Sistemas de Informação
6º Período
04/02/2021

IFMG - Ouro Branco
Diego Santos Seabra
0040251

Tarefa 3

Projeto e Análise de Algoritmos

Hospedado no Github

Contents

Exercício 3 **2**
Equação de Recorrência 2
Código Implementado 3

Exercício 4 **5**
Pseudocódigo 5
Código Implementado 6
Explicação 7

3) A ordenação por inserção pode ser expressa sob a forma de um procedimento recursivo como a seguir. Para ordenar $A[1 \ldots n]$, ordenamos recursivamente $A[1 \ldots n-1]$ e depois inserimos $A[n]$ no arranjo ordenado $A[1 \ldots n-1]$. Escreva uma recorrência para o tempo de execução dessa versão recursiva da ordenação por inserção

Solução:

Equação de Recorrência

Primeiramente, considerando que $T(n)$ é o tempo para ordenar n elementos, executar a ordenação $A[1 \ldots n-1]$ tem, então, um tempo de execução de $T(n-1)$.

Já inserir $A[n]$ no arranjo ordenado $A[1 \ldots n-1]$, em seu pior caso, terá uma complexidade de $O(n)$ e, portanto nossa equação de recorrência deve ser:

$$T(n) = T(n-1) + O(n) \tag{1}$$

Código Implementado

```cpp
////////////////////////////////////////
///                                  ///
/// Autor: Diego S. Seabra           ///
/// Matricula: 0040251               ///
///                                  ///
////////////////////////////////////////

#include <iostream>

#define TAMANHO_ARRAY 10

using namespace std;

// n >> tamanho
void insertionSort(int A[], int n)
{
    // Caso base
    if (n ≤ 1)
    {
        return;
    }

    // Ordena recursivamente A[1 .. n-1]
    insertionSort(A, n - 1);

    // Insere A[n] no arrajndo ordenado A[1..n-1]
    int chave = A[n - 1];
    int j = n - 2;

    while (j ≥ 0 && A[j] > chave)
    {
        A[j + 1] = A[j];
        j--;
    }
    A[j + 1] = chave;
}

void imprimeArray(int A[], int n)
{
    for (int i = 0; i < n; ++i)
    {
        cout << A[i] << " ";
    }
    cout << endl;
}

int main()
{
    // Inicia o arranjo A com os inteiros
    int A[TAMANHO_ARRAY] = {10, 8, 2, 5, 3, 4, 1, 9, 6, 7}; // custo: 1
```

```
51
52      cout << "Antes" << endl;
53      imprimeArray(A, TAMANHO_ARRAY);
54
55      insertionSort(A, TAMANHO_ARRAY);
56
57      cout << "Depois" << endl;
58      imprimeArray(A, TAMANHO_ARRAY);
59
60      // Finaliza o programa
61      return 0;                                    // custo: 1
62  }
```

4) Descreva um algoritmo de tempo O(n.lg(n)) que, dado um conjunto S de n inteiros e outro inteiro x, determine se existem ou não dois elementos em S cuja soma seja exatamente x

Solução:

Pseudocódigo

```
1: existeX(S, x):
2: mergeSort(S, 1, tamanho[S])
3: p ← 1
4: q ← tamanho[S]
5: while p < q do
6:    if S[p] + S[q] > x then
7:       q ← q − 1
8:    else
9:       if S[p] + S[q] < x then
10:         p ← p + 1
11:      else
12:         return true
13:      end if
14:   end if
15: end while
16: return false
```

Código Implementado

```cpp
////////////////////////////////////////
///                                  ///
/// Autor: Diego S. Seabra           ///
/// Matricula: 0040251               ///
///                                  ///
////////////////////////////////////////

#include <iostream>
#include "merge_sort.h"

#define TAMANHO_ARRAY 10

using namespace std;

bool existeX(const int S[TAMANHO_ARRAY], int x)   // custo total: O(n)
{
    // Divide o conjunto em dois lados (esquerda e direita)
    // p >> indice da esquerda / q >> indice da direita
    int p = 0;                          // custo: 1 (pseudo: p <- 1)
    int q = TAMANHO_ARRAY - 1;          // custo: 1 (pseudo: q <- tamanho[S])

    // Enquanto os indices nao forem iguais
    while (p < q)                       // custo: n
    {
        // Como o conjunto esta ordenado em ordem crescente,
        // se o valor da soma eh maior que o X passado,
        // diminui o indice da direita
        if (S[p] + S[q] > x)            // custo: n-1
        {
            q = q - 1;
        } else
        {
            if (S[p] + S[q] < x)
            {
                // Do contrario, aumenta o indice da esquerda
                p = p + 1;
            } else
            {
                // Se o valor da soma for exatamente igual a X,
                // para de rodar o loop (melhor caso)
                // No pior caso, o codigo nunca chegara
                // ate aqui e retornara false apos o while
                return true;
            }
        }
    }
    return false;                       // custo: 1
}

int main()
```

```
51 {
52     // Inicia o conjunto S com os inteiros
53     int S[TAMANHO_ARRAY] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}; // custo: 1
54
55     // Ordena usando o merge sort >> custo de n.log(n)
56     mergeSort(S, 0, TAMANHO_ARRAY - 1);      // custo: n.log(n)
57
58     // Verifica se existe o elemento X no conjunto S
59     if (existeX(S, 23))                          // custo: n
60     {
61         cout << "Existem 2 valores que somados sao iguais a x" << endl;
62     }
63
64     // Finaliza o programa
65     return 0;                                    // custo: 1
66 }
```

Explicação

Primeiramente o código foi implementado usando 2 *for loops*, o que gerou um $O(n)$ igual a n^2. Como era necessário que o $O(n)$ fosse $n \cdot log(n)$, o algoritmo foi modificado para ser feito "de uma única vez" e portanto era necessário que o conjunto estivesse sempre ordenado.

Como sabemos, o merge sort possui a complexidade de $O(n) = n \cdot log(n)$, portanto foi o algoritmo usado para ordenar os itens do conjunto.

Após os itens ordenados, são criados 2 índices (p e q), que se referem à partes da "esquerda" e "direita" do conjunto. Dependendo do valor do x e da soma dos índices, esses índices são acrescidos ou diminuídos a fim de chegar ao valor da soma que se iguala a x (essa verificação possui $O(n) = n$).

Se somarmos as complexidades, teremos ao final $O(n) = n \cdot log(n)$. Veja:

$$T(n) = n \cdot log(n) + \not{n} \tag{2}$$

$$O(n) = n \cdot log(n) \tag{3}$$

Obs.: No pior caso, o algoritmo não encontra dois elementos cuja soma é igual a x e passa por todos os itens do conjunto (por isso o $O(n) = n$ na verificação da soma).

Diego Santos Seabra
0040251