Runtime for **UNSORTED** numbers (in milliseconds):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Algorithm** | n = 5 | n = 10 | n = 50 | n = 100 | n = 500 | n = 1000 | n = 10000 |
| Selection Sort | 10.9309 | 0.0058 | 0.3417 | 0.2719 | 2.8320 | 18.6658 | 139.3359 |
| Insertion Sort | 0.00680 | 0.0044 | 0.0248 | 0.1072 | 2.8399 | 3.01470 | 171.3940 |
| Merge Sort | 0.01110 | 0.0090 | 0.0345 | 0.3305 | 1.1216 | 7.31120 | 85.46060 |
| Quick Sort | 0.00860 | 0.0057 | 0.0194 | 0.0701 | 0.2457 | 0.20200 | 8.099500 |

Chart - 1

Runtime for **ASCENDING** numbers (in milliseconds):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Algorithm** | n = 5 | n = 10 | n = 50 | n = 100 | n = 500 | n = 1000 | n = 10000 |
| Selection Sort | 6.0E-4 | 7.0E-4 | 0.0029 | 0.0400 | 0.2210 | 0.5938 | 125.0601 |
| Insertion Sort | 0.0372 | 0.0532 | 0.0336 | 0.0525 | 0.0243 | 0.0591 | 0.19880 |
| Merge Sort | 0.0039 | 0.0059 | 0.0266 | 0.0567 | 0.3813 | 1.0779 | 54.3279 |
| Quick Sort | 0.0031 | 0.0042 | 0.1350 | 0.1299 | 3.3800 | 12.4442 | 322.2035 |

Chart - 2

Runtime for **DESCENDING** numbers (in milliseconds):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Algorithm** | n = 5 | n = 10 | n = 50 | n = 100 | n = 500 | n = 1000 | n = 10000 |
| Selection Sort | 9.0E-4 | 0.0010 | 0.0058 | 0.0140 | 0.2645 | 1.9458 | 205.0326 |
| Insertion Sort | 7.0E-4 | 0.0010 | 0.0061 | 0.0188 | 0.3984 | 2.3863 | 193.2124 |
| Merge Sort | 0.0013 | 0.0032 | 0.0055 | 0.0114 | 0.1085 | 0.3040 | 48.348 |
| Quick Sort | 0.0083 | 0.0018 | 0.0035 | 0.0072 | 0.1152 | 0.4320 | 41.400 |

Chart - 3

**Discussion**

**Selection Sort:**

Since the time complexity of selection sort is *O(N2)* , as we can see from Chart-1, for smaller values of *n*, selection sort performs less efficiently than other algorithms, the running time dramatically increases as *n* increases.

It’s the best case if the array is already sorted in ascending order when selection sort is called, since the algorithm doesn’t have to do swap operation after each comparison, but it’s still *O(N2)* , the two loops still execute the same number of times, regardless of whether the array is sorted or not, so the run time is not going to improve much. (shows in Chart-2)

The worst case is that sorting a reversely sorted array, since the algorithm has to do swap operation after each comparison, consumes more time. (shows in Char-3)

**Insertion Sort:**

The time complexity of insertion sort is *O(N2)* as well, as Chart-1 shows, the running time dramatically increases as *n* increases. For smaller values of *n*, insertion sort performs efficiently like other algorithms.

It’s the best case when the array is already sorted in ascending order, since the number of searches and swaps are reduced maximally. In this case, insertion sort performs more efficiently than the other algorithms. (shows in Chart-2)

It’s the worst case when the array is already sorted in descending order, since on each iteration of its outer loop, insertion sort has to traverse all elements to find the correct place to insert the next item. (shows in Char-3)

**Merge Sort:**

The time complexity of merge sort is *O(Nlog N) ,* it’s a stable algorithm, for smaller values of *n*, merge sort performs efficiently like other algorithms, the running time slowly increases as *n* increases. It shows stable characteristics in unsorted, ascending and descending all three cases.

**Quick Sort:**

The time complexity of quick sort in practice is *O(Nlog N) ,* for smaller values of *n*, quick sort performs efficiently like other algorithms, in unsorted and descending cases, the running time slowly increases as *n* increases, and performs most efficiently in all four algorithms. (shows in Chart-1, Chart-3)

It’s the worst case when the array is already sorted in ascending order, since in this case, the pivot is the smallest, the calls is from a linear tree rather a balanced binary tree. The time complexity of quick sort in worst case is *O(N2) ,* the running time dramatically increases as *n* increases, and performs least efficiently in all four algorithms. (shows in Chart-2)

**Source Code:**

**import** java.util.Arrays;  
**import** java.util.Collections;  
**import** java.util.Random;  
  
**public class** TimeAlgorithms {  
 **private** Integer[] **unsorted**;  
 **private** Integer[] **ascending**;  
 **private** Integer[] **descending**;  
  
 **public void** generateArrays(**int** n, String s) {  
 Random num = **new** Random();  
 Integer[] temp = **new** Integer[n];  
 **for** (**int** i = 0; i < n; ++i) {  
 temp[i] = num.nextInt(n);  
 }  
 **if** (s.equals(**"unsorted"**)) {  
 **unsorted** = **new** Integer[n];  
 **unsorted** = temp;  
 } **else if** (s.equals(**"ascending"**)) {  
 Arrays.*sort*(temp);  
 **ascending** = **new** Integer[n];  
 **ascending** = temp;  
 } **else** {  
 Arrays.*sort*(temp, Collections.*reverseOrder*());  
 **descending** = **new** Integer[n];  
 **descending** = temp;  
 }  
 }  
  
 **public void** runTest(**int** n, Integer[] array) {  
 System.***out***.printf(**"n = %d%n"**, n);  
  
 *// test for Selection Sort* Integer[] arrayS = **new** Integer[n];  
 System.*arraycopy*(array, 0, arrayS, 0, n);  
 **long** startS = System.*nanoTime*();  
 SortingAlgorithms.*selectionSort*(arrayS, **false**);  
 **long** endS = System.*nanoTime*();  
 **double** timeS = (endS - startS) / 1000000.0;  
 System.***out***.println(**"Selection Sort Runtime: "** + timeS + **" ms"**);  
  
 *// test for Insertion Sort* Integer[] arrayI = **new** Integer[n];  
 System.*arraycopy*(array, 0, arrayI, 0, n);  
 **long** startI = System.*nanoTime*();  
 SortingAlgorithms.*insertionSort*(arrayI, **false**);  
 **long** endI = System.*nanoTime*();  
 **double** timeI = (endI - startI) / 1000000.0;  
 System.***out***.println(**"Insertion Sort Runtime: "** + timeI + **" ms"**);  
  
 *// test for Merge Sort* Integer[] arrayM = **new** Integer[n];  
 System.*arraycopy*(array, 0, arrayM, 0, n);  
 **long** startM = System.*nanoTime*();  
 SortingAlgorithms.*mergeSort*(arrayM, **false**);  
 **long** endM = System.*nanoTime*();  
 **double** timeM = (endM - startM) / 1000000.0;  
 System.***out***.println(**"Merge Sort Runtime: "** + timeM + **" ms"**);  
  
 *// test for Quick Sort* Integer[] arrayQ = **new** Integer[n];  
 System.*arraycopy*(array, 0, arrayQ, 0, n);  
 **long** startQ = System.*nanoTime*();  
 SortingAlgorithms.*quickSort*(arrayQ, **false**);  
 **long** endQ = System.*nanoTime*();  
 **double** timeQ = (endQ - startQ) / 1000000.0;  
 System.***out***.println(**"Quick Sort Runtime: "** + timeQ + **" ms"**);  
 }  
  
 **public static void** main(String[] args) {  
 **int**[] cases = {5, 10, 50, 100, 500, 1000, 10000};  
 TimeAlgorithms test = **new** TimeAlgorithms();  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\* test for unsorted numbers \*\*\*\*\*\*\*\*\*\*"**);  
 **for** (**int** i = 0; i < 7; ++i) {  
 test.generateArrays(cases[i], **"unsorted"**);  
 test.runTest(cases[i], test.**unsorted**);  
 }  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\* test for ascending numbers \*\*\*\*\*\*\*\*\*\*"**);  
 **for** (**int** i = 0; i < 7; ++i) {  
 test.generateArrays(cases[i], **"ascending"**);  
 test.runTest(cases[i], test.**ascending**);  
 }  
  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\* test for descending numbers \*\*\*\*\*\*\*\*\*\*"**);  
 **for** (**int** i = 0; i < 7; ++i) {  
 test.generateArrays(cases[i], **"descending"**);  
 test.runTest(cases[i], test.**descending**);  
 }  
 }  
}