Relatório

Metodologias Experimentais em Informática
- 1ª Meta -

João Silva - 2016252535
Rafael Neves - 2016250690

Introdução

O presente trabalho elaborado no âmbito da cadeira de metodologias experimentais em informática, tem como objetivo efetuar uma análise exploratória de dados com o intuito de experimentar, colecionar dados e usar técnicas de análise de dados para tirar conclusões sobre um problema em questão.

Concretamente, o problema por solucionar tem como questão principal: *"How are sorting algorithms affected by memory faults?".* Assim, é esperado que seja feita uma avaliação da performance de algoritmos de ordenação quando afetados por falhas de memória, através de uma análise detalhada de cada secção do problema e entender como cada algoritmo altera o resultado obtido ao presenciar uma falha na ordenação de um elemento da lista.

O problema

Com o objetivo de perceber qual é o comportamento de cada um dos algoritmos de ordenação sobre falhas de memória, foi necessário repartir o problema em três questões chave que irão ser o foco da análise realizada.

Em primeiro lugar, com o objetivo de ter uma perspetiva geral do problema, iremos analisar cada um dos algoritmos sobre as mesmas condições:

1. *Qual o algoritmo que menos sofre e mais sofre com falhas de memória sobre as mesmas condições?*

Por fim, foi variado individualmente cada um dos fatores de teste para ter uma melhor noção do efeito de cada um no resultado final:

2. *Como é que mudanças na probabilidade de falhas de memória afeta cada um dos algoritmos mantendo o tamanho da amostra fixo?*

3. *Como é que mudanças no tamanho da amostra afeta cada um dos algoritmos mantendo a probabilidade de ocorrência de falhas de memória fixa?*

Identificação de variáveis

Para responder às questões formuladas foi importante declarar variáveis dependentes e independentes dentro do problema.

O **número de elementos da maior lista ordenada** de cada ordenação realizada, assim como, o **número de elementos mal ordenados** (Ai+1 < Ai, em que i é o índice de um elemento da array A) representam as variáveis dependentes da análise. Em adição, declaramos como variáveis independentes a **probabilidade de ocorrência de falhas de memória** (*eps*), assim como, o **tamanho da amostra de teste** (*n*).

Hipóteses

Tendo em conta os fatores questão do problema podemos deduzir que:

- Independentemente do algoritmo em análise, **um acréscimo na probabilidade** de falhas de memória irá ter impacto negativo na performance de cada ordenação, por isso, é esperado que com o aumento da mesma probabilidade, o **número de erros também aumente** e o **número de elementos da maior lista ordenada diminua**.

- Por outro lado, um **maior número de elementos** da amostra em análise irá corresponder a um **maior número de elementos da maior lista ordenada** pois, ao existir mais elementos, existe uma maior a probabilidade de ter uma lista ordenada maior, assim como, **um maior número de erros.**

Ferramentas e procedimentos usados / cenário

Para efetuar a experiência foi utilizado um *script* em *Python* para gerar dados aleatórios dada uma probabilidade de falha, *eps,* e um tamanho de amostra, *n*:

- *data.in*

No mesmo *script* são chamados os algoritmos de ordenação *Quick sort*, *Bubble sort*, *Merge sort* e *Insertion Sort* que ordenam a *array* de items e gerarem os ficheiros com a *array* ordenada, o número de elementos da maior lista ordenada e a quantidade de erros detectados:

- *quick.out*

- *bubble.out*
- *merge.out*
- *insertion.out*

Ainda no *script* de *Python*, são gerados dois ficheiros com todos os dados recolhidos ao longo dos testes simulados:

- *Xsort_results.in*
- *Xsort_errors.in*

Por fim, foi utilizado um *script* R para gerar os gráficos que serão apresentados no ficheiro:

- *Rplots.pdf*

Todo o procedimento de simulação é autómato, pelo que, apenas é necessário estipular os valores de probabilidade e tamanho da amostra para correr os testes e, se necessário, correr o *script* R para apresentar os gráficos dos dados obtidos.

Dados recolhidos e análise

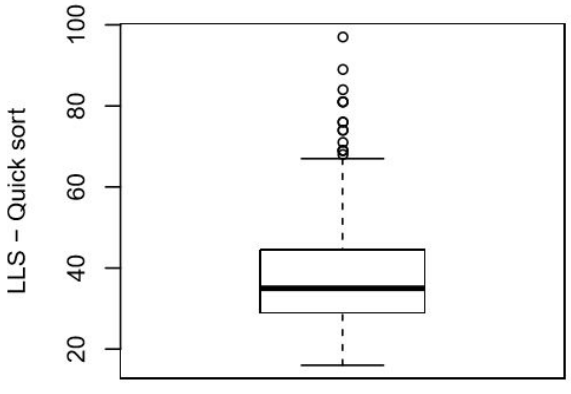
Para analisarmos o problema e respondermos às questões feitas previamente, estudamos a performance de cada algoritmo através da análise de dados e consequentes análises gráficas.

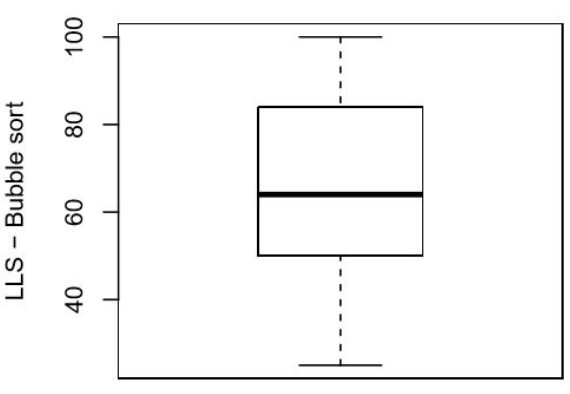
Será feita uma apresentação gráfica dos dados obtidos seguida de uma análise aos mesmo, tendo como objetivo tirar conclusões e responder às questões em causa.

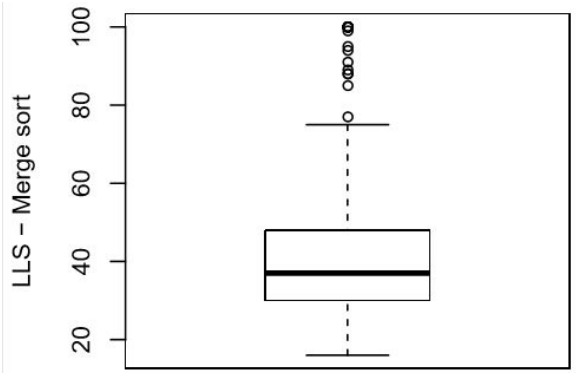
- **Qual o algoritmo que menos sofre e mais sofre com falhas de memória sobre as mesmas condições?**

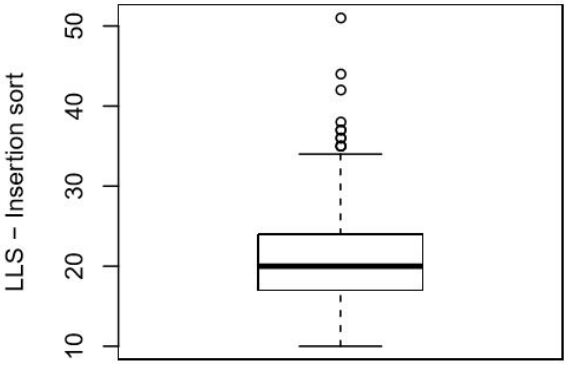
Numa simples abordagem ao problema em questão, realizamos testes sobre as mesmas condições a todos os algoritmos de estudo, observando a quantidade de erros gerados através de gráficos *Box*.

Cada caso de teste é composto por *n=100* elementos com valores entre [1, 100] e uma probabilidade *p=0.01* fixa de ocorrência de falhas. Em adição, para obter uma melhor perspectiva do impacto das falhas no algoritmo, foram feitas **500 repetições** de cada caso de teste.









Análise:

Com base nos dados anteriores, observamos que, dos algoritmos em estudo, destaca-se o *Bubble sort* que, apesar de ter valores dispersos, apresenta em mediana acima de 50% de elementos ordenados consequentemente e existem casos em que 98%-99% da array está ordenada.

De seguida, podemos observar que o *Quick sort* e o *Merge sort* produzem resultados semelhantes. Cerca de 40 elementos fazem parte da maior lista ordenada.

Também é possível analisar que o *Insertion sort* produz resultados medíocres em que observamos que a maior lista ordenada é composta em mediana por pouco mais que 20 elementos.

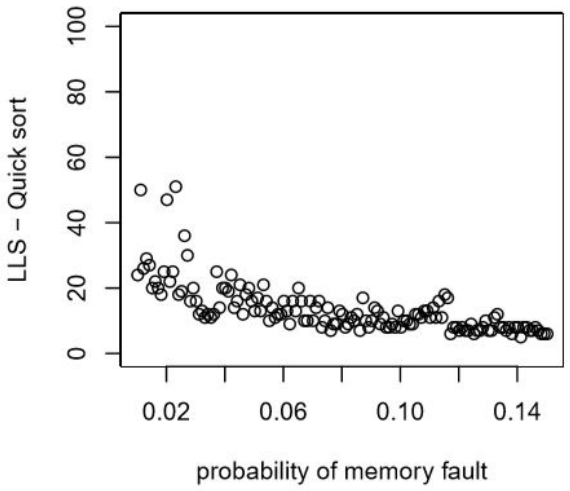
Por fim, podemos ainda observar que, à excepção do algoritmo *Bubble sort* todos os outros algoritmos apresentam vários *outliers*.

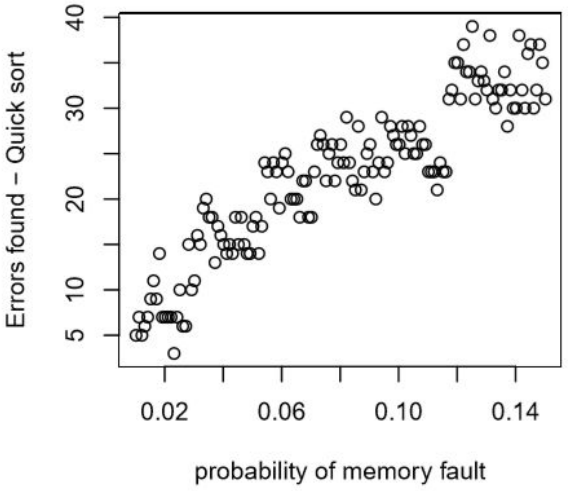
- **Qual a significância de mudar a probabilidade de uma falha de memória?**

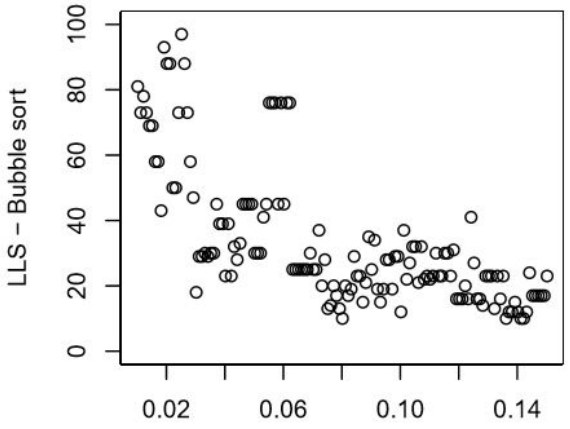
Para responder a esta pergunta variamos a probabilidade de ocorrência de falhas de memória, mantendo o tamanho da array fixo e apresentamos os resultados obtidos lado-a-lado para visualizar as mudanças ocorridas.

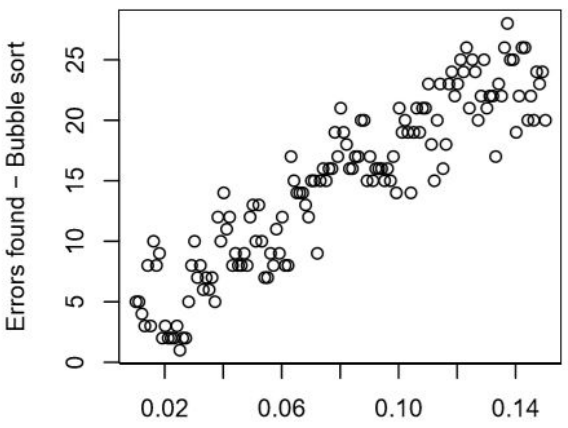
Cada coleção de testes realizados sobre uma variável produz dois gráficos. Um tem como objetivo apresentar o número de elementos da maior lista ordenada em cada um dos testes realizados, o segundo representa a quantidade de erros que cada teste presenciou.

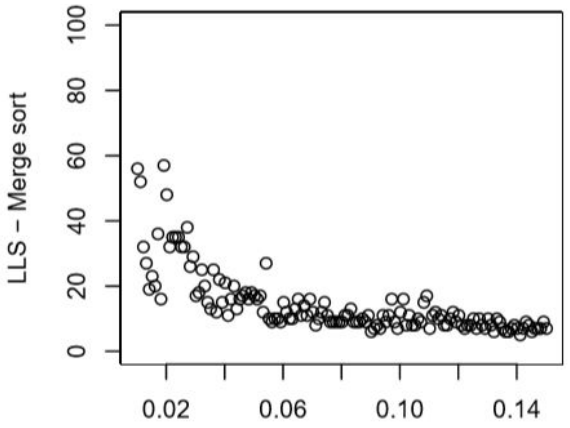
Para esta análise definimos o valor de n = 100 e a probabilidade foi inicializada a 1/n e aumentada até atingir o valor de 15/n, num salto de 0.1/n.

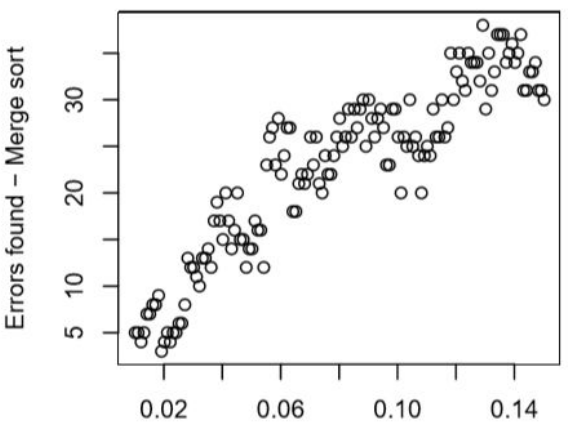

LLS – Quick sort
0
20
40
60
80
100
0.02
0.06
0.10
0.14
probability of memory fault

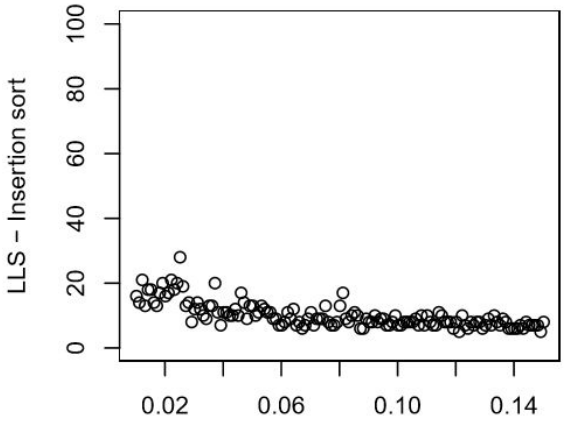

Errors found – Quick sort
5
10
20
30
40
0.02
0.06
0.10
0.14
probability of memory fault

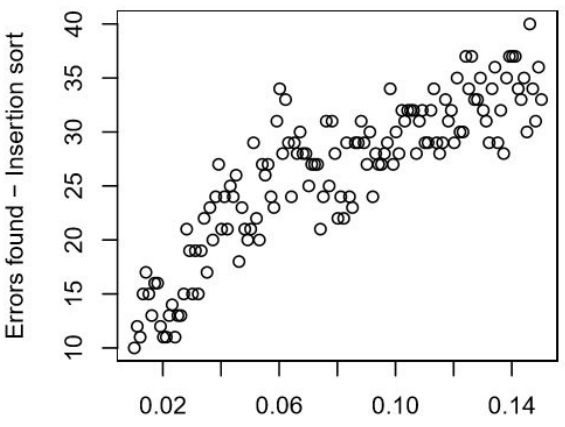

LLS – Bubble sort
0
20
40
60
80
100
0.02
0.06
0.10
0.14


Errors found – Bubble sort
0
5
10
15
20
25
0.02
0.06
0.10
0.14


LLS – Merge sort
0
20
40
60
80
100
0.02
0.06
0.10
0.14


Errors found – Merge sort
5
10
20
30
0.02
0.06
0.10
0.14


LLS – Insertion sort
0
20
40
60
80
100
0.02
0.06
0.10
0.14


Errors found – Insertion sort
10
15
20
25
30
35
40
0.02
0.06
0.10
0.14

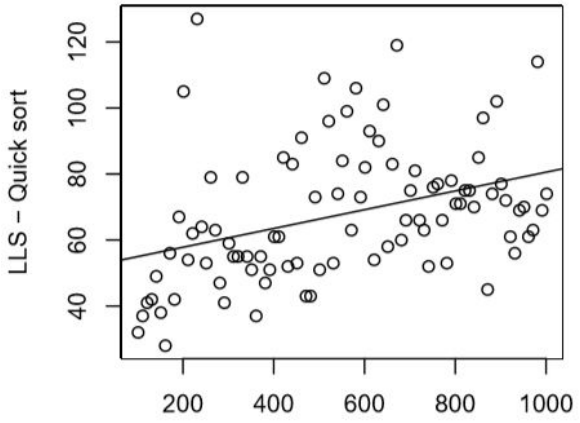
Análise:

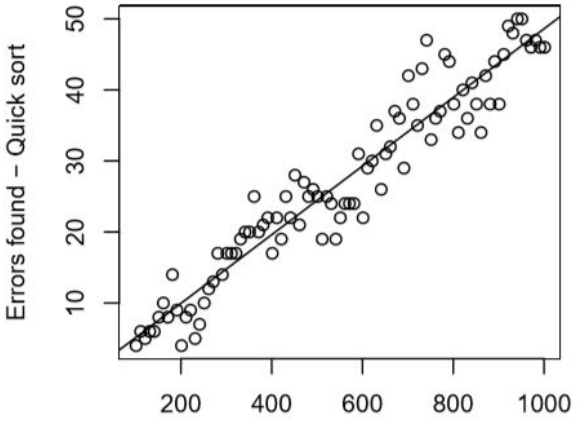
Pela análise dos gráficos podemos observar que, conforme a probabilidade de falha aumenta, os valores da maior lista ordenada tendem a diminuir e o número de erros tende a subir. Este resultado era o esperado previamente, porém, podemos observar que os algoritmos *Quick sort*, *Merge sort* e *Insertion sort* têm um comportamento semelhante e "previsível" pois diminuem (no caso da LLS) ou aumentam (no caso dos erros) de forma quase constante, enquanto que, o algoritmo de ordenação *Bubble*, apesar de continuar a ser o algoritmo com melhor performance, tem um comportamento mais "instável" e aleatório demonstrando muitas nuances nos resultados obtidos.

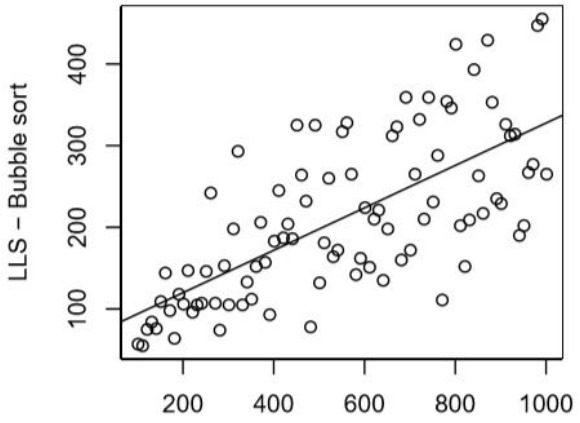
- **Qual a significância do tamanho da array a ordenar?**

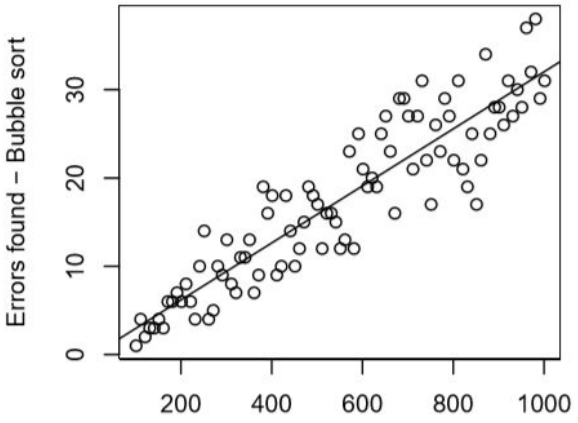
Para responder a esta pergunta utilizamos o mesmo método que a pergunta anterior, porém, alteramos apenas o tamanho da array de elementos mantendo fixa a probabilidade de **0.01**.

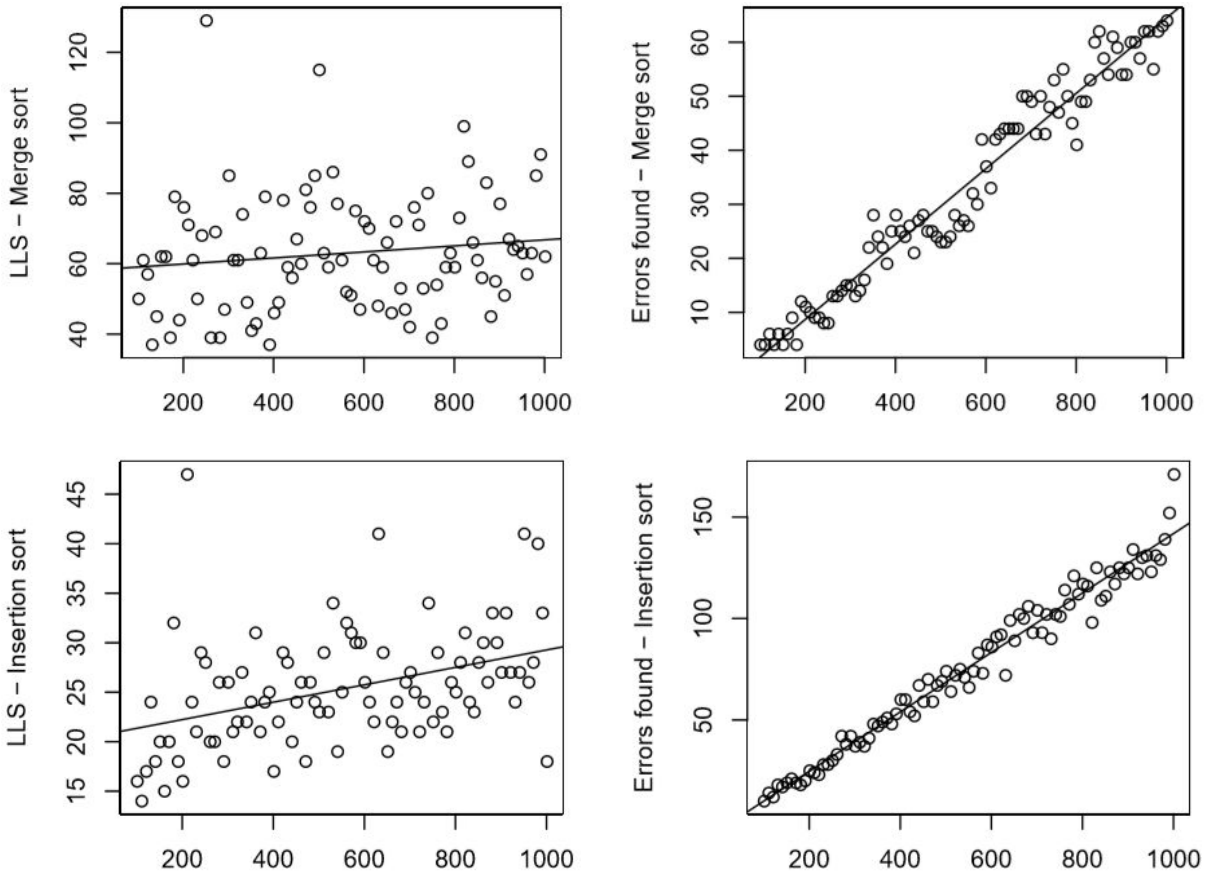
A amostra é constituída inicialmente por n=**100** elementos e é aumentada gradualmente em 10 elementos até chegar ao valor de n=**1000**.











Pelo facto de existirem muitos *outliers* nos dados, optamos por apresentar uma linha de regressão nos gráficos sendo mais fácil analisar o comportamento dos algoritmos.

Análise:

Podemos concluir que, mais uma vez, a quantidade de erros aumenta de forma linear à medida que o número da amostra sobe e os valores da maior lista ordenada aumentam gradualmente, porém, de forma inconstante, apresentando diversos *outliers* e resultados muito “dispersos”.

Podemos ainda observar que:

- O algoritmo *Merge sort*, apesar de apresentar um maior número de erros com o aumento da amostra, aumenta muito pouco os valores da maior lista ordenada, pelo que, é o algoritmo que menos sofre com a variação do número de elementos a ordenar.
- O algoritmo *Quick sort* e *Insertion sort* tendem a aumentar gradualmente o valor da maior lista ordenada.
- O algoritmo *Bubble sort* continua a ser o que apresenta melhores resultados, assim como, é o que mais beneficia de uma amostra maior tendo uma subida acentuada no valor LLS.

Conclusões

Após a análise exploratória de dados sobre o problema em questão é possível desenhar algumas conclusões.

Em primeiro lugar, é fácil concluir que o algoritmo de ordenação *Bubble*, apesar de inconstante nos valores apresentados, é sem dúvida o algoritmo que apresenta menos impacto na sua performance na presença de falhas de memória. Em contraste, podemos concluir que o algoritmo de ordenação *Insertion* é o algoritmo que mais sofre no mesmo ambiente. Os algoritmos *Quick* e *Merge sort* tendem a manter um comportamento semelhante e, apesar de sofrerem com as falhas de memória, tendem a manter um desempenho melhor que o algoritmo *Insertion sort.*

Em segundo lugar, podemos concluir que um algoritmo de ordenação sofre mais com falhas de memória quanto menor for a sua redundância. Como o algoritmo *Bubble sort* faz várias vezes a mesma comparação de elementos, tende a corrigir elementos que não foram ordenados previamente devido a uma falha. Porém, algoritmos como *Quick* e *Merge sort*, que comparam um par de elementos apenas uma vez tendem a reproduzir mais falhas pois não têm oportunidades de corrigir o erro efetuado. Por fim, a performance medíocre do algoritmo *Insertion sort* é previsível pois, um elemento é comparado com os anteriores e, como existe uma probabilidade de falha em cada comparação que o algoritmo faz, um elemento fica "facilmente" mal ordenado e, por consequência, todos os elementos posteriores vão sofrer com a falha, provocando um efeito de "bola de neve".

Por fim, podemos concluir que muitos dos *outliers* visíveis nos testes realizados (especialmente nos *outliers* visíveis nos gráficos *Box*) devem-se ao facto de estarmos a experimentar com valores de probabilidade, ou seja, um algoritmo cuja performance sofre na presença de uma falha pode não ter sofrido pois, naquele teste, não sofreu uma falha. Assim, também é relativamente simples perceber porque é que o algoritmo *Bubble sort* não presenciou muitos *outliers*. Como o algoritmo tem muita redundância, acaba sempre por corrigir as falhas de memória independentemente se estas aconteceram, ou não.