(n^2)**과 실제 실행 시간이 일치함을 알 수 있었다.

weird sort는 insertion보다 2배 이상 빠른 모습을 보였지만 대체로 증가폭이 n^2을 따랐다. 이를 통해 먼저 min heap을 만든 다음 insertion sort를 이용하면 시간복잡도는 O(n^2)으로 일치하지만 실제 수행 속도는 2.3배가량 빠름을 확인했다.

② Heap/quick sort



	2^15/2^14	2^16/2^15	2^17/2^16	2^18/2^17	2^19/2^18	2^20/2^19	평균 증가량
heapsort	2.182393	2.838778	2.069325	3.070774	2.996488	3.365134	2.753816

	2^15/2^14	2^16/2^15	2^17/2^16	2^18/2^17	2^19/2^18	2^20/2^19	평균 증가량
quicksort	2.839462	1.663014	2.297075	2.216161	2.106892	2.212225	2.222471

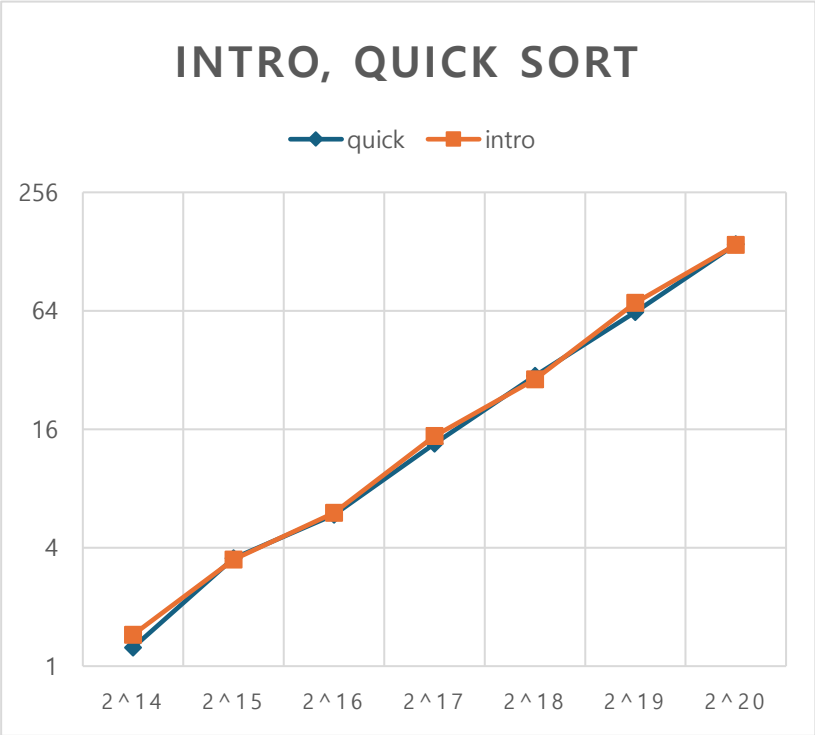
이론적 시간 복잡도 O(nlogn)에서, 입력 크기가 2배로 커졌을 때 실행 시간은 (2n)log(2n)으로 대략 **2.6배** 커지는 것이 이론적이다.

Heap sort는 평균 증가량 **2.75**로 대체로 이를 따랐고, quicksort는 평균증가량 **2.22**로 이보다 빨랐지만 전체적으로 일정한 시간 증가율을 보였다. 다만 입력 크기가 작은 2^14, 2^15 에서는 실행속도가 비교적 컸다.

quick sort의 평균 증가량인 2.22보다 약 20% ($\frac{2.75-2.22}{2.22} \times 100(\%)$) 느린 것을 알 수 있었다.

③ Intro/quick

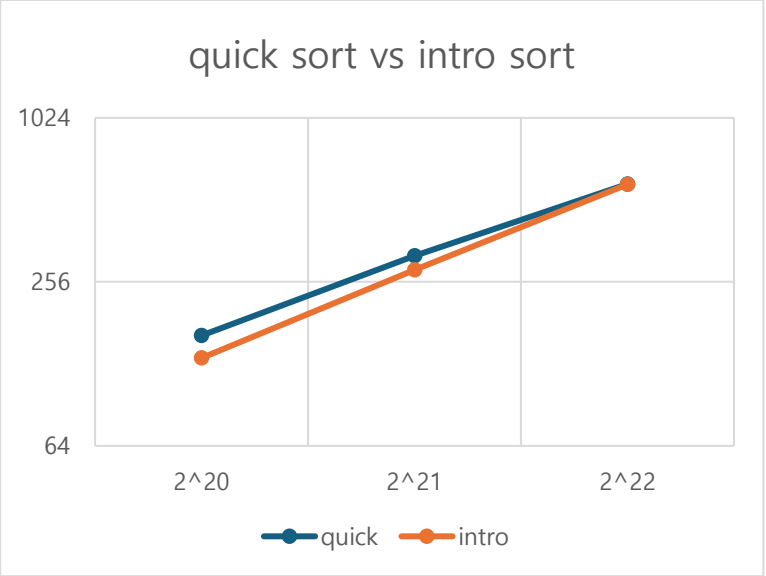
	quick	intro	**Intro/quick**
2^14	1.24382	1.44404	**1.160972**
2^15	3.53178	3.47532	**0.984014**
2^16	5.8734	6.00818	**1.022948**
2^17	13.49164	14.79476	**1.096587**
2^18	29.89964	28.66062	**0.958561**
2^19	62.9953	70.33718	**1.116546**
2^20	139.3598	138.5184	**0.993962**



Intro sort는 quick sort를 최적화한 알고리즘이다. 따라서 대체로 둘의 실행시간이 대체로 일치함을 그래프 모양을 통해 알 수 있었다. 하지만 intro sort가 과연 quick sort보다 빠른지는 명확하지 않아 input data를 키워서 다시 실험해보았다.

2^20, 2^21, 2^22에 대해 5번 실행하여 얻은 평균값으로 비교했다.

Input data	quick	intro	**Intro/quick**
2^20	162.5898	134.4618	**0.827**
2^21	319.4552	283.3322	**0.886923**
2^22	585.2148	583.6004	**0.997241**



Input data의 값이 커졌을 때 intro sort가 quick sort보다 명확히 빠름을 알 수 있었다. Intro sort가 quick sort보다 10퍼센트가량 빨랐다.

- 평균 증가량

2^14부터 2^20까지 입력 데이터가 2배 커졌을 때 실행시간이 몇 배 커지는지 실험했다.

	2^15/2^14	2^16/2^15	2^17/2^16	2^18/2^17	2^19/2^18	2^20/2^19	평균 증가량
Intro sort	2.406665	1.728813	2.462436	1.937214	2.45414	1.969348	2.159769

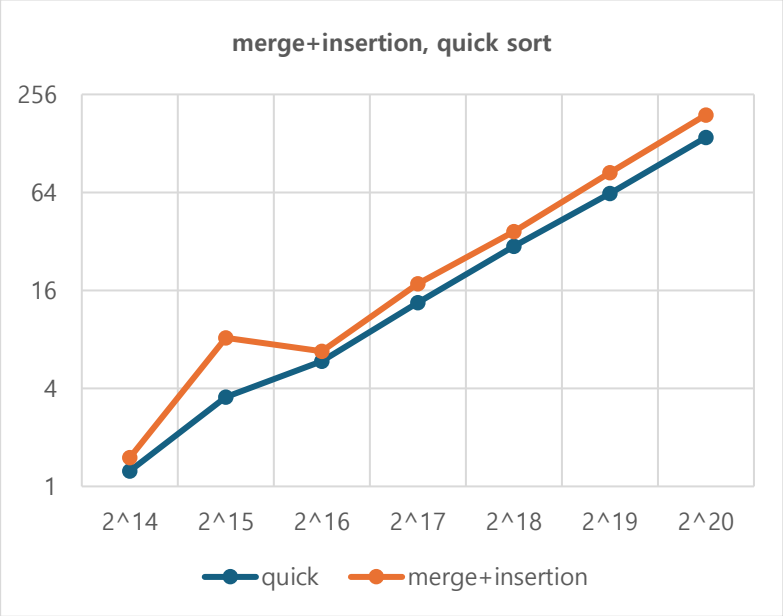
여러 sorting 기법을 결합한 함수인 만큼 시간 증가량이 다른 함수와 비교했을 때 일정하진 않았지만, 대체로 2배 내외로 증가함을 알 수 있었다.

특히 quick sort의 평균 증가량인 2.22보다 약 3% ($\frac{2.22-2.15}{2.22} \times 100(\%)$) 빠른 것을

보았을 때 intro sort가 효과적이었음을 알 수 있었다.

④ Merge+insertion / quick sort

	quick	Merge +insertion	**merge/quick**
2^14	1.24382	1.50246	**1.20794**
2^15	3.53178	8.19004	**2.318955**
2^16	5.8734	6.75664	**1.15038**
2^17	13.49164	17.56404	**1.301846**
2^18	29.89964	36.85914	**1.232762**
2^19	62.9953	84.67204	**1.344101**
2^20	139.3598	191.6112	**1.374939**

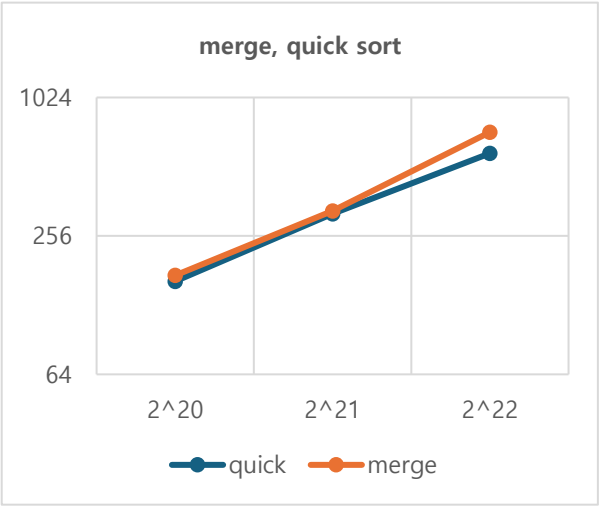


Merge+insertion의 증가 양상이 대체로 quick sort의 증가 그래프와 비슷했다. 2^15에서 약간 실행 시간이 더 커졌던 quick sort와 달리 merge sort는 급격히 증가하는 모습을 보였지만, 이외에는 대략 quick sort보다 20~30% 느린 양상이 일정하게 유지됐다.

Merge sort의 세 단계 divide, conquer, combine중에서 combine 과정을 제외하여 발전시킨 알고리즘이 quick sort인 만큼 quick sort가 확연히 빠른 모습을 보여주었다.

더 큰 input인 2^20, 2^21, 2^22에 대해 5번 실행하여 얻은 평균값도 비교했다.

	quick	merge	**Merge/quick**
2^20	162.5898	172.4162	**1.060437**
2^21	319.4552	328.6302	**1.028721**
2^22	585.2148	723.309	**1.235972**



2^20, 2^21에서는 quick과 비슷했지만, 2^22에서 다시 quick보다 20%가량 느린 양상이 유지됐다.

- 평균증가량

2^14부터 2^20까지 실행시간의 평균을 바탕으로 입력 크기가 2배 증가할 때 실행시간이 몇 배 증가하는지 계산해보았다.

	2^15/2^14	2^16/2^15	2^17/2^16	2^18/2^17	2^19/2^18	2^20/2^19	평균 증가량
Merge+ insertion	5.451087	0.824983	2.599523	2.098557	2.297179	2.262981	2.589052

2^15에서 실행시간이 급격히 커진 탓에 1,2열의 증가량이 비정상적이지만, 이외에는 평균적으로 2배보다 조금 더 증가했다.

quick sort의 평균 증가량인 2.22보다 약 15% ($\frac{2.58-2.22}{2.22} \times 100(\%)$) 느린 것을 보았을 때 quick sort가 더 빠름을 알 수 있었다.

이를 통해 merge_with_insertion sort는 O(n)에서 O(nlogn)의 사이의 시간복잡도를 가짐을 확인했다. Merge sort의 이론적 시간복잡도가 O(nlogn)이므로 insertion을 추가한 최적화 방법이 효과적이었음을 알 수 있었다.

5. 결과 논의

이번 실험을 통해 실제 수행시간과 시간복잡도가 비례함을 알 수 있었다.

특히 insertion함수는 O(n^2)을 거의 정확히 따랐고, min heap을 만든 후 insertion을 적용한다면 O(n^2)은 그대로이지만 실행시간은 2.3배가량 증가한다.

Heap 정렬은 O(nlogn)을 따랐지만 이론적으로 같은 시간 복잡도를 갖는 quick sort에 비해서는 20%가량 느렸다.

마찬가지로 merge+insertion 정렬도 O(nlogn)을 따랐지만 이론적으로 같은 시간 복잡도를 갖는 quick sort에 비해서는 15%가량 느렸다.

최적화를 진행한 intro sort는 quick sort보다 빨랐고, 이는 input data의 크기가 커질수록 두드러졌다. 평균적으로 intro sort가 3%정도 빨랐다.

1. Intro sort
2. Class Quick sort
3. Merge+insertion sort
4. Heap sort
5. Weird sort
6. Insertion sort

순서로 속도가 빠름을 알 수 있었다.