Universidade de Brasília
Instituto de Ciências Exatas

Departamento de Ciência da Computação

# 117536 - Projeto e Análise de Algoritmos
# Turma: B

# Formalização do algoritmo de ordenação Bubble Sort

Manuela Matos Correia de Souza - 160135281
Gabriel Teixeira Ribeiro - 15/0126891

25 de novembro de 2019

## 1 Introdução

Nesse trabalho, foram formalizadas propriedades do algoritmo de ordenação Bubble Sort, conforme a especificação fornecida pelo professor. O resultado principal do trabalho é que a complexidade temporal de Bubble Sort é $\Theta(n^2)$, onde $n$ é o tamanho do vetor de entrada. Ao longo do texto foram expostas as etapas necessárias para realizar essa análise, desde a formulação de um palpite para o tempo de execução até a prova concreta de sua complexidade. Todas os lemas e teoremas usados nessa teoria foram provados usando o assistente de provas PVS. Por esse motivo, a explicação das provas foi centrada no raciocínio que as conduziu, e não no passo a passo realizado no assistente.

# 2 Apresentação do Problema

O presente trabalho consiste na formalização de propriedades do algortimo Bubble Sort, mais especificamente da sua complexidade temporal e sua corretude. O estudo da complexidade temporal de um algoritmo fornece uma estimativa do tempo de execução desse algoritmo em função do tamanho de entrada. Nesse caso, o que se deseja é entender o comportamento assintótico da função que modela o tempo de execução do algortimo, e portanto, não é preciso muito rigor na hora de criar essa função. Assim, para modelar o tempo de execução, considera-se que cada operação básica do algoritmo é realizada em tempo constante e, em seguida, conta-se quantas operações são realizadas em uma execução do algortimo para um dada entrada. É comum que apenas as operações realizadas repetidamente sejam levadas em conta, porque do contrário estaríamos somando constantes que são dominados pelas operações realizadas em loop.

## 2.1 Análise assintótica

### 2.1.1 Elaborando uma estimativa do tempo de execução

Para fazermos uma análise assintótica de Bubble Sort, é necessário elaborar um palpite para o tempo de execução do algortimo. Bubble sort, segundo a especificação fornecida, toma uma lista $l$ e aplica a ela a função *bubblesort_aux*, passando como segundo parâmetro o comprimento da lista menos um. Até aí, temos tempo constante, pois a chamada da função é uma operação básica.

A função *bubblesort_aux*, por sua vez, é uma função recursiva que recebe como parâmetros uma lista $l$ e um natural $n$. Ela executa uma chamada recursiva de si mesma passando como primeiro parâmetro uma nova lista e como segundo parâmetro *n - 1*. Assim, a função *bubblesort_aux* é executada $n$ vezes, porque em cada chamada o parâmetro numérico diminui em uma unidade, e quando ele chega a zero a função retorna a lista passada na última chamada. Como na primeira chamada *bubblesort_aux* começa em *length(l) - 1*, são $length(l)$ execuções. Agora, como a única operação realizada por essa função é a chamada dela mesma com diferentes parâmetros, seu tempo de execução é $length(l)$ multiplicado pelo tempo de passar uma lista e decrementar o natural. O que acontece, no entanto, é que essa nova lista usada na próxima chamada da função é o retorno de uma terceira função, denominada

*bubbling*, e portanto, temos que somar ao tempo total o tempo de calcular *bubbling* em cada chamada recursiva de *bubblesort_aux*.

A função *bubbling* recebe como parâmetros uma lista $l$ e um natural $n$. Assim como *bubblesort_aux*, ela é executada recursivamente diminuindo $n$ a cada chamada até que $n$ chegue a zero e a função retorne a lista da última chamada. Porém, diferentemente da primeira função, *bubbling* realiza outras operações, como comparações e construção de listas. Mas como todas essas operações têm custo constante, o que nos importa é saber quantas vezes *bubbling* é executada.

Sempre que *bubbling* é chamada, ela é executada recursivamente $n$ vezes, mas esse $n$ é um parâmetro passado por *bubblesort_aux*, e sabe-se que ele diminui a cada chamada. Ou seja, a cada chamada de $bubblesort_aux$, a função *bubbling* é executada menos vezes.

Inicialmente, *bubblesort_aux* recebe como parâmetros $l$ e *length(l) - 1*. Daí, ela chama a si mesma com os novos parâmetros *bubbling(l, length(l) - 1)* e *length(l) - 2*. Sabemos que *bubbling(l, length(l) - 1)* toma tempo *length(l)*. Em seguida, *bubblesort_aux* recebe como parâmetros a lista *bubbling(bubbling(l,length(l) - 1), length(l) - 2)* e *length(l) - 3*. Nesse caso, a função *bubbling* toma tempo *length(l) - 1*. Se continuarmos detalhando cada chamada de *bubblesort_aux* e somarmos os tempos de cada chamada, temos que o tempo total de execução é dado por:

$$\begin{aligned}\sum_{k=1}^{length(l)-1} k &= 1 + ... + (length(l) - 2) + (length(l) - 1) \\ &= \frac{(length(l) - 1)((length(l) - 1) + 1)}{2} \\ &= \frac{length(l)^2 - length(l)}{2}\end{aligned}$$

onde $k$ representa o tempo de execução de *bubbling* em cada uma das chamadas de *bubblesort_aux*.

Pela estimativa que obtivemos, o tempo de execução do algortimo Bubble Sort é $O(n^2)$. No entanto, para provarmos isso, precisamos primeiro mostrar que a estimativa obtida está correta. Isso será feito na próxima sessão.

### 2.1.2 Mostrando a correção da estimativa obtida

Para mostrar que o tempo de execução de *Bubblesort(l)* é $\frac{length(l)^2 - length(l)}{2}$, precisamos primeiro criar uma função que nos dá o tempo de execução do algoritmo para uma determinada entrada. Essa função vai ser uma adaptação da original *Bubblesort*, e para fornecer o tempo de execução, conta quantas comparações foram feitas pelo agoritmo. Também será necessário adaptar as funções *bubblesort_aux* e *bubbling*, pois elas interferem no número de comparações feitas.

Inicialmente, as três novas funções, chamadas *cbubbling*, *cbubblesort_aux* e *cbubblesort*, são modificadas para retornarem um par de elementos: a lista que está sendo ordenada e um contador de comparações. Além disso, *cbubblesort_aux* passa receber um contador como parâmetro, além da lista a ser ordenada. Isso é necessário porque a quantidade de comparações se acumula a cada chamada recursiva, e portanto ela precisa repassar para a próxima chamada o valor atual do contador. A função *cbubbling*, assim como a anterior e pelo mesmo motivo, precisa receber como parâmetro um contador. Mas além disso, essa é a função que faz as comparações de fato, e portanto, toda vez que ela é chamada, é preciso incrementar o contador antes de passar para a próxima chamada.

Queremos mostrar então que o contador retornado pela função *cbubblesort*, aplicada a uma lista $l$, é exatamente $\frac{length(l)^2 - length(l)}{2}$. Para provar algo sobre as comprações feitas por *cbubblesort*, precisamos antes de resultados similares para *cbubblesort_aux* e para *cbubbling*, já que a primeira usa as seguintes.

As funções recursivas dessa especificação têm uma característica importante: sempre que a função chama a si mesma, ela passa como parâmetro uma lista diferente da anterior. Por causa dessa característica, é preciso certo cuidado ao enunciar lemas que necessitem de provas indutivas. Se o lema for enunciado para toda lista $l$ e todo $n$ menor que o comprimento da lista, não conseguiremos usar a hipótese de indução. Isso acontece porque quando fazemos indução sobre $n$, a função na hipótese de indução tem como parâmetro a mesma lista da primeira chamada, mesmo que agora seu parâmetro numérico seja $n-1$. Como a lista muda a cada chamada recursiva, a hipótese de indução não pode ser utilizada. Por isso, é preciso enunciar os lemas sem atrelar $n$ a uma lista, de modo que possamos usar a hipótese para uma lista qualquer.

Dito isso, começaremos as provas por *cbubbling*, que é a função mais interna.

Quantas comprações faz *cbubbling*? Intuitivamente, ela é executada $n$ vezes, onde $n$ é o parâmetro da recursão, e a cada execução faz uma compração. Portanto, a cada execução soma-se um ao contador inicial e isso nos dá que $cbubbling(l, c, n)$ faz $n+c$ comparações. Essa afirmação está expressa no lema *comp_cbubbling*; vamos usar indução fraca sobre o parâmetro da recursão para tentar prová-lo. A base de indução, isto é, $n = 0$, é trivial, porque nesse caso a função retorna a própria lista e o contador inicial. Agora vamos tentar mostrar que a propriedade vale para $n + 1$. Como a função *cbubbling* é definida usando a cabeça e calda das listas, seria bom poder considerar apenas as listas de comprimento maior que um nessa etapa, ou seja, com cabeça e cauda não vazias. De fato, podemos fazer isso, porque nos casos em que a lista é nula ou unitária, temos que $n = 0$ ($n < length(l)$), e já provamos esse caso. Portanto, queremos mostrar que a propriedade vale para $n+1$ no caso em que a lista tem pelo menos dois elementos. Para usarmos a hipótese de indução, temos que expandir a definição de *cbubbling*. Isso nos leva a duas possibilidades: ou $car(l) < car(cdr(l))$ ou o contrário. No primeiro caso, a função retorna o contador obtido de $cbubbling(cons(car(l), cdr(cdr(l)), c, n))$ somado a um. No segundo caso, retorna o contador de $cbubbling(cdr(l), c, n))$ também somado a um. Em ambos os casos, *cbubbling* está sendo aplicada a uma lista de comprimento menor do que o comprimento de $l$ [1] e com o contador $n$. Daí, podemos usar a hipótese de indução e dizer que esses contadores obtidos de $cbubbling(cons(car(l), cdr(cdr(l)), c, n))$ e $cbubbling(cdr(l), c, n))$ são $c + n$. Isso nos dá que o contador obtido de $cbubbling(l, c, n)$, independente do caso, é igual a $(c + n) + 1 = c + (n + 1)$, como queríamos.

Agora, queremos saber quantas comparações faz *cbubblesort_aux*, para que possamos finalmente mostrar quantas comparações *cbubblesort* faz. Quantas comparações faz $cbubblesort\_aux(l, c, n)$? Essa função é executada $n$ vezes, e pra cada vez as únicas comparações que faz vêm indiretamente pela função *cbubbling*. Na primeira chamada, $cbubblesort\_aux(l, c, n)$ retorna *cbubblesort_aux* aplicado à lista resultado de $cbubbling(l, c, n)$ com o contador resultante de $cbubbling(l, c, n)$ e $n - 1$. Portanto, ela vai acumulando a cada chamada os contadores de *cbubbling*. Isso nos leva a seguinte fórmula para a quantidade de comparações:

[1] Esse fato, embora trivial, está enunciado no lema *length_cdr*

$$c + \sum_{k=1}^{n} k = c + 1 + ... + (n-1) + n = c + \frac{n(n+1)}{2}$$

Vamos então provar um lema da seguinte forma:

**Lema:** Para toda lista $l$ e $n$ natural,

$$n < length(l) \text{ implica } cbubblesort\_aux(l, c, n)\text{`}2 = c + \frac{n(n+1)}{2}$$

Como *cbubblesort_aux* é um algortimo recursivo, vamos usar novamente indução sobre o parâmetro da recursão $n$ para provar essa propriedade. O caso base, com $n = 0$, é trivial, pois a função retorna a própria lista e o contador $c$, e portanto a fórmula serve. Para provarmos a propriedade para $n+1$, vamos expandir a definição de *cbubblesort_aux* para cairmos na hipótese de indução. Agora temos *cbubblesort_aux* aplicada em $cbubbling(l, c, n)$ e com parâmetro $n$. Para usar a hipótese de indução, precisamos mostrar que $n$ é menor que o comprimento da lista retornada por $cbubbling(l, c, n)$. Isso é provado no lema *cbubbling_preserves_length*, o qual não é difícil de acreditar, pois *cbubbling* só altera a ordem dos elementos da lista. Tendo esse lema, podemos usar a hipótese de indução e dizer que

$$cbubblesort\_aux(l_1, c_1, n) = c_1 + \frac{n(n+1)}{2}$$

onde $l_1$ é $cbubbling(l, c, n)$`1 e $c_1$ é $cbubbling(l, c, n)$`2. Pelo lema provado anteriormente, $c_1 = c + n$, e então temos que

$$\begin{aligned} cbubblesort\_aux(l_1, c_1, n) &= c + n + \frac{n(n+1)}{2} \\ &= c + \frac{2n + n^2 + n}{2} \\ &= c + \frac{(n+1)(n+2)}{2} \end{aligned}$$

E assim, provamos o que queríamos.

Finalmente, podemos agora provar que o número de comparações feitas por

$cbubblesort(l)$ é $\frac{length(l)*(length(l)-1)}{2}$.

Se $l$ é um lista nula, $cbubblesort(l)$ retorna o contador 0, o que não contradiz a formula dada. Se $l$ não é uma lista nula, então a função chama $cbubblesort\_aux(l, 0, length(l) - 1)$. Agora, pelo lema anterior, sabemos exatamente o número de comparações feitas nesse caso:

$$\frac{(length(l) - 1)(length(l))}{2}$$

Que é o que queríamos mostrar.

### 2.1.3 Provando o comportamento assintótico do algoritmo

Agora que sabemos o número de comparações feitas por $cbubblesort(l)$, isto é, o seu tempo de execução, podemos mostrar que o comportamento do algoritmo é $\Theta(n^2)$, onde $n = length(l)$. Para mostrar isso, precisamos encontrar constantes $c_1, c_2 > 0$ tais que existe uma constante $n_0 > 0$ a partir da qual

$$c_1 n^2 \leq \frac{n(n-1)}{2} \leq c_2 n^2 \tag{1}$$

Se fizermos $c_2 = \frac{1}{2}$, o lado direto da desigualdade vale, pois no termo do meio estamos subtraindo de $\frac{n^2}{2}$ uma parcela postiva. Já do lado esquerdo da desigualdade, se fizermos $c_1 = \frac{1}{4}$ temos:

$$\frac{n^2}{4} \leq \frac{n^2 - n}{2} \quad \Rightarrow \quad \frac{n}{2} \leq n - 1 \quad \Rightarrow \quad \frac{n}{2} - n + 1 \leq 0 \quad \Rightarrow$$

$$\frac{-n + 2}{2} \leq 0 \quad \Rightarrow \quad n \geq 2$$

Portanto, basta pedirmos $n \geq 2$ para que $c_1$ sirva na desigualdade.

Assim, obtivemos constantes $c_1, c_2 > 0$ tais que a partir de $n_0 = 2$, a equação (1) sempre acontece. Portanto, *bubblesort* é $\Theta(n^2)$.

### 2.1.4 Mostrando a equivalência das funções com contador

Na formalização da análise assintótica de Bubble Sort, assumimos que as funções *cbubbling*, *cbubblesort_aux* e *cbubblesort* são equivalentes às funções

*bubbling*, *bubblesort_aux* e *bubblesort* no sentido de que retornam as mesmas listas, respectivamente. Para finalizarmos a prova construída até aqui, precisamos mostrar essa equivalência.

Começamos pela função *cbubbling*, uma vez que as outras dependem dessa. Como ela é uma função recursiva, vamos usar indução sobre o parâmetro da recursão $n$. Para $n = 0$, $cbubbling(l, c, 0)$ retorna a lista $l$, assim como $bubbling(l, 0)$. Suponhamos então que essa propriedade vale para todo $n < n + 1$. O que acontece com $l$ em $cbubbling(l, c, n + 1)$? Temos 2 casos: se $car(l) < car(cdr(l))$ ou o contrário. No primeiro caso, a função retorna a lista $cons(car(cdr(l)), cbubbling(cons(car(l), cdr(cdr(l))), c, n))$ e no segundo caso retorna $cons(car(l), cbubbling(cdr(l), c, n))$. Em ambos os casos, por hipótese de indução, as caudas das listas retornadas são iguais às retornadas por *bubbling*. Portanto, *cbubbling* retorna $cons(car(cdr(l))$, $bubbling(cons(car(l), cdr(cdr(l))), n))$ ou $cons(car(l), bubbling(cdr(l), n))$, que são exatamente as listas retornadas por *bubbling*.

Usamos a mesma estratégia para provar que as listas obtidas como saída de *cbubblesort_aux* e *bubblesort_aux* são as mesmas. No caso em que $n = 0$, $cbubblesort\_aux(l, c, 0)$ retorna $l$, assim como $bubblesort\_aux(l, 0)$. Agora, $cbubblesort\_aux(l, c, n + 1) = cbubblesort\_aux(cbubbling(l, c, n + 1)`1, c_2, n)$, onde $c_2$ é o contador retornado de $cbubbling(l, c, n + 1)$. Por hipótese de indução, a lista retornada de $cbubblesort\_aux(cbubbling(l, c, n + 1)`1, c_2, n)$ é a mesma que de $bubblesort\_aux(cbubbling(l, c, n + 1), n)$[2]. Daí, usamos o lema de equivalência entre *cbubbling* e *bubbling* e obtemos a expressão que queríamos.

Por último, mostraremos a equivalência entre *cbubblesort* e *bubblesort*. Primeiro analisamos o caso em que a lista é nula, por conta das definições das funções. Se $l$ é nula, as duas funções retornam a própria lista $l$. Se $l$ não é nula, então a função com contador retorna *cbubblesort_aux(l, 0, length(l)-1)* e a sem contador retorna *bubblesort_aux(l, length(l)-1)*. Como as funções *cbubblesort_aux* e *bubblesort_aux* são equivalentes, segue que as listas retornadas são iguais e portanto *cbubblesort* e *bubblesort* são equivalentes.

[2]Precisamos usar a hipótese de indução para uma lista específica, nesse caso poderíamos escolher tanto $cbubbling(l, c, n + 1)$ quanto $bubbling(l, n + 1)$, uma vez que usaremos que são equivalentes

# 3 Conclusão

É fácil perceber, após a finalização do trabalho, que formalizar propriedades relativamente simples de algortimos não é algo trivial quando se usa um assistente de provas. Para mostrarmos apenas a complexidade temporal de Bubble Sort foram necessários 29 resultados e muito tempo de dedicação. Apesar disso, é graças a esse hiper-detalhismo que se pode ter certeza da correção das provas fornecidas.

Bubble Sort, como foi comprovado, é um algoritmo que funciona em tempo quadrático e, portanto, é extremamente ineficiente para ordenação de vetores. Essa ineficiência se dá pela sua função de ordenação, que funciona como um borbulhamento, levando os elementos um a um para o fim da lista. A cada percorrimento do vetor, os elementos maiores "sobem", mas os elementos menores não são manipulados. Essa falta de otimização do percorrimento do vetor faz com que o tempo seja desperdiçado na ordenação, e por isso obtém-se um tempo ruim, em comparação com algortimos como Merge Sort e Heap Sort.