

Universidade de São Paulo
Instituto de Matemática e Estatística
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação
MAC110 - Introdução à Computação

Análise de diferentes algoritmos de ordenação

Aluno: João Viktor Souza Almeida

NUSP: 15521614

Turma: MAC0110-145-2024

Professor: Roberto Hirata Junior

Resumo

O relatório a seguir visa dissertar acerca algoritmos de ordenação clássicos, tais como bolha, contagem, inserção e seleção, bem como investigar exemplos, apresentar a implementação em Python e elucidar como o algoritmo se comporta ao ordenar uma lista.

Além disso, é verificado como a eficiência dos algoritmos variam dadas diferentes condições iniciais, tais como porcentagem de ordenação e composição da lista a ser ordenada.

Por fim, foram implementados os algoritmos na linguagem C a fim de analisar os seus comportamentos em uma linguagem compilada, em contraste com a linguagem utilizada, Python, que é interpretada.

Metodologia

Na criação do relatório, os testes dos algoritmos foram realizados utilizando a linguagem de programação Python, com a versão 3.10.11, no sistema operacional Windows. Além disso, os arquivos em C foram compilados utilizando o GCC na versão 13.1.0.

Durante a execução dos algoritmos, foi utilizada uma máquina com as seguintes configurações:

- Processador (CPU): Ryzen 5 3350G
- Memória (RAM): 16GB DDR4 @ 3200MHz
- Armazenamento: 256GB SSD
- Placa de Vídeo (GPU): Radeon Vega 11
- Sistema Operacional: Windows 11 Pro

Adicionalmente, a fim de minimizar possíveis interferências nos resultados dos testes, os códigos foram executados com o mínimo de programas em segundo plano. Para computar a média de execução, foi criada uma função na qual que retornava a média e o desvio padrão das 10 itinações do mesmo algoritmos. Além disso, foram ignoradas possíveis margens de erros da função que embaralha as listas. Contudo, essa margem de erro torna-se desprezível devido ao tamanho das listas utilizadas.

Por fim, na implementação dos códigos na linguagem C, foi importada a biblioteca externa `stdbool.h` a fim de criar variáveis booleanas. Além disso, para possibilitar a importação das funções para o arquivo em Python, foi executado, no terminal, o

Testes

Para a realização do relatório, foram realizados dois testes. O primeiro focou em observar os impactos causados pelo tamanho da lista a ser ordenada, ou seja, como o desvio padrão e a média mudam ao aumentar ou diminuir o tamanho da lista a ser ordenada por cada algoritmo. Para isso, foi criada uma malha de repetição e computados as médias e os desvios-padrões de cada algoritmo com listas de tamanhos 1000, 5000, 10000, 50000 e 100000.

O segundo, por outro lado, visou elucidar acerca das mudanças causadas pela taxa de ordenação de uma lista. Para isso, foi estabelecida uma segunda malha de repetição, na qual também foram computados as médias e os desvios-padrões de cada algoritmo, mas com uma lista de 100000 elementos e com porcentagens de ordenação de 1%, 3%, 5%, 10% e 50%.

Para computar a média, foi criada uma função cujos parâmetros são uma lista e o tamanho desta, respectivamente. O desvio padrão, sob o mesmo ponto de vista, foi calculado por meio de uma função que recebe os mesmos parâmetros que a função outrora citada.

Palavras-chave: insertion, bubble, counting, selection, algoritmos, ordenação, análise;

Conteúdo

1	Algoritmo de seleção	5
1.1	Exemplo	5
1.2	Implementação	6
1.3	Quantidade de comparações	6
2	Algoritmo de bolha	7
2.1	Exemplo	7
2.2	Implementação	8
2.3	Quantidade de comparações	8
3	Algoritmo de inserção	9
3.1	Exemplo	9
3.2	Implementação	10
3.3	Quantidade de comparações	10
4	Algoritmo de contagem	11
4.1	Exemplo	11
4.2	Implementação	12
4.3	Quantidade de comparações	12
5	Resultados	13
5.1	Variação na quantidade de elementos	13
5.2	Ordenação prévia	14
6	Conclusão	15
7	Implementação em C	16

7.1 Resultados	17
8 Considerações finais	18

1 Algoritmo de seleção

O algoritmo de seleção é um algoritmo de ordenação no qual, a cada iteração, o menor elemento da lista é garantido estar na posição correta. Dessa forma, na primeira iteração, assegura-se que o menor elemento ficará na primeira posição da lista, na segunda iteração, o segundo elemento, e assim por diante. Uma observação pertinente é que a quantidade de comparações a ser feita independe da porcentagem de ordenação da lista.

1.1 Exemplo

Seja M uma lista `[4,3,2,1,0]`. Na primeira passagem, o algoritmo detectará o menor elemento da lista e, logo após, irá permutá-lo com o primeiro elemento da lista, ou seja, o elemento 4 e 0 serão trocados de lugar. Na segunda passagem, começando pelo segundo elemento, a segunda malha de repetição irá detectar o segundo menor elemento e, similarmente, irá permutá-lo com o segundo elemento da lista. Nas próximas execuções, a forma é análoga.

Na n -ésima passagem, onde n é o tamanho da lista, todos os elementos estarão em ordem crescente e, portanto, ordenados.

Assim, seguem as impressões da lista a cada modificação.

```
1 [4, 3, 2, 1, 0]
2 [0, 3, 2, 1, 4]
3 [0, 1, 2, 3, 4]
4 [0, 1, 2, 3, 4]
5 [0, 1, 2, 3, 4]
6 [0, 1, 2, 3, 4]
```

Percebe-se que, mesmo a lista já estando ordenada, o algoritmo continuará realizando os comandos, ou seja, o número de passagens a ser feita não é mudado devido às condições iniciais de ordenação.

1.2 Implementação

A seguir, segue a implementação do código em Python.

```
1 def selection(V, n):
2     for i in range(0, n):
3         smallest_num_index = i
4         for j in range(i, n):
5             if V[j] < V[smallest_num_index]:
6                 smallest_num_index = j
7         V[i], V[smallest_num_index] = V[
            smallest_num_index], V[i]
```

De fato, pelo código percebe-se que não há nenhum mecanismo implementado a fim de detectar a ordenação total da lista e, conseqüentemente pará-lo.

1.3 Quantidade de comparações

Nesta subsecção, irá ser debatida a quantidade de comparações feitas pelo algoritmo em questão. Como a quantidade de comparações é independente da ordenação neste algoritmo, será simples de analisá-la; nos outros algoritmos, contudo, a análise é mais complexa devido à oscilação na quantidade de comparações.

Suponha-se que o algoritmo recebeu uma lista com n elementos. Assim, como não há um dispositivo que pare o código antes, tem-se que a primeira malha de repetição será executada $n - i$ vezes, em que o índice inicial $i = 0$. Na segunda execução, quando o índice $i = 1$, serão realizadas $n - 1$ comparações e assim por diante. Por consequência, em cada uma dessas execuções, a segunda malha de repetição (a qual está na primeira) será executada $n - i$ vezes, isto é, o código será executado $n + (n - 1) + (n - 2) + \dots + (1) = \sum_{i=1}^n i = \frac{n(n+1)}{2}$ vezes.

Notavelmente, não há variações na quantidade de comparações e, conseqüentemente, a forma como o código será executado é a mesma independente do estado inicial da lista.

2 Algoritmo de bolha

O algoritmo de bolha é um algoritmo cuja complexidade é, assim como o anterior, quadrática. Sua principal característica é que, na n -ésima passagem pela lista, o algoritmo assegura estarem ordenados os n últimos elementos da lista. Um fato interessante é que o nome do algoritmo faz referência às bolhas em bebidas, uma vez que as bolhas maiores sobem mais rapidamente do que as menores.

2.1 Exemplo

Seja M uma lista $[4,3,2,1,0]$. Na primeira passagem da primeira malha de repetição, será detectado que os elementos 4 e 3 estão em posições erradas e, assim, serão trocados, deixando a lista com os valores $[3,4,2,1,0]$. Posteriormente, será detectado que os elementos 4 e 2 estão trocados e, assim como no primeiro passo, ambos serão permutados e, como resultado, a lista tornar-se-á $[3,2,4,1,0]$. Sendo assim, a última execução da primeira passagem posicionará o elemento 4 na última posição da lista, garantindo que o maior elemento esteja na posição correta. O algoritmo continuará realizando estes passos, isto é, trocar o elemento j com o $j + 1$ se o primeiro for maior que o segundo até que, na 5ª passagem, (pois a lista possui 5 elementos), a lista estará totalmente ordenada.

Destarte, seguem as impressões da lista a cada modificação realizada.

```
1 [4, 3, 2, 1, 0]
2 [3, 4, 2, 1, 0]
3 [3, 2, 4, 1, 0]
4 [3, 2, 1, 4, 0]
5 [3, 2, 1, 0, 4]
6 [2, 3, 1, 0, 4]
7 [2, 1, 3, 0, 4]
8 [2, 1, 0, 3, 4]
9 [1, 2, 0, 3, 4]
10 [1, 0, 2, 3, 4]
11 [0, 1, 2, 3, 4]
```

Perceptivelmente, o maior elemento é, a cada modificação, levado uma posição para a direita até estar na posição correta.

2.2 Implementação

A seguir, encontra-se o código utilizado na implementação do algoritmo em questão no Python:

```
1 def bubble(V, n):
2     lim = n - 1
3     while lim >= 0:
4         isIncreasing = True
5         for j in range(lim):
6             if V[j] > V[j + 1]:
7                 isIncreasing = False
8                 V[j], V[j + 1] = V[j + 1], V[j]
9         if (isIncreasing == True):
10             break
11         lim -= 1
```

Caso a lista já esteja ordenada, o `isIncreasing` continuará, na primeira execução da malha de repetição, com o valor `True` e, por isso, o comando `break` será executado, parando, assim, a ordenação.

Constata-se, dessa forma, que, ao contrário do algoritmo anterior, este não realizará todas as etapas caso a lista já esteja ordenada, pois a variável `isIncreasing` funciona como um verificador.

2.3 Quantidade de comparações

Como a quantidade de comparações realizadas pelo bolha depende das características iniciais da lista, analisar-se-ão os casos.

Caso a lista com n elementos já esteja ordenada, o algoritmo detectará a ordenação na primeira passagem, pois a variável `isIncrease` continuará com o seu valor booleano verdadeiro após a malha de repetição mostrada na implementação, ou seja, serão realizadas n comparações.

Por outro lado, na hipótese da lista está totalmente desordenada, haverá $\frac{n(n+1)}{2}$ comparações, pois, pelo mesmo motivo do inserção, serão feitas n comparações na primeira iteração, $n - 1$ na segunda, e assim sucessivamente até a última, na qual será feita 1 comparação.

Finalmente, nos outros casos, a análise é mais complexa e depende de outros fatores que não estão no escopo do relatório. Contudo, em casos nos quais a lista não está nem ordenada, nem totalmente desordenada, o algoritmo possui um comportamento quadrático[1].

3 Algoritmo de inserção

O algoritmo de inserção possui, como principal característica, a asseguaração de que, a cada iteração n , os n -ésimos primeiros elementos estarão ordenados. Além disso, assim como o de bolha, este varia com a porcentagem de ordenação da lista recebida, ou seja, caso a lista já esteja ordenada, será detectado logo na primeira passagem e, assim, não haverá nada a ser feito.

3.1 Exemplo

Seja, para fins de exemplificação, V a lista $[4, 3, 2, 1, 0]$.

O algoritmo em questão funcionará da seguinte forma: a lista começa dada como ordenada até que se ache um j tal que $V[j] > V[j + 1]$, onde j é um inteiro maior ou igual a zero e menor que o tamanho da lista menos um. Caso isso ocorra, o algoritmo trocará os dois valores e comparará, da mesma forma, $V[j - 1]$ e $V[j]$. Quando o valor da antiga posição j for menor do que a posição sucessora, garante-se, então, que a lista está ordenada de 0 até $j + 1$. Contudo, caso não exista um j , então o algoritmo indica que a lista já está ordenada e, assim, evita mais comparações.

Logo, tomando a lista M , caso esta fosse impressa a cada modificação, os resultados seriam:

```
1 [4, 3, 2, 1, 0]
2 [3, 4, 2, 1, 0]
3 [3, 2, 4, 1, 0]
4 [2, 3, 4, 1, 0]
5 [2, 3, 1, 4, 0]
6 [2, 1, 3, 4, 0]
7 [1, 2, 3, 4, 0]
8 [1, 2, 3, 0, 4]
9 [1, 2, 0, 3, 4]
10 [1, 0, 2, 3, 4]
11 [0, 1, 2, 3, 4]
```

Visivelmente, observa-se que, sempre que é encontrado um elemento e menor que o anterior, ambos são trocados e, em seguida, o algoritmo começa a verificar e mover e para trás até o elemento em questão fique em sua posição correta.

3.2 Implementação

Como se pode perceber, quando é encontrado um elemento j menor do que o anterior, este é levado para trás até que esteja na posição correta

```
1 def insertion(V, n):
2     last_index = 0
3     for i in range(last_index, n - 1):
4         if V[i] > V[i + 1]:
5             j = i
6             while V[j] > V[j + 1]:
7                 V[j + 1], V[j] = V[j], V[j + 1]
8                 if j <= 0:
9                     break
10            j -= 1
```

3.3 Quantidade de comparações

Tendo em vista que as comparações realizadas depende da lista, os casos serão analisados.

Caso a lista com n elementos esteja em ordem, o algoritmo detectará na primeira iteração devido à permanência do valor verdadeiro da variável `isIncreasing`, ou seja, serão realizadas n comparações.

Por outro lado, na hipótese da lista está totalmente desordenada, haverá $\frac{n(n+1)}{2}$. A prova é análoga à do bolha, uma vez que haverá n comparações na primeira execução, $n-1$ na segunda, e assim por diante, ou seja, serão realizadas $n + (n-1) + (n-2) + \dots + 1 = \frac{n(n+1)}{2}$ comparações.

Finalmente, nos outros casos, a análise é, assim como no algoritmo anterior, complexa. Contudo, em cenários nos quais a lista não se enquadra nos casos supracitados, o algoritmo possui um comportamento quadrático[2].

4 Algoritmo de contagem

Diferentemente dos algoritmos outrora discutidos, o contagem possui uma complexidade linear, isto é, o tempo levado para ordenar cresce proporcionalmente com o tamanho da lista recebida.

Em relação ao uso de memória, contudo, há uma grande desvantagem no algoritmo: devido ao uso de uma lista auxiliar, esta possuirá $\max(lista) - \min(lista)$ elementos; ou seja, caso o maior elemento seja 5000 e o menor 1000, a lista auxiliar possuirá de 4000 elementos.

Dessa forma, o algoritmo possui um melhor proveito se utilizado para listas com pouca variação de tamanho entre os seus elementos.

4.1 Exemplo

Seja M a lista $[4, 2, 3, 1, 4, 2, 2, 0]$. Como a diferença entre o maior e o menor é elemento é 4, a lista auxiliar Aux será de tamanho 4. Assim, a lista auxiliar Aux será da seguinte forma (assumindo que o primeiro elemento é $Aux[0]$): o primeiro elemento terá o valor igual à quantidade de zeros na lista original, que é 1, o segundo elemento também será 1 por motivo análogo, e o terceiro elemento de Aux , por outro lado, receberá o valor 3, uma vez que há três elementos 2 na lista original. Essa lógica prevalecerá até último elemento. Assim, de início, a lista Aux será $[1, 1, 3, 1, 2]$.

Em seguida, começará a modificação da lista original: serão adicionados à lista original n vezes o elemento referente ao índice da lista auxiliar, ou seja, como $Aux[0]$ é igual a 1, será adicionado um zero a M (adicionando da esquerda para a direita e um ao lado do outro). Em $Aux[2]$, por exemplo, o elemento é 3 e, assim, será adicionado o elemento 2 três vezes seguidas à lista M .

Seguem os valores de `Aux` a cada modificação:

```
1 [0, 0, 0, 0, 0]
2 [1, 0, 0, 0, 0]
3 [1, 1, 0, 0, 0]
4 [1, 1, 3, 0, 0]
5 [1, 1, 3, 1, 0]
6 [1, 1, 3, 1, 2]
```

Para fins elucidativos, serão substituídos por "0" os elementos já existentes em M a fim de mostrar os valores adicionados posteriormente, ou seja, supor-se-á que a lista M teve todos os seus valores trocados para 0.

```
1 [0, 0, 0, 0, 0, 0, 0, 0]
2 [0, 0, 0, 0, 0, 0, 0, 0]
3 [0, 1, 0, 0, 0, 0, 0, 0]
4 [0, 1, 2, 0, 0, 0, 0, 0]
5 [0, 1, 2, 2, 0, 0, 0, 0]
6 [0, 1, 2, 2, 2, 0, 0, 0]
7 [0, 1, 2, 2, 2, 3, 0, 0]
8 [0, 1, 2, 2, 2, 3, 4, 0]
9 [0, 1, 2, 2, 2, 3, 4, 4]
```

Percebe-se, portanto, que a lista foi ordenada realizando-se nenhuma comparação.

4.2 Implementação

Segue a implementação do algoritmo em questão em Python:

```
1 def counting(V, n):
2     max_element = max(V)
3     hist_list = [0 for _ in range(max_element + 1)]
4     for i in range(max_element + 1):
5         hist_list[i] = count_element_in_array(i, V)
6     index = 0
7     for i in range(max_element + 1):
8         for _ in range(hist_list[i]):
9             V[index] = i
10            index += 1
```

Como se pode perceber, o algoritmo de contagem, assim como os outros, possui duas malhas de repetição. Contudo, em vez de as malhas estarem aninhada, uma ocorre após a outra. Apesar disso, uma singularidade desse algoritmo é que, ao contrário dos citados, a quantidade de iterações na primeira malha de repetição depende do tamanho do maior elemento (pensando apenas na implementação com números positivos), tornando-o eficaz para listas grandes e com elementos menores.

4.3 Quantidade de comparações

Diferentemente de todos os algoritmos outrora citados, este não realiza qualquer comparação.

5 Resultados

Nesta seção, será debatido acerca dos resultados obtidos por meio dos dois testes realizados. O primeiro visou analisar o tempo médio de cada algoritmo, bem como a sua variância. No segundo teste, por outro lado, buscou-se avaliar o comportamento dos algoritmos bolha e inserção quando submetidos a listas com diferentes taxas de ordenação, sendo estas 1%, 3%, 5%, 10% e, por fim, 50%.

5.1 Variação na quantidade de elementos

Como comentado, os resultados comprovam o comportamento quadrático do bolha, seleção e inserção. Entretanto, é preciso ressaltar que, se existisse uma variação significativa nos números da lista a ser ordenada, o algoritmo de contagem apresentaria um tempo para ordenar maior, podendo ultrapassar o bolha, por exemplo.

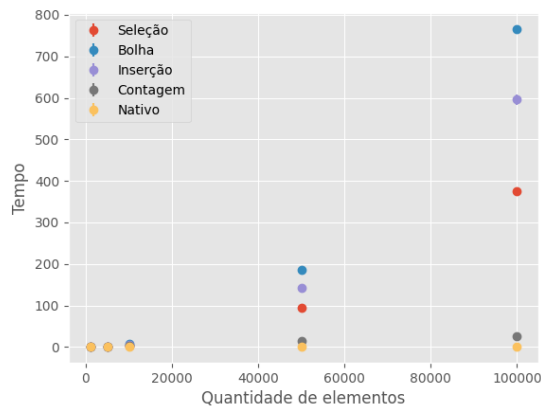


Figura 1: Gráfico elucidando o comportamento de cada algoritmo em tópico. No eixo x, há a quantidade de elementos; no eixo y, o tempo médio gasto em segundos

O bolha, apesar de variar dependendo da taxa de ordenação, possui o maior tempo necessário para ordenar. Em seguida, o algoritmo de inserção detém o segundo maior período para organizar a lista, e isso se dá pelo fato de que o inserção possui uma complexidade quadrática independente da posição inicial dos elementos.

O algoritmo de seleção, como esperado, contém o melhor tempo de resposta dentre os algoritmos de comparação, tendo em vista que, além de ser mais eficiente em listas com alguns elementos ordenados, realiza menos permutações que o bolha.

Por último, o contagem apresentou o melhor desempenho dos implementados. Contudo, é necessário destacar que o contagem obteve este performance devido à faixa de números aleatórios escolhida (de 0 a 9999), pois, como analisado, o algoritmo em questão varia com os números recebidos.

Assim, percebe-se que, para listas com muitos elementos, o algoritmo contagem é o mais eficiente entre os implementados, enquanto o bolha apresentou o resultado menos satisfatório para o teste.

5.2 Ordenação prévia

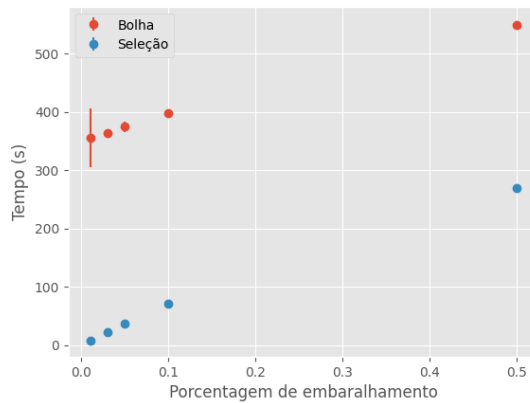


Figura 2: Gráfico ilustrando o tempo médio 'gasto' dos algoritmos bolha e inserção em relação à taxa de ordenação inicial dos vetores.

Inicialmente, quando ambos os algoritmos são submetidos a uma lista já ordenada, o tempo de execução do algoritmo de seleção é quase imediato; o bolha, por outro lado, possui uma duração considerável para verificar que a lista já está ordenada. Adicionalmente, ao aumentar a permutação dos elementos, o bolha mantém a sua alta duração, principalmente devido à quantidade de comutações realizadas por este.

Por outro lado, o algoritmo de seleção apresentou um crescimento maior quando intensificada a desordenação da lista, uma vez que a quantidade de comutações realizadas pelo algoritmo tende a se aproximar a do bubble, ou seja, quanto mais ordenada a lista, mais eficiente o seleção é em comparação ao segundo.

Portanto, o algoritmo de seleção é mais eficiente do que o bolha, ainda quando submetido a listas com diferentes taxas de ordenação. Contudo, o seleção possui uma taxa de crescimento ligeiramente maior do que o bolha quando aumentada a porcentagem de embaralhamento.

6 Conclusão

Desta maneira, apesar de terem diferentes desempenhos dependendo da ordenação da lista, os algoritmos em questão possuem um baixo proveito com listas muito grandes. Portanto, a fim de ordenar uma lista com maior rapidez, é aconselhável utilizar a função já implementada do Python, conhecida como `timsort`.

Adicionalmente, o algoritmo de contagem, ainda que o vetor passado tenha poucos elementos, pode ter um tempo de execução muito alto em razão da variação entre os números. Assim, é aconselhável a utilização do algoritmo em pauta apenas com listas com uma exígua variação.

Em outra perspectiva, o bolha foi o algoritmo que apresentou o menor rendimento quando submetido as listas com diferentes tamanhos; o contagem, por outro lado, garantiu, apesar da faixa de valores maior, o melhor desempenho, principalmente nas listas com mais elementos. Dessa forma, a utilização do bolha é recomendada apenas para fins didáticos.

7 Implementação em C

Nesta seção, buscar-se-á analisar os algoritmos comentados em C a fim de analisar os impactos causados no desempenho quando executado um algoritmo em uma linguagem interpretada, tal como o Python.

Após visto o comportamento dos algoritmos quando implementados em Python, uma linguagem interpretada, percebe-se que, mesmo para vetores pequenos, o tempo gasto para ordená-los é significativo.

Quando implementados utilizando a linguagem C, contudo, percebe-se uma considerável redução na duração dos testes, uma vez que, em razão do C ser uma linguagem compilada, as malhas de repetições tendem a ser mais eficientes. Ademais, ao utilizar a linguagem C, há um maior controle na forma como o código será executado, devido ao C ser uma linguagem com um nível mais baixo. Dessa forma, serão analisados os desempenhos de alguns dos algoritmos já implementados em Python, mas, desta vez, estes serão escritos na linguagem C.

A fim de realizar o testes, os algoritmos foram implementados na linguagem C, utilizando, como referência, os códigos já escritos em Python. Sendo assim, tendo em vista que os códigos utilizados em C não foram otimizados e nem modificados de forma a deixá-los mais lentos, ter-se-á uma comparação equânime.

Além disso, as funções foram invocadas no Python visando criar os gráficos dos resultados.

Foram realizados somente testes visando analisar o comportamento dos algoritmos em C quando estes são submetidos a listas com tamanhos variáveis, ou seja, o segundo teste não será executado pois..

7.1 Resultados

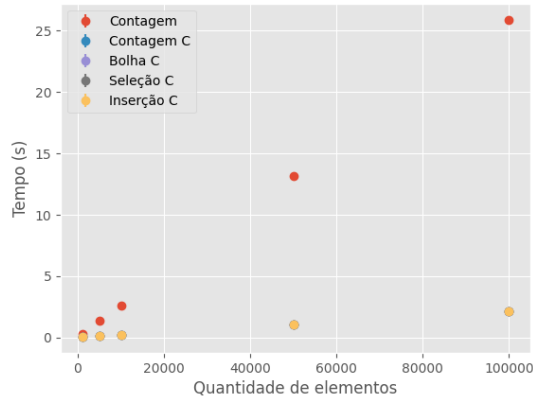


Figura 3: Gráfico ilustrando o tempo médio de execução dos algoritmos implementados em C comparado ao do contagem, implementado em Python.

Perceptivelmente, o ganho de desempenho dos algoritmos implementados em C mostrou-se significativo, mesmo os mais lentos quando implementados em Python, como o bolha, obtiveram uma performance superior ao contagem da linguagem inicial.

De fato, o contagem implementado em C mostrou-se $\approx 50\%$ mais eficiente do que a sua versão em Python. Dessa maneira,

Ademais, nota-se que, ainda que com listas pequenas, o desempenho dos algoritmos em C em relação aos do Python permaneceu quase constante, isto é, ao aumentar o tamanho das listas, a implementação em C não obteve um ganho nem uma perda expressiva. Consequentemente, ...

Portanto, visando um ganho de performance, é recomendável a utilização da linguagem C a fim de implementar algoritmos que se "aproveitam" de malhas de repetições. Caso seja necessário o uso do Python para outras tarefas no projeto, as funções em C podem ser chamadas por meio de ...

8 Considerações finais

A realização deste relatório possibilitou o aprendizado em diversas áreas do conhecimento, especialmente na análise de algoritmos, ao separá-los por casos e verificá-los individualmente. Além disso, o projeto proporcionou uma ampla experiência na criação de textos acadêmicos e na investigação da eficiência e do comportamento de um algoritmo. Portanto, o exercício programa permitiu um aprimoramento em competências que serão de extrema importância para futuros projetos.

Referências

- [1] Bubble Sort Time Complexity and Algorithm Explained, builtin, 2023. Disponível em: [https://builtin.com/data-science/bubble-sort-time-complexity#:~:text=The%20bubble%20sort%20algorithm%27s%20average,complexity%3A%20O\(n%C2%B2\).](https://builtin.com/data-science/bubble-sort-time-complexity#:~:text=The%20bubble%20sort%20algorithm%27s%20average,complexity%3A%20O(n%C2%B2).) Acesso em: 08 de jun. de 2024.
- [2] Insertion Sort Explained—A Data Scientists Algorithm Guide, 2021. Disponível em: [https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/#:~:text=The%20worst%2Dcase%20\(and%20average,O\(n\)%20time%20complexity..](https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/#:~:text=The%20worst%2Dcase%20(and%20average,O(n)%20time%20complexity..) Acesso em: 08 de jun. de 2024.