Metodologias Experimentais em Informática

Análise Exploratória de Dados

Artur Coutinho - 2014230432 — Diogo Amores - 2015231975
Maria Inês Roseiro - 2015233281

28 de Outubro de 2019

1 Introdução

Este relatório foi desenvolvido no âmbito da cadeira de *Metodologias Experiementais em Informática* tendo como objectivo a clarificação e descrição da análise exploratória de dados efectuada, assim como o processo de criação e planeamento de experiências acerca de alguns algoritmos de ordenamento. Todo este processo de recolha e análise de dados tem como objetivo a obtenção de uma resposta para a seguinte pergunta proposta:

De que forma algoritmos de ordenamento são afetados por memory faults?

Para a realização deste trabalho, foi-nos fornecida a implementação de 4 algoritmos de ordenamento: QuickSort, MergeSort, InsertionSort e BubbleSort. Para além da implementação de cada um dos algoritmos também nos foi provenciado um gerador de parâmetros para os casos que se considerassem pertinentes testar.

Os scripts para a análise de dados gerados por cada algoritmo e a criação de gráficos foram realizados com recurso à linguagem R.

Este documento está dividido em 3 secções. Na primeira secção existe uma breve introdução ao trabalho realizado. Na segunda, a identificação das variáveis e a sua importância. Na terceira secção são abordados os cenários experimentais e as metodologias de análise para as experiências efectuadas com o objectivo de retirar conclusões sobre as mesmas, sendo formuladas e testadas várias hipóteses. Por último, na quarta secção encontra-se a conclusão

1.1 Identificação do problema

Para este trabalho, este primeira fase de identificação do problema já se encontra definida, sendo já referido na Introdução deste relatório. Pretende-se portanto formular hipóteses e atribuir conclusões a: ***De que forma algoritmos de ordenamento são afetados por memory faults?***. Para isso, vários cenários experimentais foram criados, sendo estes aprofundados na secção 3.

2 Identificação das variáveis

2.1 Variáveis independentes

Consideram-se variáveis independentes as que podem ser alteradas no ínicio das experiências. Foram consideradas 4 variáveis independentes:

1. Número de elementos da sequência *(n)* ;
2. Limite de variação dos elementos da sequência *(max_r)*;
3. Probabilidade de ocorrer um *memory fault* (*eps*);
4. O algoritmo de ordenamento *(QuickSort, MergeSort, InsertionSort e BubbleSort).*

2.2 Variáveis dependentes

Variáveis dependentes são as que se determinam através da realização da experiência e que cujos valores são obtidos através da fixação das variáveis independentes. Para esta análise de dados foram consideradas duas variáveis dependentes:

1. Tamanho da máxima subsequência ordenada *(max_size)*;
2. Tempo de execução de cada algoritmo *(execution_time)*;
3. Número de comparações que cada algoritmo efectua.

3 Cenários Experimentais

Os algoritmos em análise são o BubbleSort, InsertionSort, MergeSort e QuickSort, onde já se encontra implementado um modelo de falhas para o estudo em questão.

3.1 Algoritmos de Ordenamento

3.1.1 Bubble Sort

O Bubble Sort é um algoritmo de ordenamento com complexidade no *Worst Case* $O(n^2)$ comparações e trocas, *Best Case* de $O(n)$ para comparações, $O(1)$ para trocas e *Worst Case* de $O(n^2)$ para comparações e trocas. A nível de implementação, este algoritmo é dos mais simples (e o mais lento), onde o seu ordenamento é feito através da comparação de pares adjacentes de valores do array desordenado.

3.1.2 Merge Sort

O Merge Sort é um algoritmo de ordenamento com complexidade no *Worst Case*, *Best Case* e *Average Case* de $O(n \log n)$. Este utiliza uma abordagem recursiva e de *Divide and Conquer*, onde o array é dividido em 2 subarrays. Após este processo, é chamada a recursão para cada um destesrays, fazendo mais subdivisões que serão sucessivamente ordenadas. Por fim, é chamada uma função *merge* que junta estas subestruturas num array final e as ordena.

3.1.3 Quick Sort

O Quick Sort é um algoritmo de ordenamento com complexidade no *Worst Case* de $O(n^2)$, no *Best Case* de $O(n \log n)$ e em *Average Case* de $O(n \log n)$. Como o MergeSort, este algoritmo também utiliza uma abordagem recursiva e de *Divide and Conquer*.

A principal diferença relativamente aos algoritmos anteriores é a existência de um elemento *pivot* do array, sendo que na implementação utilizada esse elemento é o último.

O princípio do Quick Sort é o uso de uma função de *partition* que posiciona o elemento *pivot* na sua posição correta e a partir dai posiciona todos os outros em relação a este (maior, menor ou igual que este) sendo depois a função recursivamente chamada para a parte direita e esquerda do atual *pivot*.

3.1.4 Insertion Sort

O Insertion Sort é um algoritmo de complexidade no *Worst Case* de $O(n^2)$ de comparações e trocas, no *Best Case* de $O(n)$ comparações, e um *Average Case* de $O(n^2)$ para as comparações e as trocas. A forma como o Insertion Sort funciona pode ser comparada à maneira como uma pessoa ordena a sua mão num jogo de cartas, existe um *array* parcialmente ordenado que começa com um elemento e a cada iteração um elemento é removido do *array* original e posto no seu lugar no *array* parcialmente ordenado através de comparação direta com cada valor neste *array*, este processo é repetido até não restar nenhum elemento no *array* original.

3.2 Modelo de Falhas

Relativamente ao modelo de falhas da experiência, foi gerado um número aleatório k a cada comparação efetuada e escolhido um valor *eps* para toda a execução do algoritmo. Este modelo é constituído por duas alternativas, se k for maior que o valor de *eps* então não existe falha, caso contrário, ocorre uma troca incorreta.

3.3 Procedimento

Numa fase inicial do trabalho, os testes realizados pretendiam verificar que relações poderiam ser obtidas entre os diferentes valores das variáveis dependentes, ou seja, apenas uma variável sofreu alterações modificada. Seguidamente, pares de variáveis independentes foram sendo alteradas. Depois de analisados os testes realizados, foram formuladas hipóteses e retiradas alguma conclusões.

3.3.1 Variação do algoritmo de ordenamento

Para este teste fixaram-se os valores de *eps*, de *max_r* e de *n*, sendo o *algoritmo* em questão o único parâmetro variado, sendo que tinha como objectivo verificar a performance de cada algoritmo de acordo com uma das variáveis dependentes analisadas, o seu *max_size*. Os resultados obtidos são descritos de seguida.

Algoritmo	n	eps	max_r	média	desvio padrão	Hits	Valor médio comparações
BubbleSort	100	1/n	n/2	79.62	20.19	35	4871
QuickSort	100	1/n	n/2	40.97	13.23	0	722.03
MergeSort	100	1/n	n/2	40.93	10.84	0	672
InsertionSort	100	1/n	n/2	19.9	4.17	0	1235.08

Tabela 1: Resultados para 100 elementos

Como se pode observar, o algoritmo que apresenta melhores resultados é o BubbleSort, sendo o único que de facto consegue ordenar 35% dos arrays em questão (*max_size* é igual a *n*). De seguida, e com performances bastante semelhantes encontram-se o QuickSort e o MergeSort. Com uma performance bastante fraca encontra-se o InsertionSort.

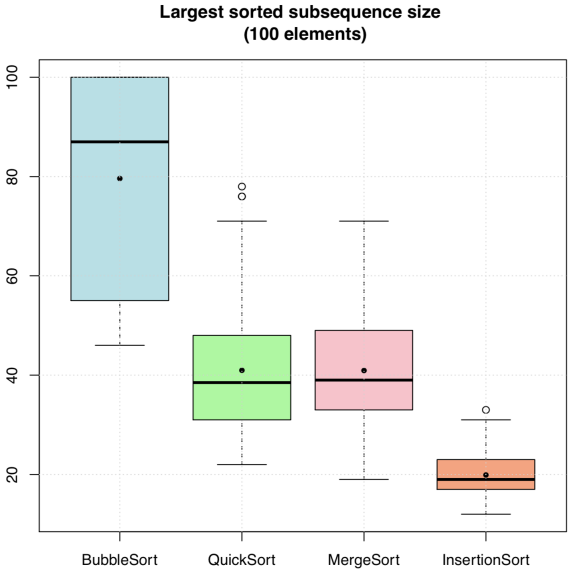


Figura 1: Resultados de *max_size* para 100 elementos

3.3.2 Variação do número de elementos

Hipótese: Com o aumento do valor de *n*, os valores relativos de *max_size* irão diminuir.

Para este teste fixaram-se os valores de *eps*, e de *max_r*. Compararam-se sequências de 100, 1000 e 10000 elementos para todos os algoritmos de ordenamento, de maneira a compreender possíveis alterações a nível dos seus *max_size*. Os resultados obtidos encontram-se nas Figura 1, Figura 2 e Figura 3 e respectivas tabelas.

Algoritmo	n	eps	max_r	média	desvio padrão	Hits	Valor médio comparações
BubbleSort	1000	1/n	n/2	713.19	181.52	1	498543
QuickSort	1000	1/n	n/2	352.57	111.50	0	11769.22
MergeSort	1000	1/n	n/2	395.32	194.59	4	9976
InsertionSort	1000	1/n	n/2	78.47	19.49	1	203372.5

Tabela 2: Resultados para 1000 elementos

Como é possível observar, o BubbleSort, ainda que com uma prestação cada vez mais baixa, é o algoritmo que continuamente apresenta os melhores resultados. Por outro lado, o InsertionSort oferece sempre performances muito inferiores relativamente aos outros algoritmos.

De acordo com o crescente tamanho dos arrays (valor de n a 100, 1000 e 10000) é visível a redução na performance de todos os algoritmos, ainda que no caso do MergeSort com 1000 elementos, com a existência de vários *outliers* (pontos não preenchidos presentes nos gráficos), essa redução se encontre mascarada. Estes resultados corroboram a hipótese inicialmente colocada, de facto os valores relativos de *max_size* decrescem com o aumento de n, ainda que os valores da probabilidade de erro sejam cada vez menores (respectivamente 1%, 0.1% e 0.01%).

Relativamente ao número de hits (número de vezes que a sequência é ordenada sem falhas de memória), os melhores resultados foram obtidos no teste de 100 elementos, e com o algoritmo BubbleSort, onde cerca de 35% das sequências se encontram corretamente ordenadas.

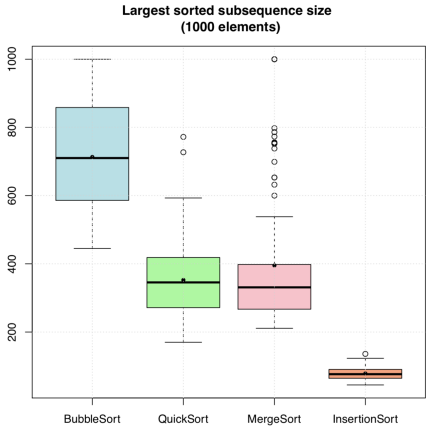


Figura 2: Resultados de *max_size* para 1000 elementos

Algoritmo	n	eps	max_r	média	desvio padrão	Hits	Valor médio comparações
BubbleSort	10000	1/n	n/2	7057.66	2101.56	0	49984666
QuickSort	10000	1/n	n/2	2090.47	602.77	0	163271
MergeSort	10000	1/n	n/2	2775.63	809.23	0	133616
InsertionSort	10000	1/n	n/2	256.95	49.05	0	1799357

Tabela 3: Resultados para 10000 elementos

3.3.3 Variação do *range* de elementos da sequência

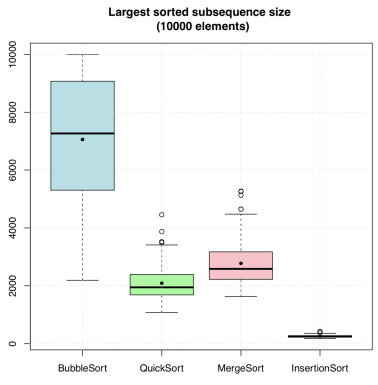
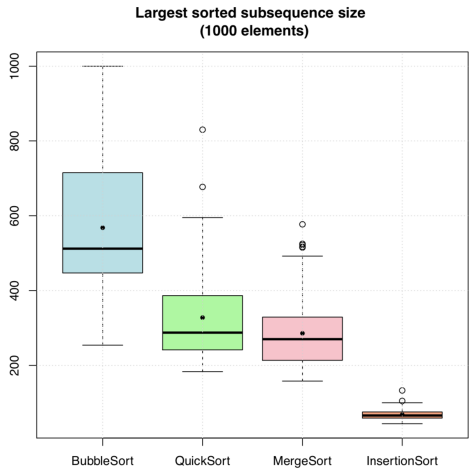


Figura 3: Resultados de *max_size* para 10000 elementos

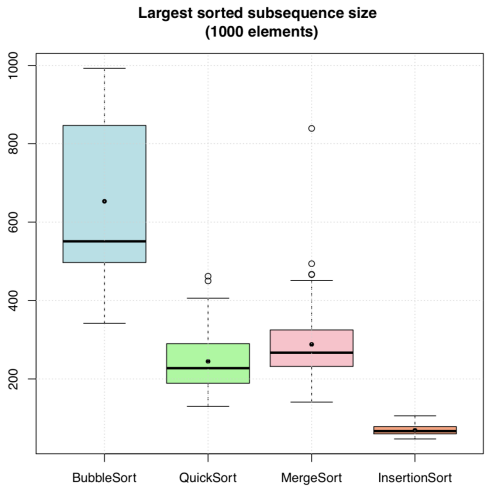
Para este teste fixaram-se os valores de *eps* e de n. Compararam-se sequências de 1000 elementos ($n = 1000$) para todos os algoritmos de ordenamento, com diferentes valores de *max_r*. Apresentamos de seguida os resultados relativos a esta experiência, nas Figuras e Tabelas 2, 4, 5 e 6, assim como a hipótese formulada para esta experiência.

Hipótese: Com a diminuição de *max_r*, obtém-se valores mais altos para *max_size*.

Com a diminuição do intervalo que os valores do *array* podem tomar, era esperado que os algoritmos obtivessem valores mais altos de *max_size*. Fazendo uso de um caso extremo, se o *max_range* de um array for o valor 1, então



(a) Resultados para 10000 elementos com max_r a n



(b) Resultados para 1000 elementos com max_r a $2 \times n$

o array terá apenas n elementos de valor 1, o que fará com que o array se encontre ordenado à partida. Com o aumento do valor n, seria esperada uma maior dispersão dos valores, e portanto uma maior possibilidade de desordenamento e uma maior probabilidade de ocorrer um *memory fault*.

Algoritmo	**n**	**eps**	**max_r**	**média**	**desvio padrão**	**Hits**	**Valor médio comparações**
BubbleSort	1000	1/n	n/4	738.78	209.53	0	498475.1
QuickSort	1000	1/n	n/4	317.46	96.72	0	11577.83
MergeSort	1000	1/n	n/4	416.26	112.92	0	9976
InsertionSort	1000	1/n	n/4	72.38	14.18	0	15626.32

Tabela 4: Resultados para 1000 elementos com max_r a $\frac{n}{4}$

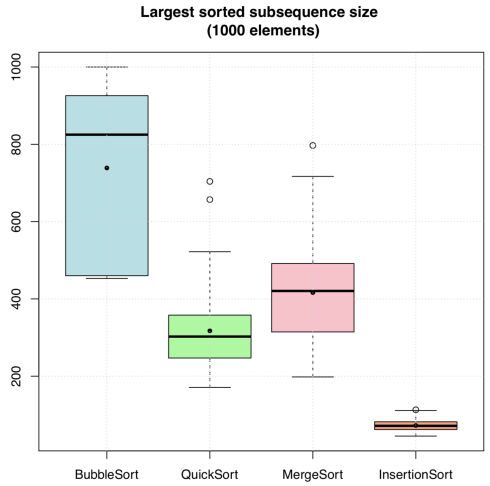


Figura 4: Resultados para 10000 elementos com max_r a $\frac{n}{4}$

No entanto, e de acordo com os dados obtidos e apresentados nas figuras e tabelas 2,4 e 5, não é possível obter conclusões acerca da hipótese anterior. Apesar de possível essa relação, a aleatoriedade das experiências realizadas não permitem estabelecer relações que permitam validar ou não a hipótese em causa.

Algoritmo	n	eps	max_r	média	desvio padrão	Hits	Valor médio comparações
BubbleSort	1000	1/n	n	567.95	207.25	1	498472
QuickSort	1000	1/n	n	327.81	119.87	0	11604.11
MergeSort	1000	1/n	n	285.73	93.03	0	9976
InsertionSort	1000	1/n	n	68.47	14.97	0	15628.16

Tabela 5: Resultados para 1000 elementos com *max_r* a n

Algoritmo	n	eps	max_r	média	desvio padrão	Hits	Valor médio comparações
BubbleSort	1000	1/n	2n	653.14	192.69	0	498424
QuickSort	1000	1/n	2n	244.79	75.01	0	11617.5
MergeSort	1000	1/n	2n	288.14	93.32	0	9976
InsertionSort	1000	1/n	2n	69.11	12.41	0	15841.2

Tabela 6: Resultados para 1000 elementos com *max_r* a $2 \times n$

3.3.4 Variação da probabilidade de erro *(eps)*

Algoritmo	n	eps	max_r	média	desvio padrão	Hits	Valor médio comparações
BubbleSort	1000	0.01	n/2	274.22	107.37	0	498555.6
QuickSort	1000	0.01	n/2	64.55	14.86	0	11560.43
MergeSort	1000	0.01	n/2	61.67	13.08	0	9976
InsertionSort	1000	0.01	2n	28.23	5.17	0	15932.58

Tabela 7: Resultados para 10000 elementos com *eps* a 1%

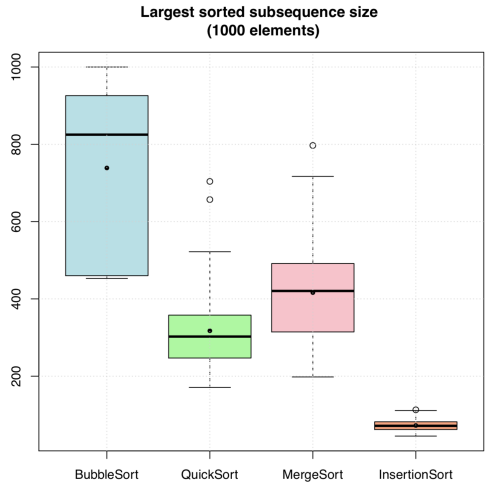


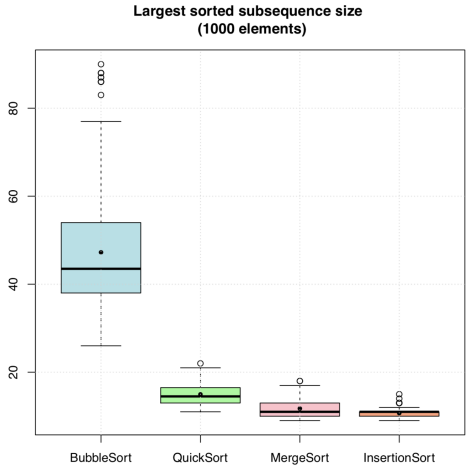
Figura 6: 10000 elementos com *eps* a 1%

Hipótese: Com o aumento de *eps*, os valores relativos a *max_size* vão diminuir. Para o teste desta hipótese, fixaram-se os valores de n e *max_r*, com o valor de n a 1000. Os resultados obtidos encontram-se nas Figura 6, e nas Figuras 7(a) e (b) e Tabelas 7, 8 e 9.

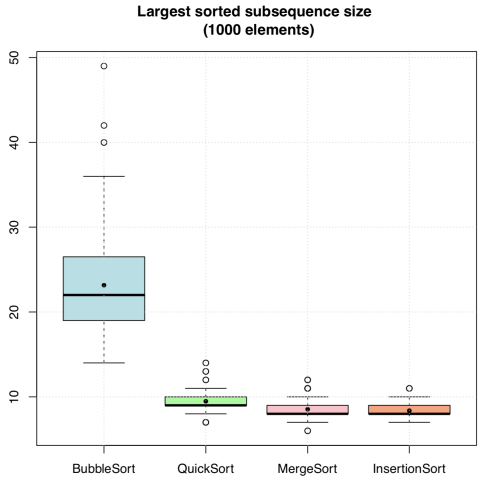
Relativamente ao teste de hipóteses, é possível estabelecer uma correlação entre o sucessivo aumento do valor de *eps* (1%, 10% e 20%) e a redução do valor de *max_size*. Tal como como previsto, existe uma decréscimo abrupto do valor de *max_size* com o aumento do valor de *eps*.

Para *eps* a 1% o comportamento é o esperado, não tendo existindo nenhum hit.

Com o aumento para 10% é visível uma redução drástica no valor de *max_n*, sendo que o algoritmo que apresenta os melhores resultados tem uma média de *max_size* de 47,25. Finalizando para um *eps* de 20%, são obtidos os valores mais baixos para *max_n* corroborando assim a nossa hipótese.



(a) Resultados para 1000 elementos com *eps* a 10%



(b) Resultados para 1000 elementos com *eps* a 20%

Algoritmo	**n**	**eps**	**max_r**	**média**	**desvio padrão**	**Hits**	**Valor médio comparações**
BubbleSort	1000	0.1	n/2	47.25	13.64	0	499383.7
QuickSort	1000	0.1	n/2	14.97	2.46	0	10413
MergeSort	1000	0.1	n/2	11.75	2.02	0	9976
InsertionSort	1000	0.1	n/2	10.7	1.34	0	4054.58

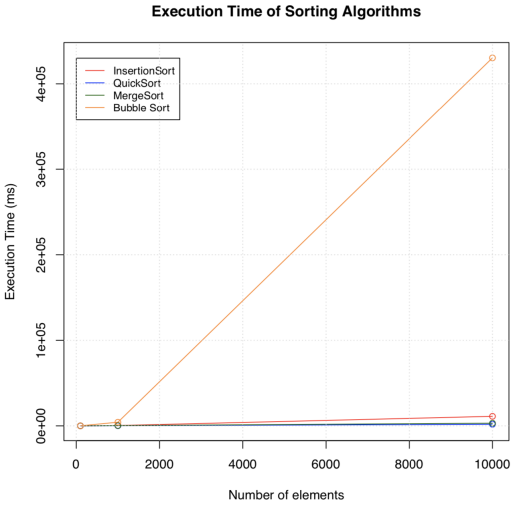
Tabela 8: Resultados para 1000 elementos com *eps* a 10%

Algoritmo	**n**	**eps**	**max_r**	**média**	**desvio padrão**	**Hits**	**Valor médio comparações**
BubbleSort	1000	0.2	n/2	23.16	6.33	0	499492.5
QuickSort	1000	0.2	n/2	9.48	1.40	0	9720.77
MergeSort	1000	0.2	n/2	8.54	1.34	0	9976
InsertionSort	1000	0.2	n/2	10.7	1.34	0	4054.58

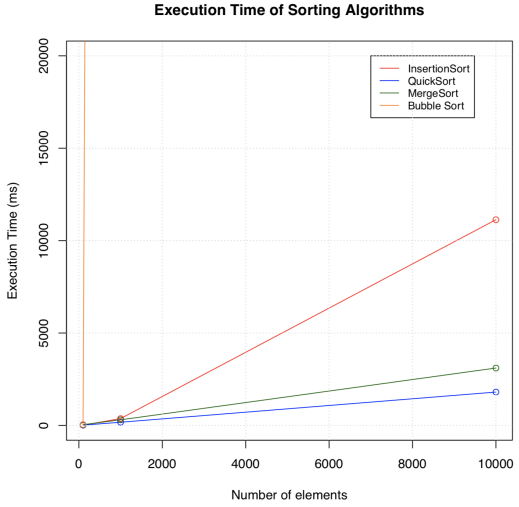
Tabela 9: Resultados para 1000 elementos com *eps* a 20%

3.3.5 Tempo de execução

Como última experiência, foi feita uma análise do tempo de execução para os valores reportados nas Tabelas 1, 2 e 3. É possível então notar que o *BubbleSort* tem um tempo de execução bastante superior relativamente a todos os outros. Essa discrepância deve-se à análise exaustiva que este algoritmo produz, quando comparados com os outros. Uma explicação mais clara será apresentada na Secção 4 - Conclusão.



(a) Resultados para o tempo de execução para os diferentes tipos de algoritmos



(b) Resultados para tempo de execução para os diferentes algoritmos sem BubbleSort

4 Conclusão

Através da análise dos vários resultados é possível concluir que o *BubbleSort* é o algoritmo que obtém os valores mais altos de *max_size*, com uma grande discrepância relativamente aos outros em estudo. O QuickSort e MergeSort mantêm-se relativamente próximos nesta questão, sendo que existem poucas alterações com base na análise que realizámos. Finalmente, o InsertionSort é o algoritmo em que o pior desempenho se verifica.

Para fundamentar estas conclusões, foi criado um parâmetro de análise que verifica o número de comparações que cada algoritmo realiza em cada teste. De acordo com esta análise, é possível estabelecer que este valor médio de comparações se relaciona com a quantidade de *memory faults*, mais especificamente, o BubbleSort realiza um número de comparações bastante mais elevado do que todos os outros algoritmos.

Em questão de ordenamento, o BubbleSort trata pares adjacentes de valores a cada iteração, terminando este processo quando existe uma iteração em que nenhuma troca é feita. Por esta razão, os resultados das análises efetuadas estão em conformância com a análise exaustiva que o BubbleSort efetua. O BubbleSort, devido à sua natureza de comparação exaustiva a e à sua simplicidade de implementação (não contém otimizações relativas à ordenação, o que se traduzirá num tempo de execução muito mais elevado).

A nível do QuickSort e do MergeSort é possível verificar que ambos mantém resultados relativamente equiparáveis e com resultados inferiores ao BubbleSort. Pensamos que tal se deva à implementação destes algoritmos em que são usados os métodos *Divide and Conquer* e recursão. Apesar de em termos de otimização se tornar muito mais eficiente do que o *BubbleSort*, o número de comparações efectuadas é bastante mais pequeno, e torna-se muito maior a probabilidade de ocorrer um *memory fault*, e de não ordenar correctamente a sequência.

Para justificar o desempenho do InsertionSort, mais uma vez, a sua implementação têm um papel fulcral na ocorrência de memory faults.

5 Referências

1. https://www.r-bloggers.com/box-plot-with-r-tutorial/
2. https://towardsdatascience.com/exploratory-data-analysis-in-r-explore-one-variable-using-pseudo-facebook-dataset-29031767eb07
3. https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/