자료구조 정렬

20180283 김주호
20191203 최혜민
20200284 김정현

목차

---

0. 구현 환경

CPU : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz1.69GHz
RAM : 16.00GB
SSD : 512GB
OS : Windows 10 Education
시스템 종류 : 64비트 운영 체제, x64 기반 프로세서
IDE : Eclipse (2020.03)
JAVA : 16.0.1

1.0 Stable sort / Non-stable sort

Stable sort

: 중복된 키를 순서대로 정렬하는 정렬 알고리즘들을 지칭한다. 즉, 같은 값이 2개 이상 리스트에 등장할 때 기존 리스트에 있던 순서대로 중복된 키들이 정렬된다는 것을 의미

-> insertion sort, merge sort, radix sort

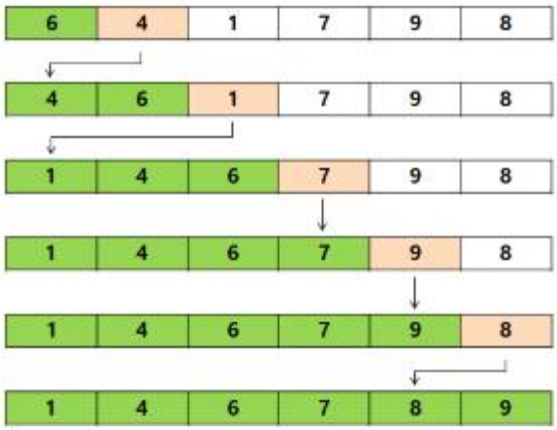
Non-stable sort

: sorting을 할 때 같은 key 값을 가진 node들이 sorting 전과 sorting 후에 순서가 뒤바뀌게 되는 것을 의미

-> selection sort, shell sort, quick sort, heap sort

1.1 stable sort 분석

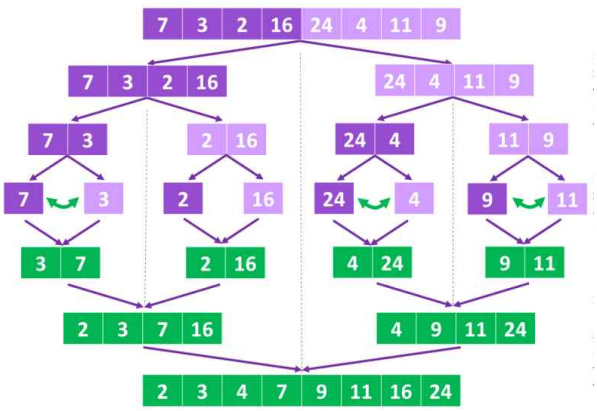
1.1.1 > Insertion sort



Insertion sort는 배열이 정렬된 부분과 정렬되지 않은 부분으로 나뉘며, 정렬 안 된 부분의 가장 왼쪽 원소를 정렬된 부분에 '삽입'하는 방식의 정렬 알고리즘이다.

stable한 이유 : 주어진 배열, 리스트의 모든 원소를 앞에서부터 차례대로 이미 정렬된 부분과 비교하여 자신의 위치를 찾아 삽입하고 같은 경우에는 같은 값의 요소 오른쪽에 삽입하므로 stable한 정렬을 완성할 수 있다.

1.1.2 > Merge sort

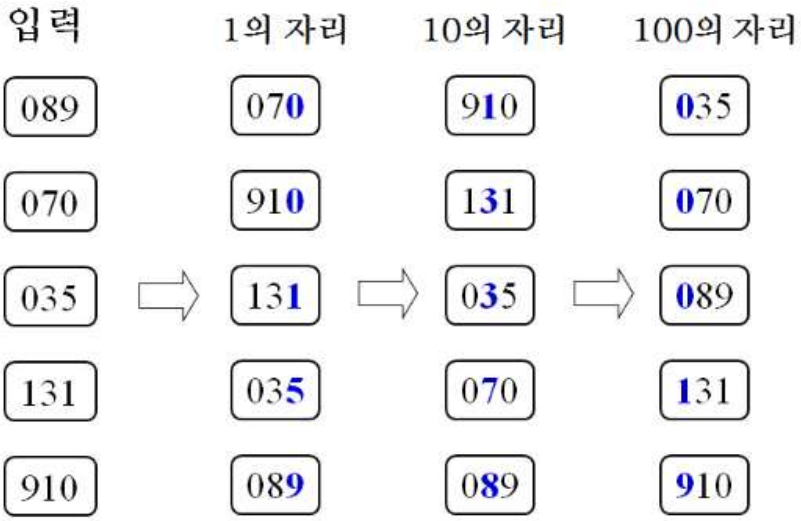


Merge sort는 크기가 N인 입력을 ½N 크기를 갖는 입력 2개로 분할하고, 각각에 대해 재귀적으로 합병정렬을 수행한 후, 2개의 각각 정렬된 부분을 합병하는 정렬 알고리즘이다.

stable한 이유 : 원소를 최대한 작게 분할하여 인접한 원소들끼리 비교하고 순서대로 합쳐나가기 때문에 원소 중 같은 값이 있는 경우에도 순서가 유지되므로 stable한 정렬을 완성

할 수 있다.

1.1.3 > Radix sort



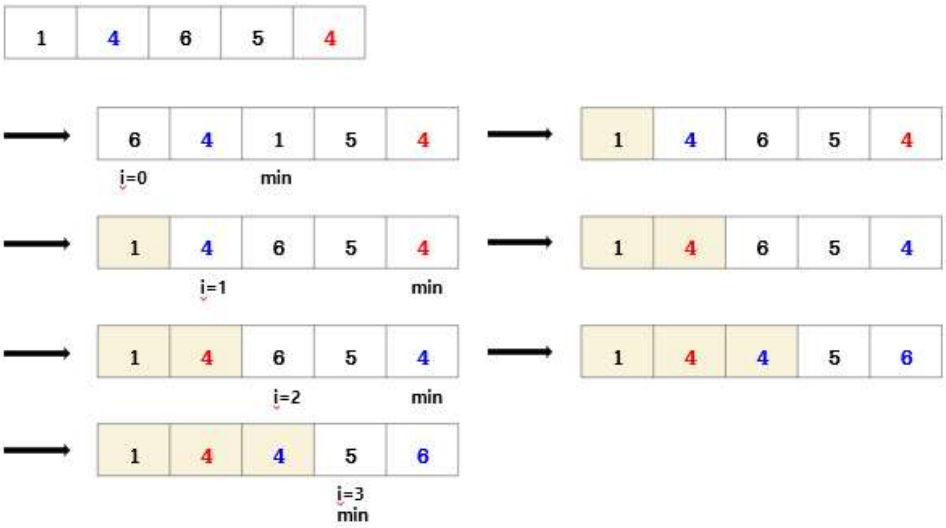
LSD 기수정렬

Radix sort는 키를 부분적으로 비교하는 정렬 알고리즘으로 Key가 숫자로 되어있으면, 각 자릿수에 대해 Key를 비교한다.

stable한 이유 : 자릿수에 맞게 0~9까지의 bucket이 있고, 이 bucket에 해당하는 자릿수를 포함한 수를 넣을 때 자리를 바꿔가며 버킷에 넣는 것을 반복하여 자동으로 정렬이 되는데 bucket은 Queue를 사용하여 이전 자릿수의 bucket을 0부터 차례대로 읽어나가므로 stable하다.

1.2 Non-stable sort 분석

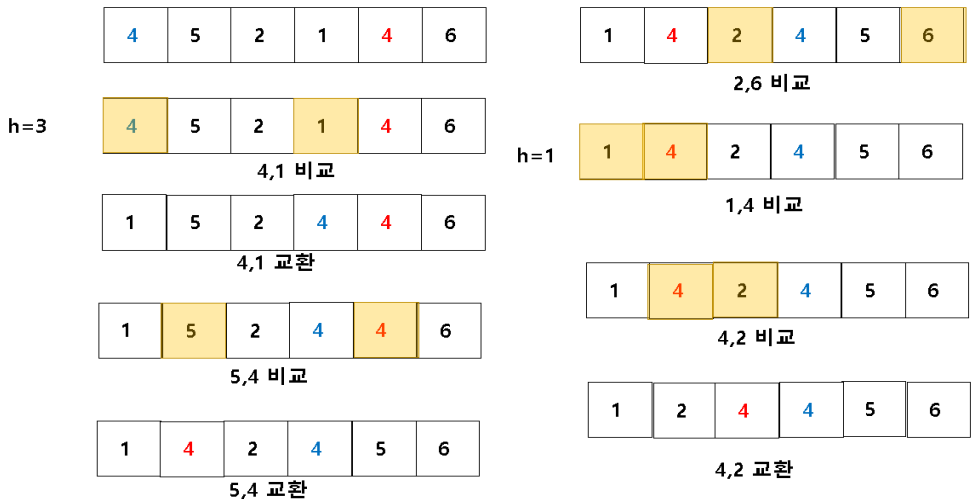
1.2.1 > Selection sort



Selection sort는 배열에서 아직 정렬되지 않은 부분의 원소 중에서 최솟값을 '선택'하여 정렬된 부분의 바로 오른쪽 원소와 교환하는 정렬 알고리즘이다.

non-stable한 이유 : 정렬된 부분의 오른쪽이 파란색 '4'일 때 최솟값이 빨간색 '4'이므로 둘이 swap하게 되면 정렬되지 않은 부분의 빨간색 '4'가 파란색 '4'보다 앞으로 가게 되어 순서가 뒤바뀌기 때문에 stable한 결과를 보장할 수 없기에 non-stable한 정렬이다.

1.2.2 > Shell sort

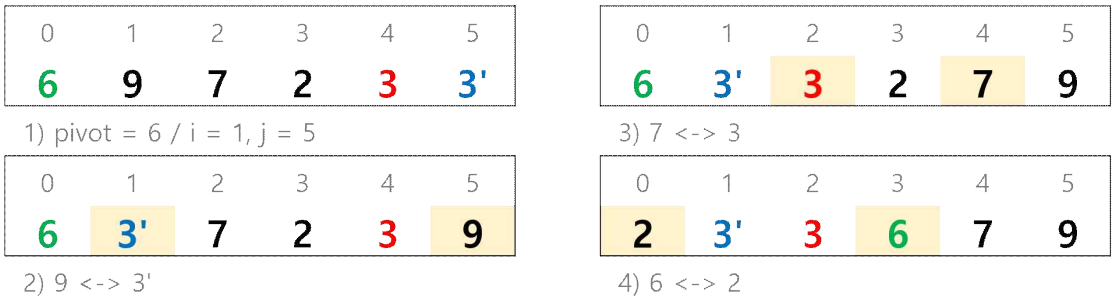


Shell sort는 Insertion sort에 작은 값을 가진 원소들을 배열의 앞부분으로 옮기며 큰 값

을 가진 원소들이 배열의 뒷부분에 자리 잡도록 만드는 과정인 전처리과정을 추가한 정렬 알고리즘이다.

non-stable한 이유 : 바로 옆의 원소끼리 교환하는 것이 아니므로 동일한 값의 원소가 나오면 순서가 보장되지 않아 순서가 바뀔 수 있기에 stable한 결과를 보장할 수 없게 되어 non-stable한 정렬이다.

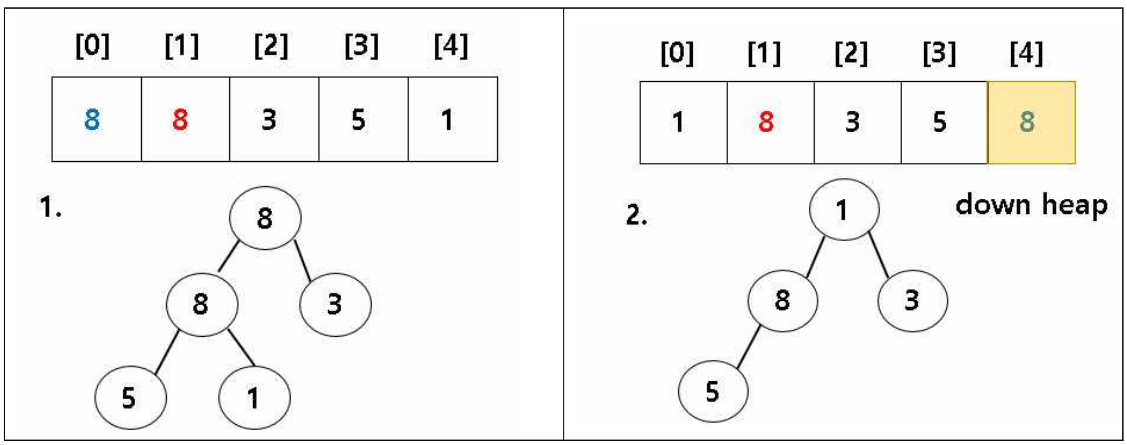
1.2.3 > Quick sort

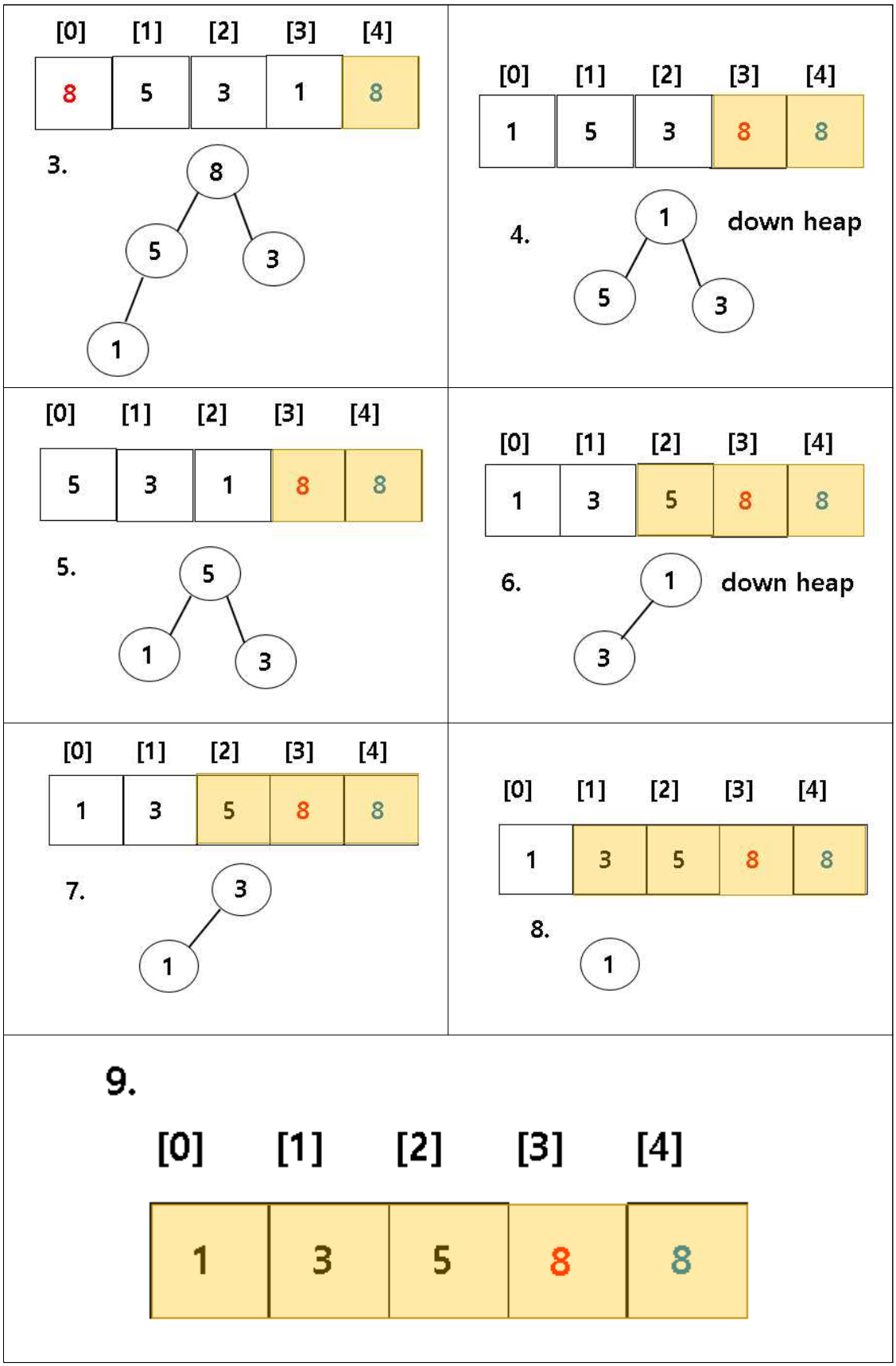


Quick sort는 입력의 맨 왼쪽 원소(피벗, Pivot)를 기준으로 피벗보다 작은 원소들과 큰 원소들을 각각 피벗의 좌우로 분할한 후, 피벗보다 작은 원소들과 피벗보다 큰 원소들을 각각 재귀적으로 정렬하는 알고리즘이다.

non-stable한 이유 : 각 원소의 원래 위치를 고려하지 않고 피벗을 기준으로 정렬을 수행하기 때문에 순서가 보장되지 않기에 non-stable한 정렬이다. 예제의 정렬에서도 정렬 전 3, 3' 순서와 정렬 후 3, 3' 순서가 변경됨을 알 수 있다.

1.2.4 > Heap sort





Heap sort는 힙 자료구조를 이용하는 정렬로 먼저 배열에 저장된 데이터의 키를 우선순위

로 하는 최대힙(Max Heap)을 구성하고, 루트 노드의 숫자를 힙의 가장 마지막 노드에 있는 숫자와 교환한 후 힙 크기를 1 감소시키고 루트 노드로 이동한 숫자로 인해 위배된 힙속성을 downheap 연산으로 복원하는 과정을 반복하여 나머지 원소들을 정렬하는 알고리즘이다.

non-stable한 이유 : Heap sort의 경우 동일한 항목의 상대적인 순서가 변경될 수 있어서 stable이 아니다. 위 정렬 과정 중 과정 4에서 정렬 전 순서와 다르게 8, 8로 바뀌어 고정된 이후로 앞부분 정렬만 진행되어 정렬 전 순서를 보장할 수 없음을 보인다.

2. 정렬 알고리즘 비교

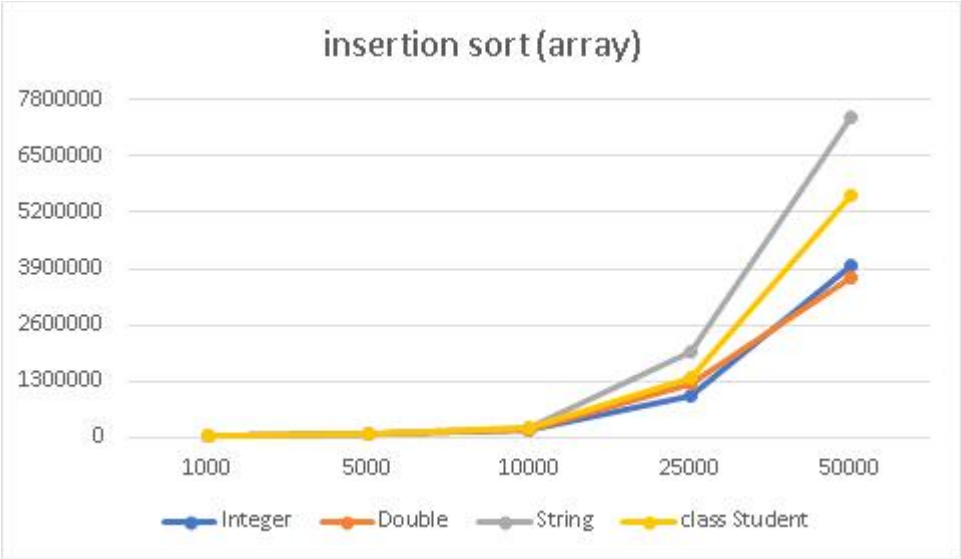
2.0 분석, 정렬 조건 및 도출 방법

- 분석, 정렬 조건
    - 입력 개수(n) : {1000, 5000, 10000, 25000, 50000}
    - 수행시간 단위 : micro second(μs)
    - 문자열 길이 : 15
    - int, double은 Wrapper Class인 Integer, Double로 실행

- 수행시간 도출 방법 : 상각 분석을 통한 수행시간을 도출하려 하였지만, 반복문을 통한 상각 분석 시 시간이 급등, 급감하는 경우가 존재하여 20번의 컴파일을 통한 평균치를 계산하여 도출하였다.

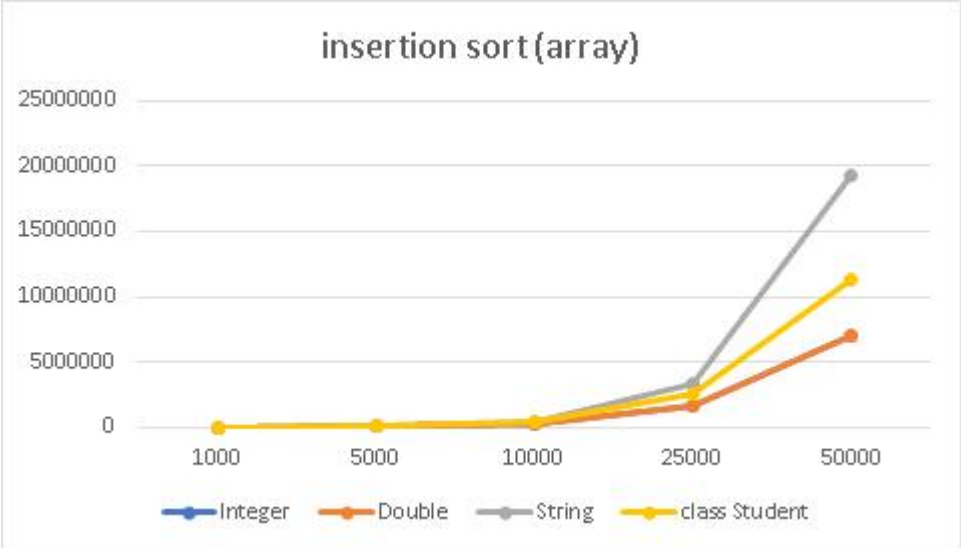
2.1 Insertion Sort (array)

2.1.1 Random



2.1.2 Increase

2.1.3 Decrease



2.1.4 분석

- Insertion sort는 정렬 안 된 부분의 가장 왼쪽 원소를 정렬된 부분에 '삽입'하는 방식의 정렬 알고리즘이다.

- 이미 정렬된 배열에 대해서는 N-1 번의 탐색만 하면 되므로 입력을 오름차순으로 받았을 경우의 정렬은 $O(N)$의 시간 복잡도를 가지므로 다른 정렬되지 않은 배열보다 수행시간이 빠르다.

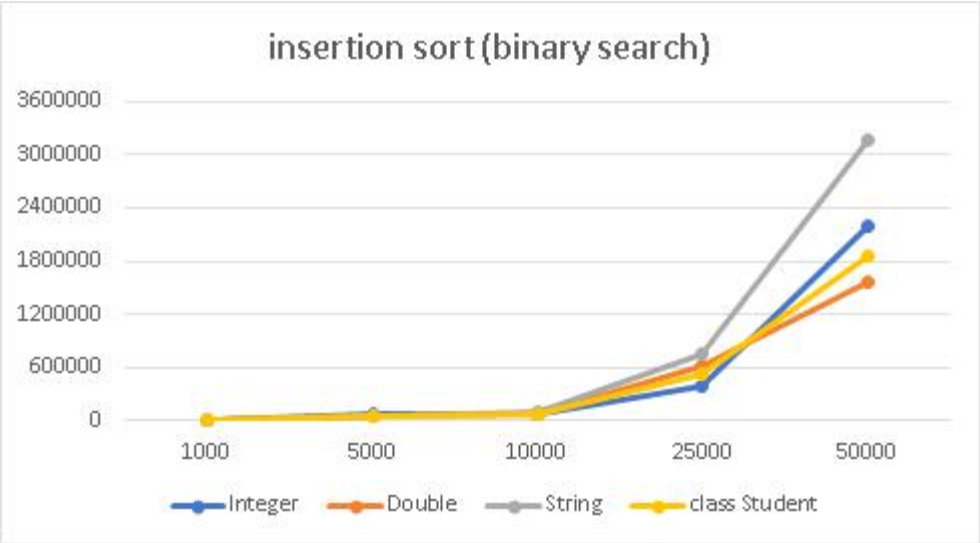
- 정렬이 되지 않은 평균 경우인 Random 값의 배열에서는 $O(N^2)$의 시간 복잡도를 가진다.

- 최악 경우인 입력이 역으로 정렬되어있는 내림차순에서는 $O(N^2)$의 시간 복잡도를 가지지만 같은 $O(N^2)$의 복잡도를 가지는 Random에 비해서 수행시간이 길다는 것이 보인다.

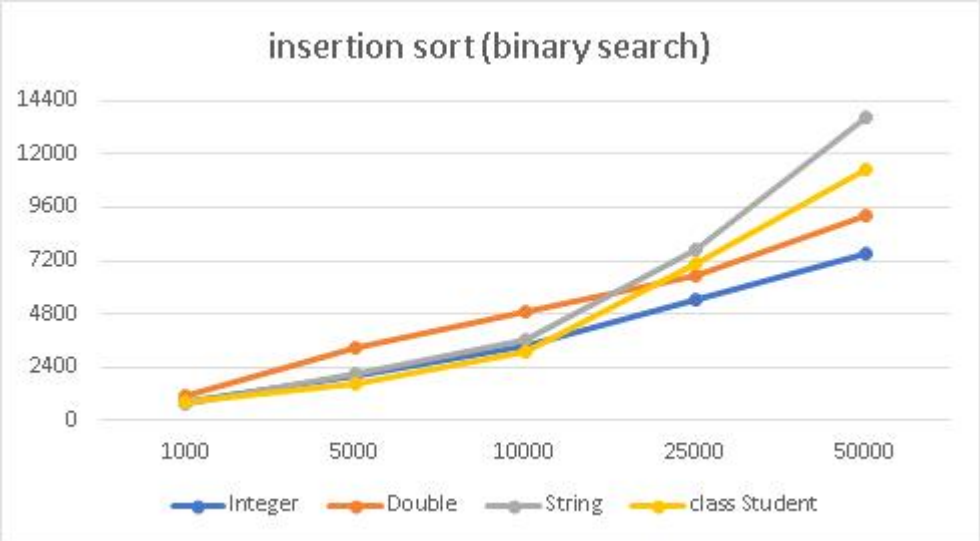
- 자료형에 따른 수행시간 차는 평균적으로 String이 가장 느린 속도를 보여주었고, Integer와 Double은 비슷한 수행시간을 보여주었으며 Student의 경우에는 Random 값 입력의 경우에서 String 다음으로 느린 수행시간을 보여주어 Increase와 Decrease의 경우에서도 String 다음으로 느린 수행시간을 기대하였지만, Increase의 경우에서 가장 빠른 속도를 보였다.

2.2 Insertion Sort (binary search)

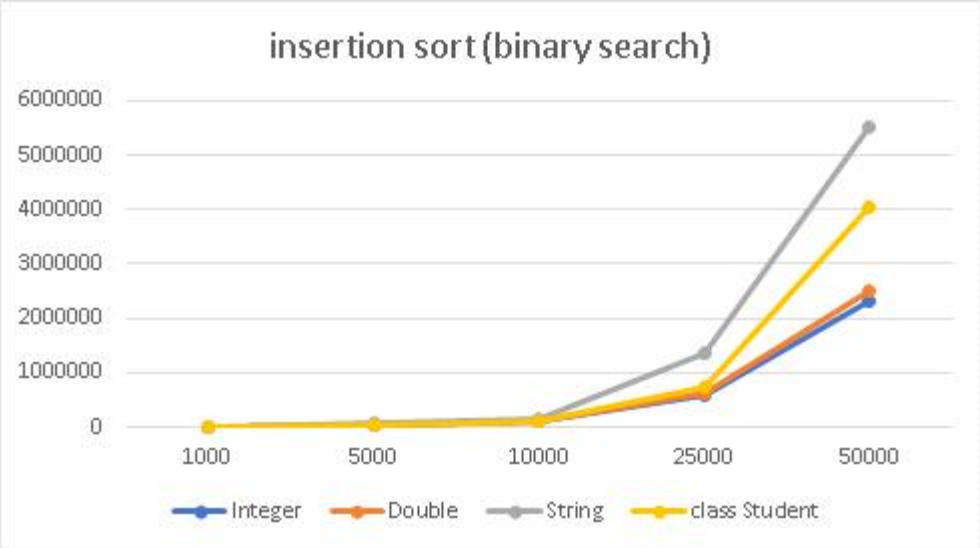
2.2.1 Random



2.2.2 Increase



2.2.3 Decrease



2.2.4 분석

- binary search의 복잡도는 $O(\log N)$이고 배열 탐색의 복잡도는 $O(N)$이므로 배열 탐색을 사용한 삽입 정렬보다 더 빠른 속도를 보인다.

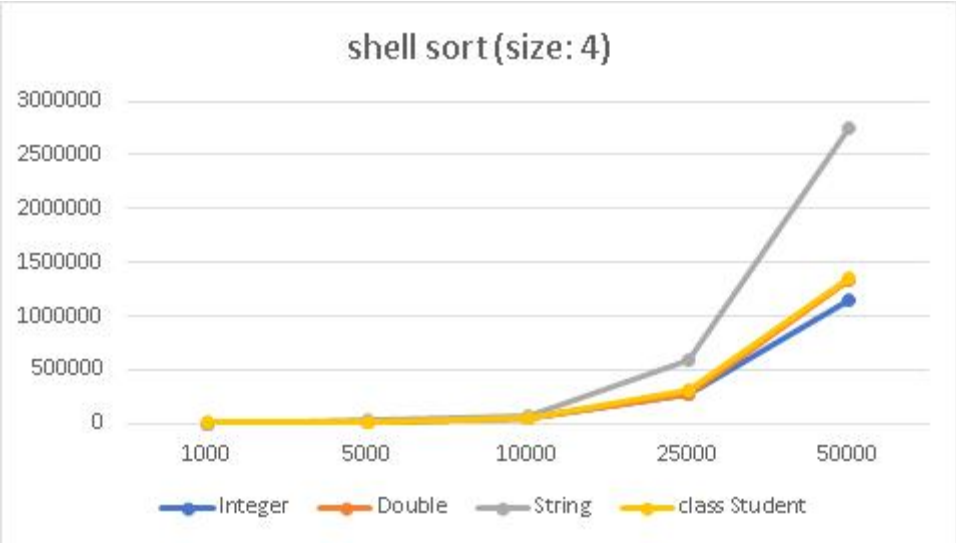
- 자료형에 따른 수행시간 차는 String이 가장 느린 속도를 보여주었으며 Random 값을 제외한 Increase, Decrease의 경우에서는 평균적으로 Student, Double, Integer 순으로 느린 속도를 보였다.

- Random에서 Integer가 Double, Student보다 느린 속도를 보였는데, 25000까지는 더 빠른 속도를 보이다가 50000에서 느려진 것으로 보아 IDE의 과부하로 인한 속도 감소로 보인다.

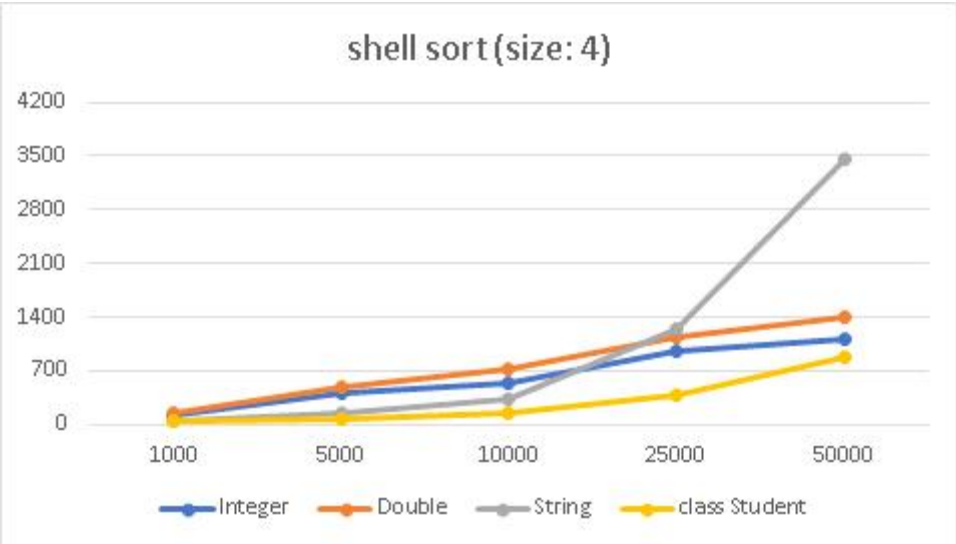
- Array를 이용한 Insertion Sort와 Binary Search를 이용한 Insertion Sort의 정렬을 비교하면 Binary Search를 이용한 Insertion Sort가 Random, Decrease의 경우에서는 빠른 속도를 보여주었지만, Increase의 경우에서는 Array를 이용한 Insertion Sort가 더욱더 빠른 속도를 보여주었다.

2.3 Shell Sort (size: 4)

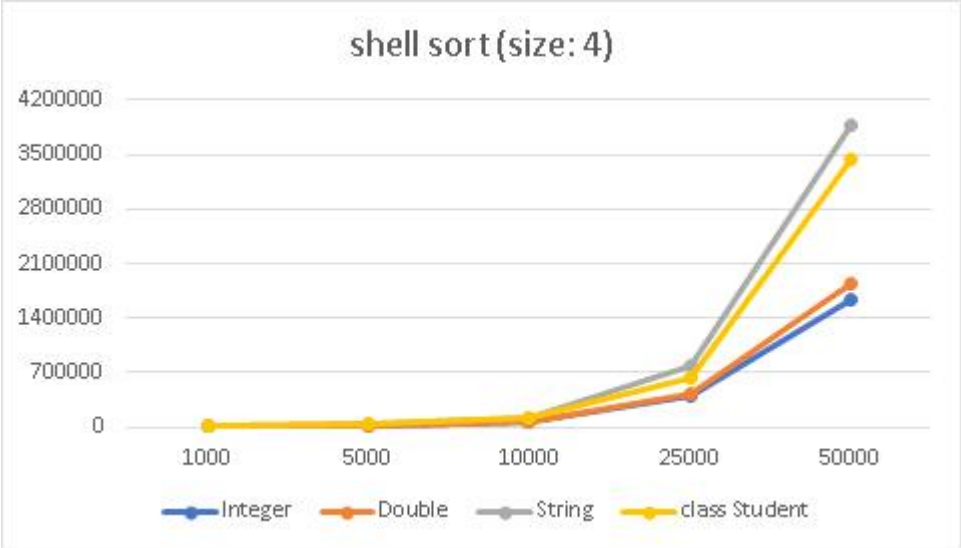
2.3.1 Random



2.3.2 Increase



2.3.3 Decrease



2.3.4 분석

- Shell Sort는 Insertion sort에 작은 값을 가진 원소들을 배열의 앞부분으로 옮기며 큰 값을 가진 원소들이 배열의 뒷부분에 자리 잡도록 만드는 과정을 추가한 정렬 알고리즘이다.

- Shell 크기에 따라 속도가 달라지기 때문에 Shell의 size를 4로 고정하고 수행한 결과 모든 경우에서 String이 가장 느린 속도를 보였고 Random, Decrease의 경우에서는 Student, Double, Integer 순으로 느린 속도를 보여주었지만, Increase의 경우에서는 Student가 가장 빠른 속도를 보여주었다.

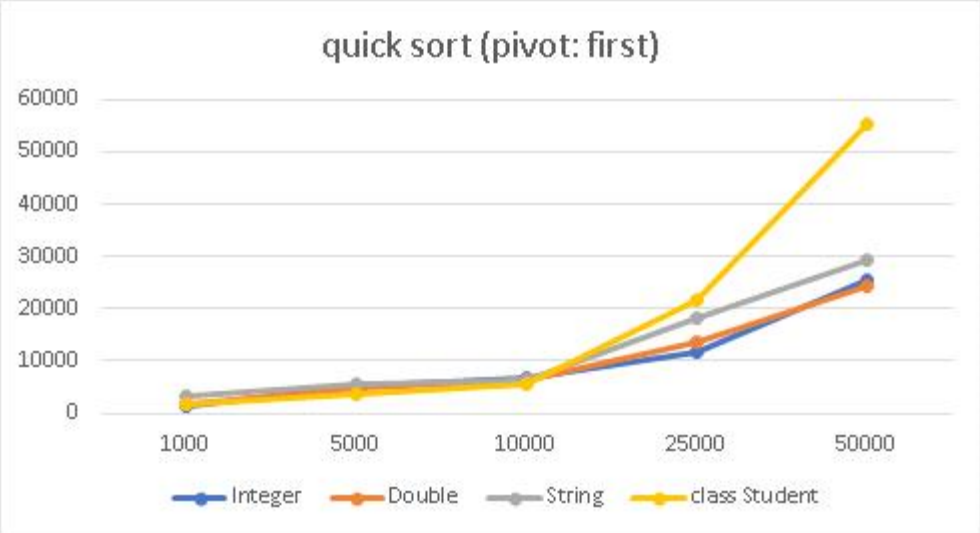
- 정확한 수행시간은 아직 풀리지 않은 정렬이지만 Decrease, Random, Increase의 입력 순으로 느린 수행시간을 보였다.

- $O(N^2)$의 시간 복잡도를 가지는 Insertion Sort보다 더 나은 성능을 보여주지만 $O(NlogN)$의 복잡도를 갖는 다른 정렬보단 느린 것으로 보아 Shell Sort의 시간 복잡도는 $O(NlogN)$보다는 크고 $O(N^2)$보다는 작다는 것을 유추할 수 있다.

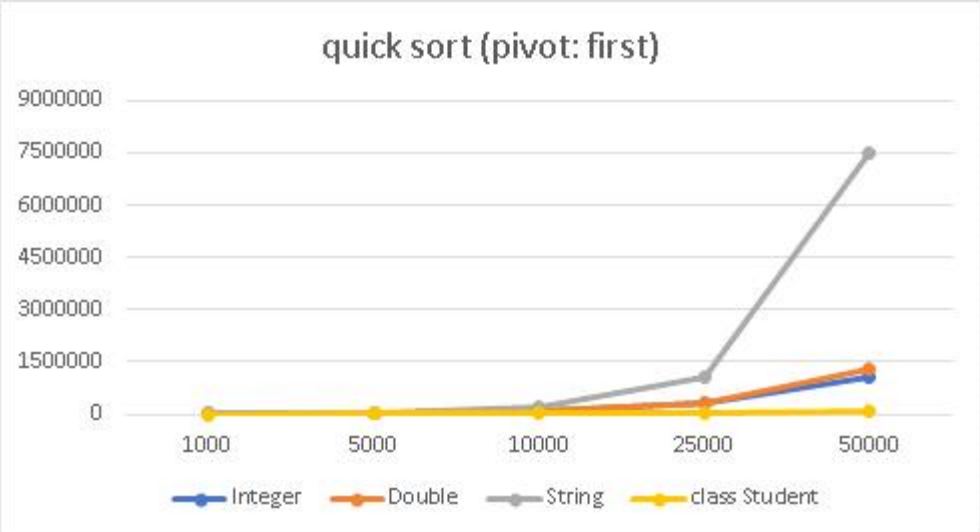
- Shell Sort는 Insertion Sort를 기반으로 둔 알고리즘이기 때문에 이미 값이 정렬되어있는 Increase의 경우에서는 매우 빠른 속도를 보였다.

2.4 Quick Sort (first pivot)

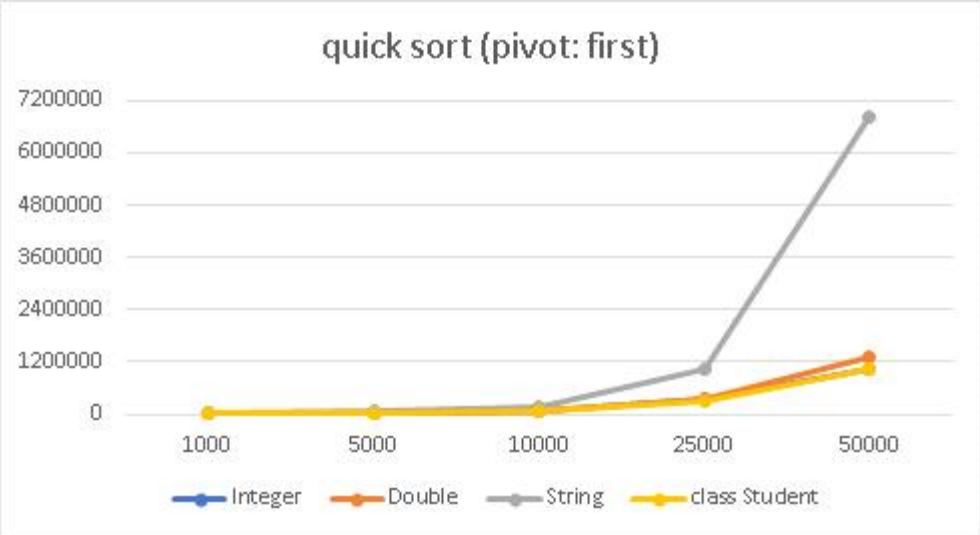
2.4.1 Random



2.4.2 Increase



2.4.3 Decrease



2.4.4 분석

- 피벗을 기준으로 피벗보다 큰 원소와 작은 원소들을 좌우로 분리한 후 각각 재귀적으로 정렬하는 알고리즘이다.

- Random 값을 입력받는 경우가 평균 경우의 시간 복잡도인 $O(NlogN)$의 수행시간을 가지므로 최악 경우인 오름차순, 내림차순 배열보다 수행시간이 빨랐다.

- 오름차순, 내림차순 배열에서 가장 처음 값을 pivot으로 정하면 피벗이 매번 가장 작거나 가장 클 경우이므로 최악 경우 $O(N^2)$의 시간 복잡도가 발생하여 Random 값을 입력한 경우에 비해 훨씬 느린 속도를 보였다.

- Random의 경우에서 다른 정렬들과 같이 String이 가장 느린 수행시간을 보일 것이라고 기대하였지만 Student가 가장 느린 수행시간을 보였다.

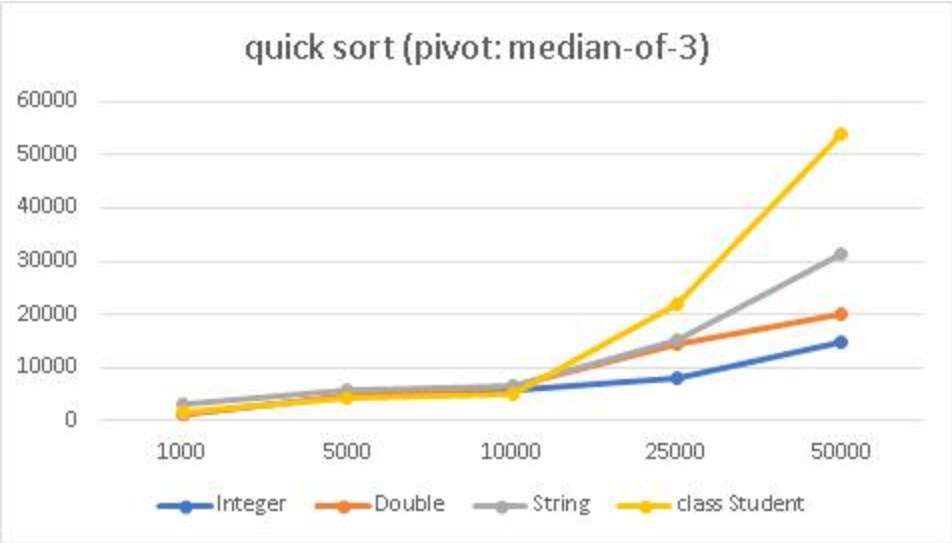
- 오름차순과 내림차순으로 경우에서는 String, Double, Integer, Student 순으로 느린 수행시간을 보였고 Integer와 Double은 거의 유사한 속도를 보였다.

2.4.5 분석 중 오류

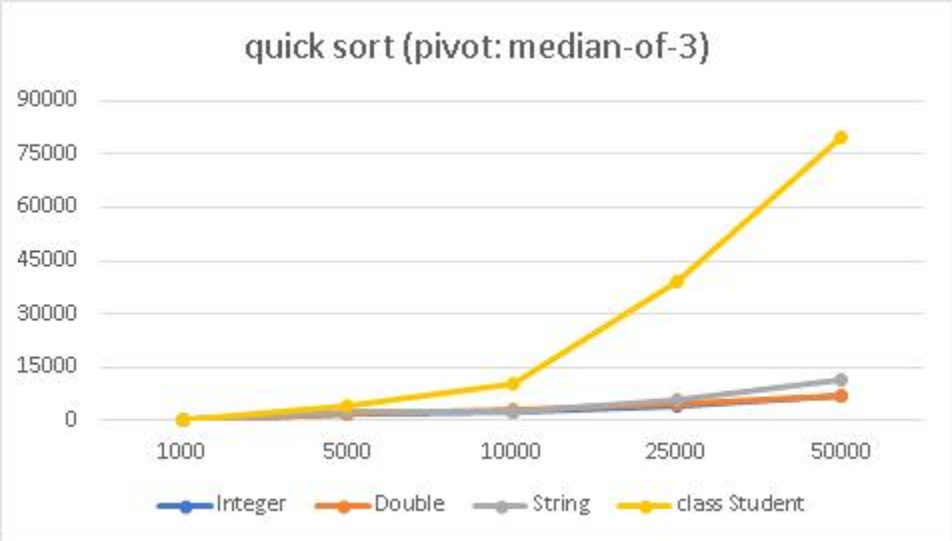
- 스택이 해소되지 못하고 쌓여 StackoverflowError가 20000의 입력부터 발생하기 시작하여 성능을 향상하기 위하여 50 이하의 값에서 Insertion sort를 이용하여 성능향상을 하였지만 해결되지 않았고, IDE 자체의 스택을 수정하여 해결하였다. 부록에서 해소방법을 볼 수 있다.

2.5 Quick Sort (median-of-3)

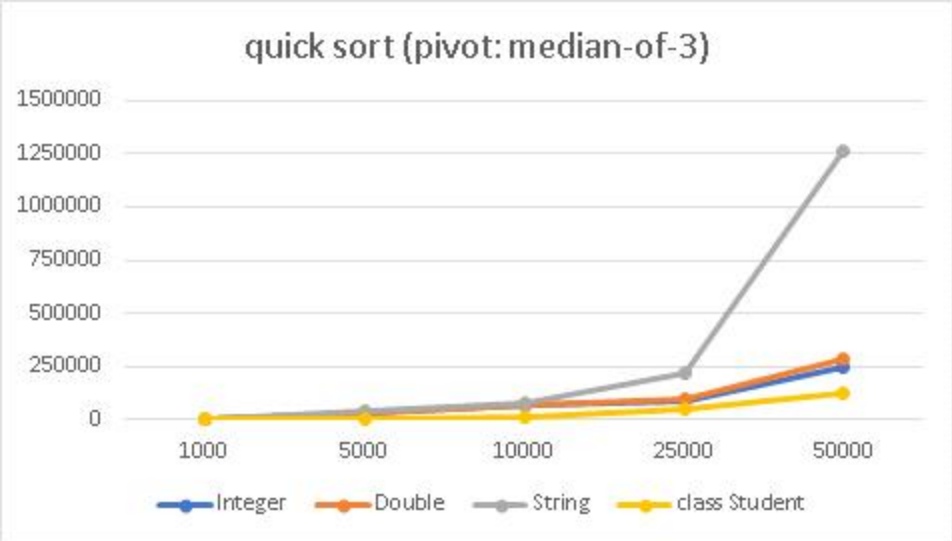
2.5.1 Random



2.5.2 Increase



2.5.3 Decrease



2.5.4 분석

- pivot을 기준으로 분할될 시 한쪽의 크기가 너무 커지는 것을 방지하기 위해 Random 선택된 3개의 값 중 중간값을 pivot으로 사용하여 성능을 개선한 알고리즘이다.

- Random의 경우 String이 가장 느린 속도를 보일 것이라고 예상하였지만 Student, String, Double, Integer 순으로 느린 속도를 보였다.

- Increase의 경우 Decrease와 같이 최악 경우 $O(N^2)$의 시간 복잡도를 가지므로 Decrease와 유사한 속도를 기대하였지만, 오히려 $O(NlogN)$의 시간 복잡도를 가지는 Random과 유사한 속도를 보였다.

- Decrease의 경우 String이 가장 느린 속도를 보였으며 Student가 그다음으로 느린 속도를 보일 것으로 기대하였지만 가장 빠른 속도를 보였다.

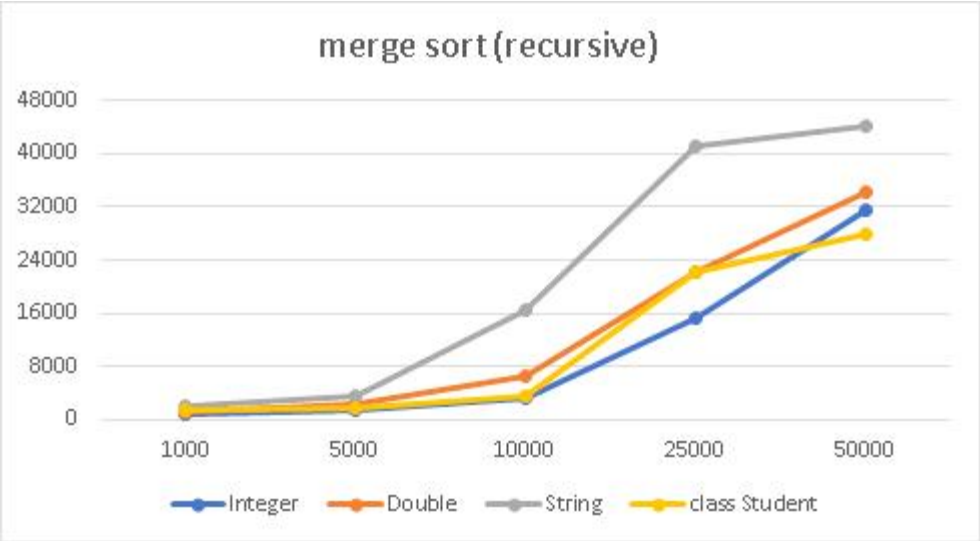
2.5.5 Quick Sort(first pivot) v.s. Quick Sort(median-of-3)

- Random의 경우에는 둘 다 비슷한 속도를 보였지만 최악 경우인 Decrease, Increase의 경우에서는 median-of-3가 first pivot보다 더 빠른 속도를 보였다.

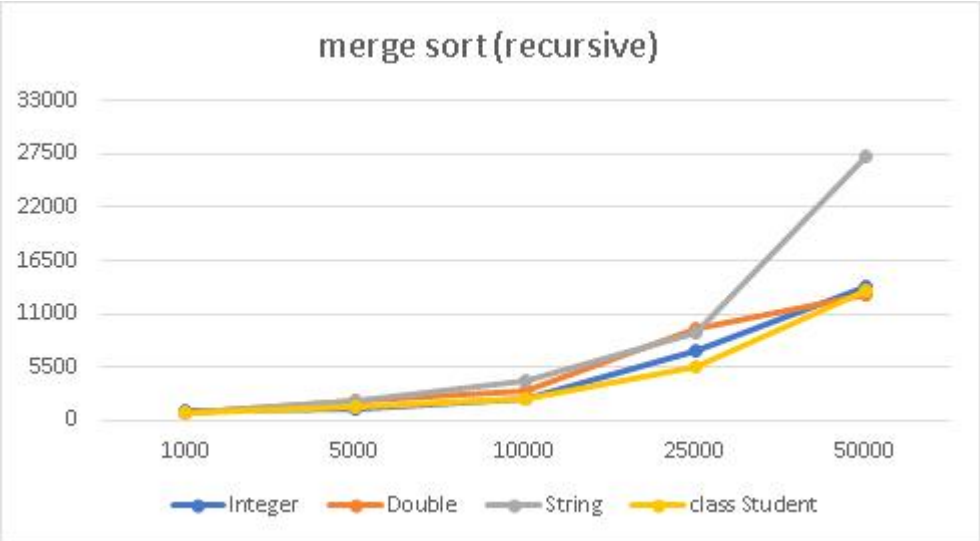
- median-of-3가 first pivot보다 pivot이 가장 큰 경우나 가장 작은 경우, 즉 최악의 경우를 발생시킬 확률을 낮추어 주기 때문에 더 빠른 속도를 내는 것으로 보인다.

2.6 Merge Sort (recursive)

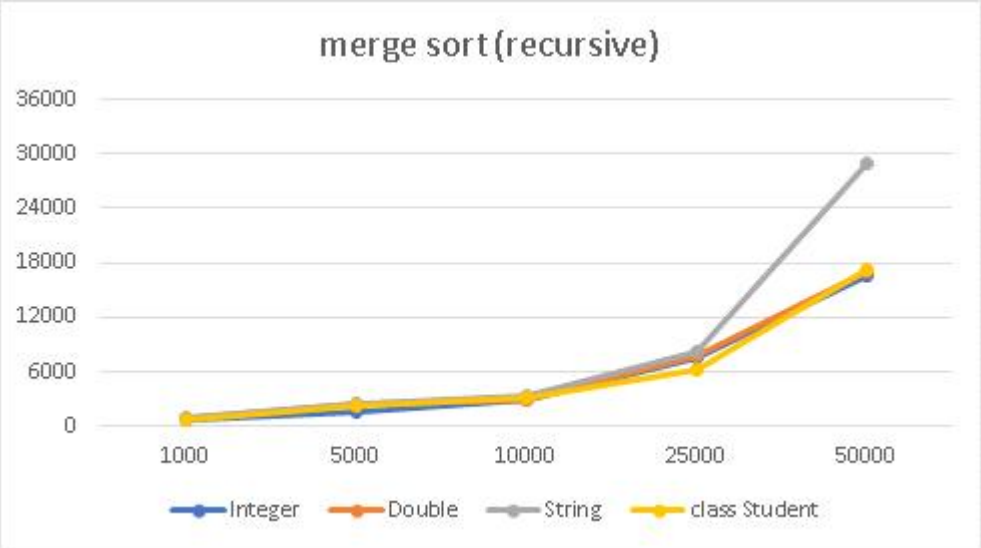
2.6.1 Random



2.6.2 Increase



2.6.3 Decrease



2.6.4 분석

- Merge sort는 크기가 N인 입력을 ½N 크기를 갖는 입력 2개로 분할하고, 각각에 대해 재귀적으로 합병정렬을 수행한 후, 2개의 각각 정렬된 부분을 합병하는 정렬 알고리즘이다.

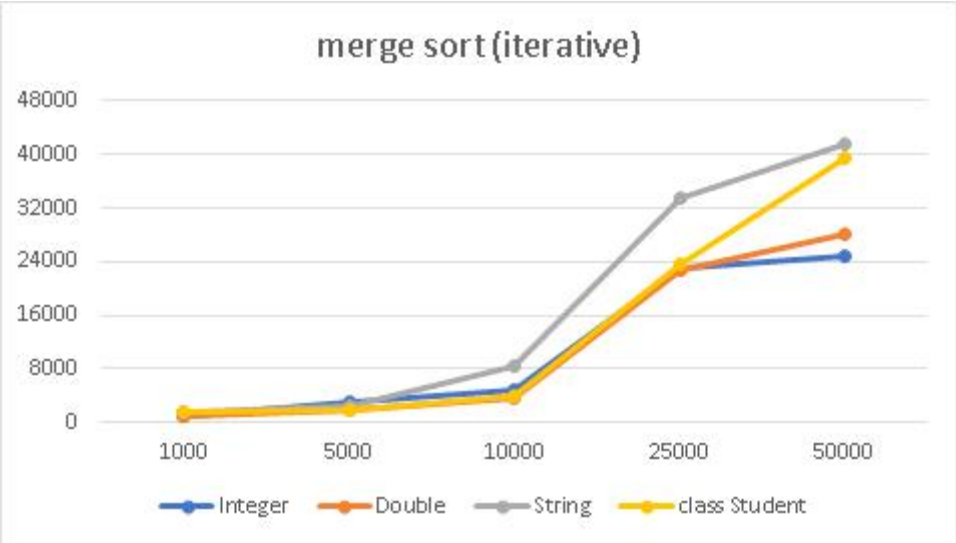
- Random, Increase, Decrease 경우 모두 $O(NlogN)$의 시간 복잡도를 가진다.

- 모든 경우에서 $O(NlogN)$의 시간 복잡도를 가지기 때문에 Random, Increase, Decrease 모두 유사한 속도를 보일 것으로 예상한 것과 같게 비슷한 속도를 보였다.

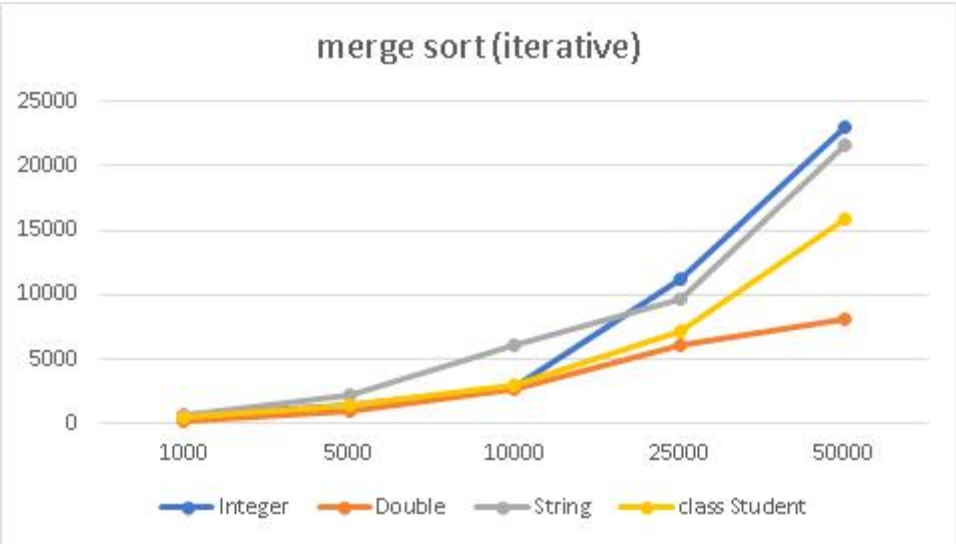
- 모든 경우에서 String이 가장 느린 수행시간을 보였고 Integer, Double, Student는 평균적으로 비슷한 수행시간을 보였다.

2.7 Merge Sort (iterative)

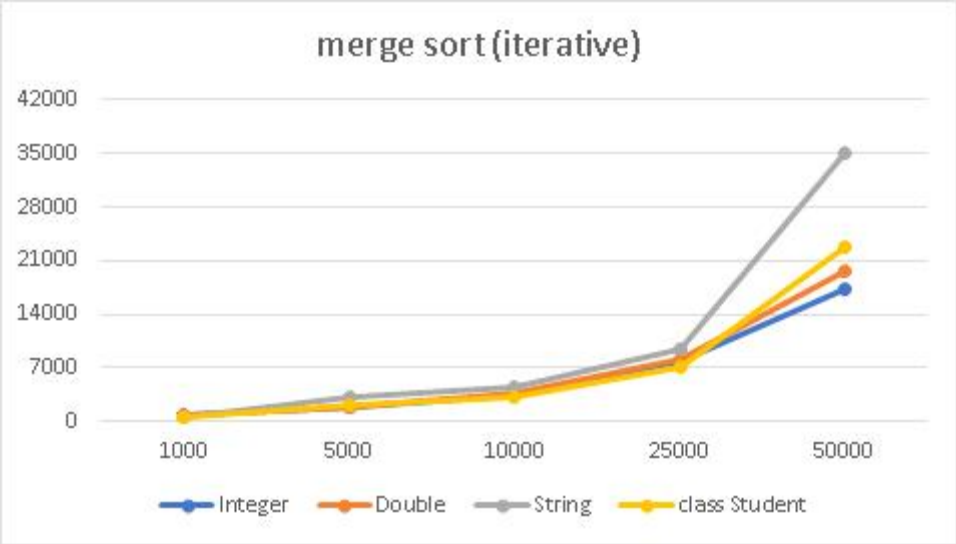
2.7.1 Random



2.7.2 Increase



2.7.3 Decrease



2.7.4 분석

- 분할정복 방식과 배열에서 바로 2개씩 짝지어 합병한 뒤, 다시 4개씩 짝지어 합병하는 상향식 (Bottom-up)으로도 수행 가능한 정렬이다.

- 평균적으로 String, Student 순서대로 수행 속도가 느렸고, Random, Increase 경우에서는 Double, Integer 순서대로 느렸다.

- 시간 복잡도는 $O(NlogN)$이므로 Recursive Merge Sort와 같다.

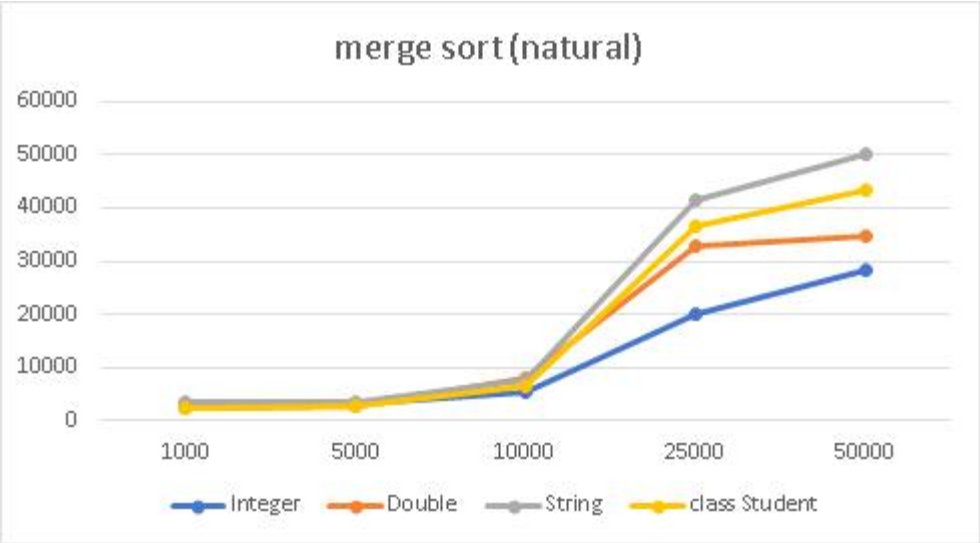
- Recursive를 사용한 Merge Sort와 거의 비슷한 수행시간을 보였다.

- iterative Merge Sort는 recursive Merge Sort에 비교했을 때 입력 크기만큼의 보조 배열을 사용하므로 메모리 효율이 떨어진다.

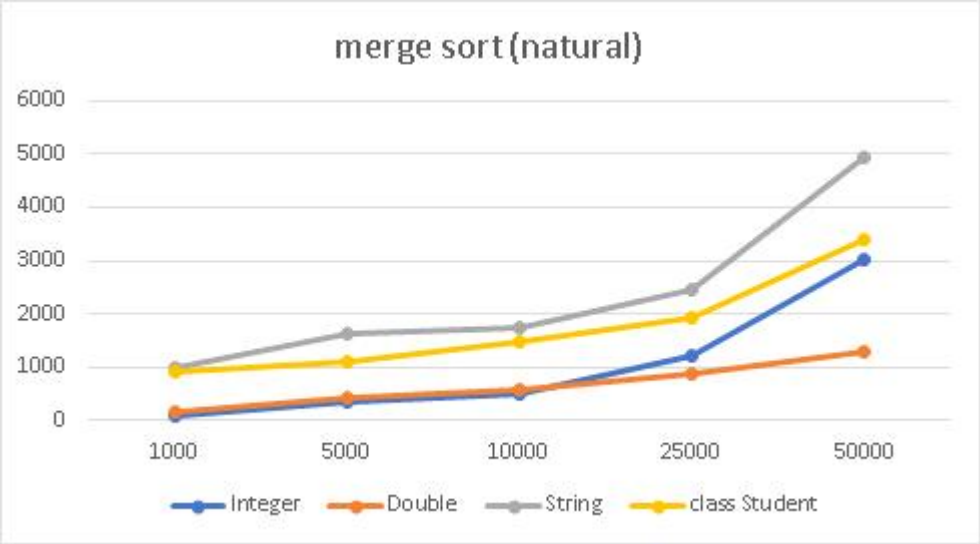
- Increase에서 Integer의 수행시간이 다른 것들에 비해 높게 나왔는데 순간적인 과부하나 IDE 내의 지연으로 보인다.

2.8 Merge Sort (natural)

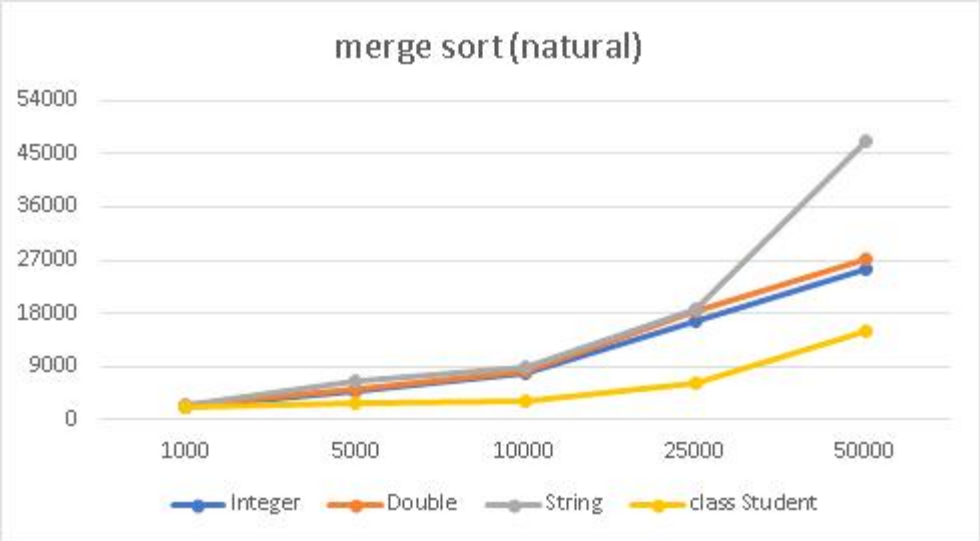
2.8.1 Random



2.8.2 Increase



2.8.3 Decrease



2.8.4 분석

- 주어진 배열 속에 이미 순서에 맞는 원소로 된 부분 배열, 즉 Run만을 합병하는 방식이다.

- Random, Increase의 경우에는 String, Student 순으로 느렸고 값이 25000 이상부터 Random의 경우에서는 Double, Increase에서는 Integer가 느렸다.

- 정렬되어있을 때가 그렇지 않을 때보다 빨랐다. 이미 정렬된 값을 이용하기 때문에 다른 경우에서보다 빠른 것으로 보인다.

- Decrease에서는 String, Double, Integer, Student 순으로 느렸다.

- Random, Decrease의 경우에서 natural, recursive, iterative 순으로 느렸고, Increase의 경우에서는 natural Merge Sort가 가장 빨랐다.

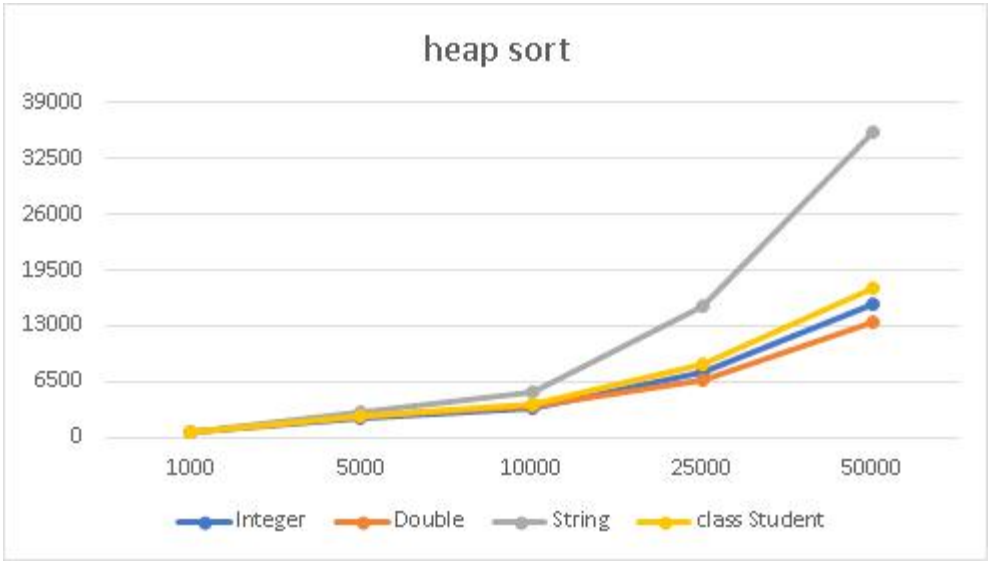
- Increase의 경우에서는 $O(N)$의 시간 복잡도를 가지기 때문에 다른 Merge Sort들과 입력에 대해서 가장 빠른 속도를 보였다.

2.8.5 Merge Sort(recursive) v.s. Merge Sort(iterative) v.s. Merge Sort(natural)

- 수행시간이 $O(NlogN)$의 시간 복잡도를 보장받아 평균적으로 유사한 수행시간을 보였지만 Increase의 경우 natural은 $O(N)$의 시간 복잡도를 가지는 최선의 경우이기 때문에 다른 Merge Sort보다 굉장히 빠른 속도를 보였다.

2.9 Heap Sort

2.9.1 Random



2.9.2 Increase

2.9.3 Decrease

2.9.4 분석

- Heap sort는 max heap 또는 min heap 트리를 이용하여 정렬하는 알고리즘이다.

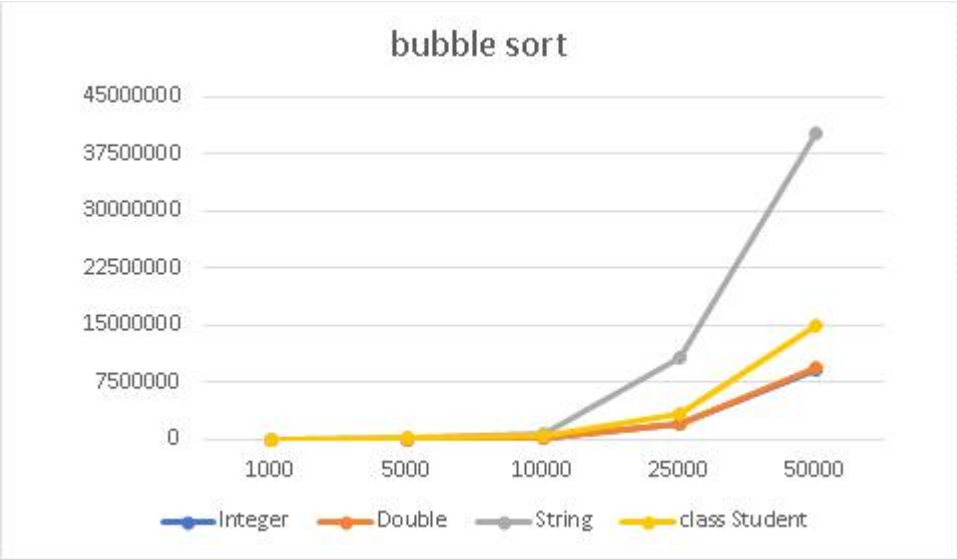
- 어떠한 경우이든지 $O(NlogN)$의 시간 복잡도를 가진다.

- 평균적으로 String, Student, Double, Integer 순으로 느렸다.

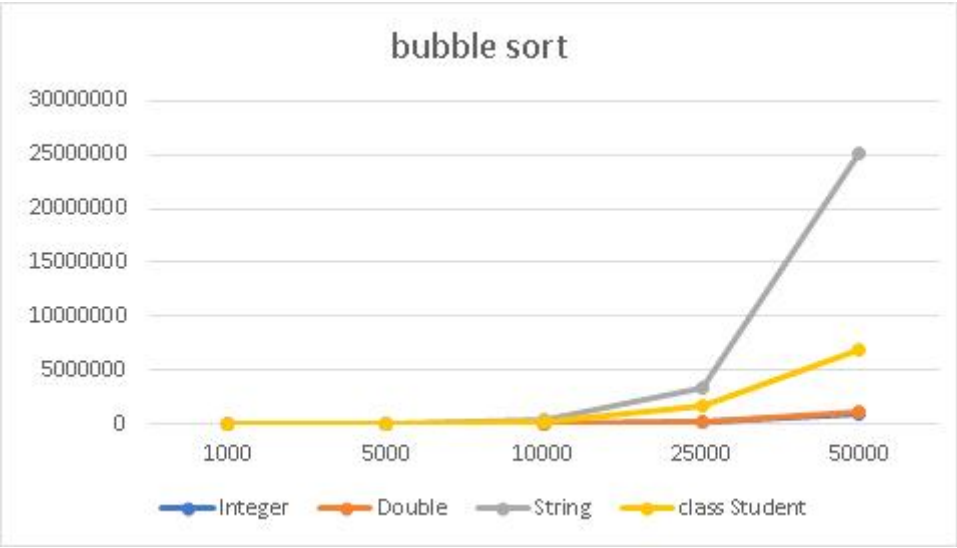
- Decrease, Random, Increase 순서대로 느렸다. Increase의 경우에는 max heap을 만들 때 swap을 한번 실행하면 되지만, Random이나 Decrease는 여러 번 비교하여 swap을 여러 번 실행하기 때문에 더 느리다고 생각한다.

2.10 Bubble Sort

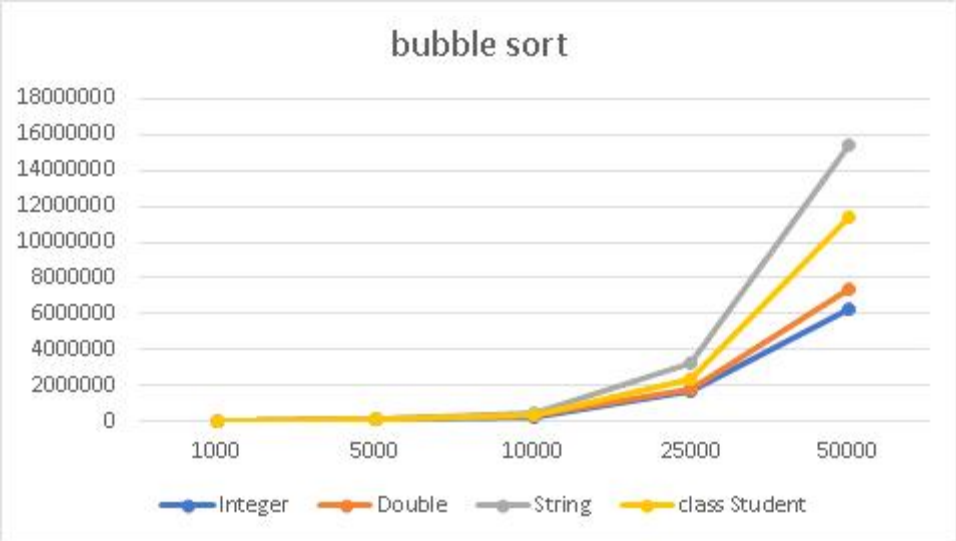
2.10.1 Random



2.10.2 Increase



2.10.3 Decrease



2.10.4 분석

- 서로 인접한 두 개의 값이 정렬되어있지 않으면 정렬하는 것을 반복하는 알고리즘이다.

- 입력된 값에 상관없이 각각의 값을 비교하기 때문에 항상 $O(N^2)$의 시간 복잡도를 가진다.

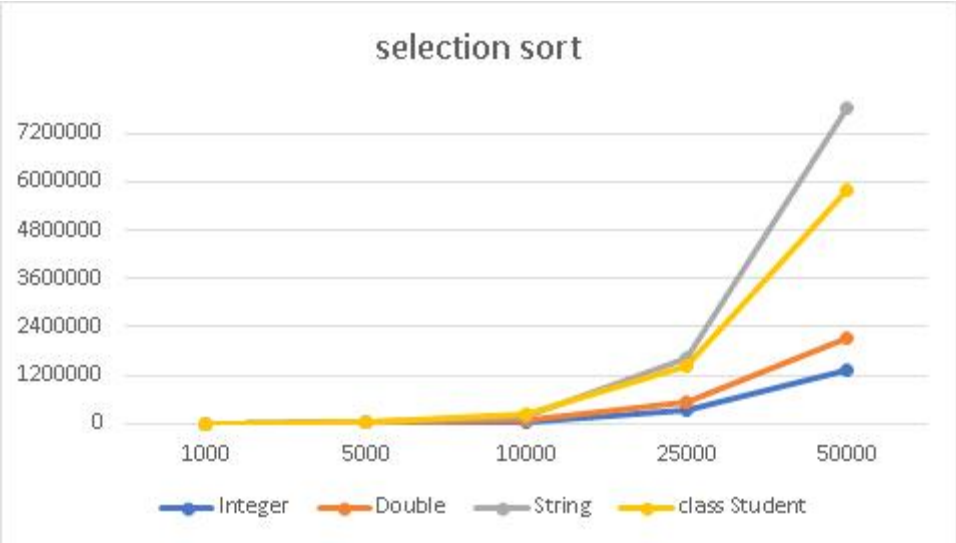
- Random 배열이 내림차순, 오름차순 배열보다 swap 메소드 호출 빈도가 높아 느린 수행시간을 보였다.

- Student, Integer, Double이 매 경우 유사한 수행시간을 보일 것으로 기대했지만, Integer, Double이 매우 유사하고 Student는 그보다 시간이 더 소요되는 모습을 보였다.

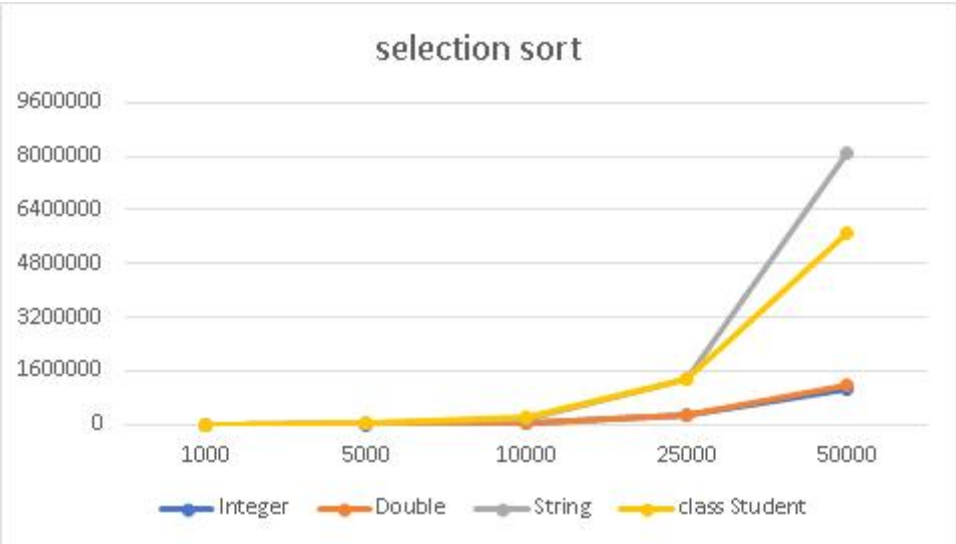
- 내림차순 String 배열에서 값이 50,000개일 때 특이하게 오름차순의 동일 조건보다 더 빠른 속도를 보였다.

2.11 Selection Sort

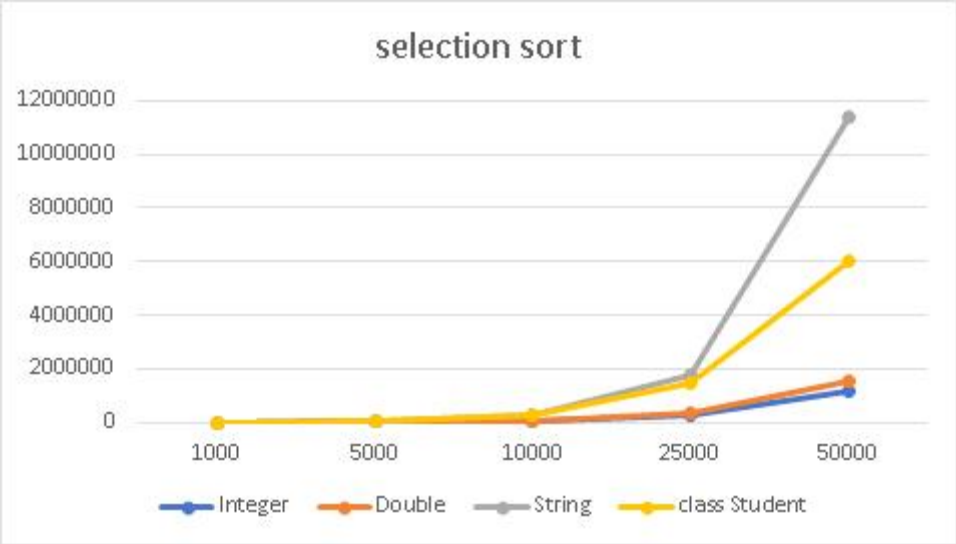
2.11.1 Random



2.11.2 Increase



2.11.3 Decrease



2.11.4 분석

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

- 입력된 값에 상관없이 각각의 값을 비교하기 때문에 항상 $O(N^2)$의 시간 복잡도를 가진다.

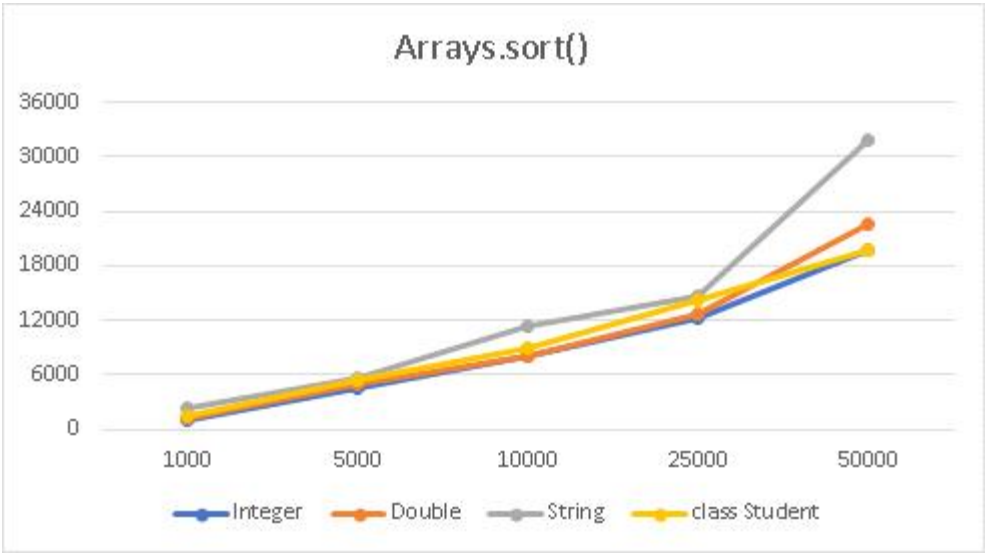
- Integer, Double 자료형의 경우 Increase, Decrease, Random 배열 순으로 속도가 느렸다.

- Student는 Increase 경우에만 속도가 빠른 편이었으며, Decrease와 Random 경우에서는 비슷한 수행시간을 보인다.

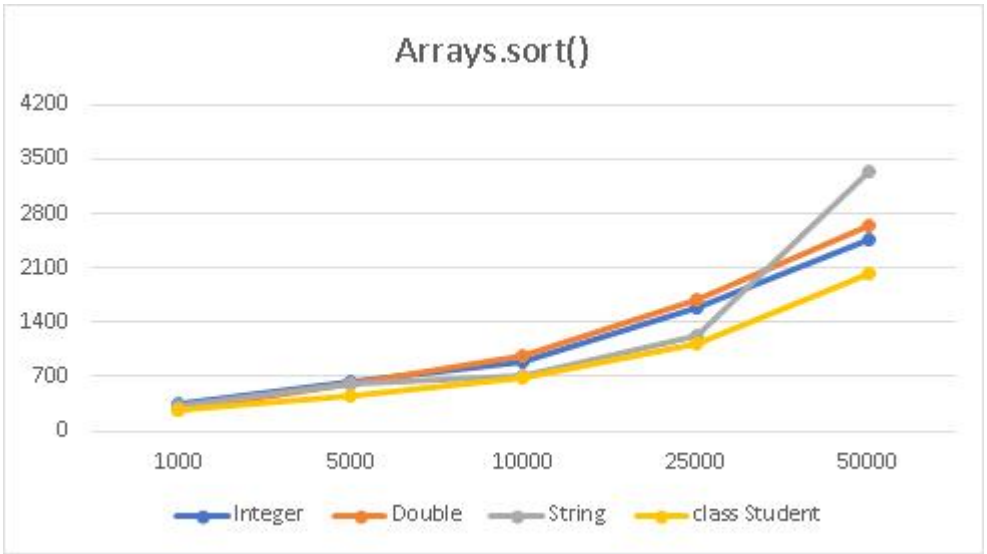
- 항상 $O(N^2)$의 시간 복잡도를 가짐에도 불구하고, String 자료형의 경우 Increase 경우를 제외하고 수행시간이 눈에 띄게 느려지는 모습을 보였다.

2.12 Arrays.sort()

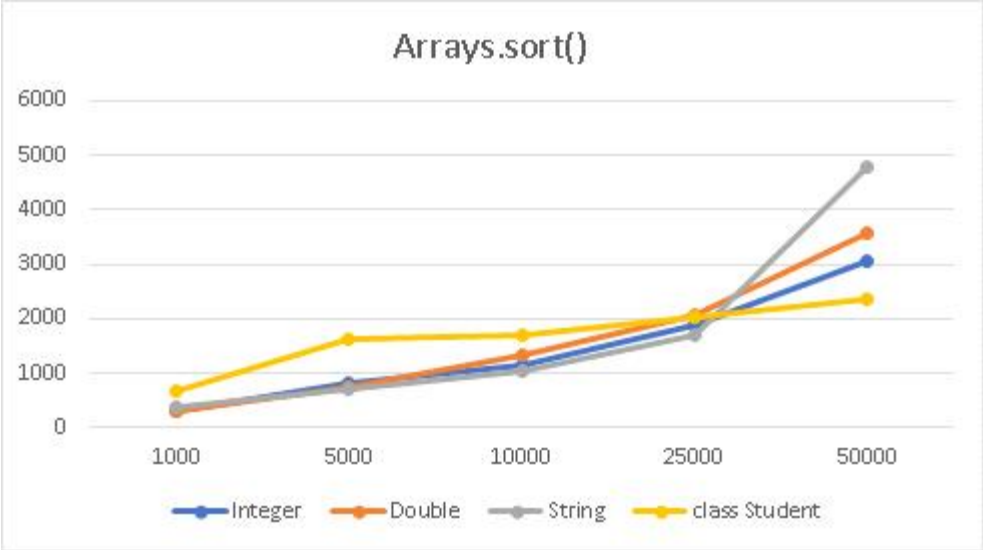
2.12.1 Random



2.12.2 Increase



2.12.3 Decrease



2.12.4 분석

- java.util.Arrays 클래스의 static 메소드 sort(Object[] a)를 사용하고 배열을 이루는 각 원소는 Comparable을 상속받는다.

- 위 메소드는 기본 자료형 배열 정렬 시 보통 Dual-Pivot Quick Sort를 사용하고 원소 개수가 많을 때는 Count Sort를 사용하기도 하며, Object 배열 정렬 시 Tim Sort()를 채택하지만, Merge Sort를 사용하도록 설정할 수 있다. 본 분석에는 Object 배열이 쓰였으므로 Tim Sort가 채택된 Arrays.sort()의 결과를 기술했다.

- Tim Sort는 Insertion Sort와 Merge Sort를 결합하여 만든 stable sort이며, 최선의 경우 $O(N)$, 평균 또는 최악 경우 $O(NlogN)$의 수행시간을 가진다.

- 오름차순 배열에서 어떤 경우라도 다른 배열 정렬보다도 빠른 결과를 보이는 것으로 미루어 Tim Sort의 최선의 경우가 정렬된 배열임을 알 수 있다.

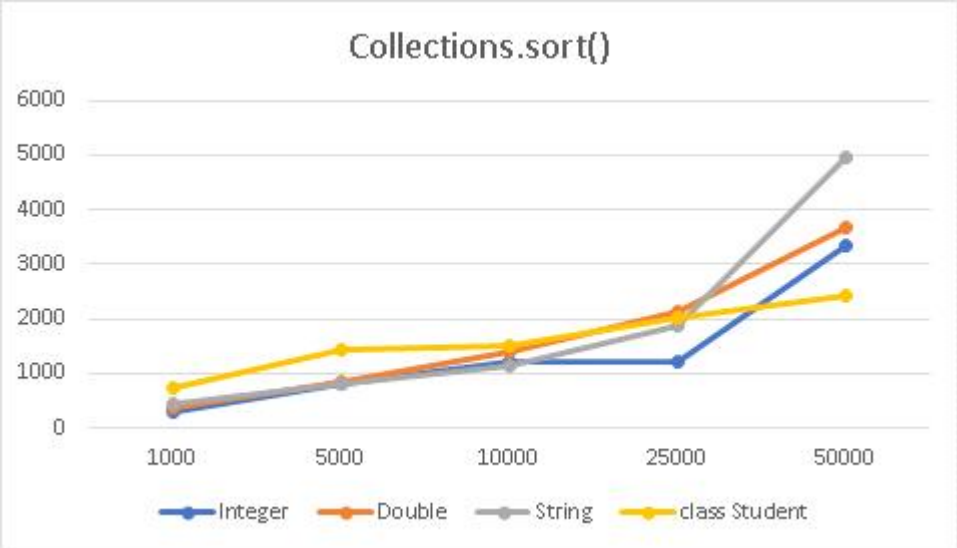
- 다른 자료형과 비교하여 String이 원소 개수에 따른 수행시간 변동 폭이 커 원소가 50,000개일 때 수행시간이 대폭 증가하는 것으로 보인다.

2.13 Collections.sort()

2.1.1 Random

2.1.2 Increase

2.1.3 Decrease



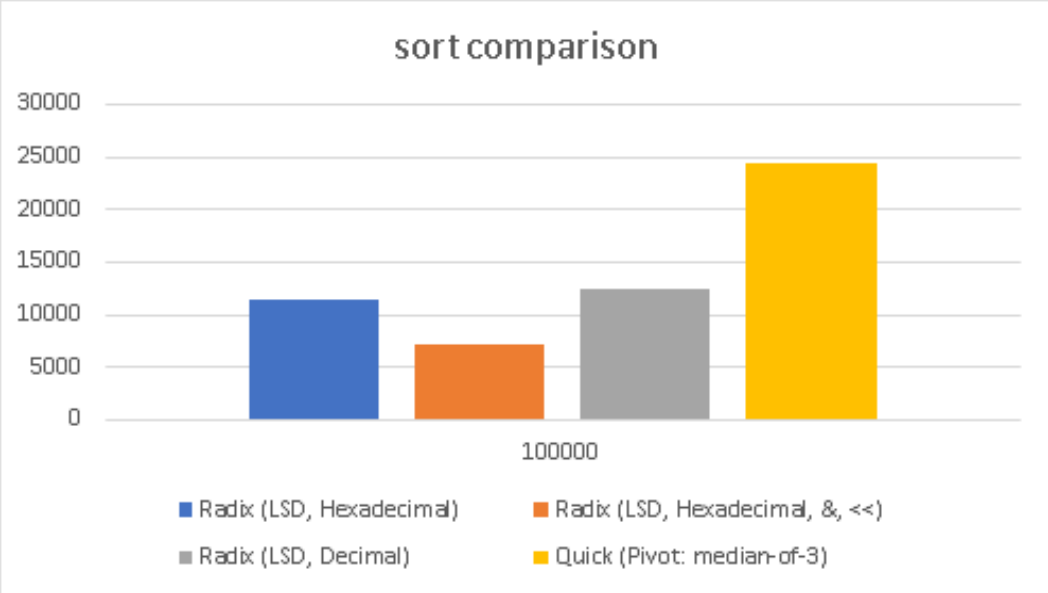
2.1.4 분석

- java.util.Collections 클래스의 static 메소드 sort(List<T> list)를 사용하는데, 이 메소드는 java.util.Arrays.sort()를 List에 사용할 수 있도록 구현한 것이다.

- String의 경우 원소 개수가 커질수록 그에 따른 편차가 다른 자료형보다 커서, 원소 개수 50,000개일 때 속도가 크게 느려지는 모습을 보인다.

- 내림차순 배열에서 String, Double, Student, Integer 순서로 속도가 느릴 것으로 예측했으나 원소의 개수에 따라 변동이 발생했고, 원소 10,000개까지는 Student가 예상외로 느린 모습을 보였다.

3. Comparison based v.s. Non-comparison based & Arithmetic operation v.s. Logical operation



3.1 분석

- 위의 결과로 나타나 있듯이 masking과 shift 연산을 사용한 16진수 Radix sort가 가장 빨랐고, 그다음으로 16진수, 10진수 modular 연산을 사용한 radix sort, 가장 느린 것은 Quick Sort였다.

- radix sort는 기수에 따라 원소를 버킷에 집어넣기 때문에 비교 연산이 불필요하다. 따라서 전체 시간 복잡도는 $O(d(N+R))$(d: 키의 자릿수, N: 입력의 크기, R: radix)이다. 즉 $O(N)$만큼의 복잡도를 가지므로 $O(NlogN)$의 시간 복잡도를 갖는 다른 정렬 방식보다 월등히 빠르다. 따라서 위와 같은 결과가 나온 것으로 보인다.

- masking과 shift 연산을 사용한 Radix sort가 modular 연산자를 사용한 radix sort보다 빠른 이유는 modular와 같은 산술 연산자보다 shift, masking과 같은 비트 연산자가 계산 수행 속도가 더 빠르기에 위와 같은 결과가 나온 것으로 보인다.

부록. Quick sort (first pivot)의 first pivot에서의 StackoverflowError 시도 방법 및 해결 방법

- 첫 번째 시도 방법

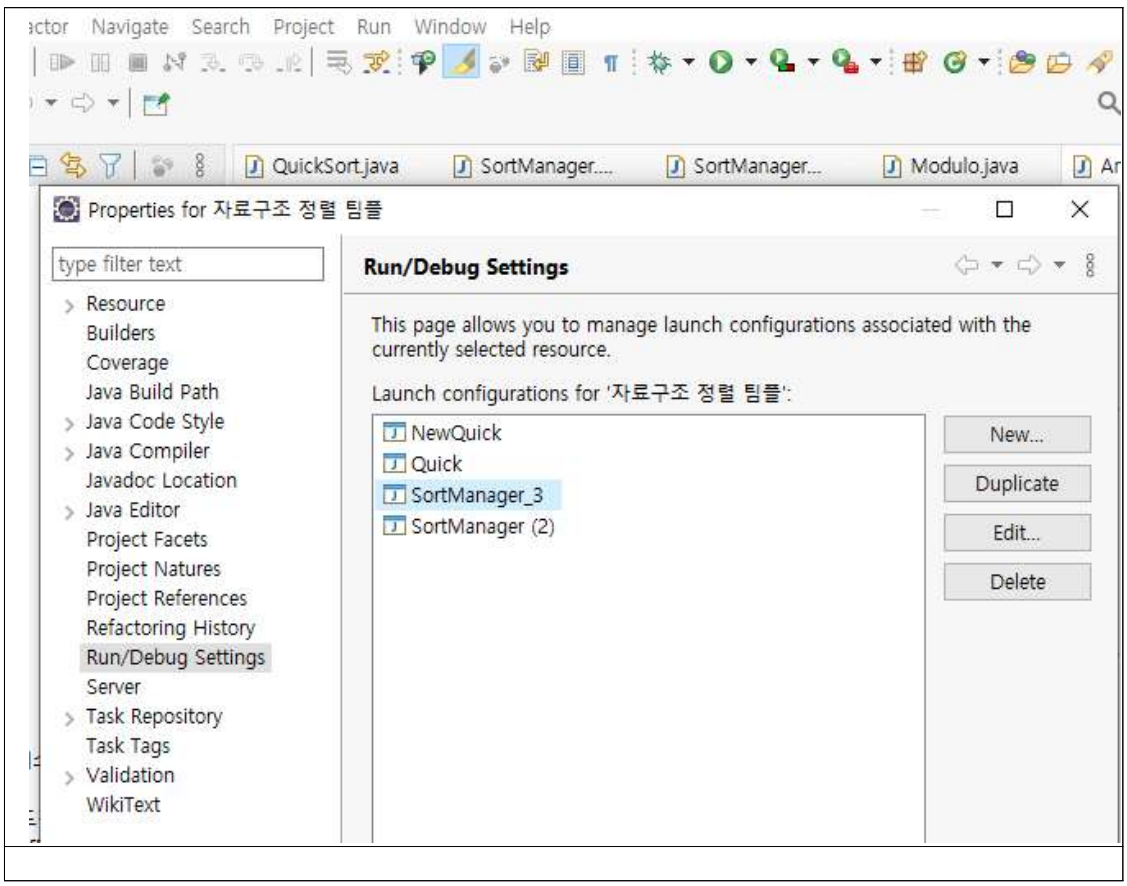
Insertion Sort를 50 이하의 값에서 호출되게 하여 성능의 향상을 통한 오류 해결을 기대하였지만, 입력 수 20000 이하에서는 수행시간이 미세하게 증가하는 것을 찾을 수 있었지만 StackoverFlowError는 해결하지 못하였다.

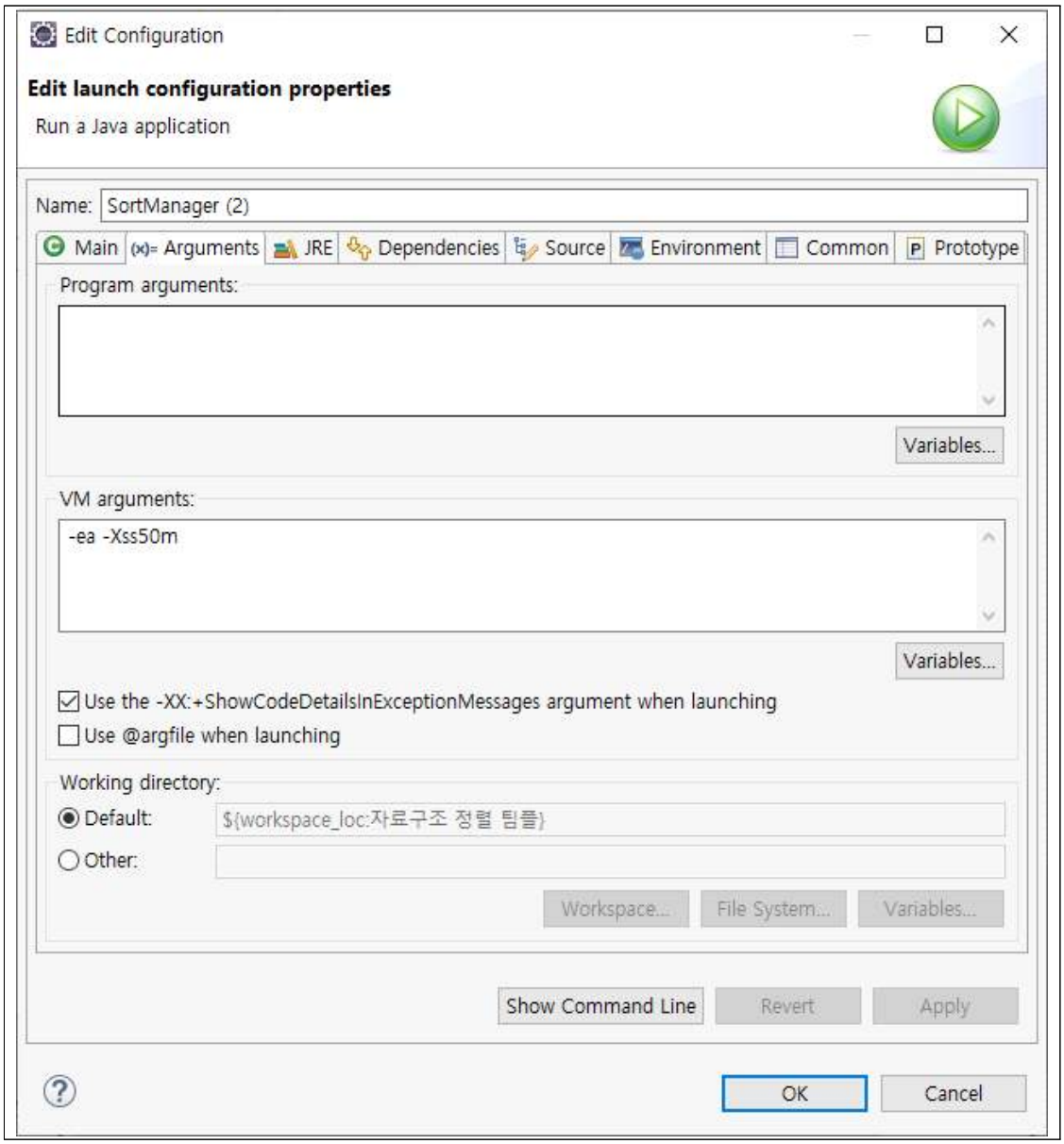
- 두 번째 시도 방법

구현 IDE로 채택한 Eclipse의 eclipse.ini 파일에서의 시스템 스택 수정을 통한 해결을 시도하였지만, Heap size는 수정됨에도 불구하고 Stack size는 변경되지 않았다.

- 세 번째 시도 방법(해결 방법)

Eclipse의 툴바 중 Project -> Properties -> Run/Debug Settings에서 현재 정렬을 수행하는 main 클래스를 선택 후 Edit를 클릭하고 Arguments에 들어가 VM arguments에 "-ea -Xss50m"를 입력하여 StackoverflowError를 해결하였다.




Edit Configuration
Edit launch configuration properties
Run a Java application
Name: SortManager (2)
Main
Arguments
JRE
Dependencies
Source
Environment
Common
Prototype
Program arguments:
Variables...
VM arguments:
-ea -Xss50m
Variables...
Use the -XX:+ShowCodeDetailsInExceptionMessages argument when launching
Use @argfile when launching
Working directory:
Default:
${workspace_loc:자료구조 정렬 팀플}
Other:
Workspace...
File System...
Variables...
Show Command Line
Revert
Apply
OK
Cancel