Escola de Ciências e Tecnologias

Disciplina: Estruturas de Dados e Algoritmos I

Docentes: Lígia Ferreira e João Coelho

ALGORITMOS DE ORDENAÇÃO

Segundo Trabalho Prático

Trabalho Elaborado Por:

Marlene Oliveira, Nº25999

Pedro Mateus, Nº 26048

Introdução

No âmbito da disciplina de Estrutura de Dados e Algoritmos I, foi-nos solicitada a implementação, o teste e a análise de nove algoritmos de ordenação. Os algoritmos a analisar são os seguintes: *bubblesort, insertionsort, selectionsort, quicksort, mergesort, heapsort, ranksort, shellsort e radixsort*. Note-se que, no nosso contexto, estes algoritmos ordenam apenas arrays de números aleatórios – que devem todos distintos.

A ordenação pode ser definida como o processo em que se altera a ordem dos elementos de uma sequência[1]. A ordenação pode ser efectuada, por exemplo, para que "o acesso aos dados ordenados seja efectuado de um modo mais eficiente"[2].

Um algoritmo de ordenação permite utilizar o processo descrito anteriormente de modo a ordenar os elementos de uma sequência de um modo específico – podendo ordenar totalmente ou apenas parcialmente os elementos. "As ordens mais usadas são a numérica e a lexicográfica"[3]. Existem vários algoritmos de ordenação, tendo cada um deles características mais ou menos específicas e sendo cada um deles mais ou menos eficientes. O output destes algoritmos deverá constituir uma "permutação do input"[4] dos mesmos e deverá "encontrar-se na ordem crescente"[4].

Uma descrição de cada um dos algoritmos a analisar pode ser consultada de seguida, bem como os resultados recolhidos durante os testes efectuados.

---

[1] - "*(…)significa alterar a ordem pela qual surgem os elementos de uma sequência (…)*". Fonte: *Slides da disciplina de Programação e Algoritmos* (ver **Bibliografia**)

[2] – "(…)possibilidade se acessar seus dados de modo mais eficiente." Fonte: Wikibook "*Algoritmos e Estruturas de Dados / Algoritmos de Ordenação"* (ver **Bibliografia**)

[3] – Citando o Artigo da Wikipedia "*Algoritmo de Ordenação*" (ver **Bibliografia**)

[4] – Citando o Artigo da Wikipedia "*Sorting Algorithm*" (ver **Bibliografia**)

1. Algoritmos de Ordenação

NOTA: Os valores nas tabelas foram obtidos executando 100 testes com arrays cujas dimensões correspondem aos vários valores de N. Os ficheiros resultantes dos testes são criados pela classe Testes.java e os arrays de números aleatórios são gerados pela classe RandomArray.java.

1.1 Bubblesort

O *Bubblesort* é um algoritmo de ordenação muito simples que percorre uma estrutura de dados – lista ou *array* – e ordena os seus elementos. Este algoritmo compara dois elementos adjacentes e troca-os se estes se encontrarem na ordem errada (se o primeiro elemento tem um valor superior ao do segundo elemento). Após a primeira passagem pela estrutura de dados, o elemento com o valor maior já se encontra na última posição do *array*. O algoritmo percorre a estrutura de dados até que todos os elementos da mesma se encontrem ordenados, ou seja, até que não seja possível efectuar mais trocas.

A complexidade temporal deste algoritmo é $O(N^2)$ no pior caso e $O(N)$ [5] no melhor caso.

A implementação deste algoritmo que foi utilizada durante os testes é a mesma apresentada nas aulas teóricas [6], com pequenas alterações.

Bubblesort			
N	**Tempo Máximo(ns)**	**Tempo Mínimo(ns)**	**Tempo Médio(ns)**
10	5987	2138	2657
100	334411	12829	29410
1000	4975539	886916	9323710
10000	126119550	111142044	120772107
100000	13707933066	13359752655	13601610270

Quadro 1 – Valores dos tempos de execução máximos, mínimos e médios do algoritmo para os vários valores de N.

[5] – Citando o Artigo da Wikipedia "*Bubble sort*" (ver **Bibliografia**)

[6] – Slides das Aulas Teóricas – Aula 10 (Slide 27) (ver **Bibliografia**)

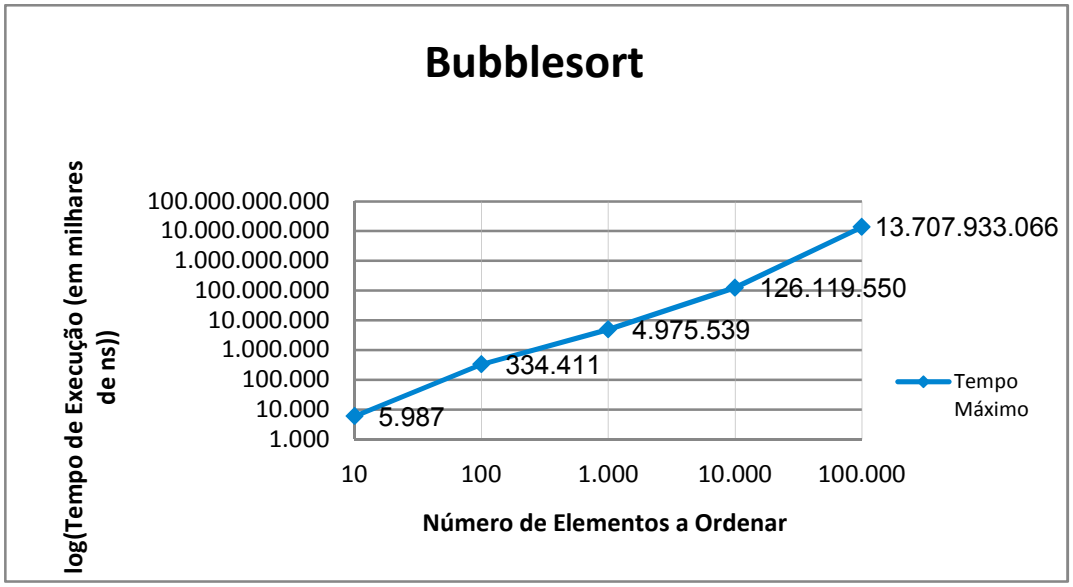

Gráfico 1 – Evolução do Tempo de Execução Máximo do Algoritmo Bubblesort à medida que o número de elementos a ordenar aumenta.

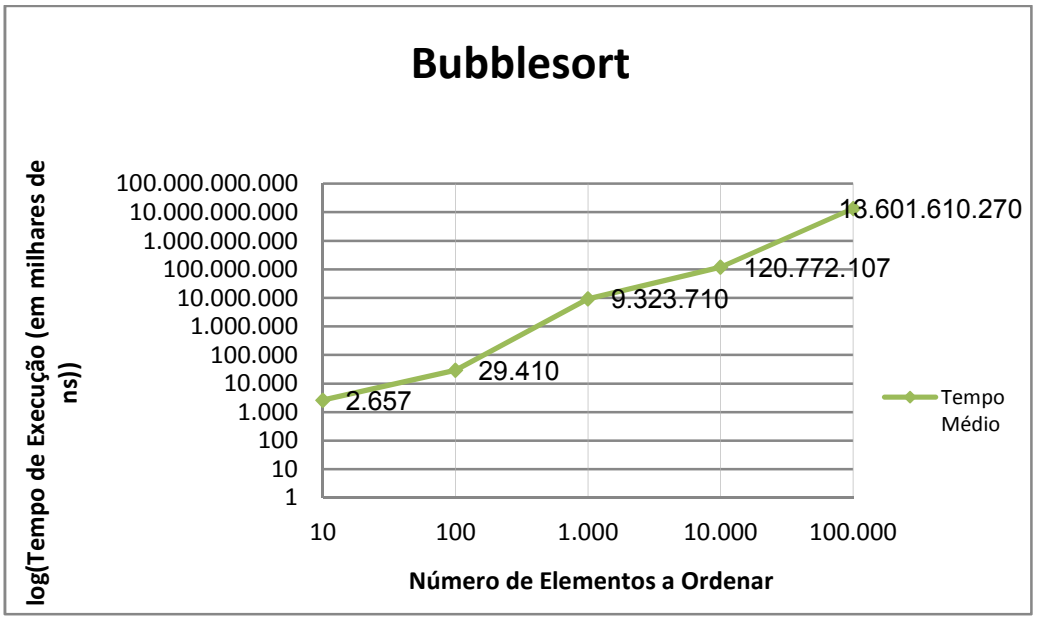

Gráfico 2 – Evolução do Tempo de Execução Médio do Algoritmo Bubblesort à medida que o número de elementos a ordenar aumenta.

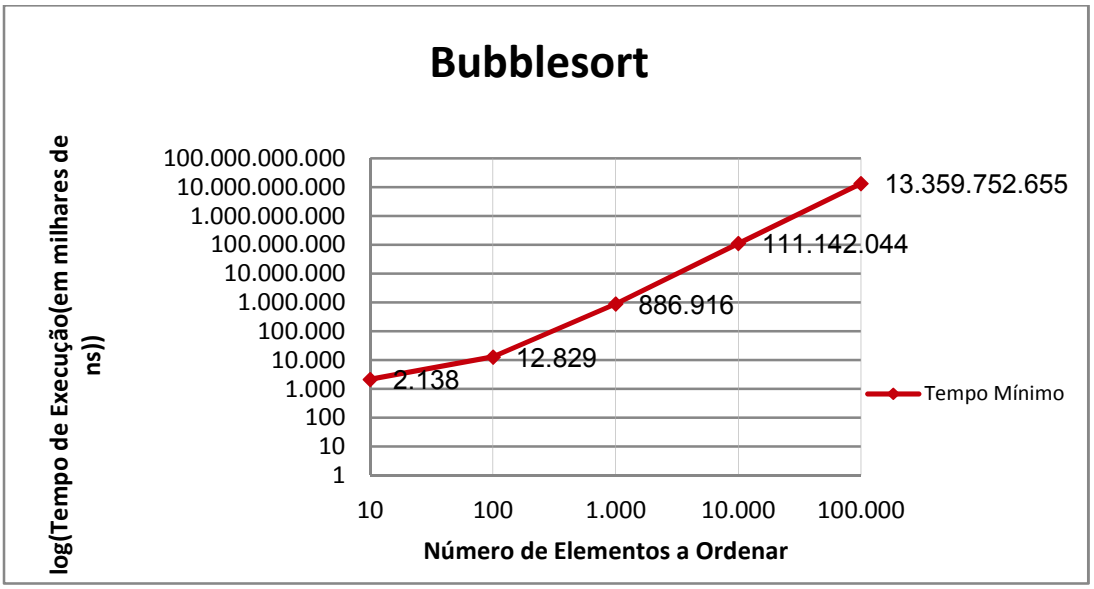

Gráfico 3 – Evolução do Tempo de Execução Mínimo do Algoritmo Bubblesort à medida que o número de elementos a ordenar aumenta.

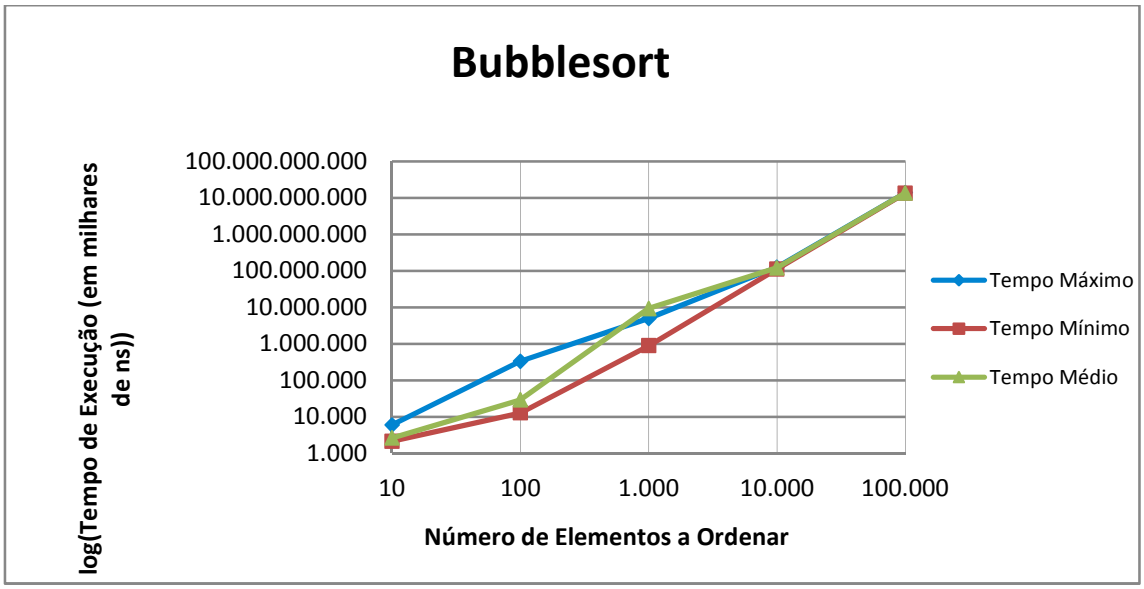

Gráfico 4 – Evolução dos vários tempos de execução à medida que o número de elementos a ordenar aumenta.

1.2 Heapsort

O *Heapsort* é um algoritmo de ordenação da família do algoritmo *Selectionsort*, sendo por isso um algoritmo que efectua a ordenação dos elementos de uma estrutura de dados utilizando uma estratégia "*comparison-based*". Este algoritmo é composto por dois passos: criar a *heap* e trocar o primeiro elemento da *heap* com o último, heapificando até que o *array* se encontre ordenado.

Durante o primeiro passo, os elementos do *array* são reordenados numa *heap*. Uma *heap* é uma estrutura de dados semelhante a uma árvore binária que é completa para todos os seus níveis, excepto para o nível cuja profundidade da árvore é máxima. A reordenação faz-se heapificando de baixo para cima, ou seja, troca-se o pai com o maior dos filhos partindo das folhas até chegar à raiz. O processo continua até que o maior elemento do *array* esteja na raiz. Quando os dados se encontram reorganizados na *heap*, podemos verificar que o elemento que se encontra no nó pai tem um valor superior ao de ambos os elementos nos nós filhos (filho direito e filho esquerdo).

O segundo passo do processo de ordenação consiste em trocar o primeiro elemento da *heap* com o último elemento, remover este elemento da *heap* e colocar o mesmo no *array* ordenado e heapificar de cima para baixo (partindo da raiz até chegar às folhas) a *sub-heap*. O processo repete-se até que já não existam elementos na *heap*.

A complexidade temporal do algoritmo *Heapsort*, no melhor e no pior caso, é O(N log N)[7].

A implementação utilizada durante os testes deste algoritmo é semelhante à fornecida nas aulas teóricas, salvo pequenas alterações.

Heapsort			
N	**Tempo Máximo(ns)**	**Tempo Mínimo(ns)**	**Tempo Médio(ns)**
10	8211	4618	5204
100	97498	12828	35679
1000	1166383	77485	109745
10000	4568035	1080688	1154224
100000	20430438	14108979	14530016

Quadro 2 – Valores dos tempos de execução máximos, mínimos e médios do algoritmo para os vários valores de N.

[7] – Citando o Artigo da Wikipedia "*Heapsort*" (ver **Bibliografia**)

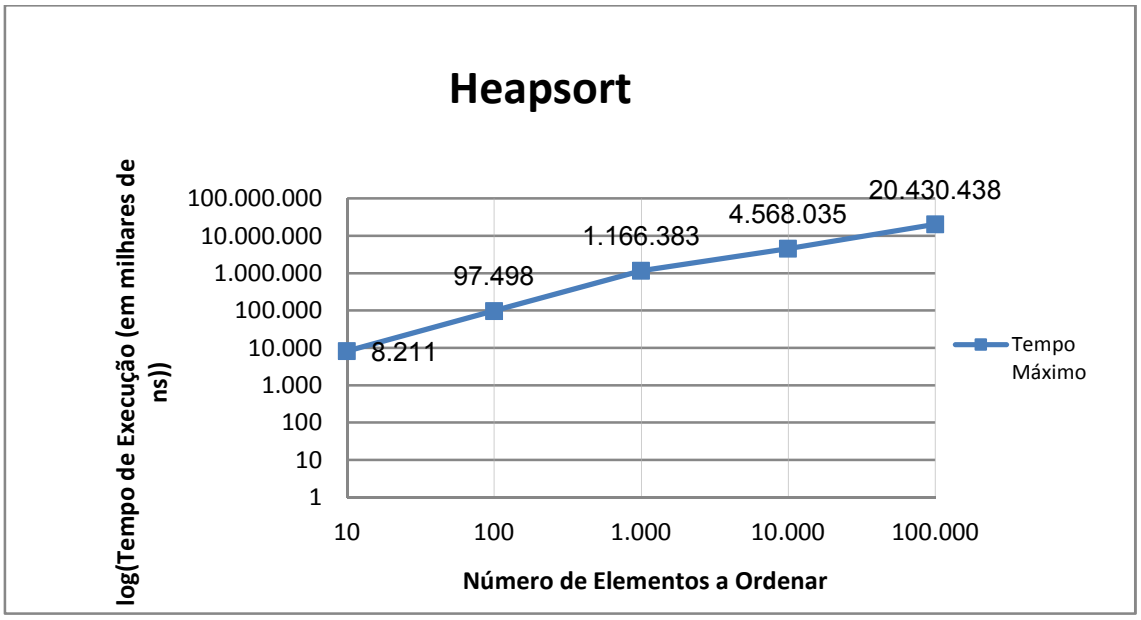

Gráfico 5 – Evolução do Tempo de Execução Máximo do Algoritmo Heapsort à medida que o número de elementos a ordenar aumenta.

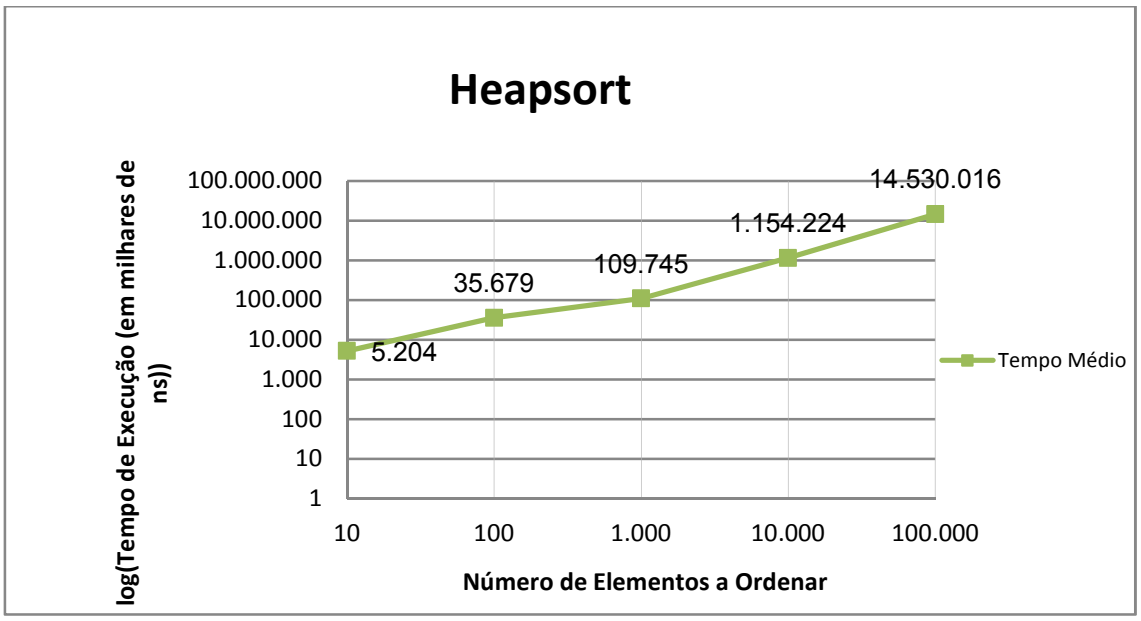

Gráfico 6 – Evolução do Tempo de Execução Médio do Algoritmo Heapsort à medida que o número de elementos a ordenar aumenta.

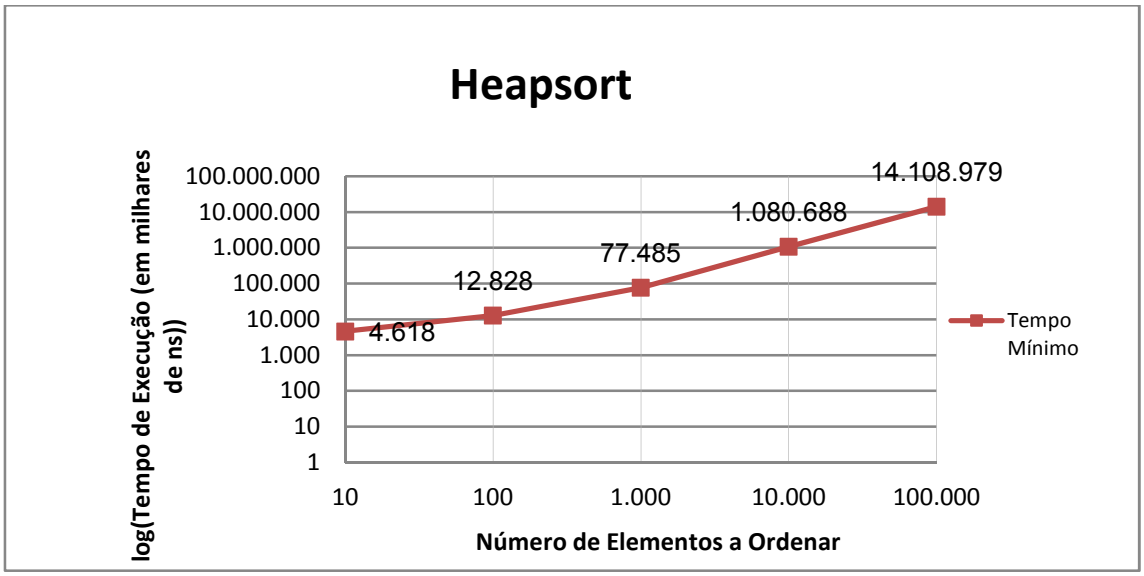

Gráfico 7 – Evolução do Tempo de Execução Mínimo do Algoritmo Heapsort à medida que o número de elementos a ordenar aumenta.

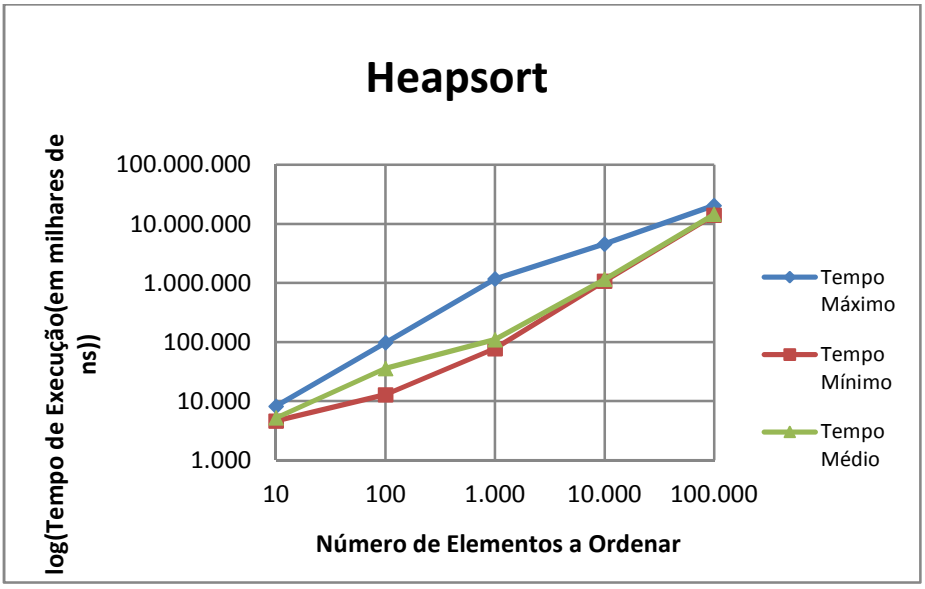

Gráfico 8 – Evolução dos vários tempos de execução à medida que o número de elementos a ordenar aumenta.

1.3 Insertion sort

O *Insertion sort* é um algoritmo de ordenação simples que permite ordenar os elementos de uma dada estrutura de dados – que pode ser um *array* ou uma lista. Este é o algoritmo utilizado quando, por exemplo, se pretendem ordenar cartas de um baralho ou testes de alunos.

Durante o processo de ordenação considera-se um item da estrutura de dados de cada vez, sendo esse item inserido no local apropriado entre os elementos da estrutura que já foram considerados. Os elementos à esquerda do elemento considerado já se encontram parcialmente ordenados, porém não se encontram nas suas posições finais. Estes elementos serão movidos à medida que se ordenam os restantes elementos – inserindo-se os elementos a ordenar nas posições adequadas entre os elementos parcialmente ordenados. O processo de ordenação termina quando o índice do elemento considerado é igual ao do elemento mais à direita no array, ou seja, do último elemento do *array*.

No pior dos casos, este algoritmo tem complexidade temporal $O(N^2)$[8].

No melhor dos casos, a complexidade temporal deste algoritmo é $O(N)$[8].

A implementação utilizada durante os testes deste algoritmo é semelhante à fornecida na aula teórica, salvo pequenas alterações.

Insertion sort			
N	**Tempo Máximo(ns)**	**Tempo Mínimo(ns)**	**Tempo Médio(ns)**
10	2138	855	1239
100	109047	2138	23127
1000	4296882	155232	171145
10000	16278801	12286396	15927633
100000	1602126204	1194676365	1579672271

Quadro 3 – Valores dos tempos de execução máximos, mínimos e médios do algoritmo para os vários valores de N.

[8] – De acordo com os slides das Aulas Teóricas – Aula 10 (Slide 33) (ver **Bibliografia**)

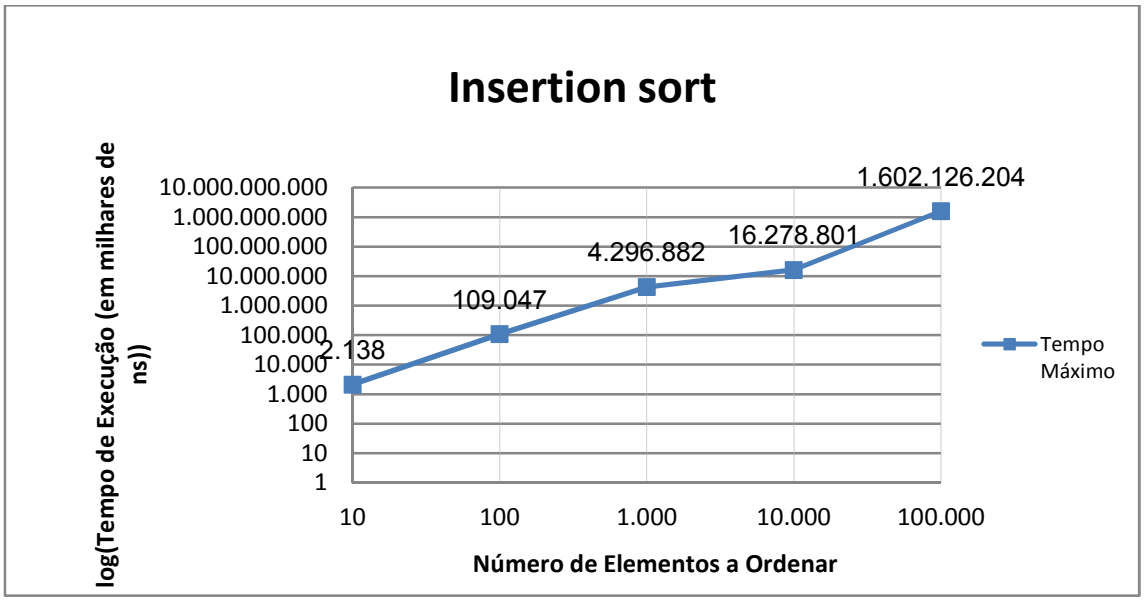

Gráfico 9 – Evolução do Tempo de Execução Máximo do Algoritmo Insertion sort à medida que o número de elementos a ordenar aumenta.

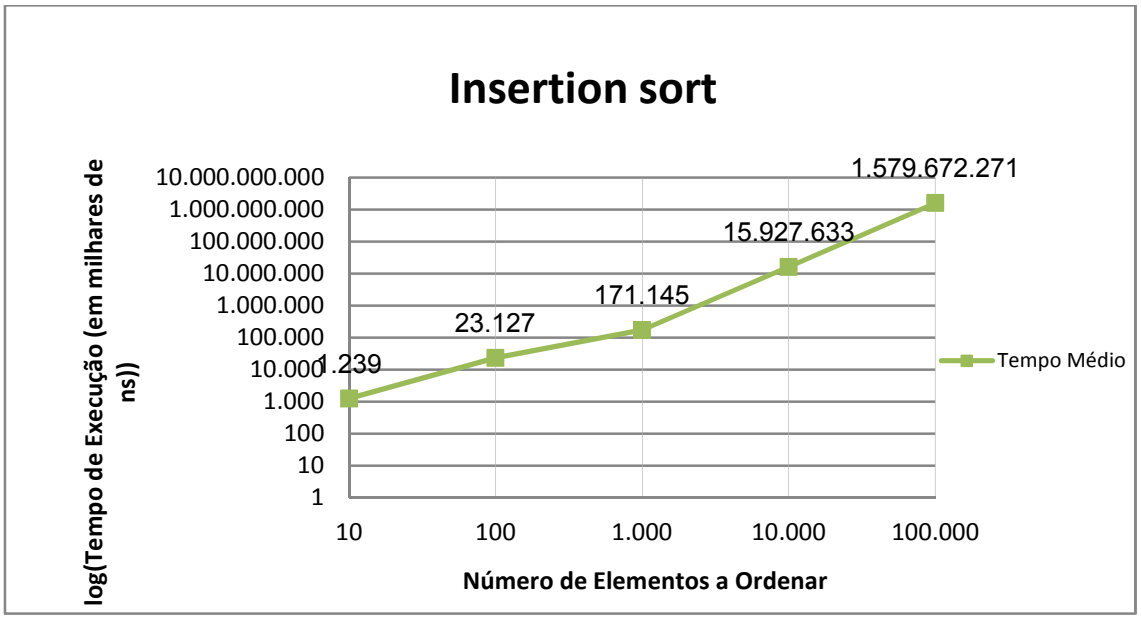

Gráfico 10 – Evolução do Tempo de Execução Médio do Algoritmo Insertion sort à medida que o número de elementos a ordenar aumenta.

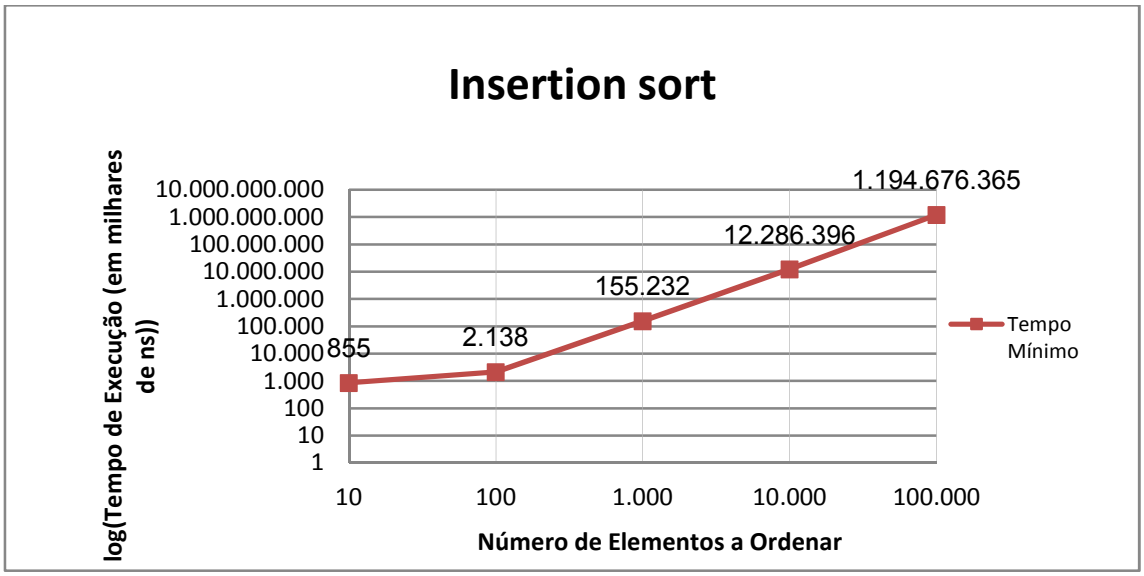

Gráfico 11 – Evolução do Tempo de Execução Mínimo do Algoritmo Insertion sort à medida que o número de elementos a ordenar aumenta.

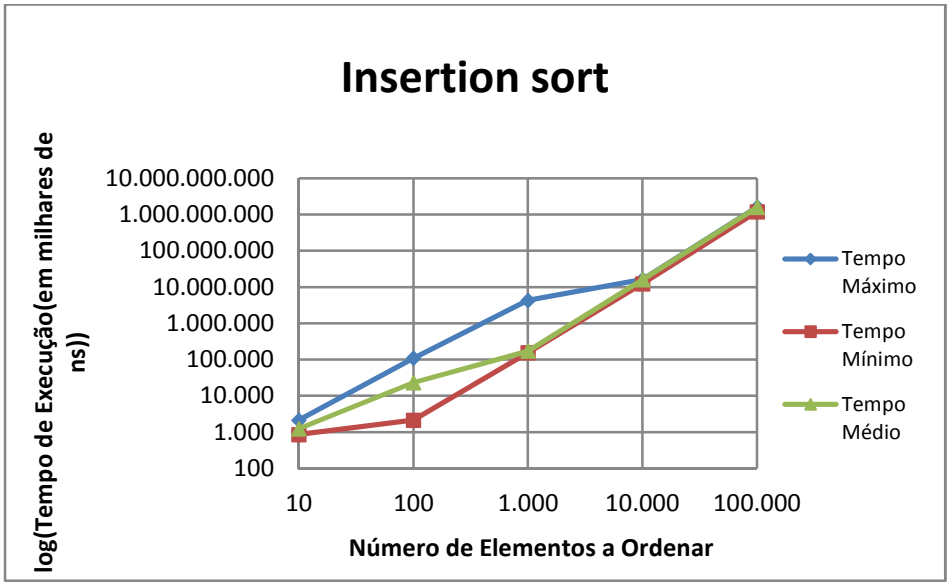

Gráfico 12 – Evolução dos vários tempos de execução à medida que o número de elementos a ordenar aumenta.

1.4 Merge sort

Um dos primeiros algoritmos não triviais a ser implementado num computador, o *Merge sort* – criado por Jon von Neumann[9], em 1945 - é um algoritmo de ordenação que utiliza a operação união (*merging*). Esta operação permite combinar dois *arrays* ordenados num array maior, que também estará ordenado. Este algoritmo utiliza um método "*divide-and-conquer*" para ordenar os elementos de uma estrutura de dados.

Utilizando o algoritmo *Merge sort* para ordenar um *array*, começa-se por dividir o mesmo em duas metades. De seguida ordenam-se recursivamente ambas as metades do *array*. Quando a ordenação de ambas as metades do *array* inicial for concluída, ambas as partes são fundidas, sendo o resultado obtido correspondente ao *array* inicial ordenado. A ordenação das metades do *array* inicial é também efectuada utilizando este algoritmo.

Existem duas implementações possíveis deste algoritmo:

- **Top-Down:** corresponde à implementação normalmente utilizada. O processo de ordenação é o descrito anteriormente.

- **Bottom-Up:** a diferença entre esta implementação e a anterior é que efectua todas as fusões de *sub-arrays* mais pequenos – começando em *sub-arrays* de um elemento - numa passagem, efectuando posteriormente uma segunda passagem para que seja possível juntar estes *sub-arrays* em pares. Este processo continua até que seja efectuada uma fusão que compreenda todo o *array*. Todas as fusões irão envolver *sub-arrays* de tamanhos semelhantes, sendo duplicado o tamanho do *sub-array* na iteração seguinte, dado que passa a compreender os elementos de um par de *sub-arrays* anteriores.

A complexidade temporal do algoritmo, no melhor e no pior caso, é O(N log N)[10].

A implementação utilizada durante os testes deste algoritmo é semelhante à fornecida nas aulas teóricas.

Merge sort			
N	**Tempo Máximo(ns)**	**Tempo Mínimo(ns)**	**Tempo Médio(ns)**
10	14540	4276	5074
100	188159	6842	71170
1000	1644686	78257	120052
10000	8805871	1084911	1138356
100000	29755307	13617626	13784198

Quadro 4 – Valores dos tempos de execução máximos, mínimos e médios do algoritmo para os vários valores de N.

---

[9] – "Donald Knuth cites von Neumann as the inventor, in 1945, of the merge sort algorithm" Fonte: Artigo da Wikipedia "*John von Neumann*" (ver **Bibliografia**)

[10] – Citando o Artigo da Wikipedia "*Merge sort*" (ver **Bibliografia**)

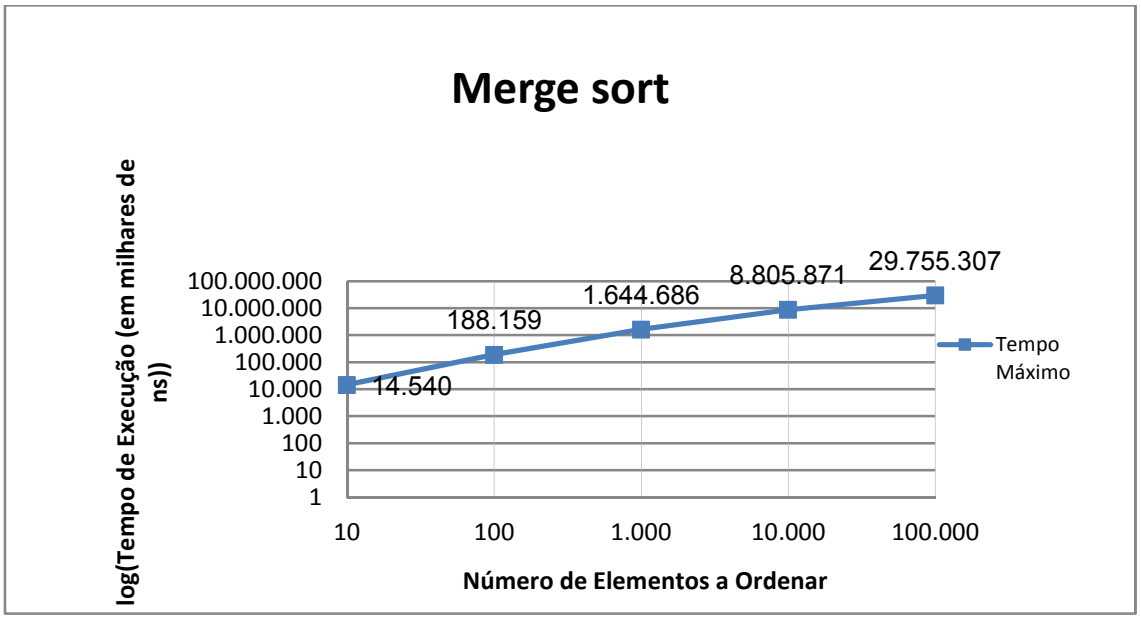

Gráfico 13 – Evolução do Tempo de Execução Máximo do Algoritmo Merge sort à medida que o número de elementos a ordenar aumenta.

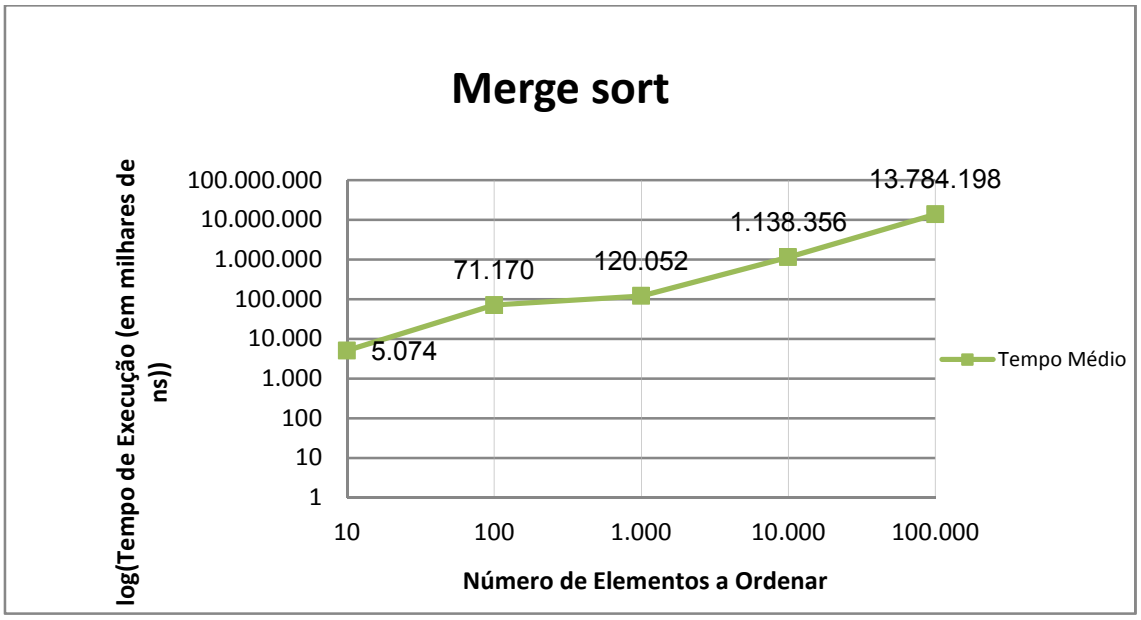

Gráfico 14 – Evolução do Tempo de Execução Médio do Algoritmo Merge sort à medida que o número de elementos a ordenar aumenta.

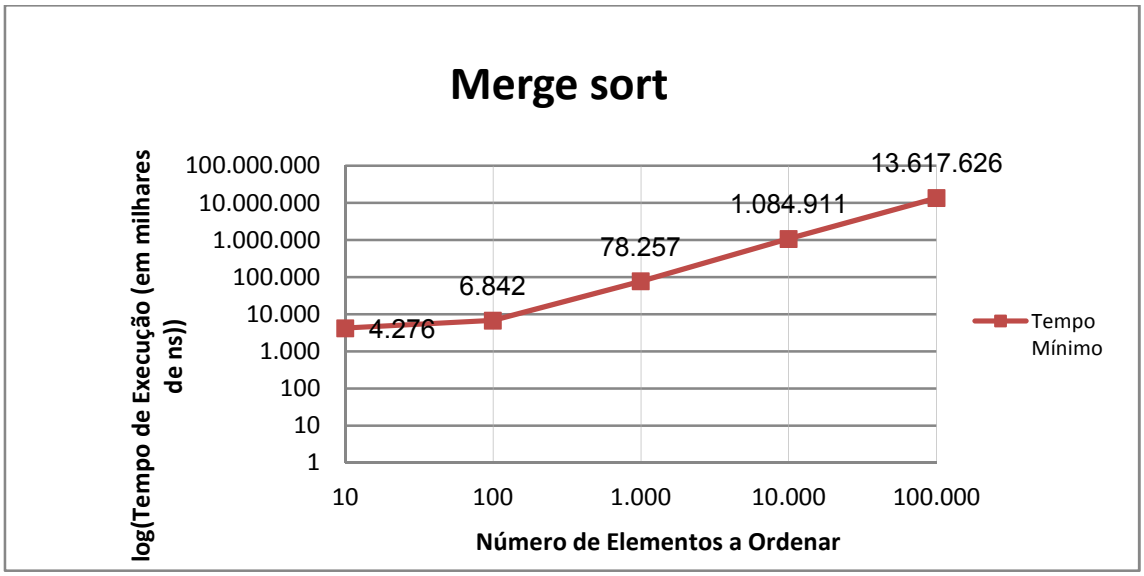

Gráfico 15 – Evolução do Tempo de Execução Mínimo do Algoritmo Merge sort à medida que o número de elementos a ordenar aumenta.

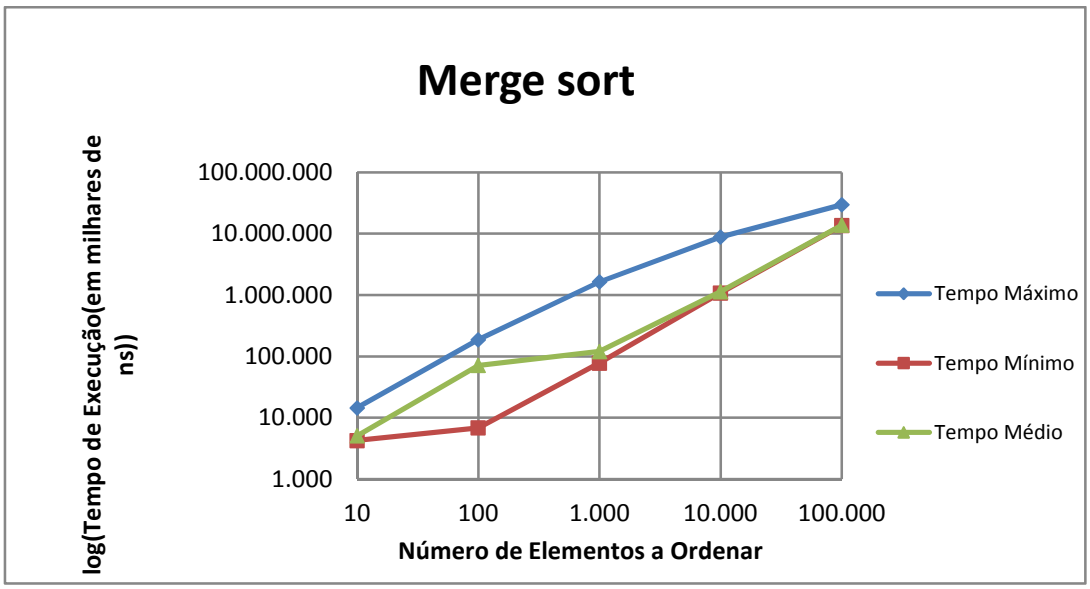

Gráfico 16 – Evolução dos vários tempos de execução à medida que o número de elementos a ordenar aumenta.

1.5 Quicksort

O *Quicksort* é um algoritmo de ordenação rápido e bastante popular descoberto em 1962 por C.A.R. Hoare. Este algoritmo utiliza um método "*divide-and-conquer*" para ordenar os elementos de uma estrutura de dados. Os principais pontos fortes deste algoritmo são a sua rapidez e a utilização de uma pequena *stack* auxiliar durante o processo de ordenação. A desvantagem principal está relacionada com a implementação do algoritmo, que deve ser efectuada de um modo cuidadoso, de modo a que seja possível evitar problemas de má performance.

O *Quicksort* funciona do seguinte modo:

- Escolhe-se um elemento – pivô;
- Ao efectuar esta escolha, "parte-se" o *array* a ordenar em dois *sub-arrays*;
- Compara-se o primeiro elemento do *sub-array* da esquerda e o último elemento do *sub-array* da direita com o pivô;
- Caso o elemento do *array* da esquerda seja maior que o pivô e o elemento do *array* da direita seja menor que o pivô, trocam-se estes elementos.

A ordenação dos *sub-arrays* efectua-se utilizando o algoritmo recursivamente.

A parte crucial deste processo de ordenação prende-se com o procedimento em que se escolhe o pivô e se "parte" o *array*. Este processo altera a ordem dos elementos do *array* de modo a que: nenhum elemento do *sub-array* da esquerda é menor que o pivô, o pivô encontra-se na sua posição final e nenhum elemento do *sub-array* da direita é menor que o pivô. Se isto se verificar, o *array* encontra-se ordenado. É também este processo o principal factor que pode condicionar a eficiência do algoritmo, tornando-o mais eficiente se as partições forem equilibradas ou menos eficiente caso as partições sejam desequilibradas (sendo o pior caso aquele em que existe um elemento numa partição e N-1 elementos na outra).

A complexidade temporal do *Quicksort*, no pior caso, é $O(N^2)$[11]. A complexidade temporal do *Quicksort*, no melhor caso, é $O(N \log N)$ [11].

A implementação utilizada durante os testes deste algoritmo é semelhante à que foi fornecida nas aulas teóricas, salvo pequenas alterações.

Quicksort			
N	**Tempo Máximo(ns)**	**Tempo Mínimo(ns)**	**Tempo Médio(ns)**
10	16250	2565	3469
100	112040	11118	24453
1000	400266	71415	90737
10000	3574605	853988	900801
100000	19944920	10101606	10338258

Quadro 5 – Valores dos tempos de execução máximos, mínimos e médios do algoritmo para os vários valores de N.

[11] – Citando o Artigo da Wikipedia "*Quicksort*" (ver **Bibliografia**)

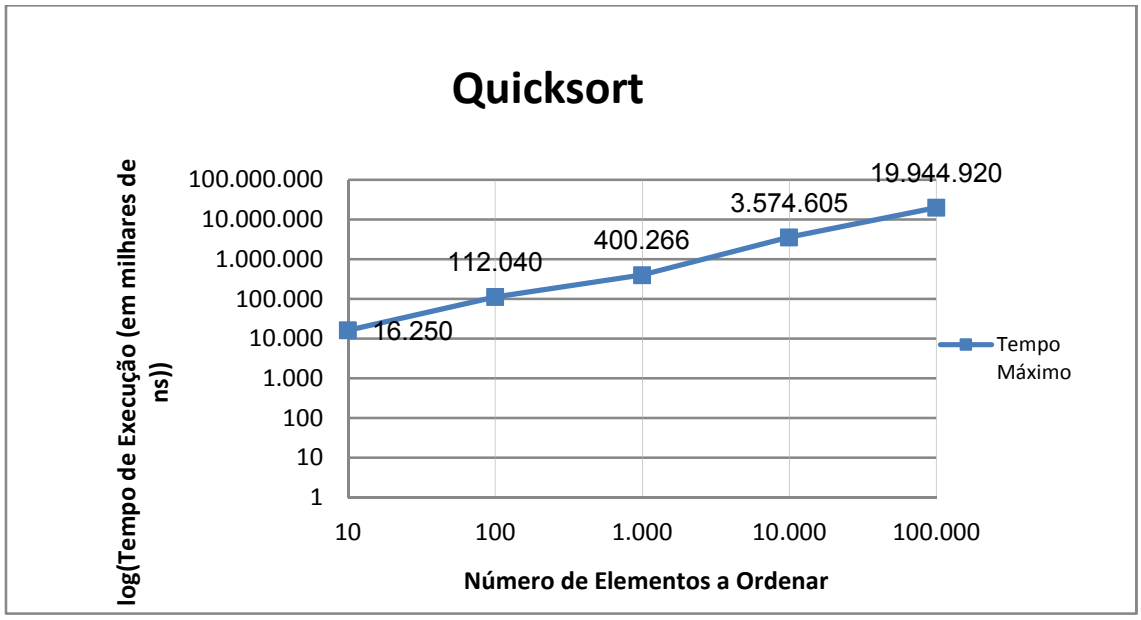

Gráfico 17 – Evolução do Tempo de Execução Máximo do Algoritmo Quicksort à medida que o número de elementos a ordenar aumenta.

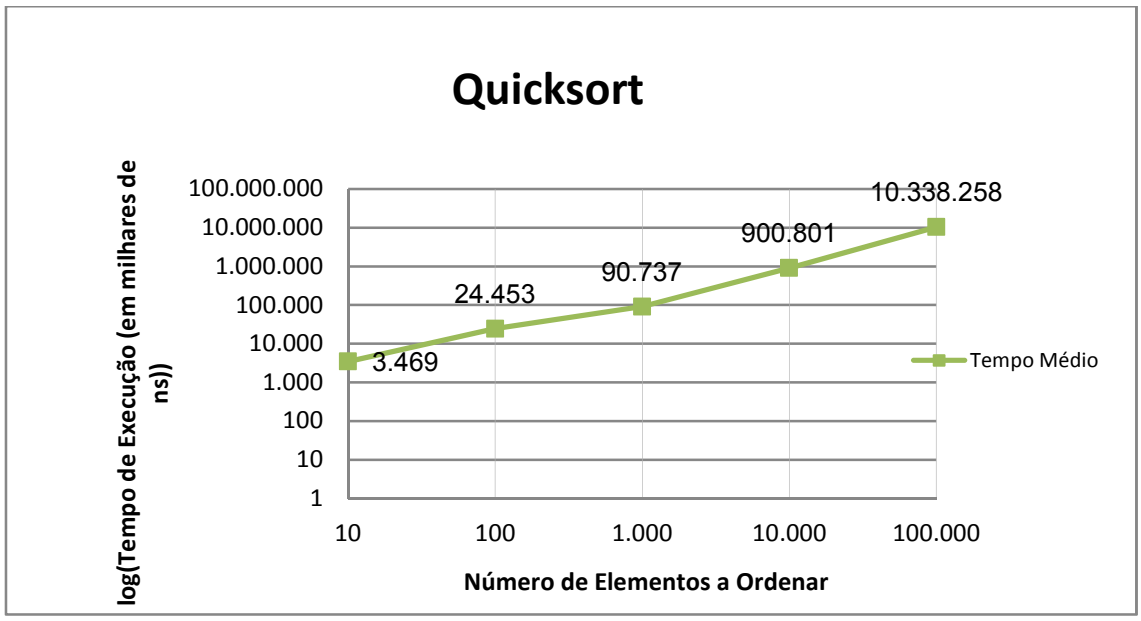

Gráfico 18 – Evolução do Tempo de Execução Médio do Algoritmo Quicksort à medida que o número de elementos a ordenar aumenta.

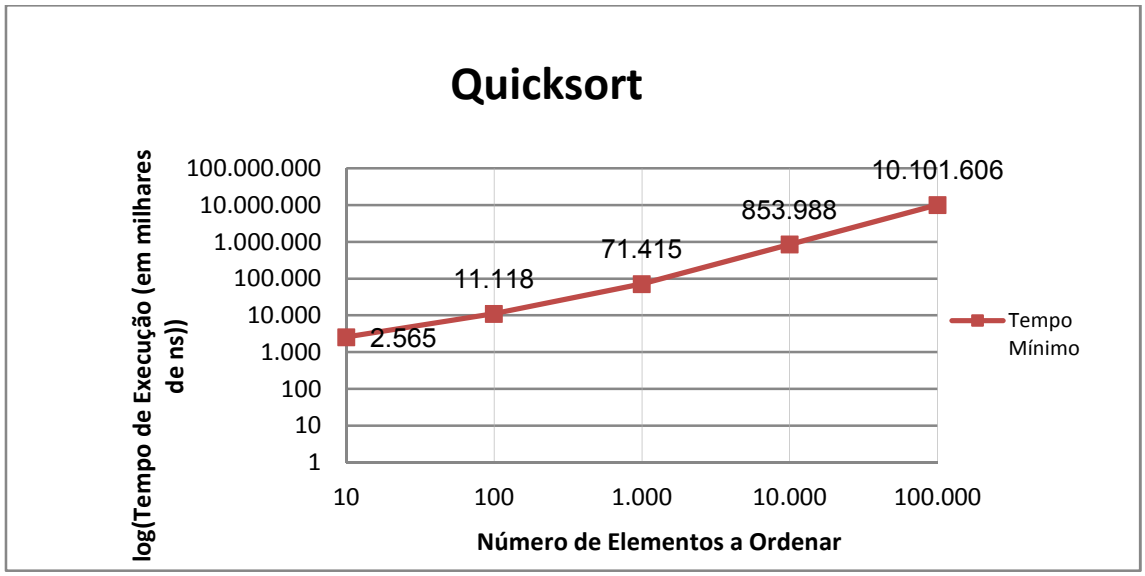

Gráfico 19 – Evolução do Tempo de Execução Mínimo do Algoritmo Quicksort à medida que o número de elementos a ordenar aumenta.

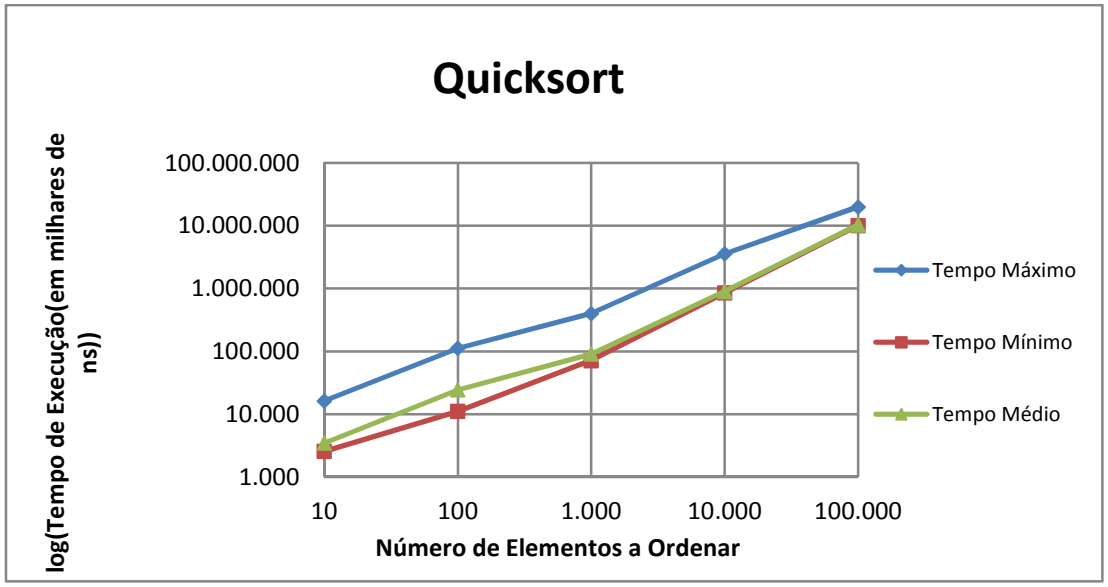

Gráfico 20 – Evolução dos vários tempos de execução à medida que o número de elementos a ordenar aumenta.

1.6 Radix sort

O *Radix sort* é um algoritmo de ordenação que não utiliza comparações. Este algoritmo decompõe cada elemento a ordenar de modo a obter os seus dígitos e ordena o *array* com base no valor dos dígitos de cada um dos elementos.

O processo de ordenação envolve examinar cada dígito separadamente, começando pelo menos significativo. Este processo inicia-se com a divisão dos elementos em dez grupos de acordo com o valor do dígito correspondente às unidades (de 0 a 9). De seguida estes grupos são de novo unificados de modo a que os elementos se encontrem por ordem crescente de acordo com o valor do seu dígito correspondente às unidades. Seguidamente os itens são novamente distribuídos por dez grupos, desta vez de acordo com o valor do algarismo correspondente às dezenas e repete-se o processo de unificação, tal como tinha acontecido anteriormente. Este processo (ordenar por grupos de acordo com o valor de um dado dígito e reunificação) repete-se tantas vezes quantas as necessárias consoante o número com o maior número de dígitos (exemplo: se o número com mais dígitos do array tem 10 dígitos, então repete-se o processo 10 vezes). O processo de divisão dos itens por grupos deve ocorrer sem prejuízo das ordenações anteriores.

A complexidade deste algoritmo, no melhor caso, é O(N) [12].

A complexidade deste algoritmo, no pior caso, é O(N log N) [13].

A implementação deste algoritmo baseia-se na descrição do mesmo presente no livro "*Data Structures and Algorithms in Java*" e no vídeo "*Radix Sort Tutorial*" [14].

Radix sort			
N	**Tempo Máximo(ns)**	**Tempo Mínimo(ns)**	**Tempo Médio(ns)**
10	357931	88093	123857
100	1039582	60296	254643
1000	7114575	348950	567079
10000	31104510	8287152	8792687
100000	654496440	585939167	597058441

Quadro 6 – Valores dos tempos de execução máximos, mínimos e médios do algoritmo para os vários valores de N.

---

[12] – Citando o Artigo da Wikipedia "*Radix sort*" (ver **Bibliografia**)

[13] – (Lafore, 2003; Página: 358)

[14] – Fonte do video.

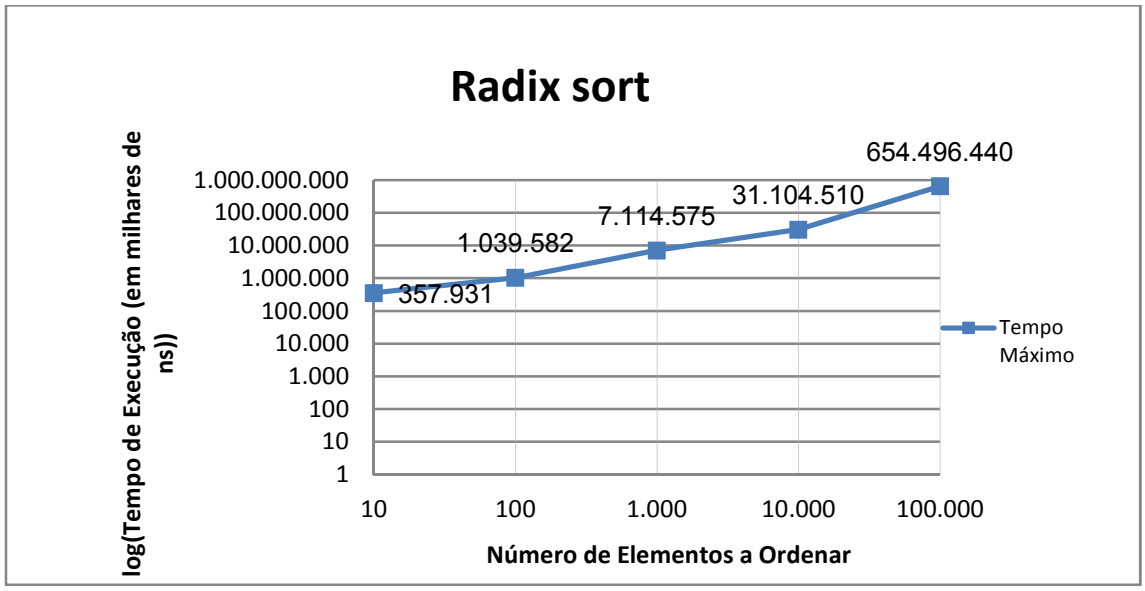

Gráfico 21 – Evolução do Tempo de Execução Máximo do Algoritmo Radix sort à medida que o número de elementos a ordenar aumenta.

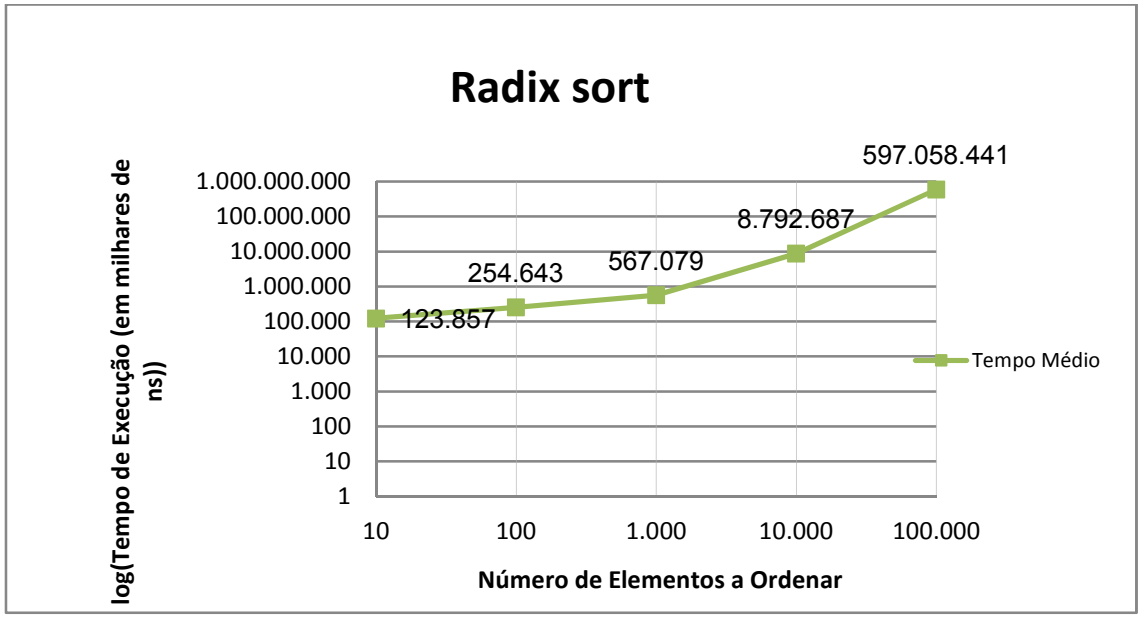

Gráfico 22 – Evolução do Tempo de Execução Médio do Algoritmo Radix sort à medida que o número de elementos a ordenar aumenta.

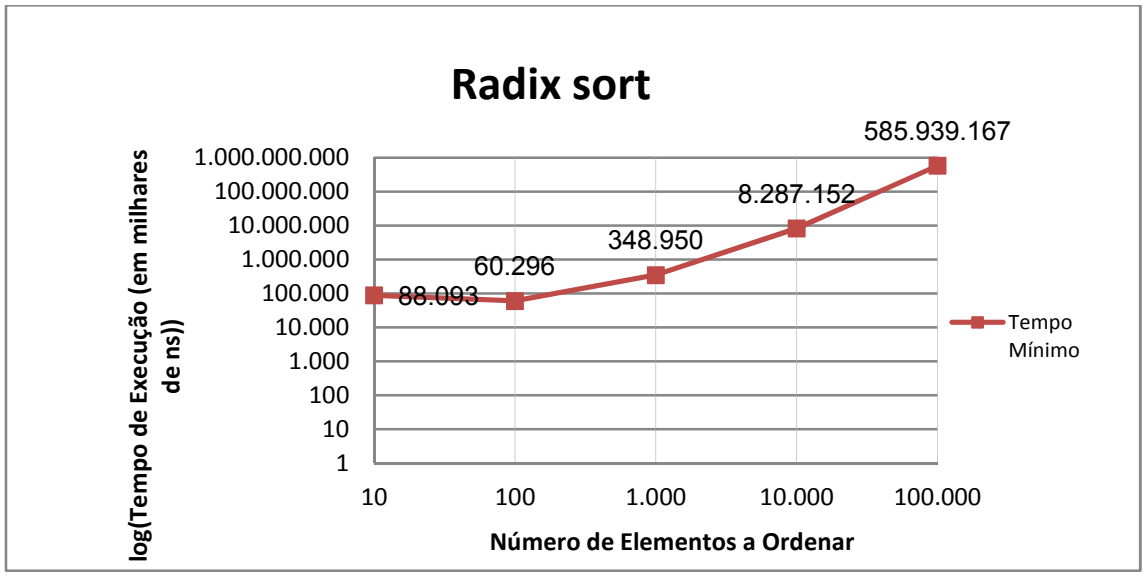

Gráfico 22 – Evolução do Tempo de Execução Mínimo do Algoritmo Radix sort à medida que o número de elementos a ordenar aumenta.

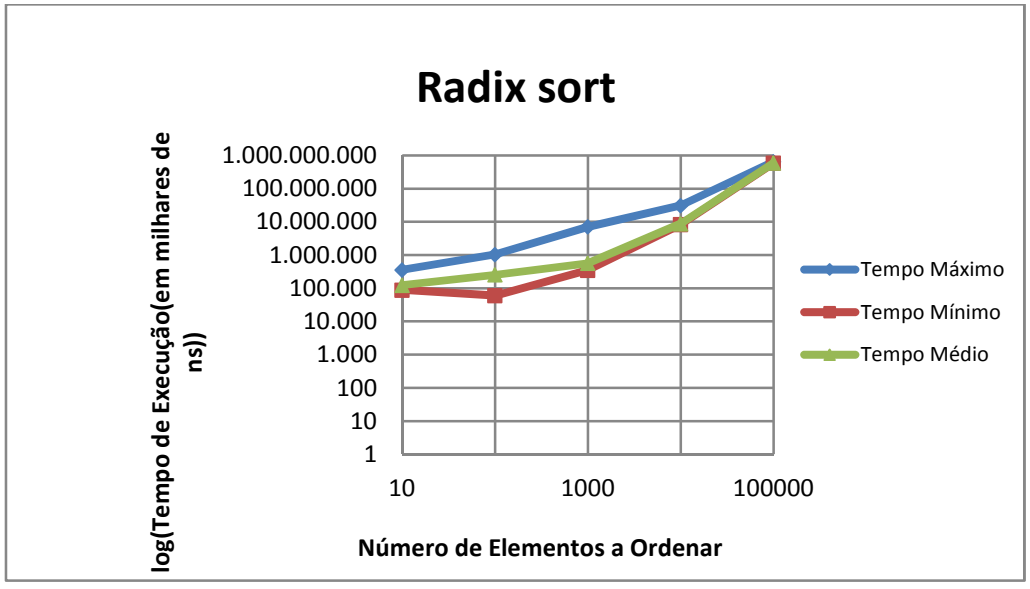

Gráfico 23 – Evolução dos vários tempos de execução à medida que o número de elementos a ordenar aumenta.

1.7 Rank sort

O *Rank sort* é outro algoritmo de ordenação relativamente simples.

O processo de ordenação ocorre do seguinte modo: selecciona-se cada um dos elementos do *array*, contando o número de elementos do *array* que são menores que o elemento considerado. A este valor chama-se *rank*, sendo também o indicador da posição do elemento no *array* ordenado. Colocam-se os elementos no *array* ordenado com as posições dos mesmos de acordo com o seu *rank*.

A complexidade temporal deste algoritmo é, no melhor e no pior caso, $O(N^2)$[15].

A implementação deste algoritmo baseou-se na descrição do algoritmo e é semelhante à implementação apresentada no slide 422 da disciplina de "*Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*" [16].

Rank sort			
N	**Tempo Máximo(ns)**	**Tempo Mínimo(ns)**	**Tempo Médio(ns)**
10	10263	2565	2962
100	283094	8125	27277
1000	5971077	809086	850798
10000	86847679	80530645	82709387
100000	8362727016	8144588334	8260104667

Quadro 7 – Valores dos tempos de execução máximos, mínimos e médios do algoritmo para os vários valores de N.

---

[15] – "Overall sequential sorting time complexity of $O(n^2)$" Fonte: "*Slides for Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*".

[16] – "*Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers 2nd ed*" (ver **Bibliografia**)

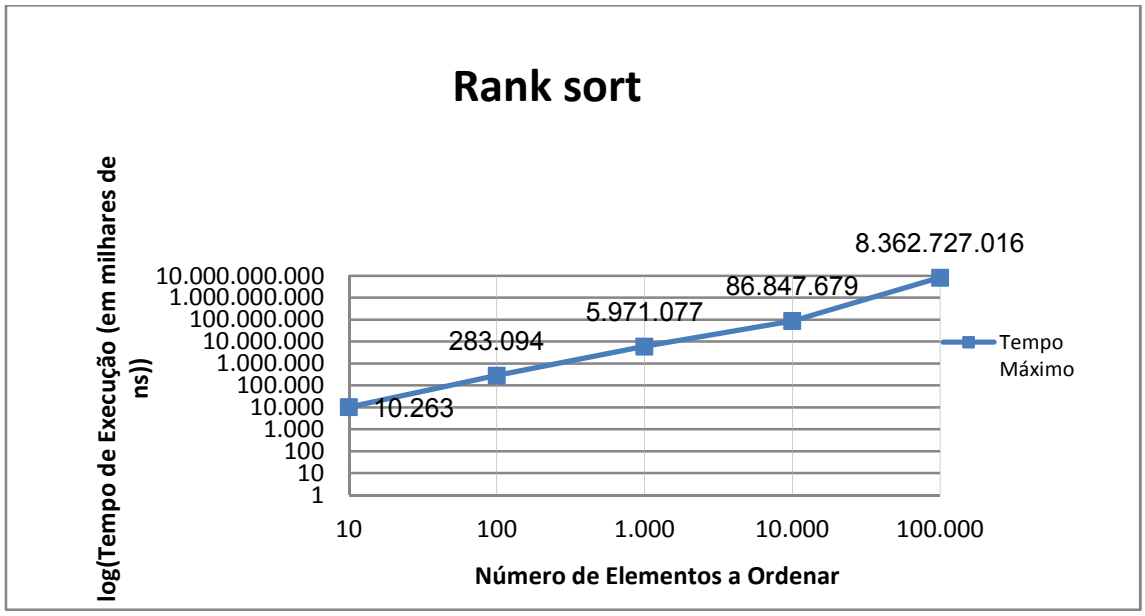

Gráfico 24 – Evolução do Tempo de Execução Máximo do Algoritmo Rank sort à medida que o número de elementos a ordenar aumenta.

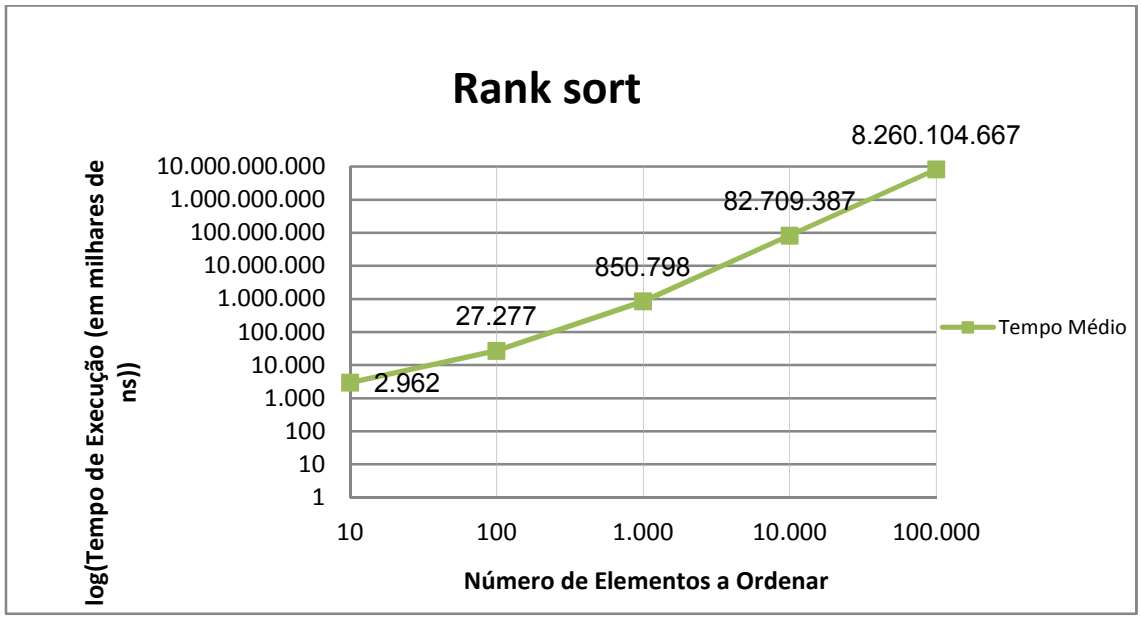

Gráfico 25 – Evolução do Tempo de Execução Médio do Algoritmo Radix sort à medida que o número de elementos a ordenar aumenta.

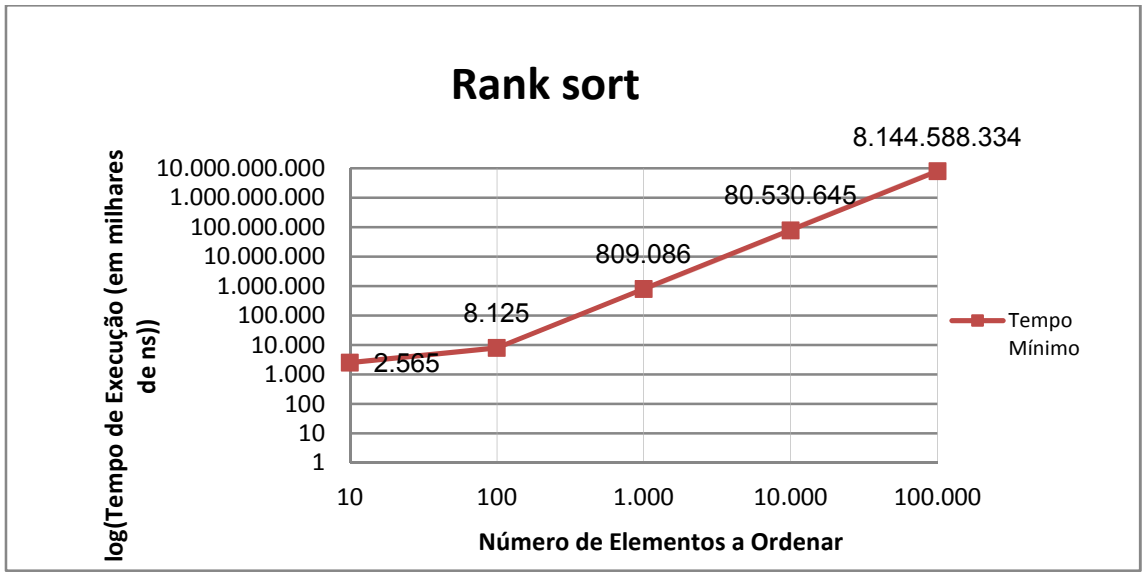

Gráfico 26 – Evolução do Tempo de Execução Mínimo do Algoritmo Radix sort à medida que o número de elementos a ordenar aumenta.

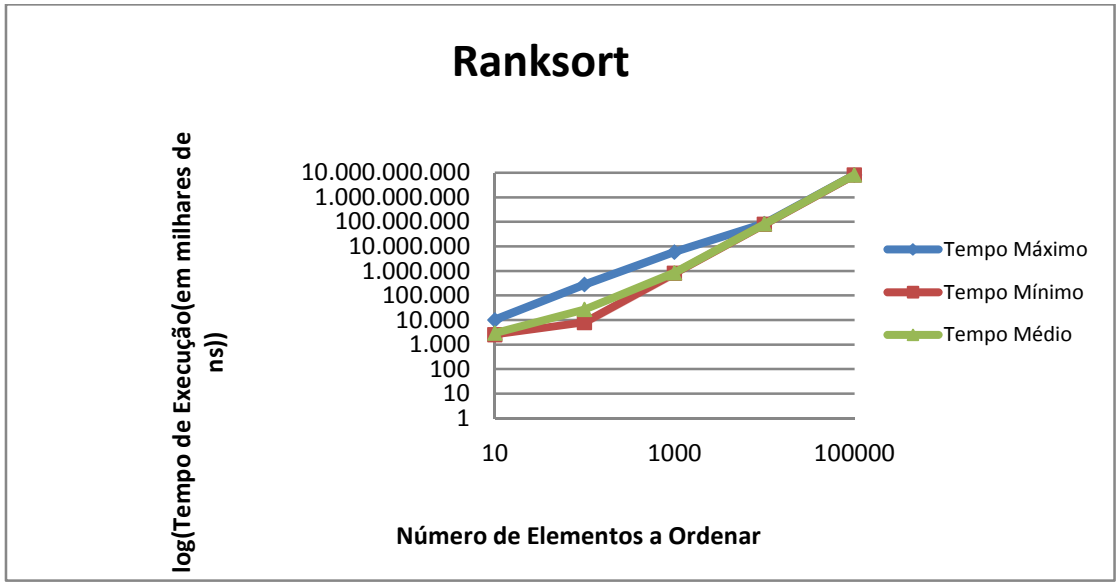

Gráfico 27 – Evolução dos vários tempos de execução à medida que o número de elementos a ordenar aumenta.

1.8 Selection sort

O *Selection sort* é outro algoritmo de ordenação mais simples.

O algoritmo começa por encontrar o elemento do *array* cujo valor é menor e troca esse elemento com o primeiro elemento do *array*. De seguida, o algoritmo troca o segundo elemento do *array* com o elemento do *array* cujo valor é o segundo menor. Este processo continua até que todos os elementos do *array* estejam ordenados. O número de trocas é N (número de elementos do *array*), uma vez que os valores são colocados nas suas posições finais. "*O tempo de execução é afectado pelo número de trocas efectuadas*" [17].

A complexidade temporal deste algoritmo é, no melhor e no pior caso, $O(N^2)$ [18].

A implementação deste algoritmo que foi utilizada baseia-se na descrição efectuada no livro "Algorithms" [19] e é muito semelhante à fornecida no mesmo.

Selection sort			
N	**Tempo Máximo(ns)**	**Tempo Mínimo(ns)**	**Tempo Médio(ns)**
10	73981	1710	2273
100	184739	7269	105730
1000	4919091	431056	454279
10000	45251963	40060895	40386945
100000	4058111390	3875909580	3935755251

Quadro 8 – Valores dos tempos de execução máximos, mínimos e médios do algoritmo para os vários valores de N.

[17] – Fonte: Artigo da Wikipedia "*Selection sort*" (ver **Bibliografia**)

[18] - "Thus, the running time is dominated by the number of compares." (Sedgewick e Wayne, 2011; Página: 248)

[19] – (Sedgewick e Wayne, 2011; Página: 249)

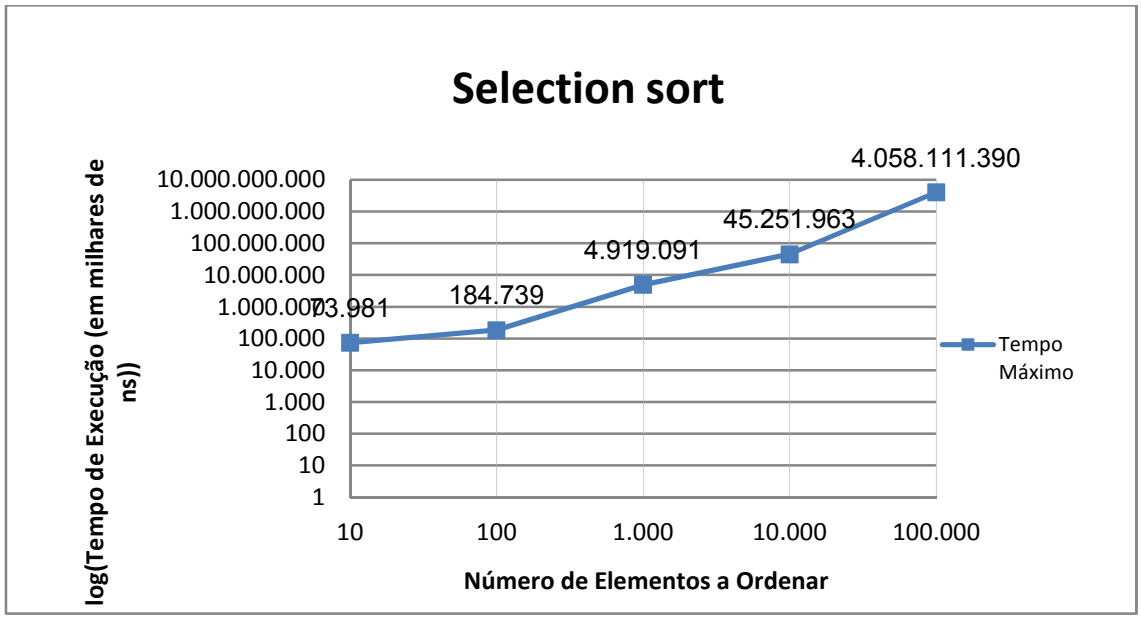

Gráfico 28 – Evolução do Tempo de Execução Máximo do Algoritmo Selection sort à medida que o número de elementos a ordenar aumenta.

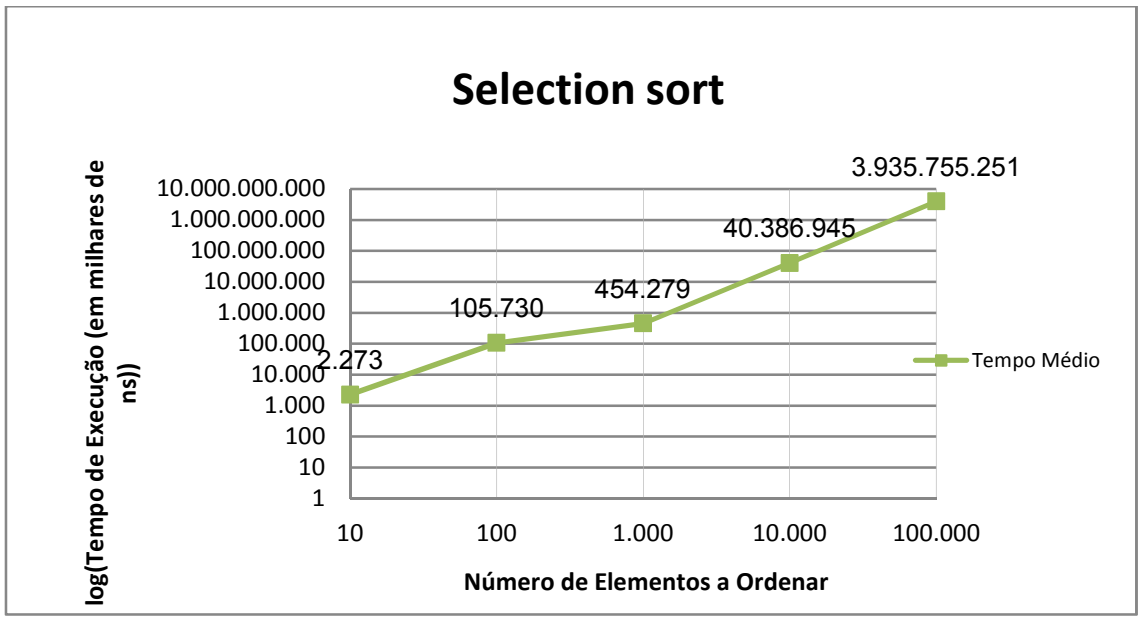

Gráfico 29 – Evolução do Tempo de Execução Médio do Algoritmo Selection sort à medida que o número de elementos a ordenar aumenta.

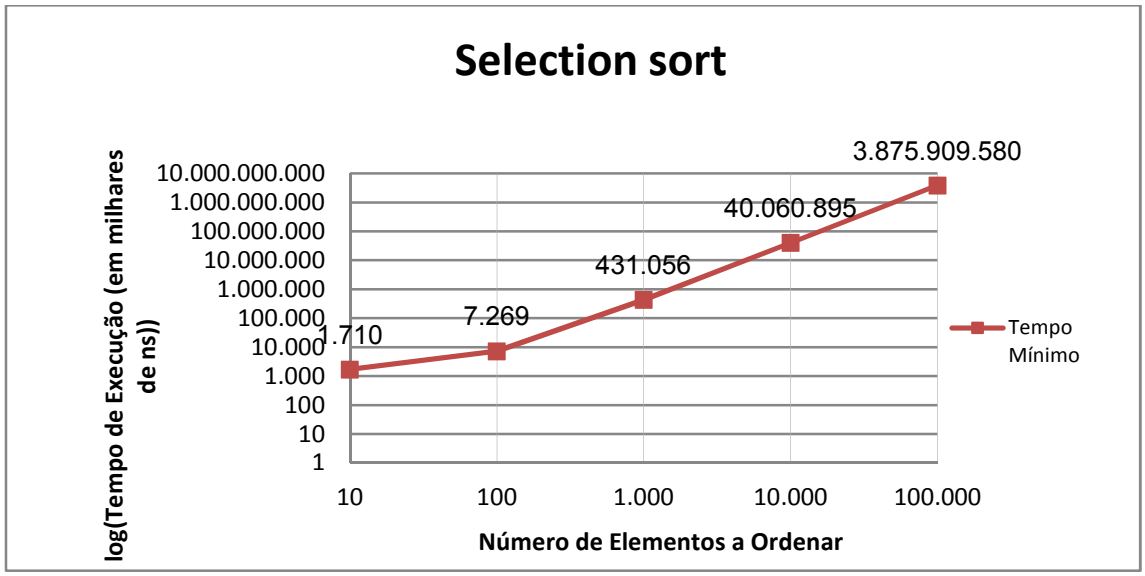

Gráfico 30 – Evolução do Tempo de Execução Mínimo do Algoritmo Selection sort à medida que o número de elementos a ordenar aumenta.

Gráfico 31 – Evolução dos vários tempos de execução à medida que o número de elementos a ordenar aumenta.

1.9 Shellsort

O *Shellsort* é um algoritmo de ordenação simples baseado no algoritmo *Insertion sort*. Concebido por Donald L. Shell, este algoritmo permite ordenar os elementos de uma estrutura de dados de um modo mais eficiente que o utilizado pelo *Insertion sort*, uma vez que permite que sejam trocados elementos da estrutura que se encontram distantes uns dos outros, criando subsequências de elementos ordenados. No início do processo de ordenação, estas subsequências são pequenas. À medida que o processo de ordenação avança, estas subsequências ficam parcialmente ordenadas. Parafraseando Sedgewick e Wayne, *a ideia fundamental consiste em modificar as posições dos elementos do array inicial de modo a que seja possível obter uma subsequência ordenada a partir de qualquer h-ésima entrada do array – começando em qualquer ponto do mesmo*[20]. Um *array* ordenado deste modo diz-se h-ordenado. Utilizando este método para grandes valores de h, torna-se possível simplificar a ordenação (utilizando o mesmo processo) com valores menores de h, uma vez que é possível mover por grandes distâncias itens do *array*.

O processo de ordenação utilizado pelo algoritmo *Shellsort* pode ser visto de um modo mais simples, uma vez que as subsequências são independentes. Assim, poderíamos utilizar o algoritmo *Insertion sort* com uma pequena modificação para obter um comportamento semelhante ao do *Shellsort*. Esta modificação passará por decrementar por h o valor correspondente à posição no *array* ao invés de decrementar por 1[21]. Dado que utiliza o *Insertion sort* para ordenar as subsequências de elementos, o algoritmo *Shellsort* pode ser visto como uma extensão do algoritmo *Insertion sort*.

A complexidade temporal do algoritmo, tendo por base experiências efectuadas, varia entre $O(N^{3/2})$ [22] e $O(N^{7/6})$ [22].

A implementação deste algoritmo teve como base o pseudo código presente no artigo da Wikipedia [23] sobre o Shellsort e no vídeo "*Shell-sort with Hungarian (Székely) folk dance*" [24].

Shellsort			
N	**Tempo Máximo(ns)**	**Tempo Mínimo(ns)**	**Tempo Médio(ns)**
10	11289	2052	2550
100	91341	4105	17001
1000	1291084	80051	93592
10000	7229763	1121745	1211079
100000	26112637	16729186	17596277

Quadro 9 – Valores dos tempos de execução máximos, mínimos e médios do algoritmo para os vários valores de N.

---

[20] – "*The idea is to rearrange the array to give it the property that taking every h-th entry (starting anywhere) yields a sorted subsequence.*" (Sedgewick e Wayne: 2011; Página: 258)
[21] - Esta variável corresponde à variável j no código do Insertion sort, disponível na classe Sort.java.
[22] – (LaFore: 2003; Página: 324)
[23] – (ver **Bibliografia**)
[24] – Fonte do vídeo.

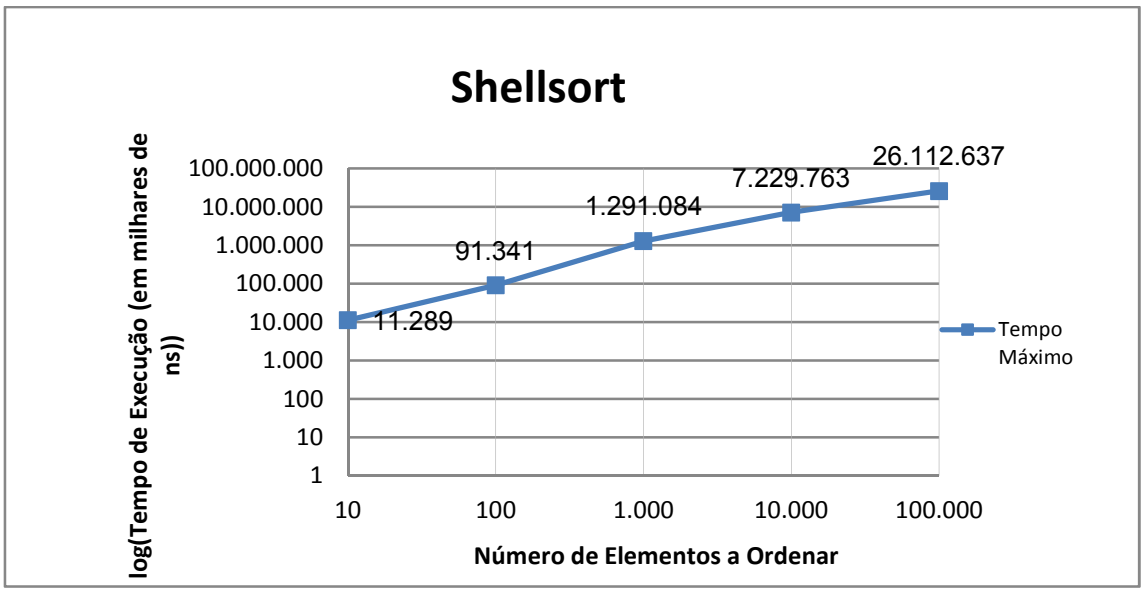

Gráfico 32 – Evolução do Tempo de Execução Máximo do Algoritmo Shellsort à medida que o número de elementos a ordenar aumenta.

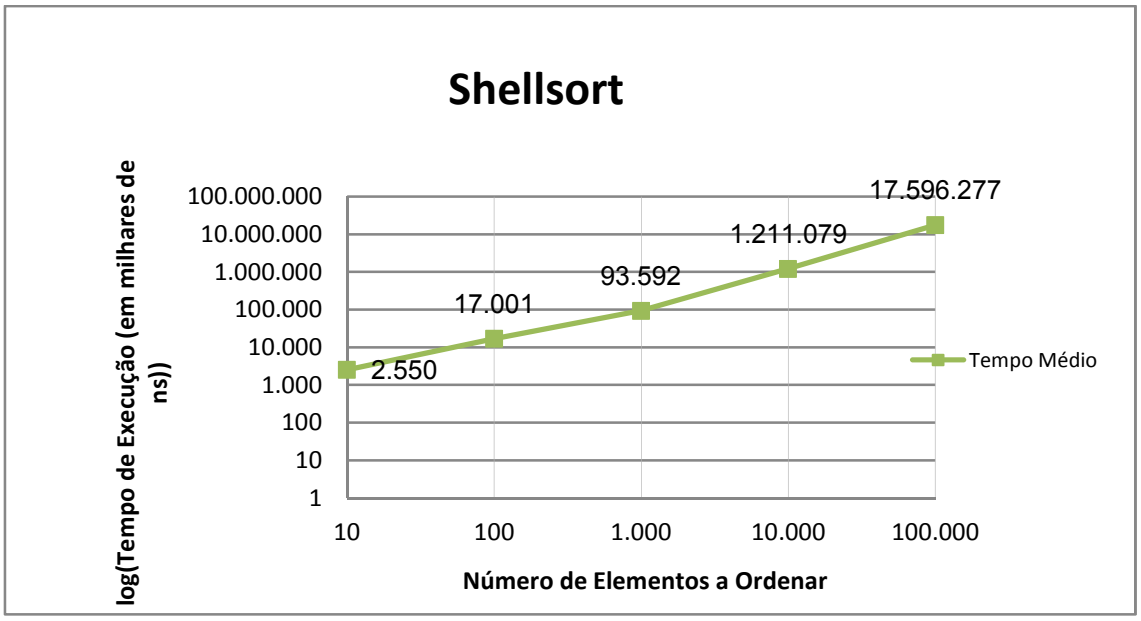

Gráfico 34 – Evolução do Tempo de Execução Médio do Algoritmo Shellsort à medida que o número de elementos a ordenar aumenta.

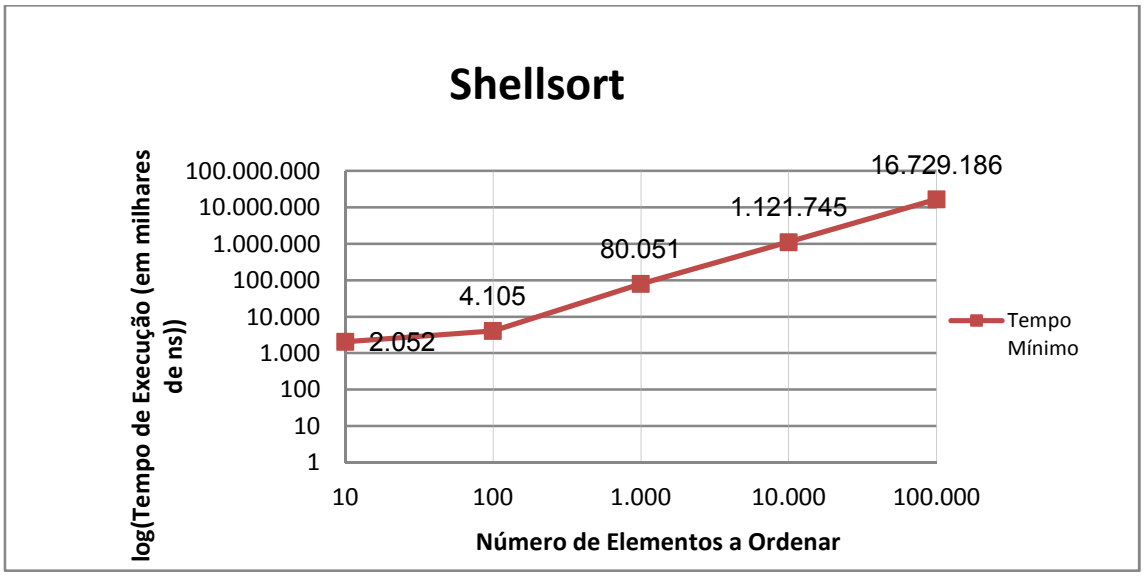

Gráfico 35 – Evolução do Tempo de Execução Mínimo do Algoritmo Shellsort à medida que o número de elementos a ordenar aumenta.

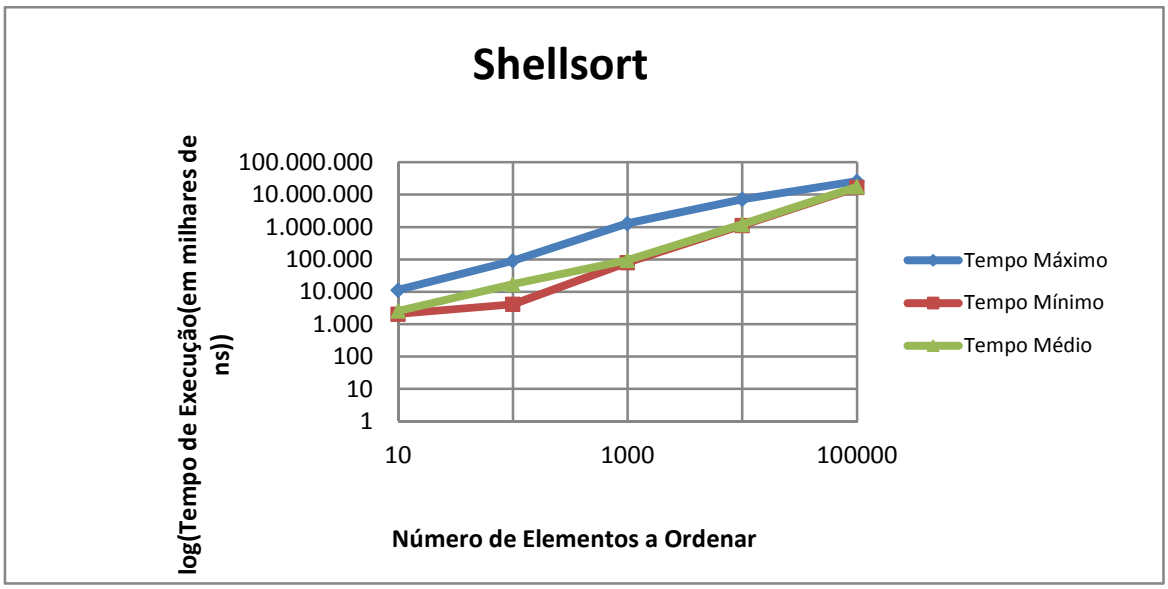

Gráfico 36 – Evolução dos vários tempos de execução à medida que o número de elementos a ordenar aumenta.

Conclusões Após a Análise dos Resultados Globais dos Testes

De modo a facilitar a análise dos resultados globais dos testes efectuados, foram criados gráficos em que constam os tempos de todos os algoritmos para os vários valores de N. Seguidamente podem ser consultados os gráficos elaborados e as conclusões retiradas a partir dos mesmos.

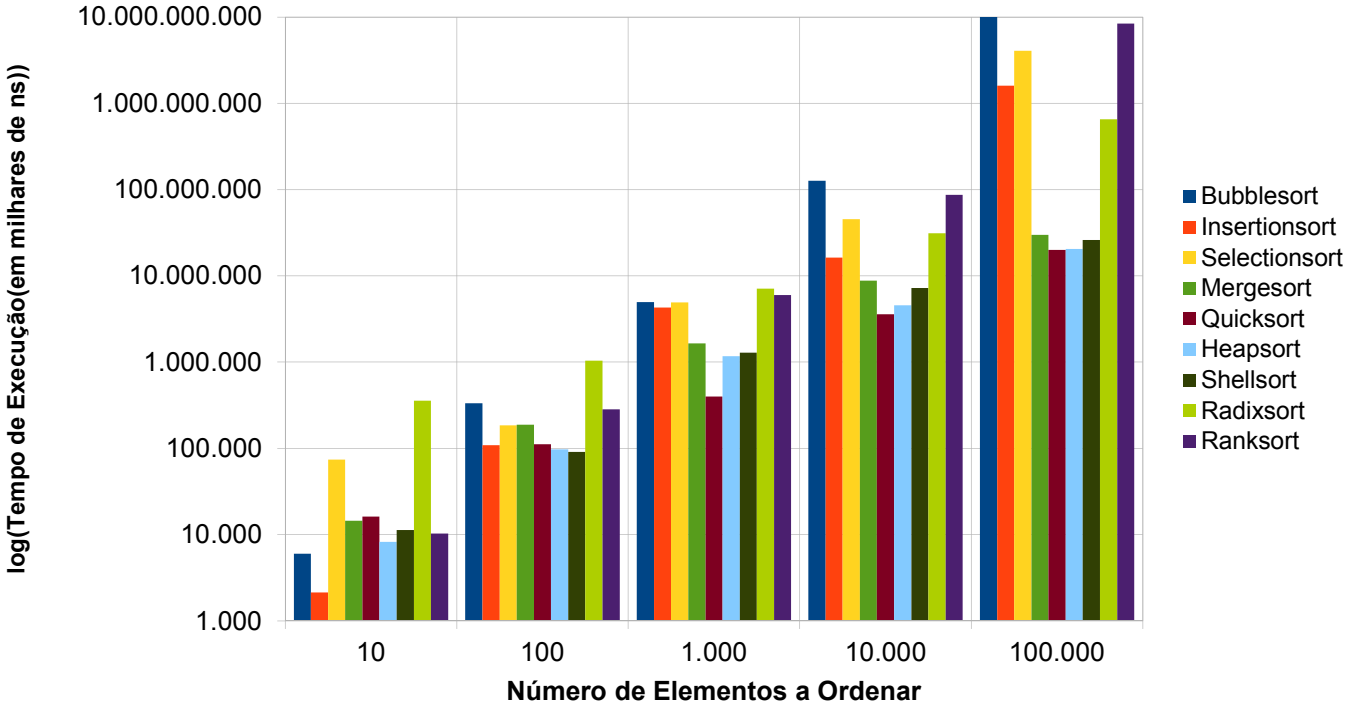

Gráfico 37 – Tempos máximos, para os vários valores de N, de todos os algoritmos testados.

O gráfico anterior representa os tempos de execução máximos verificados, ou seja, os piores tempos registados após ter sido utilizado cada um dos algoritmos para ordenar os vários valores de N.

Quando o valor de N é 10, verifica-se que o algoritmo Insertion sort é o mais eficiente, ordenando todos os elementos do array mais depressa que algoritmos como o Quicksort e o Bubblesort. Para este valor e N, os algoritmos que verificam tempos máximos mais elevados são o Selection sort e o Radix sort.

Se o valor de N for 100, verifica-se que o algoritmo mais rápido a ordenar os elementos do array é o Shellsort. Porém, os tempos máximos de algoritmos como o Heapsort e Quicksort – considerados mais eficientes que o Shellsort – ficam relativamente próximos. Os algoritmos que registam um maior tempo máximo para ordenar 100 elementos são o Radix sort e o Bubblesort.

Quando o número de elementos a ordenar aumenta para 1000, verifica-se que o Quicksort é o algoritmo que tem menor tempo máximo, seguido do Heapsort e do Shellsort. O algoritmo que apresenta maior tempo máximo é o Radix sort, seguido de perto pelos algoritmos Rank sort e Bubblesort. De notar que o algoritmo Insertion sort, que para N = 10 apresentava o menor tempo máximo, é agora um dos algoritmos que tem um dos valores mais elevados.

Se N for igual a 10.000, o algoritmo Quicksort continua a ser o que apresenta o menor tempo máximo, sendo seguido pelos algoritmos Heapsort e Shellsort. Os algoritmos que apresentam os tempos máximos mais elevados são o Bubblesort, seguido do Radix sort e do Selection sort.

Caso o valor de N seja igual a 100.000, os algoritmos que apresentam os menores tempos máximos são o Quicksort e o Heapsort, seguidos do algoritmo Shellsort. Os algoritmos que apresentam os tempos máximos mais elevados são o Bubblesort, seguido do Rank sort e do Selection sort.

Tendo em conta a variação do valor de N, podemos concluir o seguinte relativamente aos tempos de execução máximos registados durante os testes:

- O algoritmo Radix sort tem os tempos máximos mais elevados quando $N \leq 1000$;
- Quando $N \geq 1000$, o algoritmo que apresenta os tempos máximos mais elevados é o Bubblesort;
- O algoritmo Insertion sort, que tem o menor tempo máximo para N = 10, passou a ser um dos algoritmos mais lentos quando $N \geq 1000$;
- O algoritmo que apresenta tempos máximos progressivamente maiores é o Bubblesort, passando a ser o segundo algoritmo com o tempo máximo maior quando N passa de 10 para 100 elementos;
- Os algoritmos que se mantêm com tempos máximos mais baixos à medida que N aumenta são o Quicksort, o Heapsort, o Merge sort e o Shellsort. Destes algoritmos, o Quicksort é o que apresenta melhores resultados.

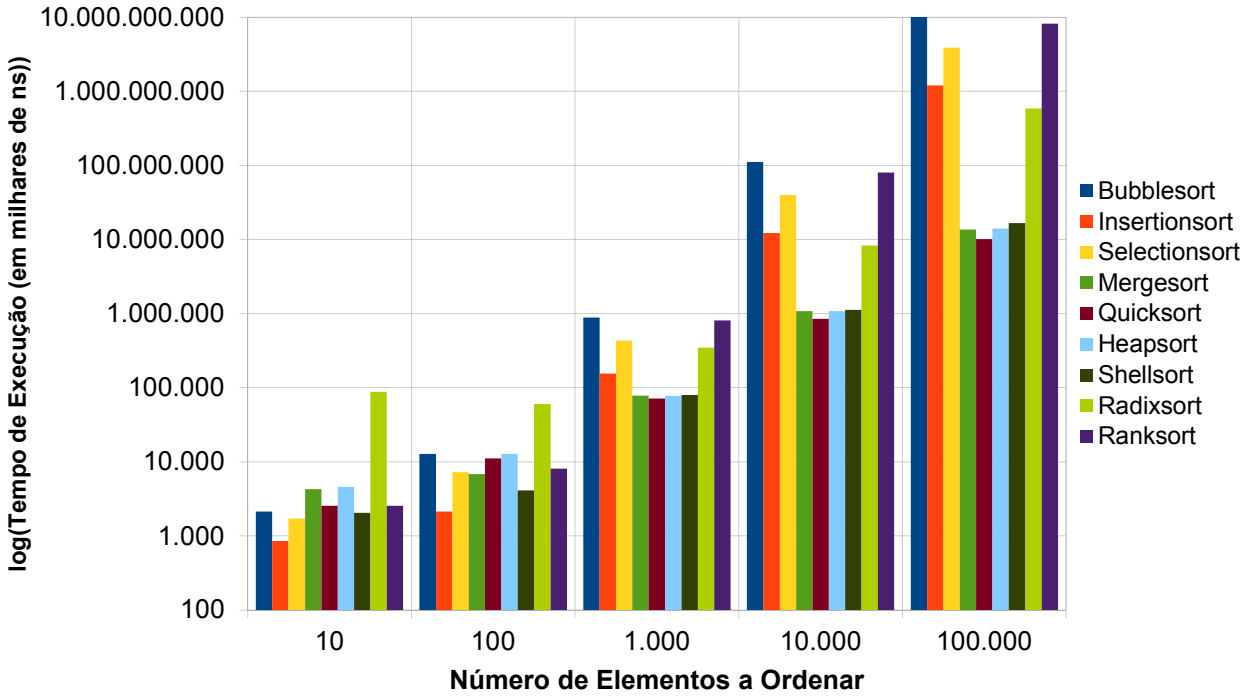

Gráfico 38 - Tempos mínimos, para os vários valores de N, de todos os algoritmos testados.

O gráfico anterior representa os tempos de execução mínimos verificados, ou seja, os melhores tempos registados após ter sido utilizado cada um dos algoritmos para ordenar os vários valores de N.

Quando N = 10, o algoritmo Insertion sort é o que ordena os elementos no menor espaço de tempo. O algoritmo que regista o pior tempo mínimo é o Radix sort, cujo tempo é bastante superior ao dos outros algoritmos.

Se o valor de N for 100, o algoritmo Insertion sort continua a ter o menor tempo mínimo e o algoritmo Radix sort continua a ter o maior tempo mínimo.

Quando o valor de N é igual a 1000, o Quicksort passa a ser o algoritmo que tem o menor tempo mínimo, seguido dos algoritmos Heapsort, Merge sort e Shellsort. O algoritmo que regista o maior tempo mínimo é o Bubblesort, seguido dos algoritmos Rank sort e Selection sort. O algoritmo Radix sort é agora o que regista o quarto pior tempo mínimo.

Aumentando o valor de N para 10.000, verifica-se que o Quicksort continua a ser o algoritmo que regista o melhor tempo mínimo e, tal como acontecia para N = 1000, seguem-se os algoritmos Heapsort, Merge sort e Shellsort. O algoritmo que regista o pior tempo mínimo é

o Bubblesort e, tal como acontecia anteriormente, seguem-se os algoritmos Rank sort e Selection sort.

Se N = 100.000, o algoritmo Quicksort continua a ser o mais rápido, seguido dos algoritmos Heapsort, Merge sort e Shellsort. O algoritmo que apresenta o pior tempo mínimo é o Bubblesort.

As conclusões obtidas após a análise do gráfico foram semelhantes às obtidas após a análise do gráfico dos tempos máximos, mantendo-se as mesmas tendências verificadas anteriormente.

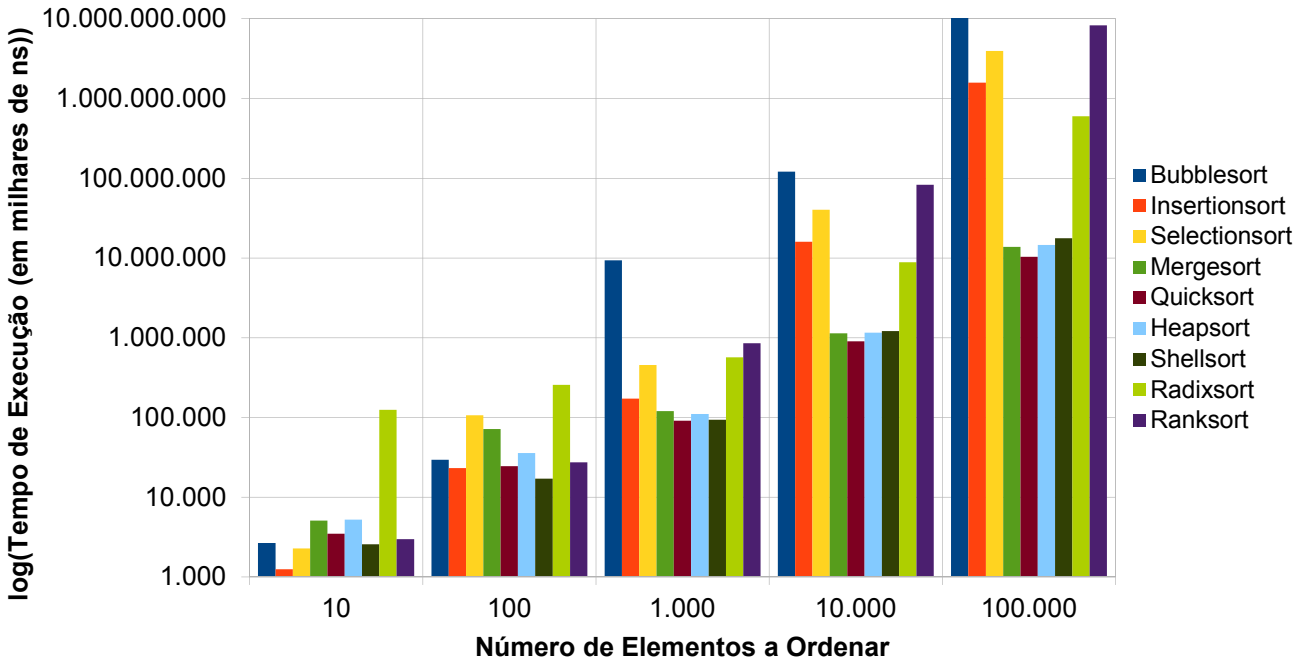

Gráfico 39 - Tempos médios, para os vários valores de N, de todos os algoritmos testados.

O gráfico anterior representa os tempos médios de todos os algoritmos testados para os vários valores de N. Note-se que estes valores representam a média dos tempos de execução registados, desprezando o tempo mínimo e o tempo máximo.

Se N = 10, o Insertion sort é o algoritmo que tem a melhor média e o Radix sort é o algoritmo que tem a pior média.

Quando o valor de N sobe para 100, verifica-se um aumento generalizado das médias dos tempos, passando o Shellsort a ser o algoritmo que regista a melhor média. O Radix sort continua a ser o algoritmo que regista a pior média.

Se N = 1000, o Quicksort passa a ser o algoritmo com a melhor média, seguido do Shellsort. O Bubblesort passa agora a ter a pior média, que corresponde a um valor muito mais elevado que o dos algoritmos que possuem as segunda e terceira piores médias (Rank sort e Radix sort, respectivamente).

Aumentando o valor de N para 10.000, o Quicksort continua a ser o algoritmo que tem a melhor média, registando o Heapsort, Merge sort e Shellsort médias próximas da do Quicksort. No outro extremo encontra-se o Bubblesort, que continua a registar a pior média. Verifica-se também um aumento na média dos tempos de execução dos algoritmos Insertion sort e Selection sort, que ultrapassam a média do algoritmo Radix sort. O algoritmo Rank sort mantém a segunda pior média de tempos.

Quando N = 100.000, o Quicksort mantém a melhor média e o Bubblesort mantém a pior média de tempos. Verificam-se aumentos nas médias de todos os algoritmos, mantendo-se a tendência verificada anteriormente (em N = 10.000).

Tendo em conta a variação do valor de N, podemos concluir o seguinte relativamente às médias dos tempos de execução registados durante os testes:

- O Radix sort é o algoritmo que apresenta uma das piores médias, sendo mesmo a pior quando N = 10;
- Exceptuando o Radix sort, todos os outros algoritmos apresentam boas médias dos tempos de execução para ordenar 10 elementos;
- O Shellsort é o algoritmo que mantém a melhor média quando se pretendem ordenar elementos de arrays pequenos, sendo apenas ultrapassado pelos algoritmos Insertion sort e Selection sort quando N = 10 (estes algoritmos registam médias piores assim que o número de elementos a ordenar sobe para 100);
- O algoritmo que é melhor para ordenar muitos elementos é o Quicksort, que regista as melhores médias quando N ≥ 1000. Quando comparado com os outros algoritmos, o Quicksort regista sempre uma boa média de tempo quer se pretendam ordenar os elementos de um array que seja grande (tenha muitos elementos) ou seja pequeno (tenha poucos elementos);
- O algoritmo que é pior para ordenar muitos elementos é o Bubblesort, que regista as piores médias quando N ≥ 1000;
- O Radix sort regista sempre uma das piores médias quer o array tenha muitos ou poucos elementos, correspondendo a este algoritmo a pior média registada quando N = 10.

Bibliografia

Livros:

- Sedgewick, R. e Wayne, K. 2011. *Algorithms*. 4th, Pearson Education, Indianapolis.
- Lafore, R. 2003. *Data Structures and Algorithms in Java*. 2nd. Sams Publishing.

Fontes da Internet:

- Barrico, C. "*Slides da Disciplina de Programação e Algoritmos - Algoritmos de Ordenação*" (Online). (Acedido em: 10-01-2013). Disponível em: http://www.di.ubi.pt/~cbarrico/Disciplinas/ProgramacaoAlgoritmos/Downloads/Teorica_Ordenacao_Iterativa.pdf
- Wikibooks, "*Algoritmos de Ordenação*" (Online). (Acedido em: 02-01-2013). Disponível em: http://pt.wikibooks.org/wiki/Algoritmos_e_Estruturas_de_Dados/Algoritmos_de_Ordena%C3%A7%C3%A3o
- Wikipedia, "*Algoritmo de Ordenação*"(Online). (Acedido em: 03-01-2013). Disponível em: http://pt.wikipedia.org/wiki/Algoritmo_de_ordena%C3%A7%C3%A3o
- Wikipedia, "*Sorting Algorithm*"(Online). (Acedido em: 03-01-2013). Disponível em: http://en.wikipedia.org/wiki/Sorting_algorithm
- Wikipedia, "*Bubblesort*"(Online). (Acedido em: 27-12-2012). Disponível em: http://en.wikipedia.org/wiki/Bubble_sort
- Wikipedia, "*Heapsort*"(Online). (Acedido em: 29-12-2012). Disponível em: http://en.wikipedia.org/wiki/Heapsort
- Wikipedia, "*John Von Neumann*"(Online). (Acedido em: 07-01-2013). Disponível em: http://en.wikipedia.org/wiki/John_von_Neumann
- Wikipedia, "*Quicksort*"(Online). (Acedido em: 10-01-2013). Disponível em: http://en.wikipedia.org/wiki/Quicksort
- Wikipedia, "*Radix Sort*"(Online). (Acedido em: 11-01-2013). Disponível em: http://en.wikipedia.org/wiki/Radix_sort
- Wilkinson, B. e Allen, M. "*Slides for Parallel Programming Techniques & Applications Using Networked Workstations & Parallel Computers 2nd ed*"(Online). (Acedido em: 09-01-2013). Disponível em: http://www.cs.nthu.edu.tw/~ychung/slides/para_programming/slides10.pdf
- Wilkinson, B. e Allen, M. "*Slides for Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*"(Online). (Acedido em: 09-01-2013). Disponível em: http://www.tjhsst.edu/~rlatimer/assignments2005/WilkinsonOddEvenVer2.pdf
- Wikipedia, "*Selection sort*"(Online). (Acedido em: 09-01-2013). Disponível em: http://en.wikipedia.org/wiki/Selection_sort
- Harwood, A. "*Rank sort*"(Online). (Acedido em: 09-01-2013). Disponível em: http://ww2.cs.mu.oz.au/498/notes/node32.html

- Shedden, K. "*Sorting, Ranking, Indexing, Selecting*"(Online). (Acedido em: 10-01-2013). Disponível em: http://www.stat.lsa.umich.edu/~kshedden/Courses/Stat606/Notes/sorting.pdf
- Qureshi, K. "*A Practical Performance Comparison of Parallel Sorting Algorithms on Homogeneous Network of Workstations*"(Online). (Acedido em: 27-12-2012). Disponível em: http://ww1.ucmss.com/books/LFS/CSREA2006/PDP5081.pdf
- Sedgewick, R. "*Analysis of Shellsort and Related Algorithms*"(Online). (Acedido em: 07-01-2013). Disponível em: http://www.cs.princeton.edu/~rs/shell/paperF.pdf

Outros:

- Slides da Disciplina "*Estruturas de Dados e Algoritmos I*" – Aula 10.