Finitude, Corretude & Complexidade de Algoritmos

Oseas Francisco De Lemos Andre

20 de janeiro de 2025

Resumo

Neste trabalho é feita a análise de finitude, corretude e complexidade(complexidade de tempo) dos algoritmos de ordenação “Selection Sort” e “Buble Sort”. Visando praticar aspectos teóricos de ciência da computação e, avaliar o aproveitamento, das semanas 01 e 02 do curso “Tópicos de Programação”, oferecido pelo programa de verão "verão-IME-USP-2025".

Palavras-chave: Algoritmos, Complexidade, Selection, Bubble, Sort, BigO.

1 Introdução

O design de algoritmos é uma das fundações da Ciência da Computação, sendo essencial para a construção de soluções eficientes e corretas. No entanto, o ato de projetar algoritmos requer um ferramental teórico, exige atenção aos seguintes aspectos fundamentais: finitude, corretude e complexidade. Eles fornecem um framework sólido para avaliar a qualidade e a viabilidade de soluções computacionais.
Neste trabalho, exploramos esses três pilares teóricos por meio da análise dos algoritmos de ordenação Selection Sort e Bubble Sort. Apesar de simples, esses algoritmos oferecem um terreno fértil para examinar finitude, corretude e complexidade de algoritmos.

Atualmente temos grandes desafios como o crescimento exponencial de dados, a necessidade de respostas em tempo real e uso de computação distribuída, isso torna relevante o domínio desses conceitos teóricos.

2 Selection Sort

O Selection Sort é um dos algoritmos clássicos para ordenação de vetores.

Algorithm 1: Selection Sort*

Input: Um número inteiro n e uma lista $A[1...n]$

Output: A permutação da lista $A[1..n]$ com os n elementos ordenados

```
selectionsort(n, A)
    var i, j, menor : integer;
    for i ←1 to n do
        menor ← i;
        for j ← i + 1 to n do
            if A[j] < A[menor] then
                menor ← j;
            end
        end
        swap(A[menor], A[i]);
    end
end
```

* Adaptação do algoritmo “Selection Sort” do livro (ROBERT, 1983, p. 95).

2.1 Finitude

1. O laço externo executa exatamente n vezes,
2. A soma comutativa do número de execuções do laço interno é $(n^2 - n)/2$.

De 1) e 2 temos that esse algoritmo é finito, pois todos os laços executam um número finito de vezes.

2.2 Corretude

- Invariante: No final da i-ésima iteração do laço externo o sub vetor A[1...i] está ordenado. E o i-ésimo elemento está na posição correta da permutação A[1...n] ordenada.

"Prova por indução":

1. caso base, primeira iteração do laço externo: O sub vetor A[1...1], com um elemento está trivialmente ordenado.
2. Hípotese de indução: supondo que o invariante vale para uma iteração i qualquer.

3. Passo indutivo, mostrar que vale para $i + 1$: aqui é necessário discutir como o Algoritmo funciona.

    a) Na i-ésima iteração do laço externo(linhas 3 a 11) o algoritmo determina/seleciona o menor valor do sub vetor $A[i...n]$ e coloca ele no índice i. Portando no final da i-ésima iteração, do laço externo, o i-ésimo menor elemento do vetor, está na posição correta. Como?

    b) Inicialmente o índice do menor valor do sub vetor $A[i...n]$ é definido como i, linha 4,"$menor \leftarrow i$", em seguida o laço interno percorre o subvetor $A[i+1...n]$, linhas 5 a 9, compara $A[j]$ com $A[menor]$ e atualiza o valor de $menor$, caso algum $A[j]$ seja menor que $A[menor]$, linhas 6 a 8.

    c) No final do laço interno, linha 10, o algoritmo coloca o menor valor encontrado, no subvetor $A[i...n]$, no índice i e, o valor do índice i vai para o indice $menor$.

    d) Sabemos, por hipotese de indução, que no início da $i + 1$ iteração o sub vetor $A[1...i]$ está ordenado. De a (a) e (b) sabemos que no final da $i + 1$-ésima iteração o $i + 1$-ésimo elemento, da permutação ordenada, está na posição correta, o sub vetor $A[1...j + i]$ está ordenado.

4. Conclusão: De 3.(a), 3.(b) e 3.(c) e pelo fato de o laço externo ir progressivamente de 1 ate n, e o algoritmo ser finito, no final da n-ésima iteração o n-ésimo elemento vai estar na posição correta, completando a permutação ordenada. Então concluimos que o "Selection Sort" é correto.

2.3 Complexidade

Analisando complexidade de tempo do Selection Sort.

2.3.1 Pior caso

No pior caso o vetor está na ordem inversa. Da tabela abaixo temos que a complexidade no pior caso é $O(n^2)$.

Linha	Quantidade de Execuções
2	n
3	n
4	$(n^2 - n)/2$
5	$(n^2 - n)/2$
6	$(n^2 - n)/2$
9	n
Total	$f(n) = 3n + 3.(n^2 - n)/2$

Tabela 1 – Quantidade de execuções no Selection Sort: análise do pior caso.

2.3.2 Caso Médio

No caso médio a metade esquerda do vetor está ordenada. Da tabela abaixo temos que a complexidade no caso médio é $O(n^2)$.

Linha	Quantidade de Execuções
2	n
3	n
4	$(n^2 - n)/2$
5	$(n^2 - n)/4$
6	$(n^2 - n)/4$
9	n
Total	$f(n) = 3n + (n^2 - n)/2$

Tabela 2 – Quantidade de execuções no Selection Sort: análise do caso médio.

2.3.3 Melhor caso

No melhor caso o vetor já está ordenado. Da tabela abaixo temos que a complexidade no melhor caso é $O(n^2)$.

Linha	Quantidade de Execuções
2	n
3	n
4	$(n^2 - n)/2$
5	$(n^2 - n)/2$
6	0
9	n
Total	$f(n) = 3n + n^2 - n$

Tabela 3 – Quantidade de execuções no Selection Sort: análise do melhor caso.

3 Bubble sort

O Bubble Sort é outro algoritmo classico de ordenação ensinado nos cursos de ciência da computação.

Algorithm 2: Bubble Sort*

Input: Um número inteiro n e uma lista A[1..n]

Output: A permutação da lista A[1..n] com os n elementos ordenados.

```
1  bubblesort(n, A)
2      var j, swapped : integer;
3      repeat
4          swapped ← 0;
5          for j ← 2 to n do
6              if A[j − 1] > A[j] then
7                  swap(A[j − 1], A[j]);
8                  swapped ← 1;
9              end
10         end
11     until swapped = 0;
12 end
```

*Adaptação do algoritmo "Bubble Sort" do livro (ROBERT, 1983, p. 106).

3.1 Finitude

- Invariante: No final da k-ésima iteração do laço externo, linhas 3 a 11, o k-ésimo maior elemento da permutação ordenada está na posição correta. Portanto no final da k-ésima iteraçao o sub vetor $A[(n-k)+1...n]$ está ordenado.

1. No inicio de cada iteração do laço externo $swapped = 0$,

2. A cada iteração do laço externo, o laço interno, linhas 5 a 10, percorre subvetor $A[2...n]$. E compara os valores dos índices $j-1$ e j. Se $A[j-1]$ for maior que $A[j]$ o algoritmo troca o valor desses índices, colocando o valor de um no outro. Assim, "empurrando" o valor do k-ésimo maior elemento do vetor para a direita,

3. Quando $A[j-1]$ é maior que $A[j]$ o algoritmo também atribui 1 para a variável de controle *swapped*, indicando que houve alguma troca,

4. A condição de parada do laço externo é $swapped = 0$, quado em uma iteração do laço externo não forem feitas nenhuma troca,

5. Sabemos que no final da n-ésima iteração o n-ésimo maior elemento do vetor, vai estar na posição correta da permutação ordenada,

6. Portanto na $n+1$ iteração do laço externo, nenhuma troca vai ser feita. E no fim dessa iteração $swapped = 0$. De 3) sabemos que o laço externo termina.

De 1), 2), 3), 4), 5) e 6) concluímos que o “bubble sort” é finito, pois todos os seus laços executam um número finito de vezes.

3.2 Corretude

Aqui, vamos usar a discução sobre finitude para provar a corretude deste algoritmo.

- Invariante: No final da k-ésima iteração do laço externo, linhas 3 a 11, o k-ésimo maior elemento da permutação ordenada está na posição correta. Portanto no final da k-ésima iteraçao o sub vetor $A[(n-k)+1...n]$ está ordenado.

"Prova por indução"

1. caso base, primeira iteração do laço externo: de 3.1.2) sabemos que o sub vetor $A[n...n]$ está ordenado.
2. Hipótese de indução: supondo que o invariante vale para uma iteração i qualquer.
3. Passo indutivo, mostrar que vale para i + 1: No início da $i+1$-ésima iteração o sub vetor $A[(n-i)+1...n]$ está ordenado. De 3.1.2) sabemos que o $i+1$-ésimo elemento do vetor vai estar na posição correta da permutação ordenada. Portando no final da $i+1$-ésima iteração o sub vetor $A[(n-i...n)]$ está ordenado.
4. Conclusão: De 3.1 sabemos que o laço externo tem um número de iterações progressiva de 1 até $n+1$. Portanto de 1), 2) e 3) temos que na n-ésima iteração o vetor $A[1...n]$ está ordenado. Então concluímos que o “Bubble sort” é correto.

3.3 Complexidade

3.3.1 Pior caso

No pior caso do bubble sort o vetor está invertido, então além das comparações é feito o número máximo de trocas. Da tabela abaixo concluímos que a complexidade é $O(n^2)$ no pior caso.

Linha	Quantidade de Execuções
2	1
3	$n+1$
4	$n+1$
5	n^2-1
6	n^2-1
7	n^2-1
8	n^2-1
Total	$f(n)=1+2(n+1)+4.(n^2-1)$

Tabela 4 – Quantidade de execuções no Bubble sort: análise do pior caso.

3.3.2 Caso Médio

No caso médio do bubble sort, metade do vetor já está ordenada, o subvetor $A[(n/2)...n]$. Portanto, a consequência prática disso é reduzir o número de trocas. Da tabela abaixo temos que a complexidade é $O(n^2)$ para o caso médio.

Linha	Quantidade de Execuções
2	1
3	$n+1$
4	$n+1$
5	n^2-1
6	n^2-1
7	$(n^2-1)/2$
8	$(n^2-1)/2$
Total	$f(n) = 1 + 2(n+1) + 3(n^2-1)$

Tabela 5 – Quantidade de execuções no Bubble sort: análise do caso médio.

3.3.3 Melhor caso

No melhor caso o vetor já está ordenado, a consequência prática disso é não haver troca nenhuma. Da tabela abaixo temos que a complexidade é para é $O(n^2)$ do melhor caso.

Linha	Quantidade de Execuções
2	1
3	$n+1$
4	$n+1$
5	n^2-1
6	n^2-1
7	0
8	0
Total	$f(n) = 1 + 2(n+1) + 2(n^2-1)$

Tabela 6 – Quantidade de execuções no Bubble sort: análise do melhor caso.

Referências

ROBERT, S. *ALGORITHMS*. [S.l.]: Addison-Wesley Publishing Company Inc, 1983. 2, 5