알고리즘 보고서

- [AL12] 정렬 성능 측정 (2)

충남대학교 컴퓨터공학과
알고리즘 04분반
학번: 201701975
이름: 구건모

> 주요 이론 및 과제에서 해야할 일

이번 과제의 주요 이론은 QuickSort의 여러가지 Pivot 선택 방법과 InsertionSort를 이용한 QuickSort로,QuickSort를 수행할 때 들어온 데이터에 대한 pivot 설정이 정렬 성능에 어떤 영향을 미치는지를 측정을 통해 확인해 보는 것이었습니다. 따라서 이번 과제에서 각 Left, Mid, Median, Random 으로 피봇을 설정하여 측정해보기 위해서 각각의 pivot에 따른 QuickSort를 수행하는 클래스를 구현하고, InsertionSort를 이용하여 데이터 크기가 작을 때는 InsertionSort를 이용하고 데이터 크기가 커지면 QuickSort를 이용하는 QuickSortWithInsertionSort를 구현하는 것이 과제의 주요 내용이었습니다. 따라서 위의 기능들을 수행하는 클래스와 이를 측정하기 위한 Experiment 관련 클래스를 구현하는 것이 이번 과제에서 해야할 일이라고 볼 수 있습니다.

> 주요 구현 내용

[class - QuickSort]

```
protected int partition(E[] aList, int left, int right) {
    int pivot = this.pivot(aList, left, right);
    this.swap(aList, left, pivot);
    E pivotElement = aList[left];
    int toRight = left;
    int toLeft = right + 1;
    do {
        do {
            toRight++;
        } while (this.compare(aList[toRight], pivotElement) < 0);
        do {
            toLeft--;
        } while (this.compare(aList[toLeft], pivotElement) > 0);
        if (toRight < toLeft) {
            this.swap(aList, toRight, toLeft);
        }
    } while (toRight < toLeft);
    this.swap(aList, left, toLeft);
    return toLeft;
}

protected void quickSortRecursively(E[] aList, int left, int right) {
    if (left < right) {
        int mid = partition(aList, left, right);
        quickSortRecursively(aList, left, mid - 1);
        quickSortRecursively(aList, mid + 1, right);

    }
}
```

QuickSort는 pivot을 설정하고 설정한 pivot을 기준으로 작은 값과 큰값을 나누어 partition 한 후에, 각 partition에 대해서 recursive 하게 QuickSort를 수행하는 방식으로 진행되므로 들어온 리스트가 partitioning 되다가 크기가 1이 될 때까지 수행하면 모든 데이터가 정렬된 상태가 됩니다. 따라서 QuickSort가 동작하는 방식인 divide & conquer를 수행하도록 partition 메서드와 QuickSortRecursively 메서드가 구현되어 있습니다.

> 주요 구현 내용

[class - QuickSortByPivotLeft]

```java
package sort;

public class QuickSortByPivotLeft <E extends Comparable<E>> extends QuickSort<E> {

    public QuickSortByPivotLeft(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    @Override
    protected int pivot(E[] aList, int left, int right) {
        // TODO Auto-generated method stub
        return left;

    }


}
```

QuickSort에서 pivot값을 left를 반환하도록 override 하여 구현된

QuickSortByPivotLeft 클래스 입니다.

[class - QuickSortByPivotMid]

```java
package sort;

public class QuickSortByPivotMid<E extends Comparable<E>> extends QuickSort<E> {

    public QuickSortByPivotMid(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    protected int pivot(E[] aList, int left, int right) {
        return ((left + right) / 2);
    }

}
```

QuickSort 에서 Pivot 값이 가운데 위치에 있는 값을 pivot 으로 설정하도록 override 하여 구현된 QuickSortByPivotMid 클래스 입니다.

> 주요 구현 내용

[Class - QuickSortByPivotMedian]

```
@Override
protected int pivot(E[] aList, int left, int right) {
    // TODO Auto-generated method stub
    if ((right - left) < 3) {
        return left;
    }
    int mid = (left + right) / 2;
    if (this.compare(aList[left], aList[mid]) < 0) {
        if (this.compare(aList[mid], aList[right]) < 0) {
            return mid;
        } else {
            if (this.compare(aList[left], aList[right]) < 0) {
                return right;
            } else {
                return left;
            }

        }
    } else {
        if (this.compare(aList[mid], aList[right]) < 0) {
            return left;
        } else {
            if (this.compare(aList[mid], aList[right]) < 0) {
                return right;
            } else {
                return mid;
            }

        }
    }

}
```

QuickSort에서 left, mid, right 값을 비교하여 중앙값을 pivot으로 설정하는 방식으로 pivot 설정 시 가장 큰 값이 pivot으로 설정되는 경우를 방지하므로써 정렬된 데이터를 quickSort로 Sorting할 때 최악의 성능이 나오는 것을 피할 수 있도록 해줍니다.

> 주요 구현 내용

[Class - QuickSortWithInsertionSort]

```
    private boolean insertionSort(E[] aList, int left, int right) {
        for (int i = (right - 1); i >= left; i--) {
            E insertedElement = aList[i];
            int j = i + 1;
            while (this.compare(aList[j], insertedElement) < 0) {
                aList[j - 1] = aList[j];

                j++;
            }
            aList[j - 1] = insertedElement;
        }

        return true;
    }

    @Override
    protected void quickSortRecursively(E[] aList, int left, int right) {
        int currentSize = right - left;
        if (currentSize > 0) {
            if (currentSize <= this.maxSizeForInsertionSort()) {
                this.insertionSort(aList, left, right);

            } else {
                int mid = partition(aList, left, right);
                quickSortRecursively(aList, left, mid - 1);
                quickSortRecursively(aList, mid + 1, right);
            }
        }
    }
```

QuickSortWithInsertionSort는 정렬하고자 하는 데이터의 크기가 일정 크기보다 작으면 삽입정렬을 사용하고 아닌 경우 기존처럼 QuickSort를 사용하게 되는 방식입니다. 따라서 QuickSortRecursively 메서드를 살펴보면 현재 정렬하고자 하는 데이터의 크기에 따라 insertionSort 또는 quickSort를 수행하도록 조건을 주어 구현되어 있습니다.

삽입정렬은 데이터간의 비교횟수가 적고, 교환횟수가 많은 편이기 때문에 자료가 어느정도 정렬되어 있거나 정렬해야 하는 데이터량이 적을 때는 QuickSort의 재귀의 깊이를 줄여서 재귀호출로 인한 메모리 문제등을 개선해 주는등의 줄여주는 좋은 성능을 보여주지만, 데이터 량이 많아질수록 교환 과정에서 시간을 많이 소요하게 되기 때문에 QuickSort가 더 좋은 성능을 보여주게 됩니다. 따라서 QuickSortWithInsertionSort 에서는 이러한 장단점을 취합하여 데이터의 size를 가지고 삽입정렬 또는 퀵정렬을 선택하도록하여 개선한 것으로 볼 수 있습니다.

> 결과 화면

> 결과 화면

[결과 - 삽입, 퀵 힙 정렬 결과]

```
_Main_AL12_S2_201701975_구건모 [Java Application] C:₩Users₩gmku1₩.p2₩pool₩plugins₩org.eclipse.justj.openjdk.hotspo
<<< 정렬 성능 비교 프로그램을 시작합니다 >>>

>> 3 가지 정렬의 성능 비교: 삽입, 퀵, 힙 <<
> 무작위 데이터에 대한 측정:
                <Insertion Sort>          <Quick Sort>            <Heap Sort>
 데이터 크기        Measure (Estimate)       Measure (Estimate)       Measure (Estimate)
[   1000]            712 (     649)           122 (      74)           871 (     145)
[   2000]           2783 (    2596)           187 (     164)          1626 (     320)
[   3000]           6056 (    5842)           431 (     260)          2477 (     506)
[   4000]          10493 (   10386)           509 (     359)          2076 (     699)
[   5000]          17192 (   16229)           528 (     461)           767 (     898)
[   6000]          24161 (   23370)           513 (     565)          1150 (    1101)
[   7000]          32052 (   31809)           682 (     671)          1225 (    1307)
[   8000]          41865 (   41547)           798 (     779)          1650 (    1516)
[   9000]          51840 (   52583)           811 (     887)          1671 (    1728)
[  10000]          64918 (   64917)           998 (     998)          1943 (    1943)

> 오름차순 데이터에 대한 측정:
                <Insertion Sort>          <Quick Sort>            <Heap Sort>
 데이터 크기        Measure (Estimate)       Measure (Estimate)       Measure (Estimate)
[   1000]              8 (       6)           777 (     370)           112 (      82)
[   2000]             13 (      12)          1590 (    1481)           208 (     181)
[   3000]             18 (      18)          3235 (    3333)           537 (     286)
[   4000]             26 (      25)          5895 (    5926)           399 (     395)
[   5000]             30 (      31)          9113 (    9260)           679 (     507)
[   6000]             37 (      37)         12620 (   13334)           818 (     622)
[   7000]             44 (      44)         18219 (   18149)           935 (     738)
[   8000]             49 (      50)         24416 (   23705)          1069 (     857)
[   9000]             55 (      56)         29283 (   30002)          1051 (     976)
[  10000]             63 (      63)         37040 (   37040)          1098 (    1098)

> 내림차순 데이터에 대한 측정:
                <Insertion Sort>          <Quick Sort>            <Heap Sort>
 데이터 크기        Measure (Estimate)       Measure (Estimate)       Measure (Estimate)
[   1000]           1329 (    1331)           369 (     337)           125 (      72)
[   2000]           5597 (    5327)          1371 (    1350)           202 (     160)
[   3000]          11514 (   11987)          2960 (    3038)           285 (     253)
[   4000]          21111 (   21311)          5416 (    5401)           430 (     350)
[   5000]          39940 (   33298)          8373 (    8440)           517 (     449)
[   6000]          47487 (   47949)         12079 (   12154)           610 (     551)
[   7000]          64670 (   65265)         16325 (   16543)           713 (     654)
[   8000]          84919 (   85244)         21507 (   21607)           963 (     759)
[   9000]         108211 (  107887)         27347 (   27347)           940 (     865)
[  10000]         133194 (  133194)         33762 (   33762)           973 (     973)
```

> 결과 화면

[결과 - 5가지 퀵 정렬 성능 비교]

```
>> 5 가지 퀵 정렬 버전의 성능 비교 <<
> 무작위 데이터에 대한 측정:
 데이터 크기       <Pivot Left>      <Pivot Mid>   <Pivot Median>   <Pivot Random> <Insertion Sort>
[   1000]              182              118              180              121              414
[   2000]              324              248              360              237              635
[   3000]              696              437              537              361              751
[   4000]              675              731              714              529              998
[   5000]              835              712              929              669             1278
[   6000]             1133              898             1051              783             1533
[   7000]             1378             1261             1287             1011             1781
[   8000]             1525             1540             1480             1123             2138
[   9000]             1638             1836             1490             1235             3285
[  10000]             1556             2068             1350             1322             2534

> 오름차순 데이터에 대한 측정:
 데이터 크기       <Pivot Left>      <Pivot Mid>   <Pivot Median>   <Pivot Random> <Insertion Sort>
[   1000]              374               49               58               70              220
[   2000]             1356               34               53              137              130
[   3000]             2960               58               70              199              165
[   4000]             5392               79               99              270              184
[   5000]             8321               96              267              337              211
[   6000]            13642              119              308              422              251
[   7000]            17143              136              352              478              298
[   8000]            20441              153              409              536              333
[   9000]            25774              175              458              378              372
[  10000]            31795              208              489              426              409

> 내림차순 데이터에 대한 측정:
 데이터 크기       <Pivot Left>      <Pivot Mid>   <Pivot Median>   <Pivot Random> <Insertion Sort>
[   1000]              433               43              228               40               61
[   2000]             1650               38              817               76              222
[   3000]             3630               65             1860              115              168
[   4000]             6384               79             3095              153              242
[   5000]            10015              105             4602              209              283
[   6000]            13364              132             6486              248              337
[   7000]            19186              156             9656              283              490
[   8000]            25072              165            12544              318              574
[   9000]            30326              207            14836              361              725
[  10000]            37379              220            18762              402              584
```

- 생각해 볼 점에서 해당 결과에 대해 분석한 내용이 반영되어 있습니다.

> 결과 화면

[결과 - 데이터 크기별 삽입정렬 성능 비교]

```
>> 삽입 정렬을 사용하는 퀵 성렬의 성능: 삽입 정렬을 실행하는 크기별 성능을 비교 <<
> 무작위 데이터에 대한 측정:
 데이터 크기<Pivot Random>  <Size 10>  <Size 20>  <Size 30>  <Size 40>  <Size 50>  <Size 60>  <Size 70>  <Size 80>  <Size 90>
[  10000]           969        837        809        915       1436       1192       1106       1059       1178       1299
[  20000]          2047       1829       1866       2052       2249       2794       2180       2269       2443       2723
[  30000]          3320       3055       3068       3301       5388       4095       3197       3919       3967       4022
[  40000]          4825       4169       4436       4471       6765       5084       4713       5119       5874       5479
[  50000]          5665       5335       6317       6107       5873       6937       6319       8581       7606       7690
[  60000]          7464       6684      10840      10461       8541       7784       7993      10009       8563       9012
[  70000]          8972       7989      10193      12432      11167       9306       9022       9301      11102      10805
[  80000]         10342       9044      11777      12624      13531      10450      10382      11201      12551      12274
[  90000]         11327      10080      21175      15743      11510      11630      12011      12743      13839      14227
[ 100000]         12928      11624      14230      13696      16039      14219      13733      13935      16241      14622
> 무작위 데이터에 대한 측정:
 데이터 크기<Pivot Random>  <Size 15>  <Size 16>  <Size 17>  <Size 18>  <Size 19>  <Size 20>  <Size 21>  <Size 22>  <Size 23>  <Size 24>  <Size 25>
[  10000]           971        846        971       1102        967       1062        981        916       1001        879        987        983
[  20000]          2214       1960       2084       2281       2041       2241       2099       2007       2236       1935       1992       2143
[  30000]          3524       3301       3970       3752       3224       4002       3359       3255       3945       3186       3322       3202
[  40000]          4665       4403       6061       4565       4505       5109       4696       4652       4709       4431       4365       4313
[  50000]          6438       5694       7557       5563       5868       5770       5989       6032       6088       5504       5768       8509
[  60000]          7404       6877       8414       6649       7204       7160       7262       7281       7517       7394       7259       9267
[  70000]          9221       7976      10512       8496       8469       9748       8582      10113       8484       8598       8656       9339
[  80000]         10654       9737      10198      10161       9749      10553      10168       9886      10184      10341      10169      10340
[  90000]         11598      11780      11681      10900      10795      12103      12389      11001      11506      11677      10832      11882
[ 100000]         12451      14065      14047      14506      12304      12926      12119      13155      12704      12417      12576      13206
<<< 정령 성능 비교 프로그램을 종료합니다 >>>
```

- 생각해 볼 점에서 해당 결과에 대해 분석한 내용이 반영되어 있습니다.

> 생각해 볼 점

> 생각해 볼 점

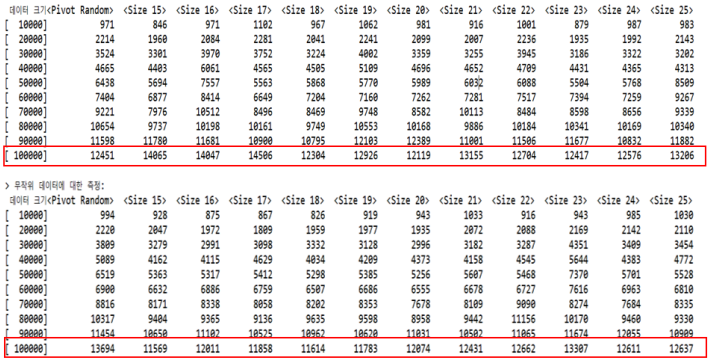
1. 퀵정렬에서 피봇의 선택이 성능에 미치는 영향

퀵 정렬의 성능은 랜덤하게 배치된 데이터의 경우에 여러가지 알려진 정렬방법들 중 좋은 편에 속하지만, 정렬된 데이터를 Sorting 할 때에는 최악의 시간 복잡도를 가지게 됩니다.

이번 실습결과를 확인해 보면 데이터가 정렬되어있을 때 피봇을 left로 설정했을 때 가장 성능이 떨어지는 것을 볼 수 있습니다. pivot을 기준으로 작은값과 큰값을 분할하고 다시 분할된 부분에 대해 Recursive 하게 수행되는 Divde & Conquer 과정에서 오름차순이나 내림차순 같이 이미 정렬된 데이터의 경우에는 left 값이 정렬할 데이터 중에서 가장 큰값이나 작은값이 되기 때문에 비효율을 초래하게 됩니다. mid 값으로 설정할 경우 오름차순 또는 내림차순과 같이 정렬된 데이터에 대한 case에서는 중앙에 위치한 값이 실제 데이터 들의 가운데 값이고 양쪽 partition을 수행할 때에도, 나뉘어진 두 파티션 또한 정렬된 상태이기 때문에 이러한 성질은 동일하므로 divide 만 일어나고 swap이 일어나지 않아서 다른 pivot 보다 좋은 성능을 보여줍니다. Median의 left, mid, right의 세 값을 비교하여 가운데 값을 이용하는데 오름차순으로 정렬시, 기존에 내림차순으로 정렬된 데이터이기 때문에 작은 값들이 다 우측에 몰려있으므로 Divide & Conquer 과정에서 swap이 많이 일어나게 됩니다. 하지만 pivot을 left로 설정한 것보다는 나은 것으로 보아서 최악의 경우는 피할 수 있는 것으로 보입니다. pivot을 Random 하게 설정할 때도 정렬된 데이터에 대해 성능을 보면 좋은 성능을 보여주고 있는데, 정렬된 데이터에 대한 pivot = left일 때의 성능이 최악의 경우이기 때문에 랜덤하게 택하는 것이 상대적으로 더 좋은 결과를 도출한 것으로 보입니다.
이렇게 정렬하려는 데이터와 pivot을 어떻게 설정할지에 대한 방법들이 quickSort에 성능에 영향을 미치고 있는 것을 확인해 볼 수 있었습니다.

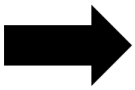
> 생각해 볼 점

2. 작은 구간에서 삽입정렬을 사용하는 퀵정렬의 경우, 삽입 정렬을 사용하는 크기가 어느 정도일 때 전체적인 성능이 좋은지 관찰.



데이터 크기<Pivot Random> <Size 15> <Size 16> <Size 17> <Size 18> <Size 19> <Size 20> <Size 21> <Size 22> <Size 23> <Size 24> <Size 25>
[10000] 971 846 971 1102 967 1062 981 916 1001 879 987 983
[20000] 2214 1960 2084 2281 2041 2241 2099 2007 2236 1935 1992 2143
[30000] 3524 3301 3970 3752 3224 4002 3359 3255 3945 3186 3322 3202
[40000] 4665 4403 6061 4565 4505 5109 4696 4652 4709 4431 4365 4313
[50000] 6438 5694 7557 5563 5868 5770 5989 6032 6088 5504 5768 8509
[60000] 7404 6877 8414 6649 7204 7160 7262 7281 7517 7394 7259 9267
[70000] 9221 7976 10512 8496 8469 9748 8582 10113 8484 8598 8656 9339
[80000] 10654 9737 10198 10161 9749 10553 10168 9886 10184 10341 10169 10340
[90000] 11598 11780 11681 10900 10795 12103 12389 11001 11506 11677 10832 11882
[100000] 12451 14065 14047 14506 12304 12926 12119 13155 12704 12417 12576 13206

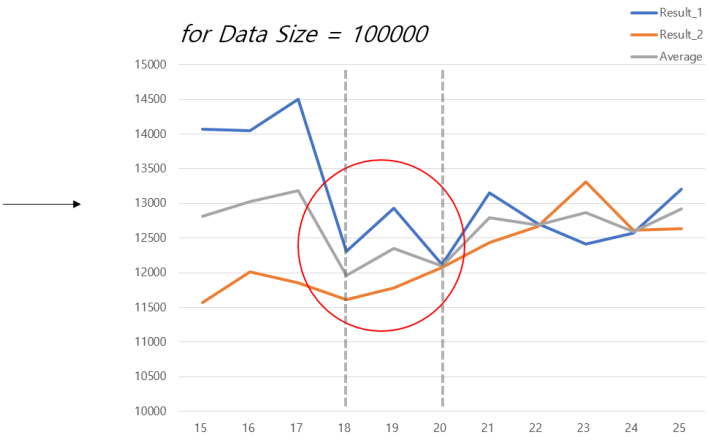
> 무작위 데이터에 대한 측정:
데이터 크기<Pivot Random> <Size 15> <Size 16> <Size 17> <Size 18> <Size 19> <Size 20> <Size 21> <Size 22> <Size 23> <Size 24> <Size 25>
[10000] 994 928 875 867 826 919 943 1033 916 943 985 1030
[20000] 2220 2047 1972 1809 1959 1977 1935 2072 2088 2169 2142 2110
[30000] 3809 3279 2991 3098 3332 3128 2996 3182 3287 4351 3409 3454
[40000] 5089 4162 4115 4629 4034 4209 4373 4158 4545 5644 4383 4772
[50000] 6519 5363 5317 5412 5298 5385 5256 5607 5468 7370 5701 5528
[60000] 6900 6632 6886 6759 6507 6686 6555 6678 6727 7616 6963 6810
[70000] 8816 8171 8338 8058 8202 8353 7678 8109 9090 8274 7684 8335
[80000] 10317 9404 9365 9136 9635 9598 8958 9442 11156 10170 9460 9330
[90000] 11454 10650 11102 10525 10962 10620 11031 10502 11065 11674 12055 10989
[100000] 13694 11569 12011 11858 11614 11783 12074 12431 12662 13307 12611 12637



	Result_1	Result_2	Average
size = 15	14065	11569	12817
size = 16	14047	12011	13029
size = 17	14505	11858	13182
size = 18	12304	11614	11959
size = 19	12926	11783	12355
size = 20	12119	12074	12097
size = 21	13155	12431	12793
size = 22	12704	12662	12683
size = 23	12417	13307	12862
size = 24	12576	12611	12594
size = 25	13206	12637	12922

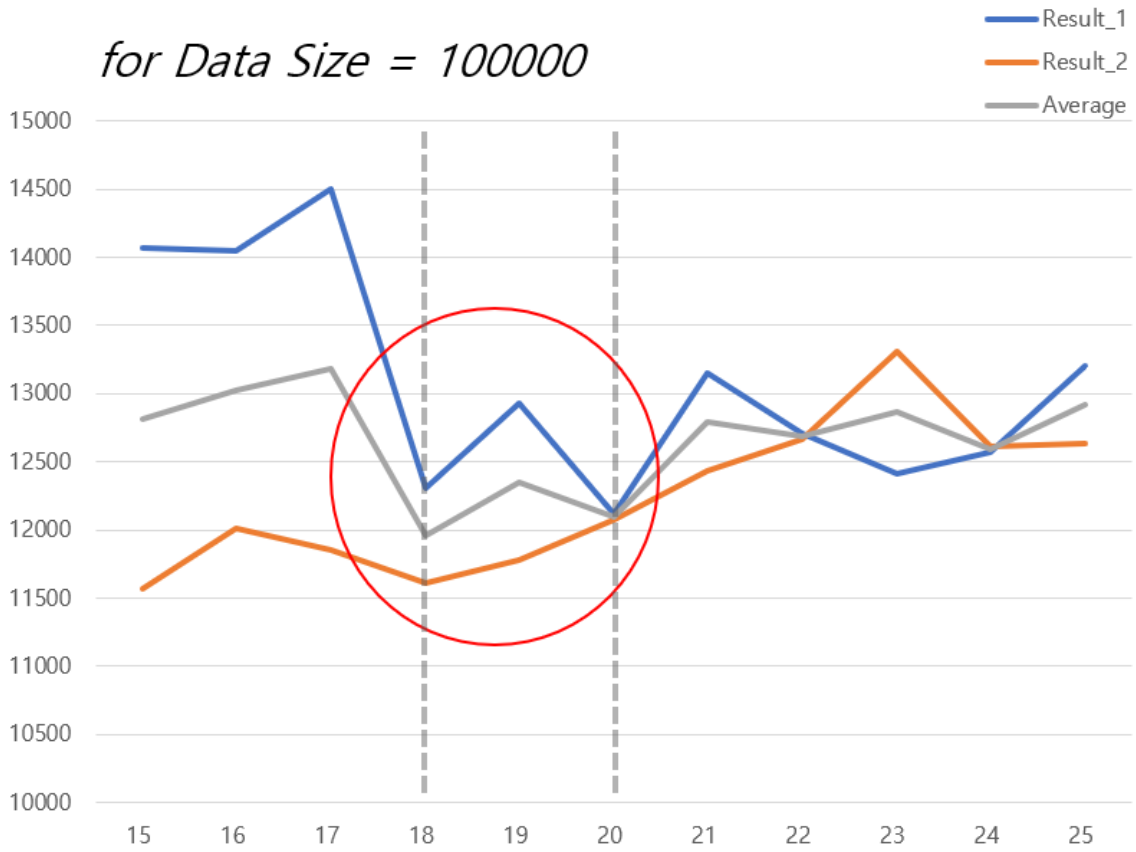
Data Size = 100000 일 때 두번의 측정값을 추출하여 그래프로 나타낸 후 두 결과의 Average 를 분석해 보았습니다. Size 를 10씩 늘리면서 보았을 때 Size = 20 부근에서 전반적으로 가장 낮은값이 나오는 것으로 확인되어 size = 15 ~ 25 사이에서 어떤 값을 취하면 성능이 좋을지에 대해 측정하였습니다.

	Result_1	Result_2	Average
size = 15	14065	11569	12817
size = 16	14047	12011	13029
size = 17	14505	11858	13182
size = 18	12304	11614	11959
size = 19	12926	11783	12355
size = 20	12119	12074	12097
size = 21	13155	12431	12793
size = 22	12704	12662	12683
size = 23	12417	13307	12862
size = 24	12576	12611	12594
size = 25	13206	12637	12922



(다음장에 이어집니다.)

> 생각해 볼 점



위의 그래프는 임의의 두 결과에 대해서 Data Size = 100000 일 때의 size 별 QuickSort WIth InsertionSort 성능을 측정한 값을 그래프로 나타낸 것입니다.
회색 라인이 두 결과에 대한 평균값으로 그래프상으로 보았을 때 18 ~ 20개 미만의 데이터에 대해 InsertionSort 를 진행하도록 하였을 때의 성능이 일반적으로 좋은 것을 볼 수 있습니다.
물론 더 일반화되고 정확한 결과는 더 많은 개수의 측정치를 얻어 Average를 구해야하겠지만, 지금 측정한 2개의 결과값만을 보고도 전반적인 경향을 판단할 수 있다고 생각하였고, 따라서 측정 결과에 따르면
20개 미만의 데이터에 대해서 적용할 때 전체적인 성능이 가장 좋을 것으로 판단됩니다.

3. Java 로 성능 측정하는 한계에 대해 관찰하고 원인을 생각해본다. 그리고 대처 방법이나 다른 더 좋은 방법이 무엇인지 생각해본다.

- Java는 javac(자바 컴파일러)를 통해 .java에서 IR인 bytecode로 이루어진 .class 파일로 변환 후 JVM이 해당 bytecode를 interpret 하면서 동작하는데, 중간에 JVM이라는 가상머신이 bytecode Interpret 하므로 컴파일 이후에 바로 실행되는 C언어보다 실행속도면에서 비효율적인 부분을 보여줍니다. 또한 메모리 할당 및 해제하는 부분을 Garbage Collector가 수행하는데

> 생각해 볼 점

이 과정이 수행되는 시점에 Thread가 멈추게되므로 측정치에 Thread가 멈춘 시간만큼의 Delay가 반영될 수 있습니다. 이것이 Java로 측정했을 때의 한계점이라고 생각합니다.

3-1. Java 환경에서 측정값이 일관성을 가지지 않는 이유는?

자바 환경에서 성능을 측정했을 때 일관성을 가지지 않는 이유에 대하여 생각을 해보았는데요, Java 라는 언어의 특성상 완전한 Native Bytecode가 아닌 JVM 바이트코드로 컴파일 된 후에 JVM 위에서 돌아가는 구조이다 보니 C언어와 같이 바로 Binary 코드로 컴파일 되어 실행되는 구조보다 동일한 프로그램을 돌리는데 소요되는 시간이 오래 걸리기도 하고, 자바에서는 Memory를 JVM이 알아서 관리해주는 기능이 있는데, Heap-Memory에 역할이 끝나고 남아있는 대상을 해제 하는 과정인 Garbage Collecting 과정을 수행할 때 Garbage Collector가 해당 Thread를 멈추게 되고(Stop the World가 수행됨), Garbage Collection이 수행되는 과정을 알 수 없기 때문에 예측 불가한 시점에 Thread가 일시정지가 될 수 있습니다. 따라서 측정 도중 가비지 컬렉팅이 임의로 일어나게 되면 짧은 시간이라도 Thread가 Stop 되기 때문에 평균적인 성능 보다 떨어진 성능으로 측정될 수 있다고 생각합니다.

3-2. C와 같은 다른 언어 환경에서는 어떨까?

- C언어 같은 경우 Compile 과정이 오래걸릴 수는 있지만 컴파일 이후에는 자바보다 실행속도가 빠른 편이므로, 실행속도 측면에서는 더 좋다고 볼 수 있습니다. 또한 자바와 다르게 메모리 할당 해제를 프로그래머가 직접 하기 때문에 자바에서 GC로 인한 Thread가 멈추는 등과 같은 측정이 실제 메서드 자체의 수행속도보다 Delay 된 결과를 도출할 수 있는 Risk가 존재하지 않기 때문에 더 일관적이고 정확한 측정결과 비교를 수행할 수 있을 것이라고 생각합니다.

감사합니다.

충남대학교 컴퓨터공학과
알고리즘 04분반
학번: 201701975
이름: 구건모