알고리즘 설계와 분석

(Design and Analysis of Algorithms)

Machine Problem 2

20161603
신민준

Table of Contents

1 Implementations

1.1 Algorithm 4(Introsort) Implementation

```
/**
 * Algorithm 4 Implementation.
 * Basically a quicksort algorithm with tail recursion optimising.
 * Uses insertion sort if current list size is less than certain number.
 * Performs heapsort if total depth of quicksort tree is below a certain
 * level, to improve execution time.
 * @param data  Integer list to sort.
 * @param from  Left index of specified list. Inclusive.
 * @param to    Right index of specified list. Inclusive.
 * @param depth Maximum depth for this function to recursively call itself.
 */
void intro_sort(int* data, int from, int to, int depth) {
   int pivot;
   int from2, to2;
   int* m;
   from2 = from;
   to2 = to;
   // improving performance using tail recursion optimisation
   while(to2 > from2) {
      if(to2-from2+1 <= 16) {
         // use insertion sort when size <= 16 elements
         insertion_sort(data+from2, to2-from2+1);
         return;
      }
      if(!depth) {
         // using heapsort when depth reached its limit
         heap_sort(&data[from2], to2-from2+1);
         return;
      }
      depth--;
      // use median-of-three method to get the value to use as pivot.
      m = median3(&data[from2], &data[(from2+to2)/2], &data[to2]);
      SWAP(*m, data[to2], int);
      pivot = partition(data, from2, to2);

      // only the smallest of the two divided list is recursively called.
      if(pivot < (from2+to2)/2) {
         intro_sort(data, from2, pivot-1, depth);
         from2 = pivot+1;
      } else {
         intro_sort(data, pivot+1, to2, depth);
         to2 = pivot-1;
```

```
        }
    }
}
```

Code 1

이 Sorting Algorithm을 구현하기 위해, 기본적으로 Quicksort를 구현하되, 몇 가지 최적화 방법을 적용해 더 효율적인 알고리즘이 되도록 했습니다.

먼저, Tail Recursion Optimising을 적용해 각 level에서 딱 한번의 recursive call을 수행하도록 구현했습니다. 이 때, recursive call이 실행되는 배열은 pivot을 기준으로 나뉜 두 배열 중 더 길이가 짧은 배열입니다. 기존의 Quicksort에서는 두번의 recursive call을 반드시 실행했기에, 이 방법을 사용하면 총 function call의 갯수를 최소화하여 실행 시간을 줄일 수 있습니다.

두번째로, Quicksort의 pivot을 정할 때, 배열의 첫 원소, 중간 원소, 그리고 마지막 원소 중 중간값을 가지는 원소를 pivot으로 정해, 특정 worst-case input에 대해 대처할 수 있도록 최적화했습니다.

다음 최적화 방법으로, Quicksort의 recursive call이 특정한 depth를 초과하게 되면, 남은 list를 heapsort를 사용해 정렬하도록 했고, 만일 input의 크기가 16 이하인 경우, insertion sort로 해당 배열을 정렬하게 구현했습니다. 이 방법을 사용하면 recursive call의 기하급수적인 증가를 완화할 수 있고, 특정 size 이하의 경우 insertion sort가 quicksort보다 실증적으로 더 빠르기에 결과적으로 계산 속도를 개선할 것입니다.

제가 소개한 최적화 방법은 각자 Linear한 시간 복잡도, 즉 $O(1)$의 time complexity를 지니고, max depth를 $2\lceil \log_2 n \rceil$로 설정해놓았기에, worst time complexity와 average time complexity는 둘 다 $O(n \log n)$의 time complexity를 가집니다.

2 Experiment

2.1 Environment Specifications

MacBook Pro (13-inch, 2018)

OS - macOS Catalina version 10.15

CPU - 2.3 GHz Intel Core i5

RAM - 8GB 2133 MHz LPDDR3

Graphics - Intel Iris Plus Graphics 655 1538MB

Shell - zsh 5.7.1 (x86_64-apple-darwin18.2.0)

Compiler - Apple clang version 11.0.0 (clang-1100.0.33.8)

2.2 Experiment Setup

Makefile의 경우, -O0 옵션을 사용해 컴파일 시의 컴파일러 최적화를 제외했습니다. Makefile의 내용은 다음과 같습니다.

```
# Flags: C11 standard & no compiler optimising
CFLAGS += -std=c11 -O0
# Libmath
LIBS += -lm

# making the executable from object file
mp2_20161603: mp2_20161603.o
   $(CC) $(CFLAGS) mp2_20161603.o -o mp2_20161603 $(LIBS)

# an object file is made from one source code
mp2_20161603.o: mp2_20161603.c
   $(CC) $(CFLAGS) -c mp2_20161603.c

clean:
   rm mp2_20161603 mp2_20161603.o
```

Figure 1 Makefile

실험에 사용할 input을 생성하는 makein 프로그램을 구현했습니다. 해당 코드와 input을 생성한 터미널 명령어는 다음과 같습니다.

```
int main(int argc, const char* argv[]) {
   FILE* fp;
   if(argc < 6) {
      fprintf(stderr, "Usage: %s filename num min max sorted [seed]\n", argv[0]);
      exit(1);
   }
   if(!(fp = fopen(argv[1], "w"))) {
      fprintf(stderr, "Error while opening file %s\n", argv[0]);
      exit(1);
```

```
    }
    int num = atoi(argv[2]);
    int min = atoi(argv[3]);
    int max = atoi(argv[4]);
    int* data;
    data = malloc(sizeof(int)*num);
    srand(atoi(argv[6]));
    for(int i=0 ; i<num ; i++) {
        data[i] = rand()%(max-min)+min;
    }

    if(atoi(argv[5])) {
        quick_sort(data, 0, num-1);
    }

    fprintf(fp, "%d ", num);
    for(int i=0 ; i<num-1 ; i++) {
        fprintf(fp, "%d ", data[i]);
    }
    fprintf(fp, "%d", data[num-1]);
    fclose(fp);

    return 0;
}
```

Code 2

```
: 1572507275:0;./makein 10.txt 10 -10000000 1000000 0 214748
: 1572507791:0;./makein 100.txt 100 -10000000 1000000 0 214748
: 1572507795:0;./makein 1000.txt 1000 -10000000 1000000 0 214748
: 1572507799:0;./makein 10000.txt 10000 -10000000 1000000 0 214748
: 1572507802:0;./makein 100000.txt 100000 -10000000 1000000 0 214748
: 1572507805:0;./makein 1000000.txt 1000000 -10000000 1000000 0 214748
: 1572507809:0;./makein 10000000.txt 10000000 -10000000 1000000 0 214748
: 1572507819:0;./makein 100000000.txt 100000000 -10000000 1000000 0 214748
: 1572702065:0;./makein 10.txt 10 -10000000 10000000 1 1337
: 1572702069:0;./makein 100.txt 100 -10000000 10000000 1 1337
: 1572702073:0;./makein 1000.txt 1000 -10000000 10000000 1 1337
: 1572702078:0;./makein 10000.txt 10000 -10000000 10000000 1 1337
: 1572702082:0;./makein 100000.txt 100000 -10000000 10000000 1 1337
: 1572702087:0;./makein 1000000.txt 1000000 -10000000 10000000 1 1337
: 1572702091:0;./makein 10000000.txt 10000000 -10000000 10000000 1 1337
: 1572702100:0;./makein 100000000.txt 100000000 -10000000 10000000 1 1337
```

Figure 2 ~/.zsh_history

[Figure 2]에서 보인 것과 같은 명령어에서 seed 값만 다르게 한 input 파일을 3번 생성해 각 input에 대해 실험을 진행했습니다.

2.3 Result Data

3번씩 진행한 각 실험에서 얻은 결과 값들의 평균으로 데이터를 도출했습니다.

모든 값들의 단위는 초(sec)입니다.

n	10	100	1000	10000	100000	1000000	10000000	100000000
Algorithm1	0.000005	0.000011	0.001095	0.062495	5.873900	578.535283		
Algorithm2	0.000010	0.000012	0.000123	0.001026	0.013155	0.149193	1.693016	19.927453
Algorithm3	0.000005	0.000013	0.000166	0.001713	0.017861	0.227917	3.639529	61.672647
Algorithm4	0.000011	0.000022	0.000117	0.001989	0.012797	0.139856	1.613722	18.649106

Table 1 Random Data

n	10	100	1000	10000	100000	1000000	10000000	100000000
Algorithm1	0.000005	0.000018	0.001452	0.121972	11.777704			
Algorithm2	0.000005	0.000025	0.001858	0.147371	14.105183			
Algorithm3	0.000004	0.000012	0.000113	0.001052	0.012521	0.130528	1.582163	17.845189
Algorithm4	0.000009	0.000019	0.000093	0.000900	0.012103	0.122697	1.448844	14.408899

Table 2 Non-increasing Data

표의 비어져 있는 부분은 실행 시간이 너무 오래 걸리므로 측정이 불가능했습니다.

2.4 Data Analysis

2.4.1 Random Data

Random Data를 input으로 사용한 [Table 1]의 자료를 그래프로 표현했을 때 다음과 같습니다.

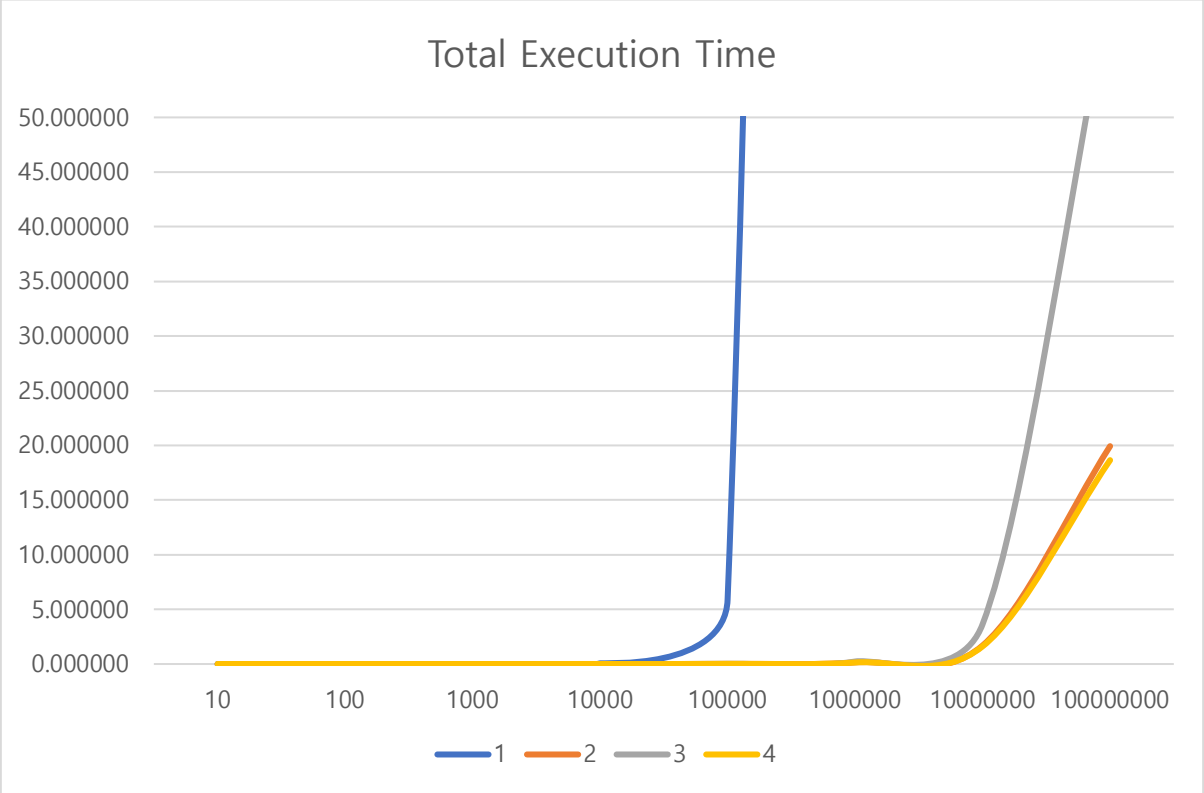

Figure 3

X축: n, Y축: execution time in seconds

[Figure 3] 의 Y-value를 최대 50s까지만 표시한 이유는, Algorithm1의 증가폭이 너무나도 커 다른 세 알고리즘들의 상관관계를 명확하게 보이기 힘들었기 때문입니다. [Figure 3]의 Y 축에 로그를 사용해 새롭게 그래프를 그리면 다음 [Figure 4]와 같습니다.

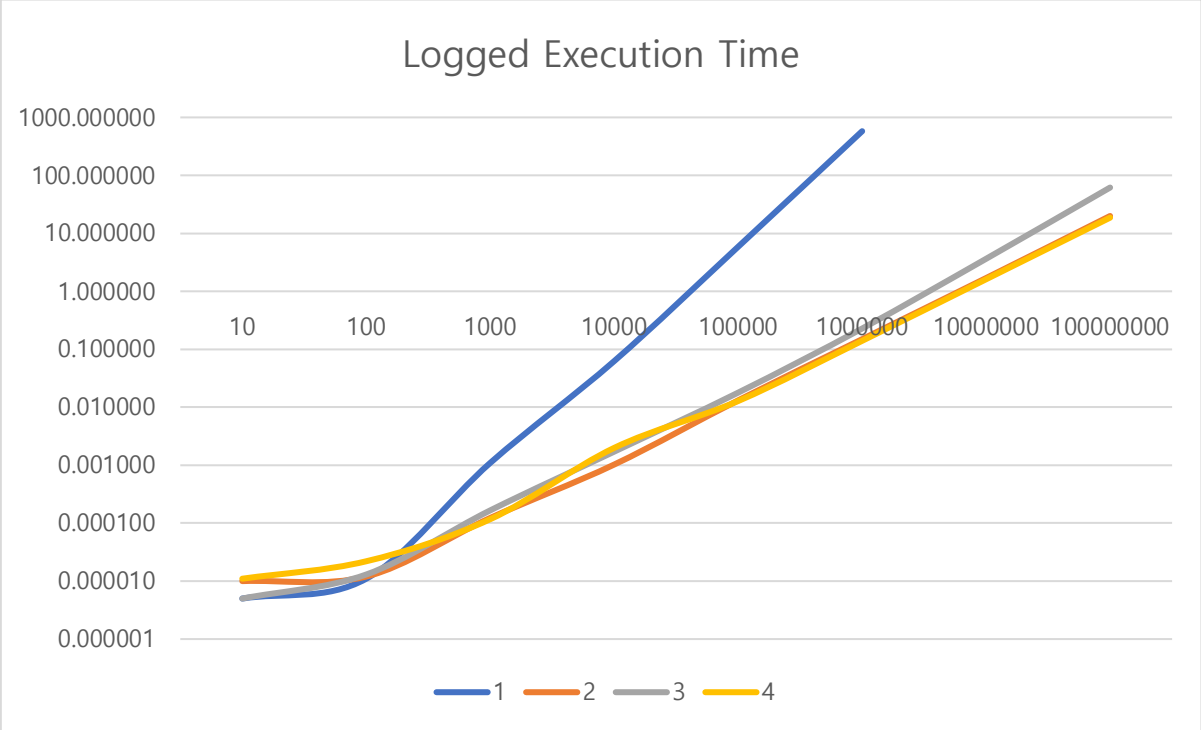

Figure 4

X축: n의 값, Y축: 로그 시간

두 그래프 [Figure 3]과 [Figure 4]로부터 네 알고리즘에서 입력 값의 개수에 따른 실행 시간 증가 추세를 확인할 수 있습니다. 특히, Algorithm1, 즉 Selection sort의 시간복잡도가 다른 세 알고리즘에 비해 매우 크다는 것을 확인할 수 있는데, 이는 실제로 Selection sort가 $O(n^2)$의 average time complexity를 가지고, 다른 세 알고리즘의 average time complexity는 $O(n \log n)$이라는 점을 고려한다면 매우 자연스러운 결과입니다.

Heapsort의 경우, 같은 시간복잡도를 가지는 Quicksort와 Introsort보다 실질적인 실행 시간은 더 큰 것으로 나타났는데, 이는 Heap을 생성하고 값을 정렬하는 데 사용되는 adjust() 함수의 실질적인 cost가 크기 때문인 것으로 보입니다. 비록 같은 시간복잡도를 가지더라도 실질적인 실행 시간에서 차이가 났다는 점이 주목할 만합니다.

Quicksort와 Introsort의 차이는 그래프 상으로 보면 미미하지만 데이터의 크기가 커짐에 따라 Introsort가 Quicksort보다 점점 더 빨리 계산 결과를 내놓는다는 것을 확인할 수 있습니다. 따라서 quicksort를 개량해서 직접 introsort를 구현하는 과정에서 생각한 최적화 방법들이 성공적이었다고 볼 수 있습니다.

위 분석을 토대로, Random Input에 대한 네 알고리즘의 실행 시간을 간단히 비교하면 다음과 같습니다.

$$T_{ins} \ll T_{heap} < T_{quick} < T_{intro}$$

2.4.2 Non-increasing Data

Non-increasing data를 input으로 사용한 [Table 2]의 값을 그래프로 나타내게 되면 다음과 같은 모습을 보입니다.

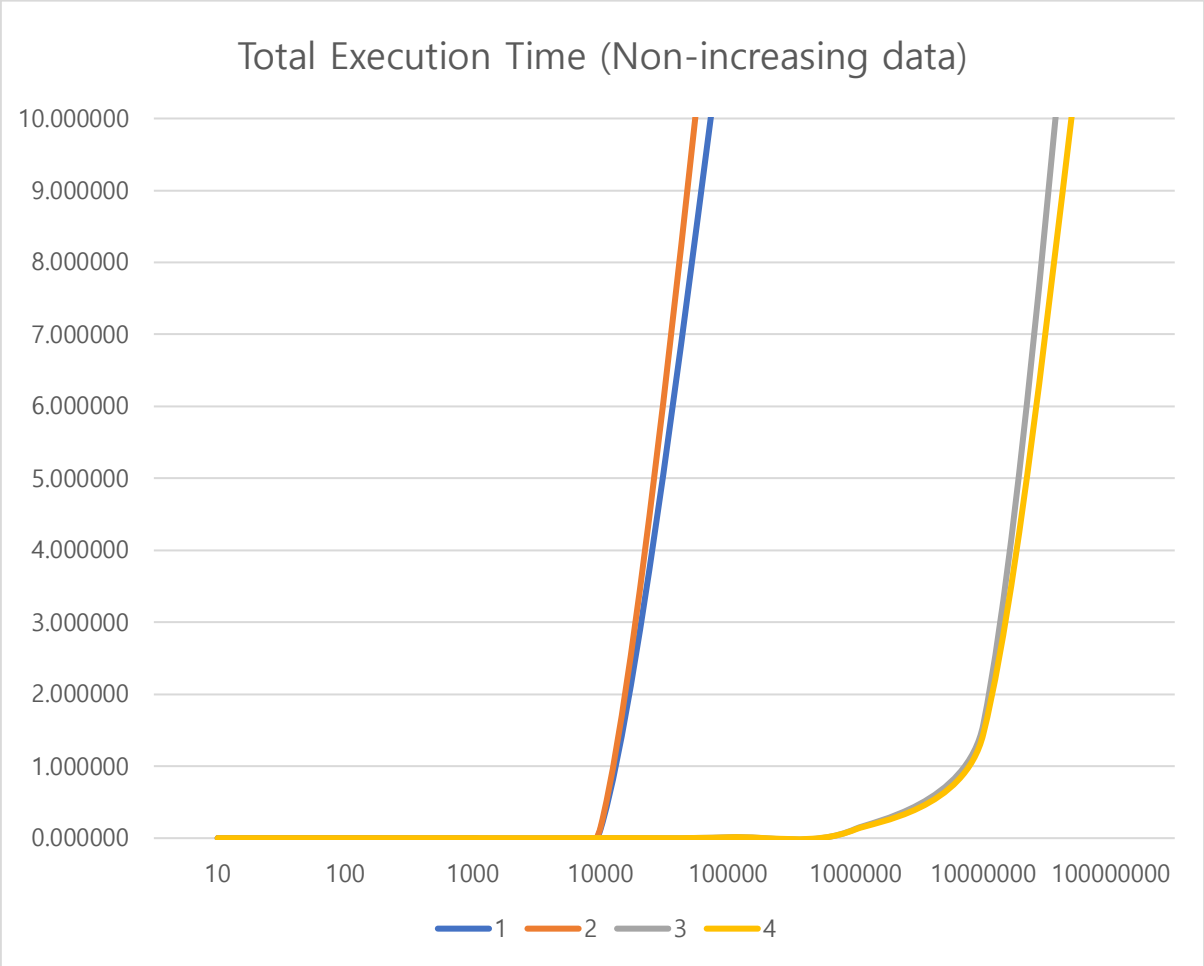

Figure 5

X축: n, Y축: execution time in seconds

이 경우에도 마찬가지로, 네 알고리즘의 상관관계를 보일 수 있도록 그래프에서 Y-value 최댓값을 10으로 제한해 그래프를 도출했습니다. [Figure 5]의 Y-value에 로그를 사용해 그래프를 그리면 다음과 같은 그래프가 나옵니다.

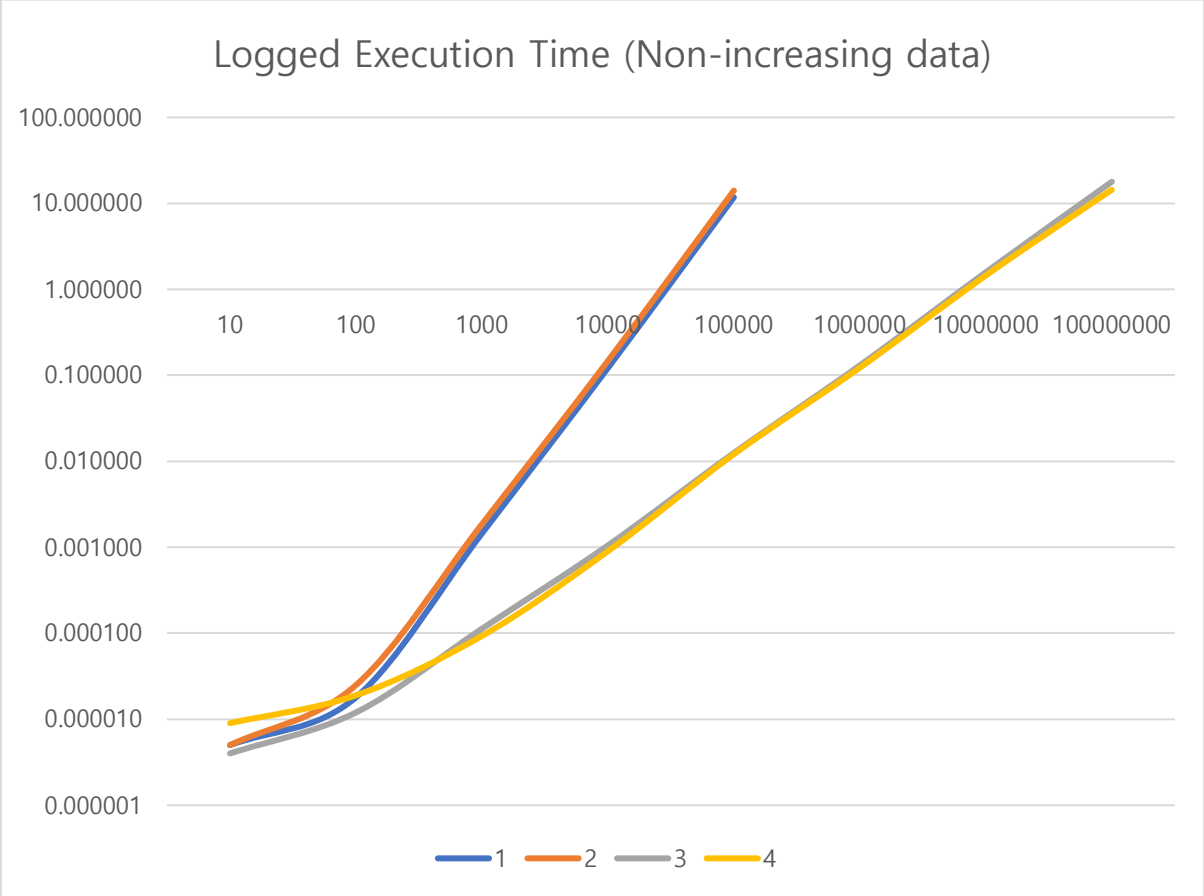

Figure 6

[Figure 5]에서, random data를 사용한 [Figure 3]의 결과와는 달리 quicksort의 실행 시간이 insertion sort보다도 약간 더 느린 모습을 확인할 수 있습니다. 이는 quicksort에서 사용한 partition() 함수의 구현 방법에 의해, 이 quicksort 알고리즘 non-increasing data가 worst case이며, 이 때의 worst time complexity는 $O(n^2)$이므로 이와 같은 결과가 나온 것이라 볼 수 있습니다.

반면, 구현한 introsort는 median-of-three 방식으로 pivot을 정하기에 같은 방법으로 worst case가 나오지 않으며, 또한 그 외의 최적화 방법을 추가적으로 구현해두었기에 여전히 네 알고리즘 중에서 가장 시간이 적게 걸린다는 것을 확인할 수 있습니다.

위 분석에서 살펴본 각 알고리즘들의 non-increasing data에 대한 time complexity를 비교하면 다음과 같습니다.

$$T_{quick} < T_{ins} \ll T_{heap} < T_{intro}$$

3 Additional Comments

- Quicksort 의 worst time complexity 는 $O(n^2)$로, Heapsort 의 $O(n \log n)$의 worst time complexity 보다 큰 값을 갖고 두 알고리즘의 average time complexity 는 $O(n \log n)$으로 같지만, 실험 결과 일반적인 무작위 데이터가 주어진 환경에서 heapsort 보다 quicksort 알고리즘이 더 빠른 실행 시간을 보였습니다. 따라서 asymptotic notation 이 실제 구현에서의 실행 속도와 반드시 일치하지는 않을 수도 있음을 확인할 수 있었습니다.
- 하지만, 직접 최적화를 통해 구현한 Introspective Sort 를 만들면서, Time complexity 에 영향을 주지 않는 최적화 방법을 사용한 결과 실제로 실행 시간이 유의미하게 개선되는 것이 확인됩니다. 따라서, asymptotic notation 을 고려하면서 알고리즘을 설계하는 것이 절대 무의미한 것이 아니라고 볼 수 있습니다.