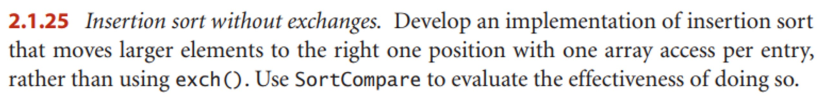
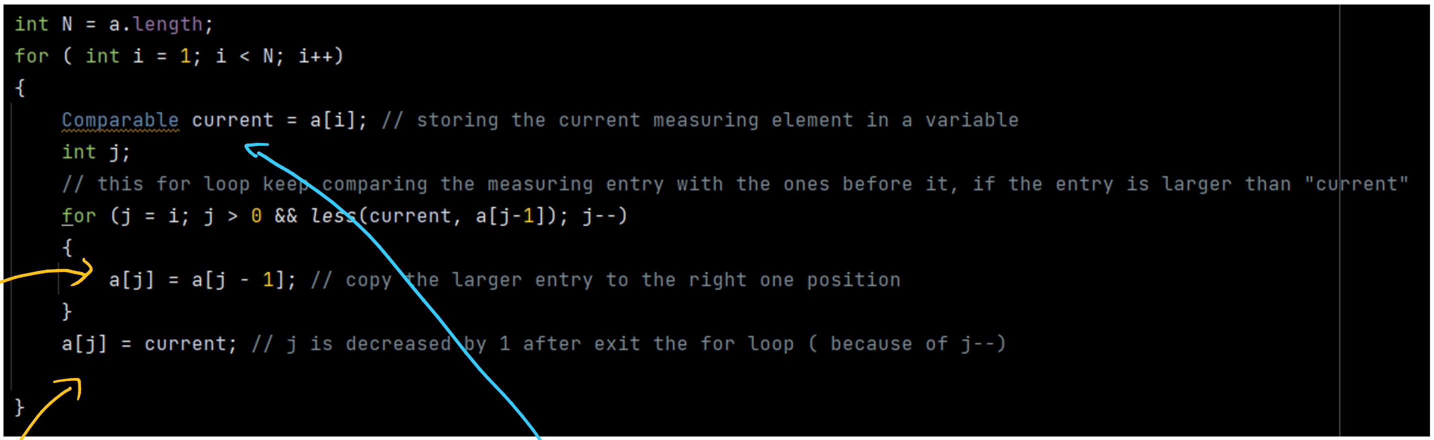
2.1.25

2022年11月11日 18:25



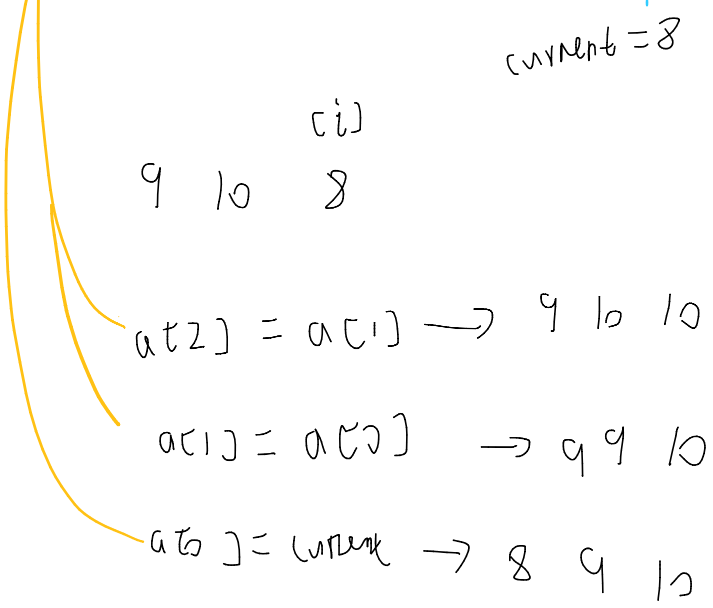
2.1.25 *Insertion sort without exchanges.* Develop an implementation of insertion sort that moves larger elements to the right one position with one array access per entry, rather than using `exch()`. Use `SortCompare` to evaluate the effectiveness of doing so.

There is no need to swap the two entries when the left entry is greater than the right entry. Below is the advanced version of insertion sort, which it does not exchanges two items.



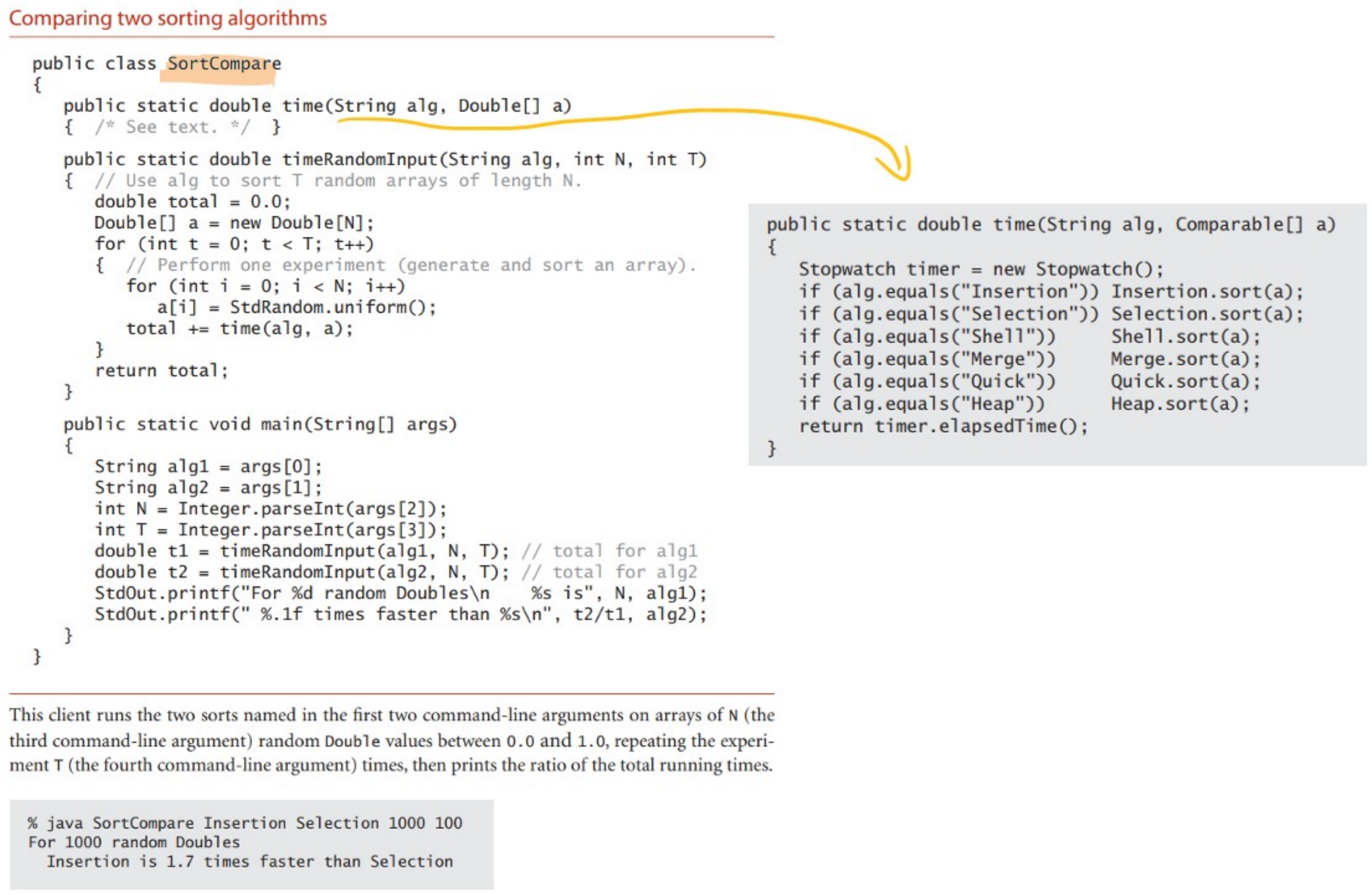
```
int N = a.length;
for ( int i = 1; i < N; i++)
{
    Comparable current = a[i]; // storing the current measuring element in a variable
    int j;
    // this for loop keep comparing the measuring entry with the ones before it, if the entry is larger than "current"
    for (j = i; j > 0 && less(current, a[j-1]); j--)
    {
        a[j] = a[j - 1]; // copy the larger entry to the right one position
    }
    a[j] = current; // j is decreased by 1 after exit the for loop ( because of j--)

}
```

Visualize the process of the above sorting algorithm. Lets say there is an array of three elements: 9 10 8, and I'm looking at the third element which is the number 8.



The default insertion sort comes with the exch() which is used to swap two adjecent entries, until the entry is being swapped to the right position. This algorithm DO NOT use the swapping method to insert the entry to its proper position. Instead, it compares the examining element with the entries before it one by one consecutively. If the previous entry is greater than that element, copying the previous entry to the right one position, then comparing the next previous entry and repeat the above action until encounter a previous entry that is smaller or equal to the examining element, and at this point, place the examining element before that previous entry.

Unlike the default insertion sort, this algorithm keep shifting the entries before the current examining element one position to the right. When the proper position is found, stop the shfiting behavior and insert the element to that proper position.



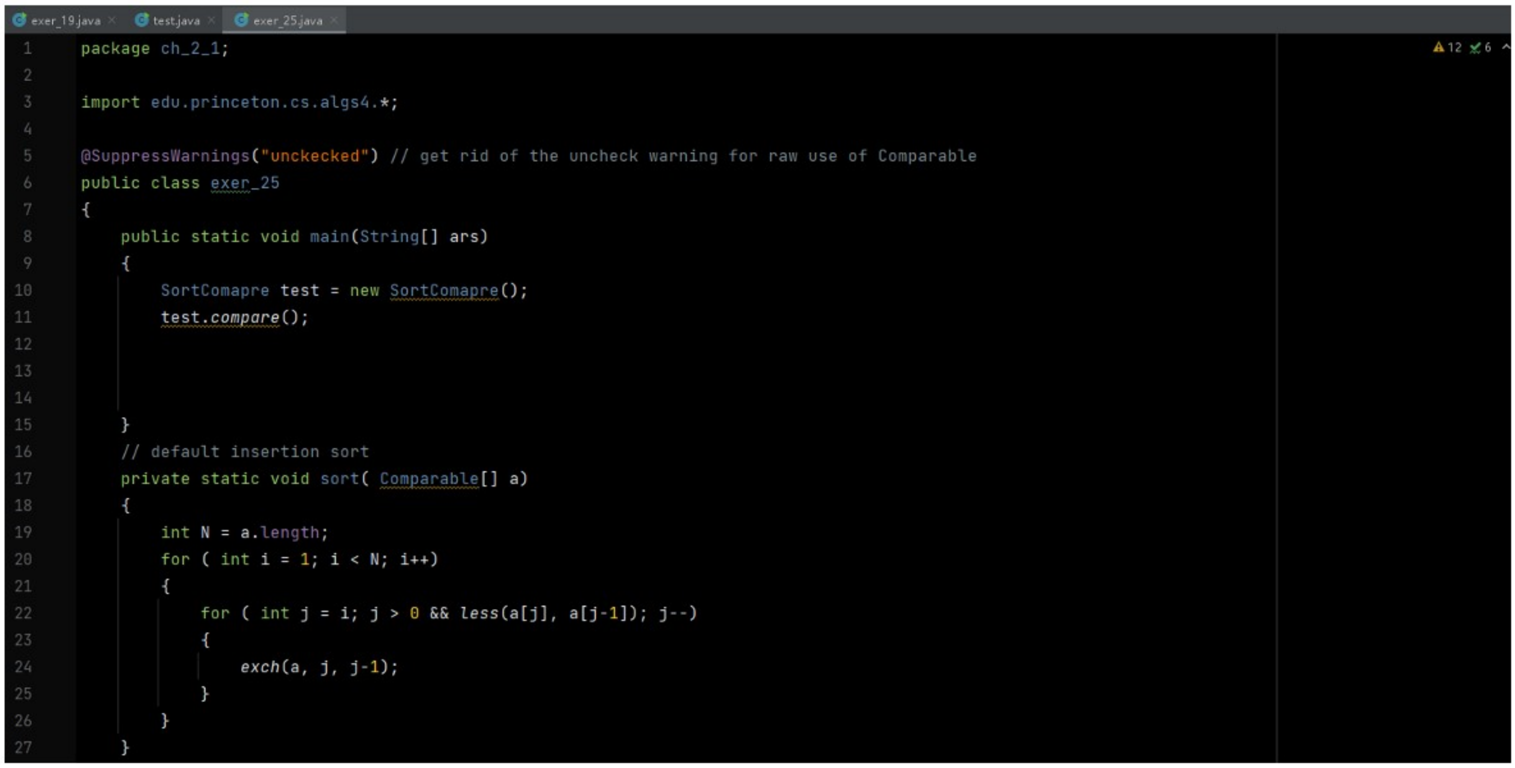
Comparing two sorting algorithms

```
public class SortCompare
{
   public static double time(String alg, Double[] a)
   {  /* See text. */  }

   public static double timeRandomInput(String alg, int N, int T)
   {  // Use alg to sort T random arrays of length N.
      double total = 0.0;
      Double[] a = new Double[N];
      for (int t = 0; t < T; t++)
      {  // Perform one experiment (generate and sort an array).
         for (int i = 0; i < N; i++)
            a[i] = StdRandom.uniform();
         total += time(alg, a);
      }
      return total;
   }

   public static void main(String[] args)
   {
      String alg1 = args[0];
      String alg2 = args[1];
      int N = Integer.parseInt(args[2]);
      int T = Integer.parseInt(args[3]);
      double t1 = timeRandomInput(alg1, N, T); // total for alg1
      double t2 = timeRandomInput(alg2, N, T); // total for alg2
      StdOut.printf("For %d random Doubles\n    %s is", N, alg1);
      StdOut.printf(" %.1f times faster than %s\n", t2/t1, alg2);
   }
}
```

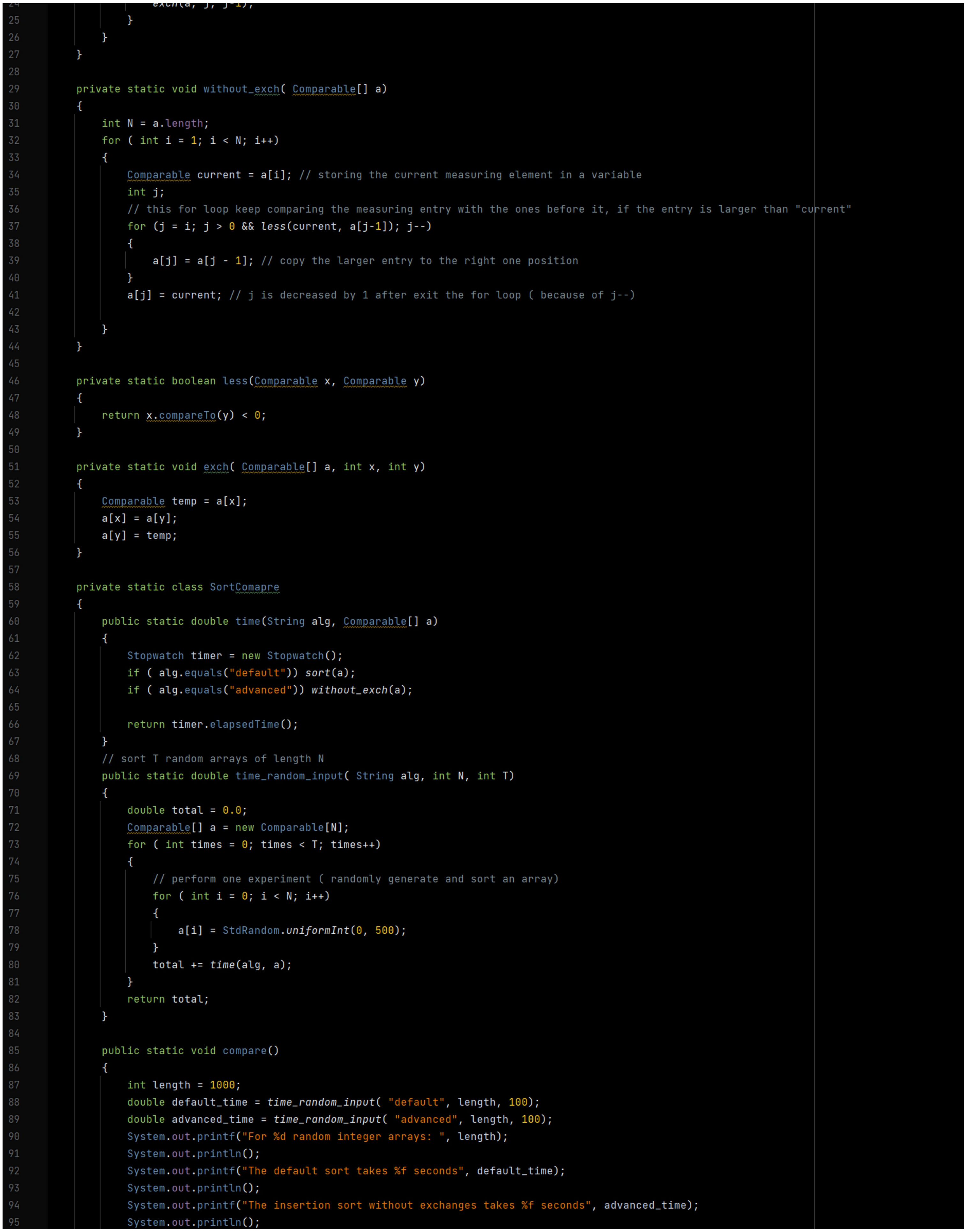
```
public static double time(String alg, Comparable[] a)
{
   Stopwatch timer = new Stopwatch();
   if (alg.equals("Insertion")) Insertion.sort(a);
   if (alg.equals("Selection")) Selection.sort(a);
   if (alg.equals("Shell"))     Shell.sort(a);
   if (alg.equals("Merge"))     Merge.sort(a);
   if (alg.equals("Quick"))     Quick.sort(a);
   if (alg.equals("Heap"))      Heap.sort(a);
   return timer.elapsedTime();
}
```

This client runs the two sorts named in the first two command-line arguments on arrays of N (the third command-line argument) random Double values between 0.0 and 1.0, repeating the experiment T (the fourth command-line argument) times, then prints the ratio of the total running times.

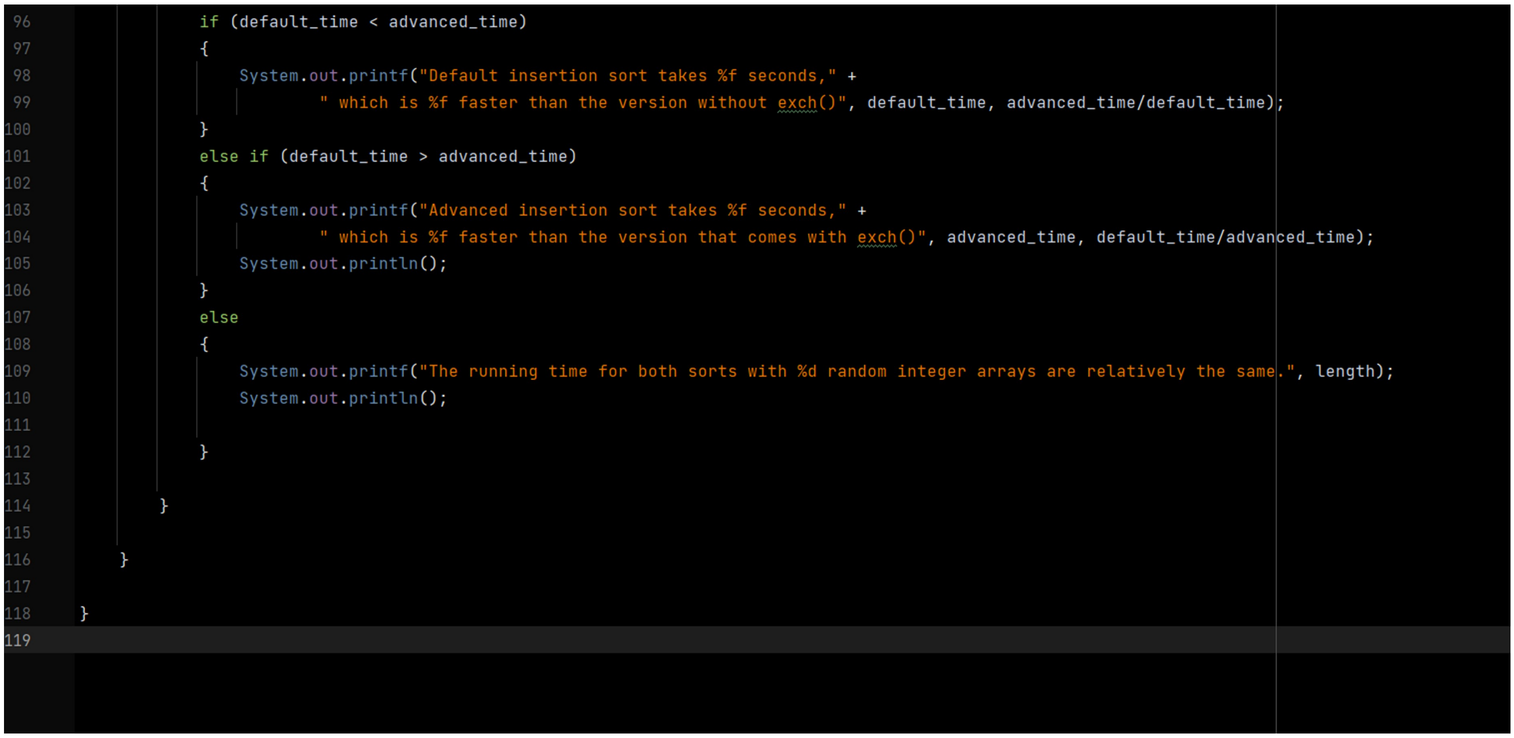
```
% java SortCompare Insertion Selection 1000 100
For 1000 random Doubles
  Insertion is 1.7 times faster than Selection
```



```
package ch_2_1;

import edu.princeton.cs.algs4.*;

@SuppressWarnings("unckecked") // get rid of the uncheck warning for raw use of Comparable
public class exer_25
{
    public static void main(String[] ars)
    {
        SortComapre test = new SortComapre();
        test.compare();



    }
    // default insertion sort
    private static void sort( Comparable[] a)
    {
        int N = a.length;
        for ( int i = 1; i < N; i++)
        {
            for ( int j = i; j > 0 && less(a[j], a[j-1]); j--)
            {
                exch(a, j, j-1);
            }
        }
    }
```



```java
                exch(a, j, j-1);
            }
        }
    }

    private static void without_exch( Comparable[] a)
    {
        int N = a.length;
        for ( int i = 1; i < N; i++)
        {
            Comparable current = a[i]; // storing the current measuring element in a variable
            int j;
            // this for loop keep comparing the measuring entry with the ones before it, if the entry is larger than "current"
            for (j = i; j > 0 && less(current, a[j-1]); j--)
            {
                a[j] = a[j - 1]; // copy the larger entry to the right one position
            }
            a[j] = current; // j is decreased by 1 after exit the for loop ( because of j--)

        }
    }

    private static boolean less(Comparable x, Comparable y)
    {
        return x.compareTo(y) < 0;
    }

    private static void exch( Comparable[] a, int x, int y)
    {
        Comparable temp = a[x];
        a[x] = a[y];
        a[y] = temp;
    }

    private static class SortComapre
    {
        public static double time(String alg, Comparable[] a)
        {
            Stopwatch timer = new Stopwatch();
            if ( alg.equals("default")) sort(a);
            if ( alg.equals("advanced")) without_exch(a);

            return timer.elapsedTime();
        }
        // sort T random arrays of length N
        public static double time_random_input( String alg, int N, int T)
        {
            double total = 0.0;
            Comparable[] a = new Comparable[N];
            for ( int times = 0; times < T; times++)
            {
                // perform one experiment ( randomly generate and sort an array)
                for ( int i = 0; i < N; i++)
                {
                    a[i] = StdRandom.uniformInt(0, 500);
                }
                total += time(alg, a);
            }
            return total;
        }

        public static void compare()
        {
            int length = 1000;
            double default_time = time_random_input( "default", length, 100);
            double advanced_time = time_random_input( "advanced", length, 100);
            System.out.printf("For %d random integer arrays: ", length);
            System.out.println();
            System.out.printf("The default sort takes %f seconds", default_time);
            System.out.println();
            System.out.printf("The insertion sort without exchanges takes %f seconds", advanced_time);
            System.out.println();
```



```java
            if (default_time < advanced_time)
            {
                System.out.printf("Default insertion sort takes %f seconds," +
                        " which is %f faster than the version without exch()", default_time, advanced_time/default_time);
            }
            else if (default_time > advanced_time)
            {
                System.out.printf("Advanced insertion sort takes %f seconds," +
                        " which is %f faster than the version that comes with exch()", advanced_time, default_time/advanced_time);
                System.out.println();
            }
            else
            {
                System.out.printf("The running time for both sorts with %d random integer arrays are relatively the same.", length);
                System.out.println();

            }

        }

    }

}
```