Relatório - Análise da complexidade do algoritmo Merge Sort

Nicholas Nishimoto Marques - 15/0019343

25/11/2019

1 Introdução

Desde os princípios do curso de ciência da computação são estudados os algoritmos de ordenação, como por exemplo o BubbleSort, o QuickSort e o InsertionSort. Algoritmos desse tipo são utilizados geralmente em vetores ou listas, podendo ter pequenas variações dependendo da estrutura de dados trabalhada.

Tais algoritmos têm como objetivo ordenar (segundo um critério) uma determinada sequência de elementos. No caso o objeto de estudo do trabalho são listas de naturais, ou seja, o algoritmo estudado (Merge Sort) tem como entrada uma lista de naturais e deve retornar a mesma lista, porém ordenada (cada sucessor de um número na lista deve ser maior ou igual ao seu antecessor).

O objetivo do projeto é verificar se o algoritmo Merge Sort tem um tempo de execução da ordem de n*log(n), ou seja, computa a lista ordenada em um número de passos menor ou igual ao tamanho da lista multiplicado pelo logaritmo do tamanho da lista. Para provar este teorema utiliza-se a lógica proposicional de primeira ordem para, por meio do assistente de provas PVS, mostrar que para determinada lista de entrada do algoritmo de inserção binária seu retorno é uma lista de mesmo tamanho, com os mesmos elementos (é uma lista permutada da lista original) e com tais elementos dispostos de forma ordenada em um tempo n*log(n). Para isso provam-se alguns lemas que permitem provar o tempo de execução do algoritmo.

2 Contextualização do Problema

Algoritmos de ordenação são muito utilizados em ciência da computação para trabalhar-se com listas e vetores, estruturas de dados muito recorrentes na construção de qualquer programa, software. Tais algoritmos buscam a partir de uma entrada qualquer (uma lista ou vetor não ordenado) retornar após sua execução a mesma entrada porém ordenada (de acordo com uma noção de ordem estabelecida, como por exemplo uma noção de lista de naturais ordenada onde cada elemento é maior ou igual ao elemento que o mesmo sucede).

O algoritmo de ordenação Merge Sort funciona seguindo a lógica de dividir para conquistar. Calcula-se o ponto médio da lista atual, divide-se a lista em duas sub-listas com base nesse ponto médio, e recursivamente se faz isso juntando as 2 sub-listas de forma ordenada. A seguir tem-se um pseudo-código do algoritmo:
MERGE-SORT(A, p, r)

if p ¡ r then
q = ((p + r) / 2)
Merge-Sort(A, p, q)
Merge-Sort(A, q + 1, r)
Merge(A, p, q, r)

Merge(A, p, q, r)

n1 = q - p + 1

n2 = r - q

sejam L[1 ... n1 + 1] e R[1 ... n2 + 1]

for i = 1 to n1

L[i] = A[p + i - 1]

for j = 1 to n2

R[j] = A[q + j]

i = 1

j = 1

for k = p to r

if L[i] ¡= R[j] then A[k] = L[i]

i = i + 1

else A[k] = R[j]

j = j + 1

Exemplo em pseudo-código do algoritmo Merge Sort (disponível em https://pt.wikipedia.org/wiki/Merge sort).

3 Descrição da Formalização

A formalização da prova do funcionamento do Merge Sort se deu a partir da prova dos lemas propostos no arquivo disponibilizado na página do curso de Projeto e Análise de Algoritmo da Universidade de Brasília (ver referências), em conjunto com a utilização de algumas bibliotecas como a nasalib (para formalização de questões matemáticas e funções como o logaritmo) e também a bilbioteca *complexity*, que possibilita as notações assintóticas.

```
mergesort : THEORY

BEGIN

IMPORTING sorting
IMPORTING complexity
IMPORTING lnexp_fnd@ln_exp


l,l1,l2 : VAR list[nat]

% Merging (sorted lists)

merge(l1, l2 : list[nat]) : RECURSIVE list[nat] =
IF null?(l1) OR null?(l2) THEN append(l1, l2)
ELSIF car(l1) <= car(l2) THEN cons(car(l1), merge(cdr(l1),l2))
ELSE cons(car(l2), merge(l1, cdr(l2)))
ENDIF
MEASURE length(l1) + length(l2)

%%%%%%% merge_sort, the main function, sorts a list recursively
%%%%%%% using the function merge.
merge_sort(l) : RECURSIVE list[nat] =
 IF length(l) <= 1 THEN l
```

Figura 1: Lemas e bibliotecas utilizadas

O lema principal a ser provado foi o merge sort is nlogn, que indicava que o merge sort no pior caso tem sua execução em n log n passos, onde n é o tamanho da lista de input. Basicamente ele utilizava-se da definição de Omicron da biblioteca *complexity* para indicar a complexidade do algoritmo. Segue o algoritmo a seguir:

```
merge_sort_is_nlogn: LEMMA % mostrar que merge sort é n log n
 FORALL (l:list[nat]):
 member(LAMBDA(n:nat): cmerge_sort(l)`2,
 Omicron(LAMBDA(n:nat):length(l) * ln(length(l))))
```

Figura 2: Lema principal a ser provado

Para saber quantas comparações eram feitas, uma função auxiliar denominada cmerge sort foi feita, que é o algoritmo equivalente ao merge sort mas com um contador de comparações, justamente para saber o tempo em que o algoritmo roda. Os lemas utilizados nas provas levam em conta esse fato do contador. Alguns lemas auxiliares foram utilizados para prova final, que são detalhados a seguir:

```
cmerge(x:nat, lc1:[list[nat], nat], lc2:[list[nat], nat]) : RECURSIVE [list[nat], nat] =
IF null?(lc1`1) OR null?(lc2`1) THEN (append(lc1`1, lc2`1), lc1`2)
ELSIF car(lc1`1) <= car(lc2`1) THEN (cons(car(lc1`1), merge(cdr(lc1`1),lc2`1)), lc1`2 + lc2`2 + 1
)
ELSE (cons(car(lc2`1), merge(lc1`1, cdr(lc2`1))), lc1`2 + lc2`2 + 1)
ENDIF
MEASURE length(lc1`1) + length(lc2`1)

cmerge_sort(l: list[nat]) : RECURSIVE [list[nat], nat] =
 IF length(l) <= 1 THEN (l,0)
 ELSE cmerge(car(l), cmerge_sort(prefix(l,floor(length(l) / 2))),
                     cmerge_sort(suffix(l,floor(length(l)/2))))
 ENDIF
MEASURE length(l)

count_cmerge_ws_general: LEMMA
                         FORALL (x:nat) (lc1:[list[nat], nat]) (lc2:[list[nat], nat]):
                         cmerge(x, lc1, lc2)`2 <= length(lc1`1) + length(lc1`1) * ln(length(lc1`1
))


count_cmerge_sort_ws_general_nlogn: LEMMA
                  FORALL (l:list[nat]): cmerge_sort(l)`2 <= (length(l) * (ln(length(l)) + 1))
```

Figura 3: Lema principal a ser provado

É claro que era necessário provar a equivalência entre o merge sort com e sem o contador, portanto um lema adicional denominado merge cmerge are equivalent foi provado, mostrando que a função principal do Merge Sort (o merge) era equivalente com e sem contador).

```
merge_cmerge_are_equivalent: LEMMA
                             FORALL (x, n1, n2:nat) (l1:list[nat]) (l2:list[nat]):
                                    merge(l1, l2) = cmerge(x, (l1,n1), (l2, n2))`1
```

Figura 4: Lema de equivalência entre as funções de merge com e sem contador

A partir da prova de todos os lemas no arquivo pôde-se provar a complexidade de Merge Sort.

4 Prova da complexidade do algoritmo

O primeiro passo para a prova da complexidade do algoritmo foi de fato utilizar as funções equivalentes com contadores (cmerge, cmerge sort). Logo foi preciso provar a equivalência entre a função de merge e cmerge (merge sem e com contador). Foi uma prova bastante simples, dado que na prova olhamos apenas para primeira

componente de retorno da função cmerge (a lista final da junção das listas). A simples expansão das definições nos deu a prova final:

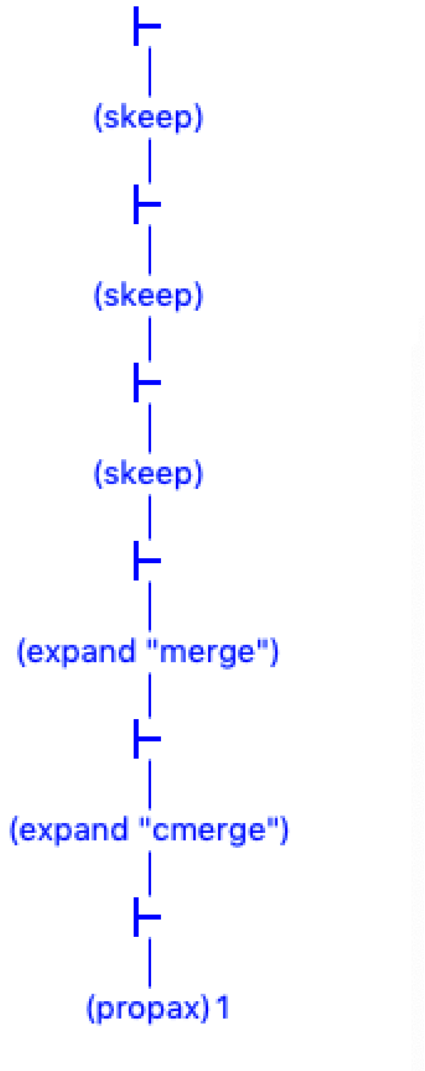

Figura 5: Árvore de prova da equivalência entre merge e cmerge

Alguns skeeps foram colocados para remover os universais do consequente, instânciando com listas genéricas l1 e l2 a prova.

Para o lema principal (que merge sort pertencia a Omicron de n log n) a árvore foi a seguinte:

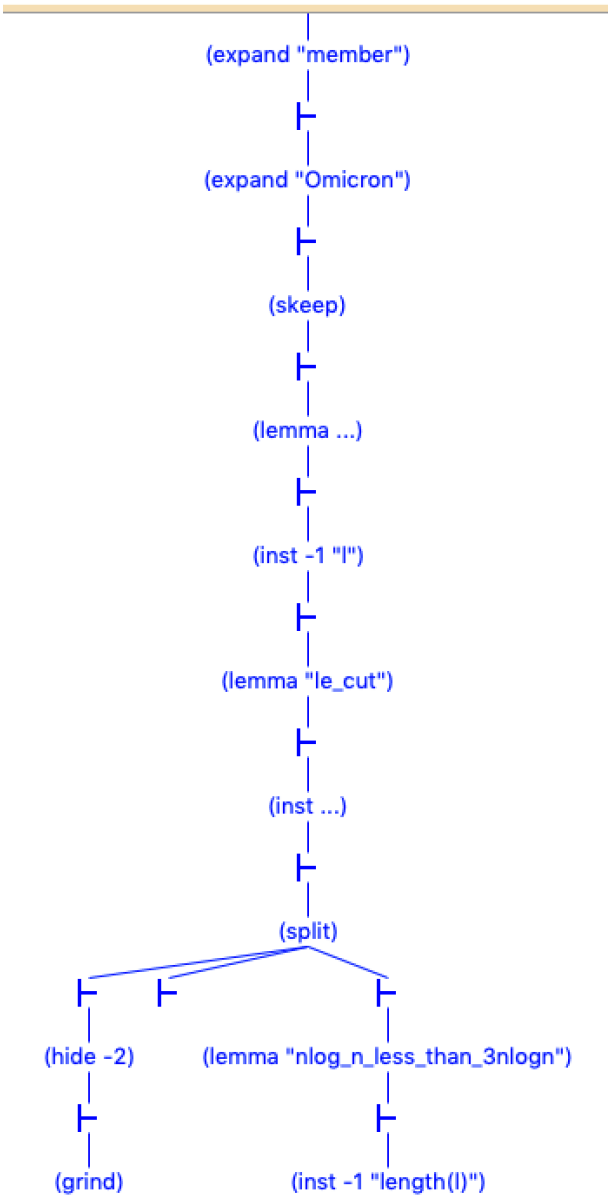

Figura 6: Árvore de prova principal para Merge Sort ser n log n

Os primeiros comandos de skeep são padrões para universais no consequente. Expandimos as definições presentes no lema (de member e Omicron) para provar que a classe estaria entre duas constantes nos naturais. Adicionamos o lema count_cmergesort_ws_general_nlogn, que nos diz que no pior caso o contador de iterações do merge sort terminará com o tamanho da lista (n) multiplicado pelo seu logaritmo.

A partir daí a prova sai de maneira natural. Na ramificação do split são gerados 3 casos. O primeiro é o que precisamos mostrar as constantes que tornam válidas merge sort pertencer a omicron de nlogn, que é fácil pois instanciamos anteriormente o lema com 3, que faz com que seja válido o ramo. No segundo ramo precisamos provar que cmerge_sort tem tempo nlogn, o que vem diretamente do lema adicionado. Por fim precisamos provar apenas algo algébrico, que

$$n * logn + 1 \leq 3 * n * logn \tag{1}$$

, onde utilizamos um lema criado anteriormente nlog_n_less_than_3nlogn, que é fácil verificar ser verdadeiro. Isso

conclui a prova do lema principal, restando ter que provar o lema auxiliar utilizado count_cmergesort_ws_general_nlogn.

A prova do lema auxiliar count_cmergesort_ws_general_nlogn foi a prova final que validou toda teoria. Para isso começamos com uma indução no tamanho da lista de entrada, para que para cada caso mostrassemos que o tempo de execução era da ordem de nlogn:

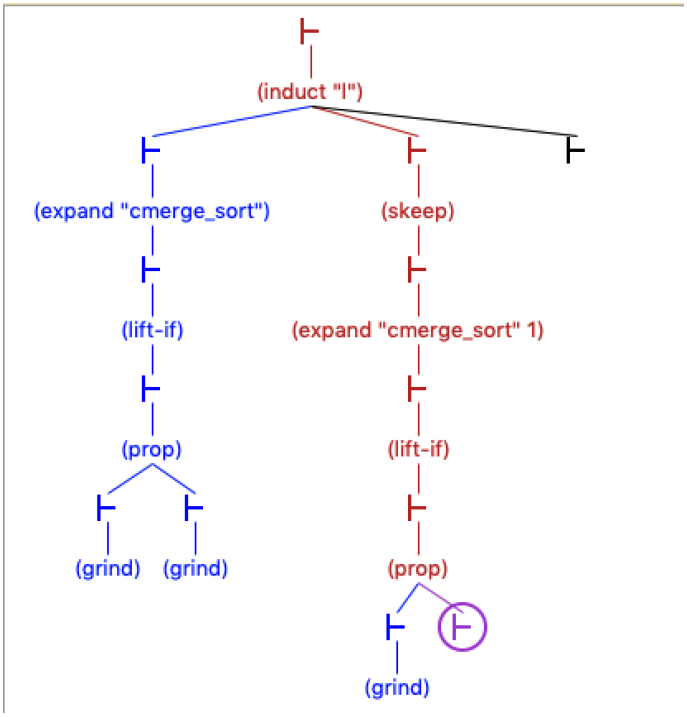

Figura 7: Árvore de prova principal para Merge Sort ser n log n

Para o caso base a resolução veio de maneira rápida, já que uma lista vazia ou com 1 elemento terá tempo de execução de 0, valor do logaritmo do tamanho da lista.

Para o caso indutivo a prova acabou ficando incompleta devido à dificuldades com o pvs com alguns comandos encima da hipótese de indução. Porém, com a prova do lema a teoria se deu por completa, provando que o algoritmo Merge Sort é executado na ordem de n logn, onde n é o tamanho da lista de entrada a ser ordenada.

5 Conclusões

A partir da formalização dos lemas, da construção das funções auxiliares e das provas foi possível concluir que o Merge Sort de fato tem seu tempo de execução da ordem de n logn. Isso condiz com a teoria de algoritmos e mostra que o algoritmo mesmo em seu pior caso tem um tempo considerado eficiente, mostrando a importância de se estudar tal algoritmo.

Apesar de um dos lemas ter ficado incompleto, a formalização das teorias, formalização dos lemas e a prova da maioria deles foi importante para o aprendizado sobre técnicas de provas formais e provas da complexidade de algoritmos, que é bastante útil não só para disciplina mas para casos práticos que precisamos otimizar a performance e prever comportamentos em aplicações reais de algoritmos.

Referências

[1] Mauricio Ayala-Rincón e Flávio L.C. de Moura. *Applied Logic for Computer Scientists*. Gewerbestrasse 11, 6330 Cham, Switzerland: Springer International Publishing AG, 2016.

[2] Paulo Feofiloff. *Mergesort: ordenação por intercalação*. URL: `https://www.ime.usp.br/~pf/algoritmos/aulas/mrgsrt.html`.

[3] Shankar N. et al. *PVS Prover Guide*. 333 Ravenswood Avenue - Menlo Park CA 94025: SRI International, 2001.

[4] Wikipédia. *Cálculo de sequentes*. URL: `https://pt.wikipedia.org/wiki/C%C3%A1lculo_de_sequentes`.

[5] Wikipédia. *Merge Sort*. URL: `https://pt.wikipedia.org/wiki/Merge_sort`.