CONCORDIA UNIVERSITY

COEN-448

Software Testing and Validation

Assignment– 2

Name: RIFADUL HAQUE

ID:

Section: W

Due Date:13/3/22

1.1 please complete the table of defuse pairs.

node i	def(i)	c-use(i)	edge(i,j)	p-use(i,j)
node 1	def (1) = {X, Y, W, Z}	c-use (1) = { Y }	edge (1,2)	p-use (1,2) ={ }
node 2	def (2) ={ }	c-use (2) = { }	edge (2,3) edge (2,4)	p-use (2,3) = {W} p-use (2,4) = {W}
node 3	def (3) = {W, Z}	c-use (3) = {X, W, Z}	edge (3,2)	p-use(3,2) ={ }
node 4	def (4) = { }	c-use (4) = { }	edge (4,5) edge (4,6)	p-use (4,5) = {Y} p-use (4,6) = {Y}
node 5	def (5) = {Z}	c-use (5) = {Z}	edge (5,6)	p-use (5,6) ={ }
node 6	def (6) = { }	c-use (6) = {Z}	edge (6,6) = null	p-use (6,6) ={ }

node i	dcu(v,i)	dpu(v,i)
node 1	dcu(X,1) = {3} dcu(W,1) = {3} dcu(Z,1) = {3,5,6}	dpu(Y,1) = {4,5} dpu(Y,1) = {4,6} dpu(W,1) = {2,4} dpu(W,1) = {2,3}
node 3	dcu(Z,3) = {3,5,6} dcu(W,3) = {3}	dpu(W,3) = {2,3} dpu(W,3) = {2,4}
node 5	dcu(Z,5) = {6}	

1.2 Given the dcu and dpu, write the test cases to cover them all.

Test dcu(X,1), Test dcu(W,1)

The test cases are same for them:

Input data < “X”, “Y” | 1, 2 > Input data < “X”, “Y” | 2, 3 > Input data < “X”, “Y” | 1, 0 >

Expected output at (3): Z=1 Expected output (3): Z=2 Expected output (3): Z=1

Test dcu(Z, 1)

Input data < “X”, “Y” | 1, 2 >
Expected output at (3): Z=1
Expected output at (6): Z=1

Input data < “X”, “Y” | 2, 1 >
Expected output (3): Z=2
Expected output (6): Z=2

Input data < “X”, “Y” | 1, 0 >
Expected output (3): Z=1
Expected output (6): Z=1

Test dcu(Z,3)

Input data < “X”, “Y” | 1, 2 >
Expected output at (3): Z=1
Expected output at (5): Z=null
Expected output at (6): Z=1

Input data < “X”, “Y” | 2, 0 >
Expected output (3): Z=2
Expected output (5): Z=null
Expected output (6): Z=2

Input data < “X”, “Y” | 1, -2 >
Expected output (3): Z=1
Expected output (5): Z=1
Expected output (6): Z=1

Test dcu(W,3)

Input data < “X”, “Y” | 1, 2 >
Expected output at (3): Z=1

Input data < “X”, “Y” | 2, 0 >
Expected output (3): Z=2

Input data < “X”, “Y” | 1, -2 >
Expected output (3): Z=1

Test dcu(Z,5)

Input data < “X”, “Y” | 1, 2 >
Expected output at (5): Z=null

Input data < “X”, “Y” | 2, 0 >
Expected output (5): Z=null

Input data < “X”, “Y” | 1, -2 >
Expected output (5): Z=1

Test dpu(Y,1)

Input data < “X”, “Y” | 1, 2 >
Expected output at (5): Z=null
Expected output at (6): Z=1

Input data < “X”, “Y” | 2, 0 >
Expected output (5): Z=null
Expected output (6): Z=2

Input data < “X”, “Y” | 1, -2 >
Expected output (5): Z=1
Expected output (6): Z=1

Test dpu(W,1)

Input data < “X”, “Y” | 1, 2 >
Expected output at (3): Z=1
Expected output at (5): Z=null
Expected output at (6): Z=1

Input data < “X”, “Y” | 2, 0 >
Expected output (3): Z=2
Expected output (5): Z=null
Expected output (6): Z=2

Input data < “X”, “Y” | 1, -2 >
Expected output (3): Z=1
Expected output (5): Z=1
Expected output (6): Z=1

Test dpu(W,3)

Input data < “X”, “Y” | 1, 2 >
Expected output at (3): Z=1

Expected output at (4): Z=1

Expected output at (5): Z=null

Expected output at (6): Z=1

Input data < “X”, “Y” | 2, 0 >
Expected output (3): Z=2

Expected output (4): Z=2

Expected output (5): Z=null

Expected output (6): Z=2

Input data < “X”, “Y” | 1, -2 >
Expected output (3): Z=1

Expected output (4): Z=1

Expected output (5): Z=1

Expected output (6): Z=1

2.1(10 Marks) Develop test cases following the black-box approach that has input domain modeling of the partition function according to the (2.1.a) best case, (2.1.b) worse case and (2.1.c) average case of the quick sort algorithm. You can choose base choice coverage or other coverage criterion to develop the test cases

2.1.a best case for BCC (Pivot is the mean of the array)

Characteristics Functionality for BCC	B1	B2
Number of elements unsorted in an array	0	1
Size of the Array that is Sorted	5	7

Here in the above table there are 4 quadrant, to make it easier for us to determine the quadrant lets name it as B1A, B1B and B2A and B2B.

My base choice here would be BestB1A and BestB1B.

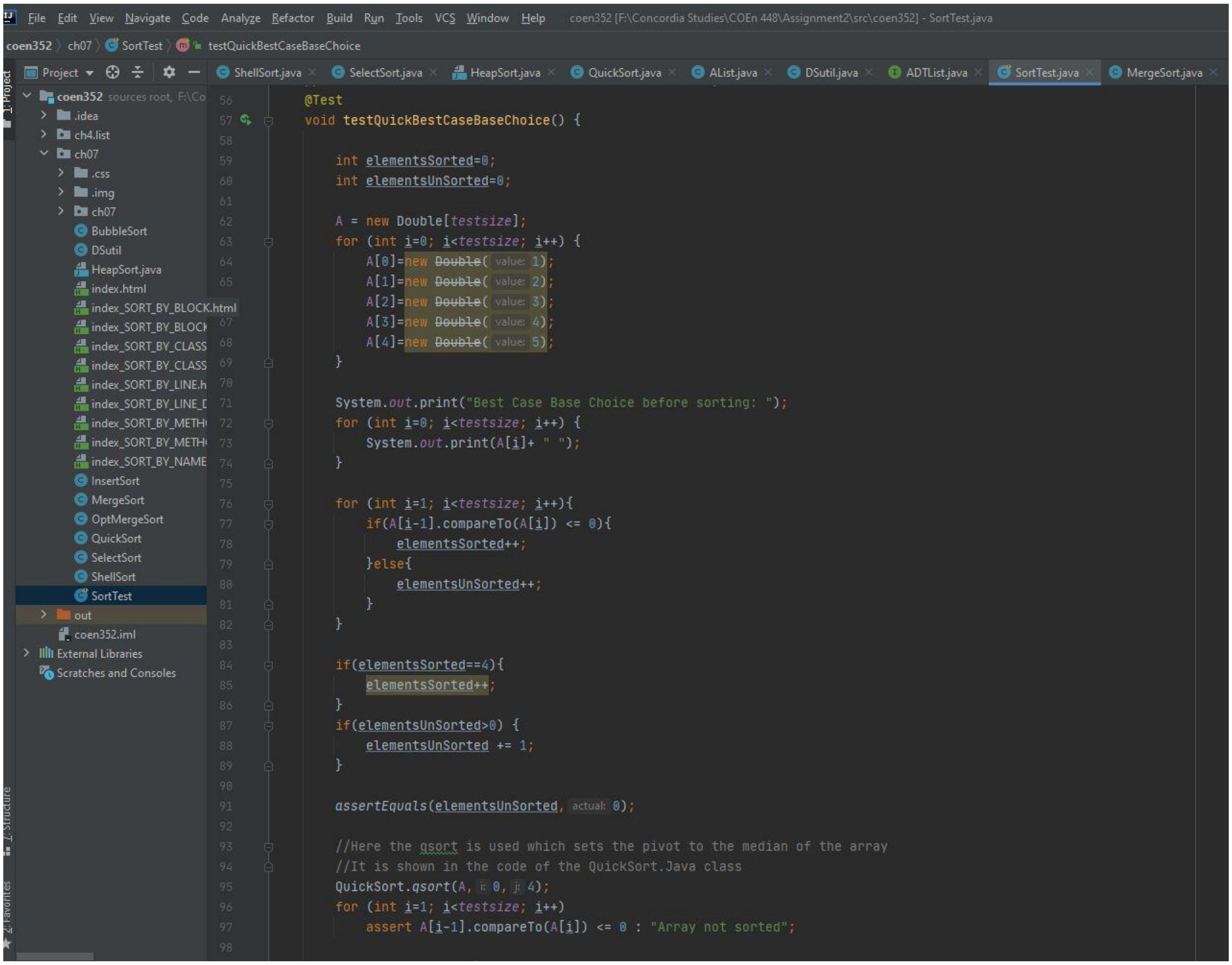
So, the combinations would be:

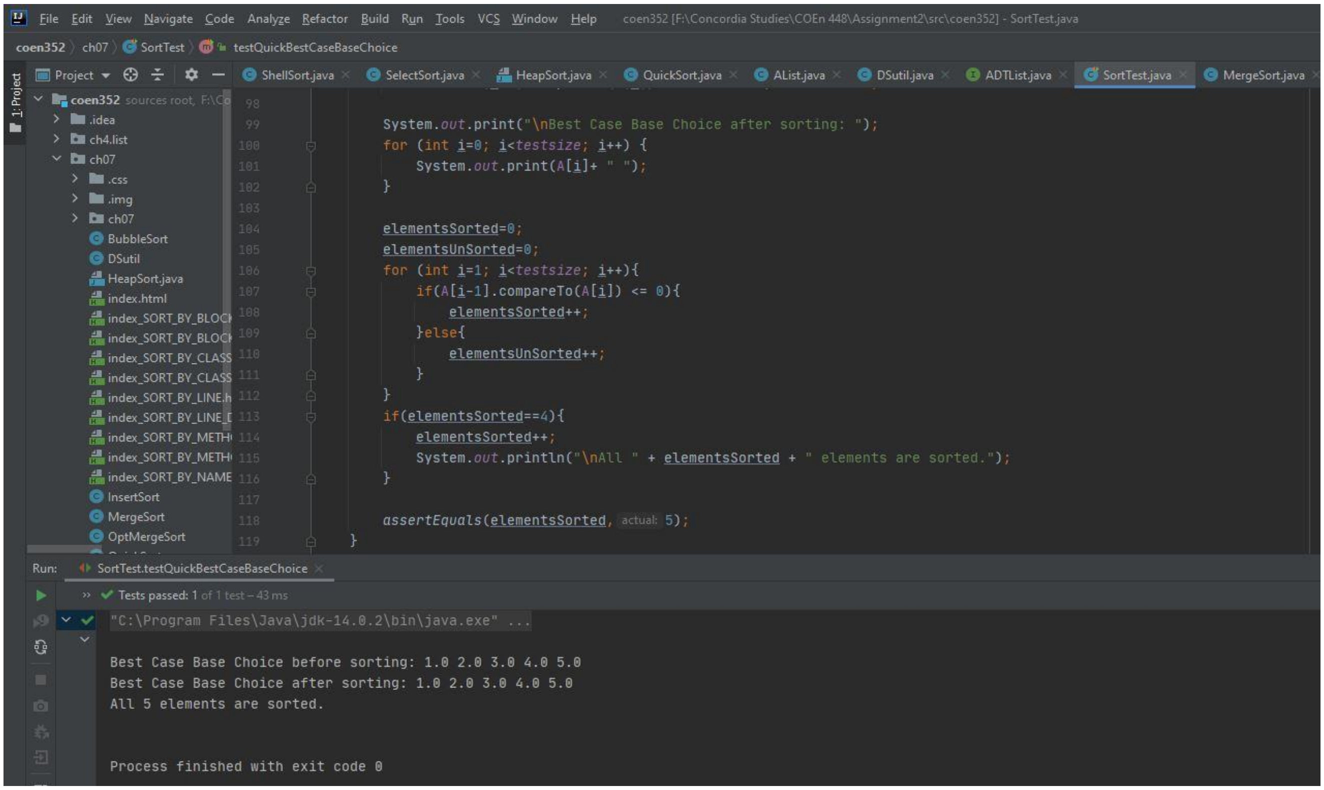
1. (BestB1A, BestB2B)
2. (BestB2A, BestB1B)

In terms of best case I have edited the test case in such a way that the pivot is the mean of the number of elements that are available. I have named it findpivot and used it in qsort method of the QuickSort.java class

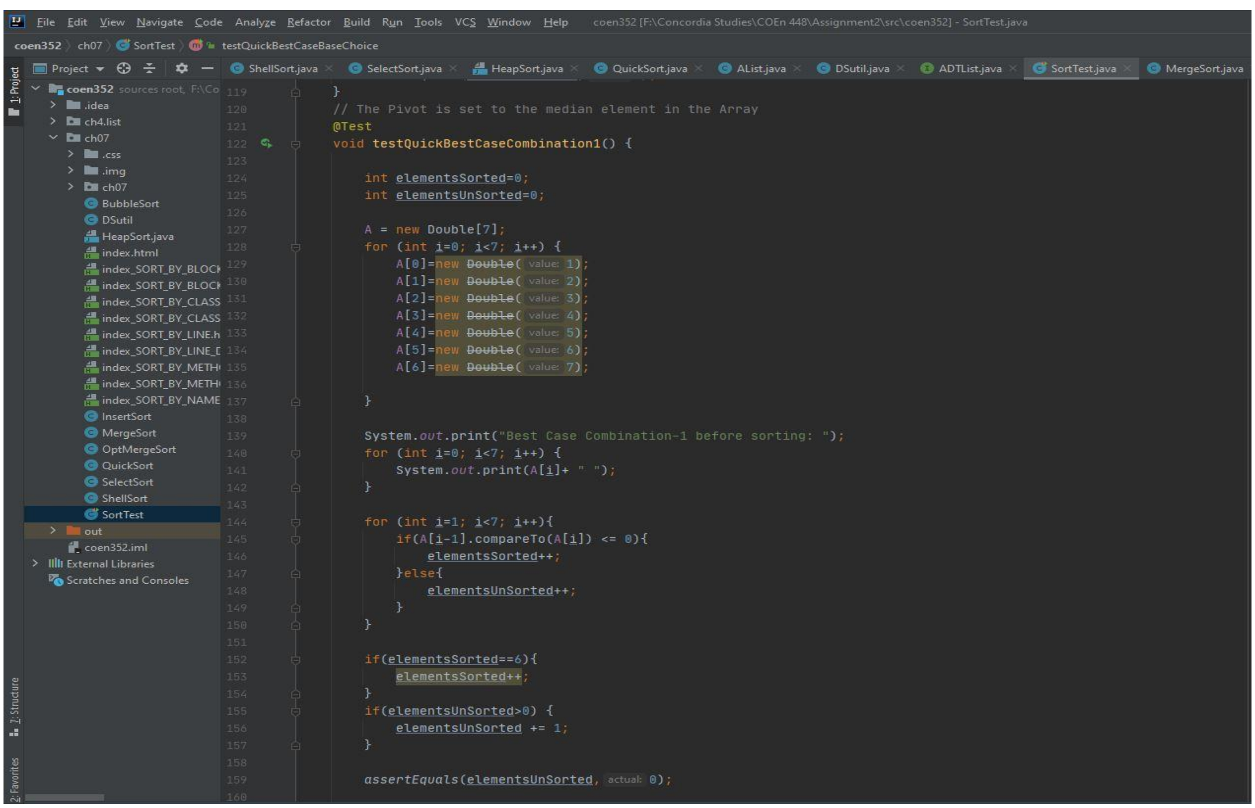
Below are the screenshots of the test for the best case:

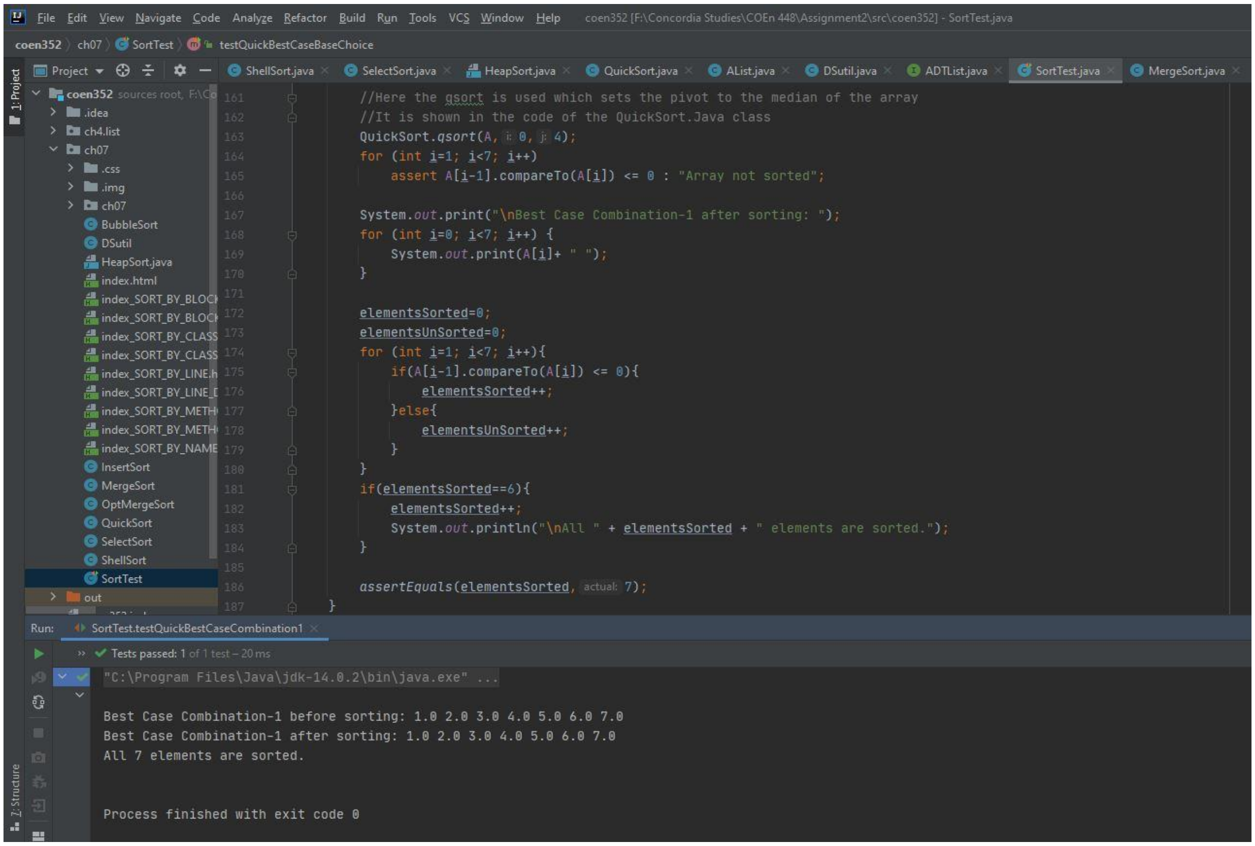
Base Case Best Case:



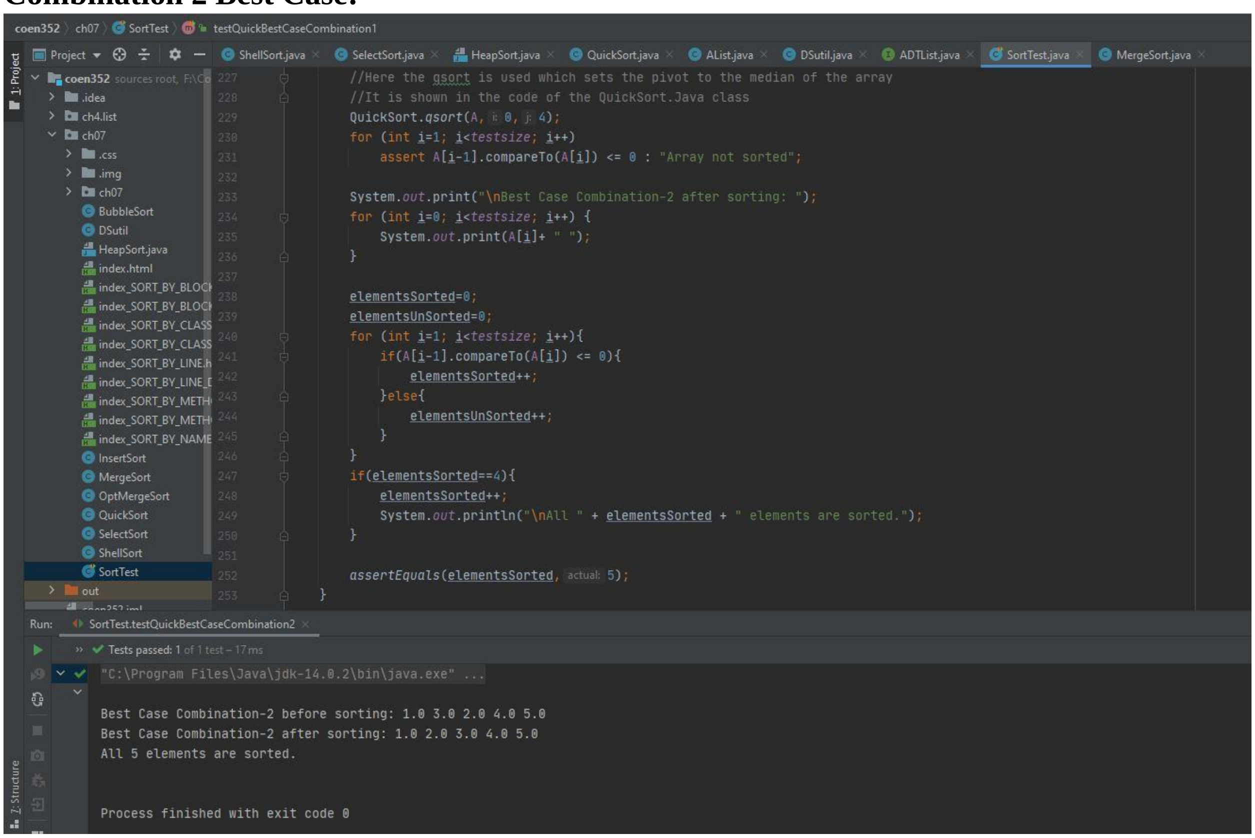


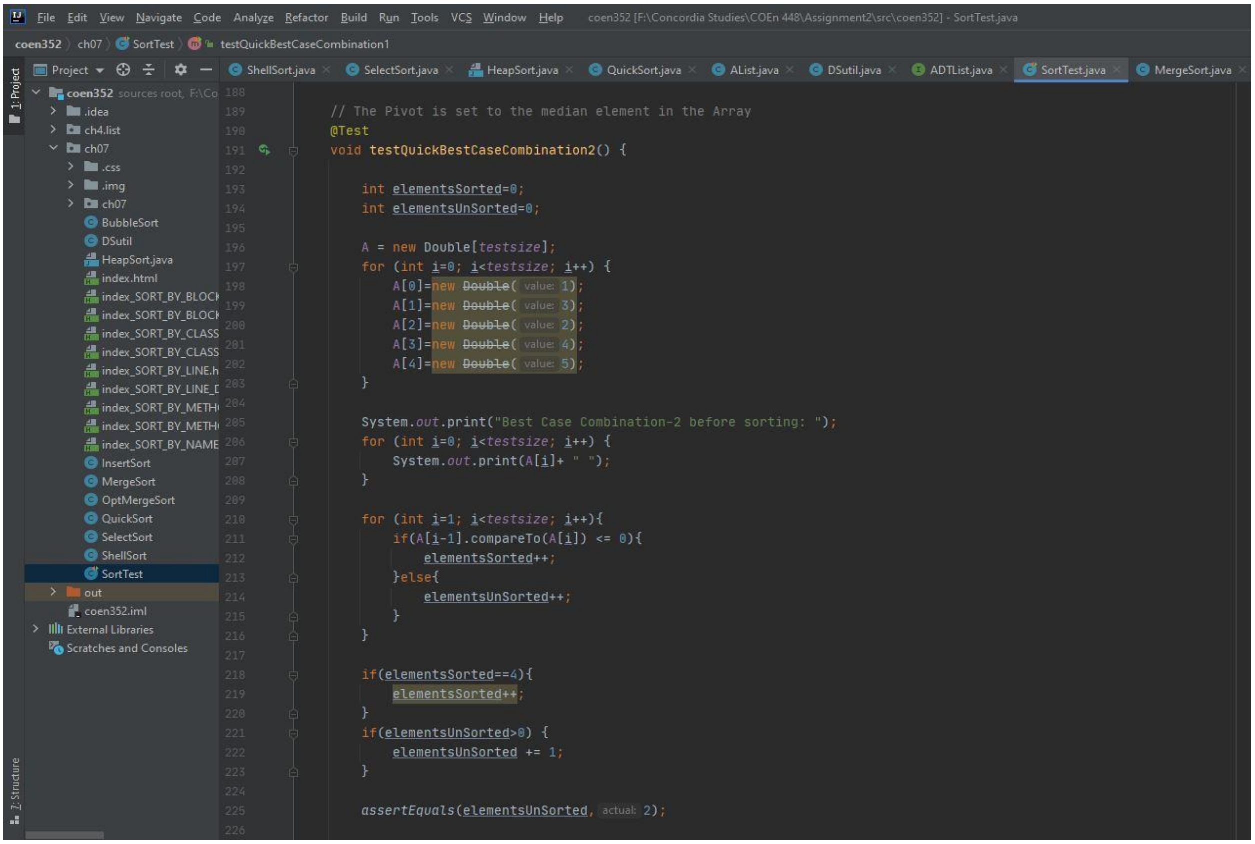
Combination 1 Best Case:





Combination 2 Best Case:





2.1.b worse case for BCC (Pivot is the largest index of the array)

Characteristics Functionality for BCC	B1	B2
Number of elements unsorted in an array	0	1
Size of the Array that is Sorted	5	7

Here in the above table there are 4 quadrant, to make it easier for us to determine the quadrant lets name it as B1A, B1B and B2A and B2B.

My base choice here would be WorseB1A and WorseB1B.

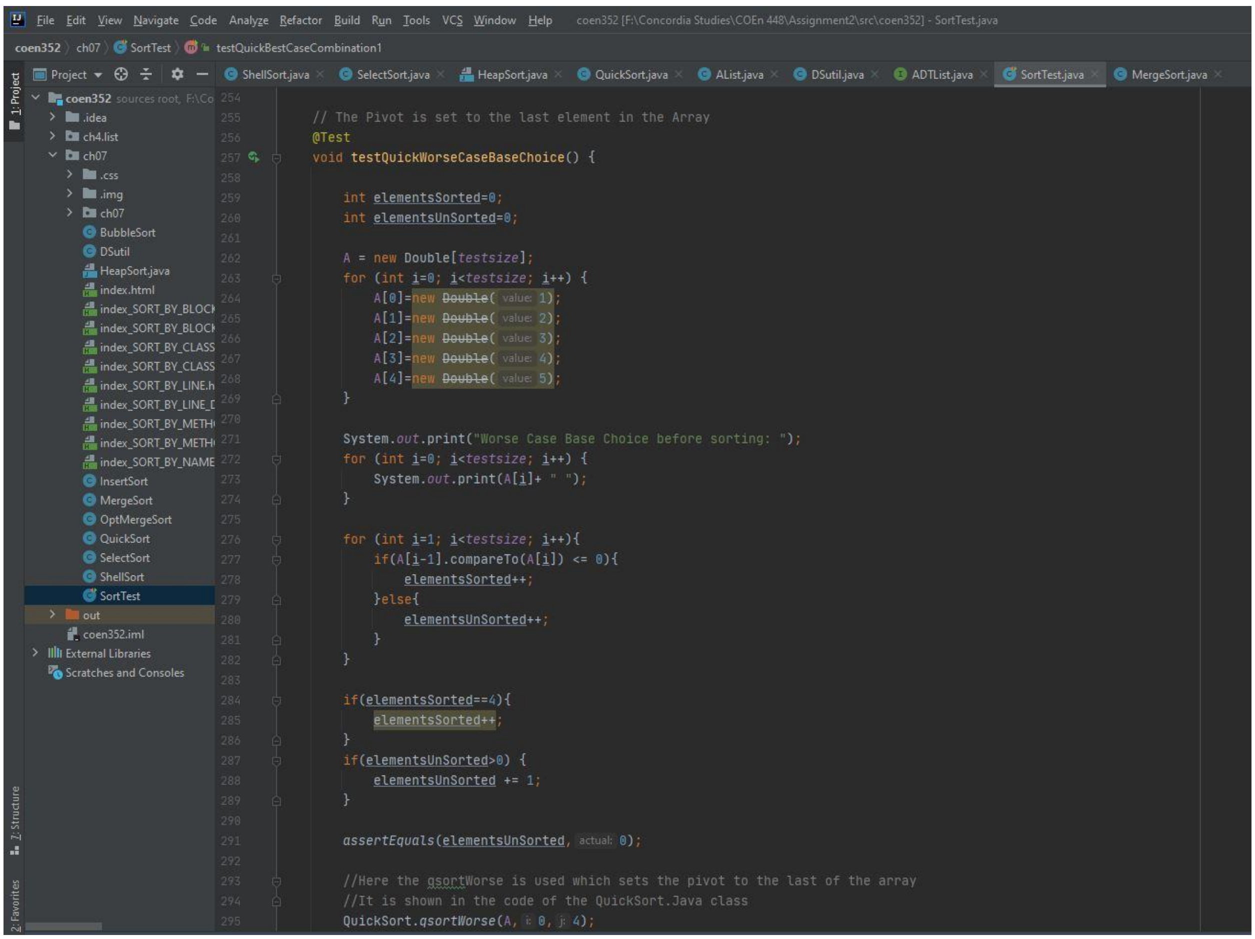
So, the combinations would be:

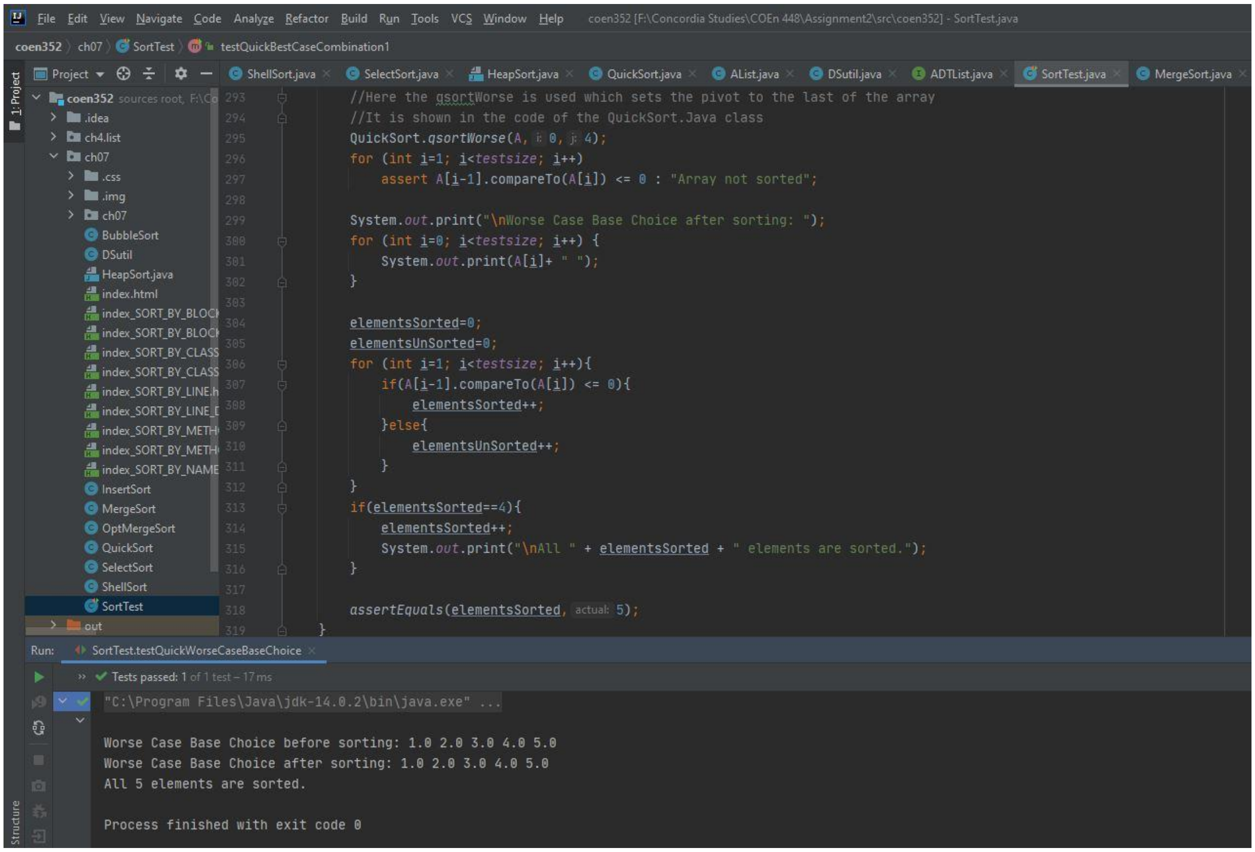
1. (WorseB1A, WorseB2B)
2. (WorseB2A, WorseB1B)

In terms of Worse case I have edited the test case in such a way that the pivot is the last element. I have named it findpivotWorse and used it in qsortWorse method of the QuickSort.java class

Below I have only included the Base Choice screenshot where as the other tests are same and I have performed them in my SortTest.java class. So please refer to the code for checking the code for the combinations.

Base choice Worse Case:





The combination 1 and 2 are in the SortTest class so please refer to the java code for the combination 1 and 2 code.

2.1.c average case for BCC (Pivot is the ¼ index of the number of elements in the array)

Characteristics Functionality for BCC	B1	B2
Number of elements unsorted in an array	0	1
Size of the Array that is Sorted	5	7

Here in the above table there are 4 quadrant, to make it easier for us to determine the quadrant lets name it as B1A, B1B and B2A and B2B.

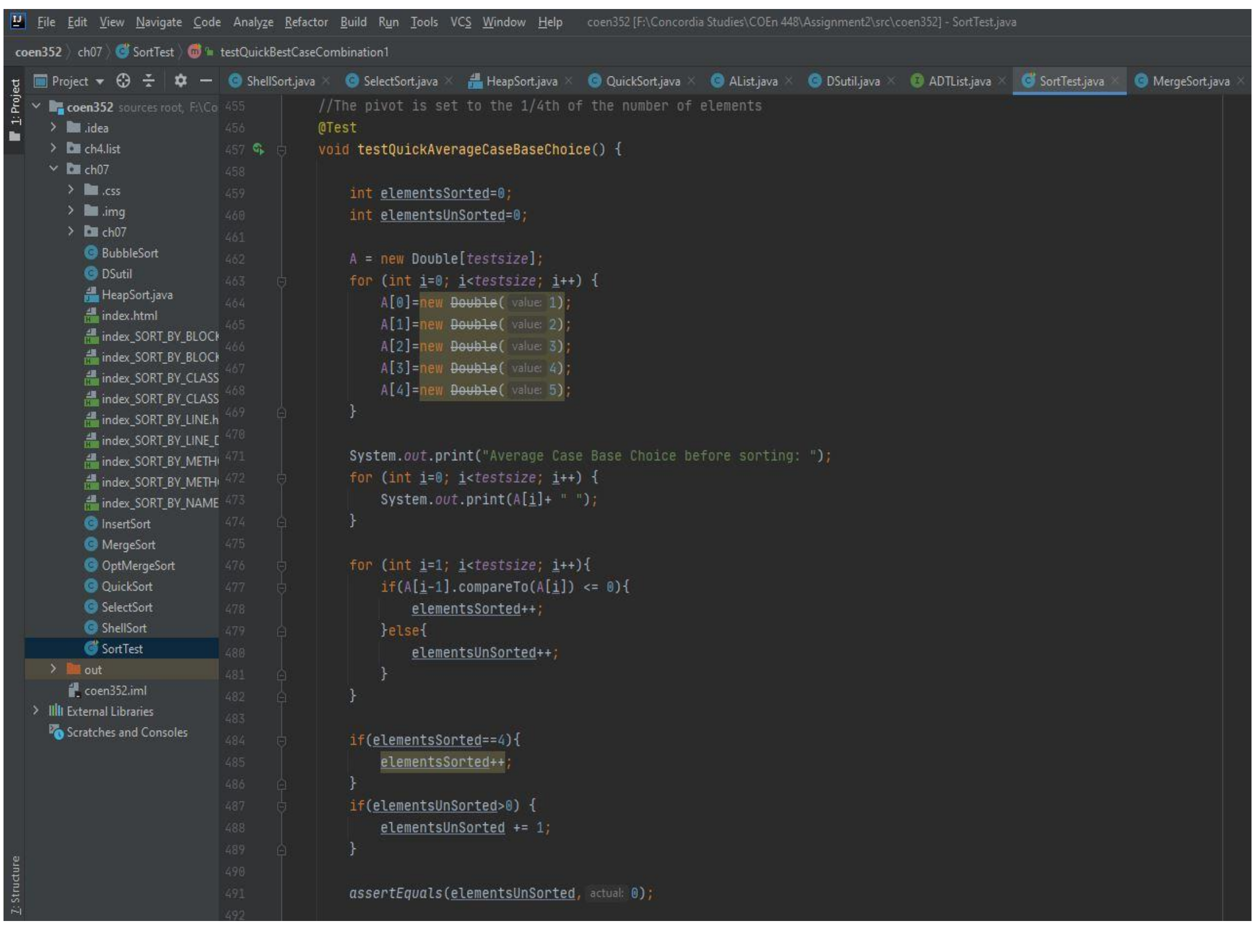
My base choice here would be AverageB1A and AverageB1B.

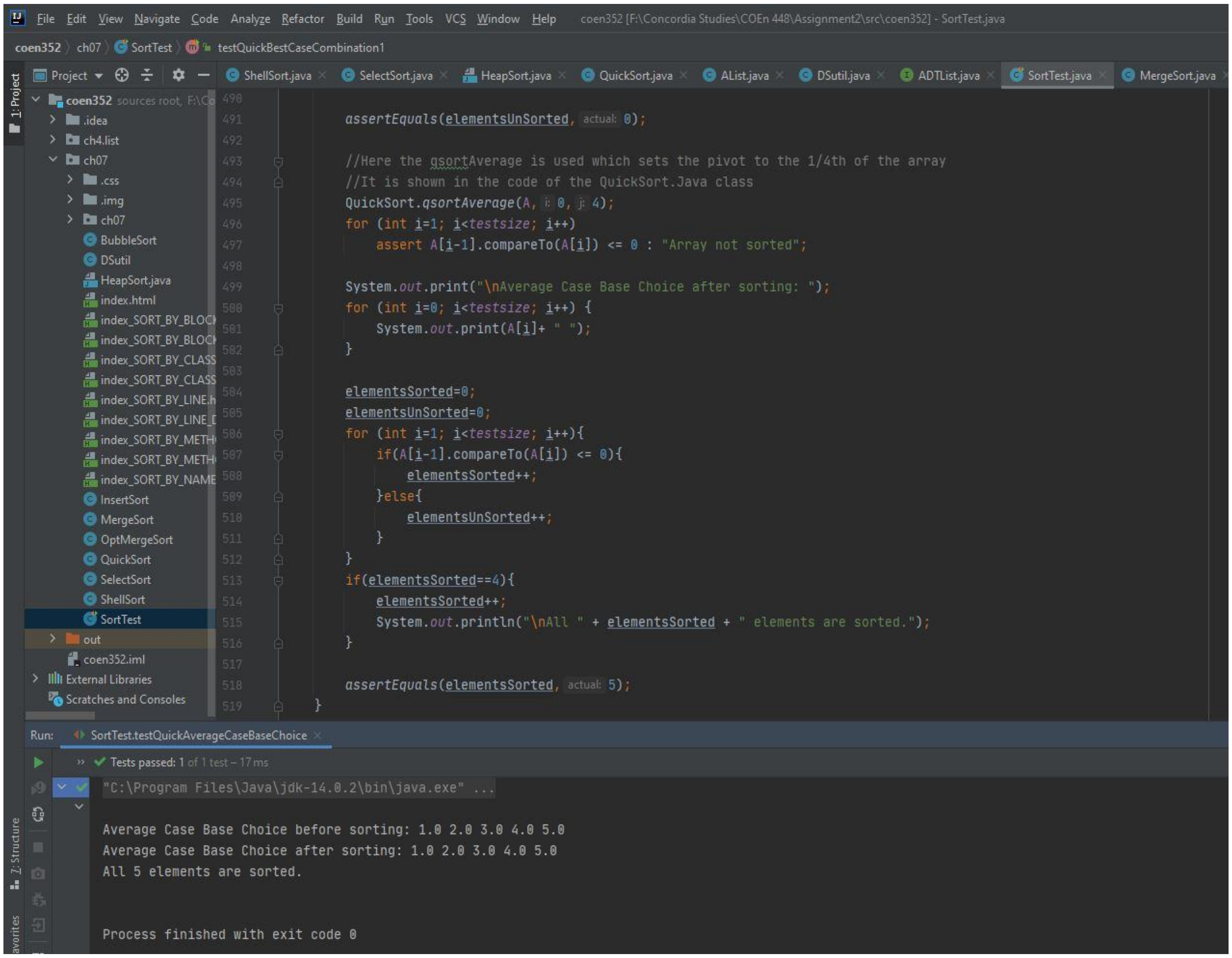
So, the combinations would be:

1. (AverageB1A, AverageB2B) 2. (AverageB2A, AverageB1B)

In terms of Average case I have edited the test case in such a way that the pivot is the last element. I have named it findpivotAverage and used it in qsortAverage method of the QuickSort.java class

Below I have only included the Base Choice screenshot whereas the other tests are same and I have performed them in my SortTest.java class. So please refer to the code for checking the code for the combinations.





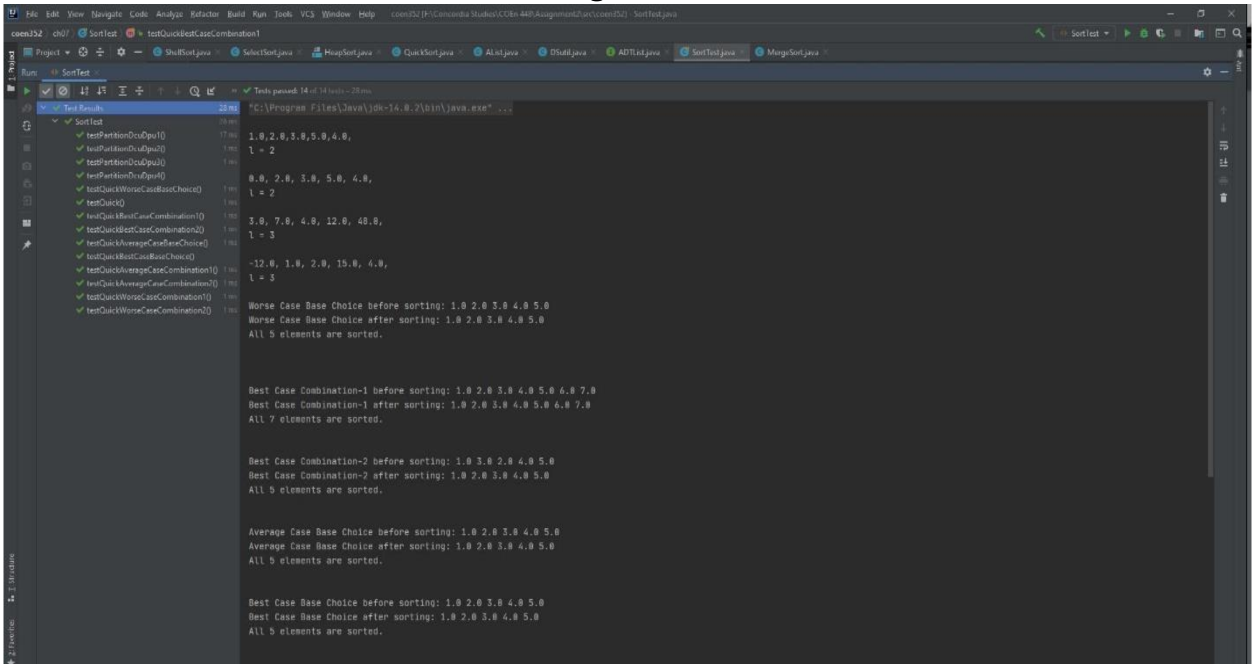
```java
        assertEquals(elementsUnSorted, actual: 0);

        //Here the qsortAverage is used which sets the pivot to the 1/4th of the array
        //It is shown in the code of the QuickSort.Java class
        QuickSort.qsortAverage(A, i: 0, j: 4);
        for (int i=1; i<testsize; i++)
            assert A[i-1].compareTo(A[i]) <= 0 : "Array not sorted";

        System.out.print("\nAverage Case Base Choice after sorting: ");
        for (int i=0; i<testsize; i++) {
            System.out.print(A[i]+ " ");
        }

        elementsSorted=0;
        elementsUnSorted=0;
        for (int i=1; i<testsize; i++){
            if(A[i-1].compareTo(A[i]) <= 0){
                elementsSorted++;
            }else{
                elementsUnSorted++;
            }
        }
        if(elementsSorted==4){
            elementsSorted++;
            System.out.println("\nAll " + elementsSorted + " elements are sorted.");
        }

        assertEquals(elementsSorted, actual: 5);
    }
```

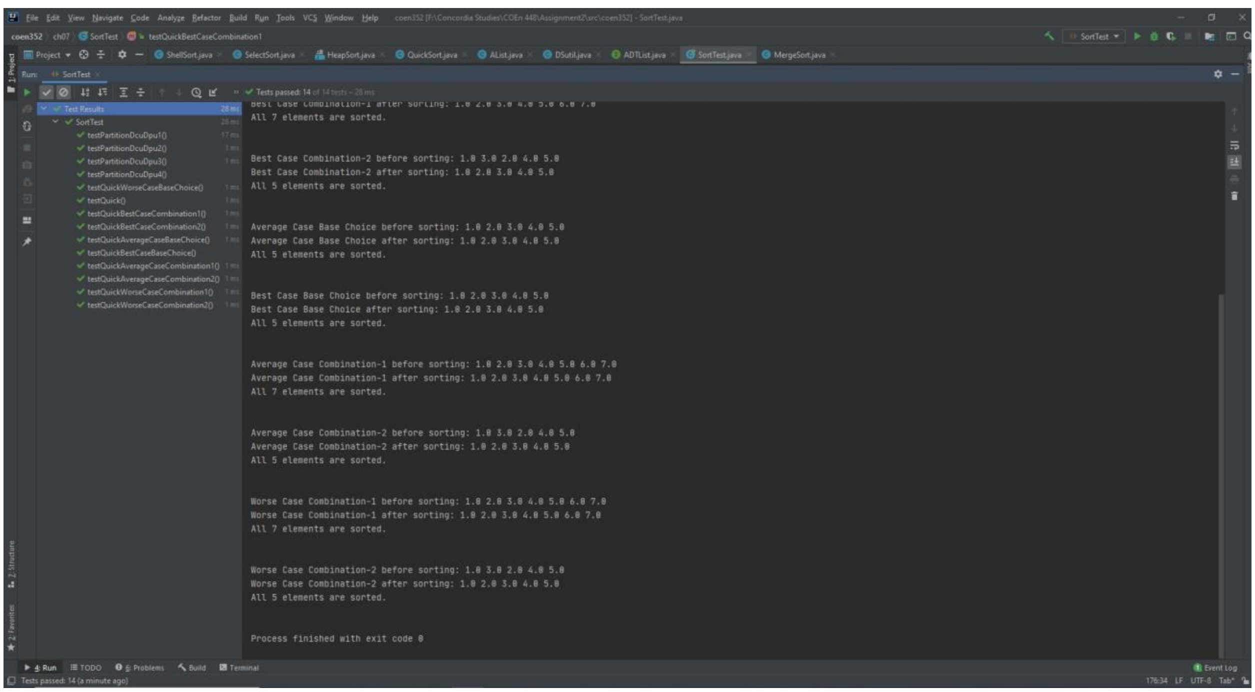
Run: SortTest.testQuickAverageCaseBaseChoice

Tests passed: 1 of 1 test – 17 ms

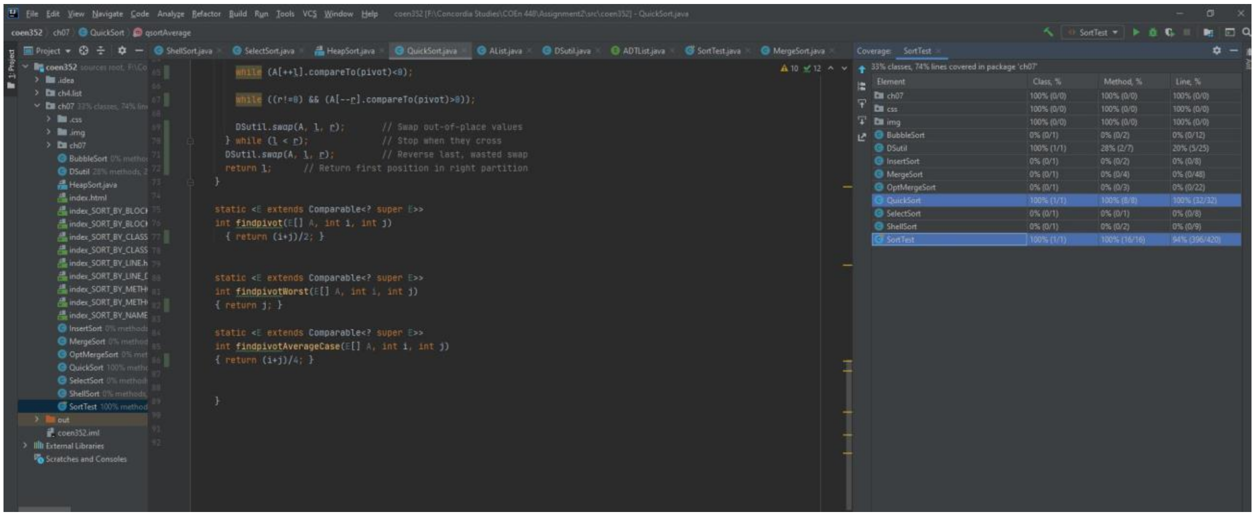
```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" ...

Average Case Base Choice before sorting: 1.0 2.0 3.0 4.0 5.0
Average Case Base Choice after sorting: 1.0 2.0 3.0 4.0 5.0
All 5 elements are sorted.


Process finished with exit code 0
```

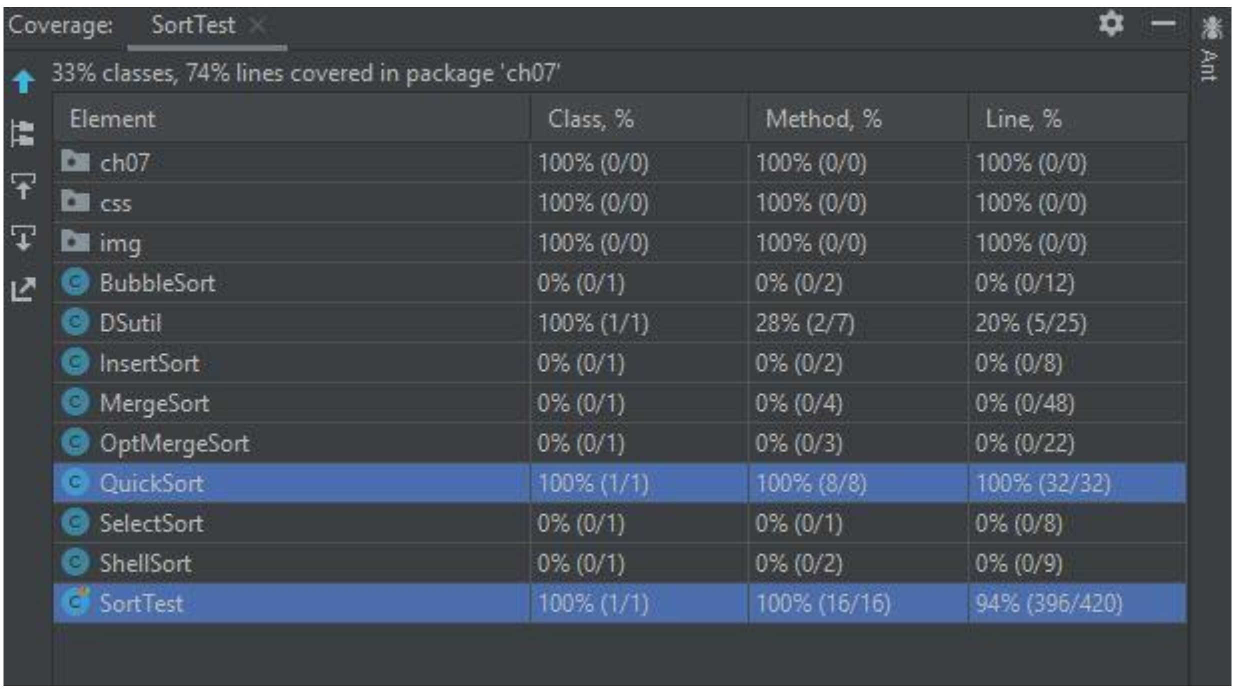
Below are the screenshot of all tests running:





Below are the Screenshot of the coverages:





Coverage: SortTest

33% classes, 74% lines covered in package 'ch07'

Element	Class, %	Method, %	Line, %
ch07	100% (0/0)	100% (0/0)	100% (0/0)
css	100% (0/0)	100% (0/0)	100% (0/0)
img	100% (0/0)	100% (0/0)	100% (0/0)
BubbleSort	0% (0/1)	0% (0/2)	0% (0/12)
DSutil	100% (1/1)	28% (2/7)	20% (5/25)
InsertSort	0% (0/1)	0% (0/2)	0% (0/8)
MergeSort	0% (0/1)	0% (0/4)	0% (0/48)
OptMergeSort	0% (0/1)	0% (0/3)	0% (0/22)
QuickSort	100% (1/1)	100% (8/8)	100% (32/32)
SelectSort	0% (0/1)	0% (0/1)	0% (0/8)
ShellSort	0% (0/1)	0% (0/2)	0% (0/9)
SortTest	100% (1/1)	100% (16/16)	94% (396/420)

[all classes] [ch07]

Coverage Summary for Class: QuickSort (ch07)

Class	Class, %	Method, %	Line, %
QuickSort	100% (1/ 1)	88.9% (8/ 9)	97% (32/ 33)

```java
package ch07;

public class QuickSort {

        public static <E extends Comparable<? super E>> void sort(E[] A) {
                qsort(A, 0, A.length-1);
        }

        static <E extends Comparable<? super E>>
        void qsort(E[] A, int i, int j) {      // Quicksort

          int pivotindex = findpivot(A, i, j); // Pick a pivot

          DSutil.swap(A, pivotindex, j);       // Stick pivot at end

          // k will be the first position in the right subarray

          int k = partition(A, i-1, j, A[j]);

          DSutil.swap(A, k, j);  // Put pivot in place

          if ((k-i) > 1) qsort(A, i, k-1);   // Sort left partition
          if ((j-k) > 1) qsort(A, k+1, j);   // Sort right partition
        }

        static <E extends Comparable<? super E>>
        void qsortAverage(E[] A, int i, int j) {      // Quicksort

                int pivotindex = findpivotAverageCase(A, i, j); // Pick a pivot

                DSutil.swap(A, pivotindex, j);       // Stick pivot at end

                // k will be the first position in the right subarray

                int k = partition(A, i-1, j, A[j]);

                DSutil.swap(A, k, j);  // Put pivot in place

                if ((k-i) > 1) qsort(A, i, k-1);   // Sort left partition
                if ((j-k) > 1) qsort(A, k+1, j);   // Sort right partition
        }

        static <E extends Comparable<? super E>>
        void qsortWorse(E[] A, int i, int j) {      // Quicksort

                int pivotindex = findpivotWorst(A, i, j); // Pick a pivot

                DSutil.swap(A, pivotindex, j);       // Stick pivot at end
                // k will be the first position in the right subarray

                int k = partition(A, i-1, j, A[j]);

                DSutil.swap(A, k, j);  // Put pivot in place

                if ((k-i) > 1) qsort(A, i, k-1);   // Sort left partition
                if ((j-k) > 1) qsort(A, k+1, j);   // Sort right partition
        }



        static <E extends Comparable<? super E>>
        int partition(E[] A, int l, int r, E pivot) {

          do {// Move bounds inward until they meet

                while (A[++l].compareTo(pivot)<0);

                while ((r!=0) && (A[--r].compareTo(pivot)>0));

                DSutil.swap(A, l, r);       // Swap out-of-place values
          } while (l < r);              // Stop when they cross
          DSutil.swap(A, l, r);         // Reverse last, wasted swap
          return l;      // Return first position in right partition
        }

        static <E extends Comparable<? super E>>
        int findpivot(E[] A, int i, int j)
          { return (i+j)/2; }


        static <E extends Comparable<? super E>>
        int findpivotWorst(E[] A, int i, int j)
        { return j; }

        static <E extends Comparable<? super E>>
        int findpivotAverageCase(E[] A, int i, int j)
        { return (i+j)/4; }


        }
```

generated on 2022-03-09 20:19

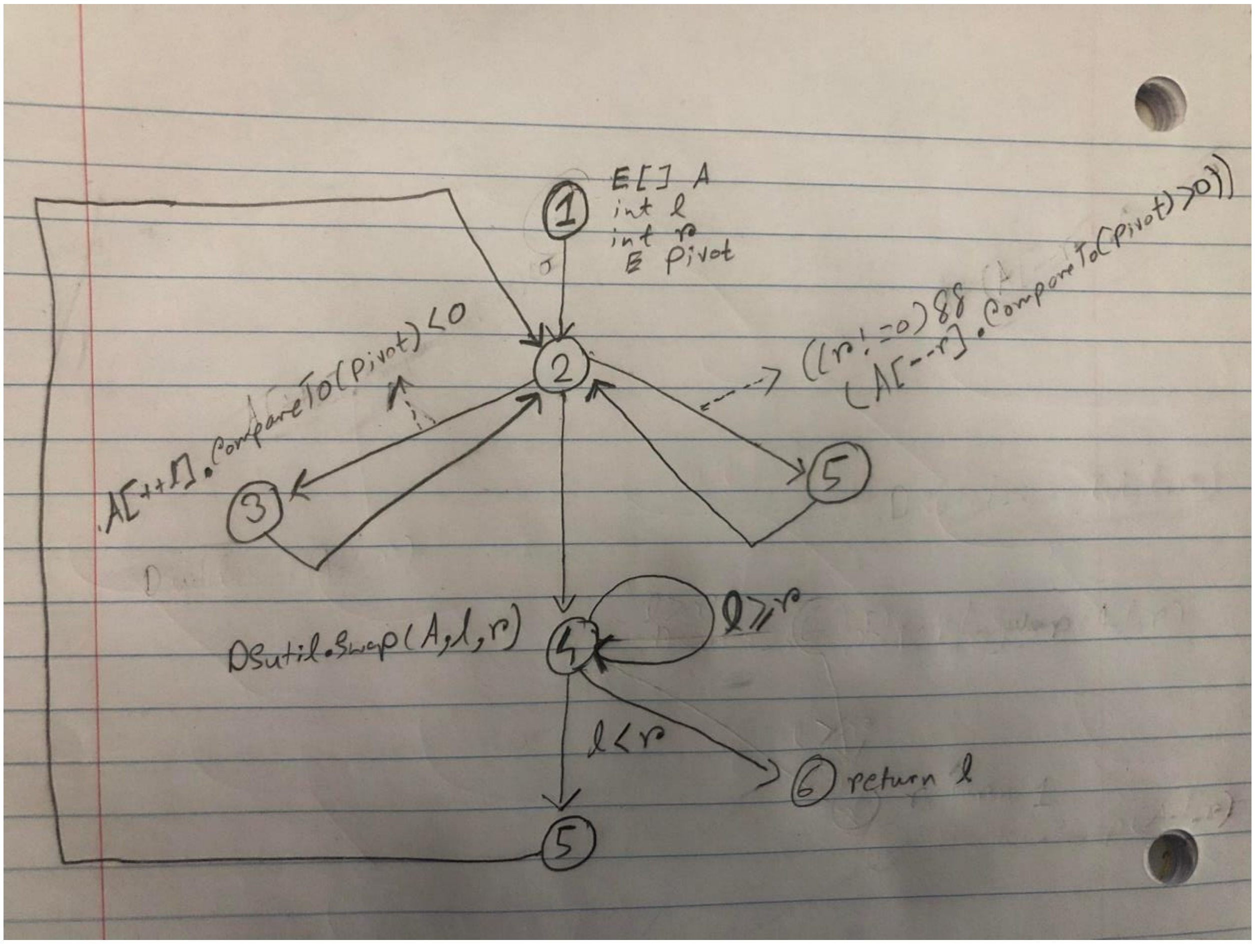
[all classes] [ch07]

Coverage Summary for Class: SortTest (ch07)

Class	Class, %	Method, %	Line, %
SortTest	100% (1/ 1)	100% (17/ 17)	94.3% (396/ 420)

```java
package ch07;
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import ch4.list.*;


public class SortTest {

        final static int testsize = 5;
        Double A[];
        static int THRESHOLD = 8;

         private static ADTList<Integer> list;

        @SuppressWarnings("removal")
        @BeforeEach
        void setUp() throws Exception {
//              A = new Double[testsize];
//          for (int i=0; i<testsize; i++) {
//                      //A[i] = new Double(DSutil.random(3));// return int basic type, new Integer from int.
//
//          }
           //list = new LList<Integer>();
            list = new DList<Integer>();
            // list = new AList<Integer>();

        }

        @Test
        void testQuick() {

                A = new Double[testsize];
                for (int i=0; i<testsize; i++) {
                        //A[i] = new Double(DSutil.random(3));// return int basic type, new Integer from int.

                        A[0]=new Double(5);
                        A[1]=new Double(3);
                        A[2]=new Double(2);
                        A[3]=new Double(1);
                        A[4]=new Double(4);

                }

                QuickSort.sort(A);
                for (int i=1; i<testsize; i++)
                        assert A[i-1].compareTo(A[i]) <= 0 : "Array not sorted";

//              for (int i=0; i<testsize; i++) {
//                      System.out.print(A[i]+ " ");
//              }
        }
        // The Pivot is set to the median element in the Array
        @Test
        void testQuickBestCaseBaseChoice() {

```

2.3.1 Produce a CFG of the partition function. Leverage the table based def-use pair approach in Question 1, and produce the table below for variable pivot.



node i	dcu(v,i)	dpu(v,i)
node 1	dcu(A,1) = {4} dcu(l,1) = {4,6} dcu(r,1) = {4} dcu(Pivot,1) = { }	dpu(A,1) = {2,3} dpu(A,1) = {2,5} dpu(l,1) = {2,3} dpu(l,1) = {4,4} dpu(l,1) = {4,5} dpu(r,1) = {2,5} dpu(r,1) = {4,4} dpu(r,1) = {4.5} dpu(pivot,1) = {2,3} dpu(pivot,1) = {2,5}
node 4	dcu(A,4) = {4} dcu(l,4) = {4,6} dcu(r,4) = {4}	dpu(l,4) = {4,5} dpu(l,4) = {4,4} dpu(r,4) = {4,5} dpu(r,4) = {4,4}

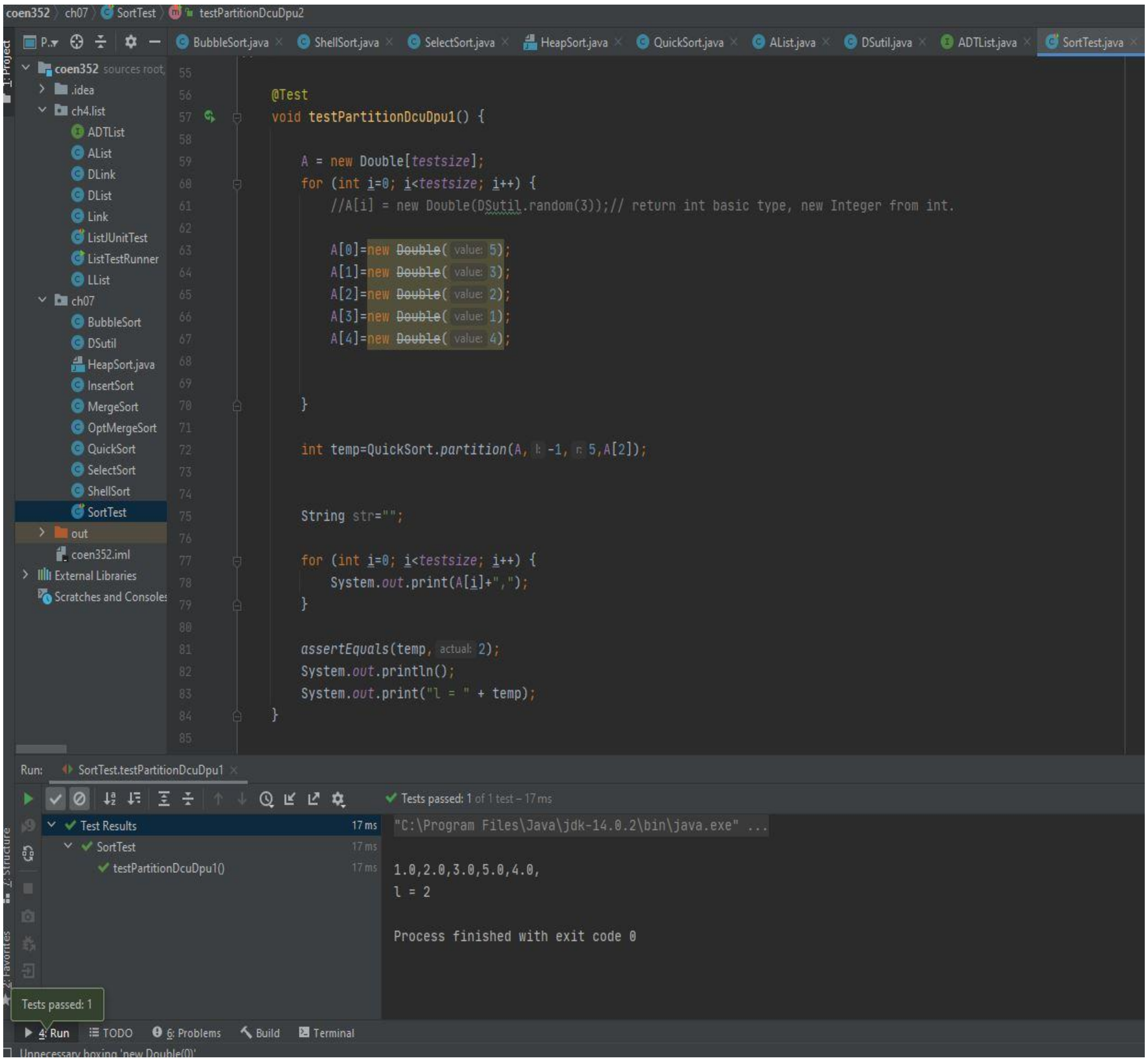
(2.3.2) Develop test cases to cover all the dcu and dpu.

Test1 dcu and dpu all nodes

Input data < “A”, “l”, “r” | <5,3,1,2,4>,-1, 5,A[2] >

Expected output: l=2

After partition: 1, 2, 3, 5, 4 → here l=2 (A[2]=3), r =1(A[1]=2), and pivot is A[1]=2.

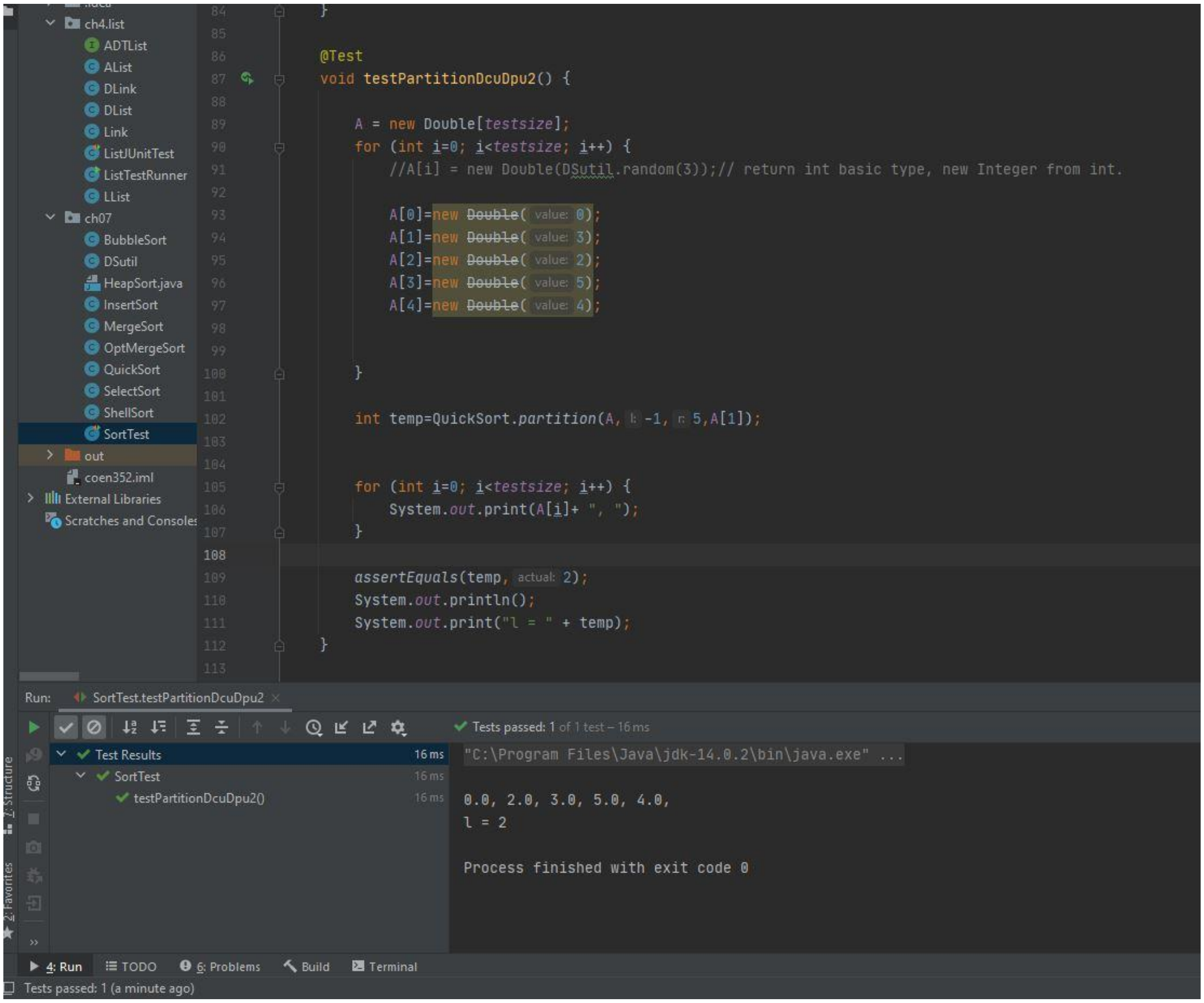


Test2 dcu and dpu all nodes

Input data < “A”, “l”, “r” | <0,3,2,5,4>,-1, 5,A[1] >

Expected output: l=2

After partition: 0, 2, 3, 5, 4 → here l=2 (A[2]=3), r =1(A[1]=2), and pivot is A[2]=3.

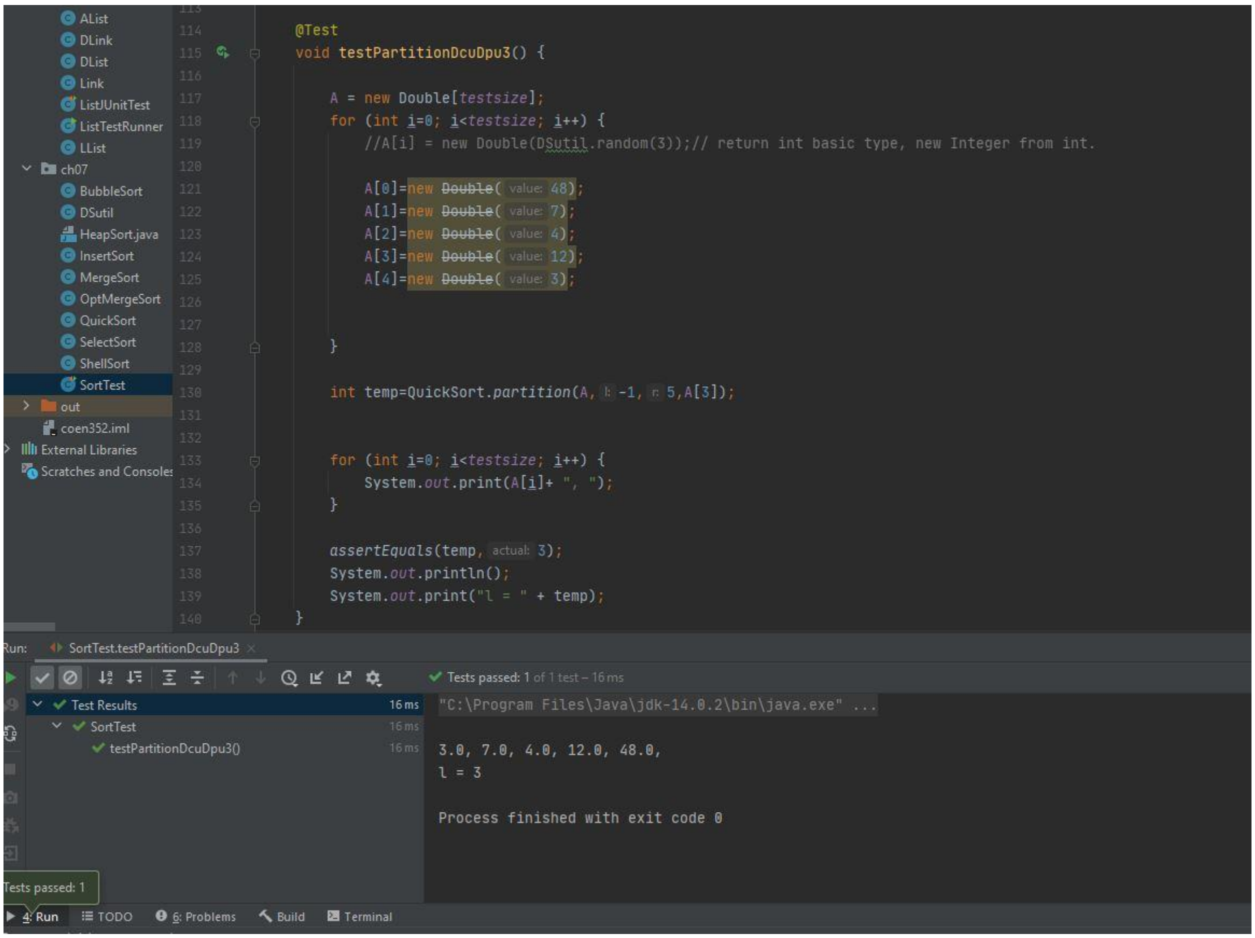


Test3 dcu and dpu all nodes

Input data < “A”, “l”, “r” | <48,7,4,12,3>,-1, 5,A[3] >

Expected output: l=3

After partition: 3, 7, 4, 12, 48→ here l=3 (A[3]=12), r =2(A[2]=4), and pivot is A[3]=12.

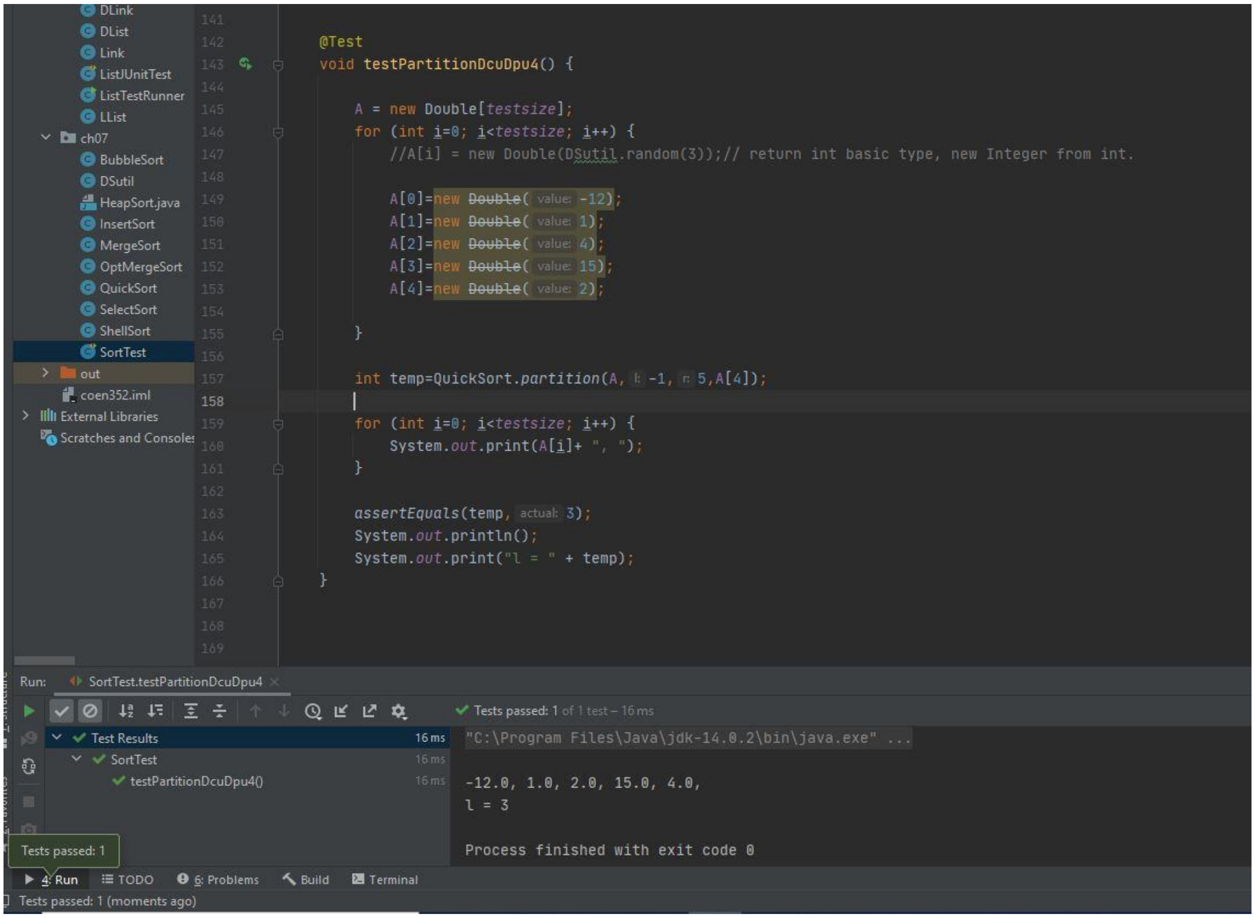


Test4 dcu and dpu all nodes

Input data < “A”, “l”, “r” | <-12,1,4,15,2>, -1, 5,A[4] >

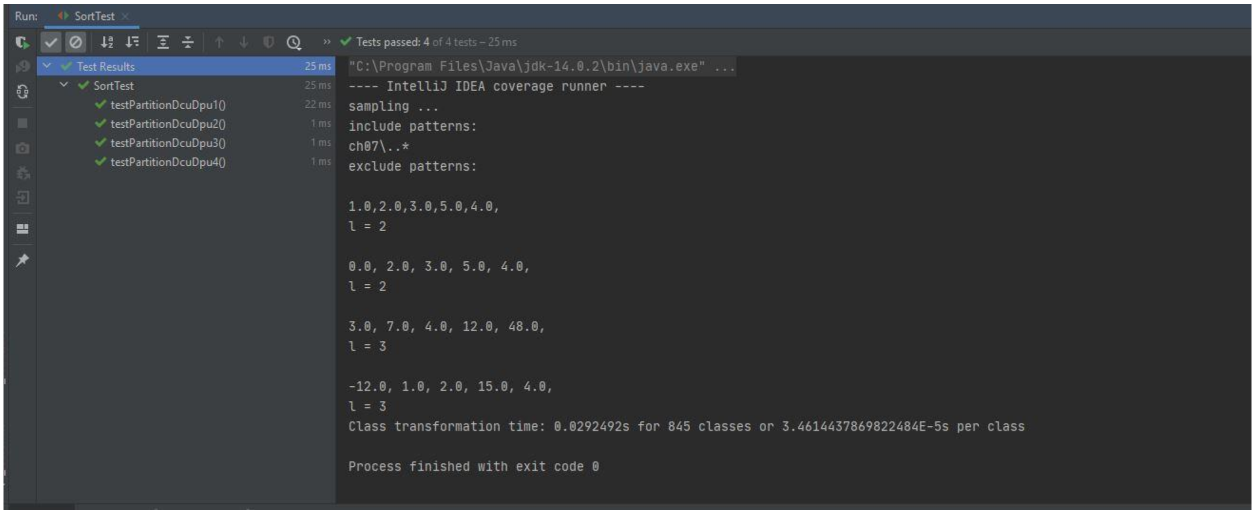
Expected output: l=3

After partition: -12, 1, 2, 15, 4,→ here l=3 (A[3]=2), r =2(A[2]=2), and pivot is A[2]=4.

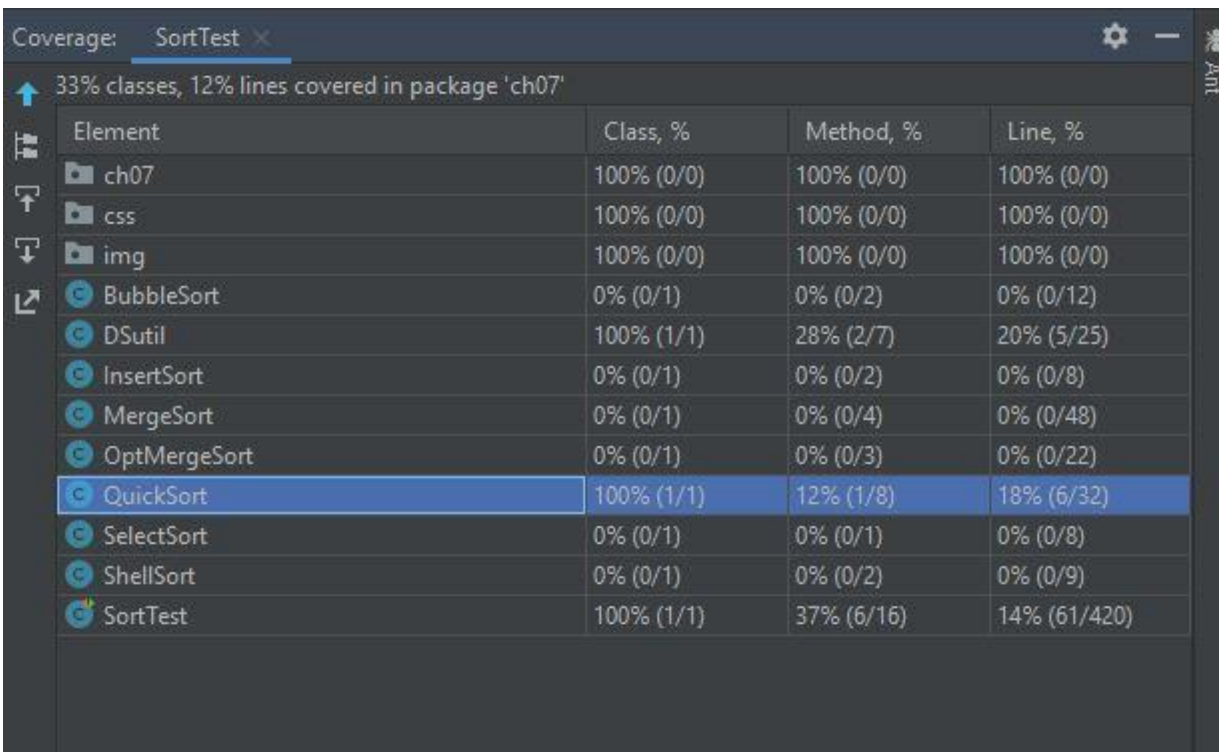


(2.3.3) Program unit test cases, run the test cases and produce a coverage report from you IDE.

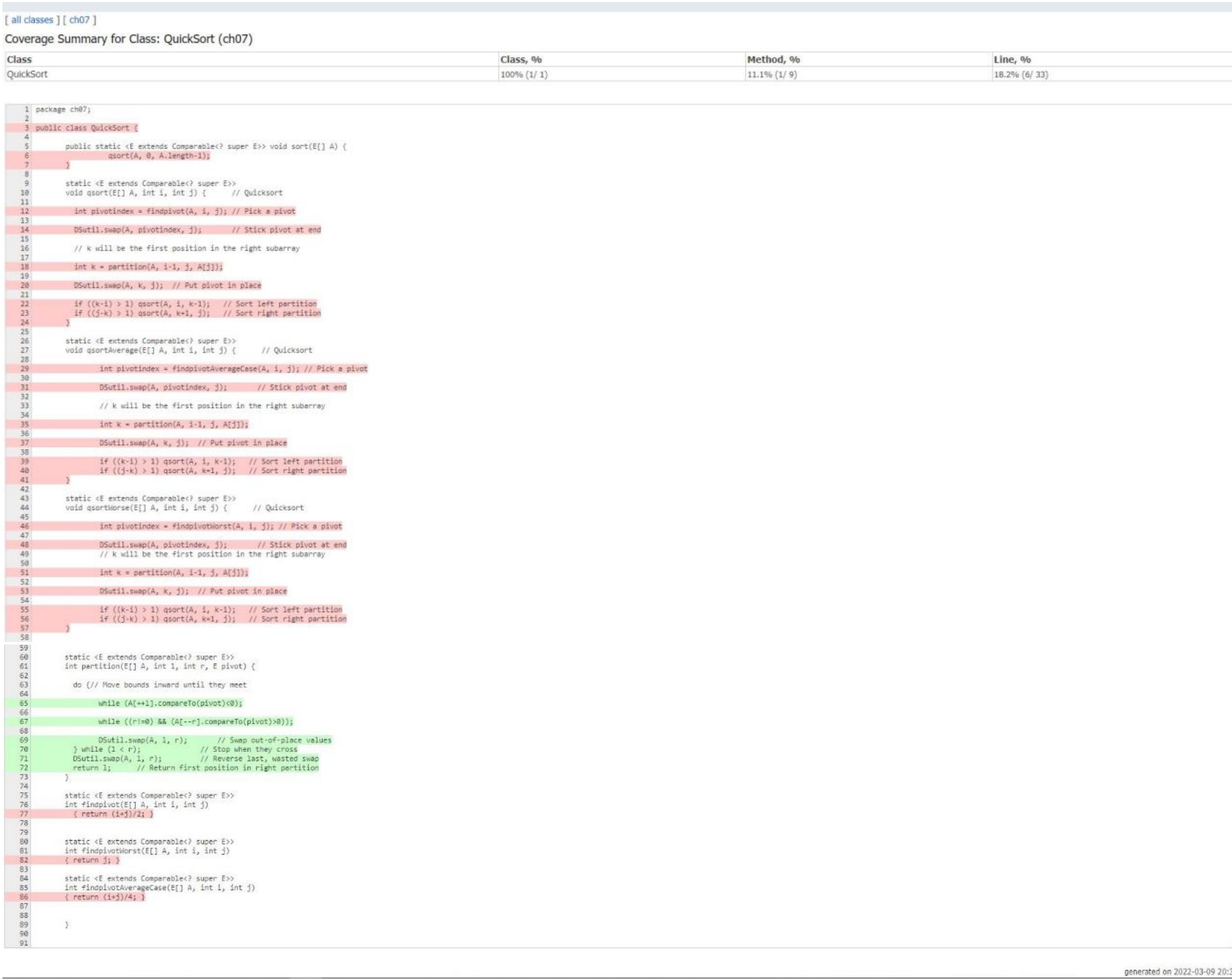
Test Case Running:



Test Case Coverage Report:



Test Coverage Lines:



[all classes] [ch07]

Coverage Summary for Class: QuickSort (ch07)

Class	Class, %	Method, %	Line, %
QuickSort	100% (1/ 1)	11.1% (1/ 9)	18.2% (6/ 33)

```
package ch07;

public class QuickSort {

        public static <E extends Comparable<? super E>> void sort(E[] A) {
                qsort(A, 0, A.length-1);
        }

        static <E extends Comparable<? super E>>
        void qsort(E[] A, int i, int j) {      // Quicksort

          int pivotindex = findpivot(A, i, j); // Pick a pivot

          DSutil.swap(A, pivotindex, j);       // Stick pivot at end

          // k will be the first position in the right subarray

          int k = partition(A, i-1, j, A[j]);

          DSutil.swap(A, k, j);  // Put pivot in place

          if ((k-i) > 1) qsort(A, i, k-1);   // Sort left partition
          if ((j-k) > 1) qsort(A, k+1, j);   // Sort right partition
        }

        static <E extends Comparable<? super E>>
        void qsortAverage(E[] A, int i, int j) {      // Quicksort

                int pivotindex = findpivotAverageCase(A, i, j); // Pick a pivot

                DSutil.swap(A, pivotindex, j);       // Stick pivot at end

                // k will be the first position in the right subarray

                int k = partition(A, i-1, j, A[j]);

                DSutil.swap(A, k, j);  // Put pivot in place

                if ((k-i) > 1) qsort(A, i, k-1);   // Sort left partition
                if ((j-k) > 1) qsort(A, k+1, j);   // Sort right partition
        }

        static <E extends Comparable<? super E>>
        void qsortWorse(E[] A, int i, int j) {      // Quicksort

                int pivotindex = findpivotWorst(A, i, j); // Pick a pivot

                DSutil.swap(A, pivotindex, j);       // Stick pivot at end
                // k will be the first position in the right subarray

                int k = partition(A, i-1, j, A[j]);

                DSutil.swap(A, k, j);  // Put pivot in place

                if ((k-i) > 1) qsort(A, i, k-1);   // Sort left partition
                if ((j-k) > 1) qsort(A, k+1, j);   // Sort right partition
        }


        static <E extends Comparable<? super E>>
        int partition(E[] A, int l, int r, E pivot) {

          do {// Move bounds inward until they meet

                while (A[++l].compareTo(pivot)<0);

                while ((r!=0) && (A[--r].compareTo(pivot)>0));

                DSutil.swap(A, l, r);       // Swap out-of-place values
          } while (l < r);              // Stop when they cross
          DSutil.swap(A, l, r);         // Reverse last, wasted swap
          return l;      // Return first position in right partition
        }

        static <E extends Comparable<? super E>>
        int findpivot(E[] A, int i, int j)
          { return (i+j)/2; }


        static <E extends Comparable<? super E>>
        int findpivotWorst(E[] A, int i, int j)
        { return j; }

        static <E extends Comparable<? super E>>
        int findpivotAverageCase(E[] A, int i, int j)
        { return (i+j)/4; }


        }
```

generated on 2022-03-09 20:33

Discussion:

Upon looking at the both the coverage report, the difference I can indicate is that the coverage for quick sort is greater than that of the partition, the main reason I would say is that the quick Sort method basically calls all the functions in that class and hence it covers more methods and lines compared to the partition function. For the partition test it only covers 11.1% of the method and 18.2% of the lines whereas for the qsort method it covers 88.9% of the methods and 97% of the lines as the qsort method calls all the methods in that class and hence it has more coverage.

So, the difference in the coverage is that one coverage is for the partition and another is for the qsort(quick sort). The quick sort covers more as it calls all the methods and partition is just a part of the quick sort and hence the difference in coverage is high. Specially as partition is just a part of qsort which is called in the qsort method. Hence when we do the partition the %coverage is very less compared to the qsort.

The main pros of data flow testing are that it can define intermediary control analysis criteria between all nodes and all paths. It can also handle variables definition and usage, moreover it spans the gap between all paths and branch testing. And the cons for data flow testing are that it is unscalable data-flow algorithm for large real-world programs and Test cases design difficulties are higher compared with Control flow testing. Moreover, infeasible test objects can lead to wastage of time on testing.

The Pros of input domain modeling is that it checks for all the possible inputs to that program. Even for small programs the domain is infinite. Moreover, this testing checks fundamentally with finite sets of values making sure every single thing is working fine. The cons would be that it is time consuming, and the expense is high as checking every method fundamentally takes time and money.

So here, to conclude, I would say that for this assignment we could go for input domain modelling to check each method fundamentally as it does not have much methods in Quick Sort class. Hence, input domain modelling is a good way of testing the quicksort method fundamentally.

I have included the entire project along with the report, I have made some modifications on the QuickSort class and sortTest class.