

# HW3 보고서 숙제

전공: 컴퓨터공학과 학년: 4학년

학번: 20212021

이름: 원대호

## 1. 컴퓨터 실험 환경

[1] OS 에디션 : Windows 11 Home

[2] 버전 : 23H2

[3] OS 빌드 : 22631.4169

[4] CPU : 12th Gen Intel(R) Core(TM) i7-1260P

[5] RAM : 16.0GB

[6] Compiler: Visual Studio 2022 Release Mode / x64 Platform

```
C:\Users\sed26\source\repos\SortingMethods\x64\Release\SortingMethods.exe(프
로세스 435236개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

```
[[[[[[[[[ Input Size = 16384 ]]]]]]]]
*** Time for sorting with insertion sort = 53.414ms
*** Sorting complete!

*** Time for sorting with heap sort = 1.380ms
*** Sorting complete!

*** Time for sorting with weird sort = 16.336ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 0.973ms
*** Sorting complete!

*** Time for sorting with intro sort = 0.829ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 1.411ms
*** Sorting complete!
```

```
[[[[[[[[[ Input Size = 32768 ]]]]]]]]
*** Time for sorting with insertion sort = 160.104ms
*** Sorting complete!

*** Time for sorting with heap sort = 2.636ms
*** Sorting complete!

*** Time for sorting with weird sort = 77.324ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 2.646ms
*** Sorting complete!

*** Time for sorting with intro sort = 1.725ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 3.013ms
*** Sorting complete!
```

```
[[[[[[[[[ Input Size = 65536 ]]]]]]]]]]  
*** Time for sorting with insertion sort = 621.554ms  
*** Sorting complete!  
*** Time for sorting with heap sort = 5.891ms  
*** Sorting complete!  
*** Time for sorting with weird sort = 271.381ms  
*** Sorting complete!  
*** Time for sorting with classic quick sort = 4.033ms  
*** Sorting complete!  
*** Time for sorting with intro sort = 3.695ms  
*** Sorting complete!  
*** Time for sorting with merge+insertion sort = 7.399ms  
*** Sorting complete!
```

```
[[[[[[[[[ Input Size = 131072 ]]]]]]]]]]  
*** Time for sorting with insertion sort = 2695.401ms  
*** Sorting complete!  
*** Time for sorting with heap sort = 12.587ms  
*** Sorting complete!  
*** Time for sorting with weird sort = 1149.575ms  
*** Sorting complete!  
*** Time for sorting with classic quick sort = 11.217ms  
*** Sorting complete!  
*** Time for sorting with intro sort = 8.955ms  
*** Sorting complete!  
*** Time for sorting with merge+insertion sort = 18.702ms  
*** Sorting complete!
```

```
[[[[[[[[[ Input Size = 262144 ]]]]]]]]]]  
*** Time for sorting with insertion sort = 17092.322ms  
*** Sorting complete!  
*** Time for sorting with heap sort = 67.300ms  
*** Sorting complete!  
*** Time for sorting with weird sort = 11301.644ms  
*** Sorting complete!  
*** Time for sorting with classic quick sort = 38.266ms  
*** Sorting complete!  
*** Time for sorting with intro sort = 29.683ms  
*** Sorting complete!  
*** Time for sorting with merge+insertion sort = 67.764ms  
*** Sorting complete!
```

```
[[[[[[[[[ Input Size = 524288 ]]]]]]]]]]  
*** Time for sorting with insertion sort = 214853.859ms  
*** Sorting complete!  
*** Time for sorting with heap sort = 178.723ms  
*** Sorting complete!  
*** Time for sorting with weird sort = 55313.000ms  
*** Sorting complete!  
*** Time for sorting with classic quick sort = 115.000ms  
*** Sorting complete!  
*** Time for sorting with intro sort = 111.533ms  
*** Sorting complete!  
*** Time for sorting with merge+insertion sort = 184.900ms  
*** Sorting complete!
```

```
[[[[[[[[[ Input Size = 1048576 ]]]]]]]]]]  
*** Time for sorting with insertion sort = 1204692.625ms  
*** Sorting complete!  
  
*** Time for sorting with heap sort = 906.669ms  
*** Sorting complete!  
  
*** Time for sorting with weird sort = 285373.688ms  
*** Sorting complete!  
  
*** Time for sorting with classic quick sort = 851.994ms  
*** Sorting complete!  
  
*** Time for sorting with intro sort = 946.627ms  
*** Sorting complete!  
  
*** Time for sorting with merge+insertion sort = 361.865ms  
*** Sorting complete!
```

## 2. 실험 방법

[1] 수업시간에 배운 몇 가지 정렬 알고리즘의 구현을 통하여 정렬기법에 대한 이해도를 높이도록 한다.

[2] 정렬 알고리즘을 구현할 때 수행성을 제고할 수 있는 최적화기법을 하나씩 적용해가면서 그 효과를 측정하여 봄으로써, 최적의 소프트웨어 구현기술에 대한 이해도를 높인다.

[3] 실험의 정확성을 높이기 위해 3번의 실험을 진행해 평균을 계산한다.

[4] 3번의 실험 후 시간측정결과가 각 정렬방법의 이론적인 time complexity와 일치하는지 비교 분석한다.

### 3. 구현 방법 ( 6가지 )

```
void sort_records_insertion(int start_index, int end_index);
```

- 교과서적인 insertion sort 방법(이미 구현되어 있음)

```
void sort_records_heap(int start_index, int end_index);
```

- 교과서적인 heap sort 방법

```
void sort_records_weird(int start_index, int end_index);
```

- 먼저 min heap을 만든 후 insertion sort를 적용하는 방법

```
void sort_records_quick_classic(int start_index, int end_index);
```

- 최적화 방법을 적용하지 않은 교과서적인 quick sort 방법

```
void sort_records_intro(int start_index, int end_index);
```

- <https://en.wikipedia.org/wiki/Introsort>에 기술한 quick sort의 변형 방법

```
void sort_records_merge_with_insertion(int start_index, int end_index);
```

- insertion sort를 사용하여 merge sort의 속도를 향상시키는 방법

#### [1] 교과서적인 insertionsort 방법

```
void RECORDS::sort_records_insertion(int start_index, int end_index) {
    // 교과서적인 insertionsort 방법
    for (int i = start_index + 1; i <= end_index; i++) {
        // records[0]은 고정한다.
        RECORD tmp = records[i];

        int j = i;
        while ((j > start_index) && (compare_keys((const void*)&tmp, (const void*)&records[j - 1]) < 0)) {

            records[j] = records[j - 1];
            //왼쪽이 오른쪽보다 크면 자리를 바꾼다.
            j--;
        }
        records[j] = tmp;
        //insertion 대상 자리 세팅
    }
}
```

이 함수는 삽입 정렬을 사용하여 records 배열의 start\_index부터 end\_index까지 오름 차순으로 정렬합니다. 처음에 records[0]를 고정하고 while문을 반복합니다. 반복 과정에서 왼쪽 요소가 오른쪽 요소보다 크면 자리를 교환합니다. 이론적인 시간복잡도는  $O(n^2)$ 입니다.

## [2] 교과서적인 heapsort 방법

```
void RECORDS::adjust_max_heap(int internal_node, int end_index) {
    // adjust the binary tree to establish the max heap
    int child;

    RECORD temp = records[internal_node];

    child = 2 * internal_node + 1;
    //왼쪽 자식의 인덱스 설정

    while (child <= end_index) {
        if (child < end_index && compare_keys((const void*)(&records[child]), (const void*)(&records[child + 1])) < 0) {
            child++;
        }
        // 오른쪽 자식이 왼쪽 자식보다 크면 child를 오른쪽 자식으로 setting

        if (compare_keys((const void*)(&temp), (const void*)(&records[child])) >= 0) {
            break;
            // 부모노드가 자식노드보다 크거나 같으면 break!
        }
        else {
            records[(child - 1) / 2] = records[child];
            // 자식을 부모노드로 이동시킨다.
            child = 2 * child + 1;
        }
    }
    // 최종 위치에 temp 값 저장
    records[(child - 1) / 2] = temp;
}
```

```
void RECORDS::sort_records_heap(int start_index, int end_index) {
    // 교과서적인 heapsort방법

    for (int i = (start_index + end_index) / 2; i >= start_index; i--) {
        adjust_max_heap(i, end_index);
    }
    // 자식노드가 있는 internal 노드를 검사하면서 max_heap을 구축한다.

    for (int i = end_index; i > start_index; i--) {

        SWAP(records[start_index], records[i]);
        // 루트노드와 제일 끝 노드를 바꾼다.
        adjust_max_heap(start_index, i - 1);
        // max_heap 과정을 반복한다.
    }
}
```

이 함수는 교과서적인 힙 정렬을 사용해 records 배열의 start\_index부터 end\_index까지 오름차순으로 정렬합니다. 먼저, 자식 노드가 있는 internal node를 검사하면서 max-heap을 구축합니다. 이때 adjust함수를 통해 자식노드가 부모노드보다 크면 자리를 교환하는 과정을 반복합니다. 이후 root노드와 배열의 마지막 노드를 바꾸고, 다시 max-heap으로 조정하는 과정을 반복하며 오름차순으로 정렬합니다. 시간 복잡도는 O(nlogn)입니다.

### [3] 먼저 minheap을 만든 후 insertionsort를 적용하는 방법

```
void RECORDS::adjust_min_heap(int internal_node, int end_index) {
    int child;
    RECORD temp = records[internal_node];

    child = 2 * internal_node + 1;

    while (child <= end_index) {
        if (child < end_index && compare_keys(&records[child], &records[child + 1]) > 0) {
            child++;
        }

        if (compare_keys(&temp, &records[child]) <= 0) {
            break;
        }
        else {
            records[internal_node] = records[child];
            internal_node = child;
            child = 2 * internal_node + 1;
        }
    }
    records[internal_node] = temp;
}
```

```
void RECORDS::sort_records_weird(int start_index, int end_index) {
    // A weird sort with a make-heap operation followed by insertion sort

    // min_heap을 만든다.
    for (int i = (start_index + end_index) / 2; i >= start_index; i--) {
        adjust_min_heap(i, end_index);
    }

    sort_records_insertion(start_index, end_index);
}
```

이 함수는 min heap 정렬을 만든 후 insertion 정렬을 적용해 records 배열의 start\_index부터 end\_index까지 오름차순으로 정렬합니다. 기본 원리에 의하면 어느 정도 정렬이 이루어진 상태에서 insertion sort를 적용하면 좀 더 빠르게 오름차순으로 정렬 할 수 있다. 이론적으로 min-heap 구성 후에 삽입 정렬을 수행하는 전체 알고리즘의 시간 복잡도는 거의 정렬된 data 상태에 대해  $O(n+d)$ 로 근사할 수 있습니다.

#### [4] 최적화방법을 적용하지 않은 교과서적인 quicksort 방법

```
int RECORDS::partition(int left, int right) {
    int pivot = left;
    for (int i = left; i < right; i++) {
        if (compare_keys(&records[i], &records[right]) < 0) {
            SWAP(records[i], records[pivot]);
            pivot++;
        }
    }
    SWAP(records[pivot], records[right]);
    return pivot;
}

void RECORDS::sort_records_quick_classic(int start_index, int end_index) {
    // Classic quick sort without any optimization techniques applied
    if (end_index - start_index > 0) {
        int pivot = partition(start_index, end_index);
        sort_records_quick_classic(start_index, pivot - 1);
        sort_records_quick_classic(pivot + 1, end_index);
    }
}
```

이 함수는 최적화방법을 적용하지 않은 교과서적인 quicksort를 적용해 records 배열의 start\_index부터 end\_index까지 오름차순으로 정렬합니다. 이때 pivot은 right로 두고, pivot보다 작으면 왼쪽에 두고 pivot보다 크면 오른쪽에 두는 방식을 반복적으로 이행한다. 시간복잡도는 평균적으로  $O(n\log n)$ 이지만 최악의 시간복잡도는  $O(n^2)$ 입니다.

## [5] <https://en.wikipedia.org/wiki/Introsort>에 기술한 quicksort의 변형 방법

```
int RECORDS::introsort_partition(int left, int right) {
    int mid = left + (right - left) / 2;

    if (compare_keys(&records[left], &records[mid]) > 0){
        SWAP(records[left], records[mid]);
    }

    if (compare_keys(&records[left], &records[right]) > 0) {
        SWAP(records[left], records[right]);
    }

    if (compare_keys(&records[mid], &records[right]) > 0) {
        SWAP(records[mid], records[right]);
    }

    SWAP(records[mid], records[right]);

    int pivot = left;
    for (int i = left; i < right; i++) {
        if (compare_keys(&records[i], &records[right]) < 0) {
            SWAP(records[i], records[pivot]);
            pivot++;
        }
    }

    SWAP(records[pivot], records[right]);
    return pivot;
}
```

```
void RECORDS::introsort(int start_index, int end_index, int max_depth) {
    int len = end_index - start_index + 1;
    if (len < 16) {
        sort_records_insertion(start_index, end_index);
        return;
    }
    else if (max_depth == 0) {
        sort_records_heap(start_index, end_index);
        return;
    }
    else {
        int pivot = introsort_partition(start_index, end_index);
        introsort(start_index, pivot - 1, max_depth - 1);
        introsort(pivot + 1, end_index, max_depth - 1);
    }
}
```

이 함수는 len이 16보다 작을 때 insertion sorting을 이용하고 max\_depth가 0이면 heap알고리즘을 사용합니다. 두 가지 경우에 해당하지 않으면 quick sorting을 이용해 records 배열의 start\_index부터 end\_index까지 오름차순으로 정렬합니다. 이 때 pivot은 중간값으로 setting하고 pivot보다 작으면 왼쪽에 두고 pivot보다 크면 오른쪽에 두는 방식을 반복적으로 수행한다. Introsort의 경우 시간복잡도는  $O(n \log n)$ 입니다.

## [6] insertionsort를 사용하여 mergesort의 속도를 향상시키는 방법

```
void RECORDS::merge(int start, int mid, int end) {
    int L_size = mid - start + 1;
    int R_size = end - mid;

    RECORD* left = new RECORD[L_size];
    RECORD* right = new RECORD[R_size];

    for (int i = 0; i < L_size; i++) {
        left[i] = records[start + i];
    }

    for (int i = 0; i < R_size; i++) {
        right[i] = records[mid + 1 + i];
    }

    int i_left = 0, j_right = 0;
    int key = start;

    while (i_left < L_size && j_right < R_size) {
        if (compare_keys(&left[i_left], &right[j_right]) <= 0) {
            records[key] = left[i_left];
            i_left++;
        } else {
            records[key] = right[j_right];
            j_right++;
        }
        key++;
    }

    while (i_left < L_size) {
        records[key] = left[i_left];
        i_left++;
        key++;
    }

    while (j_right < R_size) {
        records[key] = right[j_right];
        j_right++;
        key++;
    }

    delete[] left;
    delete[] right;
}

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 middle = start_index + (end_index - start_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);
    }
}
```

이 함수는 배열 길이가 16보다 작을 때 insertion sorting을 이용하고, 이 경우에 해당하지 않으면 merge sort를 적용해 records 배열의 start\_index부터 end\_index까지 오름차순으로 정렬합니다. 분할 단계에서는 배열을 중간 인덱스를 기준으로 두 부분으로 나누어 재귀적으로 정렬하고, 최종적으로 두 부분을 병합합니다. 병합 과정에서 임시 배열을 사용하여 각 부분 list의 요소를 비교하며 정렬된 상태로 원래 배열에 저장합니다. 전체 시간 복잡도는  $O(n \log n)$ 입니다.

## 4. 실험 진행 (3번의 시행)

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

\*\*\* Time for sorting with insertion sort = 52.647ms 45.228ms 44.634ms

평균 : 47.503ms

\*\*\* Time for sorting with heap sort = 1.670ms 1.430ms 1.610ms

평균 : 1.570 ms

\*\*\* Time for sorting with weird sort = 19.359ms 16.798ms 16.931ms

평균 : 17.696 ms

\*\*\* Time for sorting with classic quick sort = 0.922ms 1.340ms 0.936ms

평균 : 1.066 ms

\*\*\* Time for sorting with intro sort = 0.962ms 0.925ms 0.986ms

평균 : 0.958 ms

\*\*\* Time for sorting with merge+insertion sort = 1.643ms 2.117ms 1.421ms

평균 : 1.727 ms

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

\*\*\* Time for sorting with insertion sort = 166.161ms 152.578ms 144.435ms

평균: 154.391 ms

\*\*\* Time for sorting with heap sort = 3.198ms 2.788ms 2.390ms

평균: 2.792 ms

\*\*\* Time for sorting with weird sort = 95.202ms 83.059ms 64.320ms

평균: 80.860 ms

\*\*\* Time for sorting with classic quick sort = 2.706ms 2.072ms 1.808ms

**평균:** 2.195 ms

\*\*\* Time for sorting with intro sort = 2.525ms 2.525ms 1.980ms

**평균:** 2.343 ms

\*\*\* Time for sorting with merge+insertion sort = 4.011ms 4.165ms 2.597ms

**평균:** 3.591 ms

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

\*\*\* Time for sorting with insertion sort = 723.583ms 662.402ms 633.607ms

**평균:** 673.197 ms

\*\*\* Time for sorting with heap sort = 6.251ms 8.832ms 5.839ms

**평균:** 6.974 ms

\*\*\* Time for sorting with weird sort = 354.795ms 322.152ms 278.991ms

**평균:** 318.646 ms

\*\*\* Time for sorting with classic quick sort = 5.102ms 5.019ms 4.286ms

**평균:** 4.802 ms

\*\*\* Time for sorting with intro sort = 7.252ms 4.631ms 4.930ms

**평균:** 5.604 ms

\*\*\* Time for sorting with merge+insertion sort = 8.424ms 7.501ms 8.643ms

**평균:** 8.189 ms

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

\*\*\* Time for sorting with insertion sort = 3258.000ms 2648.721ms 3335.077ms

**평균:** 3080.599 ms

\*\*\* Time for sorting with heap sort = 23.625ms 16.754ms 21.865ms

**평균:** 20.081 ms

**\*\*\* Time for sorting with weird sort** = 1486.453ms 1099.597ms 2094.505ms

**평균:** 1560.185 ms

**\*\*\* Time for sorting with classic quick sort** = 10.507ms 8.804ms 21.668ms

**평균:** 13.660 ms

**\*\*\* Time for sorting with intro sort** = 13.082ms 10.551ms 13.858ms

**평균:** 12.497 ms

**\*\*\* Time for sorting with merge+insertion sort** = 20.143ms 14.649ms 24.111ms

**평균:** 19.634 ms

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

**\*\*\* Time for sorting with insertion sort** = 12447.106ms 11245.862ms 18441.371ms

**평균:** 14044.113 ms

**\*\*\* Time for sorting with heap sort** = 34.449ms 34.487ms 47.016ms

**평균:** 38.651 ms

**\*\*\* Time for sorting with weird sort** = 4978.284ms 4530.293ms 8405.525ms

**평균:** 5964.034 ms

**\*\*\* Time for sorting with classic quick sort** = 22.572ms 18.232ms 29.306ms

**평균:** 23.370 ms

**\*\*\* Time for sorting with intro sort** = 21.918ms 20.680ms 37.801ms

**평균:** 26.133 ms

**\*\*\* Time for sorting with merge+insertion sort** = 34.406ms 31.529ms 46.591ms

**평균:** 37.509 ms

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

**\*\*\* Time for sorting with insertion sort** = 50966.383ms 47488.973ms 156808.094ms

**평균: 85154.483 ms**

\*\*\* Time for sorting with heap sort = 78.609ms 82.031ms 189.242ms

**평균: 116.627 ms**

\*\*\* Time for sorting with weird sort = 21202.281ms 20746.619ms 152948.422ms

**평균: 64965.107 ms**

\*\*\* Time for sorting with classic quick sort = 52.192ms 43.262ms 179.737ms

**평균: 91.064 ms**

\*\*\* Time for sorting with intro sort = 42.168ms 45.704ms 190.758ms

**평균: 92.210 ms**

\*\*\* Time for sorting with merge+insertion sort = 68.095ms 82.125ms 238.148ms

**평균: 129.456 ms**

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

\*\*\* Time for sorting with insertion sort = 287166.719ms 265567.219ms 1001454.875ms

**평균: 518062.271 ms**

\*\*\* Time for sorting with heap sort = 239.569ms 200.682ms 491.228ms

**평균: 310.493 ms**

\*\*\* Time for sorting with weird sort = 108750.500ms 102574.500ms 369217.969ms

**평균: 193514.323 ms**

\*\*\* Time for sorting with classic quick sort = 108.174ms 123.196ms 237.234ms

**평균: 156.868 ms**

\*\*\* Time for sorting with intro sort = 110.913ms 99.759ms 233.559ms

**평균: 148.744 ms**

\*\*\* Time for sorting with merge+insertion sort = 168.738ms 142.571ms 769.586ms

**평균: 360.965 ms**

## 5. 분석 결과

### Comparison Sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	$n^2$	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space. <sup>[5][6]</sup>
Merge sort	$n \log n$	$n \log n$	$n \log n$	$n$	Yes	Merging	Highly parallelizable (up to $O(\log n)$ ) using the Three Hungarians' Algorithm). <sup>[7]</sup>
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	$n$	$n^2$	$n^2$	1	Yes	Insertion	$O(n + d)$ , in the worst case over sequences that have $d$ inversions.
Selection sort	$n^2$	$n^2$	$n^2$	1	No	Selection	Stable with $O(n)$ extra space or when using linked lists. <sup>[11]</sup>
Bubble sort	$n$	$n^2$	$n^2$	1	Yes	Exchanging	Tiny code size.

## [Algorithm]

1. 교과서적인 insertionsort 방법
2. 교과서적인 heapsort 방법
3. 먼저 minheap을 만든 후 insertionsort를 적용하는 방법
4. 최적화방법을 적용하지 않은 교과서적인 quicksort 방법
5. <https://en.wikipedia.org/wiki/Introsort>에 기술한 quicksort의 변형 방법
6. insertionsort를 사용하여 mergesort의 속도를 향상시키는 방법

Algorithm	16384	32768	65536	131072	262144	524288	1048576
1	47.503	154.391	673.197	3080.599	14044.113	85154.483	518062.271
2	1.570	2.792	6.974	20.081	38.651	116.627	310.493
3	17.696	80.860	318.646	1560.185	5964.034	64965.107	193514.323
4	1.066	2.195	4.802	13.660	23.370	91.064	156.868
5	0.958	2.343	5.604	12.497	26.133	92.210	148.744
6	1.727	3.591	8.189	19.634	37.509	129.456	360.965

### [1] 데이터 분석 : 교과서적인 insertionsort 방법

Algorithm	16384	32768	65536	131072	262144	524288	1048576
1	47.503	154.391	673.197	3080.599	14044.113	85154.483	518062.271

#### [1] N=16384와 N=32768의 관계

값 154.391은 47.503의 약 3.25배

#### [2] N=32768과 N=65536의 관계

값 673.197은 154.391의 약 4.36배

#### [3] N=65536과 N=131072의 관계

값 3080.599는 673.197의 약 4.58배

#### [4] N=131072와 N=262144의 관계

값 14044.113은 3080.599의 약 4.56배

#### [5] N=262144와 N=524288의 관계

값 85154.483은 14044.113의 약 6.06배

#### [6] N=524288와 N=1048576의 관계

값 518062.271은 85154.483의 약 6.08배

결론 : 이 알고리즘의 이론적인 시간 복잡도는  $O(n^2)$ 이며, 입력 크기 N이 증가할 때 수행 시간이  $N^2$ 에 비례해 증가한다. 즉, 입력 크기가 두 배가 되면 약 4배의 증가율을 이론적으로 보여야 한다. 각 단계에서 실제 측정된 수행 시간은 이론적인 시간 복잡도와 매우 유사한 증가율을 보이고 있다.

## [2] 데이터 분석 : 교과서적인 heapsort 방법

Algorithm	16384	32768	65536	131072	262144	524288	1048576
2	1.570	2.792	6.974	20.081	38.651	116.627	310.493

### [1] N=16384와 N=32768의 관계

값 2.792은 1.570의 약 1.78배

### [2] N=32768과 N=65536의 관계

값 6.974은 2.792의 약 2.50배

### [3] N=65536과 N=131072의 관계

값 20.081은 6.974의 약 2.88배

### [4] N=131072와 N=262144의 관계

값 38.651은 20.081의 약 1.92배

### [5] N=262144와 N=524288의 관계

값 116.627은 38.651의 약 3.02배

### [6] N=524288와 N=1048576의 관계

값 310.493은 116.627의 약 2.66배

결론 : 이 알고리즘의 이론적인 시간 복잡도는  $O(n \log n)$ 이며,  $n$ 에  $\log$  함수가 추가된 것이다.  $N$ 이 커질수록 로그 함수의 영향이 서서히 증가하게 된다. 이론적인 증가율은 약 2.33배이다. 각 단계에서 실제 측정된 수행 시간은 이론적인 시간 복잡도와 대체적으로 유사한 증가율을 보이고 있다.

## [3] 데이터 분석 : 먼저 minheap을 만든 후 insertionsort를 적용하는 방법

Algorithm	16384	32768	65536	131072	262144	524288	1048576
3	17.696	80.860	318.646	1560.185	5964.034	64965.107	193514.323

### [1] N=16384와 N=32768의 관계

값 80.860은 17.696의 약 **4.57배**

### [2] N=32768과 N=65536의 관계

값 318.646은 80.860의 약 **3.94배**

### [3] N=65536과 N=131072의 관계

값 1560.185는 318.646의 약 **4.89배**

### [4] N=131072와 N=262144의 관계

값 5964.034는 1560.185의 약 **3.82배**

### [5] N=262144와 N=524288의 관계

값 64965.107은 5964.034의 약 **10.89배**

### [6] N=524288와 N=1048576의 관계

값 193514.323은 64965.107의 약 **2.98배**

결론 : 이 알고리즘은 먼저 min-heap을 통해 데이터의 정렬 상태를 개선하고 나서 insertion sort를 적용하여 정렬 효율을 높이려는 목적이 있습니다. 그러나 실험 결과에 따르면 입력 크기 N이 증가할수록, 실제 시간 복잡도는  $O(n+d)$ 보다는 더욱 높은 증가율을 보였습니다. 이는 데이터가 커질수록 min-heap으로 구성된 리스트와 오름 차순 정렬 리스트 간의 차이가 더욱 커져, insertion sort가 예상보다 더 많은 연산을 수행하게 된 것으로 분석된다.

### [4] 데이터 분석 : 최적화방법을 적용하지 않은 교과서적인 quicksort 방법

Algorithm	16384	32768	65536	131072	262144	524288	1048576
4	1.066	2.195	4.802	13.660	23.370	91.064	156.868

### [1] N=16384와 N=32768의 관계

값 2.195는 1.066의 약 2.06배

### [2] N=32768과 N=65536의 관계

값 4.802는 2.195의 약 2.19배

### [3] $N=65536$ 과 $N=131072$ 의 관계

값 13.660은 4.802의 약 2.84배

### [4] $N=131072$ 와 $N=262144$ 의 관계

값 23.370은 13.660의 약 1.71배

### [5] $N=262144$ 와 $N=524288$ 의 관계

값 91.064는 23.370의 약 3.90배

### [6] $N=524288$ 와 $N=1048576$ 의 관계

값 156.868은 91.064의 약 1.72배

결론 : 이 알고리즘은 최적화 방법을 적용하지 않은 교과서적인 Quicksort 방식이다. 실험에서는 pivot을 배열의 가장 오른쪽 요소로 설정하여, pivot보다 작으면 왼쪽에, 크면 오른쪽에 위치시키는 정렬 방식을 적용했다. 이 알고리즘의 평균 시간 복잡도는  $O(n \log n)$ 이며, 최악의 경우  $O(n^2)$ 이다. 이론적으로 증가율은 약 2배에서 4배 사이로 예상되며, 실험 결과 역시 이러한 이론적 증가율과 비슷한 경향을 보였다. Quicksort는 pivot 위치에 따라 성능이 달라질 수 있어 성능 분석에 유의가 필요하다.

## [5] 데이터 분석 : <https://en.wikipedia.org/wiki/Introsort>에 기술한 quicksort의 변형 방법

Algorithm	16384	32768	65536	131072	262144	524288	1048576
5	0.958	2.343	5.604	12.497	26.133	92.210	148.744

### [1] $N=16384$ 와 $N=32768$ 의 관계

값 2.343는 0.958의 약 2.45배

### [2] $N=32768$ 과 $N=65536$ 의 관계

값 5.604는 2.343의 약 2.39배

### [3] $N=65536$ 과 $N=131072$ 의 관계

값 12.497는 5.604의 약 2.23배

#### [4] N=131072와 N=262144의 관계

값 26.133는 12.497의 약 2.09배

#### [5] N=262144와 N=524288의 관계

값 92.210는 26.133의 약 3.53배

#### [6] N=524288와 N=1048576의 관계

값 148.744는 92.210의 약 1.61배

결론 : 이 알고리즘은 list의 길이가 16보다 작을 때 insertion sorting을 이용하고 max\_depth가 0이면 heap알고리즘을 사용한다. 두 가지 경우에 해당하지 않으면 quick sorting을 이용해 오름차순으로 정렬한다. 이 때 pivot은 중간값으로 setting하고 pivot보다 작으면 왼쪽에 두고 pivot보다 크면 오른쪽에 두는 방식을 반복적으로 수행한다. 시간복잡도는  $O(n\log n)$ 이다. 이론적으로 증가율은 약 2.33배 정도의 증가율을 보여야 한다. 실험결과 이론적인 시간복잡도와 대체적으로 유사한 증가율을 보임을 확인할 수 있다.

### [6] 데이터 분석 : insertionsort를 사용하여 mergesort의 속도를 향상시키는 방법

Algorithm	16384	32768	65536	131072	262144	524288	1048576
6	1.727	3.591	8.189	19.634	37.509	129.456	360.965

#### [1] N=16384와 N=32768의 관계

값 3.591은 1.727의 약 2.08배

#### [2] N=32768과 N=65536의 관계

값 8.189는 3.591의 약 2.28배

#### [3] N=65536과 N=131072의 관계

값 19.634는 8.189의 약 2.40배

#### [4] N=131072와 N=262144의 관계

값 37.509는 19.634의 약 1.91배

#### [5] N=262144와 N=524288의 관계

값 129.456는 37.509의 약 3.45배

#### [6] N=524288와 N=1048576의 관계

값 360.965는 129.456의 약 2.79배

결론: 이 알고리즘은 배열 길이가 16보다 작을 때 insertion sorting을 이용하고, 이 경우에 해당하지 않으면 merge sort를 적용해 오름차순으로 정렬한다. 분할 단계에서는 배열을 중간 인덱스를 기준으로 두 부분으로 나누어 재귀적으로 정렬하고, 최종적으로 두 부분을 병합한다. 병합 과정에서 임시 배열을 사용하여 각 부분 list의 요소를 비교하며 정렬된 상태로 원래 배열에 저장한다. 전체 시간 복잡도는  $O(n \log n)$ 이다. 이론적으로 증가율은 약 2.33배 정도의 증가율을 보여야 한다. 실험결과 이론적인 시간 복잡도와 대체적으로 유사한 증가율을 보임을 확인할 수 있다.

## 6. 실험 결과

### [Algorithm]

1. 교과서적인 insertionsort 방법
2. 교과서적인 heapsort 방법
3. 먼저 minheap을 만든 후 insertionsort를 적용하는 방법
4. 최적화방법을 적용하지 않은 교과서적인 quicksort 방법
5. <https://en.wikipedia.org/wiki/Introsort>에 기술한 quicksort의 변형 방법
6. insertionsort를 사용하여 mergesort의 속도를 향상시키는 방법

Algorithm	16384	32768	65536	131072	262144	524288	1048576
1	47.503	154.391	673.197	3080.599	14044.113	85154.483	518062.271

2	1.570	2.792	6.974	20.081	38.651	116.627	310.493
3	17.696	80.860	318.646	1560.185	5964.034	64965.107	193514.323
4	1.066	2.195	4.802	13.660	23.370	91.064	156.868
5	0.958	2.343	5.604	12.497	26.133	92.210	148.744
6	1.727	3.591	8.189	19.634	37.509	129.456	360.965

교과서적인 insertion sorting은 실험 결과 이론적인 시간 복잡도와 일치하는  $O(n^2)$  증가율을 보였다. 6가지 알고리즘 중 가장 큰 시간 복잡도를 보이는 결과가 나타났다. Minheap-insertion sorting은 min heap을 만든 후 insertion sorting을 적용한다. 어느 정도 정렬된 상태에서 insertion sorting을 적용해 교과서적인 insertion sorting 알고리즈다 sorting하는 데 적은 시간이 걸렸다. 그러나 기대했던  $O(n)$ 시간 복잡도보다는 더욱 높은 시간 복잡도를 보였다. 이는 데이터가 커질수록 min-heap으로 구성된 리스트와 오름차순 정렬 리스트 간의 차이가 더욱 커져, insertion sort가 예상보다 더 많은 연산을 수행하게 된 것으로 분석된다. 교과서적인 heap sorting은  $O(n\log n)$ 으로 실험 결과 이론적인 시간 복잡도와 대체적으로 유사한 증가율을 보였다. 대체적으로 insertion sorting-merge sorting과 비슷한 시간 복잡도를 보였다. insertion sorting-merge sorting은  $O(n\log n)$ 으로 실험 결과 이론적인 시간 복잡도와 대체적으로 유사한 증가율을 보였다. 배열의 길이가 어떤 값 이하일 때 insertion sort를 이용하는 것이 효율적인지에 대해서 유의해야 한다. 최적화 방법을 사용하지 않은 quick sort는 평균 시간 복잡도는  $O(n\log n)$ 이며, 최악의 경우  $O(n^2)$ 이다. 이론적으로 증가율은 약 2배에서 4배 사이로 예상되며, 실험 결과 역시 이러한 이론적 증가율과 비슷한 경향을 보였다. 최적화 방법을 사용하지 않았기 때문에 pivot을 right에 두고 실험을 진행했다. 최악의 경우가  $O(n^2)$ 이라  $n$ 이 증가할수록 가장 높은 시간 복잡도를 가진 알고리즘이 될 것으로 예상했으나 intro sort와 더불어 6개의 알고리즘 중 시간 복잡도면에서 매우 효율적인 알고리즘으로 판단됐다. Intro sort는 list의 길이가 16보다 작을 때 insertion sorting을 이용하고 max\_depth가 0이면 heap 알고리즘을 사용한다. 두 가지 경우에 해당하지 않으면 quick sorting을 이용해 오름차순으로 정렬한다. 이 때 pivot은 중간 값으로 설정하고 pivot보다 작으면 왼쪽에 두고 pivot보다 크면 오른쪽에 두는 방식을 반복적으로 수행한다. 실험 결과 이론적인 시간 복잡도와 대체적으로 유사한 증가율을 보였다. 이때 처음에 max\_depth를 어떤 값으로 세팅할지 유의해야 한다. 처음 예상하기로 intro sorting이 가장 빠를 것으로 예상했으나 실험 결과 특

정  $n$ 에 대해 quick sorting이 더 빠른 경우도 나타났다. 두 알고리즘의 pivot setting 위치가 달라 절대적인 비교를 할 수는 없지만, intro sort는 최악의 경우에 heap sort로 전환하며, 이 과정에서 추가적인 오버헤드가 발생한다. 또한, heap sort 전환 시 메모리 접근이 비효율적이 돼 캐시 성능이 저하될 수 있다. 이에 특정  $n$ 에 대해 교과서적인 quick sort가 intro sort보다 더 빠른 경우가 발생할 수 있는 것으로 판단된다.