* 1. N = 35

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00000 | 47.2% |
| BUBBLESORT | 0.00000 | 49.0% |
| SELECTIONSORT | 0.00099 | 41.1% |
| INSERTIONSORT | 0.00099 | 41.5% |

1.2 N = 580

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00100 | 47.2% |
| BUBBLESORT | 0.02593 | 49.0% |
| SELECTIONSORT | 0.01097 | 41.5% |
| INSERTIONSORT | 0.01496 | 41.5% |

* 1. N = 1.200

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00399 | 47.2% |
| BUBBLESORT | 0.11719 | 49.0% |
| SELECTIONSORT | 0.04738 | 41.0% |
| INSERTIONSORT | 0.05884 | 41.5% |

* 1. N = 10.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.04588 | 47.2% |
| BUBBLESORT | 8.53345 | 49.4% |
| SELECTIONSORT | 3.34610 | 41.0% |
| INSERTIONSORT | 4.21679 | 41.4% |

* 1. N = 100.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.63930 | 47.2% |
| BUBBLESORT | 877.46948 | 42.3% |
| SELECTIONSORT | 344.30601 | 41.4% |
| INSERTIONSORT | 443.95488 | 36.8% |

* 1. N = 1.000.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 8.27562 | 47.4% |
| BUBBLESORT | 0.00000 | % |
| SELECTIONSORT | 0.00099 | % |
| INSERTIONSORT | 0.00099 | % |

* 1. N = 8.000.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 82.14722 | 49.3% |
| BUBBLESORT | 0.00000 | % |
| SELECTIONSORT | 0.00099 | % |
| INSERTIONSORT | 0.00099 | % |

Observa-se uma boa margem em termos de tempo de execução para o Quicksort ordenando os conjuntos esparsos, tomando aproximadamente a metade do tempo dos demais destacados. Entretanto, observando o segundo critério, o Timsort tem melhores % de consumo de memória.

**CONJUNTOS ORDENADOS DE FORMA ASCENDENTE**

* 1. N = 35

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00000 | 36.3% |
| BUBBLESORT | 0.00000 | 38.9% |
| SELECTIONSORT | 0.00000 | 36.7% |
| INSERTIONSORT | 0.00000 | 37.1% |

* 1. N = 580

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00100 | 36.4% |
| BUBBLESORT | 0.01296 | 38.8% |
| SELECTIONSORT | 0.00997 | 36.7% |
| INSERTIONSORT | 0.00000 | 37.1% |

* 1. N = 1.200

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00299 | 36.3% |
| BUBBLESORT | 0.05992 | 38.8% |
| SELECTIONSORT | 0.04538 | 36.7% |
| INSERTIONSORT | 0.00000 | 37.1% |

* 1. N = 10.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.02693 | 36.3% |
| BUBBLESORT | 4.40928 | 38.9% |
| SELECTIONSORT | 3.25784 | 36.7% |
| INSERTIONSORT | 0.00100 | 37.1% |

* 1. N = 100.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.36404 | 36.3% |
| BUBBLESORT | 445.21163 | 38.8% |
| SELECTIONSORT | 327.51647 | 40.9% |
| INSERTIONSORT | 0.01795 | 37.2% |

* 1. N = 1.000.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 4.60725 | 36.6% |
| BUBBLESORT | 0.00000 | % |
| SELECTIONSORT | 0.00099 | % |
| INSERTIONSORT | 0.18102 | 37.4% |

* 1. N = 8.000.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 43.86189 | 38.4% |
| BUBBLESORT | 0.00000 | % |
| SELECTIONSORT | 0.00099 | % |
| INSERTIONSORT | 1.54290 | 39.3% |

Nota-se que na medida que os algoritmos executam conjuntos maiores, o Timsort e o MergeSort se mostram muito eficientes e alcançam ótimos números nos critérios estabelecidos. Em contrapartida, o QuickSort peca muito sobre os conjuntos maiores, sendo superado até pelo BubbleSort.

**CONJUNTOS ORDENADOS DE FORMA DESCENDENTE**

* 1. N = 35

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00000 | 36.6% |
| BUBBLESORT | 0.00000 | 42.7% |
| SELECTIONSORT | 0.00000 | 41.0% |
| INSERTIONSORT | 0.00000 | 41.5% |

* 1. N = 580

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00199 | 36.6% |
| BUBBLESORT | 0.03690 | 42.7% |
| SELECTIONSORT | 0.01247 | 40.6% |
| INSERTIONSORT | 0.02593 | 41.5% |

* 1. N = 1.200

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00499 | 36.5% |
| BUBBLESORT | 0.17005 | 42.7% |
| SELECTIONSORT | 0.04887 | 40.6% |
| INSERTIONSORT | 0.11868 | 41.4% |

* 1. N = 10.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.05386 | 40.8% |
| BUBBLESORT | 12.55168 | 42.8% |
| SELECTIONSORT | 3.59545 | 40.6% |
| INSERTIONSORT | 9.28793 | 41.4% |

* 1. N = 100.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.72458 | 40.0% |
| BUBBLESORT | 1339.02929 | 38.1% |
| SELECTIONSORT | 394.72673 | 41.4% |
| INSERTIONSORT | 915.94439 | 38.1% |

* 1. N = 1.000.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 8.90534 | 40.2% |
| BUBBLESORT | 0.00000 | % |
| SELECTIONSORT | 0.00099 | % |
| INSERTIONSORT | 0.00099 | % |

* 1. N = 8.000.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 78.76533 | 42.9% |
| BUBBLESORT | 0.00000 | % |
| SELECTIONSORT | 0.00099 | % |
| INSERTIONSORT | 0.00099 | % |

Semelhante ao caso anterior, o Timsort e o MergeSort também se mantiveram muito eficientes em termos de tempo de execução ordenando conjuntos que estão ordenados de trás para frente. Apesar de apresentar um tempo de execução muito elevado sobre os conjuntos maiores e ao contrário do que vimos anteriormente, o QuickSort se sai melhor nesse aspecto e e sobressai ao BubbleSort realizando as operações em até 1/3 do tempo do BubbleSort, entretanto, o BubbleSort mantém a melhor % de consumo de memória em comparação aos demais.

**CONJUNTOS CONTENDO NÚMEROS IDÊNTICOS**

* 1. N = 35

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00000 | 37.6% |
| BUBBLESORT | 0.00000 | 40.2% |
| SELECTIONSORT | 0.00000 | 38.5% |
| INSERTIONSORT | 0.00000 | 39.2% |

* 1. N = 580

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00100 | 37.5% |
| BUBBLESORT | 0.01296 | 40.0% |
| SELECTIONSORT | 0.01097 | 38.7% |
| INSERTIONSORT | 0.00000 | 39.1% |

* 1. N = 1.200

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.00199 | 37.5% |
| BUBBLESORT | 0.05935 | 40.0% |
| SELECTIONSORT | 0.04787 | 38.6% |
| INSERTIONSORT | 0.00100 | 39.1% |

* 1. N = 10.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.02793 | 37.5% |
| BUBBLESORT | 4.38586 | 40.6% |
| SELECTIONSORT | 3.38700 | 38.6% |
| INSERTIONSORT | 0.00199 | 39.1% |

4.5 N = 100.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 0.36504 | 37.6% |
| BUBBLESORT | 445.84862 | 41.2% |
| SELECTIONSORT | 338.84547 | 41.0% |
| INSERTIONSORT | 0.02094 | 39.4% |

* 1. N = 1.000.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 4.61324 | 37.8% |
| BUBBLESORT | 0.00000 | % |
| SELECTIONSORT | 0.00099 | % |
| INSERTIONSORT | 0.20146 | 39.4% |

* 1. N = 8.000.000

|  |  |  |
| --- | --- | --- |
| **ALGORITMO** | **TEMPO DE EXECUÇÃO** | **% DE MEMÓRIA** |
| TIMSORT | 44.01749 | 39.6% |
| BUBBLESORT | 0.00000 | % |
| SELECTIONSORT | 0.00099 | % |
| INSERTIONSORT | 1.72753 | 41.3% |

É de fundamental importância estabelecer testes com conjuntos que contém os elementos idênticos, seja em sua totalidade ou em grande parte. Para os conjuntos acima geramos propositalmente elementos iguais em cada um. Pode-se observar nos destaques acima que, desde os menores até os maiores conjuntos o TimSort lidera em termos de tempo de exeução e consumo de memória. Esse resultado já era esperado e comprova a eficiência desse algoritmo que não executa operações redundantes sobre os conjuntos, excepcionalmente em conjuntos cujos elementos são idênticos.

No geral, observa-se que o MergeSort tradicional possui um desempenho muito semelhante ao TimSort, mas não devemos nos esquecer que ele é uma das bases do nosso algoritmo híbrido e é esperado que eles se equiparem pois possuem uma complexidade é compartilhada.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_