정렬 과제 보고서

컴퓨터소프웨어공학과
20180661 안상근
20180670 안철민

목 차

1. 테스트 환경

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

PC
-OS : Microsoft Windows 10 Home
-컴파일러 : JDK javac
-cpu : Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1800Mhz, 4 코어, 8 논리 프로세서
-ram : 8GB

성능 비교 방법
-정렬에 사용되는 데이터는 정수, 실수, 문자열, 클래스(Student) 데이터 타입을 사용한다.
-위 데이터 타입을 랜덤상태, 오름차순, 내림차순으로 구성된 배열을 대상으로 한다.
-위 데이터를 사용하는 배열은 각 1천, 5천, 1만, 2만5천, 5만의 크기를 가진다.
(단, QuickSort는 3만보다 크기가 큰 배열을 사용하여 오름차순, 내림차순으로 구성된 배열을 대상으로 에러가 발생하여 3만의 기준으로 한다.)

2. stable 정렬 vs. non-stable 정렬 분석

stable 정렬 vs. non-stable 정렬

stable 정렬 : 동일한 값이 배열에 포함된 경우 정렬된 결과에서 동일한 값이 원래 배열의 순서대로 저장되는 정렬방식

non-stable 정렬 : 동일한 값이 배열에 포함된 경우 정렬된 결과에서 동일한 값이 원래 배열의 순서대로 저장되지 않는 정렬방식

stable 정렬 분석

(1) insertion sort

배열을 정렬된 부분과 정렬되지 않은 부분으로 나누어서 정렬되지 않은 가장 왼쪽값을 정렬된 부분의 오른쪽값과 차례대로 비교하는 정렬 알고리즘이기 때문에 같은 값이 있다면 swap이 되지 않아서 항상 stable 한 결과가 나온다.

(2) merge sort

-합병정렬

배열을 크기가 1이 될 때가지 배열을 절반으로 쪼개서 원래 배열의 가장 왼쪽에 있던 원소와 가까운 배열과 합병을 통해서 정렬을 진행하는데 값이 같은 경우 swap이 일어나지 않으므로 항상 stable한 결과가 나온다.

-반복합병정렬

배열의 크기가 모두 1이 되도록 쪼개고 인접한 원소를 2개, 4개, ... N개가 될 때까지 합병을 통해서 정렬을 진행하여서 합병정렬과 마찬가지로 값이 같은 경우 swap이 일어나지 않으므로 항상 stable 한 결과가 나온다.

-자연합병정렬

정렬된 경우의 반복합병정렬에서 배열의 원소가 정렬된 상태인 경우 배열을 쪼개지 않고 그대로 사용한다. 반복합병정렬과 마찬가지로 값이 같은 경우 swap이 일어나지 않는다.

(3) radix sort sort

기수 정렬은 키 값이 숫자 일 때 사용하며, 각 자릿수에 대해 키를 비교하는 정렬 알고리즘이다.

특정한 자릿수에 대해 키를 비교 한 경우에 키의 값이 같은 경우는 이미 배열에서 정렬된 상태이다. 그러므로 stable한 정렬이 아닌 경우 자릿수를 정렬하다 보면 정렬에 의해서 본래 입력 받은 숫자가 변할 수 있기 때문에 Stable Sort를 사용해야 한다.

그러므로 기수정렬은 stable한 정렬이 된다.

non-stable 정렬

예제 경우 Before Sort : 정렬 전 / After Sort : 정렬 후 / [값 , 삽입된 순서]

(1) selection sort
non-stable 한 예제

```
Before Sort
[1,0][2,1][0,2][2,3][2,4][0,5][1,6][0,7][1,8][1,9]
After Sort
[0,2][0,5][0,7][1,6][1,0][1,8][1,9][2,4][2,1][2,3]
```

키 값이 2의 경우 들어온 원소의 들어온 순서가 달라져있는 것을 볼 수 있다.

선택정렬의 경우 배열을 정렬된 부분과 정렬되지 않은 부분으로 나누어서 정렬되지 않은 부분에서 최솟값을 선택하여 정렬된 부분의 오른쪽에 원소와 swap하는 형식으로 정렬을 하는 과정에서 삽입한 순서가 달라 질 수 있다.

배열이 정렬이 되어 있는 상태라면 swap을 실행하는 과정이 없기 때문에 삽입한 순서에 따라서 배열이 정렬되어 있는 것을 알 수 있다.

```
Before Sort
[0,0][0,1][2,2]
After Sort
[0,0][0,1][2,2]
```

그러므로 선택정렬은 stable한 정렬 알고리즘이 아님을 알 수 있다.

(2) shell sort
non-stable 한 예제

```
Before Sort
[1,0][1,1][0,2][1,3][1,4][0,5][0,6][2,7][0,8][2,9]
After Sort
[0,8][0,5][0,2][0,6][1,3][1,0][1,1][1,4][2,7][2,9]
```

키 값이 0, 1 경우 들어온 키 값의 순서가 달라져있는 것을 볼 수 있다.
쉘 정렬에서 간격을 줄여가며 정렬을 수행하는데 삽입정렬이 되는 간격 1인 경우를 제외하고 간격에 따라 정렬을 실행하여 swap하는 경우가 발생하기 때문에 non-stable한 정렬이 된다.

위 예제는 간격이 4인 경우이다. [0, 8]의 경우 [1,4] [1,0] 비교하여 가장 앞으로 오게 되는데 [0,5]의 경우 역시 [1,1]과 비교하여 swap이 일어나게 된다. 결과적으로 [0,8]과 [0,5]는 키 값이 같기 때문에 최종적으로 정렬이 완료된 상태가 된다.
그러므로 쉘 정렬은 non-stable한 정렬이 된다.

(3) quick sort
non-stable 한 예제

```
Before Sort
[0,0][0,1][4,2][1,3][1,4][0,5][0,6][0,7][2,8][3,9]
After Sort
[0,1][0,7][0,6][0,5][0,0][1,3][1,4][2,8][3,9][4,2]
```

키 값이 0인 경우 들어온 원소의 들어온 순서가 달라져 있는 것을 볼 수 있다.

피벗에서 키 값을 기준으로 정렬하는 과정에서 같은 키 값을 가지는 원소 2개와 피벗을 각각 비교할 때, 같은 키 값을 가지는 원소의 들어온 순서와 상관없이 키 값을 기준으로 정렬을 하기 때문에 이러한 결과가 나타나는 것으로 보인다.

그러므로 퀵정렬은 non-stable한 정렬이다.

(4) heap sort
non-stable 한 예제

```
Before Sort
[0,0][2,1][0,2][0,3][0,4][1,5][0,6][0,7][0,8][2,9]
After Sort
[0,0][0,8][0,3][0,2][0,4][0,6][0,7][1,5][2,9][2,1]
```

키 값이 0, 2 인 경우 원소값이 들어온 순서가 달라져있는 것을 볼 수 있다.

최대힙을 구성한 후 최댓값을 삭제하고 해당 값을 힙의 마지막 값과 swap하는 과정이 발생하게 된다. 해당하는 최댓값을 제외하고 남은 원소들로 위의 과정을 반복하여 힙정렬을 수행하게 된다.
원소 [2,1]과 [2,9]를 보면 삽입순서와 상관없이 힙정렬 과정에서는 키 값만을 기준으로 정렬이 되었음을 볼 수 있다.

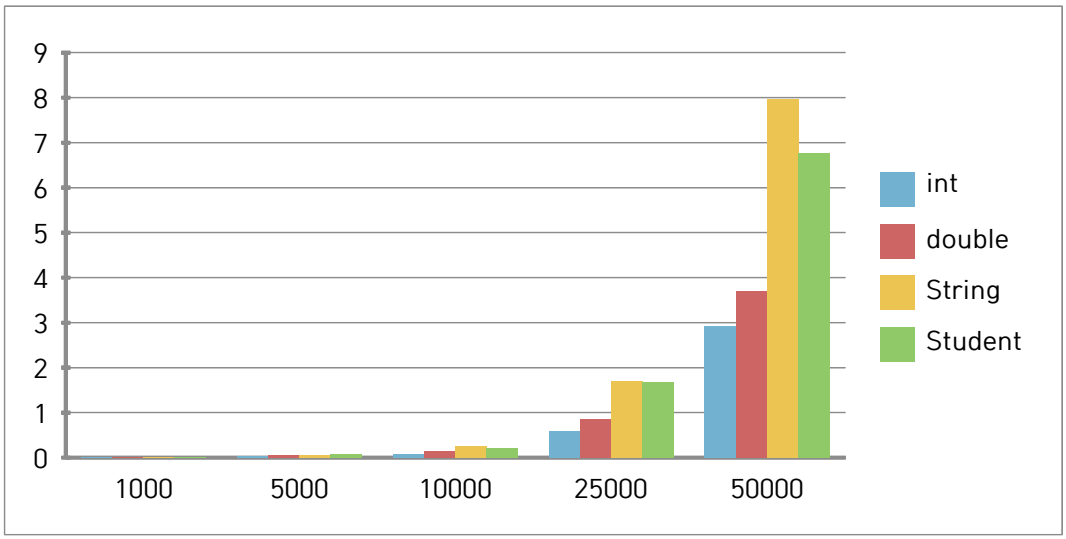
그러므로 힙정렬은 non-stable한 정렬이다.

3. sort 종류, 데이터 타입, 데이터 양에 따른 결과 분석

(1) Insertion Sort

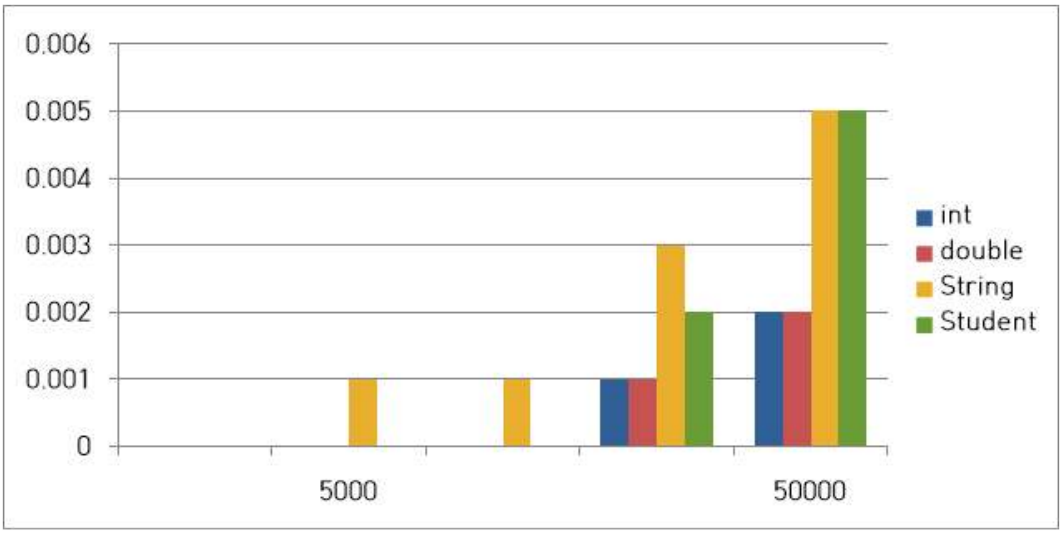
*random order values

	int	double	String	Student
1000	0.008	0.02	0.014	0.01
5000	0.029	0.067	0.051	0.072
10000	0.089	0.138	0.249	0.22
25000	0.588	0.86	1.69	1.682
50000	2.915	3.713	7.956	6.761



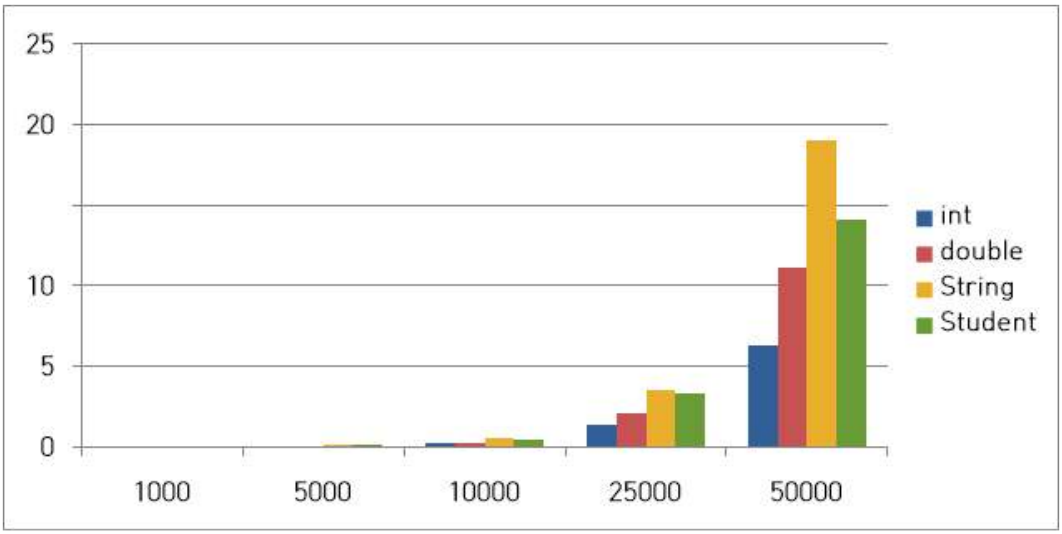
*increase order values

	int	double	String	Student
1000	0.0	0.0	0.0	0.0
5000	0.0	0.0	0.001	0.0
10000	0.0	0.0	0.001	0.0
25000	0.001	0.001	0.003	0.002
50000	0.002	0.002	0.005	0.005



*decrease order values

	int	double	String	Student
1000	0.013	0.012	0.016	0.015
5000	0.068	0.074	0.113	0.115
10000	0.209	0.193	0.527	0.456
25000	1.394	2.12	3.571	3.345
50000	6.244	11.085	19.025	14.084



*분석
insertion sort(삽입 정렬)은 배열이 정렬된 부분과 정렬되지 않은 부분으로 나누어, 정렬된 부분의 바로 우측 값을 정렬된 부분에 정렬되도록 '삽입'하는 방식이다.

Increase 정렬되어있는 경우에는 각 값들 간의 비교만 수행하고 교환은 일어나지 않으므로 최선의 경우로 O(N)의 시간복잡도를 가진다.

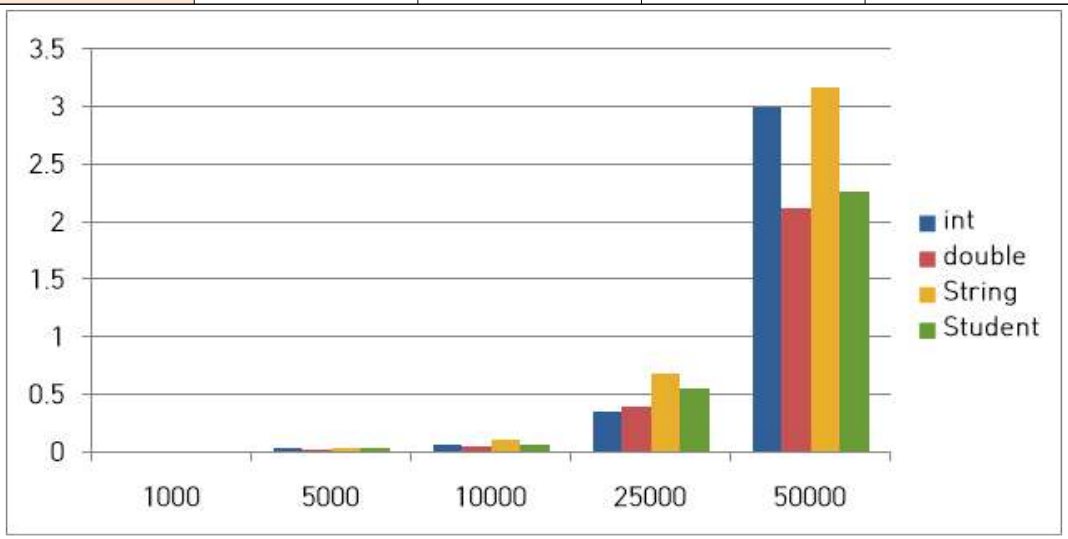
Random 정렬된 경우와 Decrease 정렬된 경우, 이론적으로는 모두 $O(N^2)$의 시간복잡도를 가진다.

수행시간을 보면 Dcrease 정렬된 경우 Random 정렬된 경우의 약2배 정도의 시간의 차이가 나는 것으로 보아 Random 정렬된 경우 평균적으로 정렬된 앞 부분의 배열에 절반 정도인 곳에 삽입하여 이러한 결과가 나타나는 것으로 보인다.

(2) Binary Search insertion sort

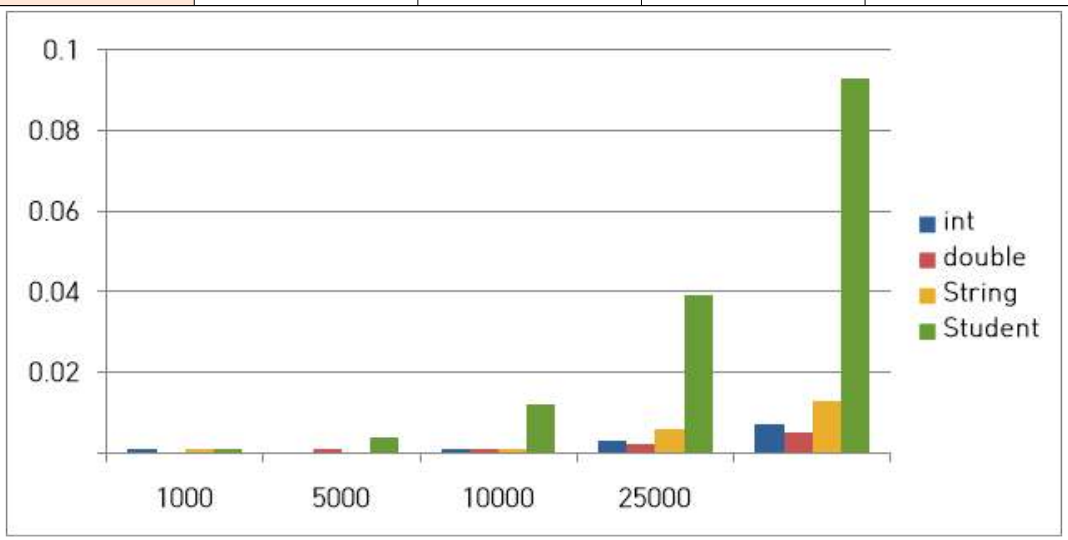
*random order values

	int	double	String	Student
1000	0.004	0.005	0.005	0.005
5000	0.026	0.025	0.027	0.032
10000	0.059	0.053	0.109	0.066
25000	0.351	0.397	0.674	0.549
50000	2.991	2.121	3.174	2.254



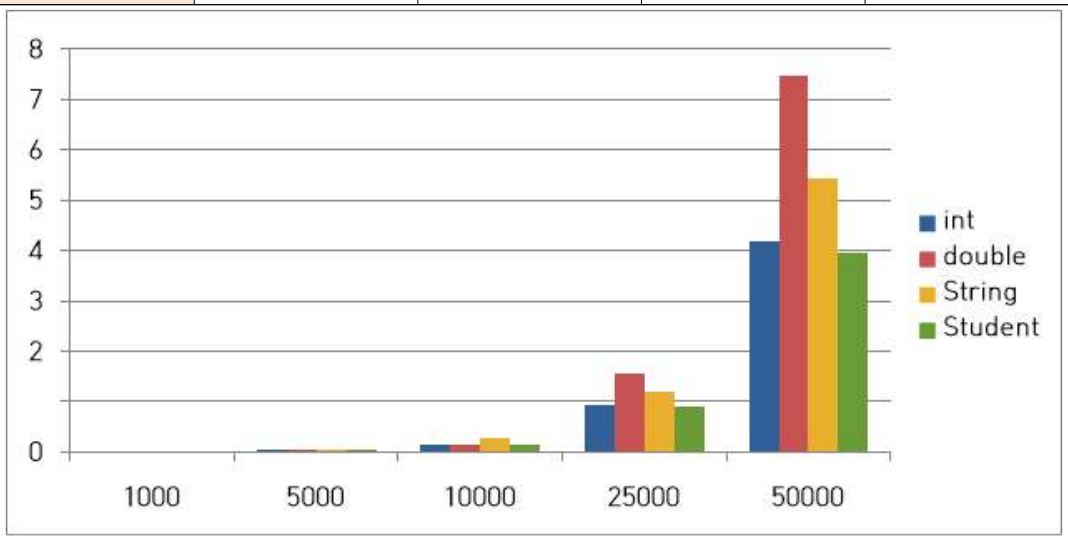
*increase order values

	int	double	String	Student
1000	0.001	0.0	0.001	0.001
5000	0.0	0.001	0.0	0.004
10000	0.001	0.001	0.001	0.012
25000	0.003	0.002	0.006	0.039
50000	0.007	0.005	0.013	0.093



*decrease values order

	int	double	String	Student
1000	0.006	0.008	0.009	0.007
5000	0.031	0.032	0.034	0.032
10000	0.144	0.132	0.256	0.127
25000	0.929	1.553	1.187	0.905
50000	4.182	7.453	5.428	3.94



*분석

삽입 정렬을 개선한 알고리즘으로, 삽입 정렬은 정렬된 부분을 순서대로 따라가면서 값과 비교하는 반면, Binary Search Insertion Sort의 경우에는 정렬된 부분을 Binary Search 하면서 값과 비교하기 때문에 삽입 정렬에 비해 좀 더 빠른 실행 시간을 가질 수 있다.

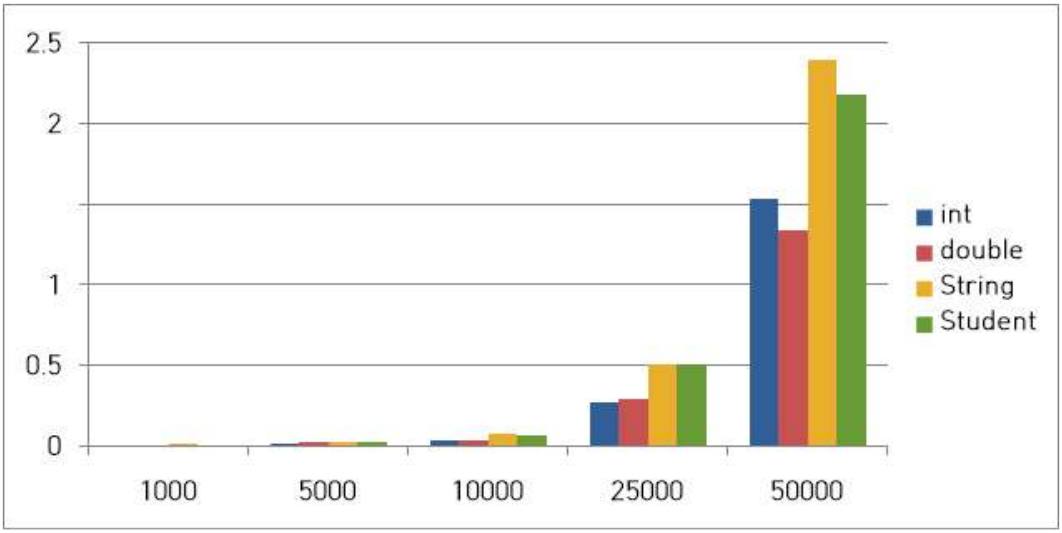
Increase 정렬되어 있는 경우에는 각 값들 간의 비교만 수행하고 교환은 일어나지 않으므로 최선의 경우 Insertion Sort와 같은 O(N)의 시간복잡도를 가진다.

Random 정렬된 경우와 Decrease 정렬된 경우 각 값들 간의 비교 후 교환이 일어나는 경우 정렬된 곳에 삽입해야 하는 상황에서 차례대로 탐색하는 것이 아니라 이진탐색을 이용하여서 정렬된 곳의 모든 값을 확인하지 않으므로 최악의 경우에 O(NlogN)의 시간복잡도를 가지며 두 개의 원소 값을 swap하는데에 $O(N^2)$의 시간복잡도를 가진다. 결론적으로 $O(N^2)$의 시간이 걸린다.

(3) Shell Sort

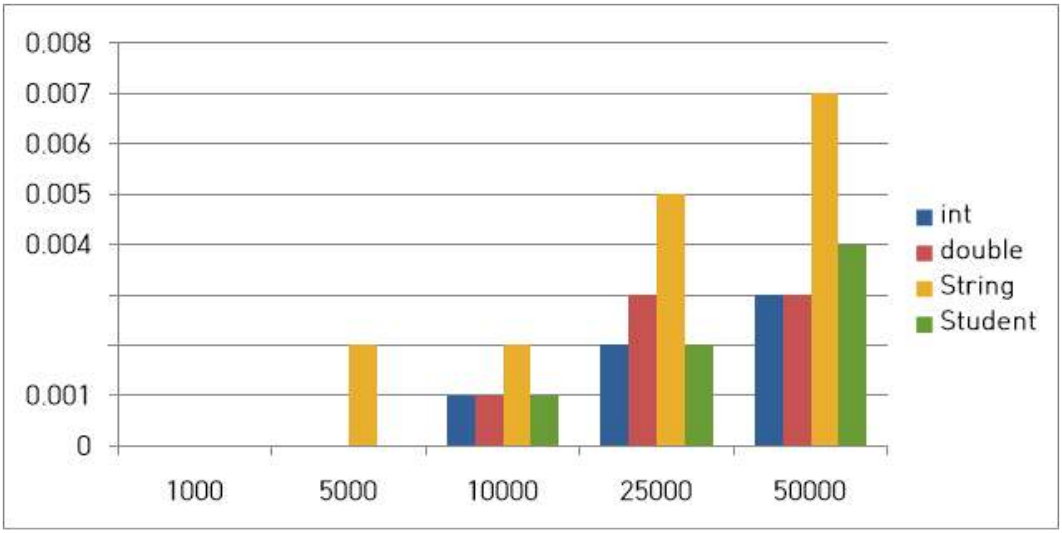
*random order values

	int	double	String	Student
1000	0.006	0.008	0.009	0.006
5000	0.018	0.02	0.024	0.025
10000	0.029	0.033	0.078	0.062
25000	0.275	0.295	0.506	0.511
50000	1.528	1.337	2.396	2.182



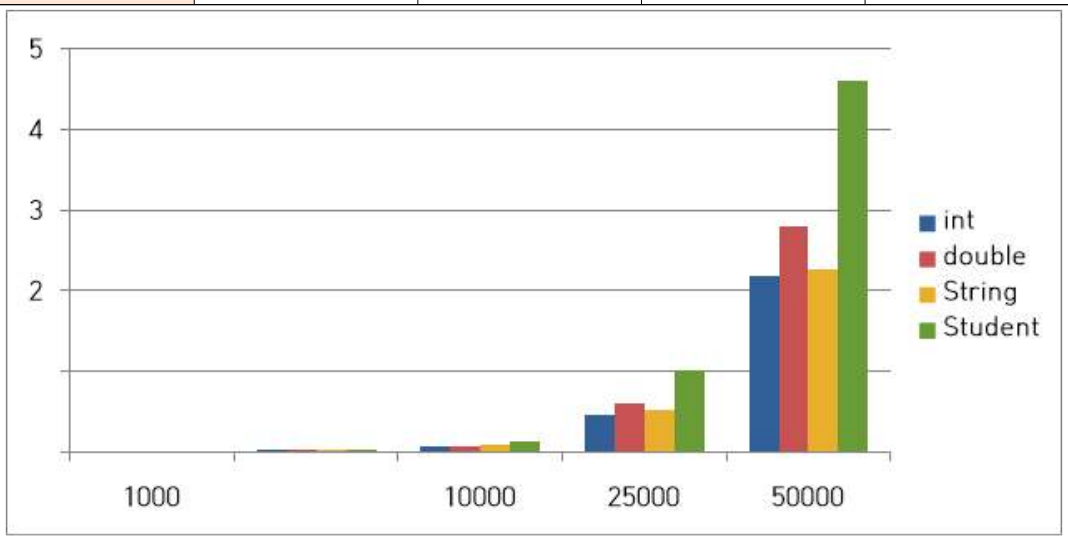
*increase order values

	int	double	String	Student
1000	0.0	0.0	0.0	0.0
5000	0.0	0.0	0.002	0.0
10000	0.001	0.001	0.002	0.001
25000	0.002	0.003	0.005	0.002
50000	0.003	0.003	0.007	0.004



*decrease order values

	int	double	String	Student
1000	0.006	0.008	0.007	0.006
5000	0.023	0.02	0.027	0.034
10000	0.066	0.066	0.079	0.137
25000	0.46	0.593	0.52	1.02
50000	2.189	2.796	2.274	4.608



*분석

정렬할 배열을 일정한 간격마다 비교하여 작은 값을 앞쪽으로, 큰 값을 뒤쪽으로 가도록 한다. 이를 간격을 줄여가며 반복 수행하여 정렬이 되도록 하는 방식이다. 그러나 속도를 개선하기 위해 어느 정도 간격이 작아지면 삽입 정렬을 수행하여 수행 시간을 줄일 수 있다.

간격이 1인 경우 삽입 정렬과 동일하다고 볼 수 있다. 이웃하는 원소끼리 비교하여 한자리씩 이동하는 단점을 보완하여 일정한 간격에 따라 삽입정렬을 수행한다.

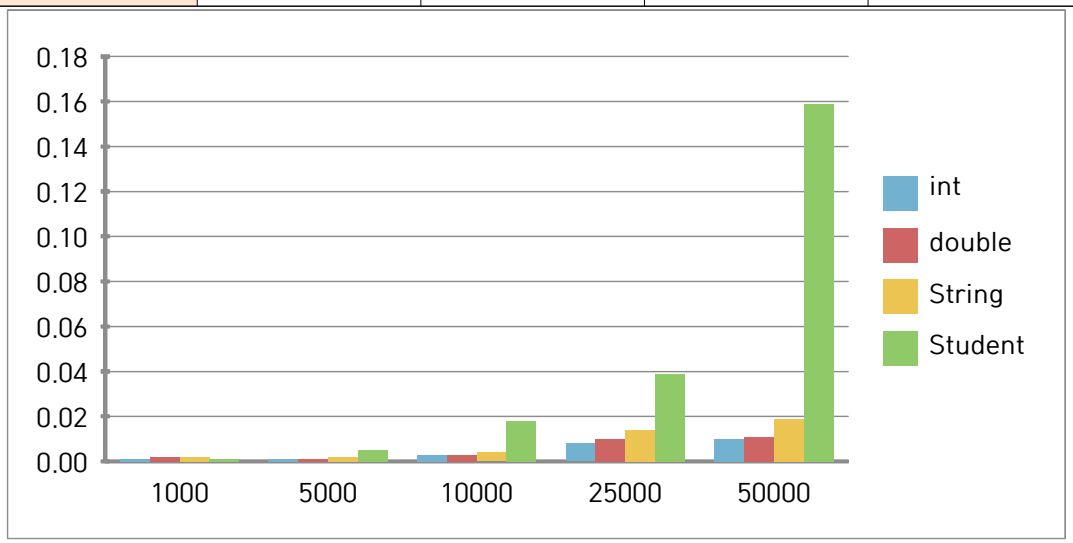
Increase 정렬된 경우 각 값들 간의 비교만 수행하고 교환은 일어나지 않으므로 최선의 경우로 O(N)의 시간복잡도를 가진다.

Random 정렬된 경우 $O(N^{1.5})$의 시간복잡도를 가지며, Decrease 정렬된 경우 $O(N^2)$의 시간복잡도를 가진다.

(4) QuickSort

*random order values

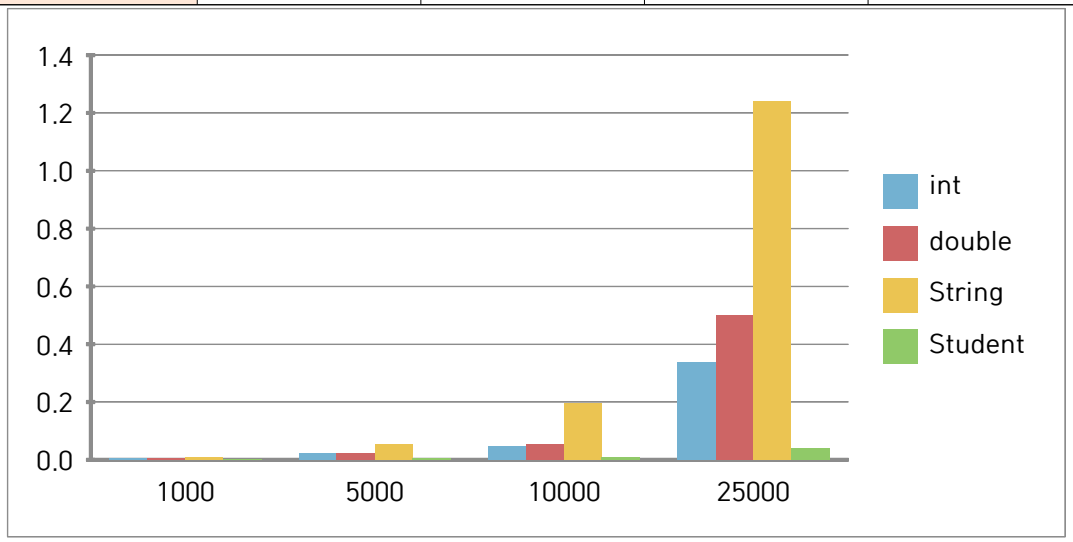
	int	double	String	Student
1000	0.001	0.002	0.002	0.001
5000	0.001	0.001	0.002	0.005
10000	0.003	0.003	0.004	0.018
25000	0.008	0.01	0.014	0.039
50000	0.01	0.011	0.019	0.159



(사용하는 메모리를 초과하는 스택오버플로우로 인해 데이터수가 50000인 경우를 시행하지 못하였음)

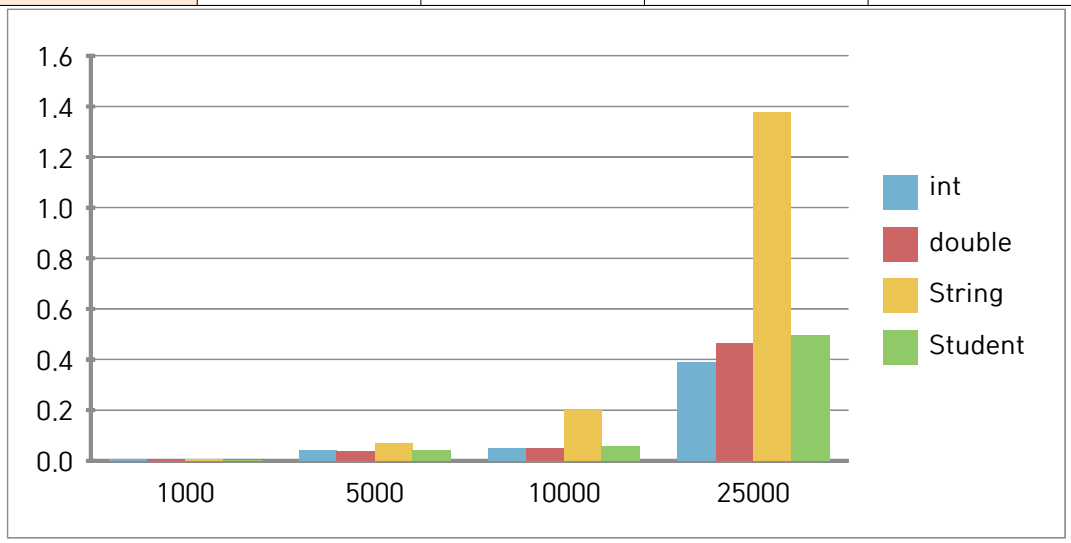
*increase order values

	int	double	String	Student
1000	0.006	0.006	0.008	0.001
5000	0.021	0.022	0.054	0.006
10000	0.047	0.052	0.196	0.01
25000	0.337	0.499	1.24	0.039



*decrease order values

	int	double	String	Student
1000	0.008	0.007	0.008	0.004
5000	0.043	0.038	0.069	0.04
10000	0.049	0.049	0.198	0.058
25000	0.388	0.462	1.376	0.495



*분석

퀵정렬은 피벗을 기준으로 피벗의 왼쪽에는 피벗보다 작은 키 값을 가지는 원소를 피벗의 오른쪽에는 피벗보다 큰 키 값을 가지는 원소를 배치하여 피벗을 제외한 나머지 배열을 재귀적인 방식으로 정렬하는 알고리즘이다.

위 예제는 피벗을 배열에서 가장 왼쪽에 있는 원소의 키 값을 이용하여 정렬을 진행하였다.

Random하게 정렬된 배열에서 피벗의 키 값이 해당 배열의 중간값이 되는 경우로 O(N)의 시간복잡도를 가진다.

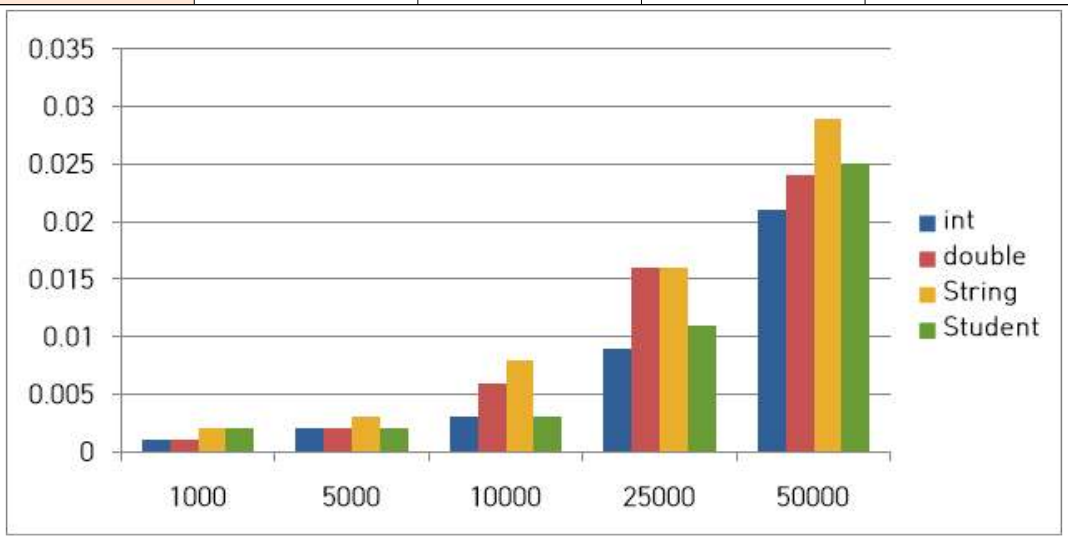
일반적으로 Random하게 정렬된 배열에서는 O(NlogN)의 시간복잡도를 가진다.

Increase 정렬된 배열에서 피벗의 키 값이 항상 다른 원소의 키 값 보다 작기 때문에 피벗에 의해서 분할되는 배열은 항상 피벗과 피벗을 제외한 배열이 남게 되어 재귀적인 방식으로 호출을 할 때 마다 동일한 결과가 나오므로 N-1, N-2, ... 1까지 시행을 하게 된다. 결과적으로 (N - 1) * N / 2 의 시행을 하게 되어서 $O(N^2)$의 시간복잡도를 가진다.

decrease 정렬된 배열역시 피벗의 키 값이 항상 다른 원소의 키 값 보다 크기 때문에 피벗에 의해서 분할되는 배열은 항상 피벗의 왼쪽에 위치하게 되며 Increase하게 정렬된 배열에서와 마찬가지로 $O(N^2)$의 시간복잡도를 가진다.

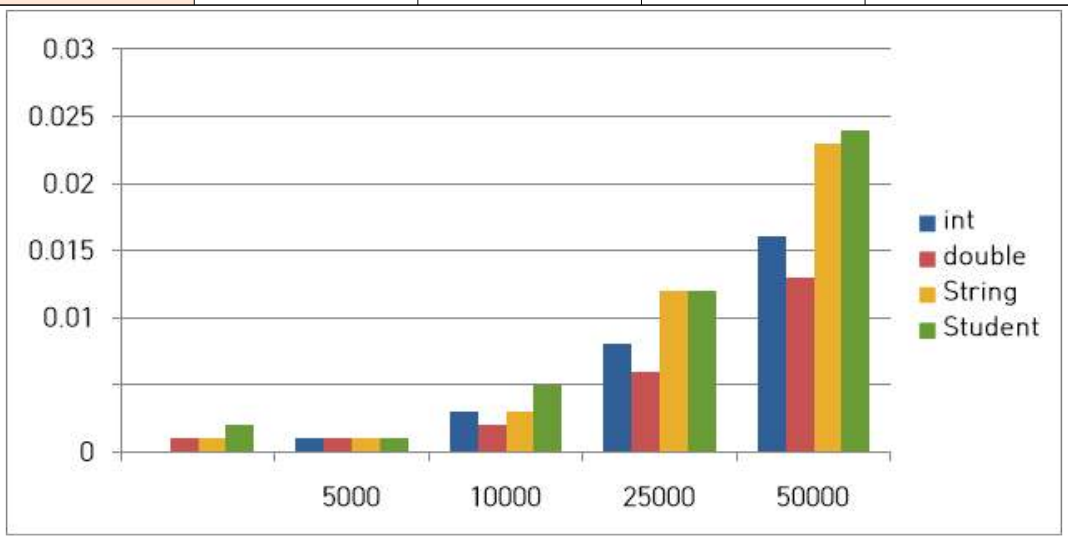
(5) Median Of Three recursive quick

	int	double	String	Student
1000	0.001	0.001	0.002	0.002
5000	0.002	0.002	0.003	0.002
10000	0.003	0.006	0.008	0.003
25000	0.009	0.016	0.016	0.011
50000	0.021	0.024	0.029	0.025



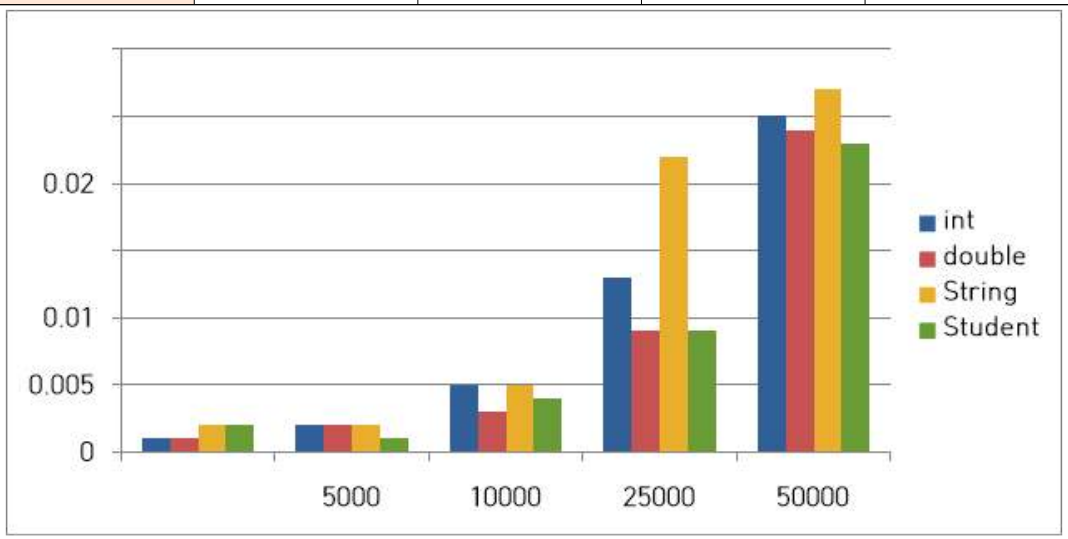
*increase order values

	int	double	String	Student
1000	0.0	0.001	0.001	0.002
5000	0.001	0.001	0.001	0.001
10000	0.003	0.002	0.003	0.005
25000	0.008	0.006	0.012	0.012
50000	0.016	0.013	0.023	0.024



*decrease order values

	int	double	String	Student
1000	0.001	0.001	0.002	0.002
5000	0.002	0.002	0.002	0.001
10000	0.005	0.003	0.005	0.004
25000	0.013	0.009	0.022	0.009
50000	0.025	0.024	0.027	0.023



*분석

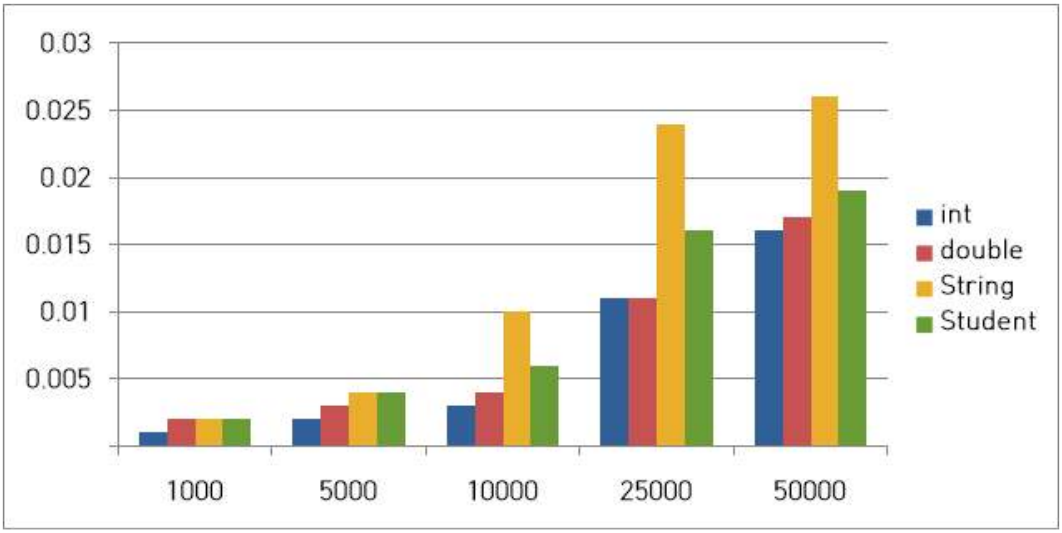
퀵정렬을 개선하여 만들어진 알고리즘으로, 피벗이 배열의 중간값에 가까울수록 퀵정렬의 성능이 높아지는 것을 감안하여 정렬할 배열 중 임의로 3개의 값을 선택한 뒤 선택한 3개의 값 중 중간값을 피벗으로 선택하여 퀵정렬을 수행하는 방식이다. 일반 퀵정렬이 임의의 1개를 피벗으로 선정하여 배열에서의 중간값과 차이가 클 수 있는 것에 비해 madian of three 퀵정렬은 선택한 3개의 값 중 중간값을 피벗으로 하여 일반 퀵정렬에 비해 피벗이 배열의 중간값에 더 가까울 수 있다.

퀵정렬은 배열의 중간값에 더 가까울수록 정렬의 효율이 올라간다. 중간값을 선택하기 위한 방식으로 Median of Three 퀵정렬은 임의로 3개의 값을 선택한 뒤 선택한 3개의 값 중 중간값을 피벗으로 수행하여 배열이 랜덤하거나 오름차순 또는 내림차순으로 정렬이 된 경우에도 유사한 시간 결과를 볼 수 있었다. 시간복잡도는 O(NlogN)이다.

(6) Recursive Merge

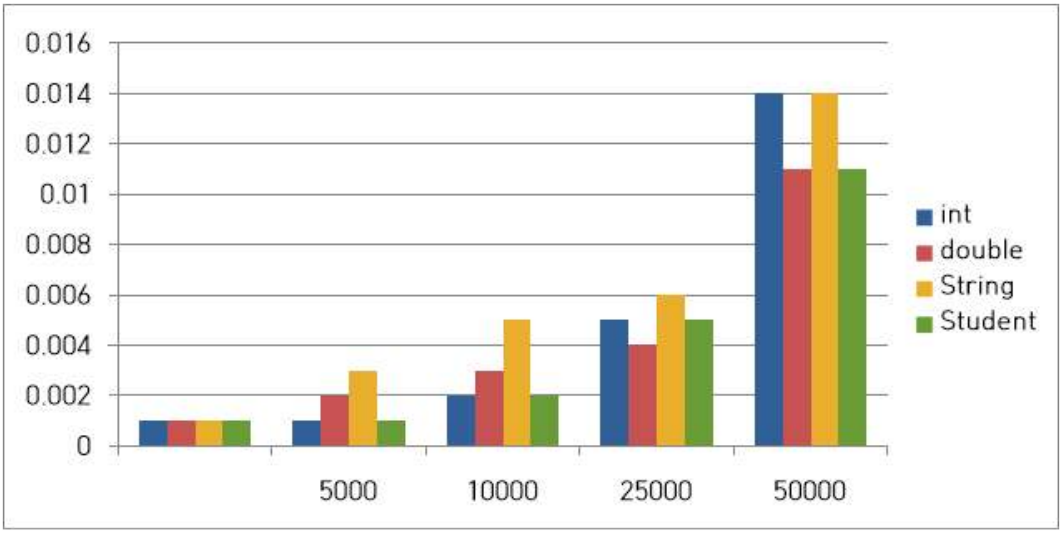
*random

	int	double	String	Student
1000	0.001	0.002	0.002	0.002
5000	0.002	0.003	0.004	0.004
10000	0.003	0.004	0.01	0.006
25000	0.011	0.011	0.024	0.016
50000	0.016	0.017	0.026	0.019



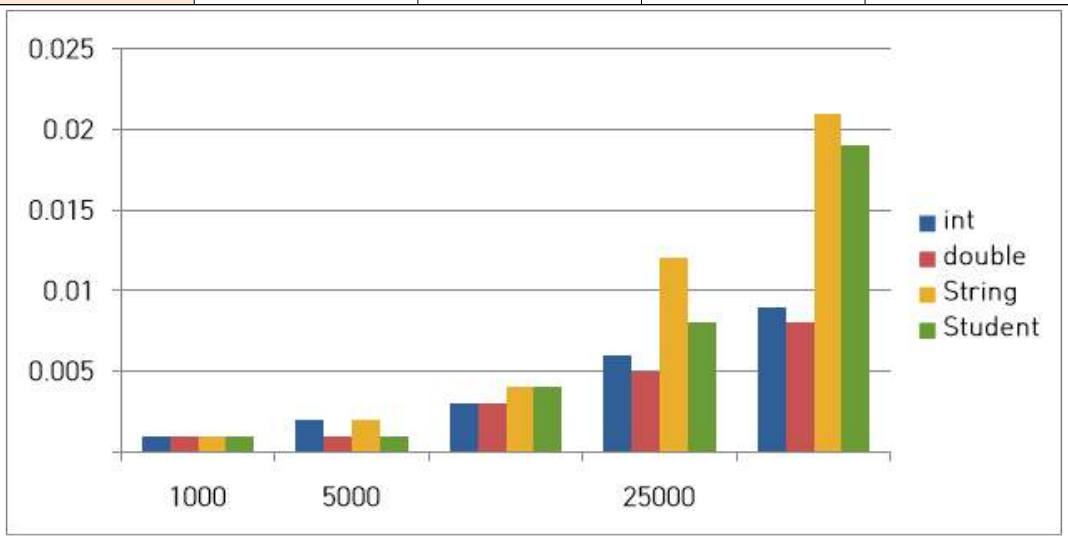
*increase order values

	int	double	String	Student
1000	0.001	0.001	0.001	0.001
5000	0.001	0.002	0.003	0.001
10000	0.002	0.003	0.005	0.002
25000	0.005	0.004	0.006	0.005
50000	0.014	0.011	0.014	0.011



*decrease order values

	int	double	String	Student
1000	0.001	0.001	0.001	0.001
5000	0.002	0.001	0.002	0.001
10000	0.003	0.003	0.004	0.004
25000	0.006	0.005	0.012	0.008
50000	0.009	0.008	0.021	0.019



*분석
합병정렬은 정렬할 배열을 절반으로 나누고, 또 나누어진 것들을 또 절반으로 나누면서 각 부분의 크기가 1이 될 때까지 나눈 후, 나누어진 것들을 정렬하면서 합병하는 방식으로 정렬을 수행한다. 이를 재귀적으로 수행하기 때문에 배열의 앞부분이 먼저 정렬되고, 뒷부분이 정렬된 다음 이 둘을 또 합병 정렬하는 방식으로 수행한다.

배열의 앞부분부터 차례대로 정렬을 수행하기 때문에 stable한 정렬이 된다.

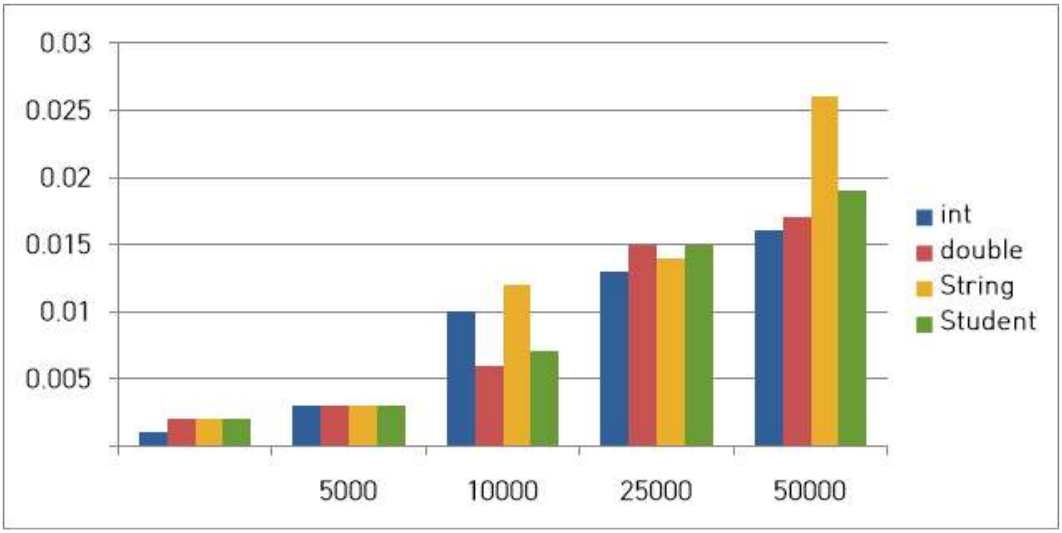
Random하게 정렬된 경우, Increase 하게 정렬된 경우, decrease하게 정렬된 경우에서 모두 비슷한 시간이 소요 되는 것으로 보인다.

배열을 나누는데 O(N)의 시간복잡도, 배열의 재귀호출에 O(logN)의 시간복잡도가 걸리며 원소의 개수가 N이므로 모두 재귀호출을 하는데 걸리는 시간복잡도는 O(NlogN)이 된다.
결과적으로 O(N) + O(NlogN)이 되어서 합병정렬의 시간복잡도는 O(NlogN)이다.

(7) iterative Merge

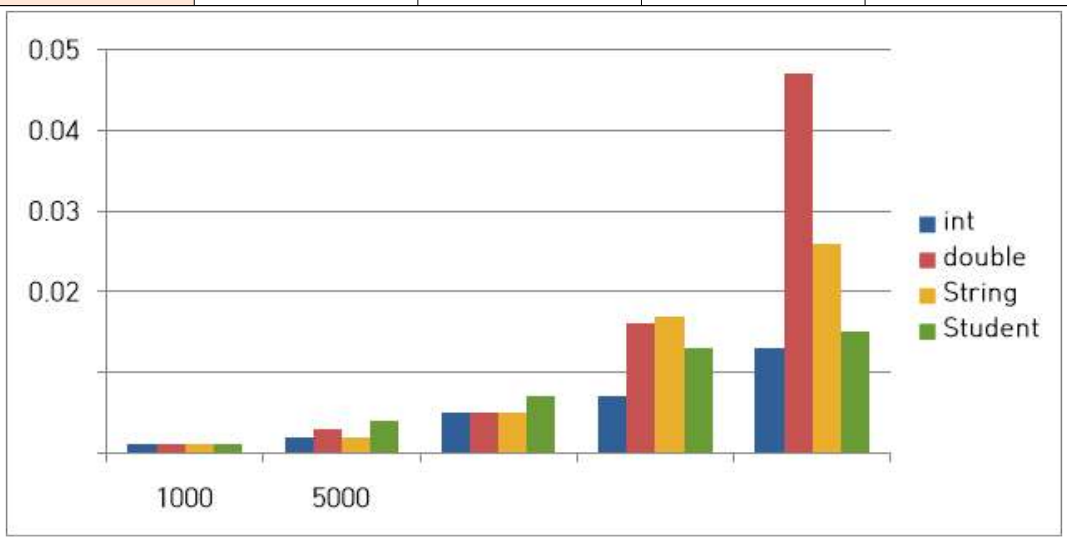
*random order values

	int	double	String	Student
1000	0.001	0.002	0.002	0.002
5000	0.003	0.003	0.003	0.003
10000	0.01	0.006	0.012	0.007
25000	0.013	0.015	0.014	0.015
50000	0.016	0.017	0.026	0.019



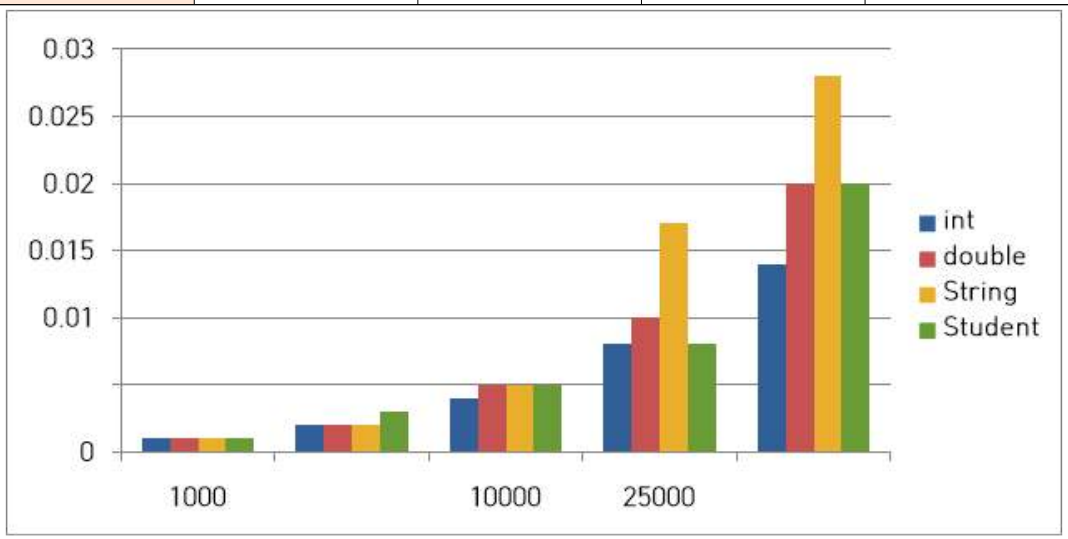
*increase order values

	int	double	String	Student
1000	0.001	0.001	0.001	0.001
5000	0.002	0.003	0.002	0.004
10000	0.005	0.005	0.005	0.007
25000	0.007	0.016	0.017	0.013
50000	0.013	0.047	0.026	0.015



*decrease order values

	int	double	String	Student
1000	0.001	0.001	0.001	0.001
5000	0.002	0.002	0.002	0.003
10000	0.004	0.005	0.005	0.005
25000	0.008	0.01	0.017	0.008
50000	0.014	0.02	0.028	0.02



*분석

합병 정렬은 앞부분과 뒷부분을 합병하면서 정렬하는 데에 반해, 반복합병정렬은 정렬할 배열의 각 인덱스를 2개씩 짝지어 합병하여 정렬하고, 그 정렬된 부분 배열을 또 짝지어 합병하여 정렬하는 방식으로 정렬을 수행한다.

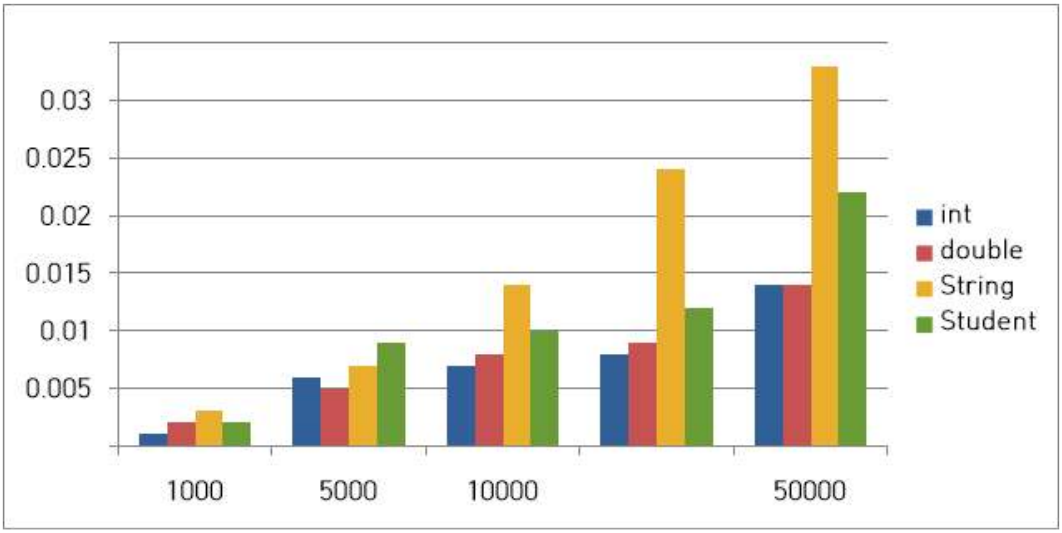
합병정렬과 동일한 시간복잡도인 O(NlogN)을 가진다.

동일한 배열을 이용하여서 정렬을 하는 것이 아닌 보조배열을 이용해 정렬을 하는 것이기 때문에 제자리 정렬(in-place sort)이 아니다.

(8) natural Merge

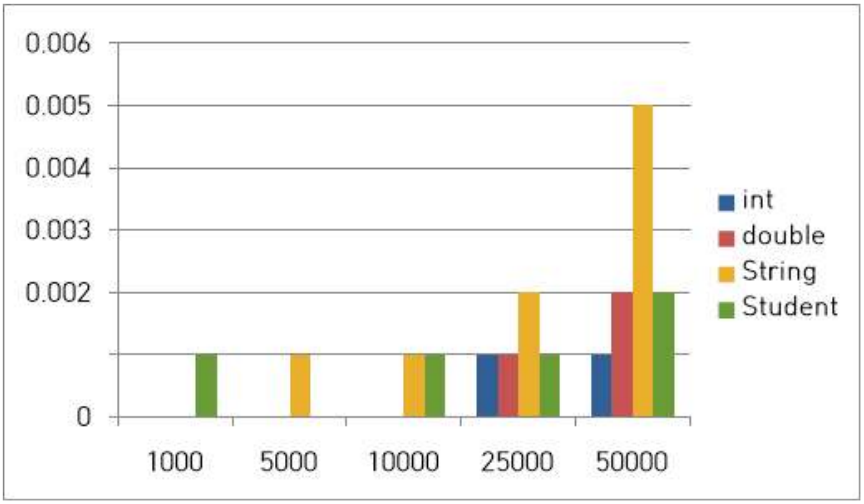
*random order values

	int	double	String	Student
1000	0.001	0.002	0.003	0.002
5000	0.006	0.005	0.007	0.009
10000	0.007	0.008	0.014	0.01
25000	0.008	0.009	0.024	0.012
50000	0.014	0.014	0.033	0.022



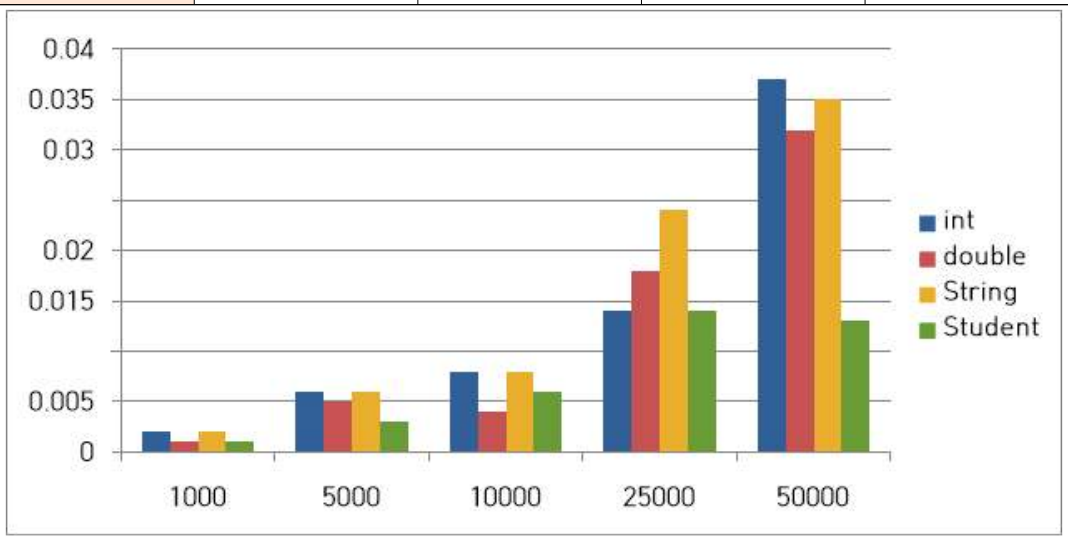
*increase order values

	int	double	String	Student
1000	0.0	0.0	0.0	0.001
5000	0.0	0.0	0.001	0.0
10000	0.0	0.0	0.001	0.001
25000	0.001	0.001	0.002	0.001
50000	0.001	0.002	0.005	0.002



*decrease order values

	int	double	String	Student
1000	0.002	0.001	0.002	0.001
5000	0.006	0.005	0.006	0.003
10000	0.008	0.004	0.008	0.006
25000	0.014	0.018	0.024	0.014
50000	0.037	0.032	0.035	0.013



*분석

자연 합병정렬은 반복합병정렬을 개선한 것으로, 이미 정렬되어 있는 부분 배열은 나누고 합병하는 과정을 생략하고 합병정렬을 수행한다.

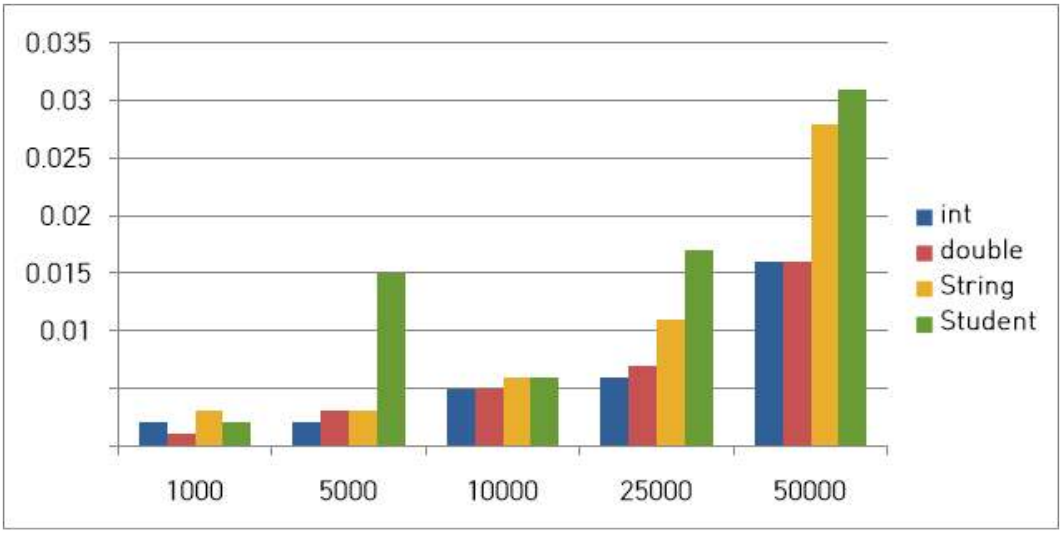
합병 정렬, 반복 합병 정렬과 마찬가지로 O(NlogN)의 시간복잡도를 가진다.

반복 합병정렬에 비해 정렬되어 있는 배열을 나누지 않고 합병하는 과정을 생략하기 때문에 Increase 정렬된 배열에서 훨씬 빠른 시간에 정렬을 끝내는 것으로 보인다.

(9) Heap Sort

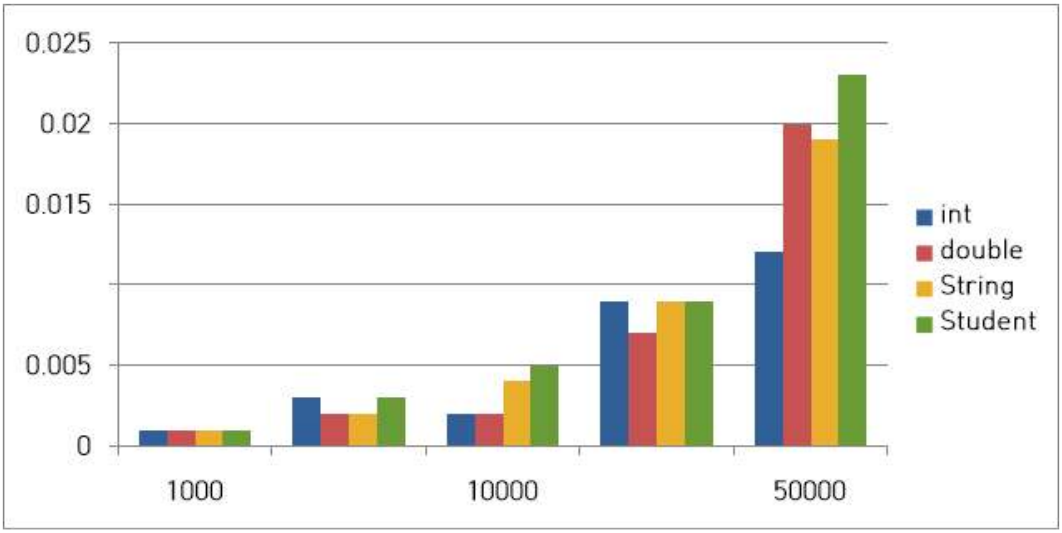
*random order values

	int	double	String	Student
1000	0.002	0.001	0.003	0.002
5000	0.002	0.003	0.003	0.015
10000	0.005	0.005	0.006	0.006
25000	0.006	0.007	0.011	0.017
50000	0.016	0.016	0.028	0.031



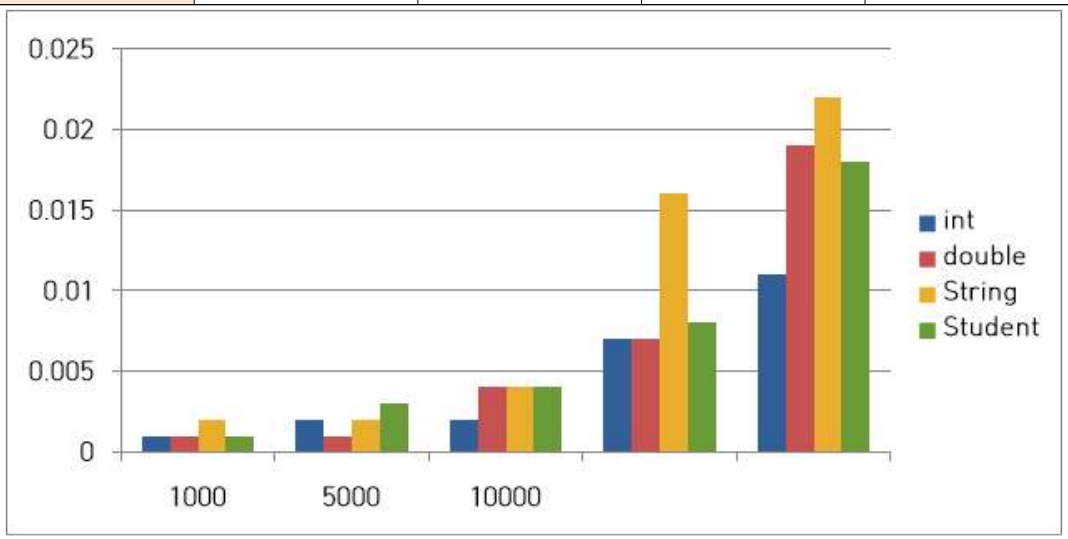
*increase order values

	int	double	String	Student
1000	0.001	0.001	0.001	0.001
5000	0.003	0.002	0.002	0.003
10000	0.002	0.002	0.004	0.005
25000	0.009	0.007	0.009	0.009
50000	0.012	0.02	0.019	0.023



*decrease order values

	int	double	String	Student
1000	0.001	0.001	0.002	0.001
5000	0.002	0.001	0.002	0.003
10000	0.002	0.004	0.004	0.004
25000	0.007	0.007	0.016	0.008
50000	0.011	0.019	0.022	0.018



*분석
오름차순 정렬인 경우 최소힙, 내림차순 정렬인 경우 최대힙을 사용해서 구성한다.

이 경우 힙을 만드는 과정에서 O(N)의 시간복잡도를 가진다.

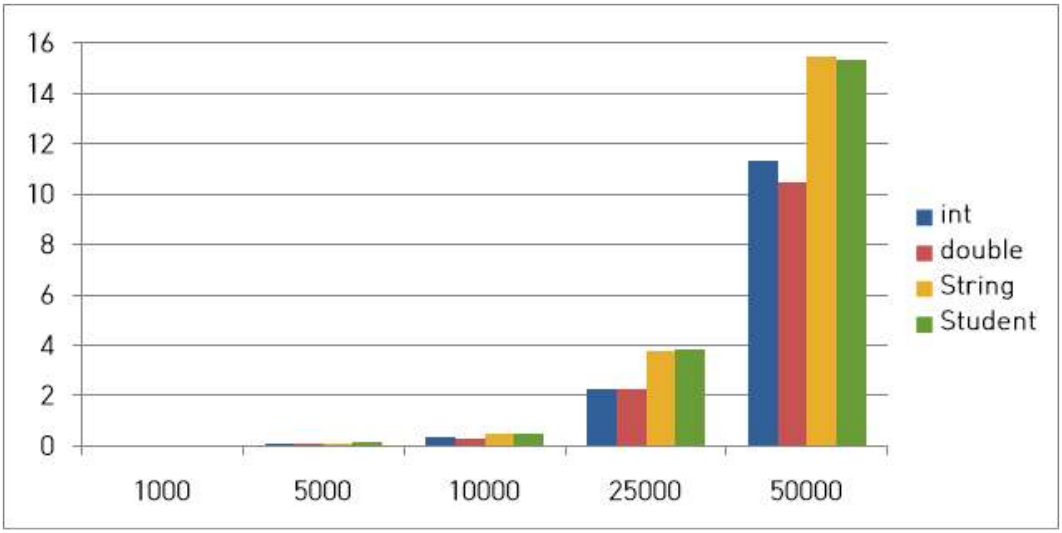
오름차순의 경우 최대값을 가지는 루트노드(내림차순의 경우 최소힙)를 마지막노드로 보내고 정렬된 값으로 보내고 downheap()을 수행하는 과정은 O(logN)의 시간복잡도를 가진다.

해당하는 경우가 N-1번의 횟수만큼 시행하기 때문에 결과적으로 (N-1)*O(NlogN), 즉 O(NlogN)의 시간복잡도를 가진다.

(10) bubble Sort

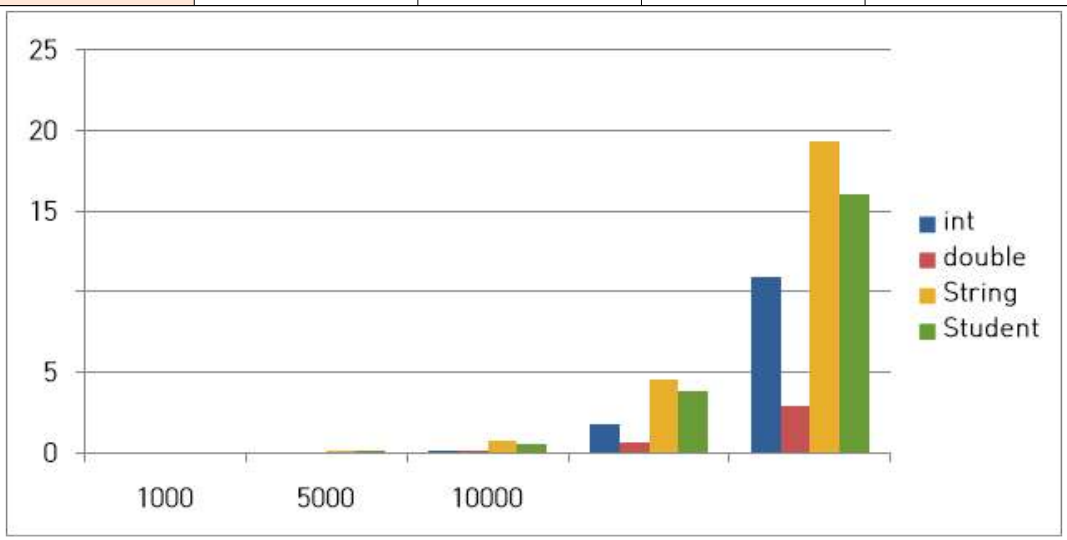
*random order values

	int	double	String	Student
1000	0.016	0.013	0.017	0.019
5000	0.075	0.069	0.113	0.121
10000	0.36	0.268	0.459	0.47
25000	2.241	2.266	3.755	3.835
50000	11.303	10.476	15.457	15.332



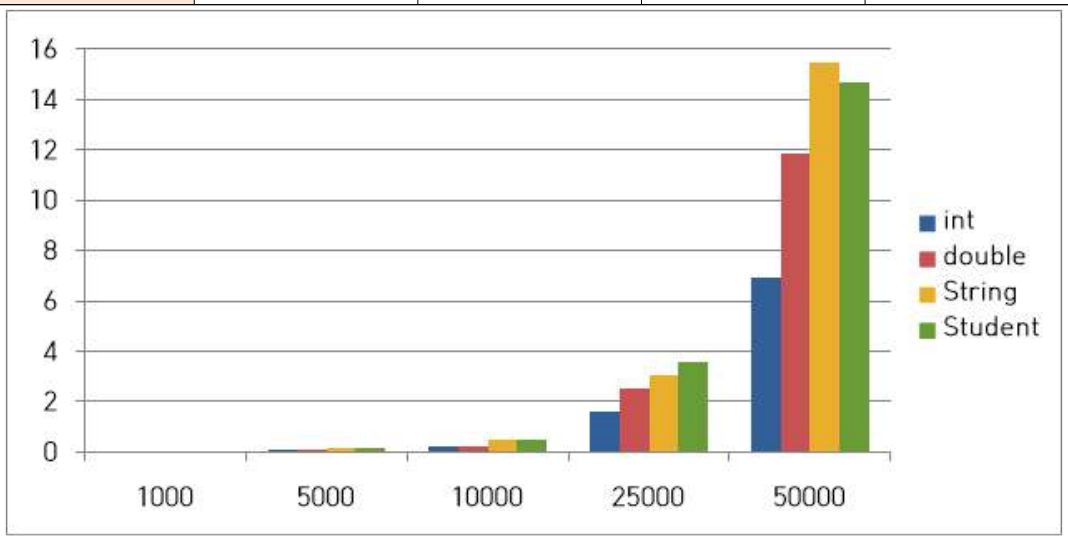
*increase order values

	int	double	String	Student
1000	0.01	0.011	0.013	0.018
5000	0.045	0.02	0.168	0.125
10000	0.141	0.085	0.717	0.519
25000	1.774	0.648	4.536	3.838
50000	10.877	2.865	19.35	16.04



*decrease order values

	int	double	String	Student
1000	0.014	0.014	0.015	0.021
5000	0.069	0.077	0.122	0.136
10000	0.203	0.241	0.485	0.481
25000	1.586	2.489	3.048	3.58
50000	6.943	11.866	15.479	14.646



*분석

N개와 N-1개의 모든 원소를 비교하기 때문에 N * (N - 1) 번의 시행이 어떠한 경우에도 시행이 되어야한다. $O(N^2)$의 시간복잡도를 가진다.

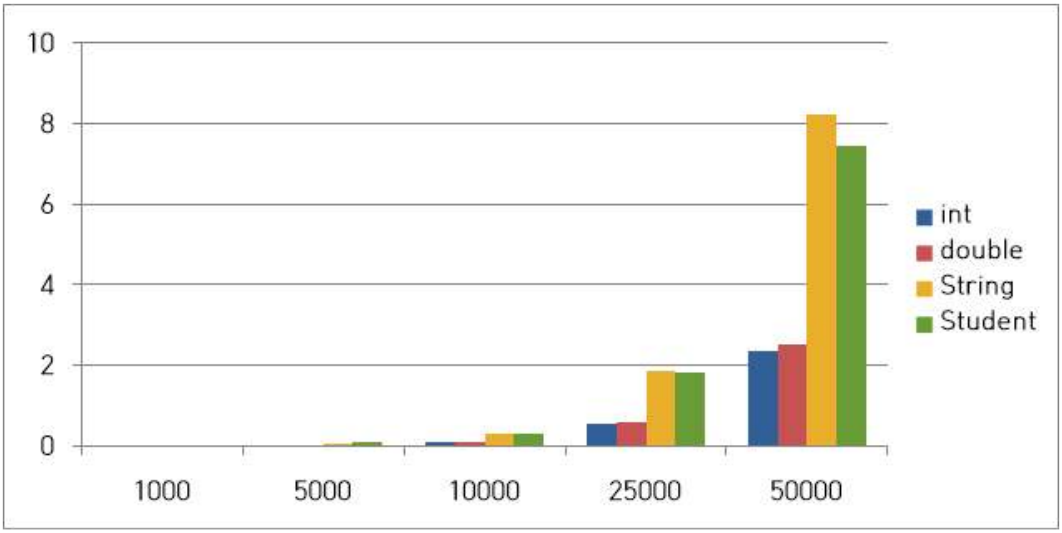
Random하게 정렬이 되거나, Increase하게 정렬이 되거나, Decrease하게 정렬이 되는 모든 경우에서 비슷한 시간이 걸린 것으로 보아 어떠한 값이 입력이 되는 경우에도 같은 시간복잡도를 가지는 것으로 보인다.

구현이 간단하지만 정렬하는데 소요되는 시간이 다른 정렬알고리즘에 비해 아주 오래 걸리는 알고리즘이다.

(11) Select Sort

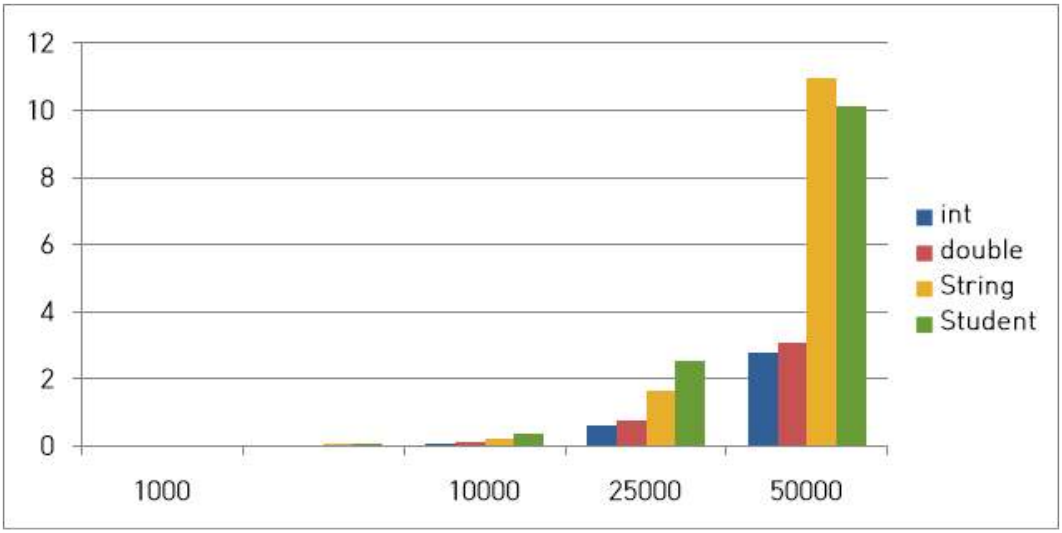
*random order values

	int	double	String	Student
1000	0.011	0.01	0.013	0.013
5000	0.021	0.022	0.065	0.08
10000	0.079	0.083	0.279	0.302
25000	0.526	0.588	1.855	1.837
50000	2.333	2.535	8.246	7.437



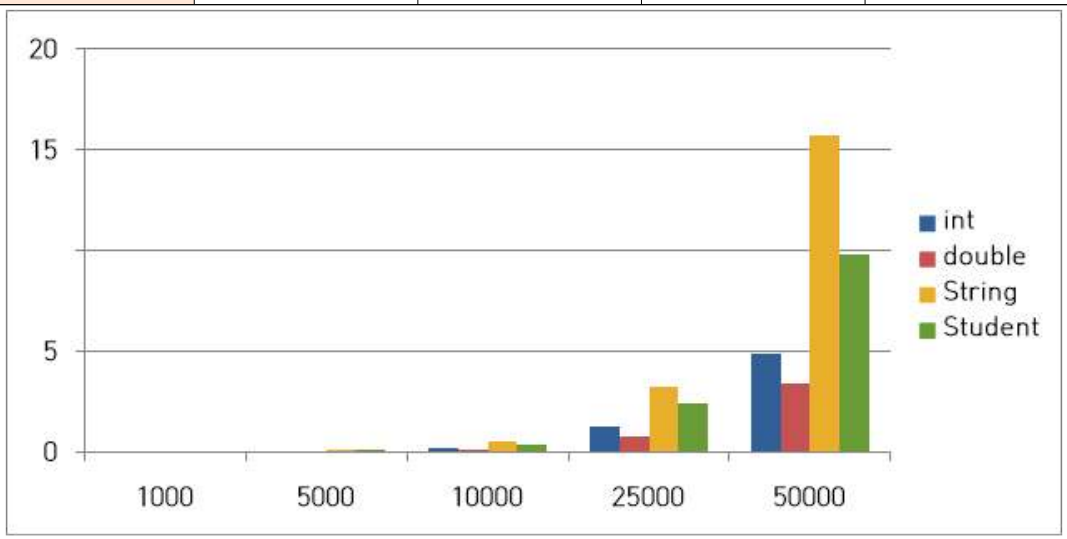
*increase order values

	int	double	String	Student
1000	0.011	0.011	0.012	0.011
5000	0.022	0.023	0.047	0.077
10000	0.077	0.094	0.221	0.349
25000	0.611	0.755	1.638	2.507
50000	2.757	3.062	10.94	10.14



*decrease order values

	int	double	String	Student
1000	0.0	0.01	0.012	0.012
5000	0.05	0.021	0.109	0.074
10000	0.196	0.127	0.541	0.336
25000	1.235	0.77	3.195	2.381
50000	4.873	3.384	15.674	9.816



*분석

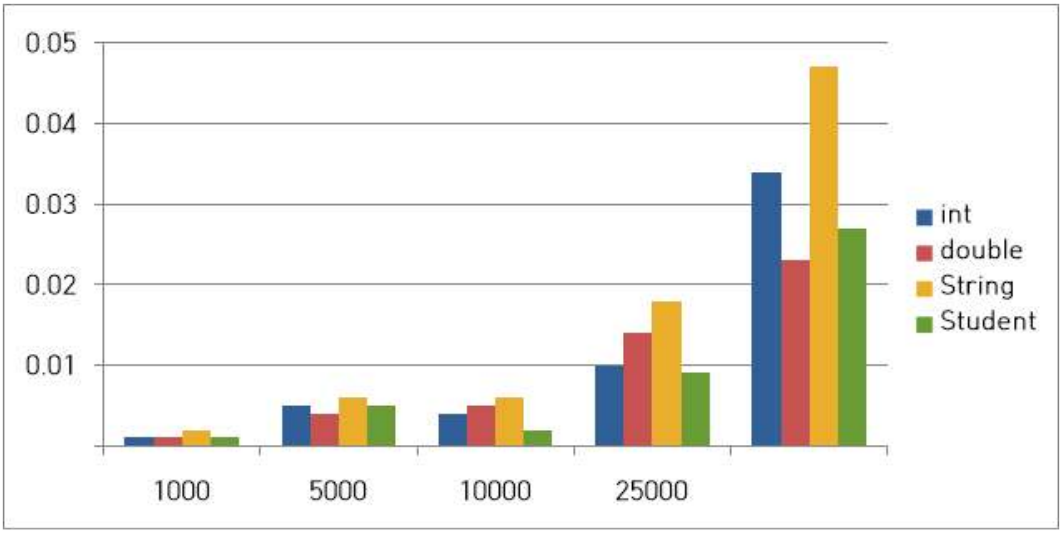
선택정렬은 배열에서 아직 정렬되지 않은 부분의 원소들 중에서 최솟값을 '선택'하여 정렬된 부분의 바로 오른쪽 원소와 교환하는 정렬 알고리즘이다.

N개의 원소 중 최솟값을 정렬된 부분의 오른쪽 원소와 swap한다. 이때 최솟값을 찾기 위해 비교하는 첫 번째 루프에서 N-1번이 시행된다. 두 번째 루프에서 N-2번이 시행 되며, 마지막에는 1번이 시행이 된다. 총 (N-1) * N / 2 번이 시행이 되므로 $O(N^2)$의 시간복잡도를 가진다.

(12) Array Sort

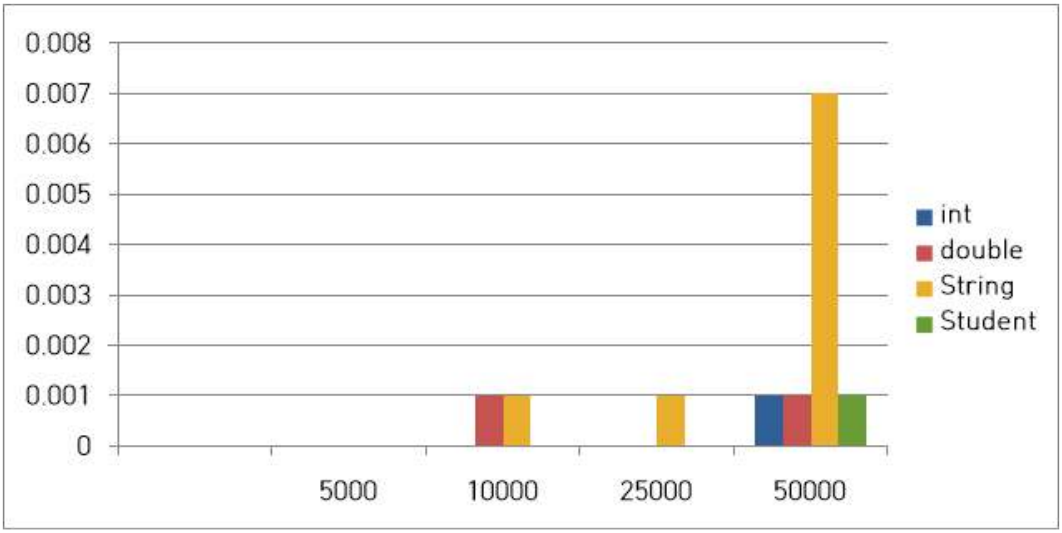
*random order values

	int	double	String	Student
1000	0.001	0.001	0.002	0.001
5000	0.005	0.004	0.006	0.005
10000	0.004	0.005	0.006	0.002
25000	0.01	0.014	0.018	0.009
50000	0.034	0.023	0.047	0.027



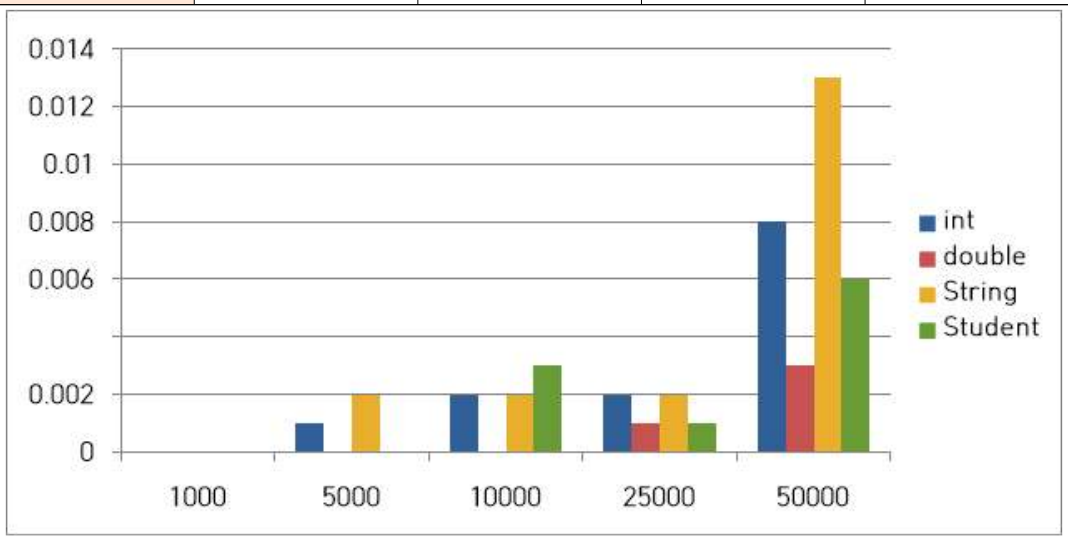
*increase order values

	int	double	String	Student
1000	0.0	0.0	0.0	0.0
5000	0.0	0.0	0.0	0.0
10000	0.0	0.001	0.001	0.0
25000	0.0	0.0	0.001	0.0
50000	0.001	0.001	0.007	0.001



*decrease order values

	int	double	String	Student
1000	0.0	0.0	0.0	0.0
5000	0.001	0.0	0.002	0.0
10000	0.002	0.0	0.002	0.003
25000	0.002	0.001	0.002	0.001
50000	0.008	0.003	0.013	0.006



*분석

java.util.Arrays 의 API중 하나인 Arrays.sort() 메서드이다.

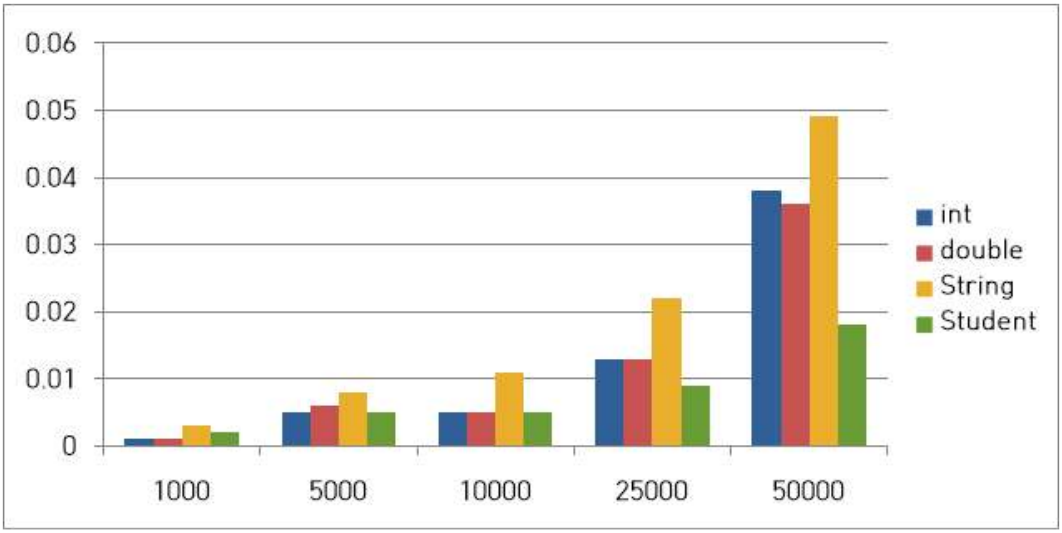
Arrays.sort()는 Dual-Pivot Quicksort로 구현되어 있으며, 일반적인 Quick Sort와 달리 정렬이 되어 있는 경우를 보완하여 모든 데이터 타입에서 O(nlogn)의 시간복잡도를 가진다.

Collections.sort() 와달리 Arrays.sort()는 Quick Sort를 기반으로 하여 non-stable한 정렬이다.

(13) Collections Sort

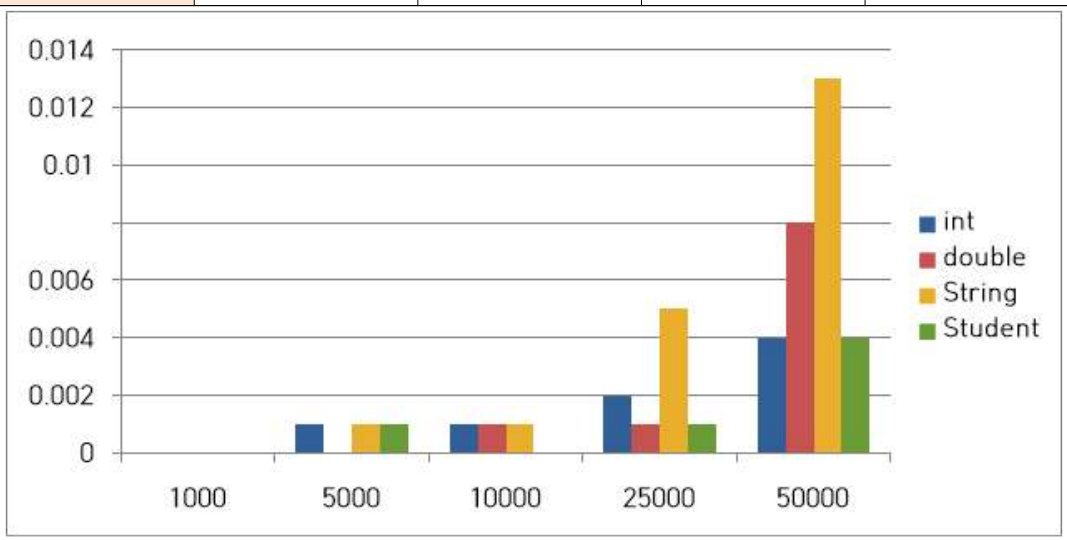
*random order values

	int	double	String	Student
1000	0.001	0.001	0.003	0.002
5000	0.005	0.006	0.008	0.005
10000	0.005	0.005	0.011	0.005
25000	0.013	0.013	0.022	0.009
50000	0.038	0.036	0.049	0.018



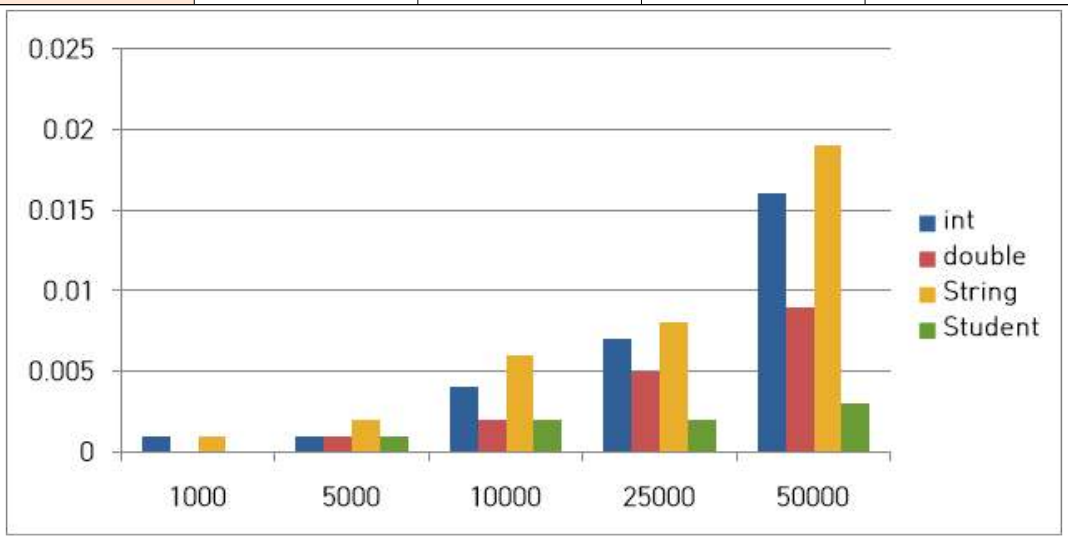
*increase order values

	int	double	String	Student
1000	0.0	0.0	0.0	0.0
5000	0.001	0.0	0.001	0.001
10000	0.001	0.001	0.001	0.0
25000	0.002	0.001	0.005	0.001
50000	0.004	0.008	0.013	0.004



*decrease order values

	int	double	String	Student
1000	0.001	0.0	0.001	0.0
5000	0.001	0.001	0.002	0.001
10000	0.004	0.002	0.006	0.002
25000	0.007	0.005	0.008	0.002
50000	0.016	0.009	0.019	0.003



*분석
java.util.Collections의 API중 하나인 Collections.sort() 메서드이다.

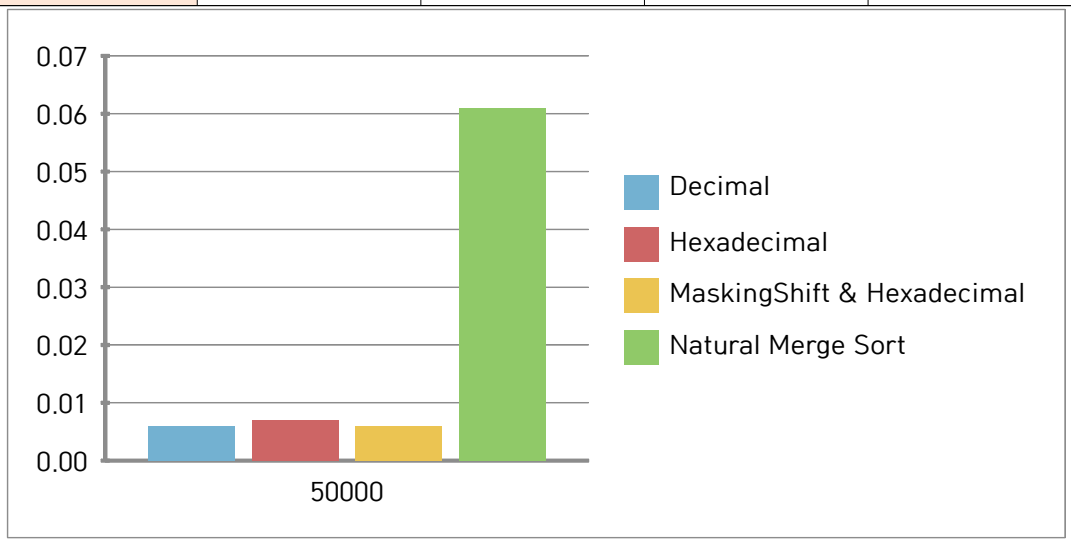
Collections.sort()는 List를 Array로 변경한 뒤 merge sort를 변형한 TimSort를 사용한다.
시간복잡도는 O(nlogn)으로 Arrays.sort()와 동일하다.

Collections.sort()는 T[](Boxed)의 배열의 형태를 가지고 때문에 unwrap한 뒤 재정렬하기 때문에 오버헤드[1]가 발생해 느리지만 stable하다(같은 값일 경우 위치를 바꾸지 않음)는 장점도 존재한다.

1) 오버헤드 : 어떤 처리를 하기 위해 들어가는 간접적인 처리 시간 및 메모리

4. 비교 기반 vs. 비비교 기반 / 산술 연산 vs. 논리 연산

	Decimal Sort	Hexadecimal Sort	MaskingShift & Hexadecimal Sort	Natural Merge Sort
50000	0.006	0.007	0.006	0.061



- 데이터 자료형 : int

- 범위 : 0 ~ 2^16 - 1 = 0 ~ 65535

- 데이터 수 : 50000

- 비교 기반 정렬에서 random하게 정렬된 배열의 정렬속도가 가장 빠른 Natural Merge Sort 와 Recursive Merge Sort 중 Natural Merge Sort를 선택하였다.

4-1. 비교 기반 정렬 vs. 비비교 기반 정렬
비비교 기반 정렬의 (기수의 수(R) + 데이터의 수(N))에서 자리수(d) 만큼 곱한 결과만큼의 시간이 소요 되므로 O(d(R + N))의 시간복잡도, 즉 O(N) 만큼의 시간복잡도를 가진다.

비비교 기반 정렬인 기수 정렬에서는 시간복잡도가 O(N)이고 비교 기반 정렬의 하나인 자연 합병 정렬의 시간복잡도는 O(NlogN)이다.

비비교 기반 정렬의 경우 키의 특정한 부분의 수를 기준으로 정렬을 하는 경우 일반적인 비교 기반의 정렬 보다 빠른 시간복잡도를 가진다.

위 예제를 보면 비비교기반의 3가지의 기수정렬이 비교기반의 자연합병정렬보다 훨씬 빠른 속도로 정렬을 하는 것으로 볼 수 있다.

4-2. 산술 연산 vs. 논리 연산
산술 연산의 경우 10진수의 기(0 ~ 9)를 사용하는 10진수 기수정렬과 16진수의 기(0 ~ 15)를 사용하는 16진수 기수정렬을 사용하였다.
논리 연산의 경우 마스킹(&)과 시프트연산(>> , <<)과 16진수의 기(0 ~ 15)를 사용하는 16진수 논리 연산 기수정렬을 사용하였다.

일반적인 기수 정렬에서 자리수 d, 기수의 개수 R, 데이터의 수 N일 때 O(d(N+R))의 시간 복잡도가 걸린다. 데이터의 수가 동일할 때 10진수 기수정렬과 16진수 기수정렬의 다른 점은 기수의 차이가 6만큼 더 많아 표현할 수 있는 데이터의 자리수가 적어질 수 있기 때문에 데이터의 입력범위가 커질수록 16진수를 이용한 산술연산이 더 효율적일 것이라 생각한다.

산술 연산에 비해 논리 연산이 비트연산을 이용하여 연산을 수행하기 때문에 산술연산에 비해 더 빠른 속도로 계산이 가능하다.

데이터의 입력수가 200,000개 일 때

```
radix hexadecimal : 0.021
radix maskingshift : 0.016
radix decimal : 0.03
array sort : 0.135
```

데이터의 입력수가 500,000개 일 때

```
radix hexadecimal : 0.049
radix maskingshift : 0.039
radix decimal : 0.053
array sort : 0.321
```