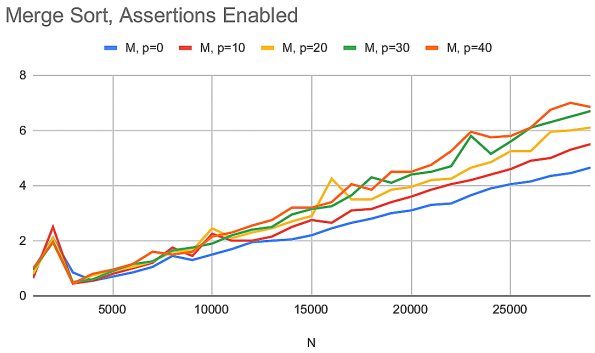
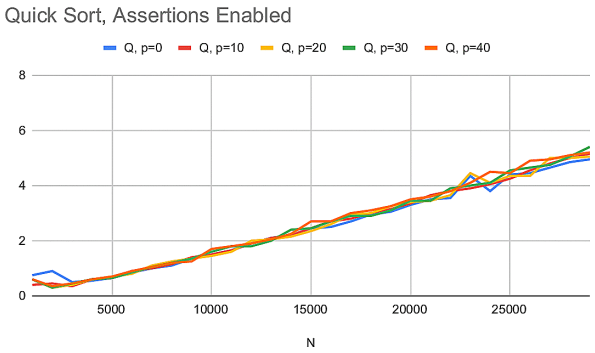
1. There is a video in our Zoom repository that shows you the process of doing an empirical comparison between Insertion Sort and Selection sort. Following the same process, do an empirical comparison between MERGE-SORT and QUICK-SORT. Use "number of numbers being sorted" and "the percentage of disorder" as your independent variables; use "completion time" as your only dependent variable. Plot your findings and decide which of these two algorithms runs faster; more importantly, quantify the speed difference between MERGE-SORT and QUICK-SORT:

   - Video: [CS 2110: Collecting Empirical Data (Selection vs Insertion sort)]
   - Spreadsheet shown in the video: [Empirical Data on Selection/Insertion Sort]
   - Java code used in the video: [Java package ps_2_ali]

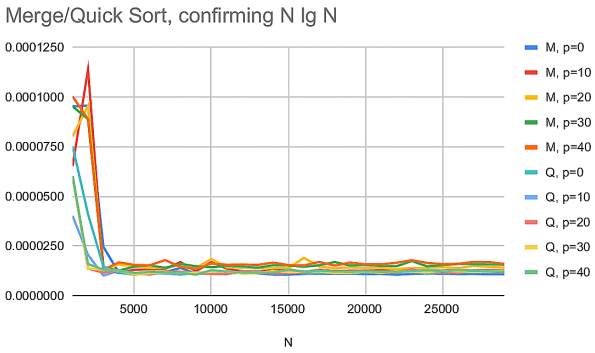
   i

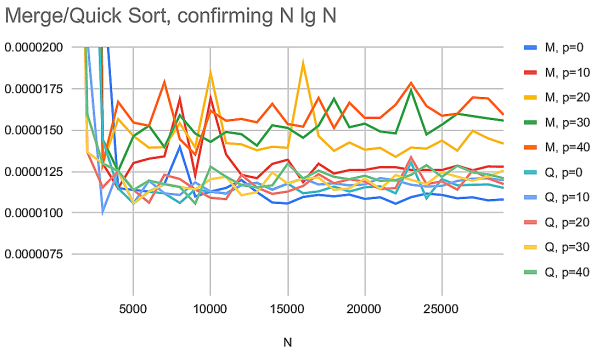
   Answer: Quick sort appears to be slightly faster; it is also immune to the p parameter because of the shuffling in its main driver method:



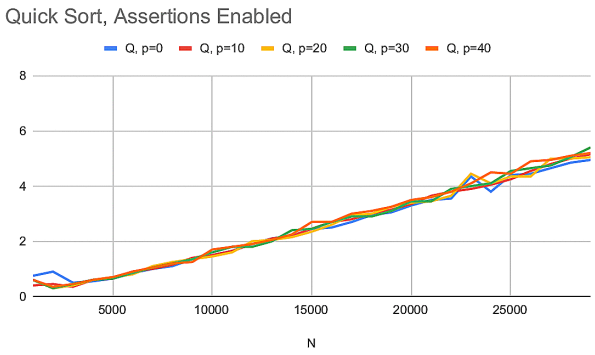


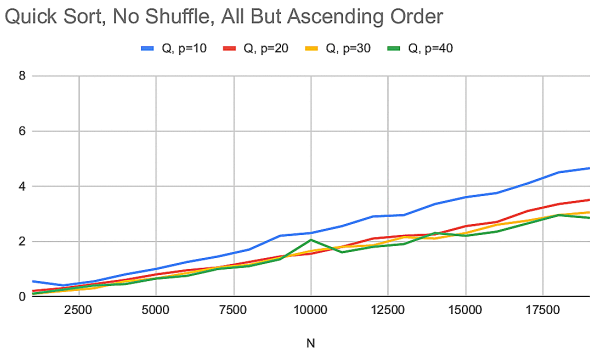
   Merge-Sort and Quick-Sort both have $N \lg N$ time regardless of the p value; the left plot divides all chart by the correspondinf $N \lg N$ value; the right chart simply zooms up the area of interest:



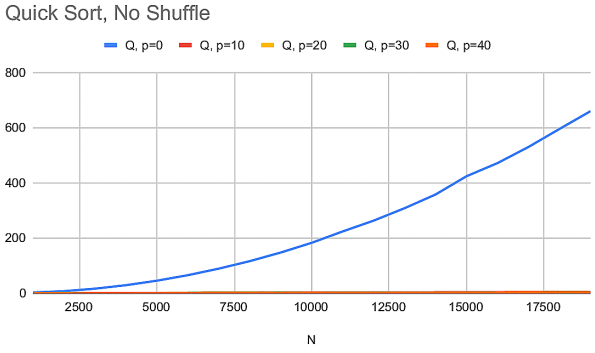


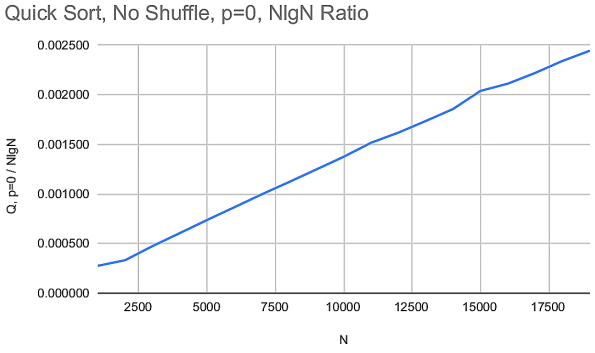
   Quick-Sort does well for non-trivial p values when the shuffling in its main driver method is disabled:





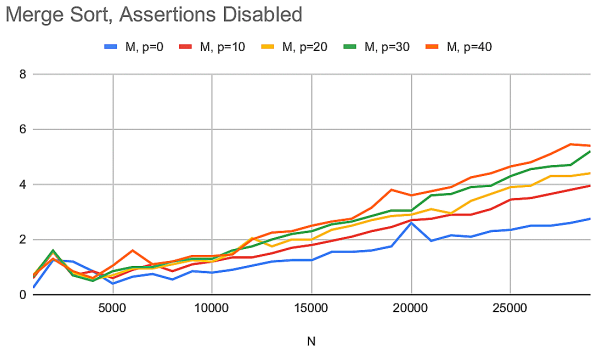
Quick-Sort does operates way worse for trivial/small p values when the shuffling in its main driver method is disabled; in fact, as can be confirmed from the chart on the right, it transitions from $N \lg N$ to N^2:

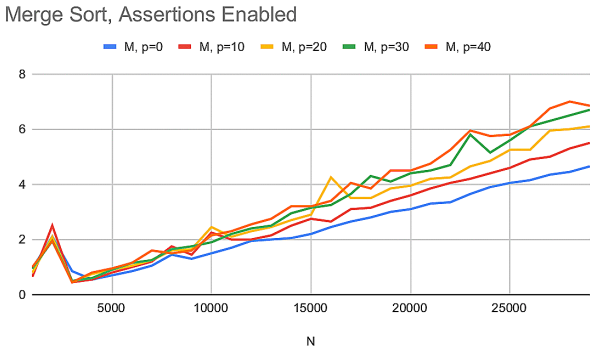




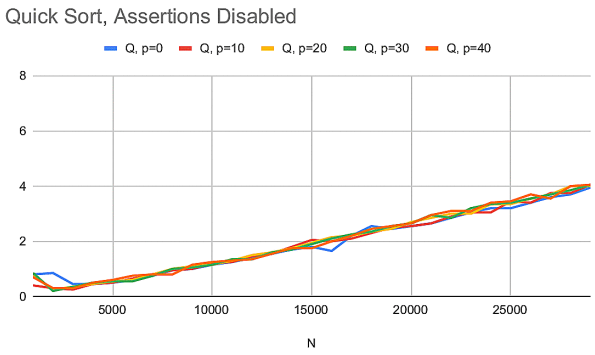
2. Using either MERGE-SORT or QUICK-SORT, do a study similar to tht one in step 1 to understand the degree to which assertions make a difference in runtime.

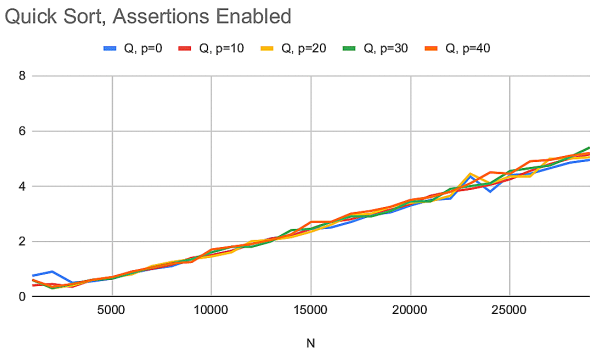
   Answer: Enabling/disabling assertions does not make a significant (**asymptotic**) difference with Merge-Sort:





Enabling/disabling assertions does not make a significant (**asymptotic**) difference with Quick-Sort:





3. Using the following array of integers, illustrate the operation of MERGE-SORT; that is, identify the particular recursive calls made, to lo/hi values, etc. Your definitive guide for MERGE-SORT should be [the implementation from our book].

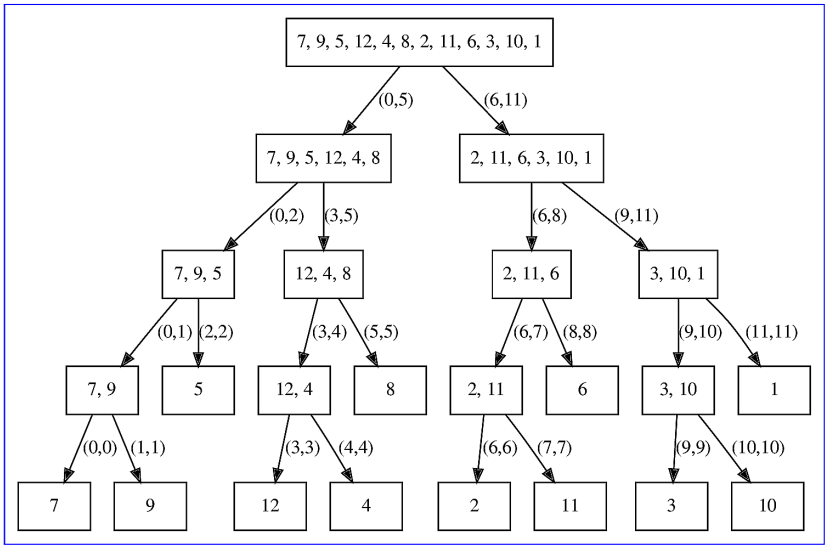
7	9	5	12	4	8	2	11	6	3	10	1

Answer: The following shows how we can add statements to the '`Merge.sort()`' method so that it displays itself in GraphViz syntax:

```
1  public static String makeLabel(Comparable[] a, int lo, int hi) {
2    StringBuffer sb = new StringBuffer();
3    for (int i = lo; i <= hi; i++) {
4      sb.append(a[i]);
5      if (i < hi)
6        sb.append(", ");
7    }
8    return sb.toString();
9  }
10
11 public static String makeName(int lo, int hi) {
12   if ( hi < 0 )
13     return "lo" + lo + "hi" + "NO"; // Neg One
14   return "lo" + lo + "hi" + hi;
15 }
16
17 private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
18   if (hi <= lo)
19     return;
20   int mid = lo + (hi - lo) / 2;
21   System.out.println("\t" + makeName(lo, hi) + //
22                      " -> " + makeName(lo, mid) + //
23                      " [label=\"(" + lo + "," + mid + ")\"]");
24   System.out.println("\t" + makeName(lo, hi) + //
25                      " -> " + makeName(mid + 1, hi) + //
26                      " [label=\"(" + (mid + 1) + "," + hi + ")\"]");
27   System.out.println("\t" + makeName(lo, mid) + //
28                      " [label=\"" + makeLabel(a, lo, mid) + "\"]");
29   sort(a, aux, lo, mid);
30   System.out.println("\t" + makeName(mid + 1, hi) + //
31                      " [label=\"" + makeLabel(a, mid + 1, hi) + "\"]");
32   sort(a, aux, mid + 1, hi);
33   merge(a, aux, lo, mid, hi);
```

```
34  }
```

Based on this modification, we get the following visualization of the sorting process:



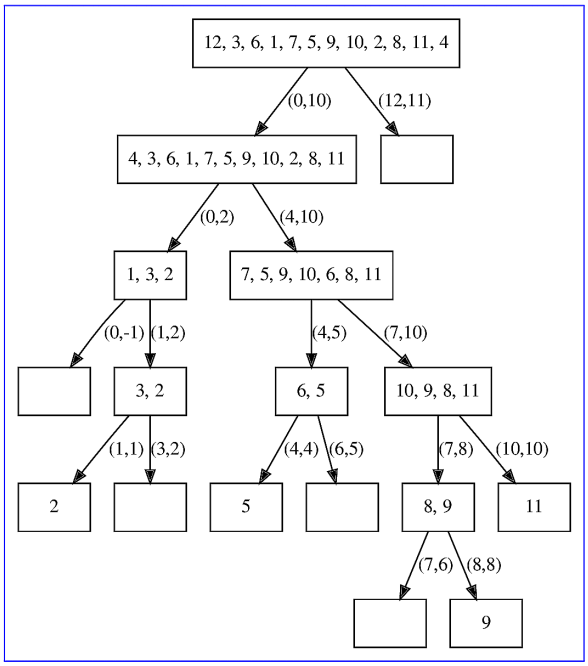
4. Using the following array of integers, illustrate the operation of QUICK-SORT; that is, identify the particular recursive calls made, pivot value selections, etc. Your definitive guide for QUICK-SORT should be [the implementation from our book].

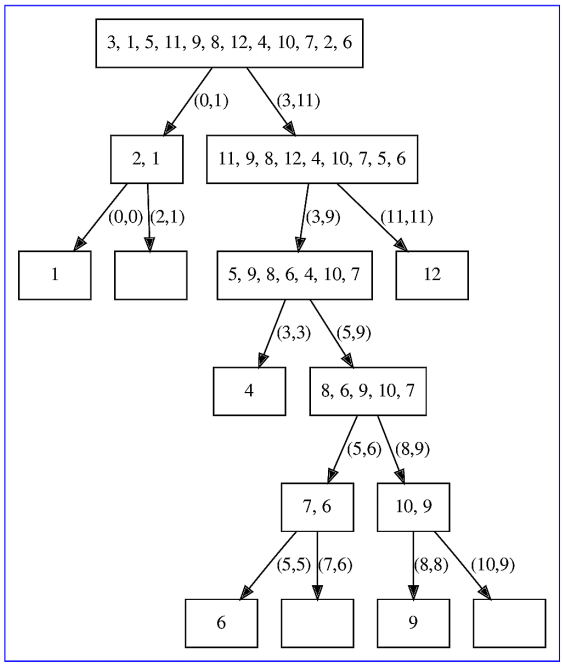
7	9	5	12	4	8	2	11	6	3	10	1

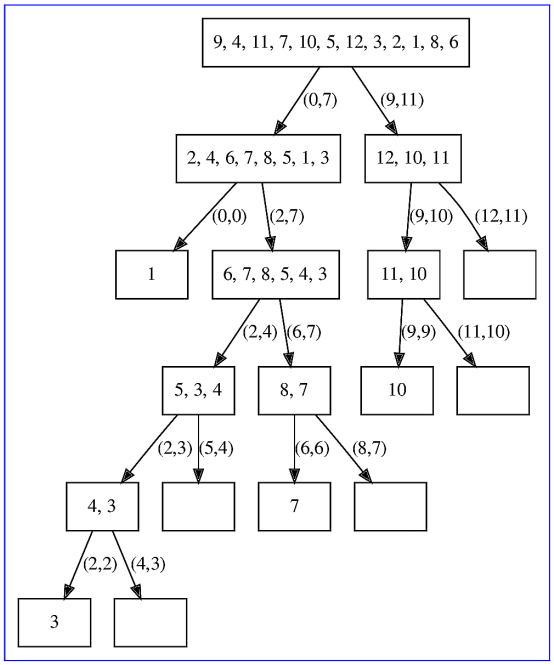
Answer: The following shows how we can add statements to the '`Quick.sort()`' method so that it displays itself in GraphViz syntax:

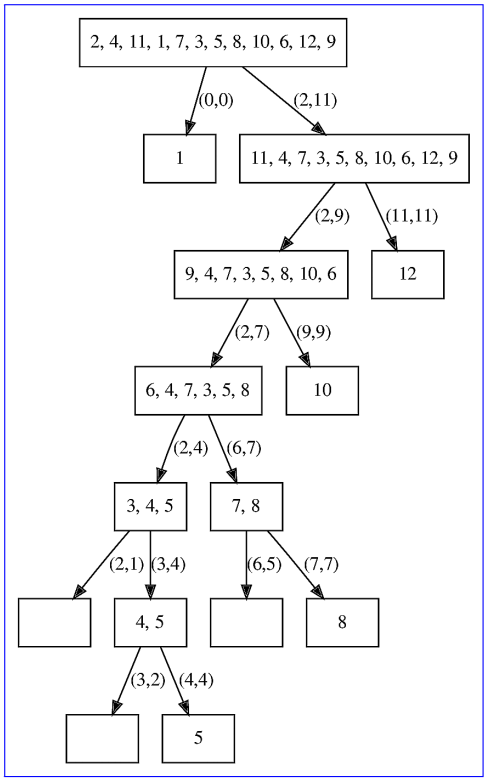
```
1   private static void sort(Comparable[] a, int lo, int hi) {
2     if (hi <= lo)
3       return;
4     int j = partition(a, lo, hi);
5     System.out.println("\t" + makeName(lo, hi) + //
6             " -> " + makeName(lo, j - 1) + //
7             " [label=\"(" + lo + "," + (j - 1) + ")\"]");
8     System.out.println("\t" + makeName(lo, hi) + //
9             " -> " + makeName(j + 1, hi) + //
10            " [label=\"(" + (j + 1) + "," + hi + ")\"]");
11    System.out.println("\t" + makeName(lo, j - 1) + //
12            " [label=\"" + makeLabel(a, lo, j - 1) + "\"]");
13    System.out.println("\t" + makeName(j + 1, hi) + //
14            " [label=\"" + makeLabel(a, j + 1, hi) + "\"]");
15
16    sort(a, lo, j - 1);
17    sort(a, j + 1, hi);
18    assert isSorted(a, lo, hi);
19  }
```

Based on this modification, we get the following visualizations of the sorting process, each based a different way the array is initially shuffled. Please note that (i) there could be many more depending on which value is picked as a pivot, and (ii) a bad pivot can make these "trees" get as deep as N (i.e. the number of nodes in the original array):









5. Using the following array of integers, illustrate the operation of QUICK-SELECT for finding the 10th ranking numbers; that is, identify the particular recursive calls made, pivot value selections, etc. Your definitive guide for QUICK-SELECT should be [the implementation from our book].

7	9	5	12	4	8	2	11	6	3	10	1

6. Suppose you have N distinct points in a three dimensional space[1]. That is, each point p_i has three coordinates

[1] That is, it is safe to assume that no two points are the same.

as in $p_i = (x_i, y_i, z_i)$. Your task is to order these points according to the following logic: for any two chosen points $p_i = (x_i, y_i, z_i)$ and $p_j = (x_j, y_j, z_j)$

- If $z_i > z_j$, then $p_i > p_j$.
- If $z_i = z_j$ but $y_i > y_j$, then $p_i > p_j$.
- If $z_i = z_j$ and $y_i = y_j$ but $x_i > x_j$, then $p_i > p_j$.

Provide an efficient algorithm to solve this problem.

Answer:

(a) The question states that x is the least significant dimension so we first sort the points by their x coordinates.

(b) The question states that y is the next significant dimension so we then sort the points by their y coordinates. **If the sorting algorithm is stable**, then any pair of points with equal y values will **not** be swapped which means they will remain sorted by their x coordinate.

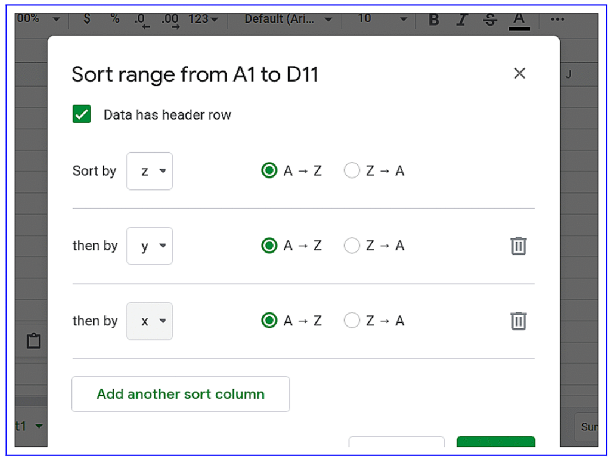
(c) The question states that z is the most significant dimension so we finally sort the points by z coordinates. **If the sorting algorithm is stable**, then any pair of points that have equal z values will **not** be swapped which means they will remain sorted by their x and y coordinates.

Two important points:

- There is a good chance the sorting sequence (from least significant dimension to most significant dimension) will appear backwards; but remember, the last sort has the final say ("has the last laugh").
- The algorithm can be generalized to any number of dimensions. The question had three dimensions so we called a stable sorting algorithm three times. If we had to work with N dimensions, we would simply call a stable sorting algorithm N times.

We can see this happening when we sort a spreadsheet; each "sort column" added simply means an additional call to the sort method. And needless to say, using the '`Comparator`' interface would make this simpler since we can redefine the comparison logic each time we call the sort method.

	A	B	C	D
1	x	y	z	PointLabel
2	0	1	1	g
3	2	0	0	i
4	2	1	1	a
5	1	1	2	b
6	2	2	0	d
7	1	2	0	h
8	0	1	0	j
9	0	2	1	e
10	1	2	1	c
11	2	2	0	f



	A	B	C	D
1	x	y	z	PointLabel
2	2	0	0	i
3	0	1	0	j
4	1	2	0	h
5	2	2	0	d
6	2	2	0	f
7	0	1	1	g
8	2	1	1	a
9	0	2	1	e
10	1	2	1	c
11	1	1	2	b

Needless to say: **Merge-Sort is the stable sort algorithm.**