Activity 2. Direct exchange or Bubble algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| n | t ordered | t reverse | t random |
| 10000 | 582 | 2043 | 1417 |
| 2\*10000 | 3023 | 8085 | 6083 |
| 2\*\*2\*10000 | 9278 | 31698 | 24770 |
| 2\*\*3\*10000 | 38678 | OOT | OOT |
| 2\*\*4\*10000 | OOT | OOT | OOT |

The bubble algorithm has a complexity of O(n^2), it works best on already ordered vectors as expected, followed by random ordered vectors, however the worse performance would be on the reversed vector, as it would have to perform too many swaps.

Activity 3. Selection algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| n | t ordered | t reverse | t random |
| 10000 | 627 | 541 | 537 |
| 2\*10000 | 1919 | 2244 | 2007 |
| 2\*\*2\*10000 | 7755 | 9560 | 8175 |
| 2\*\*3\*10000 | 33208 | 40552 | 36013 |
| 2\*\*4\*10000 | OOT | OOT | OOT |

The complexity of this method is O(n^2), it performs best on the ordered vector, as it has to search for the minimum value, for this reason, random goes 2nd and lastly the reversed, as it is one of the worst-case scenarios (it has to go all the way back to find the minimum value, consistently.

Activity 4. Insertion algorithm

The complexity of this method is O(n^2), it performs best on the ordered vector, followed by the random one. The reverse has the worst performance as elements need to be compared with all the elements before it until finding its correct position.

|  |  |  |  |
| --- | --- | --- | --- |
| n | t ordered | t reverse | t random |
| 10000 | LOR | 746 | 382 |
| 2\*10000 | LOR | 2923 | 1461 |
| 2\*\*2\*10000 | LOR | 1155 | 5813 |
| 2\*\*3\*10000 | LOR | 47299 | 30792 |
| 2\*\*4\*10000 | LOR | OOT | OOT |
| 2\*\*5\*10000 | LOR | OOT | OOT |
| 2\*\*6\*10000 | LOR | OOT | OOT |
| 2\*\*7\*10000 | LOR | OOT | OOT |
| 2\*\*8\*10000 | 59 | OOT | OOT |
| 2\*\*9\*10000 | 119 | OOT | OOT |
| 2\*\*10\*10000 | 233 | OOT | OOT |
| 2\*\*11\*10000 | 467 | OOT | OOT |
| 2\*\*12\*10000 | 945 | OOT | OOT |
| 2\*\*13\*10000 | 1864 | OOT | OOT |
| 2\*\*14\*10000 | 3720 | OOT | OOT |

Activity 5. Quicksort algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| n | t ordered | t reverse | t random |
| 10000 | LOR | LOR | LOR |
| 2\*10000 | LOR | LOR | LOR |
| 2\*\*2\*10000 | LOR | LOR | LOR |
| 2\*\*3\*10000 | LOR | LOR | LOR |
| 2\*\*4\*10000 | LOR | LOR | 78 |
| 2\*\*5\*10000 | 70 | 79 | 160 |
| 2\*\*6\*10000 | 147 | 161 | 356 |
| 2\*\*7\*10000 | 299 | 325 | 767 |
| 2\*\*8\*10000 | 613 | 673 | 2195 |
| 2\*\*9\*10000 | 1272 | 1986 | 3627 |
| 2\*\*10\*10000 | 2622 | 5531 | 9468 |
| 2\*\*11\*10000 | 5500 | 10495 | 29372 |
| 2\*\*12\*10000 | 12030 | 18173 | OOT |
| 2\*\*13\*10000 | 25461 | 30649 | OOT |
| 2\*\*14\*10000 | 52105 | OOT | OOT |

The complexity for quicksort is O (n log n), this is the fastest of all the previous ones. In this case, as before the ordered is the fastest, however in this case, the reverse vector has better performance than the random, and this is because as the median is applied, in the reversed sorted vector, the result is more predictable and tends to less variations, resulting in less recursive calls, iterations and partitions.

Based in the previous theoretical complexities, we can say that the ratio is (n log n)/n^2

Resulting in log n/n, considering that quicksort takes 23727 ms, that n is 16 million and that a day has 86,400,000 ms, we can do:

23727\*(log 16000000)/16000000 \* 1/86,400,000 = 0,0347 days

The other 3 algorithms would take 0,0347 days, which would be roughly 50 mins.

Activity 6. Quicksort + Insertion algorithm

|  |  |
| --- | --- |
| n | t random |
| Quicksort | 23727 |
| Quicksort + insertion (k=5) | 24970 |
|  |
| Quicksort + insertion (k=10) | 24602 |  |
|  |
| Quicksort + insertion (k=20) | 23983 |  |
|  |
| Quicksort + insertion (k=30) | 23563 |  |
|  |
| Quicksort + insertion (k=50) | 23051 |  |
|  |
| Quicksort + insertion (k=100) | 21156 |  |
|  |
| Quicksort + insertion (k=200) | 18522 |  |
|  |
| Quicksort + insertion (k=500) | 30844 |  |
|  |
| Quicksort + insertion (k=1000) | 60854 |  |
|  |

We can see that there is an improvement for the time on small k from 30 to 200, being 200 the most optimal value for k to improve performance.

As a conclusion we can see that Insertion is more efficient when working with smaller subarrays, that’s why it works best with larger k, however it comes to a point (k>200) where al those calls to Insertion are counterproductive.