

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 >	Input data < “X”, “Y” 2, 1 >	Input data < “X”, “Y” 1, 0 >
Expected output at (3): Z=1	Expected output (3): Z=2	Expected output (3): Z=1
Expected output at (6): Z=1	Expected output (6): Z=2	Expected output (6): Z=1

Test dcu(Z,3)

Input data < “X”, “Y” 1, 2 >	Input data < “X”, “Y” 2, 0 >	Input data < “X”, “Y” 1, -2 >
Expected output at (3): Z=1	Expected output (3): Z=2	Expected output (3): Z=1
Expected output at (5): Z=null	Expected output (5): Z=null	Expected output (5): Z=1
Expected output at (6): Z=1	Expected output (6): Z=2	Expected output (6): Z=1

Test dcu(W,3)

Input data < “X”, “Y” 1, 2 >	Input data < “X”, “Y” 2, 0 >	Input data < “X”, “Y” 1, -2 >
Expected output at (3): Z=1	Expected output (3): Z=2	Expected output (3): Z=1

Test dcu(Z,5)

Input data < “X”, “Y” 1, 2 >	Input data < “X”, “Y” 2, 0 >	Input data < “X”, “Y” 1, -2 >
Expected output at (5): Z=null	Expected output (5): Z=null	Expected output (5): Z=1

Test dpu(Y,1)

Input data < “X”, “Y” 1, 2 >	Input data < “X”, “Y” 2, 0 >	Input data < “X”, “Y” 1, -2 >
Expected output at (5): Z=null	Expected output (5): Z=null	Expected output (5): Z=1
Expected output at (6): Z=1	Expected output (6): Z=2	Expected output (6): Z=1

Test dpu(W,1)

Input data < “X”, “Y” 1, 2 >	Input data < “X”, “Y” 2, 0 >	Input data < “X”, “Y” 1, -2 >
Expected output at (3): Z=1	Expected output (3): Z=2	Expected output (3): Z=1
Expected output at (5): Z=null	Expected output (5): Z=null	Expected output (5): Z=1
Expected output at (6): Z=1	Expected output (6): Z=2	Expected output (6): Z=1

Test dpu(W,3)

Input data < “X”, “Y” 1, 2 >	Input data < “X”, “Y” 2, 0 >	Input data < “X”, “Y” 1, -2 >
Expected output at (3): Z=1	Expected output (3): Z=2	Expected output (3): Z=1
Expected output at (4): Z=1	Expected output (4): Z=2	Expected output (4): Z=1
Expected output at (5): Z=null	Expected output (5): Z=null	Expected output (5): Z=1
Expected output at (6): Z=1	Expected output (6): Z=2	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.

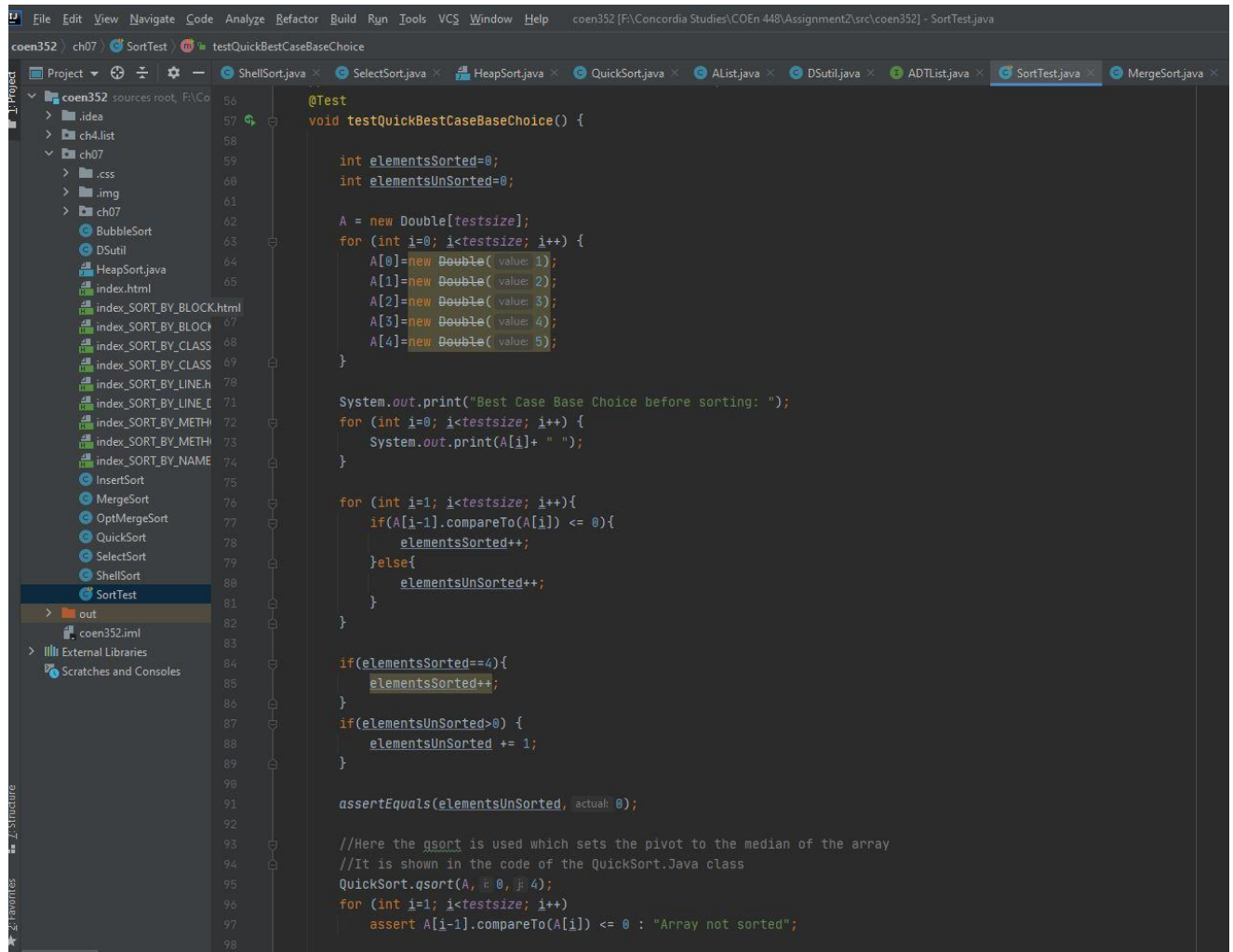
So, the combinations would be:

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

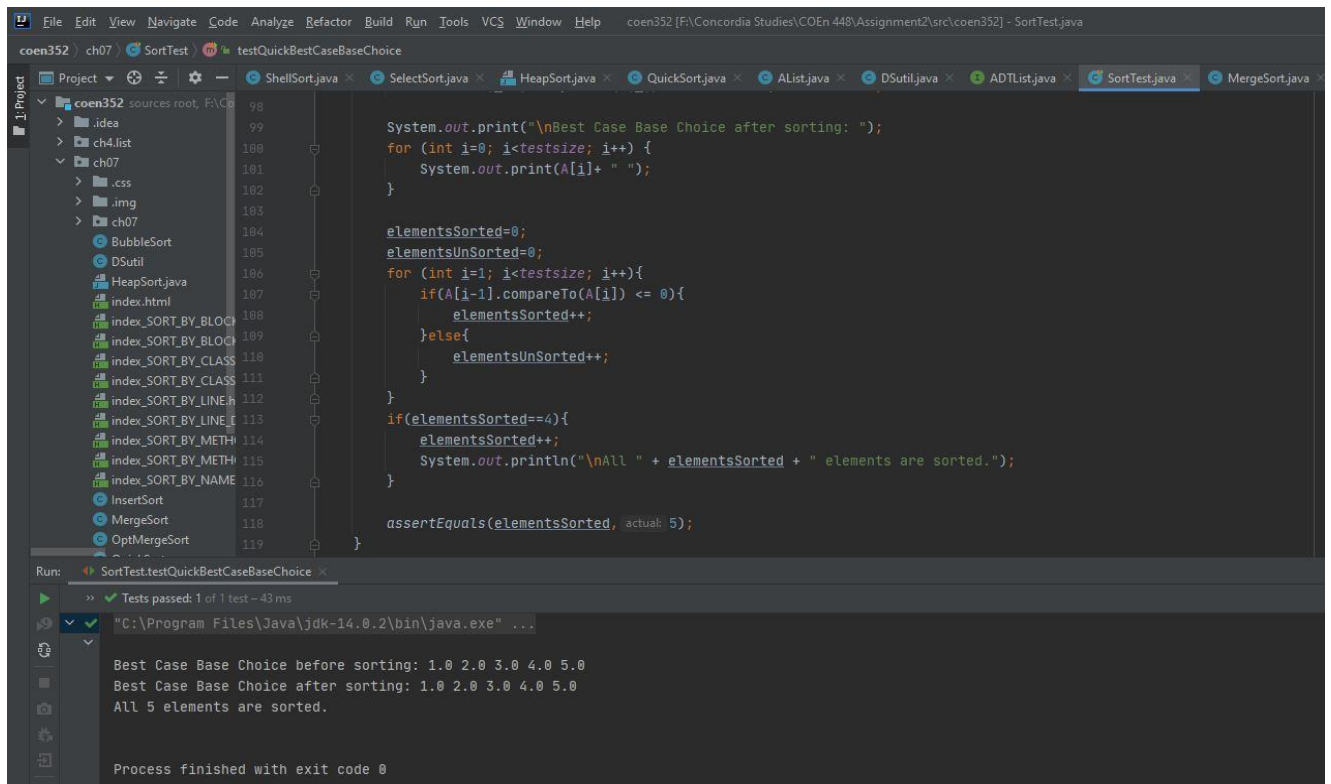
In terms of best case I have edited the test case in such a way that the pivot is the mean of the number of elements that are available. I have named it findpivot and used it in qsort method of the QuickSort.java class

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

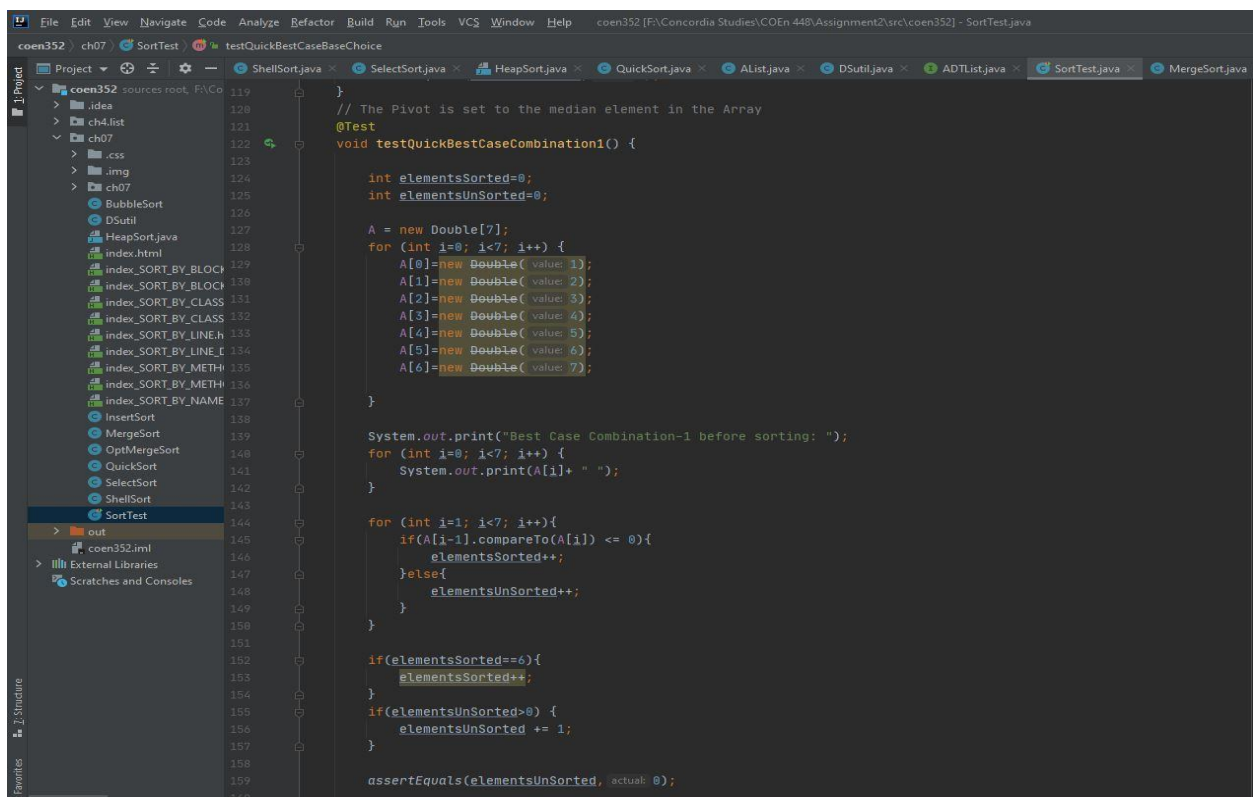
Base Case Best Case:



```
56  @Test
57  void testQuickBestCaseBaseChoice() {
58
59      int elementsSorted=0;
60      int elementsUnSorted=0;
61
62      A = new Double[testsized];
63      for (int i=0; i<testsized; i++) {
64          A[0]=new Double( value: 1);
65          A[1]=new Double( value: 2);
66          A[2]=new Double( value: 3);
67          A[3]=new Double( value: 4);
68          A[4]=new Double( value: 5);
69      }
70
71      System.out.print("Best Case Base Choice before sorting: ");
72      for (int i=0; i<testsized; i++) {
73          System.out.print(A[i]+ " ");
74      }
75
76      for (int i=1; i<testsized; i++){
77          if(A[i-1].compareTo(A[i]) <= 0){
78              elementsSorted++;
79          }else{
80              elementsUnSorted++;
81          }
82      }
83
84      if(elementsSorted==4){
85          elementsSorted++;
86      }
87      if(elementsUnSorted>0) {
88          elementsUnSorted += 1;
89      }
90
91      assertEquals(elementsUnSorted, actual: 0);
92
93      //Here the qsort is used which sets the pivot to the median of the array
94      //It is shown in the code of the QuickSort.java class
95      QuickSort.qsort(A, 0, 4);
96      for (int i=1; i<testsized; i++)
97          assert A[i-1].compareTo(A[i]) <= 0 : "Array not sorted";
98  }
```



Combination 1 Best Case:



```
coen352 [F:\Concordia Studies\COEn 448\Assignment2\src\coen352] - SortTest.java
coen352 ch07 SortTest testQuickBestCaseBaseChoice

//Here the gsort is used which sets the pivot to the median of the array
//It is shown in the code of the QuickSort.Java class
QuickSort.qsort(A, 0, 6, 4);
for (int i=1; i<7; i++)
    assert A[i-1].compareTo(A[i]) <= 0 : "Array not sorted";

System.out.print("\nBest Case Combination-1 after sorting: ");
for (int i=0; i<7; i++) {
    System.out.print(A[i]+ " ");
}

elementsSorted=0;
elementsUnSorted=0;
for (int i=1; i<7; i++){
    if(A[i-1].compareTo(A[i]) <= 0){
        elementsSorted++;
    }else{
        elementsUnSorted++;
    }
}

if(elementsSorted==6){
    elementsSorted++;
    System.out.println("\nAll " + elementsSorted + " elements are sorted.");
}

assertEquals(elementsSorted, actual: 7);

Run: SortTest.testQuickBestCaseCombination1
Tests passed: 1 of 1 test - 20 ms
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" ...
Best Case Combination-1 before sorting: 1.0 2.0 3.0 4.0 5.0 6.0 7.0
Best Case Combination-1 after sorting: 1.0 2.0 3.0 4.0 5.0 6.0 7.0
All 7 elements are sorted.
Process finished with exit code 0
```

Combination 2 Best Case:

```
coen352 ch07 SortTest testQuickBestCaseCombination1

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

System.out.print("\nBest Case Combination-2 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.testQuickBestCaseCombination2
Tests passed: 1 of 1 test - 17 ms
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" ...
Best Case Combination-2 before sorting: 1.0 3.0 2.0 4.0 5.0
Best Case Combination-2 after sorting: 1.0 2.0 3.0 4.0 5.0
All 5 elements are sorted.
Process finished with exit code 0
```

```

188 // The Pivot is set to the median element in the Array
189 @Test
190 void testQuickBestCaseCombination2() {
191
192     int elementsSorted=0;
193     int elementsUnsorted=0;
194
195     A = new Double[testsSize];
196     for (int i=0; i<testsSize; i++) {
197         A[i]=new Double( value: 1);
198         A[i]=new Double( value: 3);
199         A[i]=new Double( value: 2);
200         A[i]=new Double( value: 4);
201         A[i]=new Double( value: 5);
202     }
203
204     System.out.print("Best Case Combination-2 before sorting: ");
205     for (int i=0; i<testsSize; i++) {
206         System.out.print(A[i]+ " ");
207     }
208
209     for (int i=1; i<testsSize; i++){
210         if(A[i-1].compareTo(A[i]) <= 0){
211             elementsSorted++;
212         }else{
213             elementsUnsorted++;
214         }
215     }
216
217     if(elementsSorted==4){
218         elementsSorted++;
219     }
220
221     if(elementsUnsorted>0) {
222         elementsUnsorted += 1;
223     }
224
225     assertEquals(elementsUnsorted, actual: 2);
226

```

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.

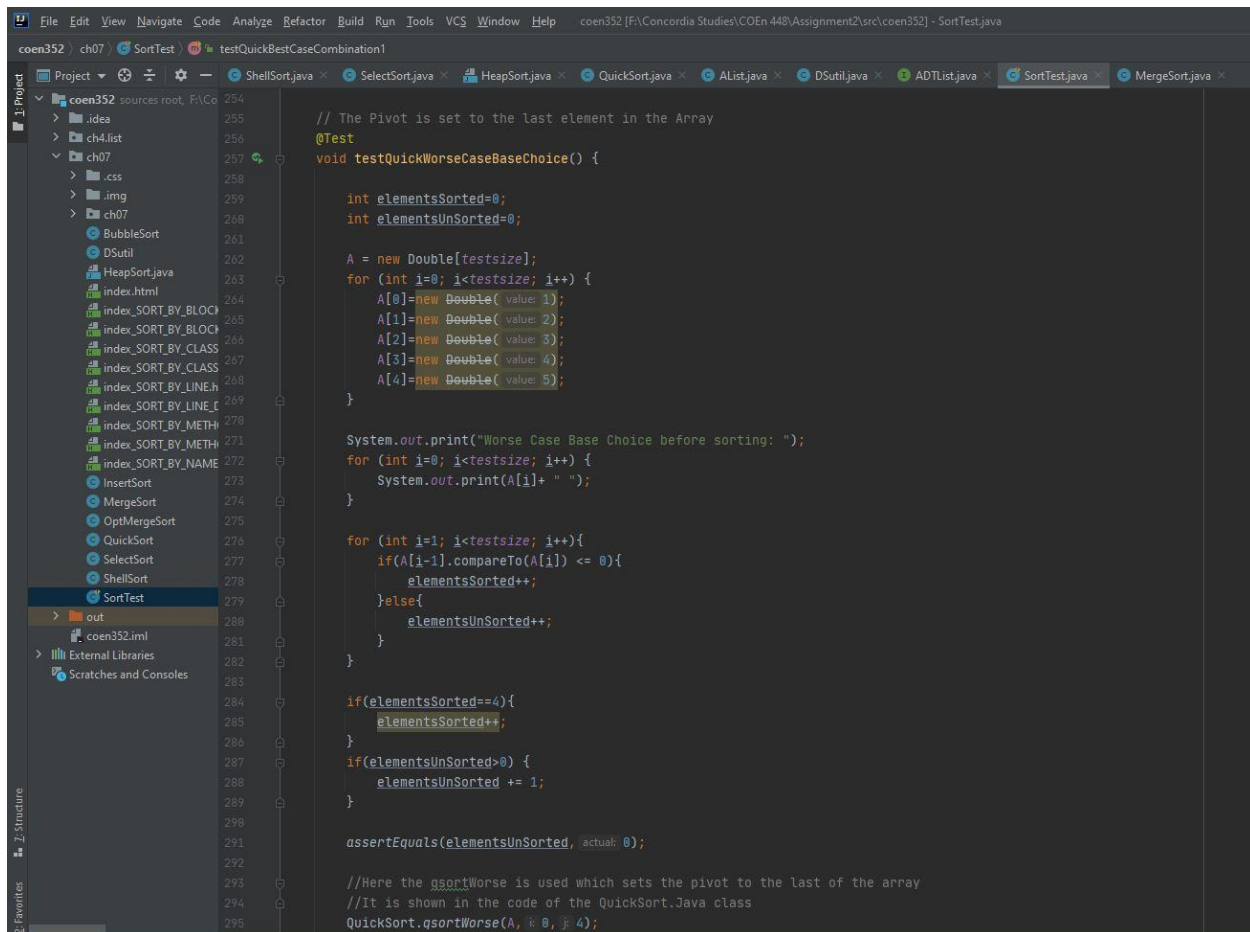
So, the combinations would be:

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 Pivot is set to the last element in the Array
@Test
void testQuickWorseCaseBaseChoice() {

    int elementsSorted=0;
    int elementsUnSorted=0;

    A = new Double[testsize];
    for (int i=0; i<testsize; i++) {
        A[0]=new Double( value: 1);
        A[1]=new Double( value: 2);
        A[2]=new Double( value: 3);
        A[3]=new Double( value: 4);
        A[4]=new Double( value: 5);
    }

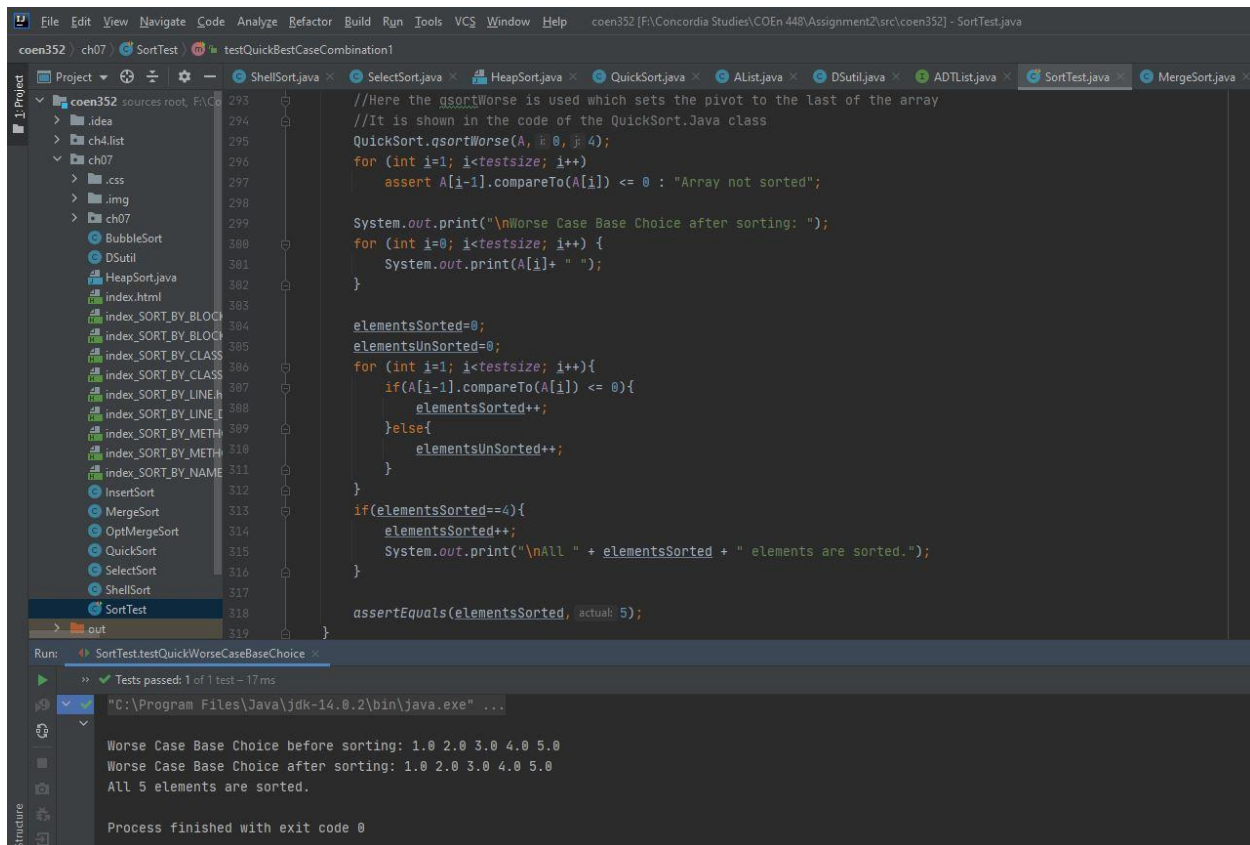
    System.out.print("Worse Case Base Choice before sorting: ");
    for (int i=0; i<testsize; i++) {
        System.out.print(A[i]+ " ");
    }

    for (int i=1; i<testsize; i++){
        if(A[i-1].compareTo(A[i]) <= 0){
            elementsSorted++;
        }else{
            elementsUnSorted++;
        }
    }

    if(elementsSorted==4){
        elementsSorted++;
    }
    if(elementsUnSorted>0) {
        elementsUnSorted += 1;
    }

    assertEquals(elementsUnSorted, actual: 0);

    //Here the qsortWorse is used which sets the pivot to the last of the array
    //It is shown in the code of the QuickSort.java class
    QuickSort.qsortWorse(A, i: 0, j: 4);
}
```



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 $\frac{1}{4}$ 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.

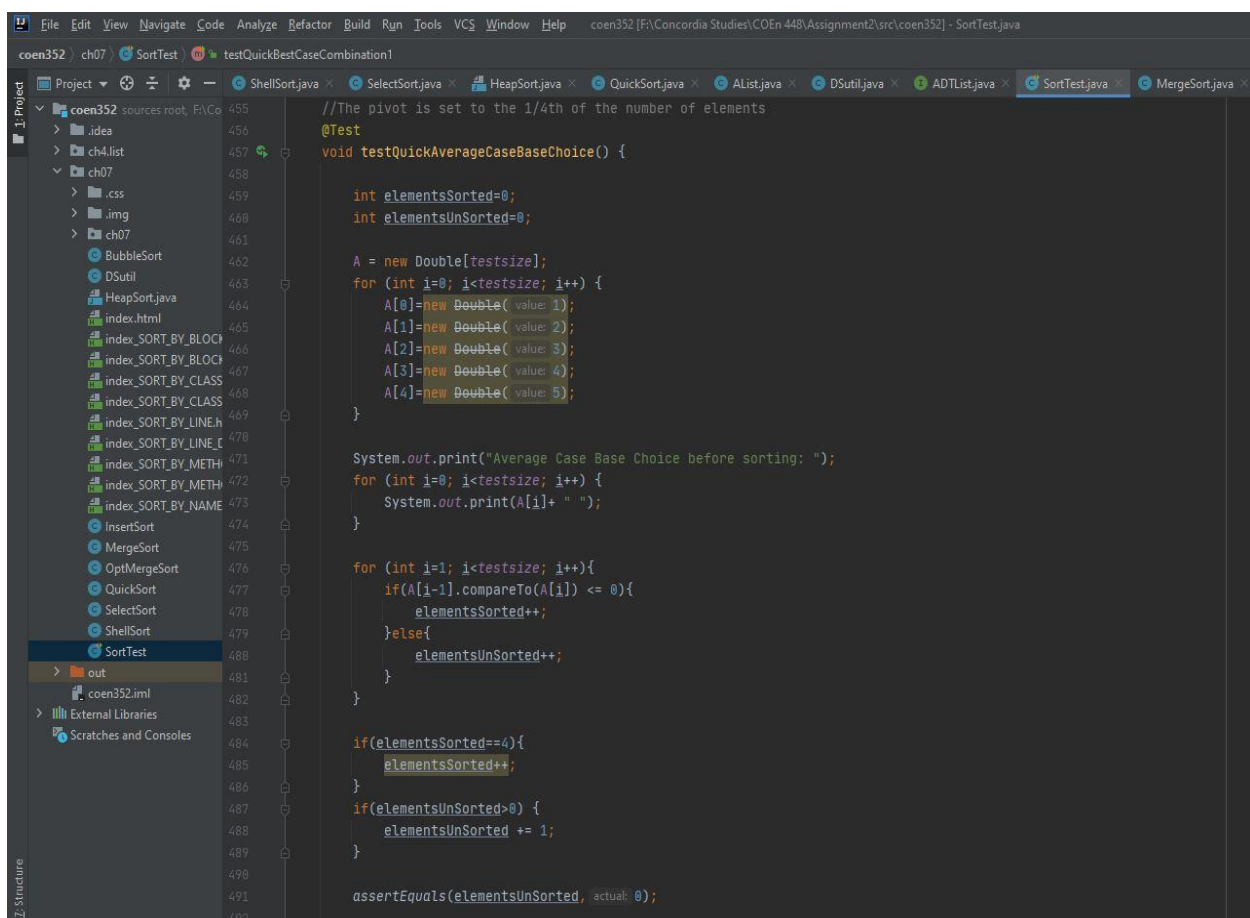
My base choice here would be AverageB1A and AverageB1B.

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.



```
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help coen352 [F:\Concordia Studies\COEn 448\Assignment2\src\coen352] - SortTest.java
coen352 ch07 SortTest testQuickBestCaseCombination1
Project
  coen352 sources root, F:\Co
    .idea
    ch4.list
    ch07
      .css
      .img
      ch07
        BubbleSort
        DSutil
        HeapSort.java
        index.html
        index_SORT_BY_BLOC
        index_SORT_BY_BLOC
        index_SORT_BY_CLASS
        index_SORT_BY_CLASS
        index_SORT_BY_LINE.h
        index_SORT_BY_LINE.h
        index_SORT_BY METH
        index_SORT_BY METH
        index_SORT_BY NAME
        InsertSort
        MergeSort
        OptMergeSort
        QuickSort
        SelectSort
        ShellSort
        SortTest
        out
        coen352.iml
    External Libraries
    Scratches and Consoles
Z-Structure
455 //The pivot is set to the 1/4th of the number of elements
456 @Test
457 void testQuickAverageCaseBaseChoice() {
458
459     int elementsSorted=0;
460     int elementsUnSorted=0;
461
462     A = new Double[testsize];
463     for (int i=0; i<testsize; i++) {
464         A[0]=new Double( value: 1);
465         A[1]=new Double( value: 2);
466         A[2]=new Double( value: 3);
467         A[3]=new Double( value: 4);
468         A[4]=new Double( value: 5);
469     }
470
471     System.out.print("Average Case Base Choice before sorting: ");
472     for (int i=0; i<testsize; i++) {
473         System.out.print(A[i]+ " ");
474     }
475
476     for (int i=1; i<testsize; i++){
477         if(A[i-1].compareTo(A[i]) <= 0){
478             elementsSorted++;
479         }else{
480             elementsUnSorted++;
481         }
482     }
483
484     if(elementsSorted==4){
485         elementsSorted++;
486     }
487     if(elementsUnSorted>0) {
488         elementsUnSorted += 1;
489     }
490
491     assertEquals(elementsUnSorted, actual: 0);
492 }
```

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help coen352 [F:\Concordia Studies\COEn 448\Assignment2\src\coen352] - SortTest.java

coen352 ch07 SortTest testQuickBestCaseCombination1

Project coen352 sources root, F:\Co
> .idea
> ch4.list
> ch07
> .css
> .img
> ch07
BubbleSort
DSutil
HeapSort.java
index.html
index_SORT_BY_BLOCK
index_SORT_BY_BLOCK
index_SORT_BY_CLASS
index_SORT_BY_CLASS
index_SORT_BY_LINE
index_SORT_BY_LINE
index_SORT_BY_METHOD
index_SORT_BY_METHOD
index_SORT_BY_NAME
InsertSort
MergeSort
OptMergeSort
QuickSort
SelectSort
ShellSort
SortTest
out
coen352.iml
External Libraries
Scratches and Consoles

```
490 assertEquals(elementsUnSorted, actual 0);  
491  
492  
493 //Here the gsortAverage is used which sets the pivot to the 1/4th of the array  
494 //It is shown in the code of the QuickSort.java class  
495 QuickSort.qsortAverage(A, 0, 4);  
496 for (int i=1; i<testsize; i++)  
497     assert A[i-1].compareTo(A[i]) <= 0 : "Array not sorted";  
498  
499  
500 System.out.print("\nAverage Case Base Choice after sorting: ");  
501 for (int i=0; i<testsize; i++) {  
502     System.out.print(A[i]+ " ");  
503 }  
504  
505 elementsSorted=0;  
506 elementsUnSorted=0;  
507 for (int i=1; i<testsize; i++){  
508     if(A[i-1].compareTo(A[i]) <= 0){  
509         elementsSorted++;  
510     }else{  
511         elementsUnSorted++;  
512     }  
513 }  
514 if(elementsSorted==4){  
515     elementsSorted++;  
516     System.out.println("\nAll " + elementsSorted + " elements are sorted.");  
517 }  
518  
519 assertEquals(elementsSorted, actual 5);  
520 }
```

Run: SortTest.testQuickAverageCaseBaseChoice

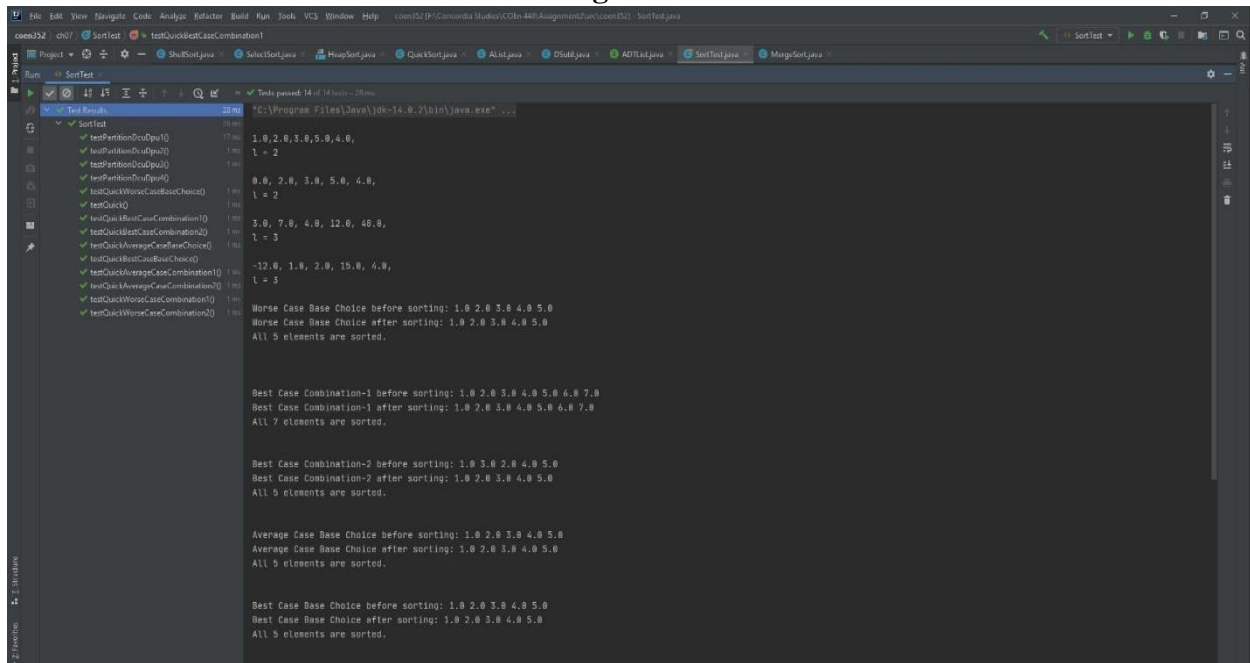
Tests passed: 1 of 1 test - 17 ms

"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:



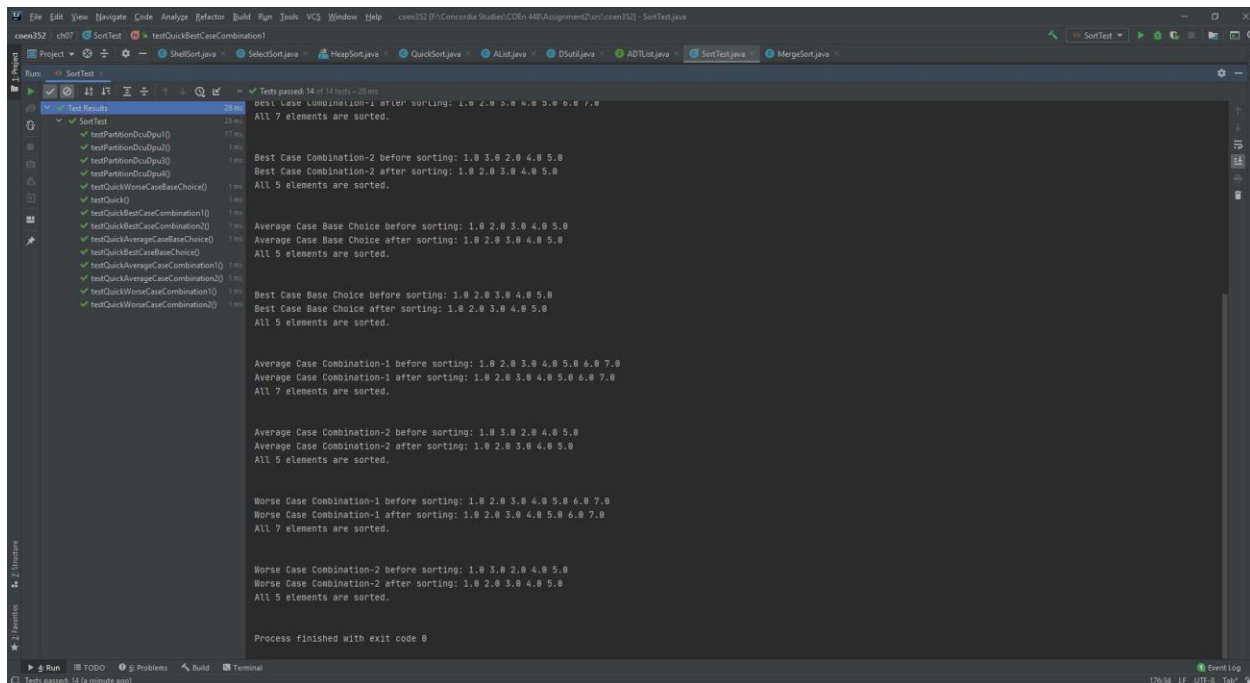
```
com132 ch07 SortTest testQuickSortCaseCombination1
Project SortTest
Run TestResults 20ms Tests passed: 14 of 14 tests - 20ms
SortTest
testPartitionQuick() 17ms 1.0 2.0 3.0 5.0 4.0,
testPartitionQuick() 1ms 1 = 2
testPartitionQuick() 1ms
testPartitionQuick() 1ms 0.0 2.0 3.0 5.0 4.0,
testQuickWorstCaseBaseChoice() 1ms 1 = 2
testQuick() 1ms
testQuickMergeCaseCombination1() 1ms 3.0 7.0 4.0 12.0 40.0,
testQuickMergeCaseCombination2() 1ms 1 = 3
testQuickMergeCaseBaseChoice() 1ms
testQuickMergeCaseBaseChoice() 1ms
testQuickMergeCaseCombination1() 1ms -12.0 1.0 2.0 15.0 4.0,
testQuickMergeCaseCombination2() 1ms 1 = 5
testQuickMergeCaseCombination1() 1ms
testQuickMergeCaseCombination2() 1ms
Worse Case Base Choice before sorting: 1.0 2.0 3.0 4.0 5.0
Worse Case Base Choice after sorting: 1.0 2.0 3.0 4.0 5.0
All 5 elements are sorted.

Best Case Combination-1 before sorting: 1.0 2.0 3.0 4.0 5.0 6.0 7.0
Best Case Combination-1 after sorting: 1.0 2.0 3.0 4.0 5.0 6.0 7.0
All 7 elements are sorted.

Best Case Combination-2 before sorting: 1.0 3.0 2.0 4.0 5.0
Best Case Combination-2 after sorting: 1.0 2.0 3.0 4.0 5.0
All 5 elements are sorted.

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.

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



```
com132 ch07 SortTest testQuickSortCaseCombination1
Project SortTest
Run TestResults 20ms Tests passed: 14 of 14 tests - 20ms
SortTest
testPartitionQuick() 17ms 1.0 2.0 3.0 5.0 4.0 6.0 7.0
testPartitionQuick() 1ms All 7 elements are sorted.
testPartitionQuick() 1ms
testPartitionQuick() 1ms
Best Case Combination-2 before sorting: 1.0 3.0 2.0 4.0 5.0
Best Case Combination-2 after sorting: 1.0 2.0 3.0 4.0 5.0
All 5 elements are sorted.
testQuickWorstCaseBaseChoice() 1ms
testQuick() 1ms
testQuickMergeCaseCombination1() 1ms
testQuickMergeCaseCombination2() 1ms
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.
testQuickMergeCaseBaseChoice() 1ms
testQuickMergeCaseBaseChoice() 1ms
testQuickMergeCaseCombination1() 1ms
testQuickMergeCaseCombination2() 1ms
Best Case Base Choice before sorting: 1.0 2.0 3.0 4.0 5.0
Best Case Base Choice after sorting: 1.0 2.0 3.0 4.0 5.0
All 5 elements are sorted.

Average Case Combination-1 before sorting: 1.0 2.0 3.0 4.0 5.0 6.0 7.0
Average Case Combination-1 after sorting: 1.0 2.0 3.0 4.0 5.0 6.0 7.0
All 7 elements are sorted.

Average Case Combination-2 before sorting: 1.0 3.0 2.0 4.0 5.0
Average Case Combination-2 after sorting: 1.0 2.0 3.0 4.0 5.0
All 5 elements are sorted.

Worse Case Combination-1 before sorting: 1.0 2.0 3.0 4.0 5.0 6.0 7.0
Worse Case Combination-1 after sorting: 1.0 2.0 3.0 4.0 5.0 6.0 7.0
All 7 elements are sorted.

Worse Case Combination-2 before sorting: 1.0 3.0 2.0 4.0 5.0
Worse Case Combination-2 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 the coverages:

The screenshot shows an IDE with a Java project named 'con352'. The main editor displays the source code for 'SortTest.java', which implements a QuickSort algorithm. The code includes a 'main' method and several static methods for sorting: 'findPivot', 'findPivotWorst', 'findPivotAverageCase', 'swap', and 'partition'. The coverage window on the right shows the following data:

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)

Coverage: SortTest x

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)

Coverage Summary for Class: QuickSort (ch07)

```

1 package ch07;
2
3 public class Quicksort {
4
5     public static <E extends Comparable<E> super E> void sort(E[] A) {
6         qsort(A, 0, A.length-1);
7     }
8
9     static <E extends Comparable<E> super E>
10    void qsort(E[] A, int l, int j) { // Quicksort
11
12        int pivotIndex = findPivot(A, l, j); // Pick a pivot
13
14        Dutil.swap(A, pivotIndex, j); // Stick pivot at end
15
16        // k will be the first position in the right subarray
17
18        int k = partition(A, l+1, j, A[j]);
19
20        Dutil.swap(A, k, j); // Put pivot in place
21
22        if ((k-l) > 1) qsort(A, l, k-1); // Sort left partition
23        if ((j-k) > 1) qsort(A, k+1, j); // Sort right partition
24    }
25
26    static <E extends Comparable<E> super E>
27    void qsortAverageCase(E[] A, int l, int j) { // Quicksort
28
29        int pivotIndex = findPivotAverageCase(A, l, j); // Pick a pivot
30
31        Dutil.swap(A, pivotIndex, j); // Stick pivot at end
32
33        // k will be the first position in the right subarray
34
35        int k = partition(A, l+1, j, A[j]);
36
37        Dutil.swap(A, k, j); // Put pivot in place
38
39        if ((k-l) > 1) qsort(A, l, k-1); // Sort left partition
40        if ((j-k) > 1) qsort(A, k+1, j); // Sort right partition
41    }
42
43    static <E extends Comparable<E> super E>
44    void qsortWorstCase(E[] A, int l, int j) { // Quicksort
45
46        int pivotIndex = findPivotWorst(A, l, j); // Pick a pivot
47
48        Dutil.swap(A, pivotIndex, j); // Stick pivot at end
49        // k will be the first position in the right subarray
50
51        int k = partition(A, l+1, j, A[j]);
52
53        Dutil.swap(A, k, j); // Put pivot in place
54
55        if ((k-l) > 1) qsort(A, l, k-1); // Sort left partition
56        if ((j-k) > 1) qsort(A, k+1, j); // Sort right partition
57    }
58
59
60    static <E extends Comparable<E> super E>
61    int partition(E[] A, int l, int r, E pivot) {
62
63        do { // Move bounds inward until they meet
64
65            while (A[l+1].compareTo(pivot)<0);
66
67            while ((r!=0) && (A[r-1].compareTo(pivot)>0));
68
69            Dutil.swap(A, l, r); // Swap out-of-place values
70        } while (l < r); // Stop when they cross
71        Dutil.swap(A, l, r); // Reverse last, wasted swap
72        return l; // Return first position in right partition
73    }
74
75    static <E extends Comparable<E> super E>
76    int findPivot(E[] A, int l, int j)
77    { return (l+j)/2; }
78
79
80    static <E extends Comparable<E> super E>
81    int findPivotWorst(E[] A, int l, int j)
82    { return 0; }
83
84    static <E extends Comparable<E> super E>
85    int findPivotAverageCase(E[] A, int l, int j)
86    { return (l+j)/4; }
87
88
89 }
90
91

```

[\[all classes \]](#) [\[ch07 \]](#)

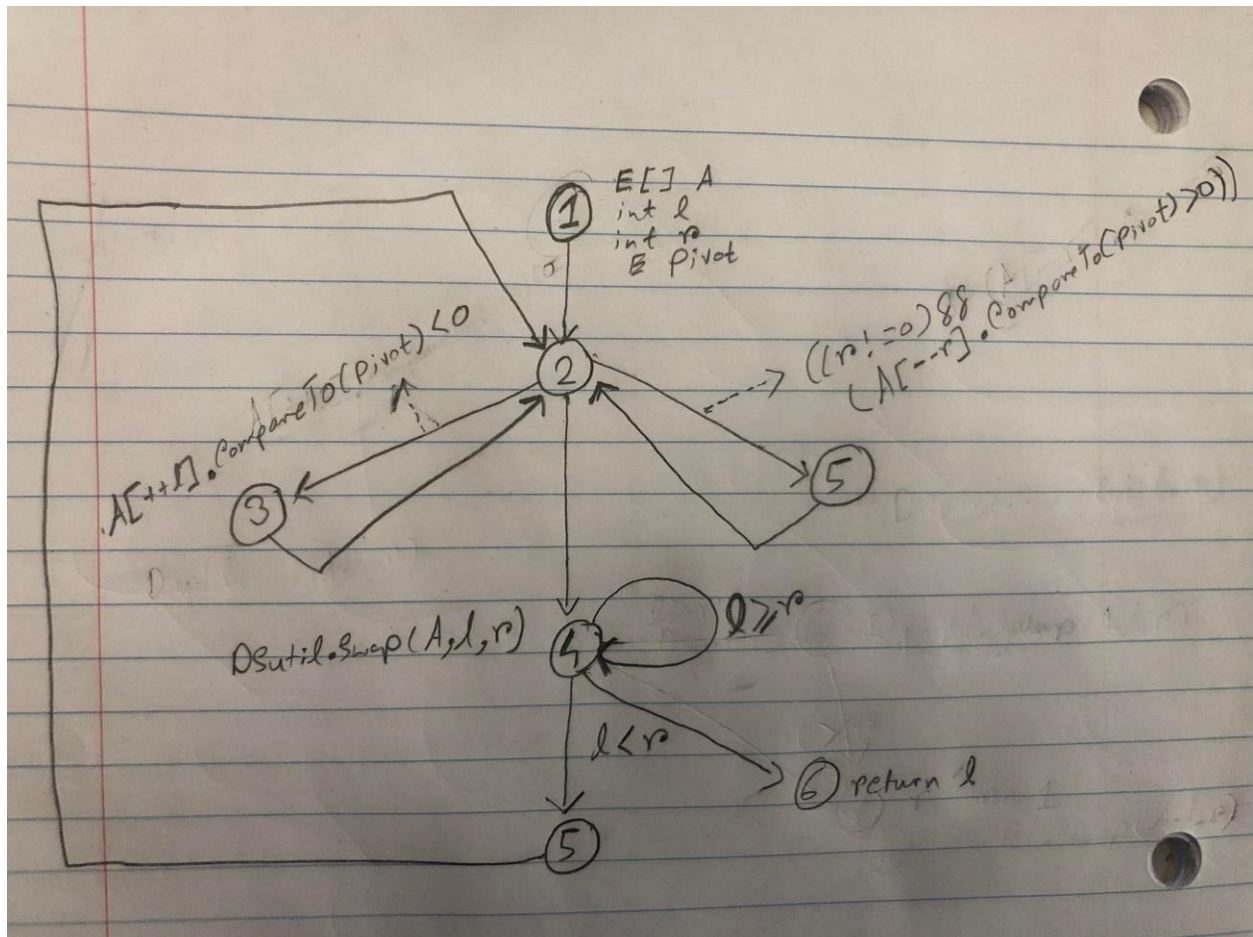
Class	Class, %	Method, %	Line, %
SortTest	100% (1/ 1)	100% (17/ 17)	94.3% (396/ 420)

```

1 package cn.hutool;
2 import static org.junit.jupiter.api.Assertions.*;
3
4 import org.junit.jupiter.api.BeforeEach;
5 import org.junit.jupiter.api.Test;
6
7 import cn.hutool.*;
8
9
10 public class SortTest {
11
12     @final static int testSize = 5;
13     Double A[];
14     static int THRESHOLD = 8;
15
16     private static ArrayList<Integer> list;
17
18     @SuppressWarnings("remove")
19     @BeforeEach
20     void setUp() throws Exception {
21         // A = new Double[testSize];
22         for (int i=0; i<testSize; i++) {
23             //A[i] = new Double(DoubleUtil.random(3)); // return int basic type, new Integer from int.
24         }
25         //list = new ArrayList<Integer>();
26         list = new ArrayList<Integer>();
27         // list = new ArrayList<Integer>();
28     }
29
30     @Test
31     void testQuick() {
32         A = new Double[testSize];
33         for (int i=0; i<testSize; i++) {
34             //A[i] = new Double(DoubleUtil.random(3)); // return int basic type, new Integer from int.
35         }
36         A[0]=new Double(5);
37         A[1]=new Double(3);
38         A[2]=new Double(2);
39         A[3]=new Double(1);
40         A[4]=new Double(4);
41
42         QuickSort.sort(A);
43         for (int i=1; i<testSize; i++)
44             assert A[i-1].compareTo(A[i]) <= 0 : "Array not sorted";
45         for (int i=0; i<testSize; i++) {
46             System.out.print(A[i]+ " ");
47         }
48         // The Pivot is set to the median element in the Array
49         @Test
50         void testQuickBestCaseBaseChoice() {
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

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 dpv.

Test1 dcu and dpv all nodes

Input data <“A”, “I”, “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.

```
55
56
57 @Test
58 void testPartitionDcuDpu1() {
59
60     A = new Double[testsSize];
61     for (int i=0; i<testsSize; i++) {
62         //A[i] = new Double(DSutil.random(3)); // return int basic type, new Integer from int.
63
64         A[0]=new Double( value: 5);
65         A[1]=new Double( value: 3);
66         A[2]=new Double( value: 2);
67         A[3]=new Double( value: 1);
68         A[4]=new Double( value: 4);
69
70     }
71
72     int temp=QuickSort.partition(A, -1, 5, A[2]);
73
74     String str="";
75
76     for (int i=0; i<testsSize; i++) {
77         System.out.print(A[i]+"," );
78     }
79
80     assertEquals(temp, actual: 2);
81     System.out.println();
82     System.out.print("l = " + temp);
83 }
84
85
```

Run: SortTest.testPartitionDcuDpu1 x

Tests passed: 1 of 1 test - 17 ms

Test Results

- SortTest
- testPartitionDcuDpu1

17 ms

17 ms

17 ms

1.0,2.0,3.0,5.0,4.0,
l = 2

Process finished with exit code 0

Tests passed: 1

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.

The screenshot shows an IDE with a project structure on the left and a code editor in the center. The project structure includes folders 'ch4.list' and 'ch07', with various Java files like 'ADTList', 'AList', 'DLink', 'DList', 'Link', 'ListJUnitTest', 'ListTestRunner', 'LList', 'BubbleSort', 'DSutil', 'HeapSort.java', 'InsertSort', 'MergeSort', 'OptMergeSort', 'QuickSort', 'SelectSort', 'ShellSort', and 'SortTest'. The 'SortTest' file is selected, and its content is displayed in the editor. The code defines a test method 'testPartitionDcuDpu2()' that initializes an array 'A' with values [0, 3, 2, 5, 4], calls 'QuickSort.partition(A, -1, 5, A[1])', and prints the result 'l'. The output of the test is shown in the 'Run' window at the bottom, indicating that the test passed and the output is 'l = 2'.

```
84 }
85
86 @Test
87 void testPartitionDcuDpu2() {
88
89     A = new Double[testsize];
90     for (int i=0; i<testsize; i++) {
91         //A[i] = new Double(DSutil.random(3)); // return int basic type, new Integer from int.
92
93         A[0]=new Double( value: 0);
94         A[1]=new Double( value: 3);
95         A[2]=new Double( value: 2);
96         A[3]=new Double( value: 5);
97         A[4]=new Double( value: 4);
98     }
99
100     int temp=QuickSort.partition(A, -1, 5, A[1]);
101
102     for (int i=0; i<testsize; i++) {
103         System.out.print(A[i]+ " ");
104     }
105
106     assertEquals(temp, actual: 2);
107     System.out.println();
108     System.out.print("l = " + temp);
109 }
110
111
112
113
```

Run: SortTest.testPartitionDcuDpu2 x

Tests passed: 1 of 1 test - 16 ms

Test Results

Test	Time	Output
SortTest	16 ms	
testPartitionDcuDpu2()	16 ms	0.0, 2.0, 3.0, 5.0, 4.0, l = 2

Process finished with exit code 0

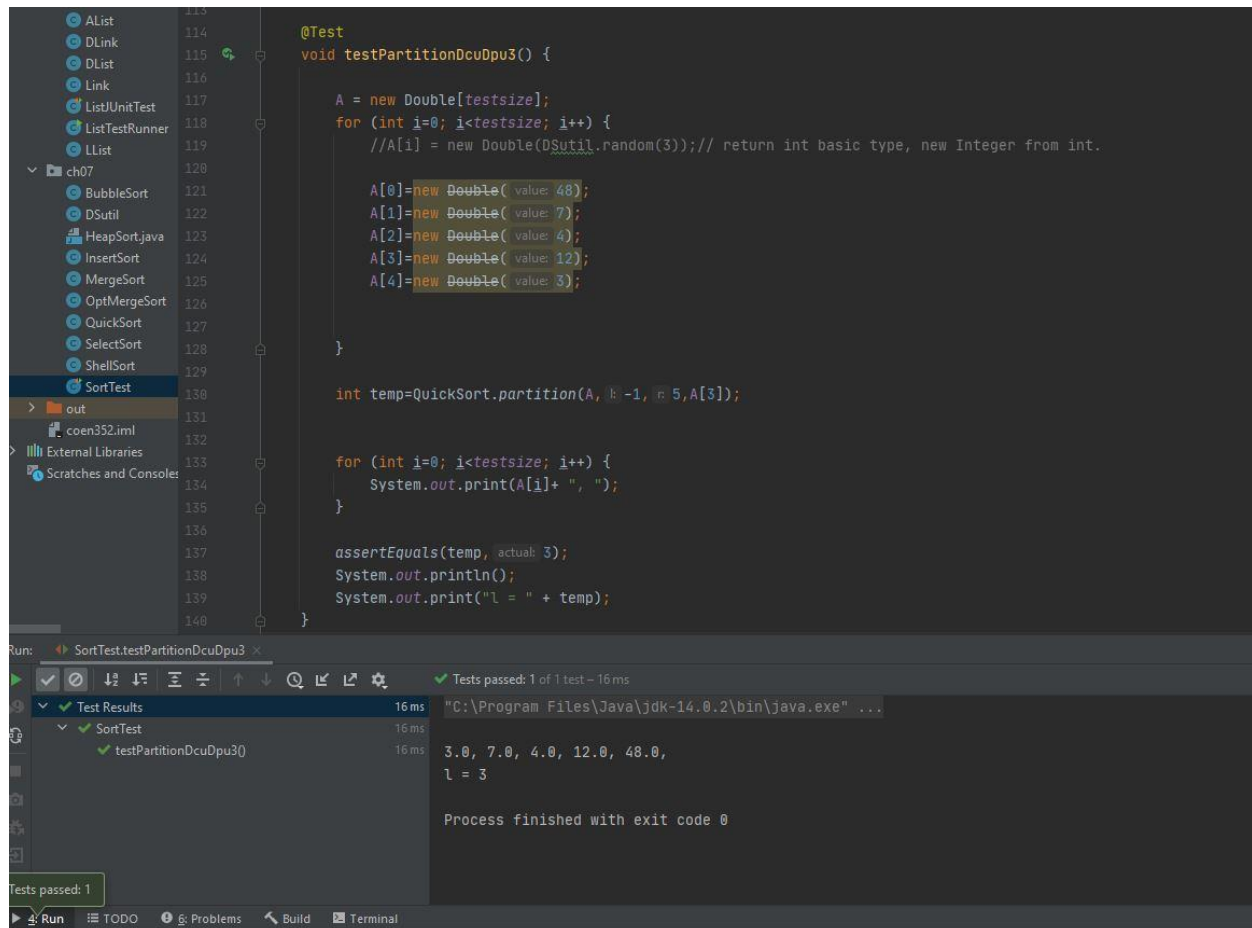
Tests passed: 1 (a minute ago)

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.



```
113
114
115 @Test
116 void testPartitionDcuDpu3() {
117
118     A = new Double[testsize];
119     for (int i=0; i<testsize; i++) {
120         //A[i] = new Double(DSutil.random(3)); // return int basic type, new Integer from int.
121
122         A[0]=new Double( value: 48);
123         A[1]=new Double( value: 7);
124         A[2]=new Double( value: 4);
125         A[3]=new Double( value: 12);
126         A[4]=new Double( value: 3);
127     }
128
129     int temp=QuickSort.partition(A, -1, 5, A[3]);
130
131     for (int i=0; i<testsize; i++) {
132         System.out.print(A[i]+ " ");
133     }
134
135     assertEquals(temp, actual: 3);
136     System.out.println();
137     System.out.print("l = " + temp);
138 }
139
140
```

Run: SortTest.testPartitionDcuDpu3 x

Tests passed: 1 of 1 test - 16 ms

Test Results 16 ms

SortTest 16 ms

testPartitionDcuDpu3() 16 ms

3.0, 7.0, 4.0, 12.0, 48.0,
l = 3

Process finished with exit code 0

Tests passed: 1

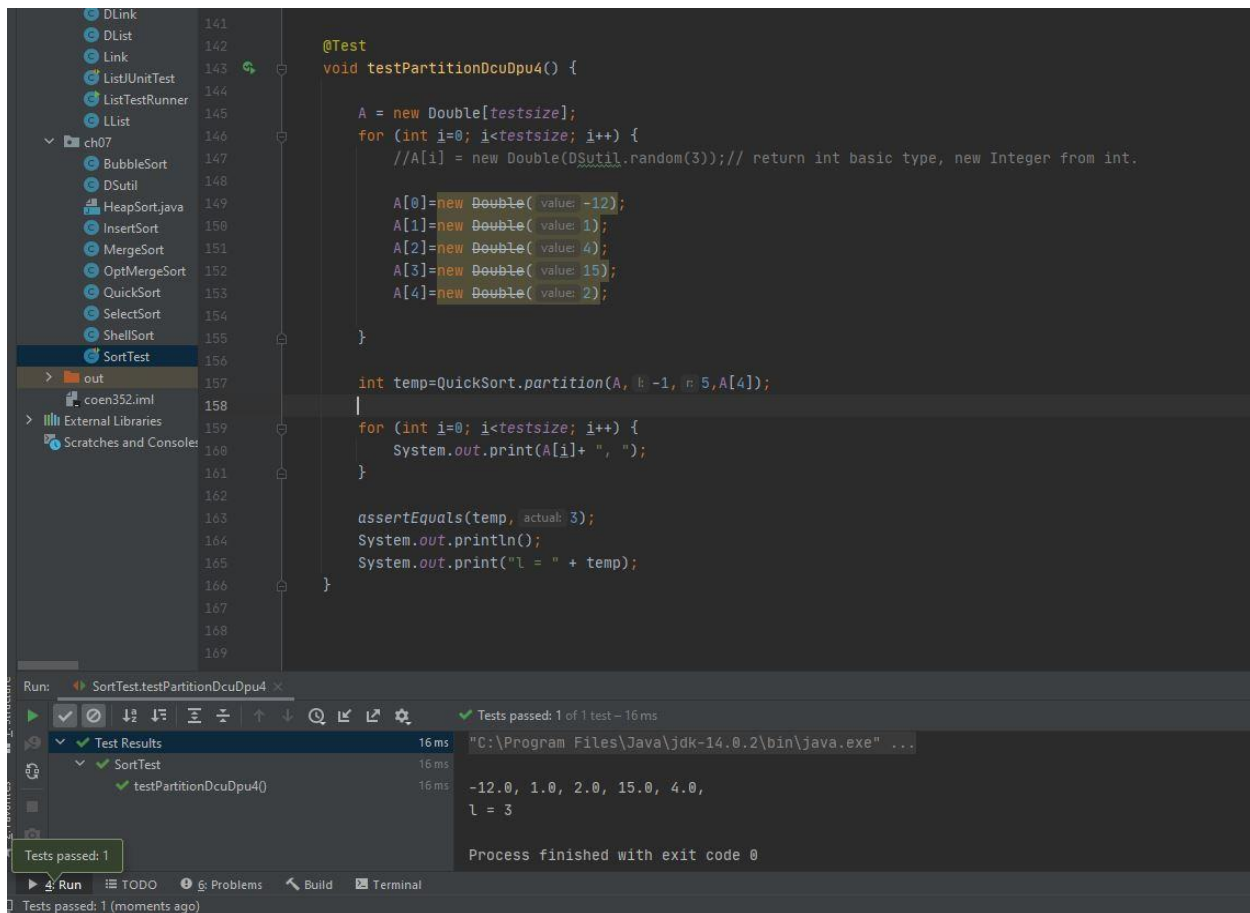
Run TODO Problems Build Terminal

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.



```
@Test
void testPartitionDcuDpu4() {
    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( value: -12);
        A[1]=new Double( value: 1);
        A[2]=new Double( value: 4);
        A[3]=new Double( value: 15);
        A[4]=new Double( value: 2);
    }

    int temp=QuickSort.partition(A, l: -1, r: 5,A[4]);

    for (int i=0; i<testsize; i++) {
        System.out.print(A[i]+ " ");
    }

    assertEquals(temp, actual: 3);
    System.out.println();
    System.out.print("l = " + temp);
}
```

Run: SortTest.testPartitionDcuDpu4

Tests passed: 1 of 1 test - 16 ms

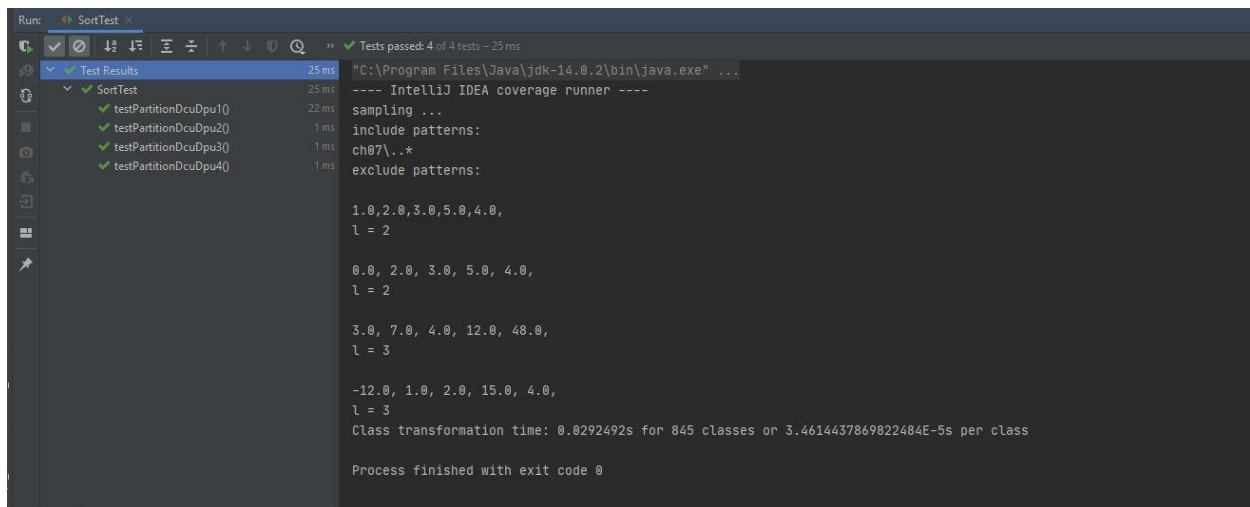
Test Results	Time	Path
SortTest	16 ms	"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" ...
testPartitionDcuDpu4()	16 ms	-12.0, 1.0, 2.0, 15.0, 4.0, l = 3

Tests passed: 1

Process finished with exit code 0

(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:

Coverage: SortTest x			
33% classes, 12% 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)	12% (1/8)	18% (6/32)
SelectSort	0% (0/1)	0% (0/1)	0% (0/8)
ShellSort	0% (0/1)	0% (0/2)	0% (0/9)
SortTest	100% (1/1)	37% (6/16)	14% (61/420)

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)

```
1 package ch07;
2
3 public class QuickSort {
4     public static <E extends Comparable<E> super E>> void sort(E[] A) {
5         qsort(A, 0, A.length-1);
6     }
7
8     static <E extends Comparable<E> super E>>
9     void qsort(E[] A, int l, int r) { // Quicksort
10
11         int pivotIndex = findPivot(A, l, r); // Pick a pivot
12
13         Dutil.swap(A, pivotIndex, r); // Stick pivot at end
14
15         // k will be the first position in the right subarray
16
17         int k = partition(A, l-1, r, A[pivotIndex]);
18
19         Dutil.swap(A, k, r); // Put pivot in place
20
21         if ((k-l) > 1) qsor(A, l, k-1); // Sort left partition
22         if ((r-k) > 1) qsor(A, k+1, r); // Sort right partition
23     }
24
25     static <E extends Comparable<E> super E>>
26     void qsor(E[] A, int l, int r) { // Quicksort
27
28         int pivotIndex = findPivot(A, l, r); // Pick a pivot
29
30         Dutil.swap(A, pivotIndex, r); // Stick pivot at end
31
32         // k will be the first position in the right subarray
33
34         int k = partition(A, l-1, r, A[pivotIndex]);
35
36         Dutil.swap(A, k, r); // Put pivot in place
37
38         if ((k-l) > 1) qsor(A, l, k-1); // Sort left partition
39         if ((r-k) > 1) qsor(A, k+1, r); // Sort right partition
40     }
41
42     static <E extends Comparable<E> super E>>
43     void qsor(E[] A, int l, int r) { // Quicksort
44
45         int pivotIndex = findPivot(A, l, r); // Pick a pivot
46
47         Dutil.swap(A, pivotIndex, r); // Stick pivot at end
48         // k will be the first position in the right subarray
49
50         int k = partition(A, l-1, r, A[pivotIndex]);
51
52         Dutil.swap(A, k, r); // Put pivot in place
53
54         if ((k-l) > 1) qsor(A, l, k-1); // Sort left partition
55         if ((r-k) > 1) qsor(A, k+1, r); // Sort right partition
56     }
57
58
59     static <E extends Comparable<E> super E>>
60     int partition(E[] A, int l, int r, E pivot) {
61
62         do { // Move bounds inward until they meet
63
64             while (A[l+1].compareTo(pivot) < 0)
65                 l++;
66             while ((r!=0) && (A[r].compareTo(pivot) > 0))
67                 r--;
68             Dutil.swap(A, l, r); // Swap out-of-place values
69         } while (l < r); // Stop when they cross
70         Dutil.swap(A, l, r); // Reverse last, wasted swap
71         return l; // Return first position in right partition
72     }
73
74     static <E extends Comparable<E> super E>>
75     int findPivot(E[] A, int l, int r) {
76         return (l+r)/2;
77     }
78
79     static <E extends Comparable<E> super E>>
80     int findPivotFirst(E[] A, int l, int r) {
81         return l;
82     }
83
84     static <E extends Comparable<E> super E>>
85     int findPivotAverageCase(E[] A, int l, int r) {
86         return (l+r)/4;
87     }
88
89 }
90
91
```

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 qsor method it covers 88.9% of the methods and 97% of the lines as the qsor 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 qsor(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 qsor which is called in the qsor method. Hence when we do the partition the %coverage is very less compared to the qsor.

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.