자료구조 정렬 보고서

팀원 : 서민정 윤겸지 이은빈

<1> Stable Sort 분석

1. stable한 sort가 어떻게 stable한 결과를 보일 수 있는지 알고리즘을 기반으로 설명하시오.

▶ Insertion sort

Insertion sort는 임의의 1~N까지 정렬된 배열에 정렬이 되지 않은 부분의 가장 왼쪽 값(N+1번째 값)을 삽입 하는 방식이기 때문에 stable한 결과를 보일 수 있다.

▶ Merge sort

Merge sort는 크기가 n인 입력을 1/2n을 크기로 하는 두 개의 입력으로 분할한다. 분할된 입력들이 최종적으로 1의 크기를 가질 때까지 분할한다. 각 분할 입력들을 재귀적으로 정렬하며 합병한다. Merge sort가 정렬하는 과정 중 분할하는 단계에서 각 값들의 위치는 변화하지 않는다. 다음으로 각 분할 입력들을 정렬하며 합병하는 단계에서 같은 key을 가지는 값의 위치가 swap 되는 경우는 발생하지 않기 때문에 Merge sort는 stable한 결과를 보일 수 있다.

▶ Radix sort

Radix sort는 각 단계가 진행될 때 반드시 stable한 결과를 가져야한다. stable한 결과를 가지지 않으면 Radix sort가 성립할 수 없기 때문이다. LSD 사용 시 stable하게 정렬하지 않으면, 10의 자리에서 정렬할 때 1의 자리에서 정렬한 값이 바뀔 수 있기 때문이다.

2. non-stable한 sort는 stable하지 않은 예제를 보여 stable 하지 않음을 보이시오.

▶ selection

* 정렬 전

9	6	7	3	6'	5

---

* 1회전

9	6	7	3	6'	5

3	6	7	9	6'	5

---

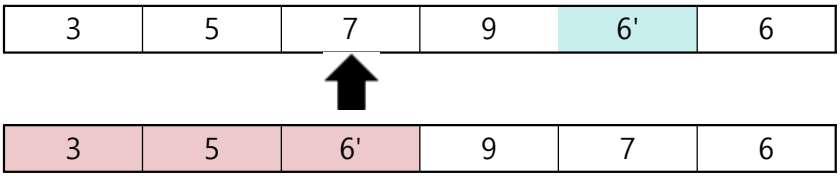
* 2회전

3	6	7	9	6'	5

3	5	7	9	6'	6

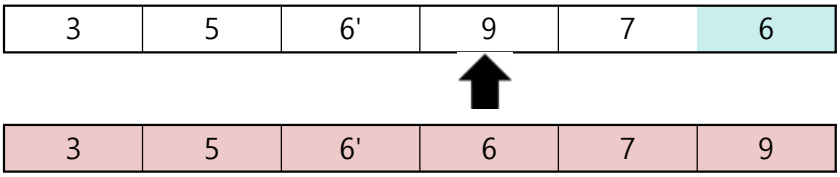
---

* 3회전



---

* 4회전



---

* 정렬 후

3	5	6'	6	7	9

=> 입력 시 6, 6'이었던 순서가 정렬 후 뒤바뀌었으므로 selection sort는 non-stable하다.

▶ shell

* 정렬 전

20	35	15	90	70	1	10	35'

(gap sequence : 4 - 2 - 1 순)

---

* 1회전 -> gap : 4

Index	0	1	2	3	4	5	6	7
Data	20	35	15	90	70	1	10	35'

20				70			
	35				1		
		15				10	
			90				35'

20				70			
	1				35		
		10				15	
			35'				90

* 1회전 결과

20	1	10	35'	70	35	15	90

---

* 2회전 -> gap : 2

20		10		70		15	
	1		35'		35		90

10		15		20		70	
	1		35'		35		90

-2회전 결과

10	1	15	35'	20	35	70	90

---

* 3회전 -> gap : 1

10	1	15	35'	20	35	70	90

-3회전 결과

1	10	15	20	35'	35	70	90

---

* 정렬 후

1	10	15	20	35'	35	70	90

=> 입력 시 35, 35'이었던 순서가 정렬 후 뒤바뀌었으므로 shell sort는 non-stable하다.

▶ quick

* 정렬 전

R1	R2	R3	R4	R5	R6	R7	R8	R9
35	60	80	20	10	5	15	10'	90

Pivot

[35	60	80	20	10	5	15	10'	90]

Pivot

[5	10'	15	20	10]	35	[80	60	90]

Pivot

5	10'	[15	20	10]	35	[80	60	90]

Pivot

5	10'	10	15	20	35	[80	60	90]

* 정렬 후

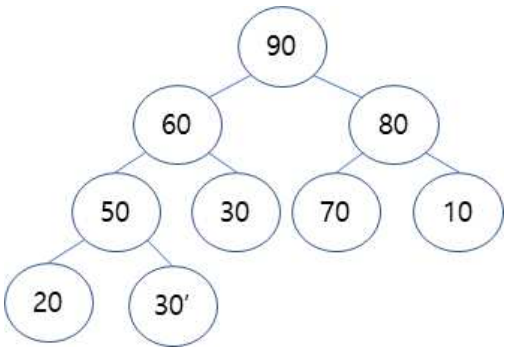
5	10'	10	15	20	35	60	80	90

=> 입력 시 10, 10'이었던 순서가 정렬 후 뒤바뀌었으므로 quick sort는 non-stable하다.

▶ heap

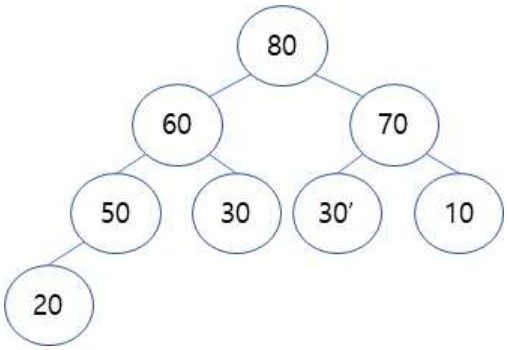
* 정렬 전

0	1	2	3	4	5	6	7	8	9
	90	60	80	50	30	70	10	20	30'



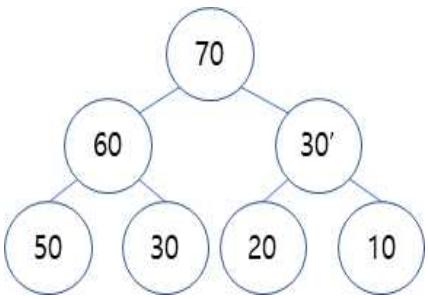
* 90과 30‘ 교환 후 downheap()

	80	60	70	50	30	30'	10	20	90



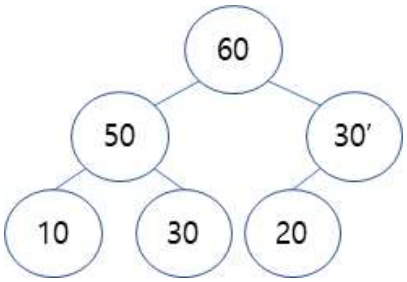
* 80과 20 교환 후 downheap()

	70	60	30'	50	30	20	10	80	90



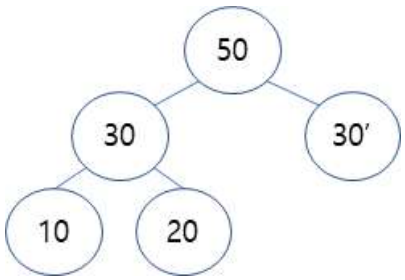
* 70과 10 교환 후 downheap()

	60	50	30'	10	30	20	70	80	90



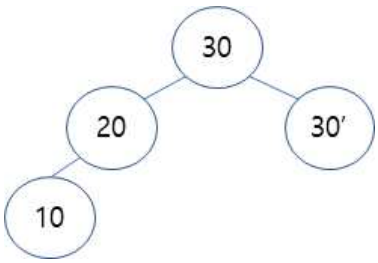
* 60과 20 교환 후 downheap()

	50	30	30'	10	20	60	70	80	90



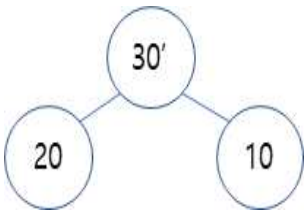
* 50과 20 교환 후 downheap()

	30	20	30'	10	50	60	70	80	90



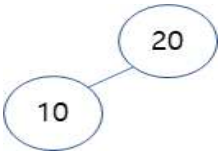
* 30과 10 교환 후 downheap()

	30'	20	10	30	50	60	70	80	90



* 30'과 10 교환 후 downheap()

	20	10	30'	30	50	60	70	80	90

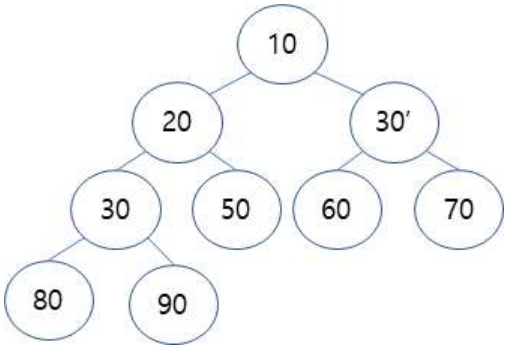


* 20과 10 교환 후 downheap()

	10	20	30'	30	50	60	70	80	90

* 정렬 후

0	1	2	3	4	5	6	7	8	9
	10	20	30'	30	50	60	70	80	90



=> 입력 시 30, 30'이었던 순서가 정렬 후 뒤바뀌었으므로 heap sort는 non-stable하다.

<2> 비교 정렬 알고리즘

테스트 환경

■ 조건

- 한 대의 pc를 사용하여 pc의 성능에 의한 속도차이가 없도록 한다.
- pc의 휴식기를 설정하여 속도저하의 영향을 최소하하여 정렬을 실행한다.

■ PC환경

- OS : Windows 10
- 컴파일러 : JDK javac
- CPU : AMD Ryzen 5 3500U 2.10 GHz
- RAM : 8.00GB

■ 성능 비교방법

- 정렬 성능 측정에 사용되는 데이터는 정수, 실수, 문자열, 클래스(Student) 데이터 타입을 쓴다.
- 위 데이터 타입을 오름차순, 내림차순, 랜덤 상태로 구성된 배열을 대상으로 한다.
- 위 데이터를 담는 배열은 각 1천, 5천, 1만, 2만5천 5만의 크기를 가진다.

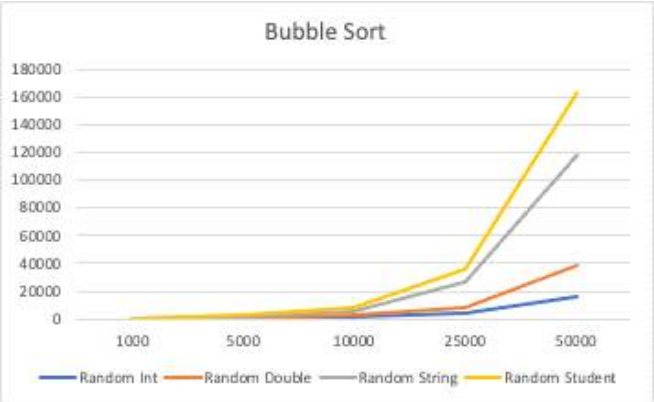
■ 시간 측정

- 현재 시간을 받아오는 System.currentTimeMillis() 함수를 이용한다.
- 시작 시간과 끝나는 시간의 차이를 계산한다.
- 수행 시간 측정 단위는 m-sec이다.

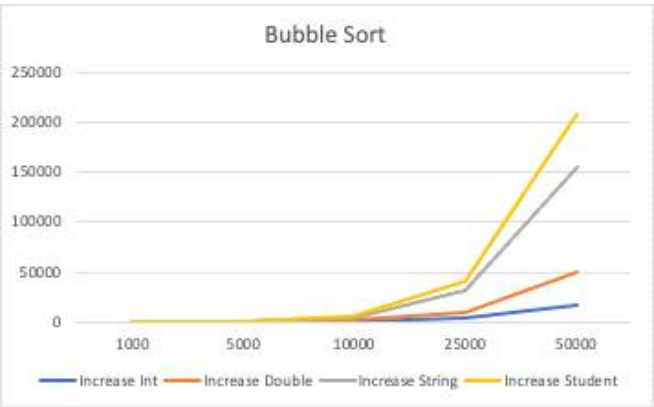
■ Bubble sort

> 자료형별 정렬시간 (random, increase, decrease 순서)

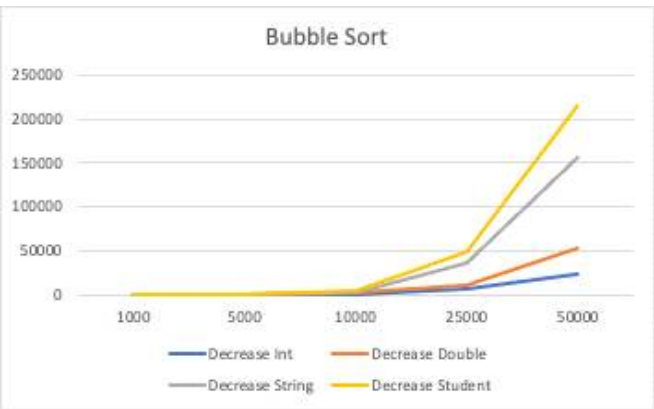
	Int	Double	String	Student
1000	64	61	31	25
5000	256	278	1429	404
10000	1203	1493	3046	1859
25000	4046	4727	17794	9285
50000	16251	22316	80091	44522



	Int	Double	String	Student
1000	12	12	21	17
5000	85	331	601	238
10000	371	1317	3304	1832
25000	4797	5307	21271	8699
50000	16704	32816	105830	53406



	Int	Double	String	Student
1000	14	8	21	14
5000	233	226	553	349
10000	964	993	1765	1282
25000	6262	5406	24805	11643
50000	23628	28849	103138	58315



> 분석

Bubble Sort는 서로 인접한 두 원소를 비교하여 크기가 순서대로 되어 있지 않으면 서로 교환하는 방법이다. 1회전을 수행하고 나면 가장 큰 자료가 맨 뒤로 이동하므로 2회전에서 맨 뒤의 자료는 정렬에서 제외된다. 1회전을 수행할 때마다 정렬에서 제외되는 데이터가 하나씩 늘어난다.

Bubble Sort의 비교횟수는 최상, 평균, 최악의 경우 모두 항상 일정하다.

1회전의 비교횟수는 N-1번이므로 2회전에서는 N-2번, 마지막은 1번으로 총 $\frac{N(N-1)}{2}$번의 비교를 한다.

교환횟수는 배열이 이미 정렬된 최상의 경우 자료의 이동이 발생하지 않는다.

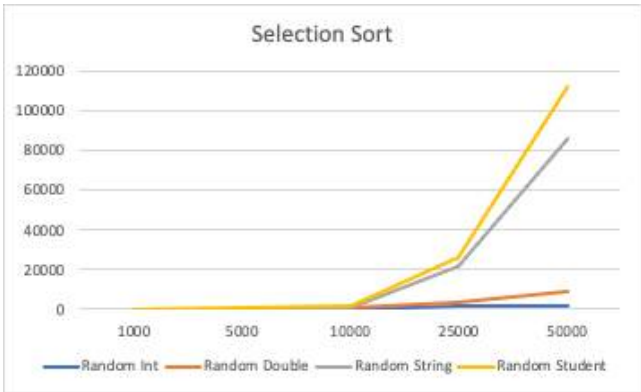
역순으로 정렬되어 있는 최악의 경우 한 번 교환하기 위해 swap 함수의 작업에서 3번의 이동이 필요하므로 비교횟수 * 3 = 3N(N-1)/2번으로 O(N^2)이다.
따라서 전체적인 시간 복잡도는 O(N^2)이다.

Bubble Sort는 구현은 매우 간단하지만 다른 알고리즘에 비해 수행 시간이 오래 걸리므로 단순성에도 불구하고 거의 쓰이지 않는다.

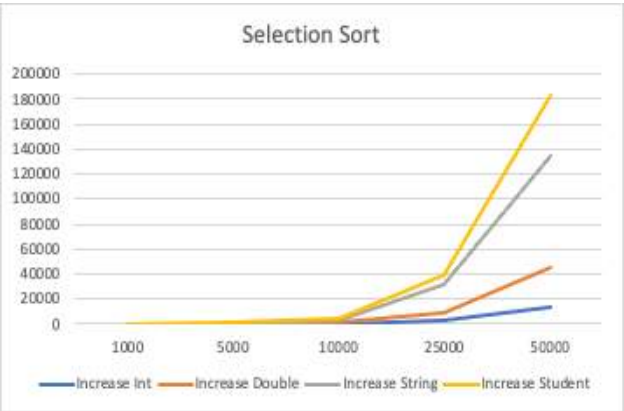
■ Selection sort

> 자료형별 정렬시간 (random, increase, decrease 순서)

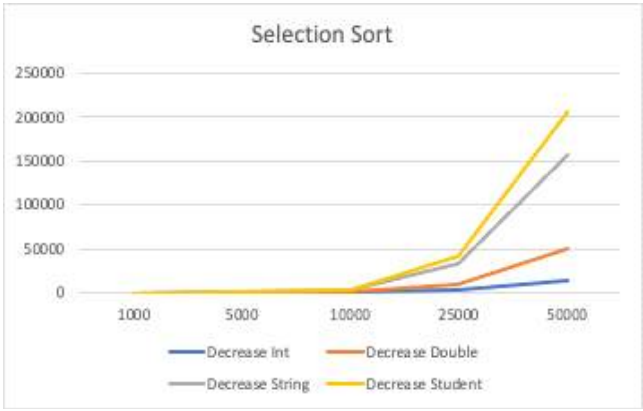
	Int	Double	String	Student
1000	38	92	77	18
5000	52	164	213	206
10000	166	298	878	913
25000	1454	2015	18178	5000
50000	4613	7300	77536	26269



	Int	Double	String	Student
1000	23	6	19	22
5000	173	159	245	228
10000	554	614	1799	1178
25000	2532	6896	21664	8234
50000	13522	31698	89184	48349



	Int	Double	String	Student
1000	11	20	31	12
5000	162	196	321	224
10000	607	552	1910	1150
25000	3691	6484	23860	8468
50000	14303	36509	107394	48526



> 분석

Selection Sort는 Insertion Sort처럼 배열을 정렬된 부분과 정렬되지 않은 부분으로 나눈다. 정렬되지 않은 부분의 원소들 중 최솟값을 '선택'하여 정렬된 부분의 바로 오른쪽 원소와 교환한다. 즉, 원소를 넣을 위치는 이미 정해져 있고, 그 위치에 어떤 원소를 넣을지 선택하는 알고리즘이다.

처음 루프 수행에서 N개의 원소들 중 최소값을 찾기 위해 N-1번 원소를 비교하고, 루프가 2번 째 수행될 때 N-1개의 원소들 중 최소값을 찾는 데 N-2번 비교한다. 같은 방식으로 루프가 마지막으로 수행될 때 2개의 원소를 1번 비교한다. 따라서 원소들의 총 비교횟수는 (N-1) + (N-2) + ⋯ + 2 + 1 = $\frac{N(N-1)}{2}$ = $O(N^2)$이다.

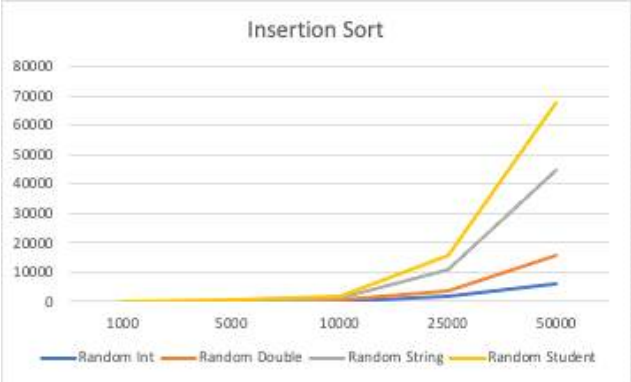
입력이 역정렬된 최악의 경우 최솟값을 찾은 후 원소를 교환하는 횟수는 N-1로 미리 정해져 있다.
따라서 최선, 평균, 최악의 경우 모두 O(N^2) 수행시간이 소요된다.

Selection Sort는 구현이 매우 간단하지만 전체 시간 복잡도가 O(N^2)이므로 효율적이지 않다.

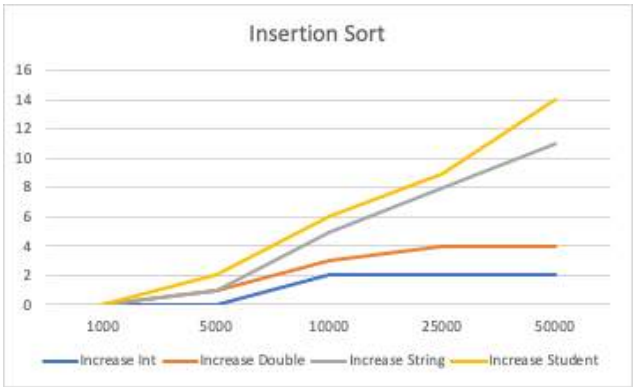
■ Insertion sort

> 자료형별 정렬시간 (random, increase, decrease 순서)

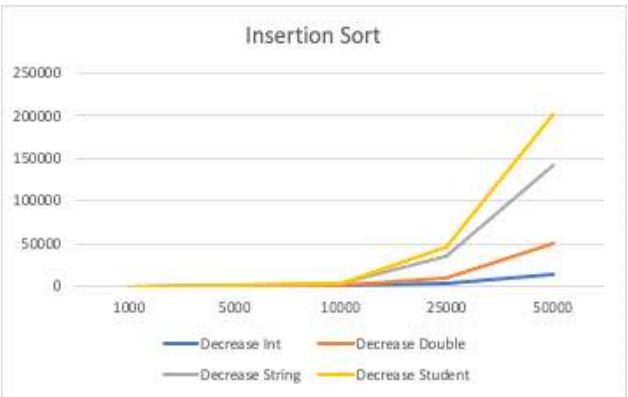
	Int	Double	String	Student
1000	0	0	0	0
5000	88	75	128	138
10000	265	247	517	551
25000	1575	1873	7633	4564
50000	6085	9469	28928	23386



	Int	Double	String	Student
1000	0	0	0	0
5000	0	1	0	1
10000	2	1	2	1
25000	2	2	4	1
50000	2	2	7	3



	Int	Double	String	Student
1000	0	0	0	0
5000	238	191	326	227
10000	541	834	1304	1083
25000	3895	6218	25248	9915
50000	14799	35616	92250	59016



> 분석

Insertion Sort는 배열을 정렬된 부분과 정렬되지 않은 부분으로 나누며, 정렬되지 않은 부분의 가장 왼쪽 원소를 정렬된 부분의 올바른 자리에 '삽입'하는 방식의 알고리즘이다.

Insertion Sort의 복잡도는 입력 자료의 구성에 따라 달라진다.
입력이 이미 정렬된 최선의 경우 이동 없이 총 N-1번의 비교가 이루어지므로 시간복잡도는 O(N)이다.
입력이 역으로 정렬된 최악의 경우 정렬되지 않은 부분의 원소들을 하나씩 확인할 때마다 왼쪽의 정렬된 원소들을 모두 확인하므로 비교횟수는 1 + 2 + ⋯ + (N-2) + (N-1) = $\frac{N(N-1)}{2}$ = $O(N^2)$이며

이동 횟수는 O(N^2)이다. 따라서 최악의 경우 시간복잡도는 O(N^2)이다.
입력 순서가 랜덤인 평균의 경우 현재 원소가 정렬된 앞 부분에 삽입되는 곳이 평균적으로 정렬된 부분의 중간이므로 시간복잡도는 $\frac{1}{2} * \frac{N(N-1)}{2} \approx \frac{1}{4}N^2 = O(N^2)$ 이다.

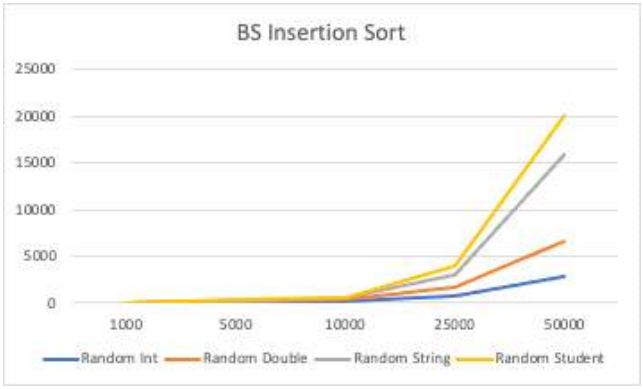
Insertion sort의 정렬 수행시간 측정 결과 이미 정렬된 오름차순 배열의 경우 데이터의 이동이 발생하지 않으므로 0초에 가까운 시간이 소요되지만, 역정렬된 내림차순 배열의 경우 데이터의 개수가 많아질수록 정렬하는 데 더 많은 시간이 소요된다.

Insertion Sort는 비교적 많은 데이터들의 이동이 일어나므로 데이터의 크기가 클 경우 적합하지 않다. 반면 구현이 간단하여 데이터의 개수가 적을수록 특히 데이터가 이미 정렬된 경우 다른 복잡한 정렬 방법보다 유리할 수 있다.

■ Binary Insertion sort

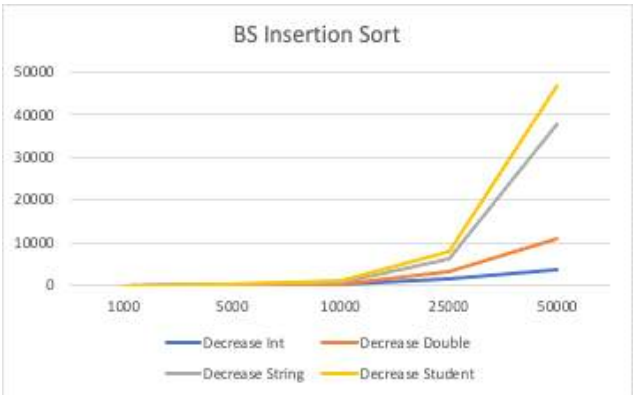
> 자료형별 정렬시간 (random, increase, decrease 순서)

	Int	Double	String	Student
1000	0	0	0	0
5000	112	67	61	55
10000	170	152	150	166
25000	838	889	1335	876
50000	2922	3770	9197	4132



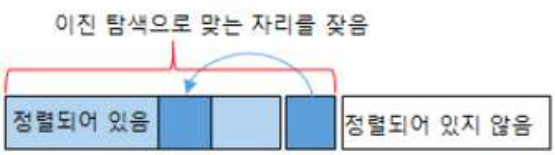
	Int	Double	String	Student
1000	0	0	0	0
5000	1	3	3	2
10000	4	2	5	5
25000	18	9	11	11
50000	39	16	46	30

	Int	Double	String	Student
1000	0	0	0	0
5000	89	80	73	59
10000	248	176	275	256
25000	1486	1649	3028	1635
50000	3475	7580	26971	8773



> 분석

Binary Insertion Sort는 Insertion Sort를 개선한 알고리즘으로, 삽입시킬 부분의 리스트가 이미 정렬되어 있다는 점에 착안하여 삽입 위치를 이진 탐색(binary search)으로 찾는 방법이다.



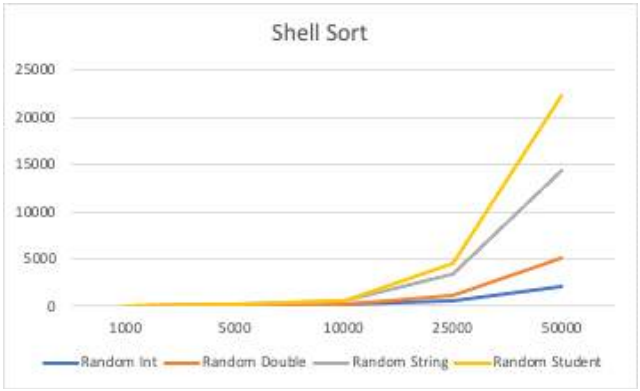
- 이진 탐색으로 삽입 위치를 찾는 Insertion sort

Insertion sort에 자료의 삽입 위치를 찾기 위한 탐색 과정에서 이진 탐색을 적용하였을 때,
최선의 경우 O(N)에서 O(logN)으로 개선되며 최악의 경우 O(N^2)에서 O(logN)으로 개선된다.
이동 횟수는 최선의 경우 데이터의 이동이 없고, 최악의 경우 이진 탐색을 적용하지 않았을 때와 마찬가지로 O(N^2)이다. 따라서 이진 탐색을 적용한 Insertion sort는 데이터를 비교할 때의 수행 시간을 개선해준다.

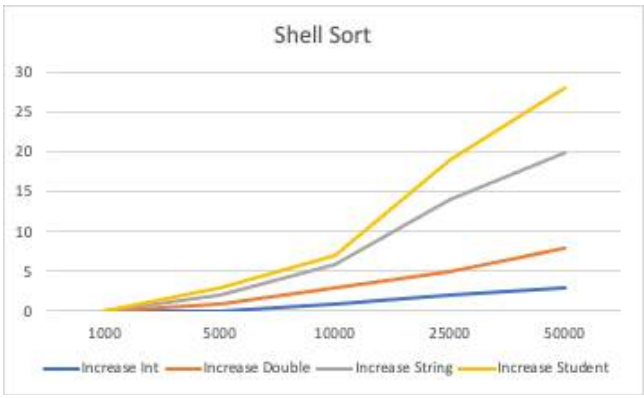
■ Shell sort

> 자료형별 정렬시간 (random, increase, decrease 순서)

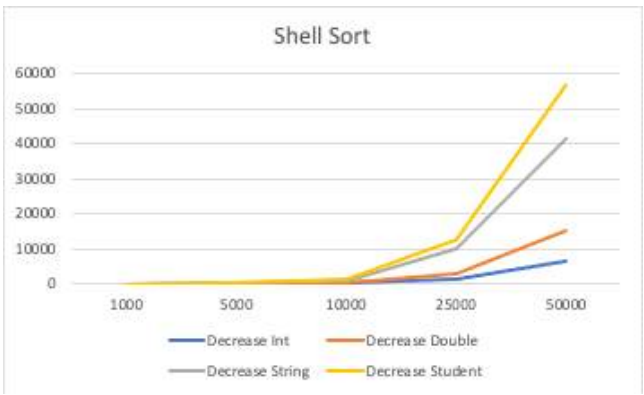
	Int	Double	String	Student
1000	53	10	6	10
5000	58	54	51	33
10000	134	137	204	145
25000	525	548	2245	1233
50000	2053	2986	9261	7889



	Int	Double	String	Student
1000	0	0	0	0
5000	0	1	1	1
10000	1	2	3	1
25000	2	3	9	5
50000	3	5	12	8



	Int	Double	String	Student
1000	3	2	4	4
5000	57	50	92	68
10000	196	302	536	386
25000	1149	1747	7067	2811
50000	6236	8874	26265	15305



> 분석

Shell Sort는 Insertion Sort에 전처리 과정을 추가한 알고리즘이다. 전처리 과정은 작은 값을 가진 원소들은 배열의 앞부분에 옮기면서 큰 값을 가진 원소들이 배열의 뒷부분에 자리잡도록 만드는 과정이다. 전처리 과정은 여러 단계로 진행되며, 각 단계에서는 일정 간격으로 떨어진 원소들에 대해 Insertion Sort를 수행한다.

간격은 3x + 1으로 처음 간격은 4로 설정하였고, 마지막 간격은 1로 하여 정렬한다.
수행시간은 간격을 어떻게 설정하느냐에 따라 달라지므로 Shell Sort에서는 간격을 잘 설정하는 것이 중요하다.

Insertion Sort의 문제는 요소들이 삽입될 때, 이웃한 위치로 한 칸씩만 이동한다는 점이다. 만약 삽입되

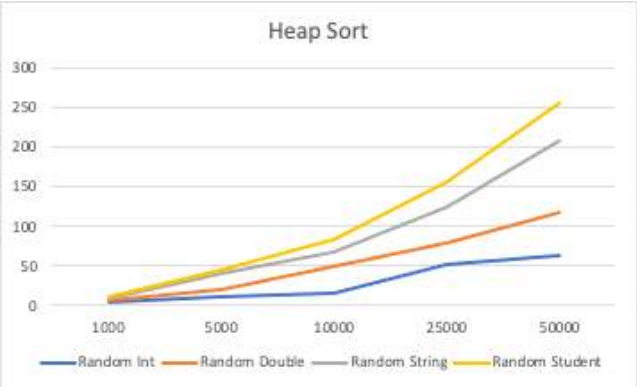
어야 할 위치가 현재 위치에서 멀리 떨어진 곳이라면 많은 이동을 해야 제자리로 갈 수 있다.
하지만 Shell 정렬은 요소들이 멀리 떨어진 위치로도 이동할 수 있다. 따라서 교환되는 요소들이 삽입 정렬보다 최종위치에 있을 가능성이 높아진다. 간격이 1이 되게 되면 Shell은 기본적으로 삽입 정렬을 수행하는 것이지만 어느 정도 정렬이 된 상태이기 때문에 Insertion Sort보다 더욱 빠르게 수행되는 것이다.

Insertion Sort와 마찬가지로 increase는 이미 정렬되어 있으므로 가장 빠르고, decrease는 역정렬 되어 있으므로 가장 느리다.

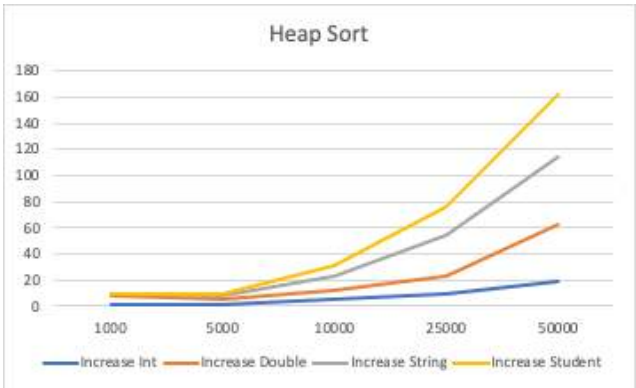
■ Heap sort

> 자료형별 정렬시간 (random, increase, decrease 순서)

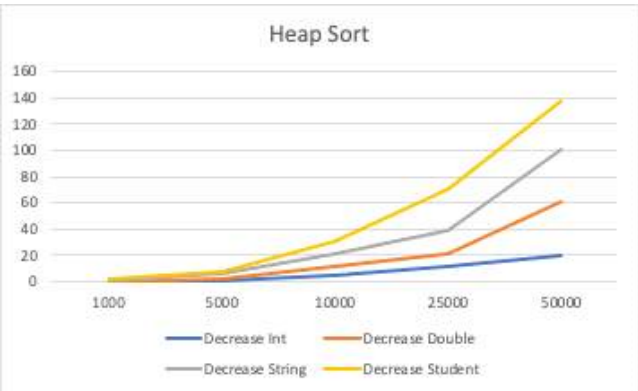
	Int	Double	String	Student
1000	5	1	4	1
5000	11	10	20	5
10000	16	33	19	15
25000	52	28	45	32
50000	63	55	90	48



	Int	Double	String	Student
1000	1	7	1	1
5000	2	3	3	2
10000	6	6	11	8
25000	10	13	32	21
50000	19	44	51	48

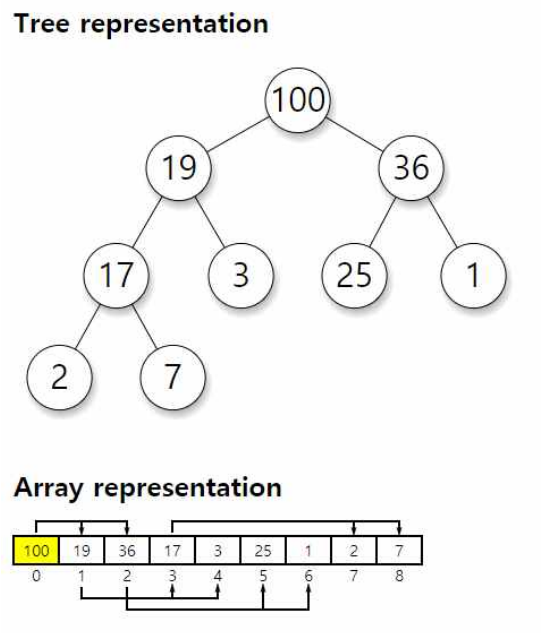


	Int	Double	String	Student
1000	0	0	1	1
5000	1	1	4	2
10000	5	7	10	9
25000	12	10	17	31
50000	20	41	39	37



> 분석

Heap Sort는 정렬할 배열을 먼저 최소 힙으로 변환한 다음, 가장 작은 원소부터 차례대로 추출하여 정렬하는 방법이다. 힙은 완전 이진 트리이므로 1차원 배열을 이용하여 기술할 수 있다.



- heap 구조를 사용하기 위한 배열 구조

Heap Sort의 수행 시간을 분석해보면 먼저 최대 힙을 구성하는 데는 O(N) 시간이 걸린다.
루트와 힙의 마지막 노드를 교환한 후 downheap()을 수행하는 데 O(logN)시간이 걸리고, 루트와 힙의 마지막 노드를 교환하는 횟수는 N-1번이므로 총 수행시간은 O(N) + (N-1) * O(logN) = O(NlogN)이다.
Heap Sort는 입력에 민감하지 않으므로 최선, 평균, 최악의 경우 모두 O(NlogN) 시간으로 빠르게 정렬이 가능하다.

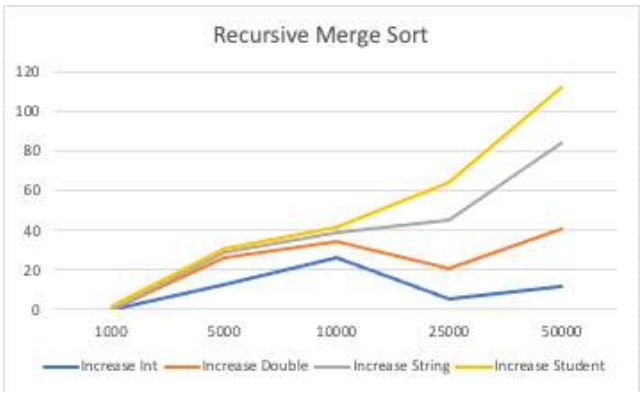
Heap Sort는 전체 리스트 중 일부만 정렬할 필요가 있는 경우 매우 유용하다. 또한, Heap Sort의 장점은 최악의 경우에도 시간 복잡도가 O(NlogN)으로 제한되고 최소 힙이 배열로 구현되기 때문에 별도의 메모리가 필요없다는 점이다.

■ Recursive Merge sort

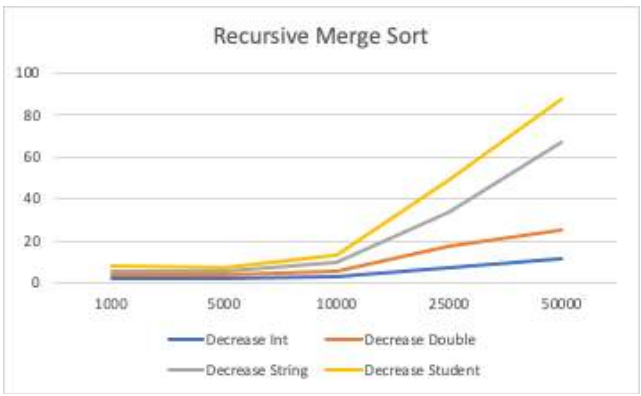
> 자료형별 정렬시간 (random, increase, decrease 순서)

	Int	Double	String	Student
1000	1	0	1	1
5000	6	2	8	19
10000	12	11	55	51
25000	89	114	82	18
50000	93	114	102	37

	Int	Double	String	Student
1000	0	0	0	2
5000	13	13	3	2
10000	26	8	5	3
25000	5	16	24	19
50000	12	29	43	28



	Int	Double	String	Student
1000	2	2	2	2
5000	2	2	2	1
10000	3	3	4	3
25000	7	11	16	15
50000	12	13	42	21



> 분석

Merge Sort는 크기가 N인 리스트를 2개의 리스트로 분할하고, 각각에 대해 재귀적으로 합병정렬을 수행한 후, 2개의 각각 정렬된 부분을 합병하는 알고리즘으로 분할 정복 알고리즘의 하나이다.

Merge Sort는 다음의 세 단계들로 이루어진다.
리스트의 길이가 1이하이면 이미 정렬된 것으로 본다. 그렇지 않은 경우에는
1) Divide : 정렬되지 않은 리스트를 절반으로 잘라 비슷한 크기의 두 부분 리스트로 나눈다.
2) Conquer : 각 부분 리스트를 재귀적으로 Merge Sort를 이용해 정렬한다.
3) Combine : 두 부분 리스트를 다시 하나의 정렬된 리스트로 합병한다. 이때 정렬 결과가 임시배열에 저장된다.

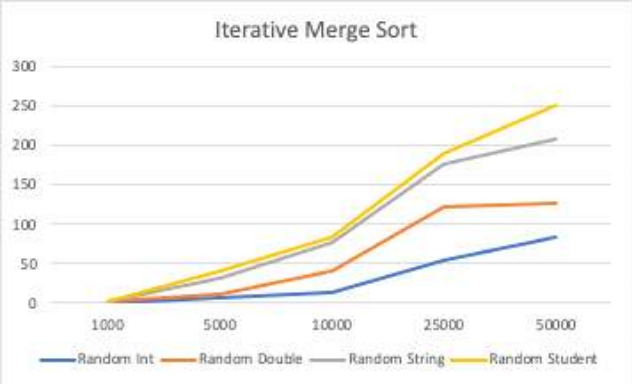
$n=2^k$라고 할 때, 순환 호출의 깊이는 k가 된다. 이때, $k = \log_2 N$가 된다.
하나의 합병 단계에서는 최대 N번의 비교연산이 필요하고 이러한 합병 단계가 $k = \log_2 N$번만큼 있으므로 총 비교 연산은 최대 $N\log_2 N$번이다. 따라서 Merge Sort의 시간복잡도는 O(NlogN)이다.

입력에 민감하지 않아 최선, 평균, 최악의 경우 모두 O(NlogN) 시간이 보장된다.
Quick Sort도 분할 정복으로 정렬을 하지만 최악의 경우 $O(N^2)$의 시간 복잡도를 가지는 반면, Merge Sort는 최악의 경우에도 O(NlogN)의 시간복잡도를 가진다.
다만 Merge Sort를 배열로 구현할 땐 임의의 별도 배열이 필요하다는 단점이 있다.

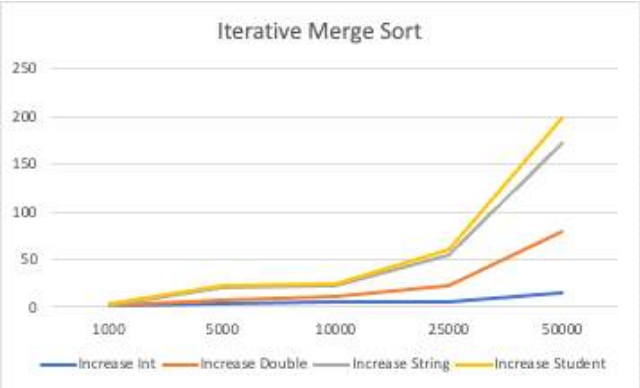
■ Iterative Merge sort

> 자료형별 정렬시간

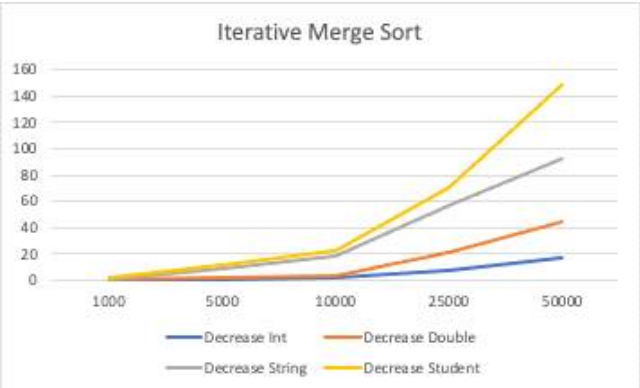
	Int	Double	String	Student
1000	1	1	1	0
5000	6	6	19	9
10000	13	28	37	6
25000	55	67	55	13
50000	84	44	81	43



	Int	Double	String	Student
1000	1	0	1	1
5000	3	5	12	3
10000	5	7	11	2
25000	6	16	32	7
50000	15	65	92	27



	Int	Double	String	Student
1000	0	1	0	1
5000	1	1	7	3
10000	2	2	14	5
25000	8	13	36	14
50000	17	27	49	56



> 분석

Iterative Merge Sort는 상향식 (bottom up) 합병 정렬이라고 하는데, 상향식 합병 정렬은 작은 서브 배열을 여러 개 정렬해 놓고, 점점 서브 배열의 수를 반씩 줄여 나가는 방법이다.

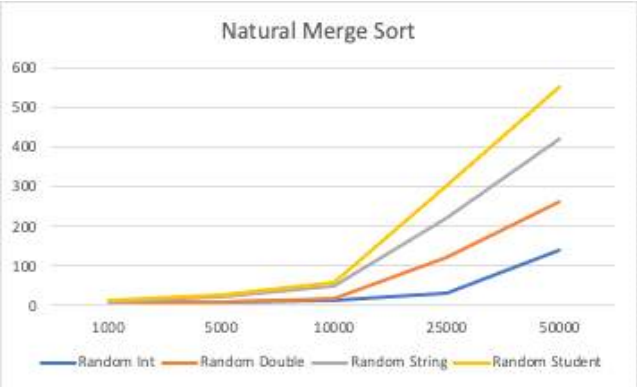
입력 리스트를 길이가 1인 n개의 정렬된 서브 리스트로 간주하는 것으로 시작한다. 첫 번째 합병 단계에서는 이 리스트들을 쌍으로 합병하여 크기가 2인 n/2개의 리스트를 얻는다. 두 번째 합병 단계에서는 이 n/2개의 리스트를 다시 쌍으로 합병하여 n/4개의 리스트를 얻는다. 한 번 합병할 때마다 서브리스트의 수는 반으로 줄어든다. 합병 단계는 하나의 서브리스트가 남을 때까지 계속한다.

Recursive Merge Sort와 Iterative Merge Sort는 작업 순서만 다를 뿐, 수행 시간은 O(NlogN)으로 동일하다.

■ Natural Merge sort

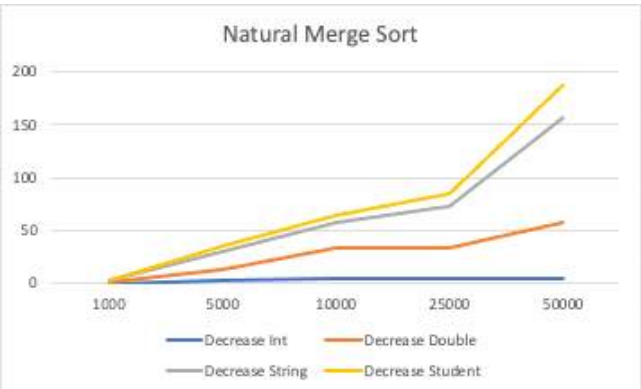
> 자료형별 정렬시간

	Int	Double	String	Student
1000	7	2	1	2
5000	7	4	10	5
10000	12	8	29	8
25000	31	93	99	79
50000	141	124	155	134



	Int	Double	String	Student
1000	1	0	0	0
5000	1	0	1	0
10000	1	0	2	1
25000	1	1	2	2
50000	1	1	3	2

	Int	Double	String	Student
1000	0	1	1	0
5000	3	10	17	6
10000	4	29	24	8
25000	5	29	39	11
50000	5	52	100	31



> 분석

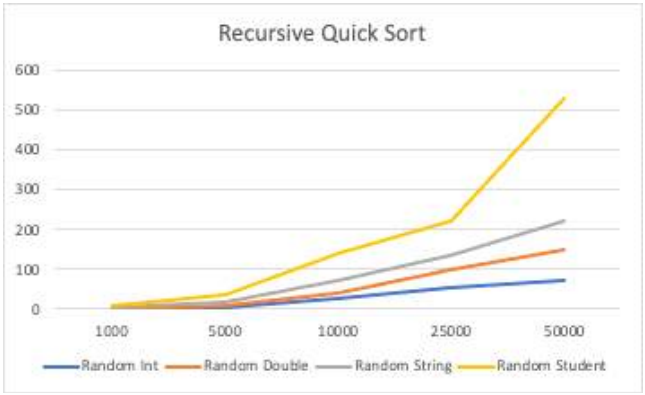
Natural Merge Sort는 bottom-up merge sort의 개선된 알고리즘이다. Natural Merge Sort는 입력 리스트 내에 이미 존재하고 있는 순서를 고려하여 리스트 속에 이미 순서에 맞는 원소로 된 부분 리스트만을 합병한다.

최선의 경우는 이미 정렬된 리스트일 경우로 수행시간은 O(N)이며, 최악의 경우 여전히 O(NlogN)이다.

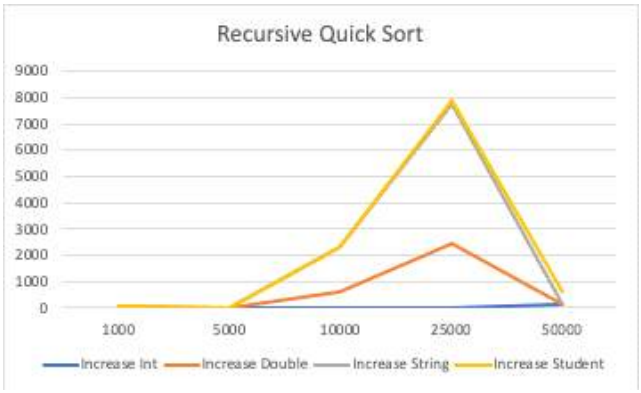
■ Recursive Quick sort

> 자료형별 정렬시간

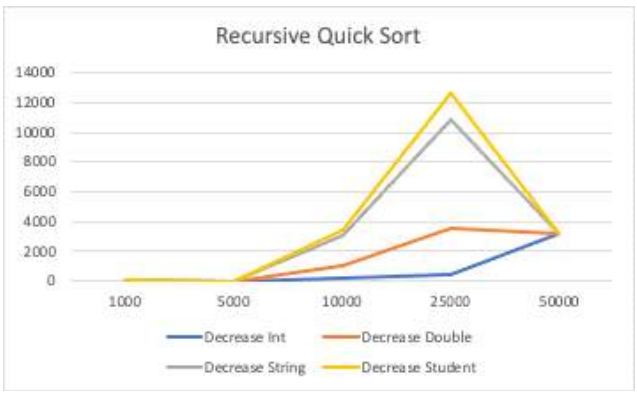
	Int	Double	String	Student
1000	1	1	1	6
5000	6	2	8	19
10000	27	12	35	68
25000	54	47	35	87
50000	72	74	74	306



	Int	Double	String	Student
1000	2	13	33	3
5000	13	13	3	2
10000	11	597	1697	33
25000	26	2432	5275	124
50000	158	측정불가	측정불가	434



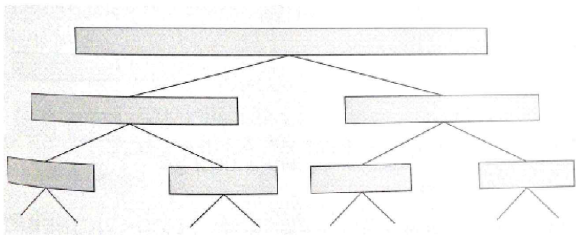
	Int	Double	String	Student
1000	14	25	13	7
5000	2	2	2	1
10000	181	819	2109	361
25000	404	3127	7318	1746
50000	3127	측정불가	측정불가	측정불가



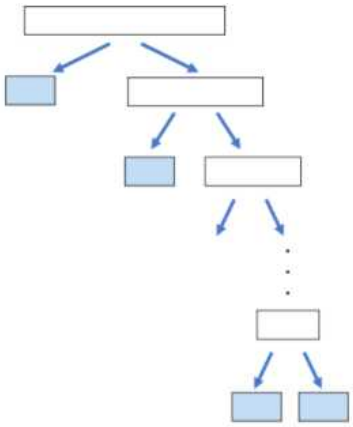
> 분석

Quick Sort는 맨 왼쪽 원소(pivot)을 기준으로 피벗보다 작은 원소들과 큰 원소들을 각각 피벗의 좌우로 분할한 후 피벗보다 작은 원소들과 피벗보다 큰 원소들을 재귀적으로 정렬하는 알고리즘이다. 분할 정복 알고리즘의 하나로, 평균적으로 매우 빠른 수행 속도를 가진다.

레코드의 개수 n이 2의 거듭제곱이라고 가정했을 때, $n = 2^3$ 의 경우, $2^3, 2^2, 2^1, 2^0$ 순으로 줄어들어 순환 호출의 깊이가 3임을 알 수 있다. 이것을 일반화하면 n=2^k의 경우, k=$\log_2 n$이다.
각 순환 호출에서는 전체 리스트의 대부분의 레코드를 비교해야 하므로 평균 n번 정도의 비교가 이루어진다. 따라서 최선의 시간복잡도는 순환 호출의 깊이 * 각 순환 호출 단계의 비교 연산 = $O(nlog_2 n)$ 가 된다. 레코드의 이동횟수는 비교횟수보다 적으므로 무시할 수 있다.



- 퀵 정렬에서의 최선의 경우



- 퀵 정렬에서의 최악의 경우

최악의 경우는 오른쪽 그림처럼 리스트가 계속 불균형하게 나누어지는 것이다. 이미 정렬된 리스트이거나 역정렬된 리스트일 경우 피벗이 매번 가장 작거나 가장 크게 되므로 리스트가 계속 불균형하게 나누어진다. 레코드의 개수 n이 2의 거듭제곱이라고 가정했을 때, 순환 호출의 깊이는 n이다. 각 순환 호출 단계의 비교 연산은 n이므로 최악의 시간 복잡도는 O(n^2)이다.

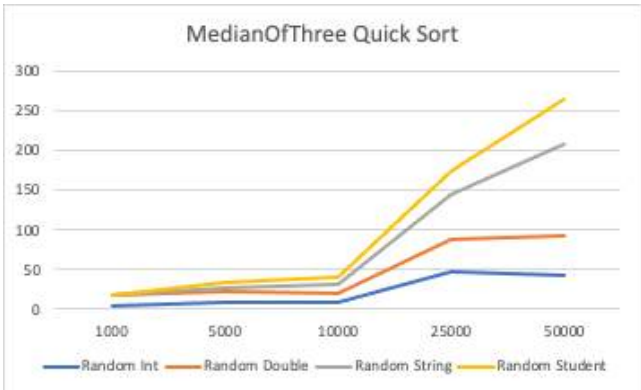
평균적인 경우의 시간 복잡도는 $O(nlog_2n)$으로 나타난다. 특히 다른 $O(nlog_2n)$의 정렬 알고리즘과 비교하였을 때도 가장 빠른 것으로 나타난다. 이는 quick sort가 불필요한 데이터의 이동을 줄이고 먼 거리의 데이터를 교환할 뿐만 아니라, 한 번 정렬된 피벗들이 추후 연산에서 제외되는 특성 등에서 기인한다고 보인다.

quick sort는 속도가 빠르고 추가 메모리 공간을 필요로 하지 않는 장점이 있는 반면 정렬된 리스트에 대해서는 오히려 수행시간이 더 오래 걸리는 단점이 있다. 이러한 불균형을 방지하기 위해 피벗을 선택할 때 더욱 리스트를 균등하게 분할할 수 있는 데이터를 선택한다. 예를 들어, Median of Three Quick Sort는 퀵 정렬의 불균형을 해결해 줄 수 있는 방법이다.

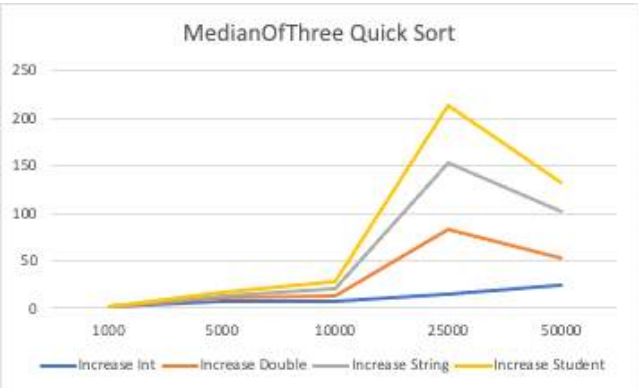
■ MedianOfThree Quick sort

> 자료형별 정렬시간

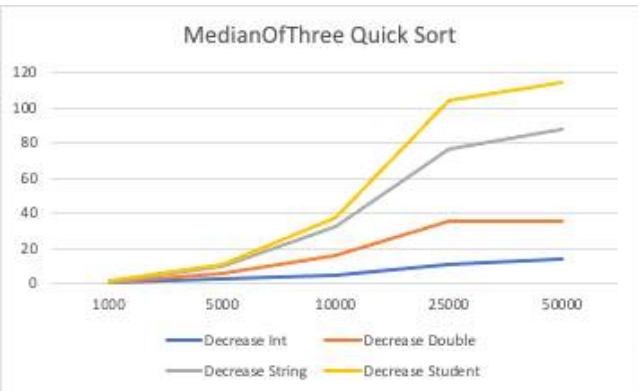
	Int	Double	String	Student
1000	5	13	1	0
5000	8	14	5	6
10000	8	12	11	10
25000	47	41	56	30
50000	44	50	114	57



	Int	Double	String	Student
1000	1	0	0	1
5000	8	3	2	4
10000	8	6	7	7
25000	16	67	69	62
50000	24	28	50	30



	Int	Double	String	Student
1000	1	0	0	1
5000	3	3	4	1
10000	5	11	16	6
25000	11	24	41	28
50000	14	21	53	26



> 분석

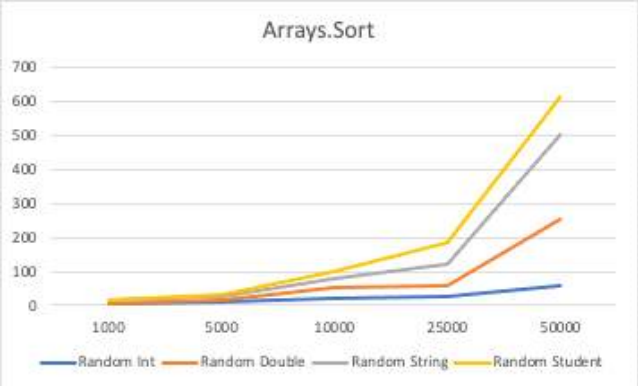
Quick Sort는 피벗의 값에 따라 분할되는 두 영역의 크기가 결정되므로 리스트의 불균형이 일어나는 것을 방지하기 위해 3개의 값을 랜덤하게 선택하여 그 중 중간값을 피벗으로 사용하여 성능을 개선할 수 있다. 3개의 값 중 중간값을 선택하므로 Median-of-Three Quick Sort라 한다.

Quick Sort는 이분된 리스트의 크기가 비슷해야 O($NlogN$)의 좋은 성능을 낼 수 있는데 이 알고리즘은 리스트의 불균형을 막았으므로 단순 Quick Sort보다 더 좋은 수행시간을 가진다.

■ Arrays.sort

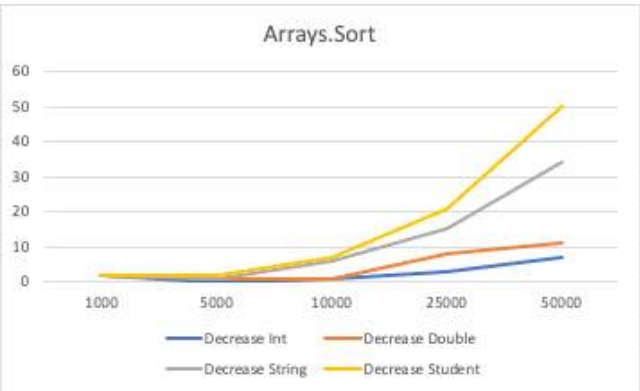
> 자료형별 정렬시간

	Int	Double	String	Student
1000	5	2	7	2
5000	9	8	8	6
10000	22	29	27	24
25000	29	31	60	63
50000	60	192	249	111



	Int	Double	String	Student
1000	0	0	0	0
5000	0	0	1	1
10000	0	1	1	1
25000	0	10	5	1
50000	1	4	7	3

	Int	Double	String	Student
1000	2	0	0	0
5000	0	1	0	1
10000	1	0	5	1
25000	3	5	7	6
50000	7	4	23	16



> 분석

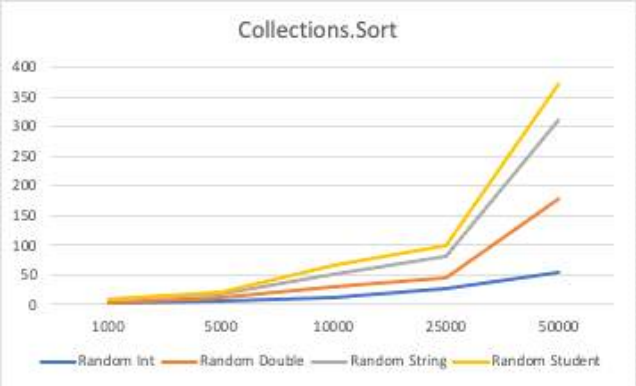
Arrays.sort는 primitive type을 정렬하는 경우와, 그렇지 않은 경우에 쓰이는 알고리즘이 다르다. primitive type인 경우 DualPivotQuicksort.sort()가, Object type인 경우 TimSort.sort()가 호출된다. 이 보고서에선 Object type을 정렬하였으므로 TimSort.sort()가 호출되었을 것이다.

Tim Sort는 Binary Insertion sort와 Merge sort를 결합하여 만든 정렬이다. 이미 정렬되어 있는 최상의 경우 성능은 O(N)이고, 평균적인 경우와 하나도 정렬되지 않은 입력을 받는 최악의 경우 성능조차 O(NlogN)을 보장하는 매우 빠른 알고리즘이다.
또한, TimSort는 stable한 정렬 방법이며 추가 메모리는 사용하지만 기존의 Merge Sort에 비해 적은 추가 메모리를 사용하여 다른 O(nlogn) 정렬 알고리즘의 단점을 최대로 극복한 알고리즘이다.

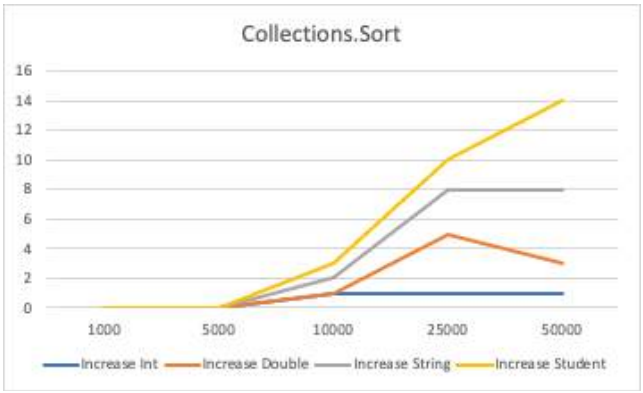
■ Collections.sort

> 자료형별 정렬시간

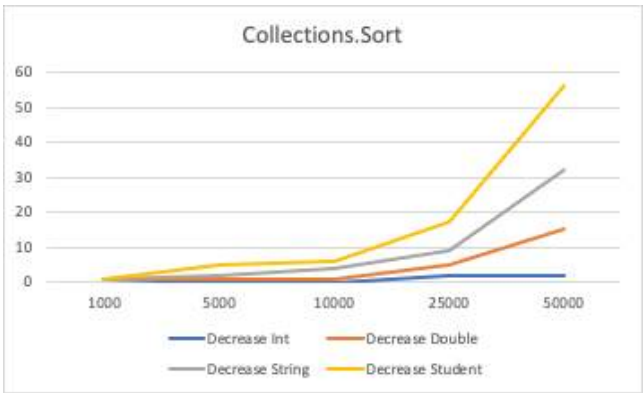
	Int	Double	String	Student
1000	2	2	4	2
5000	6	5	6	3
10000	13	16	23	13
25000	26	19	37	17
50000	55	123	133	61



	Int	Double	String	Student
1000	0	0	0	0
5000	0	0	0	0
10000	1	0	1	1
25000	1	4	3	2
50000	1	2	5	6



	Int	Double	String	Student
1000	1	0	0	0
5000	0	1	1	3
10000	0	1	3	2
25000	2	3	4	8
50000	2	13	17	24



> 분석

Collections.sort()는 내부 구현을 보면 List를 Array로 변경한 뒤 merge sort를 변형한 TimSort를 사용한다고 한다. 시간 복잡도는 O(nlogn)으로 Arrays.sort()와 동일하다,
하지만 Collections.sort()는 T[](Boxed)의 배열의 형태를 가지고 있기 때문에 unwrap한 뒤 재정렬하므로 오버헤드가 발생해 느리지만 stable하다는 장점이 있다.

<3> 비교 vs. 비비교 정렬 비교

* random unsigned int 배열

1) modular연산으로 16진수 LSB 값을 얻어 정렬하는 radix sort
2) masking(&)과 shift연산(>>)으로 16진수 LSB값을 얻어 정렬하는 radix sort
3) modular연산으로 10진수 LSB값을 얻어 정렬하는 radix sort
4) median of three quick sort

```
1. radix sort 수행 시간(1)(ms) : 10
2. radix sort 수행 시간(2)(ms) : 7
3. radix sort 수행 시간(3)(ms) : 11
4. median of three quick sort 수행 시간(ms) : 21
```

(n = 50,000, d = 5)

radix sort는 기수 별로 비교 없이 수행하는 정렬 알고리즘이다.
LSD radix sort의 수행시간은 O(d(N+R)) ≈ $O(n)$이다. 여기서 d는 키의 자리 수이고, R은 radix이며, N은 입력의 크기이다.
median of three quick sort는 O(nlogn)의 성능을 내는 알고리즘이다. 정렬 알고리즘의 한계는 O(nlogn)이지만, radix sort는 이 한계를 넘어서 수 있는 알고리즘이다.
단, LSD radix sort는 제한적인 범위 내에 있는 숫자(문자)에 대해 좋은 성능을 보인다.