Universidade de Brasília
Instituto de Ciências Exatas

Departamento de Ciência da Computação

117536 - Projeto e Análise de Algoritmos
Turma: B

# Formalização da complexidade de tempo no pior caso do algoritmo Bubble Sort

Gabriel de Oliveira Estevam - 17/0142591
Lucas Seabra Gomes Oliveira - 17/0039951

5 de dezembro de 2019

# 1 Introdução

O relatório tem como objetivo descrever as etapas de formalização da complexidade de tempo no pior caso do algoritmo de ordenação Bubble Sort. Para realizar tal formalização foi necessário utilizar o software assistente de provas PVS. A complexidade temporal de Bubble Sort encontrada no processo de formalização foi $O(n^2)$, onde n significa a quantidade de elementos da lista de entrada que será ordenada pelo algoritmo.

Omicron é definido como $O(g(n))$, onde existem constantes positivas c e n0 tais que $g(n) <= c.f(n)$, para todo $n >= n0$.

# 2 Apresentação do Problema

Como já dito, o objetivo do trabalho é formalizar a complexidade temporal de Bubble Sort. A complexidade temporal de um algoritmo é o tempo gasto para o processamento de uma entrada de determinado tamanho, porém, será feita uma análise assistótica, ou seja, vamos analisar a complexidade temporal do algoritmo considerando tamanhos de entradas extremanente altos. Para calcular o tempo será considerado que cada operação do algoritmo possui tempo constante.

## 2.1 Análise assintótica

Para encontrar a complexidae de Bubble, inicialmente será feita a análise das três funções que compõem o Bubble Sort de acordo com o arquivo bubblesort.pvs, são elas:

- bubblesort
- bubblesort_aux
- bubbling

Começando pela função principal bubblesort, é passado para ela a lista L, então, é verificado se a lista informada é nula, caso seja retorna a própria lista, caso contrário chama a função bubblesort-aux informando a lista L e o tamanho (length) da lista - 1. Podemos verificar que bubblesort executa até então em tempo constante.

A função bubblesort_aux recebe a lista L e um número natural referente ao tamanho de L - 1. Nessa função é verificado se a lista L é unitária, caso seja, retorna a função bubblesort, caso contrário, chama recursivamente a função bubblesort-aux passando como parâmetro o resultado da função bubbling((l, n), n - 1), onde n é o número natural referente ao tamanho de L - 1. A função bubblesort_aux é chamada recursivamente até n ser igual a 0, ou seja, (n-1) + (n-2) + (n-3) + ..... até n = 0.

Agora será analisada a função bubbling que é feita sempre antes da chamada recursiva de bubblesort-aux. A função bubbling é a função que realiza a ordenação dos elementos, ela recebe uma lista L e um número natural n. É verificado se o n é igual a 0, caso seja retorna lista L, caso contrário verifica se o primeiro elemento de L é maior que o primeiro elemento da calda da lista, caso seja, troca esses elementos e ordena a nova calda da lita chamando bubbling(calda de L, n - 1), caso contrário, ou seja, se o primeiro elemento da lista é menor ou igual ao primeiro elemento da calda da lista, mantém o primeiro elemento como está e ordena a calda da lista chamando bubbling(calda de L, n - 1). Observe que bubbling também executa como bubblesort-aux, ou seja, existe um n que é decrementado até 0: (n-1) + (n-2) + (n-3) + ..... até n = 0. Porém, esse n da função bubbling é o n decrementado informado pela função bubblesort_aux. Precisamos verificar qual é o resultado do somatório dessas decresmentações de n:

$$\sum_{i=1}^{n} i = (n-1) + (n-2) + ... + 0 \tag{1}$$

É possível perceber que o resultado da equação acima é o somatório de uma progreção aritmética. Então,

$$\sum_{i=1}^{n} i = (((n-1)+0)n)/2 = (n^2 - n)/2 \tag{2}$$

Considerando que essa é uma análise assistótica, então, a complexidade temporal entrada até então de Bubble Sort é $O(n^2)$, já que consideramos o polinômio de maior grau do resultado encontrado $(n^2 - n)/2$.

Agora para provar que Bubble Sort é $O(n^2)$ no assistente de provas PVS precisamos de um quantificador para contar o número de comparações feitas de acordo com o tamanho da lista. Nesse caso definimos três funções, são elas:

- c_bubblesort
- c_bubblesort_aux
- c_bubbling

Essas funções retornam um contator que indica a quantidade de comparações feitas durante sua execução. Lembrando que essas funções também são equivalentes as funções: bubblesort, bubblesort_aux e bubbling, respectivamente, para provar isso foram criados os seguintes lemmas:

- c_bubbling_equiv_bubbling
- c_bubblessort_aux_equiv_bubblesort_aux
- c_bubbling_equiv_bubbling

O somador do contador foi incluido na função mais "profunda"de Bubble Sort que é a c_bubbling. Como já dito, existem duas situações que são comparados os elementos da lista para ordenação:

- Quando o primeiro elemento da lista é MAIOR que o primeiro elemento da calda da lista, então, troca os elementos e ordena a nova calda da lista.
- Quando o primeiro elemento da lista é MENOR ou IGUAL ao primeiro elemento da calda da lista, então, mantém os elementos e ordena a calda da lista.

Em ambas as situações é adicionado 1 ao contador de comparações.

Para provar a complexidade temporal de Bubble Sort foram necessários alguns lemmas, o primeiro lemma c_bubbling_preserv_length diz que a função c_bubbling não diminui o tamanho da lista e sim retorna uma lista de mesmo tamanho. Para realizar a prova desse lemma utilizamos indução forte baseada na redução da lista. Primeiro foram tratatos os casos de lista nula e unitária usando provas a parte para evitar problemas futuros na prova. Em seguida, foi utilizada a definição do c_bubbling que dividiu a prova em duas: Ambas foram resolvidas de forma semelhante, porém, tendo a hipótese de indução instanciada de formas diferentes (baseadas na lista dada como parâmetro para sua função recursiva). Por fim, é gerada uma resolução trivial, onde:

$$(A = B) \implies (A + 1 = B + 1) \tag{3}$$

Outro lemma necessário foi c_bubbling_sum_n que define o custo de c_bubbling, onde foi provado que seu custo é igual ao segundo parâmetro da função, sendo este o número natural decrementado a cada recursão. Utilizamos os mesmos principios aplicados na prova do lemma anterior.

O próximo lemma importante foi bubblesort_aux_list_dont_chage_number que diz que duas listas de mesmo tamanho sendo dadas como primeiro parâmetro para a função c_bubblesort_aux, e esta função tendo um segundo parâmetro igual em ambas as instâncias o resultado do contator de comparações irá ser igual nas duas funções, de forma que o único valor influenciador para o contador seria o tamanho da lista. Nesse caso foi realizada uma indução forte, de uma forma um pouco diferente. Essa indução tem como base o segundo parâmetro de c_bubblesort_aux subtraido do natural '1' e possui um IF dentro de sua estrutura. Esse IF foi feito com o objetivo de descartar automaticamente os casos onde a hipótese de indução seja falsa (onde o número subtraído de '1' tenha um resultado negativo, gerando assim uma contradição, pois esse parâmetro deve obrigatoriamente ser natural), algo que sem este IF não necessariamente seria possível de ser provado. Utilizando a definição de c_bubblesort_aux foi provado automaticamente o caso simples, onde o valor do segundo parâmetro de c_bubblesort_aux é igual a zero. A hipótese da indução foi duplicada para que possa ser utilizada com dois parâmetros diferentes, o primeira hipótese foi instanciada com o parâmetro c_bubbling(lista 2, n - 1), já o segunda foi instanciada com o parâmetro c_bubbling(lista 1, n - 1), por fim, foi necessário utilizar o lemma de que c_bubbling preserva o tamanho da lista para ambos os casos, tanto para a lista 1 quanto para a lista 2. Para finalizar a prova foi utilizado o

lemma do custo de c_bubbling tanto para a lista 1 quanto para a lista 2. Assim, chegamos ao seguinte caso trivial:

( c_bubbling(lista 1, n) = n ) $\wedge$($c\_bubbling(lista2, n) = n$)$\wedge$
( c_bubblesort_aux(c_bubbling(lista 1, n - 1), n - 2) =
c_bubblesort_aux(lista 1, n - 2) ) $\wedge$
( c_bubblesort_aux(c_bubbling(lista 2, n - 1), n - 2) =
c_bubblesort_aux(lista 1, n - 2) ) $\Longrightarrow$
( c_bubblesort_aux(c_bubbling(lista 2, n - 1), n - 2) + n =
c_bubblesort_aux(c_bubbling(lista 1, n - 1), n - 2) + n )

O restante dos casos desta prova são os casos onde a hipótese de indução é falsa, então, é necessário provar um absurdo, bastando assim aplicar a própria definição dos parâmetros que geram hipótese falsa para gerar tal absurdo.

O próximo lemma a ser provado é o bubblesort_aux_sum_nquadratic onde é provado que o número de vezes que c_bubbling é executado a cada chamada de c_bubblesort_aux(l, n) = $(n^2 + n)/2$, para isso foi utilizado a indução forte de forma identica ao lemma bubblesort_aux_list_dont_chage_number. Foram provados separadamente os casos onde o n é igual a zero, (n - 1) é igual a zero, a lista é vazia e a lista é unitária, todos esses casos automaticamente, por serem casos de resultados de tamanhos definidos. Em seguida, foi instanciado a hipótese de indução com valor (n - 1). Foi aplicado o lemma bubblesort_aux_list_dont_chage_number para provar que a lista retornada por c_bubbling usada como parâmetro da recursão de c_bubblesort_aux não altera o contador de número de comparações retornado para c_bubblesort_aux. Em seguida, foi aplicado o lemma c_bubbling_preserv_length para garantir que o lemma bubblesort_aux_list_dont_chage_number é válido para a lista l dada como parâmetro de c_bubblesort_aux. Por fim, é aplicado um último lemma c_bubbling_sum_n para obter o custo de c_bubbling. Gerando assim um caso, onde: tanto a lista retornada por c_bubbling de l quanto a lista l, sendo listas usadas como parâmetro de uma função c_bubblesort_aux retornaram o mesmo valor, e c_bubblesort_aux(l, n - 1)‘2 = ( ( n - 1 ) * ( n - 1) - (n - 1) ) / 2 e que c_bubbling(l, n) = n implica que c_bubblesort_aux(c_bubbling(l, n - 1), n) + c_bubbling(l, n) = $(n^2 + n)/2$, algo que utilizando manipulações algébricas simples gera um caso trivial.

O último lemma provado foi bubbling_is_quadratic que prova que a complexidade temporal de Bubble Sort é quadrada no tamanho da lista de entrada. Para isso, não foi necessário o uso da indução, foi uma prova simples,

foram aplicadas as definições de Omicron e c_bubblesort e foram todos instanciados de forma que os elementos universais podem assumir qualquer valor e os elementos existenciais assumiram um valor específico, no caso o valor escolhido foi c = 2 e n0 = 0. Sabendo que o c é um valor que multiplicado pelo resultado de Omicron será maior que todos os possíveis resultados da função declarada no Omicron e n0 neste caso é o tamanho mínimo da lista do c_bubblesort. Essa prova foi dividida em dois casos: um caso simples onde a lista é nula, logo a definição é válida pois 0 = 0, e o caso onde entramos no c_bubblesort_aux. Para provar este caso foi aplicado o lemma do custo de c_bubblesort_aux (bubblesort_aux_sum_nquadratic) que foi instanciado com a própria lista que queremos provar e o tamanho da lista - 1. Lembrando que estes são os próprios parâmetros da função bubblesort que queremos provar. Assim, geramos uma verdade simples, onde c_bubblesort_aux(lista, n) = $((n-1)*(n-1)-(n-1))/2$, tendo n = tamanho da lista - 1, algo que é menor que $2n^2$ e então chegamos a um caso trivial. Portanto, a complexidade temporal de Bubble Sort $O(n^2)$, onde n é o tamanho da lista de entrada.

# 3 Conclusão

Conseguimos provar a complexidade temporal do algoritmo Bubble Sort que é $O(n^2)$ utilizando o software assistente de prova PVS. Esse trabalho não foi fácil, pois nenhum dos membros da dupla tinha experiência com o software e com formalização de provas, mesmo assim, foi possível aprender e aplicar vários conceitos ensinados nas aulas da matéria de Projeto e Análise de Algoritmos. Finalizamos o trabalho com sentimento de vitória e agradecemos o professor Flávio pelos ensinamentos.