Universidade de São Paulo
Instituto de Ciências Matemáticas e Computação - ICMC

Relatório do trabalho 2

Alunos: Augusto Ildefonso, Renan Trofino
Professor: Marcelo Garcia Manzato
Monitor: Lucas Padilha Modesto de Araujo

setembro
2024

Universidade de São Paulo
Instituto de Ciências Matemáticas e Computação - ICMC

Relatório

Relatório do segundo trabalho de Introdução à Ciência da Computação II.

Alunos: Augusto Ildefonso, Renan Trofino

Professor: Marcelo Garcia Manzato

Monitor: Lucas Padilha Modesto de Araujo

setembro
2024

Sumário

1 Descrição do Problema

O presente trabalho tem como objetivo implementar, testar e comparar diferentes algoritmos de ordenação, em relação ao tempo de execução, número de comparações e número de trocas. Para resolvê-lo, foram implementados os algoritmos em C e foi criado um arquivo em Python para rodar os códigos e gerar os resultados em forma de gráficos. .Os resultados gerados indicam casos de uso e possíveis aplicações para cada algoritmo, considerando suas limitações vantagens em relação aos outros.

Importante ressaltar que a análise de troca e comparações foi feita somente para os algoritmos de ordenação que usam comparação. Além disso, em muitos casos no qual o vetor já estava ordenado não houve nenhuma troca nem comparação, o que é de se esperar já que o vetor está ordenado.

2 Bubble Sort

2.1 O algoritmo

O algoritmo "flutua"os valores para sua posição fazendo comparações. A cada iteração, é garantido que o elemento $n - i$ já estará ordenado.

É necessário percorrer o vetor com dois laços: O laço externo itera sobre o tamanho do vetor. O laço interno será executado n vezes, e é responsável for flutuar os elementos.

Se a comparação for satisfeita, então os dois valores comparados são trocados (swap).

2.2 Análise

Existem dois laços no algoritmo, então pode-se concluir, inicialmente que a complexidade é $O(n^2)$. No bubble sort normal (sem melhorias), tanto no melhor como no pior caso, o algoritmo continua sendo $O(n^2)$

Porém, o caso otimizado (incluindo contagem de trocas) apresenta uma leve diferença nos cálculos, mas que resulta em $O(n^2)$também. Nesse caso, o pior caso segue igual. Mas, no caso em que o vetor já está ordenado, o algoritmo executa 1 vez o laço interno, o que resulta em $O(n)$.

3 Selection Sort

3.1 O algoritmo

Cada iteração seleciona o menor valor e coloca-o no começo do array. Depois da iteração "i", o iésimo item estará na posição correta.

Dois laços são necessários. O primeiro, mais externo, itera no tamanho do vetor. o Segundo, interno, cuida da comparação do valor. O valor mínimo será declarado como "i". Se o valor em "j"for menor que o valor na posição mínimo, então o mínimo agora é "j". Depois desse laço, se o mínimo for diferente de "i", então troque suas posições (swap).

3.2 Análise

Por ter dois laços, o algoritmo é $O(n^2)$. Por percorrer sempre duas vezes (independentemente do caso), sempre será $O(n^2)$.

4 Insertion Sort

4.1 O algoritmo

Muito similar ao Bubble Sort, este algoritmo flutua os valores para o início do vetor, garantindo que na iésima iteração o elemento na posição "i"esteja em seu lugar. A diferença, entretanto, está na extremidade atingida; nesse caso, o início do vetor é ordenado primeiro.

4.2 Análise

O algoritmo possui dois laços, então apresenta complexidade $O(n^2)$. Matematicamente, cada passo tem seu valor: $1 + 2 + 3 + ... + (n - 1)$. Essa progressão aritmética pode ser somada como $\frac{n(a_1+a_n)}{2}$, o que resulta em $O(n^2)$.

5 Shell Sort

5.1 O algoritmo

O Shell-Sort é uma variação da inserção simples. Enquanto ela compara elementos adjacentes e move uma posição a frente, o Shell-Sort permite a troca de elementos distantes. Ele ordena elementos separados por h posições, de tal forma que todo h-ésimo elemento está em uma sequência ordenada. Também, é importante que o h seja reduzido a cada iteração, para que assim possa ordenar todo o vetor.

5.2 Análise

A partir da análise do algoritmo, percebe-se que com uma sequência adequada de elementos h é aproximadamente $O(n\,(\log n)^2)$. Por exemplo, se usarmos a sequência de Pratt, ou seja, números formados por $2^i \times 3^i$.

6 Quicksort

6.1 O algoritmo

O Quicksort é um algoritmo de ordenação baseado em partições. Dado vetor, o objetivo é dividí-lo em partes menores para então ordená-las. Esse método é conhecido como "divisão e conquista", sendo muito comum na computação.

Para particionar o vetor, é necessário encontrar um pivô. Existem diferentes métodos para realizar essa tarefa, mas, aqui, o método escolhido foi o da partição por mediana. Para realizá-lo, a mediana entre os valores nas posições 0, $n/2$ e $n-1$ é calculada, tornando-se o pivô.

6.2 Análise

Para o pior caso, a complexidade é $O(n^2)$. Como o algoritmo é recursivo, é preciso fazer análise de recorrência, assim, analisando o algoritmo, vemos que para subvetores desiguais com n chamadas recursivas e eliminando 1 elemento a cada chamada, tem-se a equação de recorrência:

$$T(n) = T(\frac{n}{1}) + O(n) \tag{1}$$

$$T(n) = \frac{1}{2}\, n\, (n+1) \tag{2}$$

$$T(n) = O(n^2). \tag{3}$$

Se escolhermos um pivô que divida o vetor ao meio, teremos o melhor caso. Para esse caso, temos a seguinte equação de recorrência:

$$T(n) = 2\, T(\frac{n}{2}) + n \tag{4}$$

Aplicando o método da árvore de recorrência para resolver ela, obtemos:

$$T(n) = O(n\, \log n) \tag{5}$$

Por último, para o caso médio, a complexidade é de $O(n\, \log n)$.

7 Heapsort

7.1 O algoritmo

O Heapsort é um algoritmo eficiente de ordenação que aproveita da estrutura heap (parecida com uma árvore binária). Para iniciar o algoritmo é necessário construir uma heap usando um vetor, que contém os nós e as folhas (folhas sempre da posição $n/2$ em diante).

7.2 Análise

Analisando a heap, percebe-se que ela possui uma altura de $\log n$. Além disso, para ordenar todo o vetor é preciso repetir o processor de ordenação n vezes. Assim, considerando que repetimos n vezes um processo de custo $\log n$, temos que a complexidade é de $O(n \log n)$.

8 Merge Sort

8.1 O algoritmo

O Merge Sort é um algoritmo recursivo baseado na lógica de dividir-e-conquistar. Ele divide o vetor de entrada ao meio, recursivamente, em sub-vetores até que eles tenham tamanho 1. Então ele junta os sub-vetores já na ordem correta, ou seja, a ordenação ocorre na junção.

8.2 Análise

Analisando a implementação do Merge Sort, percebe-se que ele tem a seguinte equação de recorrência:

$$T(n) = \begin{cases} O(1), \, se \, n = 1 \\ 2\,T(\frac{n}{2}) + O(n), \, se \, n \end{cases} \tag{6}$$

A partir dessa equação, obtemos que a complexidade do Merge Sort no melhor, no pior e no caso médio é $O(n \, \log n)$.

9 Contagem de Menores

9.1 O algoritmo

A Contagem de Menores é um algoritmo de ordenação que, a partir da quantidade de números menores que a chave, insere o elemento na posição correta. Por exemplo, se há 5 valores menores que o elemento 8, sabemos que o elemento 8 será inserido na 6ª posição, ou seja, na posição de index 5.

9.2 Análise

A complexidade do algoritmo de Contagem de Menores é $O(n^2)$, pois é preciso há dois loops for aninhados. Esses loops são responsáveis por montar o vetor de menores.

10 Radix Sort

10.1 O algoritmo

O algoritmo de Radix Sort ou Ordenação de Raizes, ordena o vetor através dos dígitos dos números, do dígito menos significativo até o mais significativo. Para construir esse algoritmos fazemos uso da estrutura de dados **fila** para cada uns dos possíveis dígitos (do 0 ao 9). Os números são inseridos nas filas de acordo com os dígitos de análise e o processo é repetido até passar por todos os dígitos dos números.

10.2 Análise

O algoritmo do Radix Sort tem duas componentes que influenciam na sua complexidade: o número de elementos da entrada (n) e o número de dígitos do maior número (m). Sabendo disso, temos que a sua complexidade é $O(n \times m)$ e, para um m suficientemente pequeno, a complexidade é $O(n)$.

11 Resultados Obtidos em Cada Configuração

Algoritmo	Caso	Tamanho	Tempo	Trocas	Comparações
radix.c	normal	100	0.000002	0	0
radix.c	reverse	10000	0.000846	0	0
radix.c	random	1000	0.000033	0	0
bubble.c	normal	100	0.000000	0	0
bubble.c	reverse	10000	0.217951	49995000	49995000
bubble.c	random	1000	0.000112	12072	25164
heap.c	normal	100	0.000001	1	2
heap.c	reverse	10000	0.001461	116697	243394
heap.c	random	1000	0.000026	1566	3356
insertion.c	normal	100	0.000001	0	0
insertion.c	reverse	10000	0.129885	9999	50004999
insertion.c	random	1000	0.000034	224	12296
merge.c	normal	100	0.000000	0	0
merge.c	reverse	10000	0.000763	133616	133616
merge.c	random	1000	0.000017	1769	1769
quick.c	normal	100	0.000001	0	0
quick.c	reverse	10000	0.001260	139308	231294
quick.c	random	1000	0.000017	1248	1905
selection.c	normal	100	0.000001	0	0
selection.c	reverse	10000	0.117082	5000	49995000
selection.c	random	1000	0.000059	220	25200
shell.c	normal	100	0.000001	0	0
shell.c	reverse	10000	0.000614	120005	62560
shell.c	random	1000	0.000022	1354	1376
counting.c	normal	100	0.000002	2	2
counting.c	reverse	10000	0.000160	20000	19998
counting.c	random	1000	0.000009	450	1224

Nota: A tabela está consideravelmente reduzida, apresentando apenas alguns casos de cada algoritmo. A tabela completa está disponível no repositório do projeto (ver Anexo).

12 Análise dos Resultados

Cada algoritmo foi testado em diferentes casos de teste, que seguiam os critérios de tamanho (10, 100, 1000 e 10000) e também de ordenação (normal, reversa e aleatória). O tempo de execução das funções responsáveis pela ordenação foi armazenado em um arquivo contento, também, informações do algoritmo, o tipo de teste e o tamanho das entradas.

Usando a tabela gerada, foi possível extrair métricas e construir gráficos relevantes para a análise do comportamento dos algoritmos aqui apresentados nas condições impostas.

12.1 Comportamento

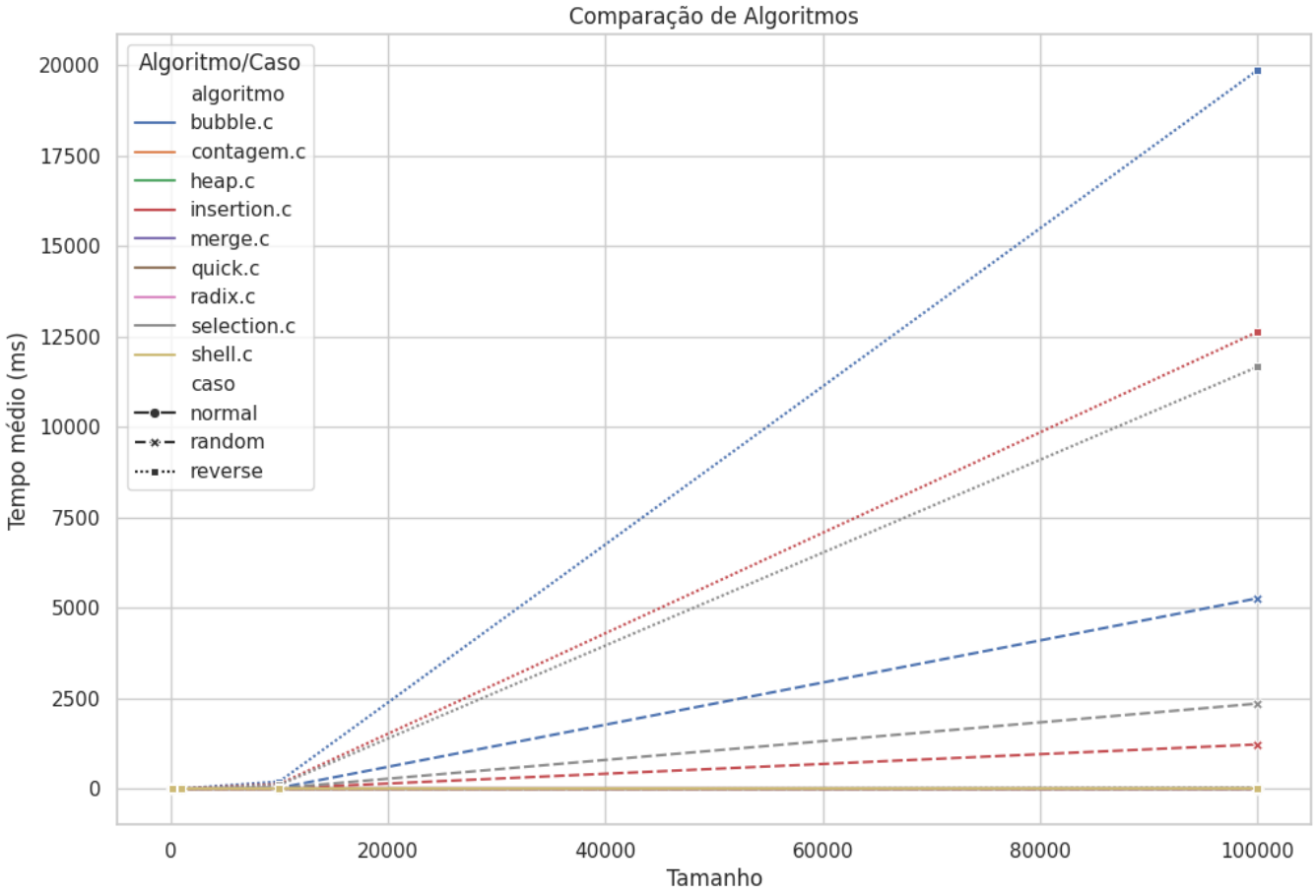

Figura 1: Comparação dos tempos de execução dos algoritmos

12.2 Variações em cada caso

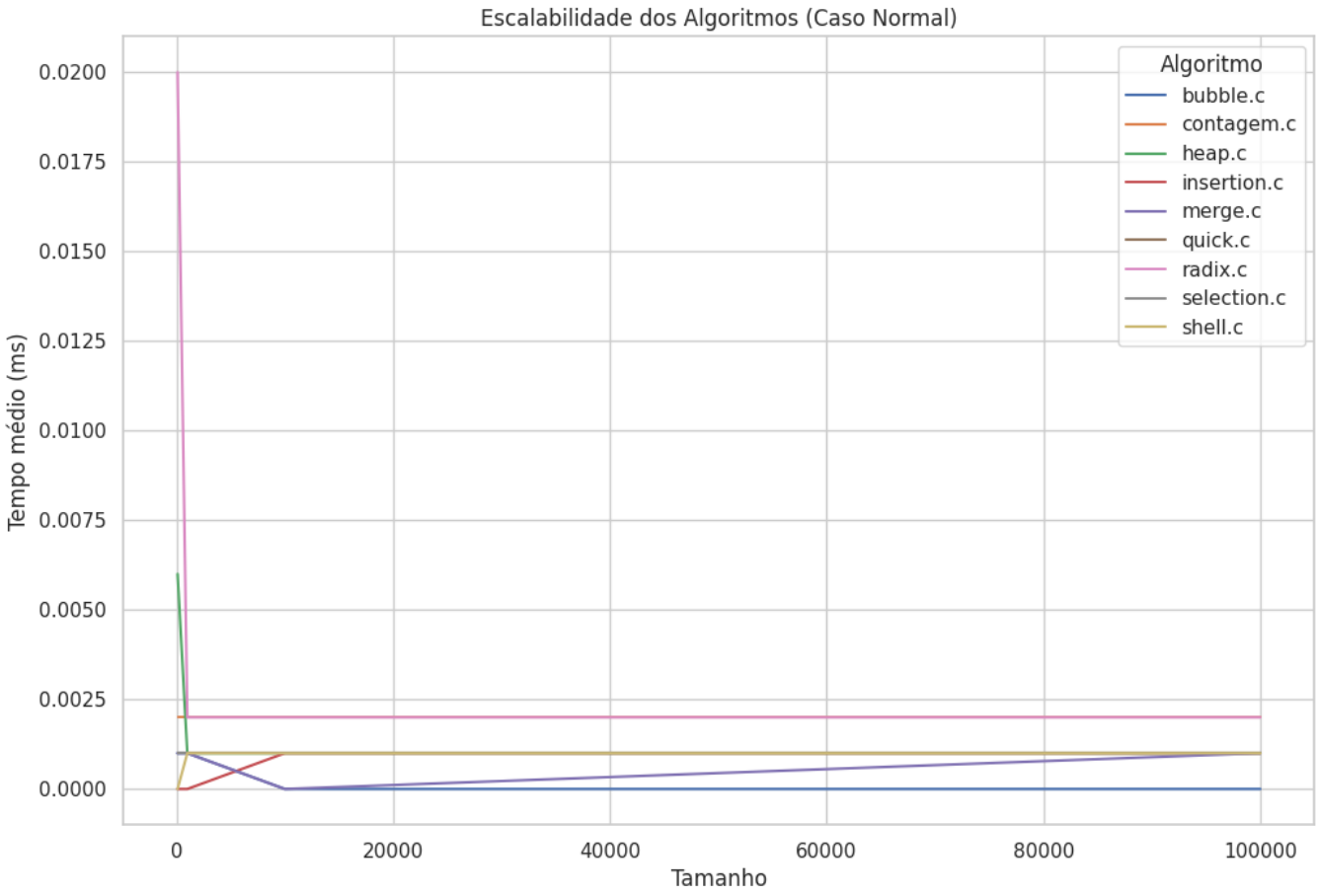

Figura 2: Variação do tempo de execução dos algoritmos no caso normal

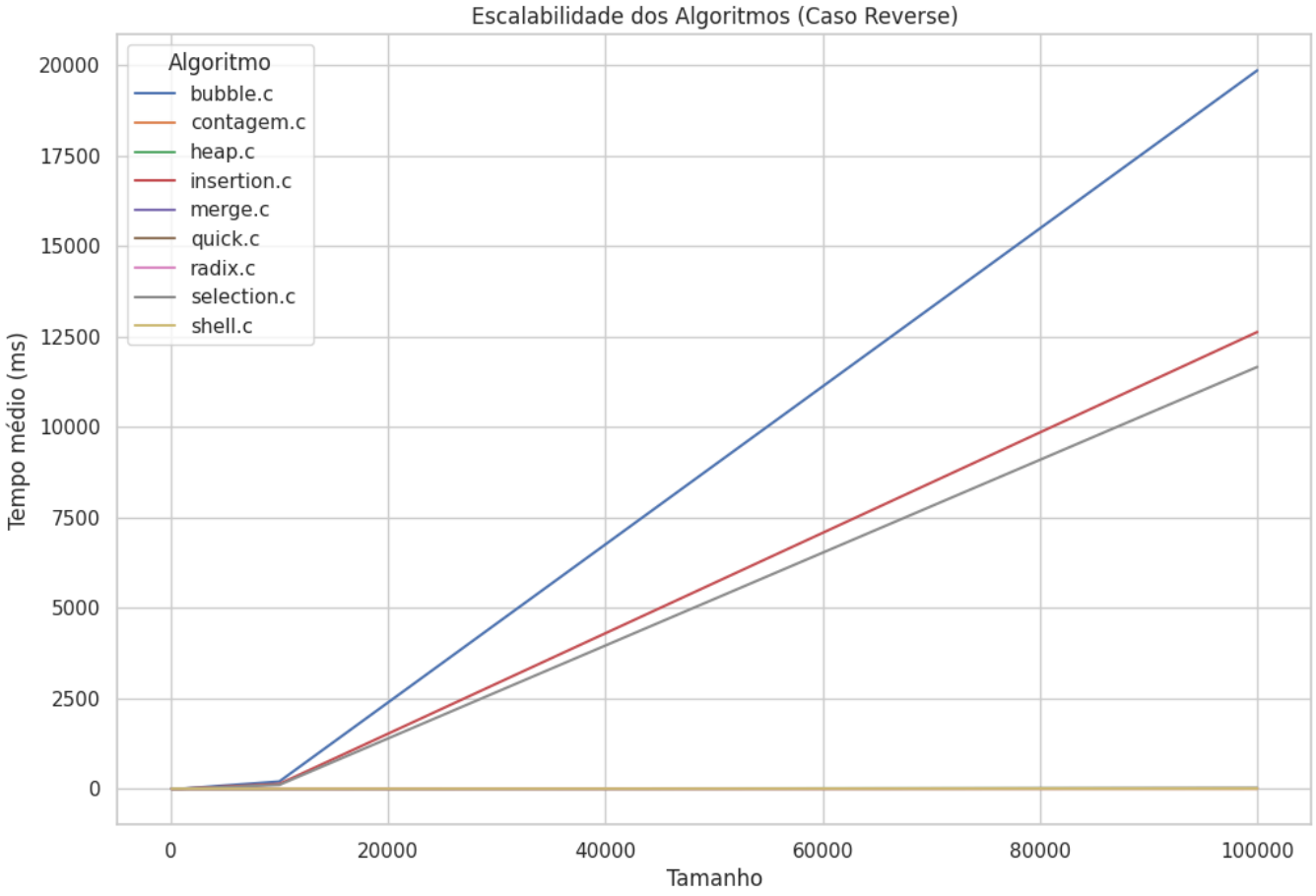

Figura 3: Variação do tempo de execução dos algoritmos no caso reverso

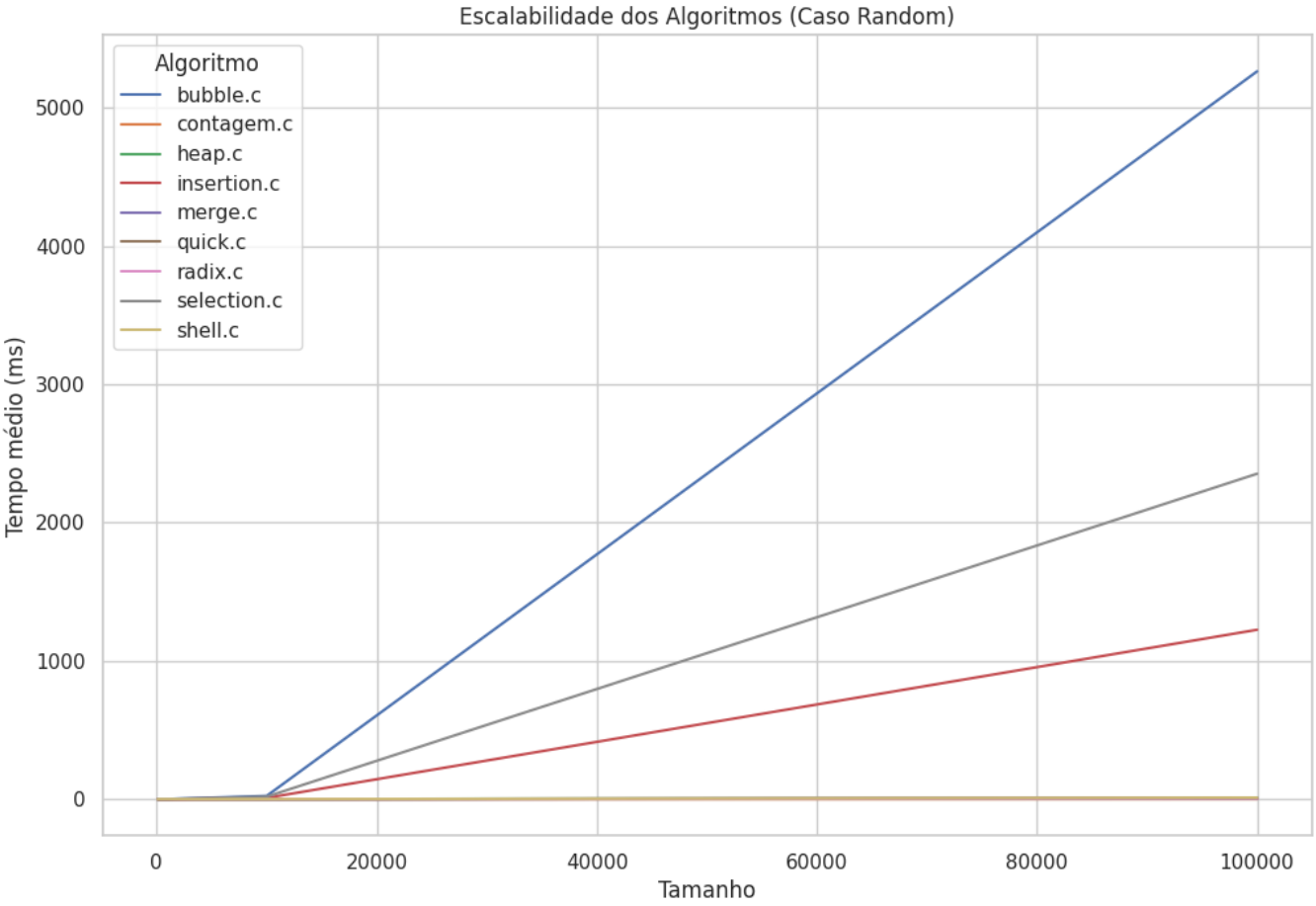

Figura 4: Variação do tempo de execução dos algoritmos no caso aleatório

12.3 Mapa de calor dos tempos de execução

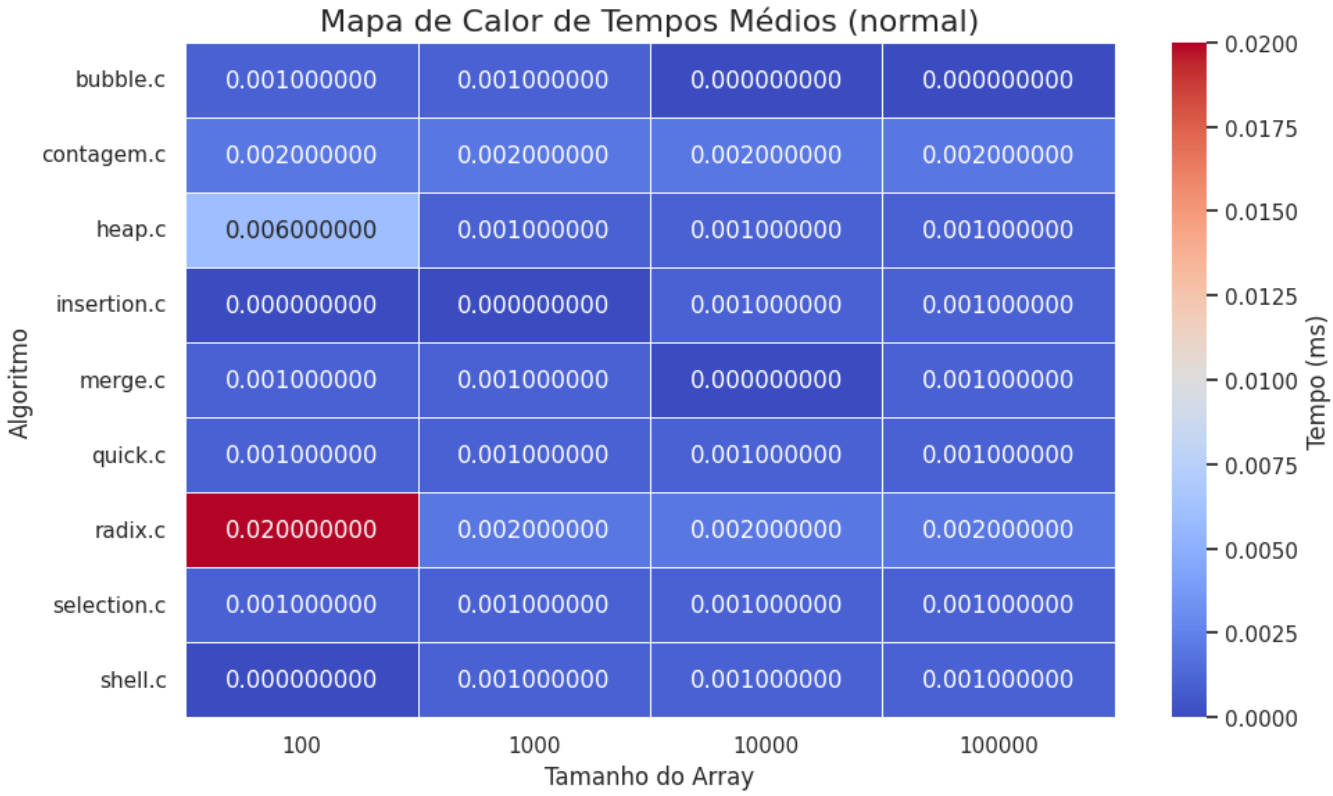

Figura 5: Mapa do tempo de execução dos algoritmos no caso normal

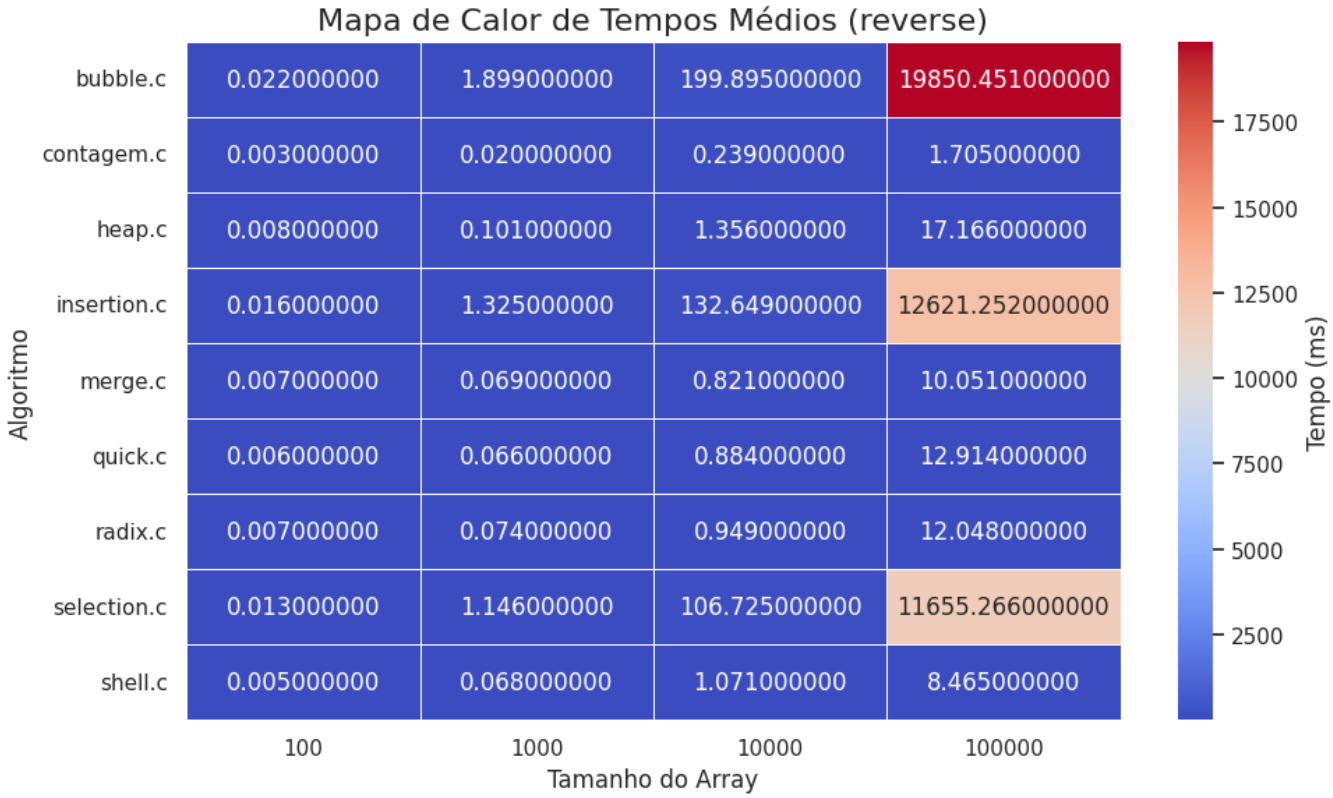

Figura 6: Mapa do tempo de execução dos algoritmos no caso reverso

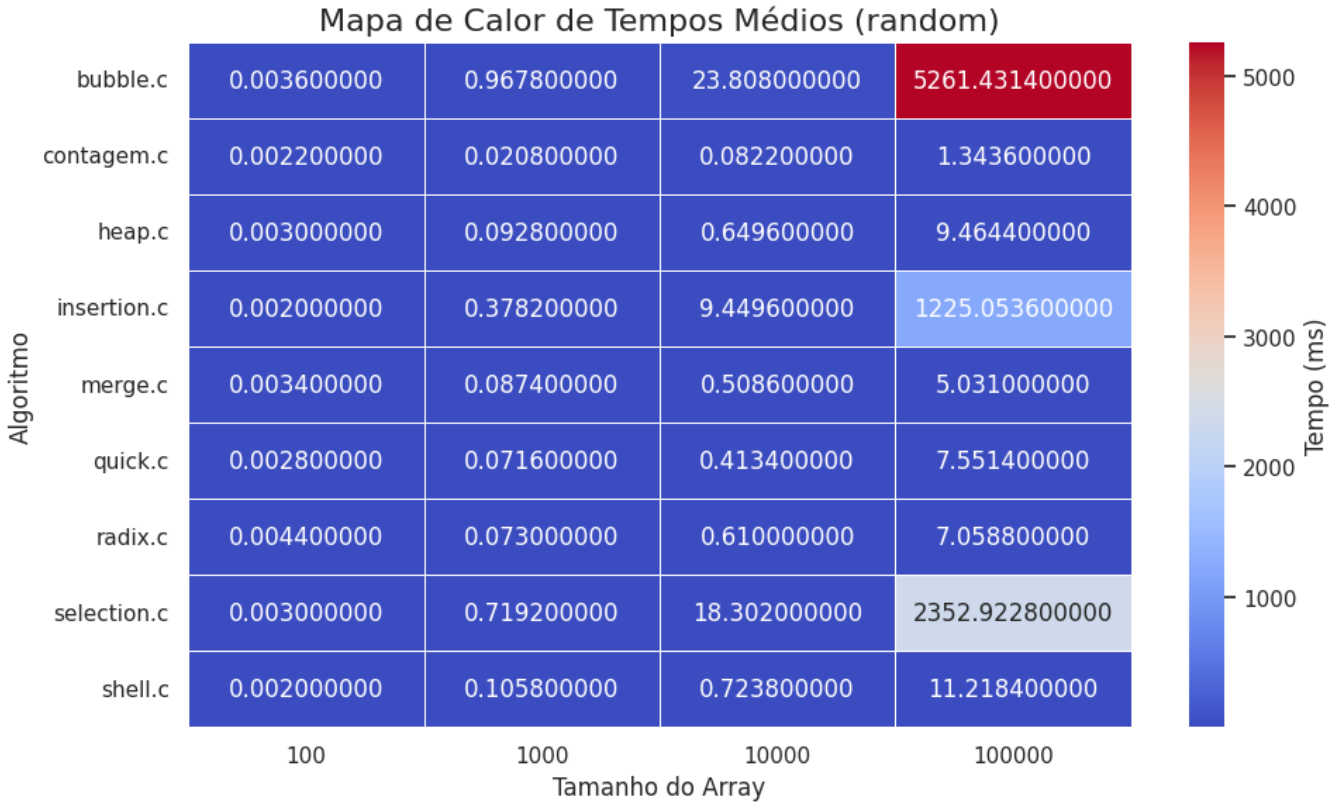

Figura 7: Mapa do tempo de execução dos algoritmos no caso aleatório

Os resultados apresentados em gráficos permitem uma melhor visualização dos tempos, trocas e comparações dos algoritmos em cada um dos casos.

Com uma média dos tempos de execução em cada caso (Figura 1) é possível notar a grande diferença nos tempos do bubble sort, insertion sort e selection sort, com tempos de execução chegando à 20 segundos, em relação a maioria dos algoritmos, que possui uma variação mais comportada (a linha mais abaixo, de tonalidade marrom claro, representa esses algoritmos, que na imagem aparecem aglomerados pela pouca diferença). O pior caso, aqui, representa o bubble sort com valores inversamente ordenados.

Os casos de variação/escalabilidade (Figuras 2, 3 e 4) permitem analisar casos individualmente, apresentando os comportamentos em cada situação.

Casos ordenados em ordem normal (Figura 2) apresentam os melhores tempos no algoritmo bubble sort, devido à checagem de ordenação e consequente parada do algoritmo, já que o vetor está devidamente ordenado. Por outro lado, os algoritmos radix sort, heapsort e merge sort possuem um pico

de execução para casos pequenos, que se tornará estável com o aumento no número de entradas.

Os casos com ordenação reversa (Figura 3) mostram a ineficiencia de algoritmos $O(n^2)$ para casos pequenos, mas principalmente em casos grandes. Tal ineficiêcia ocorre pelo maior número de trocas e comparações que ocorrem.

Para vetores com items aleatórios (Figura 4), o problema dos algoritmos $O(n^2)$ se repete, ocasionando um tempo muito maior para bubble sort, insertion sort e slection sort. Os outros algoritmos possuem um comportamento pouco variável, que não gera tantas oscilações no tempo de execução.

Os mapas de calor (Figuras 5, 6 e 7) também apresentam as médias de execução em cada caso, destacando em cores quentes os tempos maiores e em cores frias os tempos menores. Novamente, para casos cujo vetor não possui ordenação normal, os algortimos com complexidade $O(n^2)$ aparecem destacados cores mais próximas ao vermelho, indicando sua ineficiência nestes casos.

12.4 Conclusões e considerações

Algoritmos de ordenação são parte fundamental de toda a computação, estando presentes em diversas de suas áreas.

Encontrar algoritmos eficientes é a chave de toda essa questão. Aqueles que se adequam à maioria dos casos, provavelmente, serão os mais utilizados. Com melhorias e limitações, alguns algoritmos podem ser muito específicos para certas situações, mas realizam-na eficientemente.

Com a análise anteriormente apresentada, é notório o destaque do quicksort, algoritmo recursivo que possibilita, como o próprio nome sugere, uma ordenação rápida e efetiva de registros. Sua constância de tempos mostra também sua versatilidade para diferentes tipos de problemas, tornando-o uma escolha muito viável.

Entretanto, existem algoritmos mais simples (no quesito implementação) que podem realizar tarefas para montantes menores de dados, como é o caso do counting sort (ou contagem dos menores), que atua um pouco melhor que os outros algoritmos em casos pequenos.

Algoritmos como o merge sort e o heapsort são muito úteis em problemas cuja tolerância na variação de tempo seja mínima, uma vez que sempre executarão em tempo $O(n \log n)$. Porém, cabe a ressalva ao merge sort, que precisa de espaço extra para operar.

Também é perceptível a demora dos algoritmos bubblesort, insertion sort e selection sort para casos grandes que não são ordenados. Isso ocorre pela carga maior de comparações e trocas que serão feitas, com aumento

quadrático ($O(n^2)$) conforme a variação da entrada. Esses algoritmos, todavia, podem ser usados para casos pequenos, já que não possuem grande dificuldade de implementação.

Anexo

Todos os algoritmos, casos de teste e script para gerar gráficos estão disponíveis no repositório do projeto no Github (Disponível em `https://github.com/renan823/Trabalho2-ICC2/`).

Referências

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2022.

[2] Feofillof, P. *Algoritmos: Em Liguagem C*. Elsevier Brasil, 2013.

[3] Feofiloff, P. Análise de algoritmos. *Internet: http://www. ime. usp. br/pf/analise_de_algoritmos 2009* (1999).