**컴퓨터공학과 2학년 20171697 최민영**

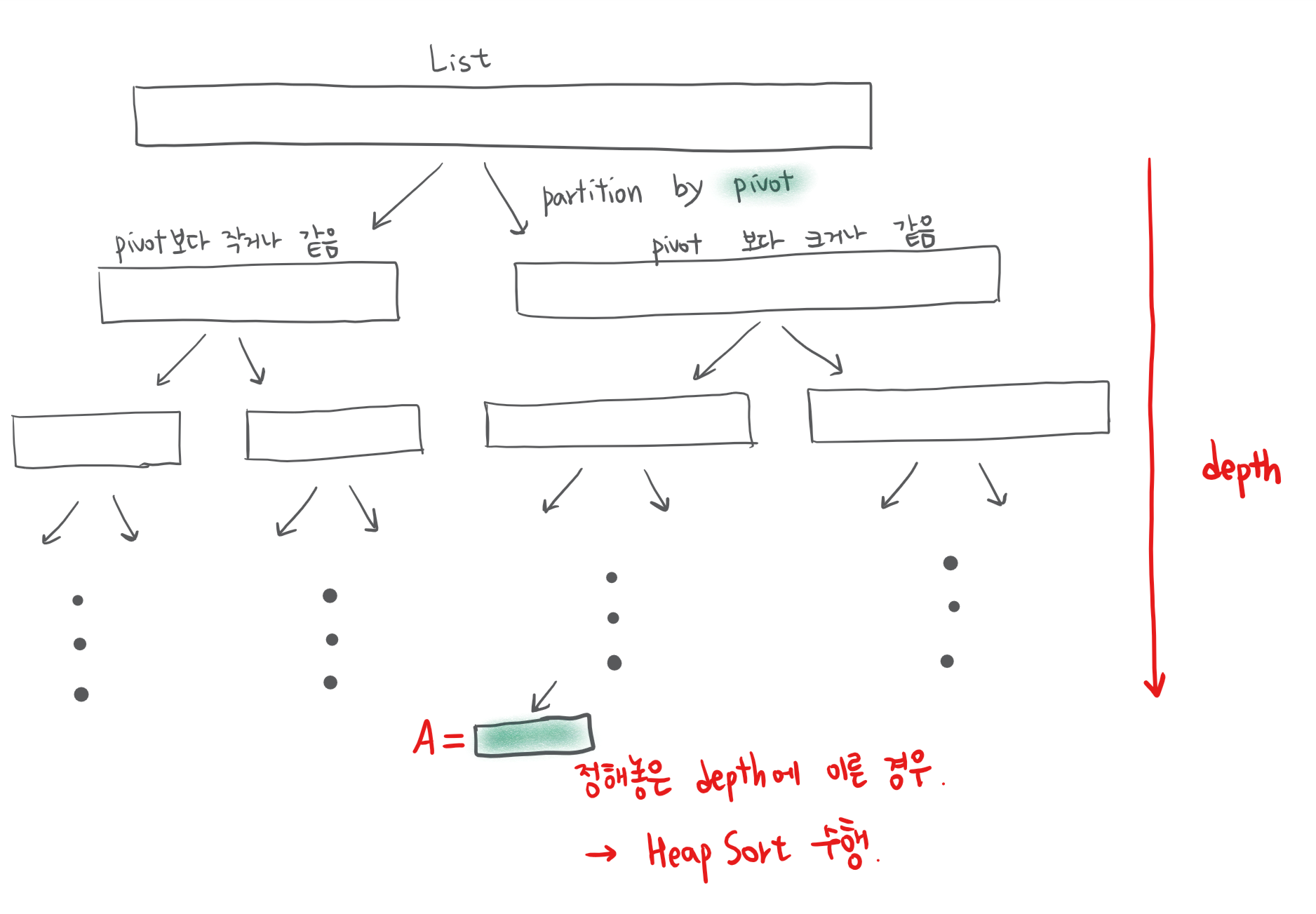
**CSE3081-1: Design and Analysis of Algorithms**

**Machine Problem 2: Master of Sorting**

**REPORT**

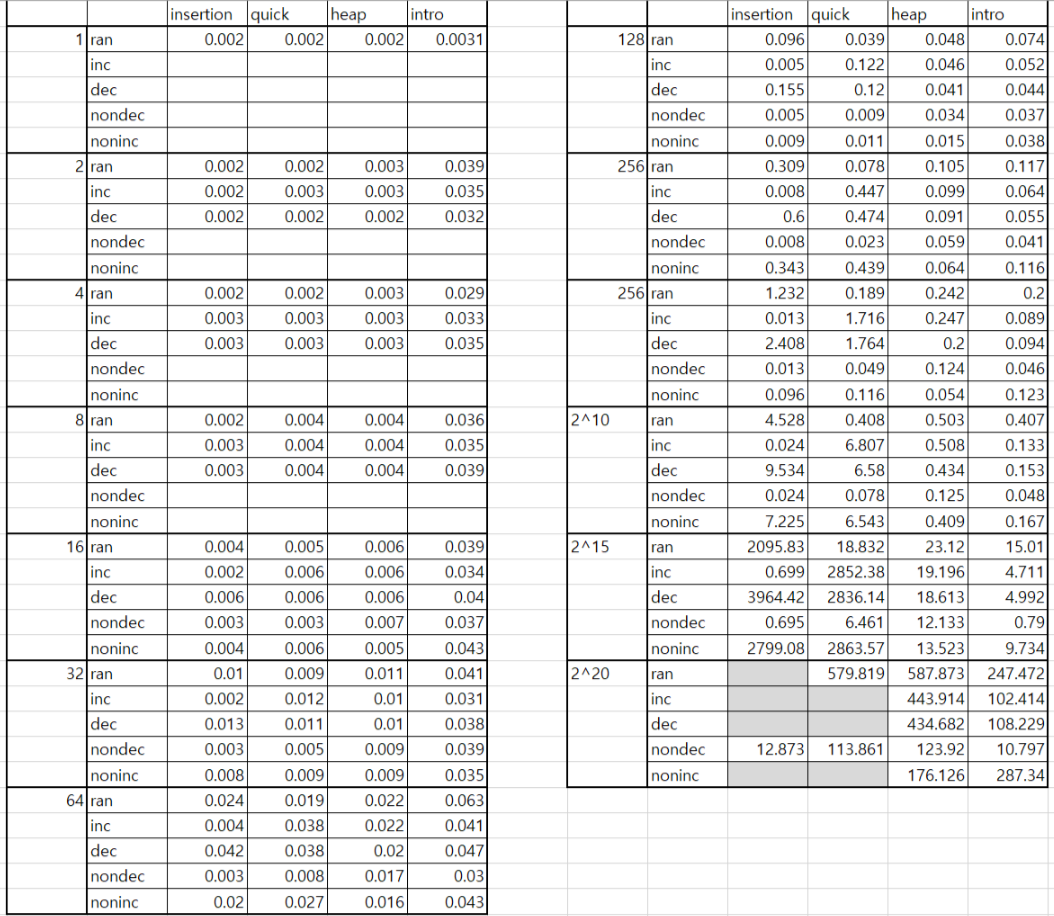
* Algorithm 4 Design

heap\_sort와 quick\_sort를 합친 Intro\_sort를 구현하였다. quick sort는 주어진 숫자열을 쪼개서 다시 살펴보는 수행을 반복하는 데 이러한 특징이 최악의 상황일 경우 O(n2)에 이르게 한다. 그래서 쪼개서 다시 살펴보는 반복 수행이 정해놓은 횟수를 넘어가면 heap\_sort로 바꾸는 방법을 취하였다. wiki pedia를 참조한 결과, 정해놓는 횟수를 ⌊log( length( List ) ) ⌋× 2 로 정하였다. 그렇게 worst한 경우에도 O(nlogn)을 유지하도록 구현하였다.



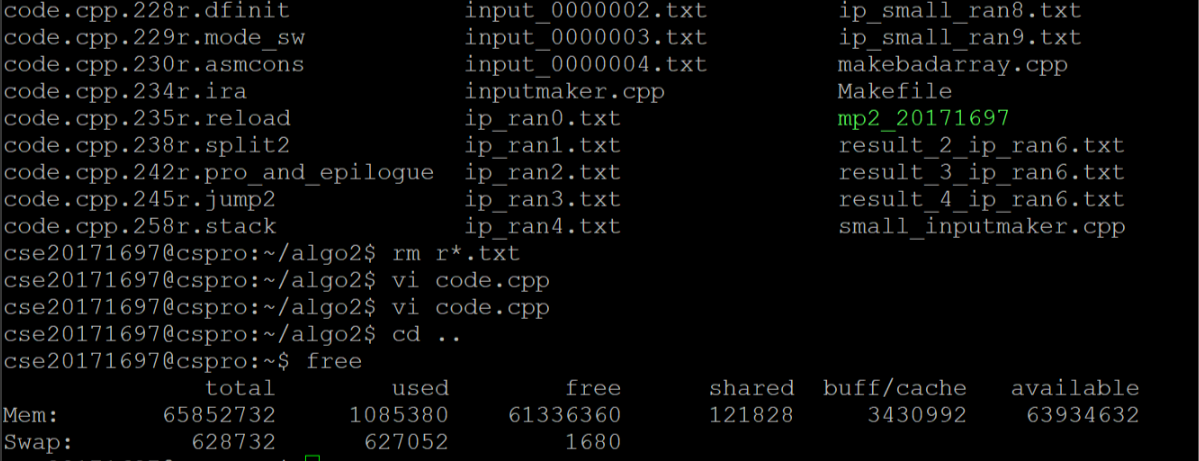
위 그림은 List에 대해 intro\_sort를 수행하는 방식을 표현한 것이다. quick sort 구현 결과, sublist A를 하나로 보면 이미 A의 위치는 정해져있는 것이다. 따라서 정해놓은 depth에 이르러 heap sort가 시작될 때, 이 A에 대해서만 정렬을 수행하도록 코드를 짰다. 다시 말해 인자로 A의 시작인덱스와 끝인덱스를 보내 A에 대해서만 max\_heap을 만들어 정렬하도록 구현하였다. 이 때 base = ( 시작인덱스 ) -1 로 설정해주고 base + (기존의 Heap sort에서의 인덱스 : 전체의 heap의 root가 인덱스 1부터 시작해서 (parent의 인덱스) = (child의 인덱스)/2를 만족시키는 인덱스 )를 이용하면 기존의 Heap Sort 방식을 그대로 적용할 수 있었다. 그리고 quick sort를 구현하는 과정에서는 pivot을 median of 3 방법을 이용하였다. 이를 설명하자면, 살펴보고 있는 숫자열의 첫번 째 인덱스 값, 마지막 인덱스 값, 중간 인덱스 값을 살펴 ( 첫번 째 인덱스 값 ) <= ( 중간 인덱스 값 ) <= ( 마지막 인덱스 값 )이 되도록 swap하고, pivot을 중간값으로 정하는 방법이다. 이는 살펴보고 있는 리스트의 중간 값으로 pivot이 정해져 최대한 partition이 1/2로 되도록 하기 위함이다.

그러나 한계로는 이미 non-increasing으로 정렬되어있는 경우나 non-decreasing으로 정렬되어 있는 경우에 대해서는 불필요한 수행을 하게된다는 점이 있다. optimal한 정렬 알고리즘을 찾다가 Tim Sort의 방법을 보고 여기서 run을 만드는 방법의 일부를 intro\_sort 시작하기 전에 수행하게 하였다. 즉 전체 List를 살펴보면서 이미 non-increasing 또는 non-decreasing으로 정렬되어있는 sublist를 찾는다. 이 때 non-decreasing일 경우 swap을 이용해 전체적으로 sublist를 뒤집어 non-incresing이 되도록 한다. 이렇게 non-increasing sublist를 만들면서 개수를 세서 이 개수가 1이면 더이상의 정렬 알고리즘을 수행시키지 않고 끝내도록 하였다. 그렇게 O(n)에 가까운 수행으로 마칠 수 있도록 design하였다.

또한 실험을 해본 결과 input 크기가 작은 경우에는 heap sort보다 느리다는 점도 발견할 수 있었다. 그래서 전체 개수가 250 미만이면서 이미 non-increasing 또는 non-decreasing으로 정렬되어 있는 경우가 아니라면 전체적으로 heap sort를 수행하도록 하였다.

* Experiment environment
* PuTTY

64-bit Windows



* Desktop

intel(R) Core(TM) i5-6300U CPU @ 2.40GHz 2.50GHz

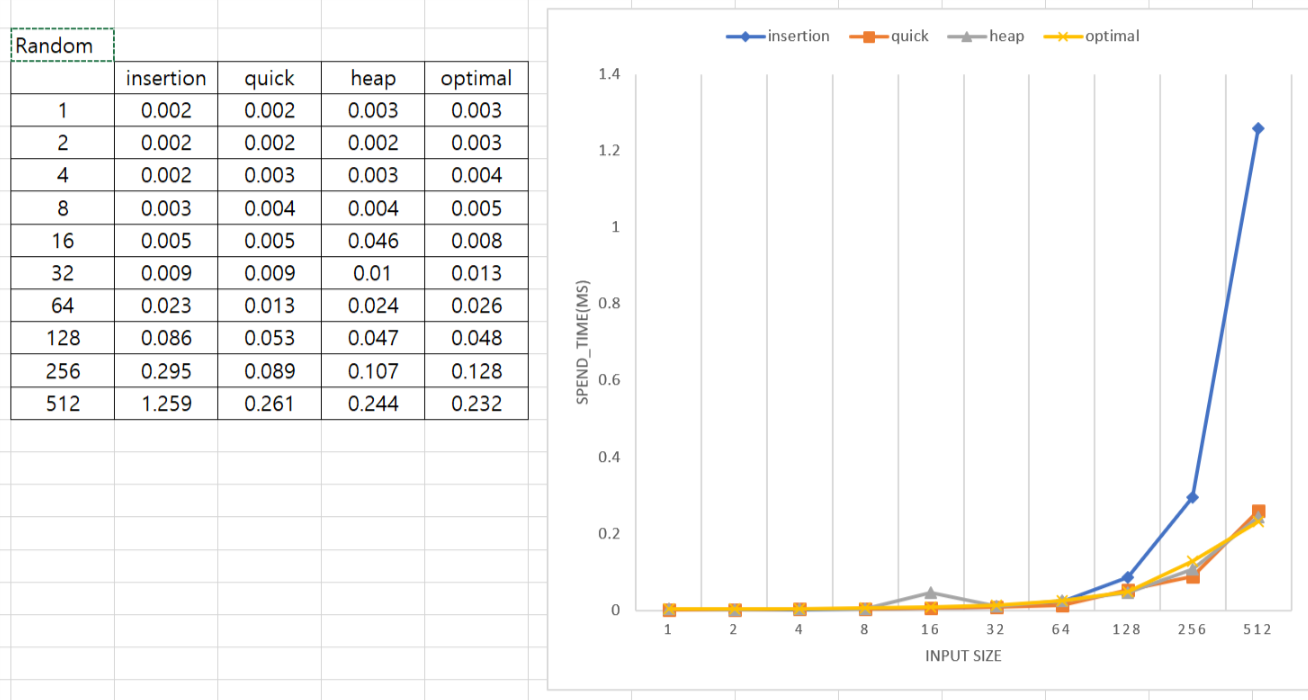
8GB RAM

* Experiment Set up

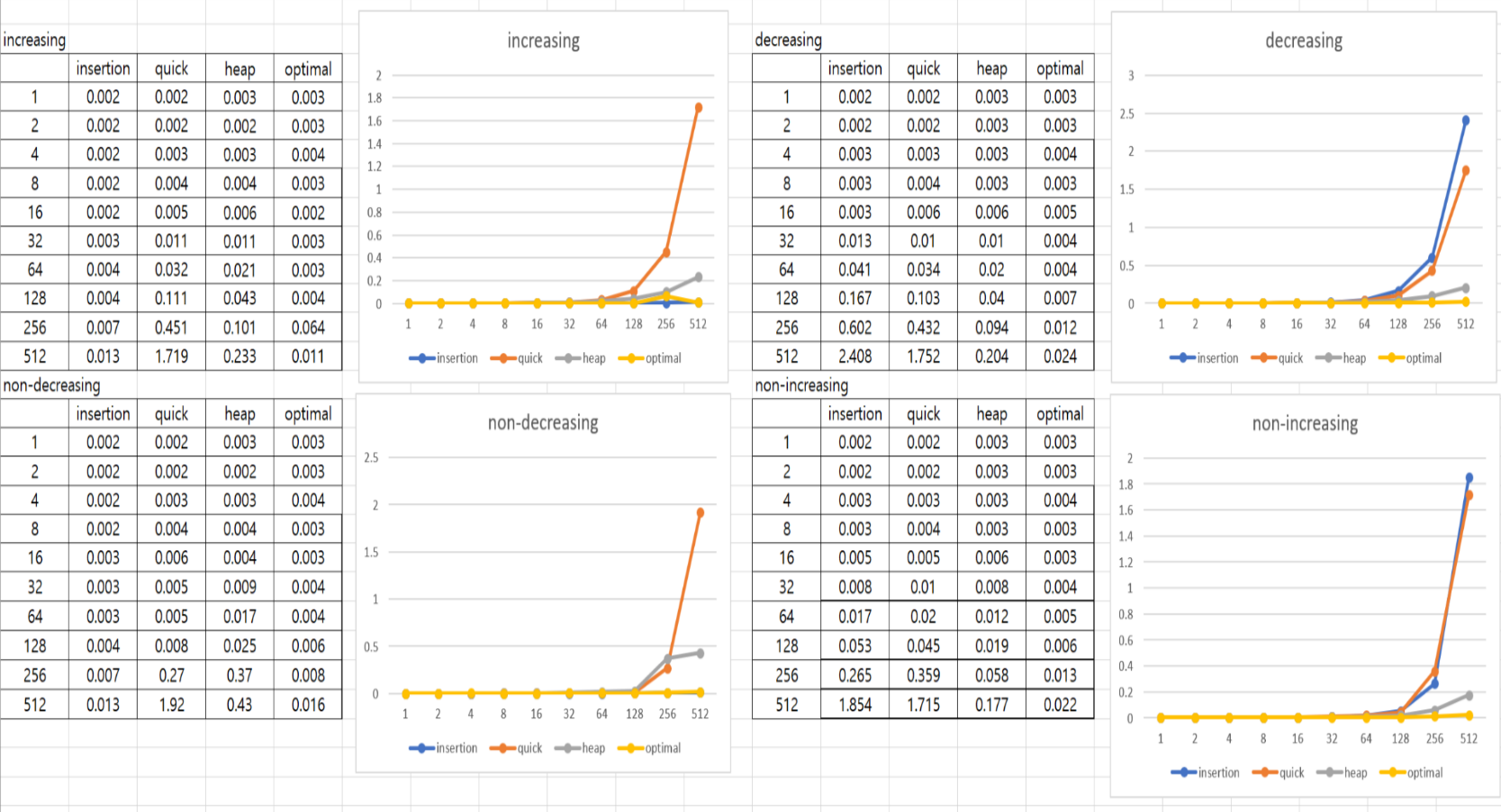
input의 크기가 커짐에 따른 수행시간(ms)를 살펴보았으며 Input 숫자열이 random한 경우, increasing인 경우, decreasing인 경우, non-decreasing인 경우, non-increasing인 경우로 나누어서 살펴보았다. input 크기는 1부터 222까지 살펴보았다.

* Comments

1. 작은 크기의 Input 경우 (1 ~ 29)

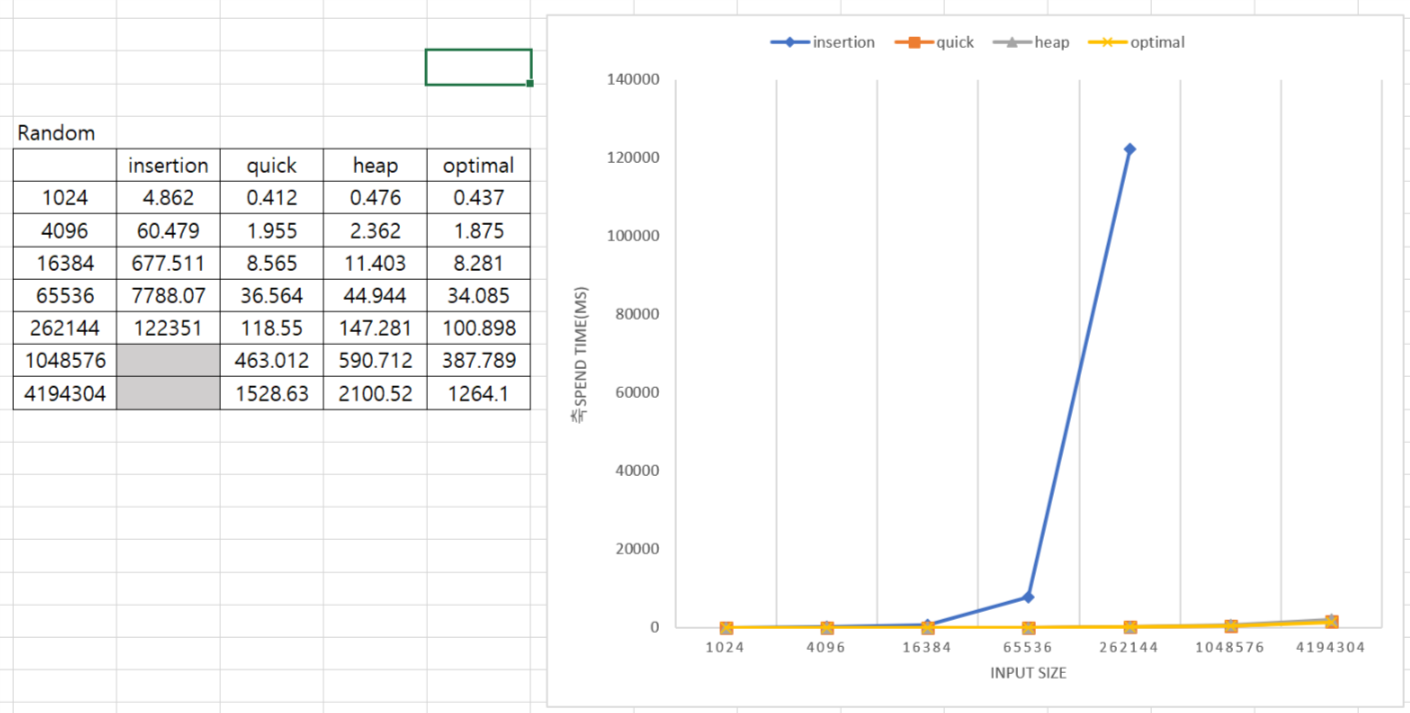
* Random input

Random input의 경우 insertion sort만 급격하게 증가하는 것을 볼 수 있었고 나머지 알고리즘에 대한 차이는 거의 알 수 없었다. 그리고 같은 크기일 때 quick sort와 heap sort, optimal sort의 수행시간 크기 순이 계속 달라졌다. 표를 살펴보면, insertion sort는 O(n2)으로 n이 2배씩 증가할 때마다 수행시간이 4배 가까이 증가함을 살펴볼 수 있었고 나머지 알고리즘은 대부분 O(nlogn)에 따라 2배 조금 더 넘게 증가함을 살펴볼 수 있었다.

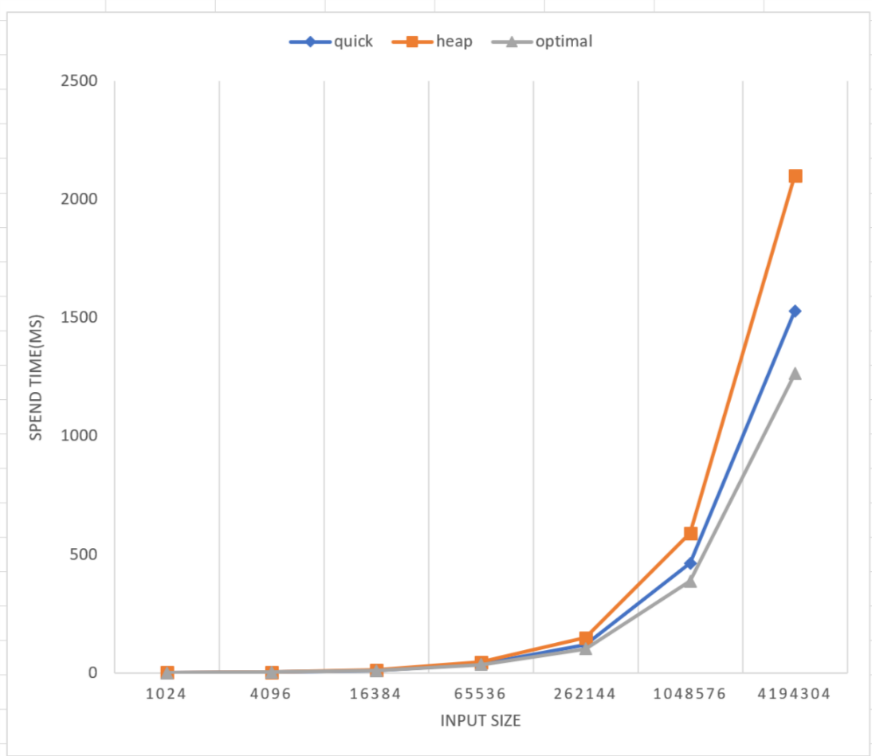
* increasing / non-decreasing / decreasing / non-decreasing input

insertion sort가 증가하는 방식으로 정렬된 input의 경우 O(n)이고 감소하는 방식으로 정렬된 input의 경우 O(n2)임을 그래프에서 확인할 수 있었다. optimal이 증가하는 방식의 insertion sort와 같은 형태를 띠고 있는 것으로보아 O(n)임을 예상할 수 있다. 즉, 의도대로 이미 정렬되어있는 숫자열에 대해서는 더이상의 수행없이 끝냈음을 확인할 수 있다. 그리고 quicksort의 특징대로 이러한 input의 경우 O(n2)에 가깝게 이루어졌음을 확인할 수도 있다. 그리고 heapsort는 이러한 input의 경우에도 quick과 optimal 사이에 있는 것으로보아 O(nlogn)의 특징을 보이고 있음도 알 수 있다.

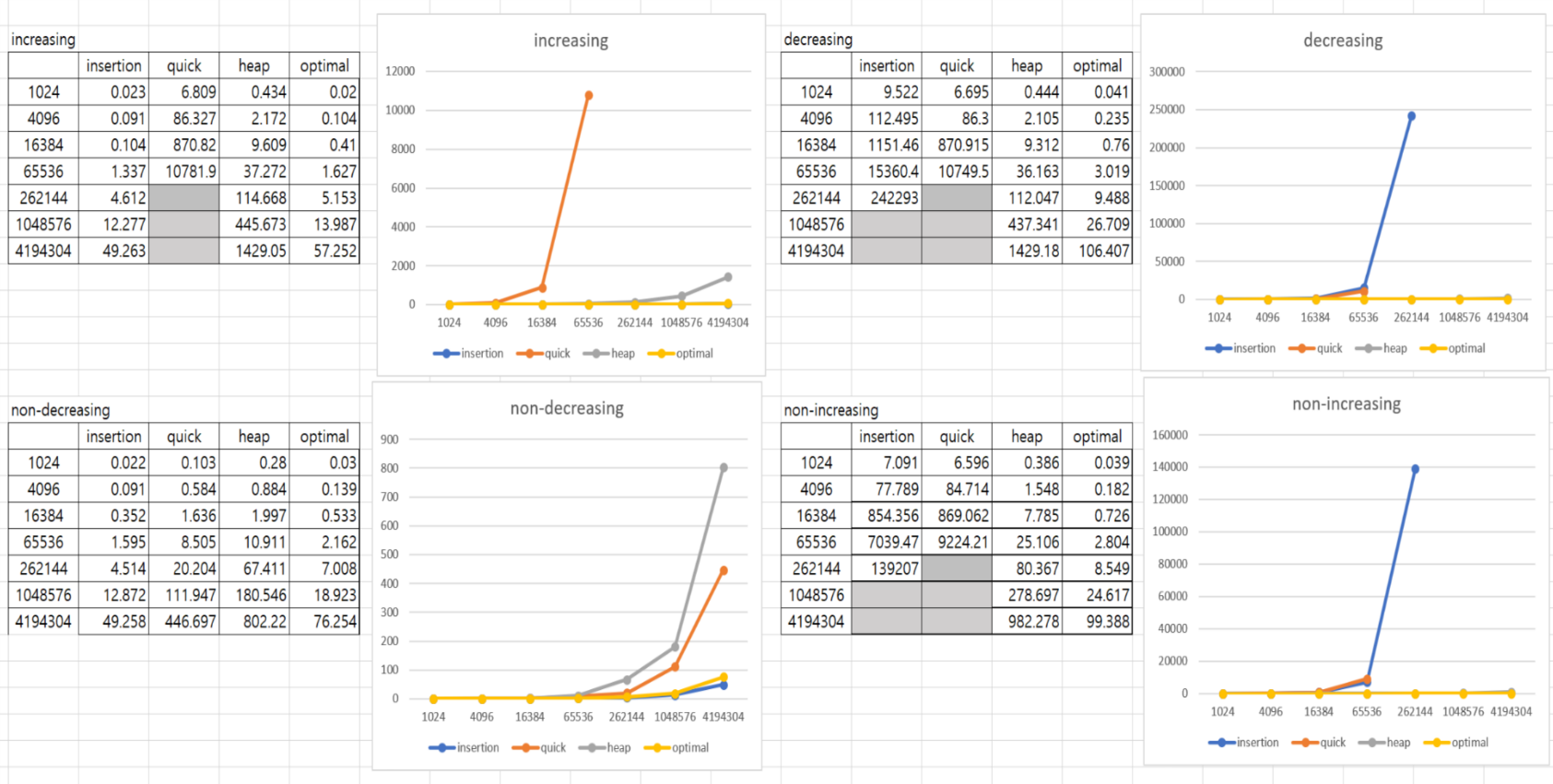
1. 큰 크기의 Input 경우 (210 ~ 222)

* Random input

insertion sort가 다른 알고리즘에 비해 급격하게 증가함을 알 수 있다. 형태를 대충보면 insertion sort는 n2꼴을, 나머지는 nlogn꼴을 보인다고 어림잡아 생각해볼 수 있다. 그리고 크기가 큰 input에서도 insertion은 O(n2)에 따라 n이 2배 증가할 때 수행시간이 4배 가까이 증가하고 나머지 알고리즘은 O(nlogn)에 따라 n이 2배 증가할 때 수행시간이 2배 조금 넘게 증가함을 확인할 수 있다.

 insertion sort 제외한 나머지 알고리즘을 비교해보았다. 그 결과 input 크기가 커질 수록 optimal( algorithm4 )가 가장 빠르게 수행되었음을 알 수 있다.

Heap sort가 가장 느린 이유는 quick sort 방법에 비해 숫자열에 있는 숫자 하나의 움직이는 보폭이 넓으며 메모리 할당이 필요하고, max heap을 구현하여 부분부분 decreasing하게 정렬한 뒤 재정렬을 하는 방식으로 시간이 더 들일 것으로 생각된다. 그리고 quick sort가 빠르게 돌아간 것으로 보아 pivot 결정이 잘 되어 partition이 1/2 r가깝게 될 수 있었던 input이었던 것으로 생각된다.

* increasing / non-decreasing / decreasing / non-decreasing input

크기가 큰 경우에도 quick sort는 이러한 input에 대해 가장 worst한 결과를 보였다. 그래프를 보아도 O(n2)에 가까운 결과를 보임을 알 수 있고 표로 살펴보아도 n이 2배 증가할 때 4배로 증가함을 확인할 수 있다. 그리고 non-decreasing 경우에만 heap sort가 가장 worst했는 데 increasing 결과와 다른 것으로 보아 같은 원소가 많을 경우 quick sort는 빨라지지만 heap sort는 느려진다고 가정해볼 수도 있을 것 같다.

1. 한계: 시간만을 고려해서 메모리 사용 정도는 고려하지 않았다는 한계가 있었고, algorithm 4가 input 크기가 큰 경우에는 가장 적은 수행시간을 보였으나 input 크기가 작은 경우 가장 오래 걸리기도 했다. 그 이유는 가장 복잡하게 짜여져 부르는 함수도 많고 정렬하기 전에 non-decreasing한 sublist를 만드는 게 비효율적으로 작용했을 것 같다.
2. 느낀점: 같은 O(nlogn)에서도 메모리 이용 방법, 함수 호출 방법 등에 따라 차이가 날 수 있음을 알 수 있었고 각 알고리즘마다의 가장 느린 결과와 빠른 결과를 내는 input의 특성을 알아볼 수도 있었다.