알고리즘 설계와 분석 Project3

서강대학교
SOGANG UNIVERSITY

담당 교수 : 임인성

이름 : 박민준

학번 : 20212020

1. Algorithm Description

1.1. void sort_records_heap (int start_index, int end_index);

[시간 복잡도]

- Average_Case: O(nlogn)

- Worst_Case: O(nlogn)

```
void max_heapify(RECORD* records, int root, int n) {
    int child, rootkey;
    rootkey = records[root].key;
    RECORD tmp = records[root];
    child = 2 * root + 1;

    while (child < n) {
        if (child < n - 1 && 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
    int n = end_index - start_index + 1;

    for (int i = start_index + n / 2 - 1; i >= start_index; i--) {
        max_heapify(records, i, n);
    }

    for (int i = end_index; i > start_index; i--) {
        std::swap(records[0], records[i]);
        max_heapify(records, start_index, i);
    }
}
```

- sort_records_heap() 함수는 최대 힙을 사용하여 records 배열을 정렬한다.

- 먼저, 주어진 배열을 최대 힙으로 변환하여, 배열의 루트에 가장 큰 값이 위치하도록 한다. child를 가지는 node 중에서 인덱스가 가장 큰 값부터 start_index 값까지 for loop를 돌며 최대 힙을 구성한다. 이 때, 최대 힙을 유지하도록 만들어주는 max_heapify() 함수를 이용한다.

- 그 다음 for loop에서는 가장 큰 값을 가지는 루트 노드를 마지막 요소와 교환하고, 힙 크기를 줄인 후 다시 최대 힙을 구성한다. 이를 반복하여 모든 요소를 오름차순 정렬한다.

- max_heapify() 함수는 힙 정렬에서 최대 힙을 유지하기 위해 사용된다. root node의 값이 child node의 값보다 작으면, child node 중 더 큰 값과 교환한다. root와 child를 교환한 후, 최대 힙 특성을 만족할 때까지 교환된 child를 새로운 루트로 설정하여 내려가면서 while loop를 반복한다. while loop가 끝나면 주어진 노드를 root로 하는 subtree는 최대 힙 특성을 가지게 되며, 해당 subtree의 최대값이 root에 위치하게 된다.

1.2. void sort_records_weird (int start_index, int end_index);

[시간 복잡도]

1) min heap 구성: O(n)

2) insertion sort 적용: $O(n^2)$, Best_Case - O(n)

- 전체 시간 복잡도: $O(n^2)$, Best_Case – O(n)

```
void min_heapify(RECORD* records, int root, int n) {
    int child, rootkey;
    rootkey = records[root].key;
    RECORD tmp = records[root];
    child = 2 * root + 1;

    while (child < n) {
        if (child < n - 1 && 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
    int n = end_index - start_index + 1;

    for (int i = start_index + n / 2 - 1; i >= start_index; i--) {
        min_heapify(records, i, n);
    }

    sort_records_insertion(start_index, end_index);
}
```

- sort_records_weird() 함수는 먼저 최소 힙을 만든 후 insertion sort를 적용하는 방법이다.

- sort_records_heap() 함수와 마찬가지로 먼저 min_heapify() 함수를 사용하여 records 배열을 최

소 힙으로 변환한다. 이후 최소 힙으로 구성된 records 배열에 대해 insertion sort를 수행하여 정렬을 마무리한다. insertion sort는 이미 구현되어 있는 sort_records_insertion() 함수를 이용한다. 해당 함수에 대한 설명은 생략하겠다.

- min_heapify() 함수는 left child와 right child의 값을 비교하는 부분, rootkey와 child값을 비교하는 부분에서 부등호의 방향이 반대인 것을 제외하고는 max_heapify() 함수와 동일하다. 따라서 max_heapify() 함수와 반대로 min_heapify() 함수의 while loop가 끝나면 주어진 노드를 root로 하는 subtree는 최소 힙 특성을 가지게 되며, 해당 subtree의 최솟값이 root에 위치하게 된다.

1.3.void sort_records_quick_classic (int start_index, int end_index);

[시간 복잡도]

- Average_Case: O(nlogn)

- Worst_Case: O(n^2)

```
int partition(RECORD* records, int left, int right) {
    int i, pivot;

    pivot = left;
    for (i = left; i < right; i++) {
        if (records[i].key < records[right].key) {
            std::swap(records[i], records[pivot]);
            pivot++;
        }
    }
    std::swap(records[right], records[pivot]);
    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(records, start_index, end_index);

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

- sort_records_quick_classic() 함수는 최적화 방법을 적용하지 않은 교과서적인 quick sort 방법이다. partition() 함수를 통해 pivot을 기준으로 배열을 나눈다.

- 이후 두 부분 배열에 대해 sort_records_quick_classic() 함수가 재귀적으로 호출되며, 정렬이 완료될 때까지 이를 반복한다.

- partition() 함수는 quick sort Algorithm의 핵심 구성 요소로, 배열을 pivot 값을 기준으로 두 부분으로 분할하는 역할을 한다. pivot보다 작은 값은 왼쪽에, pivot보다 큰 값은 오른쪽에 위치하도록 배열을 재배치하여 quick sort가 재귀적으로 두 부분 배열을 정렬할 수 있도록 만든다.

- 자세한 코드 구현 방식은 다음과 같다. 배열의 마지막 요소인 right의 위치를 기준값으로 선택한다. 즉, records[right].key가 pivot 값이며 이 값에 따라 배열을 두 부분으로 나눈다. pivot 변수는 최종적으로 pivot이 위치할 인덱스를 나타내며, 초기값은 left로 설정된다. 이 변수는 pivot보다 작은 요소를 찾았을 때 해당 요소와 교환할 위치를 추적한다. for loop를 통해 left부터 right-1까지 배열의 모든 요소를 순회하며 pivot 값과 비교한다. for loop가 끝나면 pivot 위치의 왼쪽에는 모두 pivot 값보다 작은 값들만 있고, 오른쪽에는 pivot 값보다 큰 값들이 존재하게 된다. 마지막으로 pivot 값을 반환하며 이 pivot의 위치는 quick sort의 재귀 호출 시 두 부분으로 나누는 기준이 된다.

1.4.void sort_records_intro (int start_index, int end_index);

[시간 복잡도]

- Average_Case: O(nlogn)

- Worst_Case: O(nlogn)

```
void introsort(RECORDS* obj, RECORD* records, int left, int right, int max_depth) {
    int n = right - left + 1;

    if (n < 16) {
        obj->sort_records_insertion(left, right);
    }
    else if (max_depth == 0) {
        obj->sort_records_heap(left, right);
    }
    else {
        int pivot = partition(records, left, right);
        introsort(obj, records, left, pivot - 1, max_depth - 1);
        introsort(obj, records, pivot + 1, right, max_depth - 1);
    }
}

void RECORDS::sort_records_intro(int start_index, int end_index) {
    // Introsort described in https://en.wikipedia.org/wiki/Introsort
    int n = end_index - start_index + 1;
    int max_depth = 3 * std::log2(n);
    introsort(this, records, start_index, end_index, max_depth);
}
```

- sort_records_intro() 함수는 quick sort의 변형 방법으로, quick sort와 heap sort를 결합한 정렬 알고리즘이다. 재귀의 깊이가 일정 수준을 넘으면 quick sort의 Worst_Case 성능을 방지하기 위해

heap sort로 전환한다.

- 초기에는 quick sort로 시작하며 sort_records_quick_classic() 함수에서 사용한 partition 함수를 재사용하여 배열을 pivot 기준으로 두 부분으로 분할한다.

- max_depth는 $3*\log_2 n$으로 설정하며, 만약 재귀 깊이가 max_depth에 도달하면 heap sort로 전환된다. quick sort는 최악의 경우 $O(n^2)$의 성능을 가지므로 재귀 깊이가 max_depth에 도달했을 때는 heap sort로 전환하여 최악의 경우에도 O(nlogn)의 시간 복잡도를 보장할 수 있다.

- 배열의 크기 n이 작은 경우에는 quick sort보다 insertion sort가 더 빠르게 동작할 수 있다. insertion sort는 비교와 이동이 많지만, 작은 배열에서는 오버헤드가 적어 더 효율적이기 때문이다. 따라서 n이 16보다 작으면 sort_records_insertion() 함수를 호출하여 삽입 정렬로 정렬을 완료한다.

- 처음에는 max_depth를 $2*\log_2 n$으로 설정하였지만 input_size가 2^{19}, 2^{20}와 같이 커짐에 따라 정렬 결과에 오류가 발생하였다. 크기가 큰 배열에서는 max_depth가 예상보다 더 깊어질 수 있으며, 이로 인해 quick sort의 재귀 깊이가 제한을 초과하거나 충분하지 않아 문제가 발생할 수 있다. 따라서 max_depth를 $3*\log_2 n$으로 설정하였고, 올바른 정렬 결과를 얻을 수 있었다.

1.5.void sort_records_merge_with_insertion (int start_index, int end_index);

[시간 복잡도]

- Average_Case: O(nlogn)

- Worst_Case: O(nlogn)

```
void RECORDS::sort_records_merge_with_insertion(int start_index, int end_index) {
    // Merge sort optimized by insertion sort only
    int n = end_index - start_index + 1;

    if (n < 16) {
        sort_records_insertion(start_index, end_index);
    }
    else 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(records, start_index, middle, end_index);
    }
}
```

- sort_records_merge_with_insertion() 함수는 insertion sort를 사용하여 merge sort의 속도를 향상

시키는 방법이다. 즉, insertion sort와 merge sort를 결합한 방식으로 큰 배열에서는 merge sort를 사용하여 정렬을 수행하고, 작은 배열에서는 insertion sort로 정렬을 최적화하는 알고리즘이다.

- 먼저 n을 계산하여 현재 정렬할 배열의 크기를 파악한다. 배열의 크기에 따라 merge sort와 insertion sort 중 어떤 정렬 방식을 사용할지 결정한다.

- 배열의 크기가 16보다 작을 경우 sort_records_insertion() 함수를 호출하여 삽입 정렬을 수행한다. 삽입 정렬은 $O(n^2)$의 시간 복잡도를 가지지만, sort_records_intro() 함수에서 설명하였듯이 배열의 크기가 작을 때는 실제로 오버헤드가 적고 빠르게 정렬을 완료할 수 있다.

- 배열의 크기가 16 이상인 경우에는 merge sort 방식을 사용하여 배열을 재귀적으로 두 부분으로 나누고 정렬한다. 왼쪽 부분과 오른쪽 부분이 각각 정렬되면, merge() 함수를 호출하여 두 부분을 합쳐 최종 정렬을 완료한다.

```
void merge(RECORD* records, int left, int middle, int right) {
    int size = right - left + 1;
    RECORD* buffer = new RECORD[size];
    memcpy(buffer, records + left, sizeof(RECORD) * size);

    int i_left = 0;
    int i_right = middle - left + 1;
    int i = left;

    while (i_left <= middle - left && i_right <= right - left) {
        if (buffer[i_left].key < buffer[i_right].key)
            records[i++] = buffer[i_left++];
        else
            records[i++] = buffer[i_right++];
    }

    while (i_left <= middle - left)
        records[i++] = buffer[i_left++];
    while (i_right <= right - left)
        records[i++] = buffer[i_right++];

    delete[] buffer;
}
```

- merge() 함수의 동작 방식은 다음과 같다. buffer 배열을 동적으로 생성하여 records 배열의 left부터 right까지의 데이터를 복사한다. 이 임시 버퍼를 사용하여 원본 배열을 손상시키지 않고 두 부분 배열을 병합할 수 있다.

- buffer 배열에서 i_left와 i_right pointer를 사용하여 왼쪽 부분과 오른쪽 부분을 순차적으로 비교한다. buffer[i_left]가 buffer[i_right]보다 작으면 records[i]에 buffer[i_left]의 값을 넣고 i_left를 증가시킨다. 반대로 buffer[i_right]가 작으면 records[i]에 그 값을 넣고 i_right를 증가시킨다. i는 원본

records 배열의 위치를 가리키며, 병합된 결과를 이 위치에 저장한다.

- 어느 한쪽 부분 배열이 먼저 끝나면 나머지 부분 배열의 모든 요소를 records에 그대로 복사한다. 이 과정을 통해 두 부분 배열을 하나의 정렬된 배열로 병합할 수 있다.

- 각 정렬 알고리즘에 대한 시간 복잡도를 정리하면 다음과 같다.

정렬 알고리즘	시간 복잡도
Insertion sort	**최악: $O(n^2)$, 최선: O(n)**
Heap sort	**O(nlogn)**
Weird sort (min heap + insertion sort)	**최악: $O(n^2)$, 최선: O(n)**
Quick sort	**O(nlogn), 최악: $O(n^2)$**
Intro sort	**O(nlogn)**
Merge + Insertion sort	**O(nlogn)**

2. Actual Running Time

[실험 환경]

OS: Windows 11 Home

CPU: Intel(R) Core(TM) i5-1240P @ 1.70GHz

RAM: 16.00GB

Compiler: Visual Studio 2022 Release Mode/x64 Platform

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

*** Time for sorting with insertion sort = 287506.750ms
*** Sorting complete!

*** Time for sorting with heap sort = 189.904ms
*** Sorting complete!

*** Time for sorting with weird sort = 111545.867ms
*** Sorting complete!

*** Time for sorting with classic quick sort = 93.322ms
*** Sorting complete!

*** Time for sorting with intro sort = 105.645ms
*** Sorting complete!

*** Time for sorting with merge+insertion sort = 244.091ms
*** Sorting complete!

C:\Users\mj486\OneDrive\바탕 화면\민준\서강대\4-2학기\알고리즘설계와분석\과제\Lab3\SortingM
ethods\x64\Release\SortingMethods.exe(프로세스 34256개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

[사진1. $n=2^{20}$ 인 경우 시간 측정 1회차]

- main.cpp 파일 내에서 "measure_cpu_time.h"를 이용하여 각 정렬 방식에 대한 실제 수행 시간을 측정하였고, 5회 측정 시간의 평균 값을 계산하였다.

- 데이터의 크기(input_size)는 main.cpp 파일 내용 그대로 2^{14} ~ 2^{20} 까지 설정하였다.

1) $n=2^{14}$ **(단위: ms)**

	insertion	heap	weird	quick	intro	merge+insertion
1회	47.444	1.577	19.654	0.914	0.834	1.759
2회	55.984	1.566	21.069	0.934	0.895	1.913
3회	43.768	1.361	18.170	0.885	0.849	1.811
4회	44.268	1.300	17.955	0.889	0.825	1.712
5회	60.824	1.719	24.567	1.224	1.129	1.997
평균	50.458	1.505	20.283	0.969	0.906	1.838

2) $n=2^{15}$ **(단위: ms)**

	insertion	heap	weird	quick	intro	merge+insertion
1회	162.657	2.498	70.254	1.895	1.836	3.922
2회	175.094	3.627	78.027	2.199	2.021	5.480
3회	158.955	3.494	72.853	2.384	1.888	3.734
4회	159.093	2.526	76.909	2.608	2.225	3.741
5회	237.191	3.952	101.908	2.657	2.479	4.401
평균	178.598	3.219	79.990	2.349	2.089	4.256

3) $n=2^{16}$ **(단위: ms)**

	insertion	heap	weird	quick	intro	merge+insertion
1회	721.242	6.559	318.053	4.654	4.126	7.057
2회	712.652	6.529	312.090	7.377	4.078	7.017
3회	706.193	6.608	328.217	4.619	4.175	6.887
4회	714.787	6.830	315.164	4.387	4.108	7.432
5회	962.008	8.126	425.781	5.452	5.047	8.428
평균	763.376	6.930	339.861	5.298	4.307	7.364

4) $n=2^{17}$ (단위: ms)

	insertion	heap	weird	quick	intro	merge+insertion
1회	2580.933	14.466	1240.891	9.630	9.115	14.041
2회	2862.537	14.370	1275.876	9.672	9.150	13.940
3회	2889.896	14.699	1274.415	9.574	9.070	14.154
4회	3874.455	14.593	1249.532	9.616	9.108	14.375
5회	4077.429	18.149	1714.446	11.786	11.341	18.008
평균	3257.050	15.255	1351.032	10.056	9.557	14.904

5) $n=2^{18}$ (단위: ms)

	insertion	heap	weird	quick	intro	merge+insertion
1회	12108.993	32.706	5176.860	19.724	19.251	31.004
2회	12152.108	32.083	5284.196	20.351	18.810	30.610
3회	12041.050	32.319	5204.758	19.803	18.557	30.809
4회	12026.853	34.549	5261.903	21.091	18.960	30.963
5회	16066.745	40.483	6802.707	24.810	24.196	38.638
평균	12879.149	34.428	5546.085	21.156	19.955	32.405

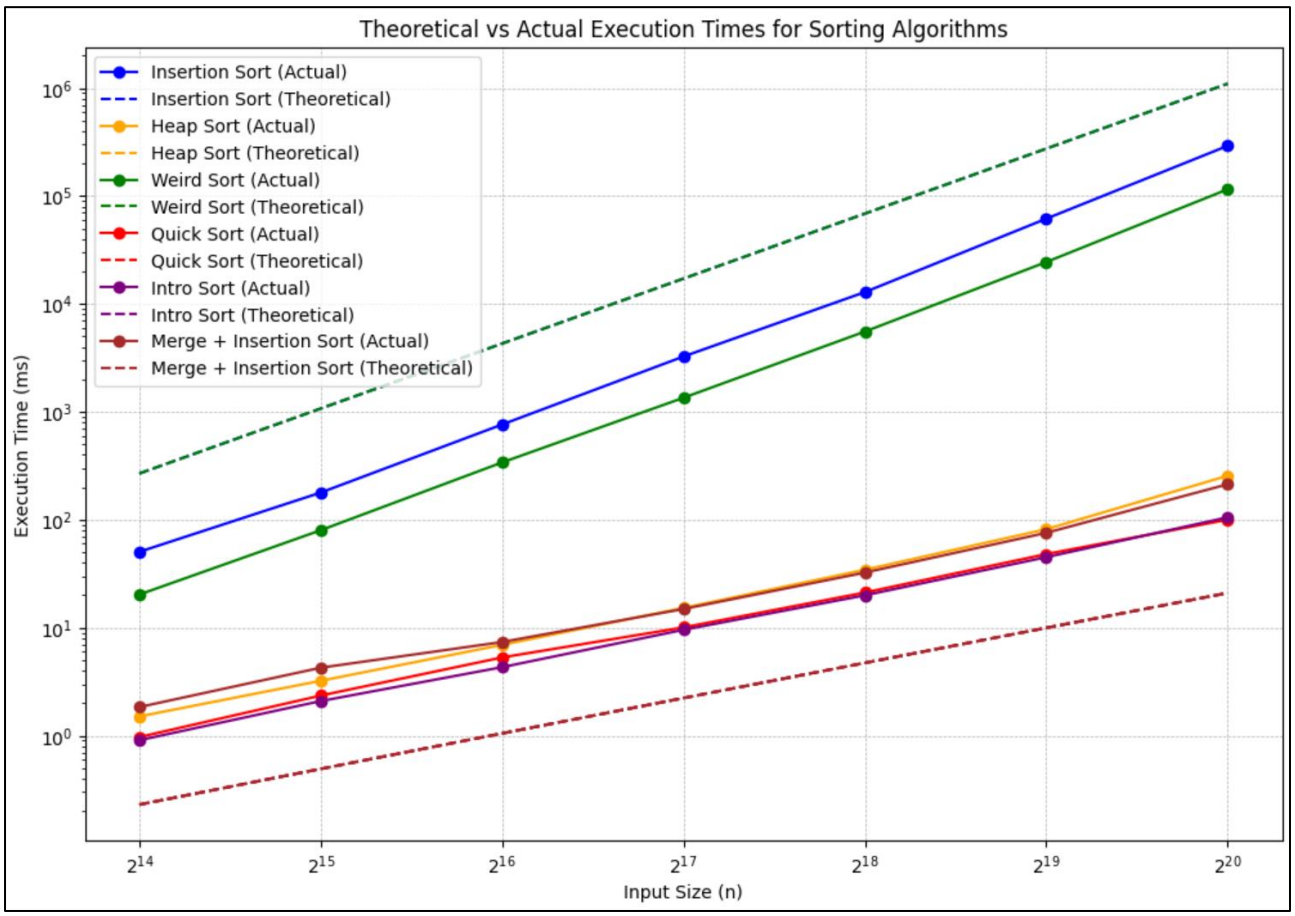
6) $n=2^{19}$ (단위: ms)

	insertion	heap	weird	quick	intro	merge+insertion
1회	52317.938	74.411	21792.900	43.544	41.224	66.882
2회	51449.355	75.177	21872.018	45.425	41.759	66.741
3회	51505.703	74.357	21612.242	42.995	41.362	67.101
4회	79139.953	80.480	25342.260	45.065	42.637	84.946
5회	73874.016	104.447	31683.680	63.281	58.104	92.498
평균	61657.393	81.774	24460.620	48.062	45.017	75.634

7) $n=2^{20}$ (단위: ms)

	insertion	heap	weird	quick	intro	merge+insertion
1회	287506.750	189.904	111545.867	93.322	105.645	244.091
2회	276005.656	203.577	110288.945	96.804	89.188	183.510
3회	257056.875	188.632	98858.156	93.093	89.159	175.785
4회	290072.438	438.156	116872.641	95.962	89.869	184.311

5회	349111.906	258.975	139678.984	121.764	152.906	279.003
평균	291950.725	255.849	115448.919	100.189	105.353	213.340



[사진2. 이론적인 수행 시간 vs 실제 수행 시간]

- 파이썬의 matplotlib.pyplot을 이용해 이론적인 수행 시간과 실제 수행 시간 관의 관계를 그래프로 그려보면 위 사진과 같다.

- Insertion Sort와 Weird Sort의 이론적인 시간 복잡도는 $O(n^2)$으로 같고, 나머지 Heap Sort, Quick Sort, Intro Sort, Merge+Insertion Sort의 이론적인 시간 복잡도도 O(nlogn)으로 같기 때문에 그래프가 하나로 겹쳐진다.

- 이론적인 수행 시간은 CPU가 초당 1억번의 연산을 수행한다고 가정하고, 연산 횟수를 1억으로 나누어 계산하였다. 이후 1000을 곱해 밀리초(ms) 단위로 값을 표현하였다.

[변환 식]

$$\textit{밀리초 단위 시간}(ms) = \frac{\textit{연산 횟수}}{10^9} x\ 10^3$$

- 예를 들어, Insertion sort의 시간 복잡도가 n^2일 때 $n=2^{14}$인 경우 2^{28}=268,435,456이 되므로

이를 1억으로 나눈 후, 1000을 곱해 ms 단위로 나타내면 약 268.435ms가 된다.

[그래프 분석]

- 그래프를 보면 Insertion Sort가 가장 높은 수행 시간을 가지며, 특히 큰 입력 n에서 가장 느린 성능을 보이는 것을 확인할 수 있다. n이 커질수록 실행 시간이 기하급수적으로 증가하는데, y축이 log scale로 설정되어 있어 증가 속도가 완만해 보이지만 실제로는 큰 입력에서 수행 시간이 급격히 증가하게 된다.

- Weird Sort는 다른 O(nlogn) 알고리즘보다 훨씬 느린 성능을 보인다. 입력 크기가 증가할수록 성능이 빠르게 저하된다. 그래프를 보면 Insertion Sort보다 조금 더 빠른 수행 시간을 갖지만, n이 커질수록 매우 비슷한 속도로 수행 시간이 증가하는 것을 확인할 수 있다. 이는 Weird Sort가 Best_Case에서 O(n)의 시간 복잡도를 가지지만 일반적으로는 Worst_Case인 $O(n^2)$의 시간 복잡도를 갖는다는 것을 알 수 있다.

- Heap Sort는 평균 및 최악의 경우에도 O(nlogn) 성능을 유지하는 안정적인 정렬 알고리즘이다. 그래프를 보면 Heap Sort의 실제 수행 시간은 n이 커짐에 따라 선형적으로 증가하여 안정적인 성능을 보이는 것을 확인할 수 있다. 또한, 중간 크기 이하의 데이터에서는 Heap Sort가 Merge+Insertion Sort보다 수행 시간이 더 빨랐지만 중간 크기 이상의 데이터에서는 반대로 수행 속도가 더 느려졌다.

- Merge+Insertion Sort는 Merge Sort와 Insertion Sort를 혼합하여 작은 부분 배열에서는 Insertion Sort를, 큰 배열에서는 Merge Sort를 사용하는 방식이며 연속적인 메모리 접근을 통해 캐시 친화적으로 작동한다. 작은 배열에서는 데이터가 메모리의 근처에 저장되기 때문에 cache hit가 자주 발생하여 메모리 접근 속도가 빨라지게 된다. 그래프를 보면 중간 크기 이상의 입력에 대해 Heap Sort와 유사하지만 조금 더 빠른 성능을 보이며, Quick Sort와 Intro Sort에 비해 다소 느리지만 캐시 효율이 높아 큰 배열에서도 좋은 성능을 나타내는 것을 알 수 있었다.

- Quick Sort는 평균적으로 O(nlogn)의 복잡도를 가지며 대부분의 경우 매우 빠른 정렬 성능을 보인다. 이번 과제에서는 최적화 방법을 적용하지 않았기 때문에 최악의 경우 $O(n^2)$ 성능이 나타날 수 있었지만 실제로는 그렇지 않았다. 그래프를 보면 Heap Sort 또는 Merge Sort에 비해 빠른 성능을 보였으며, 중간 크기 이하의 데이터에 대해서는 Intro Sort보다 느린 성능을 보였다.

- Intro Sort는 Quick Sort와 Heap Sort를 혼합하여 설계된 알고리즘으로 최악의 경우에도 안정적인 성능을 보장할 수 있다. 그래프를 보면 Quick Sort와 매우 유사한 성능을 보이며, 최악의 경우가 발생하지 않도록 설계되었기 때문에 큰 입력에서도 일관된 성능을 유지한다. 중간 크기 이하의 데이터에 대해서는 오히려 Quick Sort보다 빠른 성능을 보이는 것을 확인할 수 있었다.

- 각 정렬 알고리즘은 모두 이론적 시간 복잡도와 실제 성능이 대부분 유사하다. O(nlogn)의 시간 복잡도를 갖는 Quick Sort, Heap Sort, Intro Sort, Merge+Insertion Sort 알고리즘은 입력 크기가 커질 때도 성능이 안정적으로 증가하는 것을 확인할 수 있었다. 따라서 이러한 알고리즘들은 일반적으로 대규모 데이터셋에서도 매우 효율적이다.

- 반면, Insertion Sort는 시간 복잡도가 $O(n^2)$이기 때문에 데이터 크기가 커질수록 수행 시간이 기하급수적으로 증가하여 큰 입력에서는 매우 비효율적이다.

- Weird Sort 알고리즘은 이론적인 시간 복잡도를 정확히 예측하기는 어렵지만, 실제 수행 시간 측정 결과 $O(n^2)$과 O(nlogn) 사이의 시간 복잡도를 갖는다고 볼 수 있다. 따라서 중간 크기 이상의 데이터셋에 대해서는 다른 O(nlogn) 알고리즘에 비해 성능이 급격히 저하되며 Insertion Sort와 비슷한 성능을 갖게 되어 비효율적이다.

[최적화 기법의 효과 분석]

1) Weird Sort

- Weird Sort는 큰 배열에 대해 비효율적이므로 최적화 효과가 크지 않았다. 실제로 최적화 없이 성능이 급격히 저하되는 결과를 가졌으며 다른 알고리즘보다 비효율적인 성능을 보였다.

2) Intro Sort

- Intro Sort는 Quick Sort의 단점인 최악의 경우 성능 저하 문제를 해결하여 항상 O(nlogn) 성능을 보장한다. 이러한 최적화 기법 덕분에 안정적이면서도 빠른 성능을 유지할 수 있으며, 실제도로 가장 우수한 성능을 제공하였다.

3) Merge + Insertion Sort

- Merge Sort와 Insertion Sort를 혼합하여 작은 부분 배열에서는 Insertion Sort를 사용하여 캐시 효율성을 높였다. 이러한 최적화 기법 덕분에 큰 배열에서도 캐시 친화적으로 작동하여 성능을 개선할 수 있었다. 특히 대규모 데이터에서 Heap Sort보다 빠르고 안정적인 성능을 보였으며 캐시 효율성을 활용한 최적화 효과가 컸다.

*Python code for graph

```python
import matplotlib.pyplot as plt

# Input Size: 2^14, 2^15, ..., 2^20
n_values = [2**i for i in range(14, 21)]

# 실제 측정된 수행 시간 (단위: ms)
actual_times = {
    'Insertion Sort': [50.458, 178.598, 763.376, 3257.05, 12879.149, 61657.393, 291950.725],
    'Heap Sort': [1.505, 3.219, 6.93, 15.255, 34.428, 81.774, 255.849],
    'Weird Sort': [20.283, 79.99, 339.861, 1351.032, 5546.085, 24460.62, 115448.919],
    'Quick Sort': [0.969, 2.349, 5.298, 10.056, 21.156, 48.062, 100.189],
    'Intro Sort': [0.906, 2.089, 4.307, 9.557, 19.955, 45.017, 105.353],
    'Merge + Insertion Sort': [1.838, 4.256, 7.364, 14.904, 32.405, 75.634, 213.34]
}

# 이론적인 수행 시간
theoretical_times = {
    'Insertion Sort': [n**2 / 1e6 for n in n_values],                  # O(n^2)
    'Heap Sort': [n * np.log2(n) / 1e6 for n in n_values],             # O(n log n)
    'Weird Sort': [n**2 / 1e6 for n in n_values],                      # O(n^2)
    'Quick Sort': [n * np.log2(n) / 1e6 for n in n_values],            # O(n log n)
    'Intro Sort': [n * np.log2(n) / 1e6 for n in n_values],            # O(n log n)
    'Merge + Insertion Sort': [n * np.log2(n) / 1e6 for n in n_values]  # O(n log n)
}

# 각 알고리즘 별로 다양한 색상 설정
colors = {
    'Insertion Sort': 'blue',
    'Heap Sort': 'orange',
    'Weird Sort': 'green',
    'Quick Sort': 'red',
    'Intro Sort': 'purple',
    'Merge + Insertion Sort': 'brown'
}
```

```python
# 그래프 시각화
plt.figure(figsize=(12, 8))

for algorithm, time_list in actual_times.items():
    color = colors[algorithm]
    # 실제 수행 시간 플롯 (실선)
    plt.plot(n_values, time_list, marker='o', color=color, label=f'{algorithm} (Actual)')
    # 이론적인 수행 시간 플롯 (점선)
    plt.plot(n_values, theoretical_times[algorithm], linestyle='--', color=color, label=f'{algorithm} (Theoretical)')

# log scale 설정
plt.xscale('log', base=2)
plt.yscale('log')

# 축 및 제목 설정
plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (ms)")
plt.title("Theoretical vs Actual Execution Times for Sorting Algorithms")
plt.legend()
plt.grid(True, linestyle="--", linewidth=0.5)

# x축에 2^14 ~ 2^20 레이블 추가
plt.xticks(n_values, [f"$2^{{{i}}}$" for i in range(14, 21)])

# 그래프 표시
plt.show()
```