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 analisar "alguns"algoritmos de ordenação e 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. 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

Além disso, 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.

0.1 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, por outro lado, 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.

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]
```

Com o código, 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 da lista.

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.

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 5º passagem, (pois a lista possui 5 elementos), a lista estará totalmente ordenada.

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

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

2.1 Implementação

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

```
def bubble(V, n):
    lim = n - 1
    while lim >= 0:
        isIncreasing = True
        for j in range(lim):
            if V[j] > V[j + 1]:
                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".

3 Algoritmo de inserção

O algoritmo de inserção, assim como o de bolha, varia com a porcentagem de ordenação da lista recebida, ou seja, caso a lista já esteja ordenada, não haverá nada a ser feito.

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 uma lista $[4, 3, 2, 1, 0]$, caso esta fosse impressa a cada execução do algoritmo, os resultados seriam: Destarte, caso fosse impresso a lista a cada modificação, os resultados seriam os seguintes:

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

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

```
def insertion(V, n):
last_index = 0
for i in range(last_index,n - 1):
    if V[i] > V[i + 1]:
        j = i
        while V[j] > V[j + 1]:
            V[j + 1], V[j] = V[j], V[j + 1]
            if j <= 0:
                break
            j -= 1
```

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 terá um tamanho de 4000 elementos.

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

Segue, a seguir, os valores de `V` a cada modificação:

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

5 Comparações

Como supracitado, o algoritmo de bolha e o de inserção variam de acordo com a taxa de ordenação. Assim, em caso de ordenação total, ambos os algoritmos adquirem um comportamento linear.

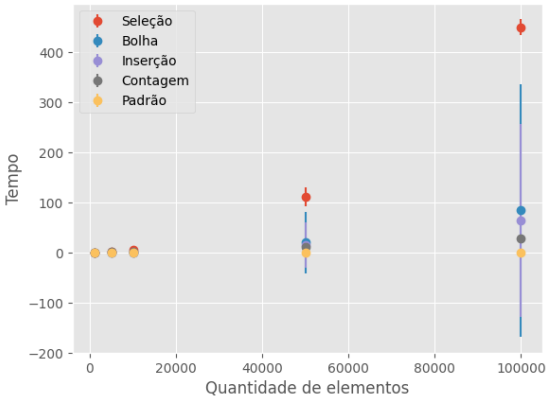

Figura 1: Gráfico elucidando o comportamento de cada algoritmo em tópico

Conclusão

Os algoritmos em questão, apesar de possuírem diferentes desempenhos dependendo da ordenação da lista, 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.