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(a): João Viktor Souza Almeida

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 ordernar uma lista. Além disso, é verificado como suas eficiências mudam dadas diferentes condições iniciais, tais como porcentagem de ordenação e número de elementos da lista a ser ordenada.

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. 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, durante a execução dos algoritmos, foram desabilitados o máximo de programas possíveis a fim de minimizar as interferências nos resutlados dos testes.

Para computar a média de execução, foi invocada uma função *timeMe*, na qual eram armazenados os resultados das 10 itinerações do mesmo algoritmo e, no final, retornava a média e o desvio padrão dos resultados.

Para fins simplificatórios, foram ignorados possíveis margens de erros da função que retorna a lista embaralhada. Contudo, essa margem de erro torna-se inotável durante os testes devido ao tamanho das listas utilizadas.

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 devios-padrões de cada algoritmo com listas de tamanhos 1000, 5000, 10000, 50000 e 100000.

O segundo, por outro lado, buscou elucidar acerca das mudanças causadas pela taxa de ordenação de uma lista. Para isso, foi estabelecida uma segunda malha de repetição, na qual também foi computada as médias e os devios-padrçoes 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

O algoritmo de Seleção é um algoritmo de ordenação no qual, a cada itineração, o menor elemento da lista é garantido estar na posição correta. Ou seja, na primeira itineração, assegura-se que o menor elemento ficará na primeira posição da lista, na segunda itineração, o segundo elemento, e assim por diante.

Acerca do seu desempenho, o algoritmo em questão possui um "desempenho"quadrático, isto é, para n elementops da lista, o algoritmo realizará $\frac{n(n+1)}{2}$ comparações a fim de ordernar totalmente a lista. 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 elento 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]
```

1.3 Quantidade de comparações

Nesta subseção, irá ser debatida a quantidade de comparações feitas pelo algoritmo em questão. Como a quantidade de comparações é "constante"neste algoritmo, será "fácil"de analisá-la; nos outros algoritmos, contudo, a análise é mais compléxa 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 vezes, em que a primeira $i = 1$, na segunda, $i = 2$ e assim por diante. Por consequência, em cada uma dessas execuções, a segunda malha será executada $n-i$ vezes (isto é, na primeira execução, o código contido na segunda malha de repetição será executado n vezes, na segunda, $n-1$, nas seguntes a lógica é análoga.). Dessa forma, o código será executado $n + (n-1) + (n-2) + ... + (1)$, ou seja, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

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

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.

2.1 Exemplo

Seja M uma lista [4,3,2,1,0]. Na primeira passagem, o maior valor da lista (4) estará em seu lugar correto, ou seja, M será [3,2,1,0,4]. Na segunda passagem, o segundo maior elemento também estará posicionado em sua posição correta; assim, M será [2,1,0,3,4]. O algoritmo continuará realizando "isto"até que, na 5^{o} 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]
```

Perceptivelmente, o maior elemento é, a cada modificação, levado uma posição para a direita, até que o elemento esteja na posição correta. De fato, o nome do algoritmo faz referência às bolhas em bebidas, uma vez que as bolhas maiores sobem mais rapidamente do que as menores.

2.2 Implementação

Segue, a seguir, o código utilizado na implementação do algoritmo em questão em 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
```

Perceptivelmente, caso a lista já esteja ordenada, na primeira execução da malha de repetição, o `isIncreasing` continuará com o valor `True` e, portanto, 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 "testador".

2.3 Quantidade de comparações

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

Caso a lista com n elementos já esteja 100% ordenada, o algoritmo detectará "isto"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.

Finalmente, nos outros casos, a análise é mais complexa e depende de outros fatores que não estão no escopo da análise. 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, pussui como principal característica, a asseguração de que, a cada itineração 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 uma lista $[4, 3, 2, 1, 0]$.

O algoritmo em questão funcionam 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 que zero e menor do 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 $j+1$(ver isto depois). 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 pode-se 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 itineração 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.

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

4 Algoritmo de contagem

Diferentemente dos algoritmos outrora apresentados, o algoritmo em tópico possui uma complexidade linear, isto é, o tempo levado para ordernar uma lista, ao contrário dos apresentados, não cresce de maneira quadrática.

Em relação ao uso de memória, contudo, há uma grande desvantagem no algoritmo, uma vez que a lista auxiliar utilizada 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 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 de Aux, por outro lado, terá o mesmo valor 3, uma vez que há três elemento 2 na lista original. Essa lógica seguirá até chegar no último elemento. Assim, de início, a lista Aux será [1,1,3,1,2].

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

Segue, a seguir, os valores de `Aux` a cada modificação e, logo abaixo, os valores de M:

```
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]
```

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

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

Notoriamente, a lista, durante as mudanças intermediárias, não possui todos os valores iniciais, isto é, alguns dos elementos (como o 4) deixam de existir em alguma etapa e voltam a aparecer somente depois.

4.2 Implementação

Segue a implementação em Python

```python
def counting(V, n):
    max_element = max(V)
    hist_list = [0 for _ in range(max_element + 1)]
    for i in range(max_element + 1):
        hist_list[i] = count_element_in_array(i, V)
    index = 0
    for i in range(max_element + 1):
        for _ in range(hist_list[i]):
            V[index] = i
            index += 1
```

Como pode-se perceber, o algoritmo de contagem, assim como os outros, possui duas malhas de repetição. Contudo, em vez da segunda estar dentro da terceira, uma ocorre após a outra. Apesar disso, uma singularidade desse algoritmo é que, ao contrário dos outros, a quantidade de itinerações na primeira malha de repetição depende do tamanho do maior elemento (pensando apenas na implementação com números positivos), tornando-o eficaz para listas grandes mas com elementos menores. Adicionalmente, este algoritmo não ordena os elementos por comparação...

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 obtivos 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 elas 1% , 3%, 5%, 10% e, por fim, 50%.

Logo abaixo, têm-se os resultados em gráficos.

5.1 Variação na quantidade de elementos

Como outrora comentado, os resultados comprovam o comportamento quadrático do bolha, seleção e inserção.

Contudo, é necessário destacar que, caso houvesse uma maior variância nos números da lista a ser ordenada, isto é, caso a diferença entre o maior e o menor valor fosse um valor muito grande, o algoritmo de contagem apresentaria um resultado ruim, podendo ultrapassar o bolha, por exemplo.

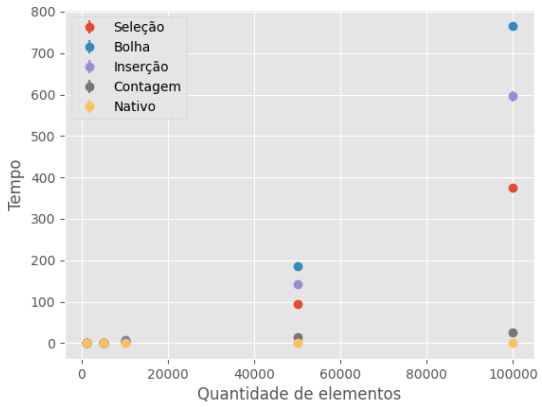

Figura 1: Gráfico elucidando o comportamento de cada algoritmo em tópico. No eixo x, há a quantidade de elementos; no y, o tempo médio gasto, em segundos, por cada algoritmo

5.2 Ordenação prévia

Como previamente comentado, o algoritmo de bolha e o de inserção variam de acordo com a taxa de ordenação. Assim, em caso de ordenação inicial total,

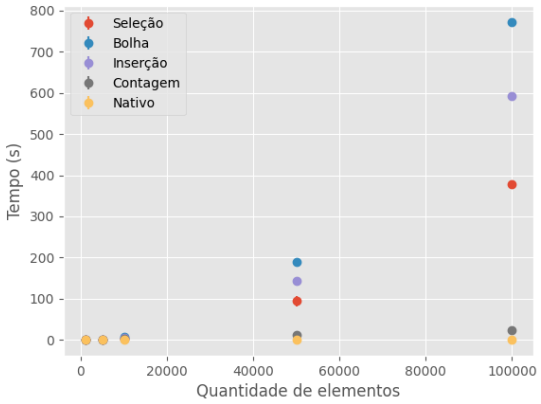

Figura 2: Gráfico ilustrando o tempo médio 'gasto' dos algoritmos bolha e inserção em relação à taxa de ordenação inicial dos vetores.

ambos os algoritmos adquirem um comportamento linear, uma vez que serão feitas n comparações até que o algoritmo detecte que a lista já se encontra ordenada e, consequentemente, parará.

6 Conclusão

Desta maneira, apesar de possuírem diferentes desempenhos dependendo da ordenação da lista, os algoritmos em questão, possuem um baixo proveito com listas muito grande. Portanto, a fim de ordenar uma lista com maior rapidez, é notável que a função built-in do Python deve, pelo menos para listas grandes, ser usada em relação aos outros métodos outrora discutidos.

Adicionalmente, o algoritmo de contagem, ainda que possua um comportamento linear em relação ao tamanho das listas, pode sofrer variações em razão da diferença entre o maior e o menor elemento da lista, isto é, ainda que a lista seja relativamente pequena, o tempo de realização do algoritmo pode ser muito alto caso dependendo da variação entre os números. Assim, é aconselhável a utilização do algoritmo em pauta apenas com listas que possuem uma exígua variação.

Em uma outra perspectiva, o bolha foi o algoritmo que apresentou o menor rendimento submetido a listas com diferentes tamanhos e taxas de ordenação. Dessa forma, o bolha é recomendado apenas para fins de demonstrações.

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). Acesso em: 08 de jun. de 2024.