Universidade Federal da Paraíba
Centro de Informática - CI

Funções de complexidade - Algoritmos de Ordenação

Professor:
Gilberto Farias

Aluno:
Rafael Praxedes

Sumário

1 Introdução

Na computação, algoritmos são fundamentais na resolução de muitos problemas. Dentre tais problemas, está o de ordenação, para o qual há diversos métodos que o resolvem, sendo uns mais eficientes do que outros. Desse modo, o presente relatório objetiva analisar tais algoritmos de ordenação, comparando-se a eficiência de cada um deles. Importante ressaltar que todas as análises realizadas tem como base os fundamentos e conceitos contidos na obra "Algoritmos: Teoria e Prática"de Thomas H. Cormen, (Cormen, 2002).

2 Selection Sort

Nesta seção será feita uma análise detalhada sobre o cálculo da função de complexidade para o método *Selection Sort*, (Sambol, 2016a), ilustrado na Fig.1. Para tanto, a Tab.1 apresenta os custos de cada uma das instruções que o compõem, bem como a quantidade de vezes em que cada uma delas é executada. Importante ressaltar que n equivale ao tamanho do vetor de entrada a ser ordenado.

```
void SelectionSort(vector<long int> &inputVector)
{
    int min_pos;                                              // (1)
    long int aux;                                             // (2)

    for (size_t i = 0; i < inputVector.size() - 1; i++){      // (3)

        min_pos = i;                                          // (4)

        for (size_t j = i + 1; j < inputVector.size(); j++){  // (5)
            if(inputVector[min_pos] > inputVector[j]){        // (6)
                min_pos = j;                                  // (7)
            }
        }
        if(min_pos != i){                                     // (8)
            aux = inputVector[i];                             // (9)
            inputVector[i] = inputVector[min_pos];            // (10)
            inputVector[min_pos] = aux;                       // (11)
        }
    }
}
```

Figura 1: Selection Sort

Tabela 1: Instruções - Selection Sort

Instrução	**Custo**	**Número de execuções**
1	C1	1
2	C2	1
3	C3	n
4	C4	n-1
5	C5	$\frac{n*(n+1)}{2} - 1$
6	C6	$\frac{n*(n-1)}{2}$
7	C7	$\frac{n*(n-1)}{2}$
8	C8	(n-1)
9	C9	(n-1)
10	C10	(n-1)
11	C11	(n-1)

Dentre tais instruções, faz-se necessária a explicação sobre o número de execuções para C5, C6, C7, C9, C10 e C11. Quanto a primeira, a análise do algoritmo permite inferir a quantidade de vez que a instrução C5 é executada para cada um dos valores do contador i do *loop* For mais externo. A Tab.2 sintetiza essa inferência.

Tabela 2: Execuções da instrução C5

i	**j**	**Número de execuções**
0	1,...,n	n
1	2,...,n	n-1
2	3,...,n	n-2
...	...	...
(n-4)	(n-3),...,n	4
(n-3)	(n-2),...,n	3
(n-2)	(n-1),...,n	2

Por meio da análise da Tab.2, tem-se que o número de execuções total da instrução C5 é dado por, (1).

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

Para a instrução C6, o raciocínio é análogo ao que foi feito para C5, exceto pelo fato de que, para cada uma das iterações, isto é, para cada um dos valores de

i, C6 é executada uma vez a menos do que C5. Por essa razão, a quantidade total de execuções da mesma equivale a, (2).

$$\sum_{k=2}^{n}(k-1) = \frac{n*(n-1)}{2} \tag{2}$$

Para a instrução C7, por se tratar de uma instrução pertencente a um bloco if, não há uma certeza sobre a quantidade exata de vezes que a mesma será executada. Desse modo, pode-se representá-la por meio da equação (3), na qual t_j representa o tempo de execução da instrução para cada um dos valores de j.

$$\sum_{j=1}^{n-1} t_j \tag{3}$$

Apesar disso, um limite superior para (3) é conhecido. Isto é, a instrução será executada no máximo a mesma quantidade de vezes que a verificação da condição do bloco if, assumindo que a mesma seja verdadeira em todos os casos. Portanto, por simplificação, assume-se que a quantidade total de execuções de C7 é dada por (4).

$$\frac{n*(n-1)}{2} \tag{4}$$

Para C9, C10 e C11, é válido o mesmo raciocínio. Isto é, dada que todas tenham suas execuções representadas por (5), sendo t_i o tempo de execução de cada uma dessas instruções, para cada um dos valores de i.

$$\sum_{i=0}^{n-2} t_i \tag{5}$$

Tem-se que o limite superior para a execução é dalas é igual a (6) e, por essa razão, é tal valor que está presente na Tab.1.

$$n-1 \tag{6}$$

Em suma, a função T_n que mede a complexidade para o algoritmo do *Selection Sort* pode ser obtido pela multiplicação das colunas “Custo”e “Número de

Execuções”da Tab.1, sendo ela a soma de todos eles. Desse modo, após algumas manipulações algébricas, tem-se que a função $T(n)$ equivale a, (7),

$$T_n = an^2 + bn + c \tag{7}$$

Sendo as contantes a, b e c representadas por (8), (9) e (10), respectivamente.

$$a = \frac{C5 + C6 + C7}{2} \tag{8}$$

$$b = C3 + C4 + \frac{C5}{2} - \frac{C6}{2} - \frac{C7}{2} + C8 + C9 + C10 + C11 \tag{9}$$

$$c = C1 + C2 - C4 - C5 - C9 - C10 - C11 \tag{10}$$

3 Insertion Sort

Nesta seção será feita uma análise detalhada sobre o cálculo da função de complexidade para o método *Insertion Sort*, (Sambol, 2016b), ilustrado na Fig.2. Para tanto, a Tab.3 apresenta os custos de cada uma das instruções que o compõem, bem como a quantidade de vezes em que cada uma delas é executada. Importante ressaltar que n equivale ao tamanho do vetor de entrada a ser ordenado.

```
void InsertionSort(vector<long int> &inputVector)
{
    long int aux;                                          // (1)
    size_t j;                                              // (2)

    for (size_t i = 1; i < inputVector.size(); i++){       // (3)
        j = i;                                             // (4)

        while (j > 0 && inputVector[j-1] > inputVector[j]){   // (5)
            aux = inputVector[j-1];                        // (6)
            inputVector[j-1] = inputVector[j];             // (7)
            inputVector[j] = aux;                          // (8)

            j--;                                           // (9)
        }
    }
}
```

Figura 2: Insertion Sort

Tabela 3: Instruções - Insertion Sort

Instrução	**Custo**	**Número de execuções**
1	C1	1
2	C2	1
3	C3	n
4	C4	n-1
5	C5	$\frac{n*(n+1)}{2} - 1$
6	C6	$\frac{n*(n-1)}{2}$
7	C7	$\frac{n*(n-1)}{2}$
8	C8	$\frac{n*(n-1)}{2}$
9	C9	$\frac{n*(n-1)}{2}$

O desenvolvimento da $T(n)$ para o *Insertion Sort* é análogo ao que foi feito para o *Selection Sort* (v. Seção 2). Desse modo, seguindo o mesmo procedimento realizado, tem-se, após algumas manipulações algébricas, que a função $T(n)$ equivale a, (11),

$$T(n) = an^2 + bn + c \tag{11}$$

Sendo as contantes a, b e c representadas por (12), (13) e (14), respectivamente.

$$a = \frac{C5 + C6 + C7 + C8 + C9}{2} \tag{12}$$

$$b = C3 + C4 + \frac{C5}{2} - \frac{C6}{2} - \frac{C7}{2} - \frac{C8}{2} - \frac{C9}{2} \tag{13}$$

$$c = C1 + C2 - C4 - C5 \tag{14}$$

Referências

Cormen, Thomas H. Algoritmos: Teoria e prática, tradução da segunda edição [americana] Vandenberg D. de Souza. Rio de Janeiro: Elsevier, 2002 - 6ª Reimpressão.

YouTube - Michael Sambol, *Selection Sort in 3 minutes*, disponível em <https://www.youtube.com/watch?v=g-PGLbMth_g>. Acesso em: 09 de abr. 2020.

YouTube - Michael Sambol, *Insertion Sort in 2 minutes*, disponível em <https://www.youtube.com/watch?v=JU767SDMDvA>. Acesso em: 09 de abr. 2020.