Activity 1. Direct exchange or Bubble algorithm

Using **Vector.java** class, we can create an ordered vector, a vector in reverse order, and a randomize vector. The **Bubble.java** class implements the bubble sort algorithm needed for sort this vector. Finally with the **BubbleTimes.java** we can increase the size of the problem and the times that it's executed.

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **t ordered** | **t reverse** | **t random** |
| 10000 | 627 ms | 1942 ms | 1422 ms |
| 20000 | 2544 ms | 7986 ms | 5827 ms |
| 40000 | 9827 ms | 31729 ms | 23609 ms |
| 80000 | 41002 ms | Oot | Oot |
| 160000 | Oot | Oot | Oot |

The times obtained matches with what was expected because the Bubble sort algorithm has a time complexity of O(n2)

Activity 2. Selection algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **t ordered** | **t reverse** | **t random** |
| 10000 | 475 ms | 538 ms | 663 ms |
| 20000 | 1994 ms | 3146 ms | 2102 ms |
| 40000 | 8403 ms | 9302 ms | 9675 ms |
| 80000 | 35484 ms | 38448 ms | 36442 ms |
| 160000 | Oot | Oot | Oot |

It can be seen that, although the complexity is the same as in the Bubble algorithm, the Selection algorithm works better for unordered vectors than the bubble one

Activity 3. Insertion algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **t ordered** | **t reverse** | **t random** |
| 10000 | LoR | 1167 ms | 382 ms |
| 20000 | LoR | 3862 ms | 1584 ms |
| 40000 | LoR | 13083 ms | 6008 ms |
| 80000 | LoR | 54204 ms | 25926 ms |
| 160000 | LoR | Oot | Oot |
| 320000 | LoR | Oot | Oot |
| 640000 | LoR | Oot | Oot |
| 1280000 | LoR | Oot | Oot |
| 2560000 | 80 ms | Oot | Oot |
| 5120000 | 120 ms | Oot | Oot |
| 10240000 | 236 ms | Oot | Oot |
| 20480000 | 468 ms | Oot | Oot |
| 40960000 | 936 ms | Oot | Oot |
| 81920000 | 1883 ms | Oot | Oot |

As it can be seen, the performance of the Insertion algorithm is really fast when the array is already sorted. It has a time complexity of O(n) for sorted arrays because it only needs to iterate the array once, without swapping any value and inserting all the values into the sorted array in order.

Activity 4. Quicksort algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **t ordered** | **t reverse** | **t random** |
| 250000 | 71 ms | 75 ms | 145 ms |
| 500000 | 150 ms | 148 ms | 294 ms |
| 1000000 | 301 ms | 314 ms | 636 ms |
| 2000000 | 622 ms | 637 ms | 1399 ms |
| 4000000 | 1185 ms | 1276 ms | 3194 ms |
| 8000000 | 2365 ms | 2680 ms | 7729 ms |
| 16000000 | 5103 ms | 5383 ms | 20487 ms |

As it can be seen, the quicksort algorithm is really fast, also for big size arrays. The fact that we can obtain better performance on the ordered and reverse algorithm is because our quicksort algorithm uses the median of three approach to obtain the pivot, so it always takes the best one.

If we introduce a random vector of that size in the bubble, selection and insertion algorithm, we will need to wait so more time:

O(n) -> 20487 ms

O(n^2) -> t2

Therefore, t2 in milliseconds will be t2 = (20487 \* 16,000,0002) / 16,000,000 = 327.792.000.000. Therefore, those algorithms will take over 3793 days to end the execution

Activity 5. Quicksort + Insertion algorithm

|  |  |
| --- | --- |
| **N** | **t random** |
| Quicksort | 20164 ms |
| Quicksort+Insertion (k = 5) | 21991 ms |
| Quicksort+Insertion (k = 10) | 21813 ms |
| Quicksort+Insertion (k = 20) | 20616 ms |
| Quicksort+Insertion (k = 30) | 21183 ms |
| Quicksort+Insertion (k = 40) | 20956 ms |
| Quicksort+Insertion (k = 50) | 20407 ms |
| Quicksort+Insertion (k = 100) | 18811 ms |
| Quicksort+Insertion (k = 200) | 15703 ms |
| Quicksort+Insertion (k = 500) | 26015 ms |
| Quicksort+Insertion (k = 1000) | 55302 ms |

As it can be seen, I’m obtaining the best performance when k =200. In this case, we’re taking advantage of the Insertion method, that has the best performance with ordered arrays, and iterating less on the quicksort, so we gain time.