2018년 2학기 알고리즘 설계와 분석
CSE3081-01반 MP2 과제 보고서

학번: 20171665
이름: 이 선 호

2018. 11. 19

목 차

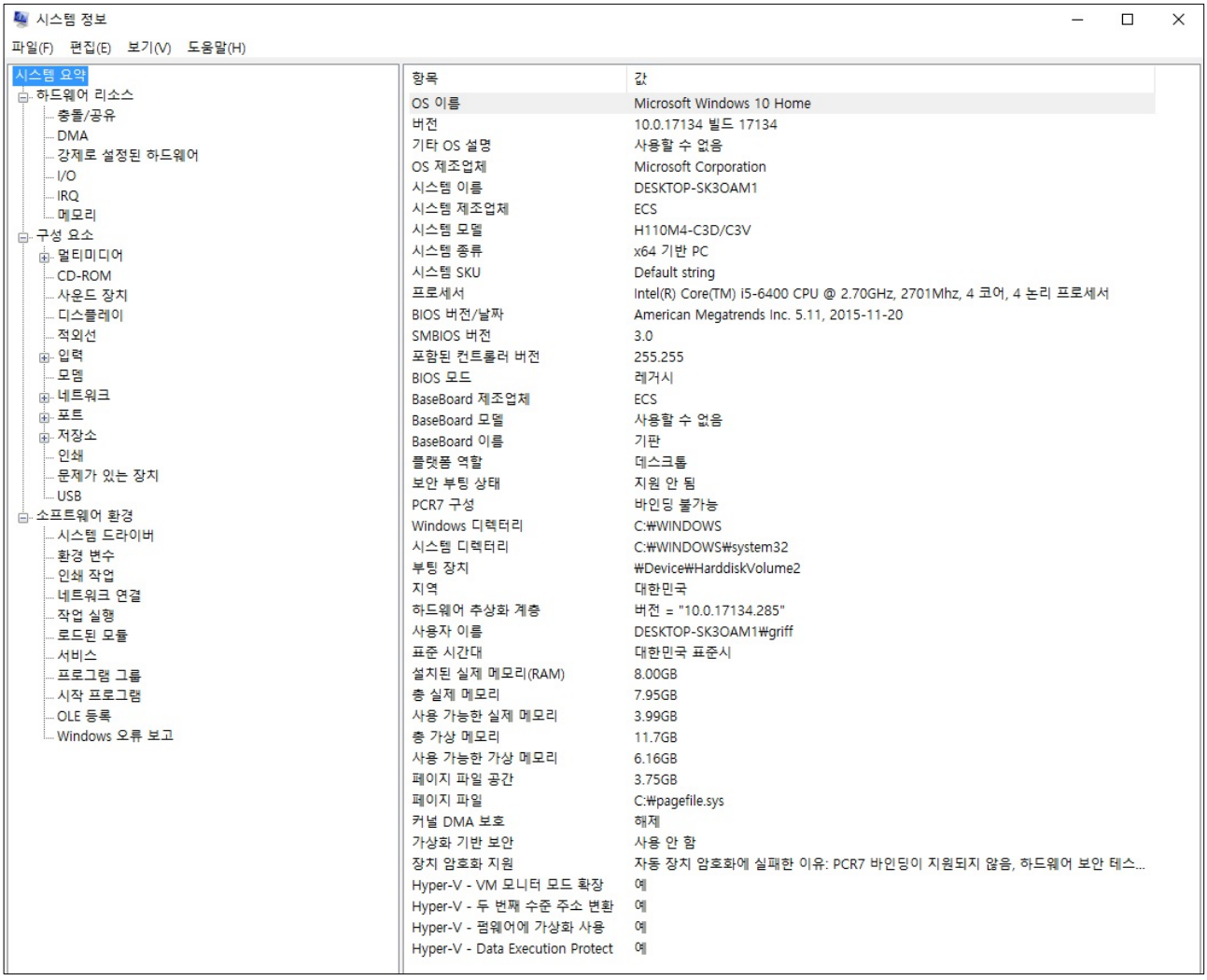
Ⅰ 실험 계획

1. 실험 목적

이번 실험에서는 다양한 정렬 알고리즘의 성능을 비교하며 이해하고, 가장 효율적인 알고리즘을 구현하는 것을 목표로 한다.

2. 실험 환경

임의의 integer type의 정수들로 나열된 list를 정렬하는 알고리즘 실험을 진행할 PC의 시스템과 하드웨어 환경을 조사했다.



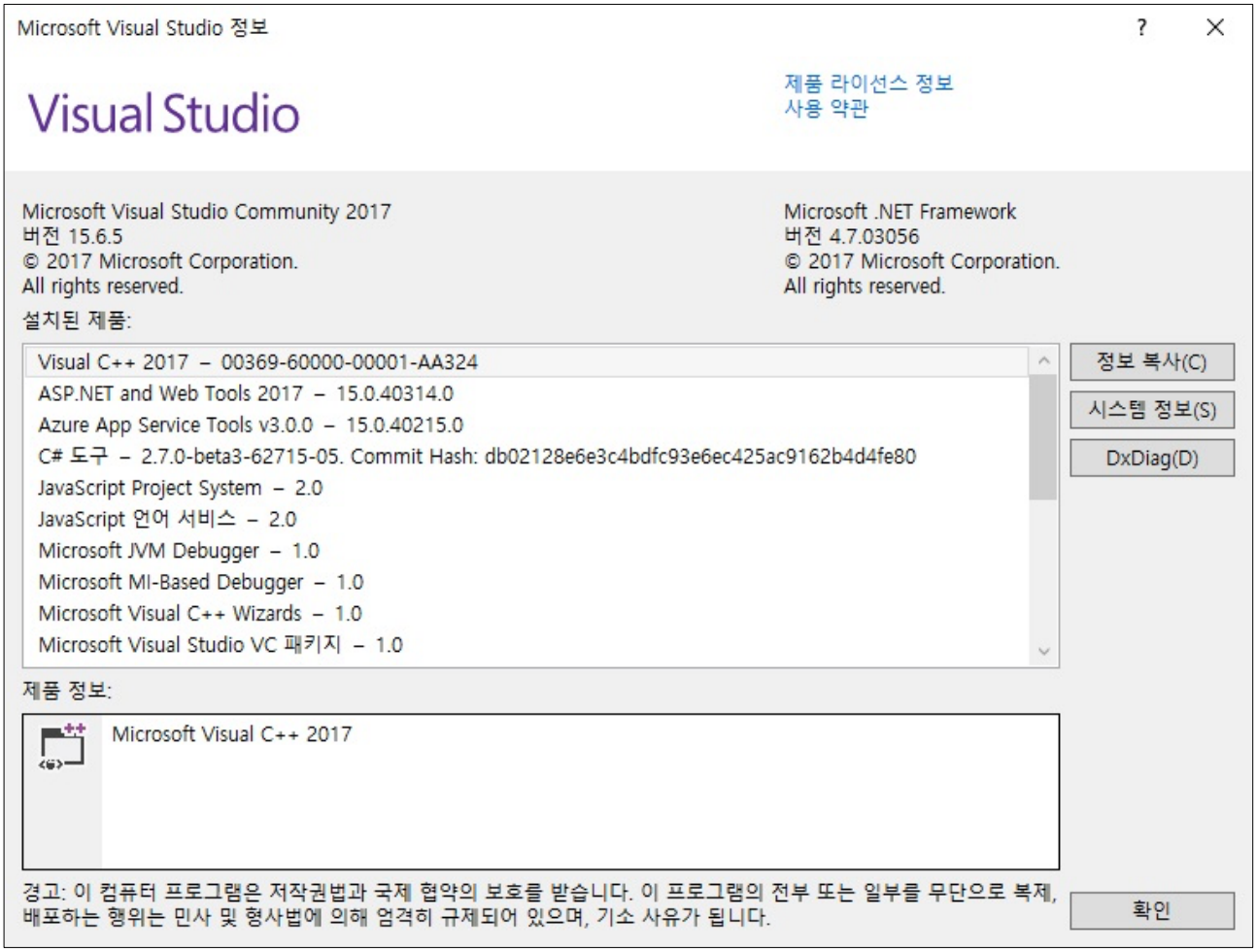
[그림 Ⅰ-1] PC 시스템 환경과 하드웨어 정보

위의 정보를 간략히 요약하자면 *[표 Ⅰ-1]*의 내용과 같다.

항목	값
OS 이름	Microsoft Windows 10 Home
버전	10.0.17134 빌드 17134
시스템 종류	64bit 운영체제, x64 기반 PC
프로세서	Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz, 2701Mhz, 4코어, 4논리 프로세서
설치 메모리	8.00GB

[표 Ⅰ-1] 시스템과 하드웨어 환경 사양 요약

소스 코드를 컴파일(Compile)하여 실행할 프로그램은 Microsoft에서 제작한 Visual Studio 2017이며, 이와 관련한 자세한 정보는 *[그림 Ⅰ-2]*의 내용과 같다.



[그림 Ⅰ-2] Visual Studio 환경 정보

2. 실험 전제

- 소스 코드는 C언어로 작성했으며, 하나의 파일로만 만들었다. Visual Studio 또는 Linux(Unix)에서 기본적으로 제공하는 라이브러리인 stdio.h, stdlib.h, string.h, time.h, 그리고 math.h를 이용했다.
- 메모리 낭비를 없애기 위해 소스 코드에서 input으로 받는 정보는 malloc 함수를 사용하여 동적으로 메모리를 할당했다.
- Linux 환경에서 파일 명령어와 함께 필요한 두 인자를 사용자가 입력해 main 함수에서 넘겨받을 수 있도록 변수 argc와 argv를 사용했다. Windows에서는 command로 직접 인수를 입력하는 것이 불가능하므로 디버깅 속성 설정에서 전달하는 인자를 입력하여 적용했다.
- 최대로 입력받을 수 있는 list의 길이는 실험에서 안내한 2^{30}을 초과하지 않기 위해 동적 할당으로 최대 $2^{31}-1$까지 가능하다. list의 각 element는 integer data type으로 입력받으며, element의 value 범위는 $-2^{31} \sim 2^{31}-1$까지 가능하다.
- 4개의 알고리즘은 각각의 함수로 별도로 만들어서 실행하며 함수마다 list와 list의 길이 또는 list에서 가장 왼쪽과 오른쪽에 있는 index를 파라미터로 받는다. 각 알고리즘의 중심 함수들은 return으로 반환되는 data type을 void로 설정했으며, 알고리즘 내 일부 서브 함수들은 필요에 따라 int 등 그에 맞는 data type을 반환한다.
- 알고리즘 실행 시간을 측정하기 위해 time.h 라이브러리에 있는 clock 함수를 사용했다. 시간 측정 단위는 millisecond로 설정하기 위해 알고리즘이 실행되기 시작하는 시간과 끝나는 시간의 interval을 구하여 CLOCKS_PER_SEC로 나눈 후 1000을 곱하였다.
- 각 알고리즘의 input의 개수에 따른 시간 복잡도(Time Complexity)를 조사하기 위해 순차적으로 10, 10×5, 10^2, $10^2\times5$, 10^3, $10^3\times5$, 10^4, $10^4\times5$, 10^5, $10^5\times5$, 10^6개의 input에 따른 세 가지 케이스를 실행했다. 실험 결과의 정확도를 높이기 위해 각각의 케이스마다 3번씩 시행하여 평균값을 구한 뒤 실험 결과에 반영했다.
- 4개의 알고리즘의 성능 실험에서는 input으로 받는 list가 내림차순으로 정렬되어 있는 경우, 그리고 무작위로 정렬되어 있는 경우 두 가지를 실험했다.
- 실험에서 사용할 테스트 케이스는 numbergenerator.org의 Random Numbers Generator(http://numbergenerator.org/)를 이용했다. 그리고 실제로 프로그램이 잘 작동했는지에 관한 검증은 http://www.endmemo.com/math/numsort.php 페이지를 이용했다.

Ⅱ 알고리즘 성능 비교 실험

이번 실험에서는 Insertion Sort, Quick Sort, Merge Sort, 그리고 직접 구현한 정렬 알고리즘인 Radix-Introspective(Radix-Intro) Sort의 성능을 비교하기 위해 input의 element 개수를 10, 10×5, 10^2, $10^2\times5$, 10^3, $10^3\times5$, 10^4, $10^4\times5$, 10^5, $10^5\times5$, 10^6로 설정하여 진행했다. 각 케이스마다 non-increasing 순으로 정렬된 list, 그리고 임의의 3개의 random 순으로 정렬된 list를 실행했으며, random 순으로 정렬한 list의 실험 결과는 3개의 list 정렬 시간의 평균값으로 계산했다.

1. Non-Increasing Order List Sort

[그림 Ⅱ-1] 4개의 알고리즘의 Non-Increasing Order List Sort 그래프

Input의 element 수(개)	Insertion	Quick	Merge	Radix-Intro
10	0	0	0	0
50	0	0	0	0
100	0	0	0	0
500	2	2	1	0

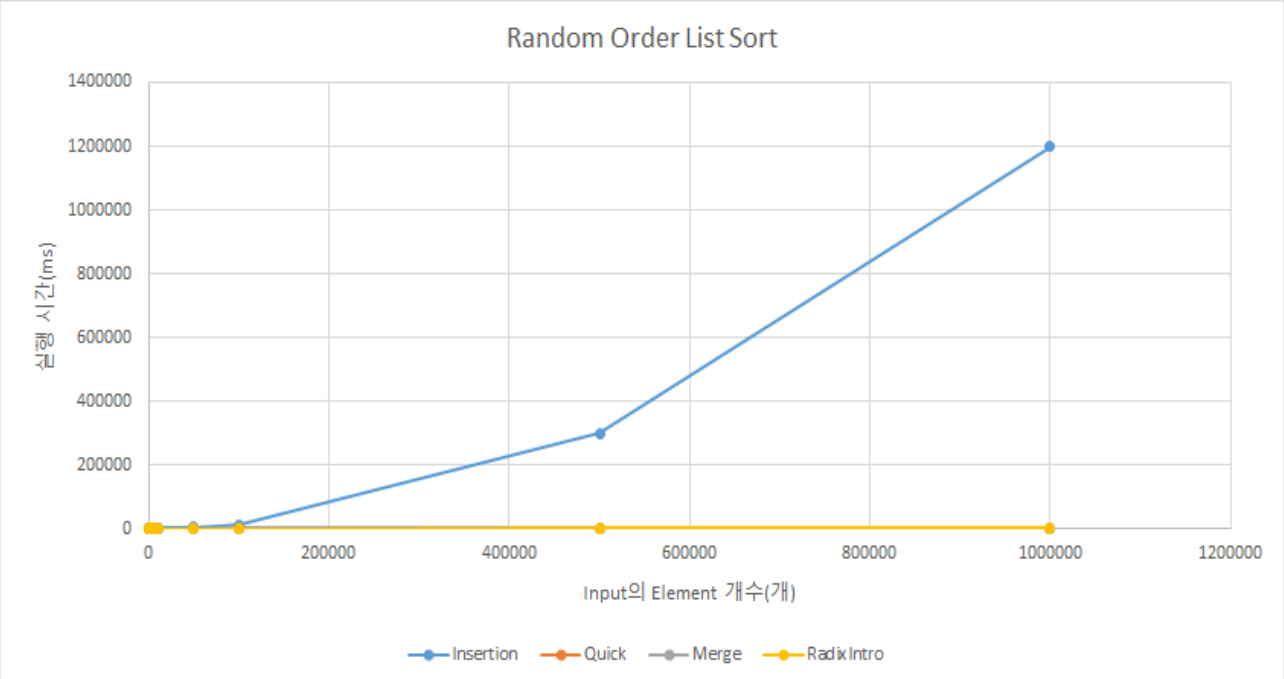
1000	4	6	2	0
5000	117	134	8	1
10000	308	402	18	1
50000	6190	7918	51	5
100000	24708	26839	101	13
500000	599425	758214	448	57
1000000	2400138	3144631	904	117

[표 Ⅱ-1] 4개의 알고리즘의 input 개수에 따라 걸린 시간 (시간 단위: milliseconds)

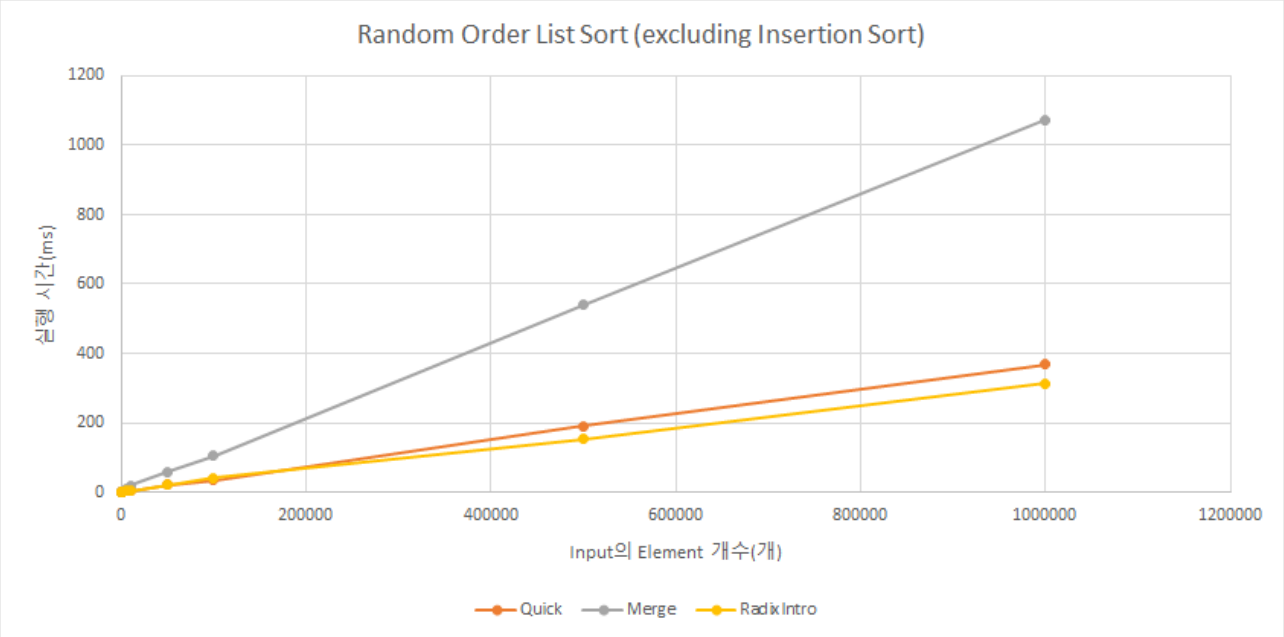
Non-Increasing 순으로 정렬된 list를 non-decreasing 순으로 정렬하는 데 걸리는 시간을 나타낸 *[그림 Ⅱ-1]*과 *[표 Ⅱ-1]*이다. Input의 element 개수가 늘어날수록 전반적으로 Quick Sort와 Insertion Sort의 실행 시간이 기하급수적으로 증가하고, Merge Sort와 Radix-Intro Sort의 실행 시간은 element 개수 증가에 비해 완만하게 증가한다는 사실을 알 수 있다. 모든 경우에서 실행 시간이 가장 적게 걸린 정렬 알고리즘은 Radix-Intro Sort이고, 가장 많이 걸린 건 Quick Sort이다.

Non-Increasing 순을 다시 Non-Decreasing 순으로 정렬하는 상황은 정렬에 있어서 최악의 경우(worst case)로 볼 수 있다. 그러므로 Insertion Sort와 Quick Sort는 정렬하는 데 걸리는 시간이 최악의 경우에 $O(n^2)$에 근접하는 반면에, Merge Sort와 Radix-Intro Sort는 최악의 경우 $O(n\log n)$에 근접함을 알 수 있다.

2. Random Order List Sort



[그림 Ⅱ-2] 4개의 알고리즘의 Random Order List Sort 그래프



[그림 Ⅱ-3] Insertion Sort를 제외한 3개의 알고리즘의 Random Order List Sort 그래프

Input의 element 수(개)	Case 1	Case 2	Case 3	Average
10	0	0	0	0
50	0	0	0	0
100	0	0	0	0
500	1	1	1	1
1000	3	4	3	3.333333
5000	68	59	66	64.33333
10000	187	192	193	190.6667
50000	3119	3367	3148	3211.333
100000	12416	12820	12559	12598.33
500000	299699	300171	298719	299529.7
1000000	1198525	1198288	1196944	1197919

[표 Ⅱ-2] Insertion Sort의 input 개수에 따라 걸린 시간 (시간 단위: milliseconds)

Input의 element 수(개)	Case 1	Case 2	Case 3	Average
10	0	0	0	0
50	0	0	0	0
100	0	0	0	0
500	0	0	0	0
1000	1	1	1	1
5000	4	4	3	3.666667
10000	4	5	4	4.333333
50000	21	22	22	21.66667

100000	30	41	33	34.66667
500000	191	188	192	190.3333
1000000	365	369	368	367.3333

[표 Ⅱ-3] Quick Sort의 input 개수에 따라 걸린 시간 (시간 단위: milliseconds)

Input의 element 수(개)	Case 1	Case 2	Case 3	Average
10	0	0	0	0
50	0	0	0	0
100	0	0	0	0
500	1	1	1	1
1000	2	2	2	2
5000	10	9	11	10
10000	20	18	20	19.33333
50000	58	61	59	59.33333
100000	105	103	103	103.6667
500000	537	541	540	539.3333
1000000	1106	1009	1105	1073.333

[표 Ⅱ-4] Merge Sort의 input 개수에 따라 걸린 시간 (시간 단위: milliseconds)

Input의 element 수(개)	Case 1	Case 2	Case 3	Average
10	0	0	0	0
50	0	0	0	0
100	0	0	0	0
500	0	0	0	0
1000	1	1	1	1
5000	4	4	4	4
10000	4	4	4	4
50000	21	21	22	21.33333
100000	42	40	40	40.66667
500000	152	156	151	153
1000000	316	311	314	313.6667

[표 Ⅱ-5] Radix-Intro Sort의 input 개수에 따라 걸린 시간 (시간 단위: milliseconds)

Random 순으로 정렬된 list를 non-decreasing 순으로 정렬하는 데 걸리는 시간을 나타낸 *[그림 Ⅱ-2]~[그림 Ⅱ-3]*과 *[표 Ⅱ-2]~[표 Ⅱ-5]*이다. Input의 element 개수가 늘어날수록 전반적으로 Insertion Sort의 실행 시간이 기하급수적으로 증가하고, Insertion Sort, Merge Sort, 그리고 Radix-Intro Sort의 실행 시간은 element 개수 증가에 비해 완만하게 증가한다는 사실을 알 수 있다. 전반적으로 실행 시간이 가장 적게 걸린 정렬 알고리즘은 Radix-Intro Sort이고, Quick Sort의 실행 시간이 이보다 근소하게 앞선 케이스가 존재한다. Non-Increasing Order List 정렬 때와는 다르게

Quick Sort의 실행 시간이 Radix-Sort에 매우 근접하여 적게 걸렸음을 알 수 있다.
Random 순을 Non-Decreasing 순으로 정렬하는 상황은 정렬에 있어서 일반적인 경우로 볼 수 있다. 그러므로 Insertion Sort는 정렬하는 데 걸리는 시간이 일반적인 경우에 $O(n^2)$에 근접하는 반면에, Quick Sort, Merge Sort, 그리고 Radix-Intro Sort는 일반적인 경우 $O(n\log n)$에 근접함을 알 수 있다.

Ⅲ 효율적인 알고리즘 구현

1. 알고리즘 설명

이번 실험에서 정렬이 빠른 알고리즘을 구현하기 위해 Radix Sort와 Quick Sort를 최적화한 Introspective Sort를 결합시켰다. 특정 조건에 따라 유리한 정렬 알고리즘을 선택하여 수행하는데, 일반적인 경우에는 Introspective Sort를 시행한다. 이 정렬 알고리즘은 다음과 같은 과정에 따라 정렬을 수행한다.

1) main 함수에서 파라미터로 받은 List의 길이만큼 해당하는 메모리를 두 개에 배열에 할당한다. 한 배열은 음수만을 저장하는 배열(NegativeNumber)이고, 다른 배열은 양수 또는 0을 저장하는 배열(NonNegativeNumber)이다.
2) 파라미터로 받은 List의 원소를 처음부터 끝까지 탐색하면서 음수는 배열 NegativeNumber에 저장하고, 그렇지 않으면 배열 NonNegativeNumber에 저장한다. 배열에 저장하면서 그 원소 값이 최댓값인지 그리고 자릿수는 얼마인지 확인하는데, 만일 이전 배열에서의 최댓값보다 크면 이를 최댓값으로 업데이트하고 이 최댓값의 자릿수도 업데이트한다. 배열을 탐색하면서 음수와 0보다 같거나 큰 수가 각각 몇 개인지 센다.
3) 음수에서의 최댓값의 자릿수와 양수에서의 최댓값의 자릿수를 비교하여 더 큰 것을 선택하고, 자릿수가 더 큰 값과 $\log_{10}(List\text{의길이})$ 값을 비교한다. 전자가 더 크면 Introspective Sort를 진행하고, 그렇지 않으면 Radix Sort를 진행한다.
4) Introspective Sort에서는 recursion을 통해 List를 일부분으로 계속 줄여간다. 오른쪽 index가 왼쪽 index보다 20만큼 크면 계속 recursion을 수행한다. 왼쪽 index, 중간 index, 그리고 오른쪽 index를 파라미터로 받는 Partition 함수를 통해 일부분으로 나눌 때 기준점이 되는 Pivot을 결정한다. 이 Pivot을 기준으로 Pivot에서 시작하여 오른쪽 index까지의 List를 recursion의 파라미터로 넘긴다. 일정 Depth만큼 도달하면 heap sort를 수행하고 알고리즘을 종료하는데, heap sort는 이미 자기 자신의 sub-tree가 이미 heap에 맞게 정렬되어 있음을 전제해야 하므로 AdjustHeap 함수를 통해 자식부터 부모까지 모두 정렬이 된 상태가 되도록 한다. Depth에 도달하지 않았는데 recursion을 통해 파라미터로 받은 list의 원소 개수가 20개 이하(이번 실험에서는 테스트했을 때 가장 효율이 좋았던 20 값으로 설정함)이면 이미 거의 다 정렬이 되었으므로 Insertion Sort를 수행하여 나머지 원소들을 정렬한다.
5) Radix Sort에서는 2)번 과정에서 얻은 최댓값을 기준으로 각 배열에 따라 자릿수를 차례대로

비교해 가면서 정렬한다. 자릿수마다 0부터 9까지의 숫자가 몇 번 나오는지를 세고 이를 다른 임시 배열에 저장하고, 각 임시 배열의 값을 처음부터 계속 누적한 값으로 업데이트한다. 저장한 배열을 차례대로 훑으면서 자릿수 값에 해당하는 숫자의 index에 해당하는 임시 배열의 원소로 찾아가서 그 값을 index로 하여 또 다른 임시 배열에 저장한다. 이 임시배열을 원래 배열에 복사하여 해당 자릿수에서는 정렬이 완료된 배열을 얻는다. 단, 음수 배열에서는 자릿수의 숫자가 작아질수록 수가 커지기 때문에 non-increasing 순으로 우선 정렬한 후 나중에 음수(-) 부호를 붙인다. 음수 배열과 양수 또는 0 배열을 List에 복사하여 합친다.

2. 효율적인 이유

일반적으로 어떤 한 수를 다른 수와 비교하면서 전체 리스트를 정렬해 가는 방법은 시간 복잡도를 최소한으로 줄여도 $O(n\log n)$을 벗어날 수 없다. 이는 n개의 수를 정렬한다고 가정할 때, 최종적으로 비교를 통해 정렬된 결과를 수열로 나타내면 경우의 수는 총 $n!$개이다. 이를 binary tree로 구성하면 최소한의 깊이는 $\log(n!) = n\log n$이다. 따라서 일반적인 비교가 아닌 다른 방식으로의 정렬이 유리할 수도 있다는 의미이다.

Radix Sort는 직접 두 수를 비교하지 않고 각 수들의 자릿수를 확인하면서 숫자가 몇 개 있는지를 bucket에 저장하는 방식이다. 그래서 정렬하고자 하는 수들의 자릿수가 몇 개인지가 관건이다. 수업 시간에 배운 바에 의하면 Radix Sort를 분석했을 때의 시간 복잡도는 $O(d(n+b)) = O((n+b)\log_b m)$ (b는 진수, m은 최댓값)이다. 그래서 최댓값의 자릿수가 작으면 작을수록 Radix Sort가 더 효율적일 수 있다.

위의 실험에서 살펴본 바와 같이 Quick Sort는 일반적인 경우는 $O(n\log n)$, 최악의 경우는 $O(n^2)$의 시간 복잡도를 갖는다. 이러한 문제점을 해결한 Introspective Sort는 시간 복잡도가 일반적으로 $O(n\log n)$인 Heap Sort를 결합하고 Pivot이 가장 큰 값이 아니도록 중간값을 찾아서 적절한 Pivot을 찾는 과정을 통해 최악의 경우에도 시간 복잡도가 $O(n\log n)$이 되도록 한다. 그러나 Radix Sort에 비하면 자릿수가 비교하고자 하는 수들의 개수보다 훨씬 클 때는 불리해진다. 반대로 비교하고자 하는 수들의 개수는 많은데 자릿수가 작으면 유리해진다.

그래서 정렬하고자 하는 list의 최댓값을 찾아서 이의 자릿수를 계산한 후 나온 값을 $\log n$과 비교하여 경우에 따라 어떠한 알고리즘을 수행할지 정하는 것이 효율적이다. 그러므로 이번 실험에서의 알고리즘은 $\log_b m$과 $\log n$을 비교하여 전자가 작으면 Radix Sort를, 후자가 작으면 Introspective Sort를 수행하도록 한 것이다. 실제로 -9999부터 9999까지의 범위에 속하는 100000개 이상의 배열을 정렬할 때 Radix Sort가 21ms, Introspective Sort가 25ms로 전자가 더 적게 걸림을 확인하였다. 그러나 이번 실험에서의 테스트 케이스에서는 범위가 int data type으로 나타낼 수 있는 전범위에 해당하므로 현실적으로는 Introspective가 더 자주 수행된다.

Ⅲ 결론

1. 결론 및 논의

- 일반적인 경우에는 Insertion Sort의 시간 복잡도가 $O(n^2)$에 수렴하여 가장 느리고, 그 외 나머지 알고리즘은 $O(n\log n)$에 수렴하는 시간 복잡도를 갖는다.
- 최악의 경우에는 Insertion Sort와 Quick Sort의 시간 복잡도가 $O(n^2)$에 수렴하여 가장 느리고, 그 외 나머지 알고리즘은 $O(n\log n)$에 수렴하는 시간 복잡도를 갖는다.
- 전반적으로 Radix-Introspective Sort의 수행 속도가 가장 빠르며, 일반적인 경우에서는 Quick Sort가 이에 근접한 속도를 보인다. 그러나 최악의 경우에는 Quick Sort의 시간 복잡도는 기하급수적으로 증가하므로 Radix-Introspective Sort가 더 효율적이다.
- input 개수가 매우 작으면 4개의 알고리즘의 걸리는 시간은 상대적으로 큰 차이가 없음을 유추할 수 있다.
- Radix-Introspective Sort를 수행할 때 최댓값의 자릿수가 전체 integer 수에 log를 취한 값보다 크면 Introspective Sort를, 그렇지 않으면 Radix Sort를 수행하는 것이 시간 복잡도를 고려했을 때 더 효율적이다.

2. 추가 의견

- Radix-Introspective Sort의 단점은 list를 파라미터로 받으면 우선 list의 처음부터 끝까지 모든 원소를 훑어보는 과정이 강제된다는 것이다. 이는 최댓값의 자릿수에 따라서 Radix Sort를 수행할지 말지를 결정하는 과정이 선행되기 때문인데, 만일 단일로 Introspective Sort만 구현한다면 이러한 과정은 필요 없을 것이다. 이번 실험에서 테스트 케이스가 대개 자릿수가 log$n$보다 확률적으로 더 크기 때문에 Radix Sort가 거의 수행되지 못한다는 점이 매우 아쉽다.