|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | n = 5 | n = 10 | n = 50 | n = 100 | n = 500 | n = 1000 | n = 10000 |
| Selection | 0 | 0 | 1 | 0 | 37 | 77 | 661 |
| Insertion | 0 | 0 | 0 | 1 | 17 | 121 | 670 |
| Merge Sort | 0 | 0 | 0 | 1 | 1 | 0 | 38 |
| Quicksort | 0 | 0 | 0 | 5 | 5 | 13 | 298 |

Table : Running Times of Unsorted Random Array

Figure : Running Times for Unsorted Random Array

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | n = 5 | n = 10 | n = 50 | n = 100 | n = 500 | n = 1000 | n = 10000 |
| Selection | 0 | 0 | 1 | 0 | 14 | 6 | 686 |
| Insertion | 0 | 1 | 0 | 0 | 1 | 1 | 3 |
| Merge Sort | 0 | 0 | 0 | 0 | 0 | 3 | 35 |
| Quicksort | 0 | 0 | 1 | 0 | 2 | 8 | 63 |

Table : Running Times of Sorted Random Array in Ascending Order

Figure : Running Times of Sorted Random Array in Ascending Order

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | n = 5 | n = 10 | n = 50 | n = 100 | n = 500 | n = 1000 | n = 10000 |
| Selection | 0 | 0 | 1 | 0 | 27 | 5 | 677 |
| Insertion | 0 | 0 | 0 | 2 | 8 | 98 | 805 |
| Merge Sort | 0 | 0 | 0 | 2 | 0 | 1 | 39 |
| Quicksort | 0 | 0 | 0 | 1 | 2 | 7 | 186 |

Table : Running Times of Sorted Random Array in Descending Order

Figure : Running Times of Sorted Random Array in Descending Order

**Analysis:**

It can be seen that Merge Sort and Quicksort generally perform significantly faster than Selection and Insertion Sort. This is consistent with the theoretical time complexity with Selection Sort and Insertion Sort being O(n2) and Merge Sort and Quicksort being O(n log n). Despite this, the difference in performance is really only obvious in the very large n = 10000 input case, with all other cases resulting in almost identical performance.

Looking at each individual sorting order (random, ascending and descending), it can be seen that selection sort takes the same amount of time with all 3 inputs. This is likely because every single value in the array is compared with every other value. Insertion sort is slow for the random and descending array, since in the ascending case only a single comparison per element is required. These results are expected when considering the implementation details of these algorithms.

While Quicksort and Merge Sort are significantly faster than the other two algorithms for almost all cases, Quicksort also performs slower than Merge Sort in almost all cases. This is despite both having the same time complexity. One possible reason for this speed difference is likely the number of copies that are made in the implementation of Quicksort in order to ensure that the sorting occurs in-place rather than in a new array. Each copy likely results in many primitive operations and is run for up to *n* iterations each time which would dominate the *nlogn* best case scenario.

**Timing Java Code**

import java.util.Arrays;  
import java.util.Collections;  
import java.util.Random;  
  
public class SortingAlgorithmTimer {  
 @FunctionalInterface  
 private interface Sorter {  
 <T extends Comparable> void sort(T[] input, boolean reversed);  
 }  
  
 public static void main(String args[]) {  
 int[] sizes = {5, 10, 50, 100, 500, 1000, 10000};  
 Sorter[] algorithms = {SortingAlgorithms::**selectionSort**, SortingAlgorithms::**insertionSort**, SortingAlgorithms::**mergeSort**, SortingAlgorithms::**quickSort**};  
 String[] algorithmNames = {"Selection", "Insertion", "Merge", "Quick"};  
  
 for (Integer size : sizes) {  
 for (int i = 0; i < algorithms.length; i++) {  
 Integer[] unsorted = **generateArray**(size);  
 long timeTaken = **timeAlgorithm**(unsorted, algorithms[i]);  
 System.**out**.printf("Random (%s, %d): %dms\n", algorithmNames[i], size, timeTaken);  
  
 Integer[] ascending = **generateAscendingArray**(size);  
 timeTaken = **timeAlgorithm**(ascending, algorithms[i]);  
 System.**out**.printf("Ascending (%s, %d): %dms\n", algorithmNames[i], size, timeTaken);  
  
 Integer[] descending = **generateDescendingArray**(size);  
 timeTaken = **timeAlgorithm**(descending, algorithms[i]);  
 System.**out**.printf("Descending (%s, %d): %dms\n", algorithmNames[i], size, timeTaken);  
   
 System.**out**.println();  
 }  
 }  
 }  
  
 private static <T extends Comparable> long timeAlgorithm(

Integer[] toSort, Sorter sortingAlgorithm) {  
 long startTime = System.**currentTimeMillis**();  
 sortingAlgorithm.sort(toSort, false);  
 long endTime = System.**currentTimeMillis**();  
 return (endTime - startTime);  
 }

private static Integer[] generateArray(int size) {  
 Random randomGenerator = new Random();  
 Integer[] generatedArray = new Integer[size];  
 for (int i = 0; i < size; i++) {  
 generatedArray[i] = randomGenerator.nextInt();  
 }  
 return generatedArray;  
 }  
  
 private static Integer[] generateAscendingArray(int size) {  
 Integer[] generatedArray = **generateArray**(size);  
 Arrays.**sort**(generatedArray);  
 return generatedArray;  
 }  
  
 private static Integer[] generateDescendingArray(int size) {  
 Integer[] generatedArray = **generateArray**(size);  
 Arrays.**sort**(generatedArray);  
 Collections.**reverse**(Arrays.**asList**(generatedArray));  
 return generatedArray;  
 }  
}