UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

Renzo Real Machado Filho

Algoritmos de Ordenação: Uma análise inaugural

São Paulo
2024

Sumário

1 Introdução

Um algoritmo é um conjunto de instruções para a resolução de um problema. Ora! Então, um algoritmo de ordenação é uma série de passos para organizar objetos. Na computação, tais objetos são, em geral, listas e/ou vetores de números ou de palavras, que podem ser ordenados de forma crescente, descrescente ou até mesmo conforme a ordem lexicográfica.

Nessa perspectiva, agrupar elementos é, sem dúvidas, algo comum em nosso dia a dia, estando presente desde áreas técnicas, como a gestão e a análise de dados, o *machine learning* e a otimização de sistemas, até para atividades mais rotineiras, como navegar na web e pesquisar seu vídeo favorito. Daí a importância desse assunto para a computação e a sociedade.

Em vista disso, nessa pesquisa avaliaremos alguns aspectos que atuam diante a execução desses algoritmos. Para isso, vamos analisar experimentalmente o comportamento de cinco deles, o *Selection Sort*, o *Bubble Sort*, o *Insertion Sort*, o *Counting Sort* e, por fim, o *sort* nativo do python, chamado de *Timsort*.

Como deve esperar, utilizaremos a linguagem *python*, com auxílio de bibliotecas gráficas para a melhoria na visualização dos resultados dos experimentos, que visaram comparar tais algoritmos frente a diferentes parâmetros presentes nas listas. Em outro momento, nos apoiaremos na rapidez da linguagem C, para comparar a performance desses mesmos algoritmos em relação ao *python*.

2 Referencial Teórico

2.1 Critérios de Performance e a Notação *Big O*

Antes de prosseguir para nosso estudo, vale introduzir alguns critérios que são utilizados na análise da performance dos algoritmos de ordenação.

- **Tempo:** Avalia-se o tempo gasto na execução do algoritmo em função do tamanho da entrada. Em termos técnicos, é comum a utilização de notações matemáticas, como a *notação Big O*, para classificar a complexidade de tempo.
- **Memória:** Avalia-se o quanto de memória adicional será necessária para a sua execução. Assim, alguns algoritmos de ordenação são comumente denotados como *out-of-place algorithms* para dizer que requerem porções extras de memória, e *in-place algorithms*, caso contrário.
- **Estabilidade:** Trata-se da capacidade de um algoritmo de preservar a ordem relativa de seus elementos originais.

Ademais, como dito anteriormente, a notação *Big O*, ou Grande O, é um modelo matemático que descreve o comportamento assintótico de uma função, i.e, analisa o que ocorre quando uma função cresce para determinados valores suficientemente grandes. Na computação, usa-se a notação *Big O* para se classificar algoritmos tanto em relação ao tempo de execução quanto ao espaço de memória utilizado.

A seguir, iremos estudar um pouco o conceito e as características de cada um deles. Atente-se que a discussão acerca dos critérios listados será reservada para a seção 4 desse artigo.

2.2 O *Selection Sort*

Esse será o primeiro algoritmo abordado e, para introduzí-lo, nos apoiaremos na seguinte descrição:

> "Um dos mais simples algoritmos de ordenação que funciona da seguinte maneira: Primeiro, encontre o menor elemento do vetor e troque-o com a primeira entrada. Então, encontre o próximo menor elemento e troque-o com a segunda entrada. Continue assim até que todo o vetor esteja ordenado" (Sedgewick, Wayne, 2011, p. 248, tradução nossa)

A seguir temos a implementação que foi utilizada desse algoritmo:

```
1
2  def selection(V, n):
3
4      """ Função que, dado uma lista `V` de tamanho `n`, ordena os elementos
5      desse vetor segundo o método selection sort.
6      Note que temos como parâmetros tal vetor `V` e seu tamanho `n`.
7      Não há valores de saída. """
8
9      for i in range(n):
10         # iteração que percorrerá de até o tamanho do Vetor fornecido,
11         # este primeiro loop fixará o elemento
12         for j in range(i+1, n):
13             # essa segunda iteração terá a função de comparar o elemento
14             # fixado com os elementos seguintes do Vetor
15             if(V[i] > V[j]):
16                 # trocaremos a posição os elementos,
17                 # se a condição anterior for satisfeita
18                 swap = V[i]
19                 V[i] = V[j]
20                 V[j] = swap
21
```

Note que a implementação desse algoritmo é bastante espontânea e, em termos de construção, temos a presença de dois *loops* para a varredura da lista desordenada. Com isso, espera-se cerca de n^2 comparações até sua ordenação completa. Dessa forma, nosso experimento buscará entender a performance do *Selection Sort*.

2.3 O *Bubble Sort*

Trata-se de um algoritmo bastante intuitivo que a cada iteração compara dois elementos, "empurrando" o maior valor para direita. Essa descrição remonta a metáfora que nomeia o algoritmo, no qual os maiores elementos "borbulham" até o final da lista. Abaixo, poderá conferir nossa implementação:

```
1
2  def bubble(V, n):
3
4      """ Função que, dado uma lista `V` de tamanho `n`, ordena os elementos
5      desse vetor segundo o método bubble sort.
6      Note que temos como parâmetros tal vetor `V` e seu tamanho `n`.
7      Não há valores de saída. """
8
9      houveTroca = False
10
11     for i in range(1,n):
12     # iteração que decrescerá a necessidade de verificação do Vetor,
13     # pois na linha seguinte terá: range(n-i)
14         for j in range(n-i):
15             # essa iteração irá atuar percorrendo o Vetor,
16             # mas a cada iteração terá que Vericar uma posição a menos
17             if (V[j] > V[j+1]):
18             # troca a posição dos elementos
19                 swap = V[j+1]
20                 V[j+1] = V[j]
21                 V[j] = swap
22
23                 houveTroca = True
24                 # indicador de passagem
25
26         if(not houveTroca): # melhor caso, o array já está ordenado
27             break
28
```

Essa versão procura a melhor otimização no processo de ordenação, verificando, atráves de um indicador de passagem, se houve trocas durante cada iteração, o que revelará o caso no qual a lista já está ordenada. Veja que, assim como o *Selection Sort*, temos dois *loops* para varrer as posições da lista. Por isso, novamente, estimamos uma média de n^2 comparações. Na seção destinada aos testes, verificaremos essa proposição.

2.4 O *Insertion Sort*

Esse é um famoso algoritmo que lembra muito a prática de ordenação em jogos de cartas. Consiste em comparar dois elementos e **inserí-los** na posição

correta, considerando os elementos anteriores já ordenados. Assim, é necessário, a cada iteração, mover os elementos desordenados "à frente" na lista e/ou no vetor.

Em nossa pesquisa, utilizamos duas implementações: uma versão autoral e outra otimizada, respectivamente. Elas podem ser vistas a seguir:

```
1
2  def insertion(V, n):
3
4      """ Funcao que, dado uma lista `V` de tamanho `n`, ordena os elementos
5      desse vetor segundo o metodo insertion sort.
6      Note que temos como parametros tal vetor `V` e seu tamanho `n`.
7      Nao ha valores de saida. """
8
9      for i in range(1,n):
10     # Percorreremos o Vetor, note que a condicao esta
11     # na forma V[i] < V[i-1], por isso partiremos de 1
12
13         if (V[i] < V[i-1]):
14         # se encontrarmos um elemento fora de ordem, trocamos suas posicoes
15             swap = V[i]
16             V[i] = V[i-1]
17             V[i-1] = swap
18
19             for j in range(1,i):
20             # Uma iteracao decrescente para voltarmos posicoes no Vetor,
21             # fazendo a mesma comparacao
22
23                 if (V[i-j] < V[i-j-1]):
24                 # Essa iteracao decrescente permite corrigir
25                 # elementos fora de ordem nas posicoes anteriores
26                     swap = V[i-j]
27                     V[i-j] = V[i-j-1]
28                     V[i-j-1] = swap
29                 else:
30                     break
```

```
1
2  def insertion_opt(V, n):
3
4      """ Versao otimizada do metodo de ordenacao Insertion Sort.
5      Temos como parametros um vetor `V` de tamanho `n`.
6      Nao ha valores de saida. """
7
8      for i in range(1, n):
9          key = V[i]
```

```
        # guarda o elemento que sera inserido na posicao correta
        j = i - 1
        # indice do elemento anterior a posicao V

        while j >= 0 and V[j] > key:
        # move os elementos maiores para a direita
            V[j + 1] = V[j]
            j = j - 1

        V[j + 1] = key
        # insere o elemento na posicao correta
```

Como é capaz de observar, as implementações são muito distintas. Na primeira, percebemos uma maior semelhança com os algoritmos anteriores, afinal, há, mais uma vez, dois *loops* que percorrem as posições da lista. Entretanto, o último *loop* busca de modo reverso, para garantir a ordenação dos elementos anteriores. Já a segunda implementação conta com o auxílio de uma variável *key*, ou chave, para armazenar o atual elemento da lista. Com isso, utiliza-se de outros dois *loops* para descobrir a posição correta desse item.

Analisaremos, mais a frente, a performance empírica dessas versões.

2.5 O *Counting Sort*

Nos últimos três algoritmos estudados, a ordenação estava baseada no processo de comparação entre elementos. Porém, o *Counting Sort* se ampara em uma outra estrutura: a ordenação por distribuição. Tal conceito utiliza ferramentas intermediárias para alcançar seu objetivo final.

Com isso, em sua execução, busca-se o maior valor dentro da lista e/ou vetor, tomemos um vetor V. Após essa informação, cria-se um vetor auxiliar em que será armazenado as ocorrências de cada elemento presente em V. Por fim, o vetor V será limpado e, seguindo a ordem de ocorrências, adicionamos os elementos já em ordem.

Obs: É válido ressaltar que nossa implementação levou em consideração listas unicamente de valores positivos. Aqui temos um exemplo:

```

def counting(V, n):

    """ Função que, dado uma lista `V` de tamanho `n`,
    ordena os elementos desse vetor segundo o método counting sort.
    !!! Estamos tratando apenas com números positivos !!!
    Note que temos como parâmetros tal vetor `V` e seu tamanho `n`.
    Não há valores de saída. """

    max = V[0]
    # esse primeiro loop busca o maior valor dentro do vetor fornecido
```

```
12      for i in range(n-1):
13          if (max < V[i+1]):
14              max = V[i+1]
15
16      assistant = [0 for i in range(max+1)]
17      # cria um vetor auxiliar/assistente
18
19      for j in range(n):
20      # nesse terceiro loop, contamos as ocorrências dos valores de V
21      # e adicionamos no vetor auxiliar
22          assistant[V[j]] = assistant[V[j]] + 1
23
24      V.clear()
25      # limpamos o vetor original para ordená-lo a seguir
26
27      for k in range(max+1):
28          for l in range(assistant[k]):
29              V.append(k)
30      # nesses quarto e quinto loops,
31      # acessaremos o vetor auxiliar que, por sua vez,
32      # mostrará as ocorrências dos elementos.
33      # Assim, adicionaremos no vetor original tais elementos de forma ordenada.
34
```

Em posse do código fonte, podemos afirmar que esse método usa uma técnica bastante elegante para ordenar listas, que foge da nossa noção habitual. Ainda, torna-se mais claro a ideia de "contagem", que nomeia o algoritmo. Perceba que nos *loops* implementados há uma minimização da necessidade de comparar a grandeza dos valores presentes na lista. Por outro lado, tal algoritmo parece exigir espaços extras de memória para sua execução, o que pode ser um fator limitante. De qualquer maneira, nosso experimento verificará se essa nova postura influencia a complexidade de tempo do algoritmo. Além disso, note temos uma variável *max* que recebe o maior elemento da lista, veremos, também, como o algoritmo se comporta quando esse valor é suficientemente grande.

3 Metodologia

Como dito anteriormente, essa pesquisa foi desenvolvida com o objetivo de analisar empiricamente a complexidade dos algoritmos de ordenação clássicos. Assim, utilizamos a linguagem *Python* em sua versão 3.12.3. Tratando-se do código-fonte, os procedimentos foram divididos em funções que executavam testes diferentes. Cada teste, essencialmente, iterava a execução dos algoritmos estudados um certo número de vezes, coletando as médias e os desvios dos tempos de execução para, em seguida, produzir gráficos, com auxílio da biblioteca *matplotlib* do python.

Para a segunda parte do projeto, implementamos os mesmos algoritmos citados na linguagem C. Com isso, geramos uma biblioteca compartilhada que poderia ser chamada no *Python* através da biblioteca *ctypes*.

Vale ressaltar que os experimentos foram realizados em uma máquina pessoal com processador *Intel (R) Core (TM) i7–8565U CPU @ 1.80GHz 1.99 GHz* rodando em uma versão 22H2 do Windows 10.

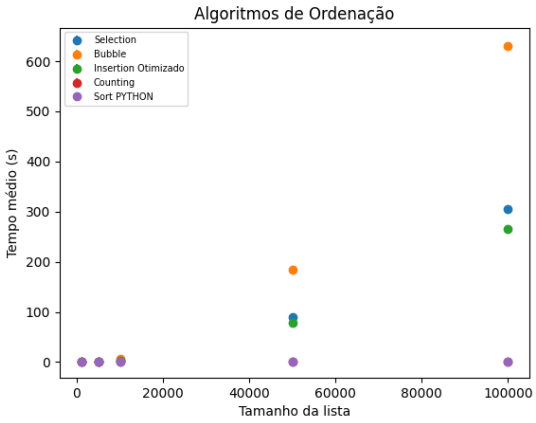
4 Desenvolvimento

4.1 Qual a complexidade empírica de tempo dos algoritmos de ordenação clássicos?

Nessa primeira parte da pesquisa, buscamos entender a performance dos seguintes algoritmos: *Selection Sort, Bubble Sort, Insertion Sort* e *Counting Sort*. Para isso, realizamos testes comparativos que tinham o propósito de calcular seus tempos médios e o seus desvios de execução à medida que se variava o tamanho da lista desordenada. Assim, pela conveção da pesquisa, a cada nova iteração, uma nova lista desordenada era criada com 1.000, 5.000, 10.000, 50.000 e 100.000 posições.

Dessa forma, com auxílio da biblioteca *matplotlib* obtivemos o seguinte resultado:

Figura 1: Gráfico comparativo entre os algoritmos estudados



Perceba na fig.1 que os algoritmos *Bubble Sort, Selection Sort e Insertion Sort* apresentaram um comportamento quadrático, i.e, o esboço dos pontos aproximam-se de uma curva da forma $f(x) = x^2$, enquanto o *Counting Sort* e o mecanismo de *sort* nativo do *python* demonstraram um comportamento constante no intervalo analisado. Mais do que isso, esses dois últimos se mostraram tão rápidos que torna-se difícil diferenciá-los no gráfico.

Com esses dados, uma de nossas hipóteses apresentadas na seção 2 se confirma. De fato, estimamos uma média de n^2 comparações para o *Selection Sort*, o *Bubble Sort* e o *Insertion Sort*, o que traduz, no gráfico acima, em um crescimento rápido da média de tempo conforme se aumenta o tamanho das listas. Assim, a conclusão mais imediata é que algortimos com complexidades quadráticas perdem eficiência em situações onde é preciso lidar com tamanhos muito grandes de listas. Como exemplo, para listas de 50.000 e 100.000 posições, os três algortimos citados distoaram-se significativamente dos demais e, em especial, o *Bubble Sort* levou, em média, cerca de 600 segundos, i.e, cerca de 10 minutos para ordenar tal lista, sendo o menos eficaz do grupo analisado.

Uma explicação razoável se dá em razão do número de comparações e/ou verificações realizadas por esses algoritmos. Tanto *o Selection Sort* como o *Bubble Sort* e o *Insertion Sort*, em sua construção, percorrem *loops* para descobrir a posição correta de cada elemento, o que, em casos mais extremos, como esse analisado (uma lista da ordem de 10^5 posições), faz com que o tempo médio de execução dispare. Portanto, na utilização de algum desses três algoritmos deve-se considerar o fator de crescimento das listas, pois, certamente, irá afetar seu tempo de execução.

Em contrapartida, o *Counting Sort* e o *Timsort* mostraram uma grande eficácia no intervalo analisado, praticamente não sofrendo alterações em seu tempo médio de execução. Contudo, em relação a suas performances específicas, os resultados vistos no gráfico, de modo isolado, mostraram-se inconclusivos, principalmente, devido ao intervalo de precisão. Por esse motivo, mais a frente dedicaremos uma seção desse artigo para discutir com mais detalhes suas performances.

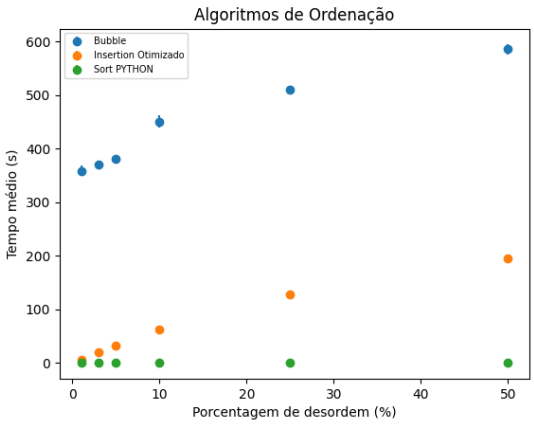
Além disso, ambos, em suas estratégias de ordenação, utilizam espaços extras de memória durante sua execução, ao contrário dos algoritmos vistos anteriormente. Esse fator também deve ser levado em consideração, pois, somente pela análise do gráfico, não é possível quantificar esse consumo extra de mémoria e nem mesmo como ele pode impactar a eficácia de tais algoritmos.

4.2 Como a complexidade de tempo de um algoritmo é afetada pela atual configuração dos elementos da lista?

Nessa segunda parte da pesquisa, um novo questionamento surge: a atual configuração dos elementos pode influenciar o tempo de execução? Para isso, realizamos uma nova tomada de testes em que foi fixado um tamanho padrão da lista: 100.000 posições. Vale lembrar que a lista fornecida já estava ordenada e nosso objetivo era analisar se o grau de desordem alteraria a complexidade de tempo dos seguintes algoritmos: *Bubble Sort, Insertion Sort e a função de sort nativa do python.* Por convenção, criamos uma função *embaralha* que a cada iteração trocava posições proporcionais a 1%, 3%, 5%, 10%, 25% e 50% do tamanho da lista já ordenada.

Desse modo, mais uma vez, com auxílio da biblioteca *matplotlib* obtivemos o seguinte resultado:

Figura 2: Gráfico comparativo em relação a desordem da lista



Note na fig.2 que, para porcentagens mínimas, tanto o *Bubble Sort* como o *Insertion Sort* apresentam comportamentos bem similares, aparentando uma tendência linear. No entanto, com o avanço das iterações e, consequentemente, com o crescimento da desordem na lista, os algoritmos anteriores mostraram um desempenho logarítmico. Isso sintetiza que o crescimento do grau de desordem da lista faz com que a disposição dos elementos aproxime-se da situação comum, na qual se fornece uma lista desordenada. Por isso, ambos os algoritmos tendem a estabilizar seus tempos médios de execução, já que, como dito, a lista previamente ordenada passa a se tornar uma lista desordenada.

Enquanto isso, o modelo de *sort nativo do python* novamente surpreende estabelecendo um comportamento constante, i.e, não sendo afetado pela desordem na lista. Novamente, em outra seção, discutiremos com mais detalhes a execução desse algoritmo, principalmente, em listas com tamanhos maiores.

4.3 Como diferentes implementações afetam a complexidadede de tempo do *Insertion Sort*?

Um leitor atento pode ter percebido que, nos resultados e nos gráficos mostrados anteriormente, destacamos a utilização da versão otimizada do algoritmo *Insertion Sort*. Entretanto, durante a sua implementação, uma versão menos eficiente, de minha autoria, também foi testada. Com isso, as diferenças nos resultados motivaram a realização de outro experimento comparativo, que será discutido nessa seção.

Para sua execução, repetimos os critérios vistos nas seções 4.1 e 4.2, nos quais fornecíamos, de início, listas com tamanhos que variavam de 1.000, 5.000, 10.000, 50.000 e 100.000 posições, e, posteriormente, ao se fixar uma lista ordenada com 100.000 posições, analizávamos o comportamento dos algoritmos quanto ao seu grau de desordem: 1%, 3%, 5%, 10%, 25% e 50%. Veja abaixo:

Figura 3: Gráfico comparativo das implementações do *Insertion Sort*

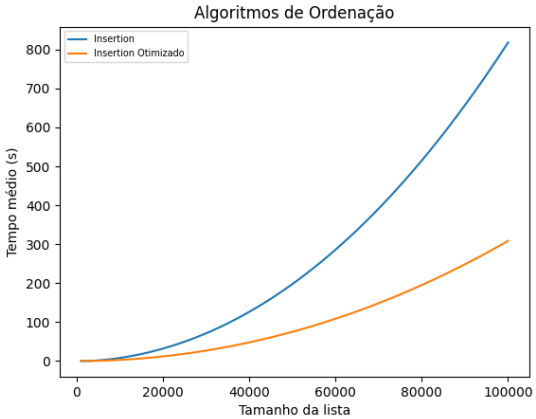
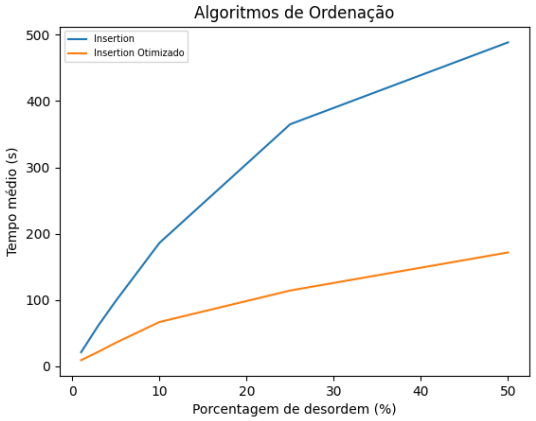


Figura 4: Gráfico comparativo das implementações do *Insertion Sort*, em relação a desordem da lista



De início, uma novidade nesse experimento foi a mudança no modelo de gráfico utilizado na biblioteca *matplotlib*. Ao contrário dos itens anteriores, utilizamos um *plot* curvilíneo na fig.3 e um *plot* padrão na fig.4.

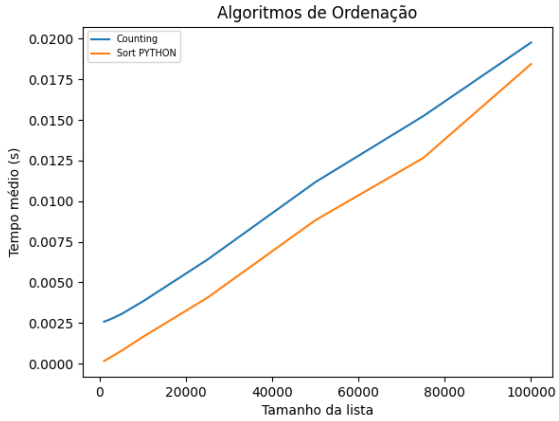
Já se tratando dos resultados empíricos, vemos que, apesar de ambos os algoritmos apresentarem os mesmos comportamentos gerais, o *Insertion Otimizado* cumpre sua função, sendo muito mais rápido dentro do intervalo analisado. Esse fato confirma nossa hipótese, sendo um reflexo direto da quantidade de comparações realizadas (consulte a seção 2.4), que é maior na versão menos eficaz.

4.4 Qual é o verdadeiro comportamento do *Counting Sort* e do *Timsort*?

Como discutimos na seção 4.1, tanto o *Counting Sort* quanto o *Timsort*, no intervalo analisado, demonstraram uma performance extremamente similar, quase que indiferenciável. Pensando nisso, a seguir avaliaremos mais a fundo tal questão. Nosso novo experimento repetiu os críterios anteriores, nos quais, a cada iteração, fornecia-se uma lista com tamanho crescente, mas, dessa vez, aumentamos a passagem do intervalo, que varia de 1.000, 2.500, 5.000, 10.000, 25.000, 50.000, 75.000 e 100.000 posições. Portanto, em linhas gerais, ampliamos o foco e os detalhes do experimento visto na seção 4.1, sob uma visão específica do *Counting Sort* e do *sort nativo do python*.

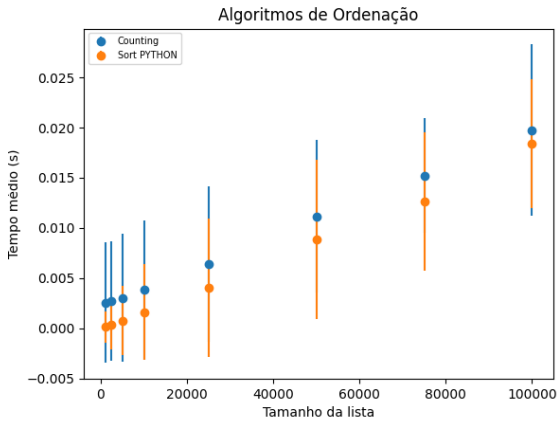
Abaixo mostraremos os gráficos obtidos com, respectivamente, sua representação em um *plot* comum do *matplotlib* e com uma análise de *errobars*.

Figura 5: Gráfico comparativo entre o *Counting Sort* e o *sort do python*



De imediato, nossa hipótese intuitiva se confirma: apesar de mostrarem tempos similares, suas performances são, de fato, distintas. Além disso, os resultados estenderam nossa investigação acerca da complexidade de tempo de tais algoritmos. Previamente, na fig.1 da seção 4.1, ambos mostravam um comportamento constante, porém, na fig.6, obtivemos uma visão mais detalhada, que revela uma performance mais próxima de uma complexidade linear.

Figura 6: *Errobar* comparativo entre o *Counting Sort* e o *sort do python*



Além disso, note que, pela primeira vez nesse artigo, pode-se perceber a presença das taxas de desvio dos algoritmos. Esse fato pode ser justificado, essencialmente, pela proximidade dos algoritmos, que coincidentemente foi o que motivou essa discussão. Assim, obtivemos uma precisão de tempo muito maior no gráfico *errobar*.

Contudo, algumas perguntas ainda podem ser feitas a respeito desses algoritmos, como, por exemplo, o que acontece quando ultrapassamos o limite de 100.000 posições? Ou ainda, como o *Counting Sort* lida com o aumento do intervalo de valores dentro das listas fornecidas? Nas próximas seções, iremos debater um pouco mais sobre essas ponderações.

4.5 Como o *Timsort* se comporta quando há um aumento brusco no tamanho das listas desordenadas?

Até aqui, é fácil admitir a eficiência do algoritmo de *sort do python*. Mesmo ao lidar com listas de 100.000 posições, o seu tempo médio de execução não ultrapassou o valor de 0.3 segundos! É um resultado admirável.

Com isso em mente, o que acontece caso o tamanho da lista aumente para a ordem de 10^6, ou ainda 10^7 posições? Queremos levar esse experimento até seu limite (literalmente) pois, infelizmente, temos uma limitação física de processamento. Por isso, para a realização desse teste mais extremo levamos em conta listas desordenadas dos seguintes tamanhos: 500.000, 1.000.000, 10.000.000, 20.000.000, 25.000.000, 40.000.000 e 50.000.000 posições. A seguir, poderá analisar os resultados. Novamente, temos, respectivamente, um gráfico do tipo *errobar* e outro *plot* padrão do *matplotlib*.

Figura 7: *Errobar* que analisa o desempenho do *Timsort*

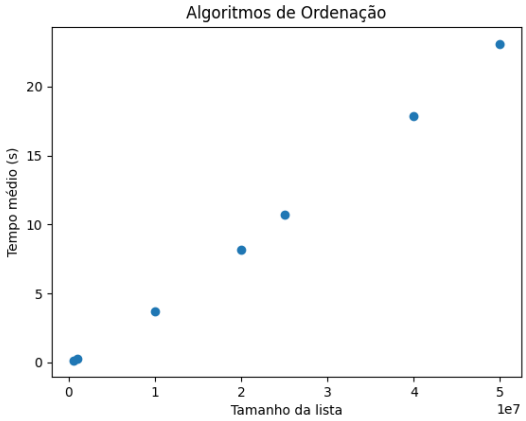
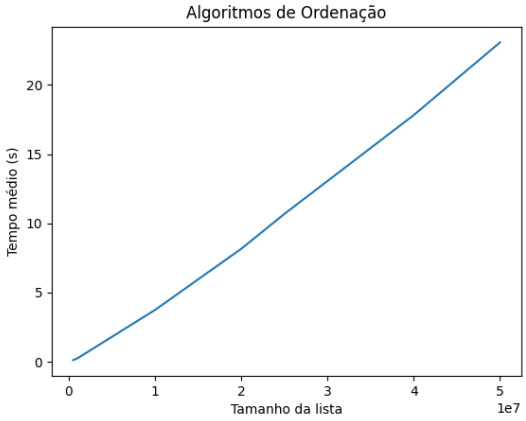


Figura 8: Gráfico que analisa o desempenho do *Timsort*



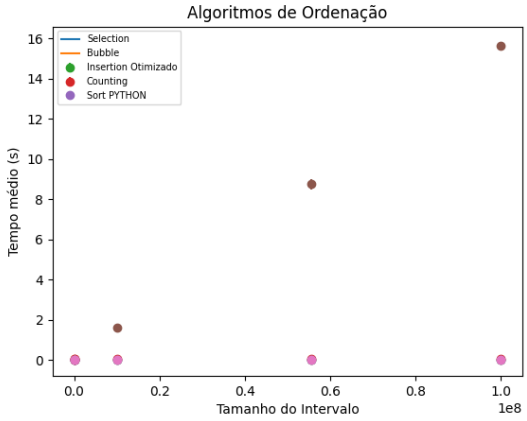
Note que a mudança na ordem de grandeza do tamanho da lista impactou severamente a performance de tempo do *Timsort*, em comparação com os testes anterirores, o que nos possibilitou analisar sua complexidade de tempo sob um novo olhar. A princípio, o resultado visto na seção 4.4 inferia que a complexidade de tempo desse algoritmo se aproximava da linear. Perceba que trajetória do gráfico acima aproxima-se muito de uma reta. Entretanto, essa afirmação deve ser feita com cuidado, pois não temos uma maneira formal para provar esse resultado. Então, vamos nos restringir aos dados obtidos e, assim, trataremos, empiricamente, que o *Timsort* apresenta uma tendência linear à medida que cresce o tamanho da lista no intervalo mostrado.

4.6 Como a grandeza dos valores contidos na lista afeta o desempenho do *Counting Sort*?

Nesse penúltimo experimento, analisaremos como o *Counting Sort* se comporta mediante a alterações no intervalo dos valores presentes na lista. Além disso, estabeleceremos um paralelo com os algoritmos estudados anteriormente.

Dessa maneira, fixamos um único tamanho de lista, apenas 1.000 posições, e, com isso, a cada iteração aumentamos o limite máximo do intervalo, variando até, no máximo, 9.999, 55.555, 9.999.999, 55.555.555 e 99.999.999. Como pode imaginar, assim como o experimento anterior (seção 4.5), existem restrições de processamento, afinal, para intervalos da ordem de 10^8, o gasto extra de memória do *Counting Sort* ultrapassa os limites físicos da máquina utilizada. Os resultados do experimento podem ser verificados nos gráficos abaixo:

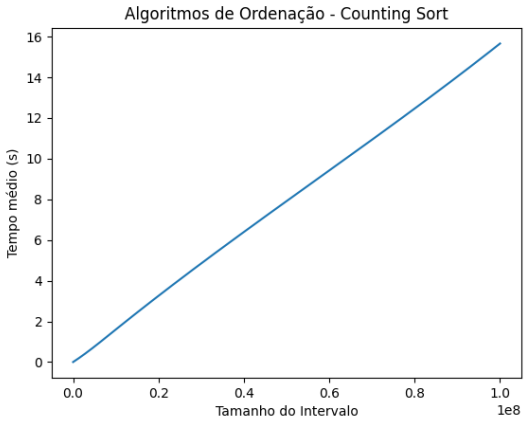
Figura 9: *Errobar* comparativo do desempenho dos algoritmos de ordenação em relação a variação do intervalo de elementos da lista



Atente-se que na primeira iteração utilizamos o intervalo convencional da pesquisa, (0, 9999), o que não impacta o tempo médio de execução dos algoritmos. Entretanto, com o crescimento do fator analisado, nota-se uma perda brusca de eficiência por parte do *Counting Sort*, que, sob uma lista fixa de somente 1.000 posições, leva, em sua última iteração, cerca de 16 segundos para ordená-lo.

A conclusão iminente desse processo se relaciona com o princípio de funcionamento do *Counting Sort*, que busca o maior valor contido na lista e, em seguida, cria uma outra lista auxiliar para a contagem de ocorrências de cada elemento. Perceba que quando aumentamos o limite de tais valores, consequentemente, estaremos, também, expandindo o tamanho das listas auxiliares. No entanto, esse crescimento é suficientemente grande para superar o tempo médio gasto pelos outros algoritmos, sobretudo, os quadráticos.

Figura 10: Gráfico que analisa o desempenho do *Counting Sort* em relação a variação do intervalo de elementos da lista



Com auxílio da notação *Big O*, esse fato torna-se mais claro, pois, para tais implementações de algoritmos e para os intervalos analisados, temos que, empiricamente, se $r > n^2 - n$, então $O(n + r) > O(n^2)$, sendo n o tamanho da lista e r o valor máximo dessa mesma lista. Veja que, de fato, isso ocorre nos gráficos.

Por fim, a fig.10 mostra que mesmo diante da queda de eficiência, a complexidade do *Counting Sort* se mantém linear, o que fortalece, também, os resultados dos experimentos anteriores.

4.7 Como as versões implementadas em C se comparam com o *Counting Sort* implementado em Python e o *Timsort*?

Agora, partimos para uma abordagem que intersecta duas linguagens bem distintas. Nosso objetivo é explorar a rapidez do *C* e as potencialidades do *Python*, como a disponibilidade de bibliotecas gráficas. Para isso, executamos de modo similar o primeiro experimento, visto na seção 4.1. Entretanto, esperamos novos resultados, em especial, dos algoritmos quadráticos, que estão implementados em C. A seguir, encontrará os resultados.

Perceba, de imediato, como houve uma drástica redução dos tempos médios de execução do *Selection Sort*, *Bubble Sort* e do *Insertion Sort*. Se relembrarmos as informações obtidas da Fig.1, veremos que, em média, os algoritmos quadráticos se tornaram 20x mais rápidos. É um dado empírico muito valioso.

Uma explicação intuitiva para esse fato se baseia em como o código-fonte é executado em cada uma das linguagens testadas. Enquanto o *Python* é dito uma "linguagem interpretada", o *C* é dito "compilado". Isso significa que

durante a execução do código *python*, o interpretador traduz linha a linha cada comando, já no *C*, o código é diretamente convertido em linguagem de máquina e executado. Portanto, em experimentos que dependem do tempo de execução de um programa, obtêm-se uma eficácia maior em linguagens compiladas.

Figura 11: Gráfico que analisa o desempenho das implementações em C com os algortimos *Counting Sort* e *Timsort* no *Python*

No entanto, mesmo com esse grande avanço, a melhoria de performance dos algoritmos citaos não foi capaz de superar o *Counting Sort* e o *Timsort*, que, novamente, se mostraram extremamente poderosos no intervalo analisado.

5 Conclusões

Essa pesquisa buscou compreender empiricamente a complexidade de tempo dos algoritmos *Selection Sort, Bubble Sort, Insertion Sort, Counting Sort* e *Timsort* mediante diversas condições.

No primeiro experimento, notamos que havia uma diferença marcante entre dois grupos: aqueles em que o tempo médio de execução possuía um rápido crescimento conforme crescia o tamanho da lista e que, portanto, apresentavam uma curva quadrática em seus gráficos, e outros algoritmos cujos tempos médios de execução pareciam não sofrer alterações no intervalo analisado, o que motivou a realização de uma nova experiência. O primeiro grupo era composto pelo *Selection Sort, Bubble Sort e Insertion Sort*, enquanto o segundo grupo unia o *Counting Sort* e o *Timsort*.

Na segunda parte da pesquisa comparamos a performance de três algoritmos em relação a proporção de desordem dos elementos na lista. Com efeito, percebemos que o *Bubble Sort* e o *Insertion Sort* mostravam, em seus gráficos, uma proximidade com a curva logarítmica, o que significava uma tendência de estabilização dos tempos médios de execução à medida que a porcentagem de

desordem crescia. Enquanto isso, o *Timsort* não era afetado por tal condição, mantendo uma média constante.

Posteriormente, decidimos testar duas implementações diferentes do *Insertion Sort*, uma versão de nossa autoria e uma outra otimizada. Como consequência, reparamos que, apesar de ambas mostrarem uma complexidade de tempo quadrática e uma tendência estabilizadora com relação a desordem dos elementos, a eficiência da versão otimizada tornava-se evidente com o avanço dos tamanhos das listas.

Ademais, ainda investigamos com mais detalhes a natureza dos algoritmos *Counting Sort* e *Timsort*, pois, no primeiro experimento, os resultados dos testes demonstravam uma proximidade muito grande quanto ao tempo de execução. Assim, repetimos o experimento, mas, dessa vez, com foco nesses dois algoritmos. Isso revelou que, no intervalo explorado, a complexidade do *Counting Sort* e do *Timsort* era particularmente linear.

Depois disso, outras perguntas sobre tais algoritmos surgiram. Primeiramente, discutimos como o *Timsort* lida com conjuntos de listas de tamanhos da ordem de 10^7 posições. Com isso, concluímos que ele mantinha uma tendência linear no intervalo experimentado, mas isso não implicava que, necessariamente, sua complexidade de tempo era linear.

Verificamos, também, em outro experimento se a grandeza dos elementos presentes na lista influenciaria a performance do *Counting Sort*. Desse modo, notamos que, de fato, a rapidez e a eficiência desse algoritmo são impactados severamente, já que em listas pequenas, porém com elementos da ordem de grandeza de 10^7, o tempo médio de execução desse algoritmo dispara em comparação com os outros estudados, que se mantêm constantes.

Por fim, em nosso último experimento, mesclamos as linguagens C e *Python*, utilizando o recurso das bibliotecas compartilhadas. Assim, buscamos entender se a implementação dos algoritmos quadráticos em uma linguagem naturalmente mais rápida, como o C, seria suficiente para superar o desempenho do *Counting Sort* e do *Timsort*. Como resultado, obtivemos uma enorme melhoria no tempo médio de execução, todavia isso não foi o bastante para concretizar a hipótese descrita.

Portanto, em posse dos dados obtidos, é válido destacar a rapidez dos algoritmos por distribuição, sobretudo para listas com tamanhos maiores. Entretanto, há parâmetros que devem ser analisados. Em particular, o *Counting Sort* possui restrições quanto ao intervalo de valores contidos na lista, visto que, nessa implementação, somente suporta números inteiros positivos e com uma ordem de grandeza pequena. Logo, em situações onde se conhece previamente os dados a serem ordenados, trata-se de uma ótima escolha.

Por outro lado, vimos que os algoritmos quadráticos não lidam bem com listas de tamanhos crescentes, mas não exigem espaços extras de memória, o que em muitos casos pode ser o ideal. Com relação aos três algoritmos, *Selection Sort, Bubble Sort e Insertion Sort*, devemos observar que o último mostrou uma vantagem empírica sobre os outros dois.

Finalmente, constata-se que existem inúmeros fatores que atuam sob o desempenho dos algoritmos de ordenação estudados, sobretudo, em relação ao

tempo gasto de execução. Dentre eles, o tamanho da lista, a desordem de seus elementos, bem como a grandeza de seus valores. Além disso, critérios como rapidez de processamento, quantidade de memória disponível e a importância de se preservar a ordem relativa dos itens são aspectos cruciais na seleção de um algoritmo. Por isso, uma escolha às cegas e sem análise prévia dos dados que serão ordenados pode levar a longas esperas até sua finalização.

6 Avaliação Crítica

Esse foi o primeiro EP *(de muitos)* do curso. Posso dizer que minha experiência foi bastante positiva, gostei muito da proposta que instiga o processo científico de experimentos e de interpretação dos resultados, apesar de alguns contratempos durante sua produção. Ademais, o desenvolvimento desse EP se deu em alguns passos, irei comentar um pouco sobre eles abaixo.

Primeiro, a implementação dos algoritmos. Acredito ter passado umas duas noites para garantir o seu funcionamento correto. Depois disso, tivemos o desenvolvimentos das funções pedidas até a primeira execução dos testes. Entretanto, os resultados obtidos não foram os esperados. Com isso, o terceiro passo foi a revisão do código em busca de erros. Nesse intervalo de tempo, decidi fazer alguns experimentos extras, pois surgiram algumas curiosidades. Ao final dessa etapa, partimos para a escrita do relatório e para a pesquisa de informações. Esse quarto e último passo foi bastante trabalhoso, mais do que a atividade em si, pois me deparei com muitas dificuldades na personalização do documento em LaTeX, em especial, envolvendo os *packages: minted, listings, cite e natbib* no ambiente vs code.

Em relação ao complemento da pesquisa, a proposta foi bastante interessante. O ato de unir duas linguagens opostas em questão de expressividade, mas que possuem ferramentas muito poderosas para determinadas situações foi algo prazeroso de se realizar. Como dificuldades, posso dizer que a maior delas foi a criação da biblioteca compartilhada, visto que utilizo Windows, o que gerou alguns problemas. Porém, com algumas horas debruçado sobre esses erros, finalmente consegui resolver.

Nesse momento, já finalizado o trabalho, posso dizer que valeu a pena o esforço, afinal esse incentivo a experimentação e a descoberta são ferramentas importantes para a nossa trajetória no curso.

7 Referências Bibliográficas

[1] Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th Edition, Addison-Wesley Professional, March 24, 2011. Disponível em `https://algs4.cs.princeton.edu/home/`.

[2] Robert Sedgewick and Philippe Flajolet, *An Introduction to the Analysis of Algorithms*, 2th Edition, Addison-Wesley Professional, 1996. Disponível em `https://aofa.cs.princeton.edu/home/`.

[3] Robert Sedgewick and Kevin Wayne, *Computer Science: An Interdisciplinary Approach*, 1th Edition, Addison-Wesley Professional, 2016. Disponível em `https://introcs.cs.princeton.edu/java/home/`.

[4] 1.124 Lecture 10, *Sorting*, 2000. [Online; accessed 12-June-2024].

[5] Wikipedia contributors, *Big O notation — Wikipedia, The Free Encyclopedia*, 2024. [Online; accessed 12-June-2024].

[6] _____, *Time complexity — Wikipedia, The Free Encyclopedia*, 2024. [Online; accessed 13-June-2024].

[7] _____, *Sorting algorithm — Wikipedia, The Free Encyclopedia*, 2024. [Online; accessed 13-June-2024].

[8] Geek for Geeks contributors, *TimSort – Data Structures and Algorithms Tutorials*, 2023. [Online; accessed 13-June-2024].

[9] freeCodeCamp, *Interpreted vs Compiled Programming Languages: What's the Difference?*, 2020. [Online; acessed 29-June-2024].