Exame CAL 2020.2

Daniella Martins Vasconcellos

[1]Departamento de Ciência da Computação - Universidade do Estado de Santa Catarina (UDESC)

daniellavasconc@gmail.com

1. Questão 1

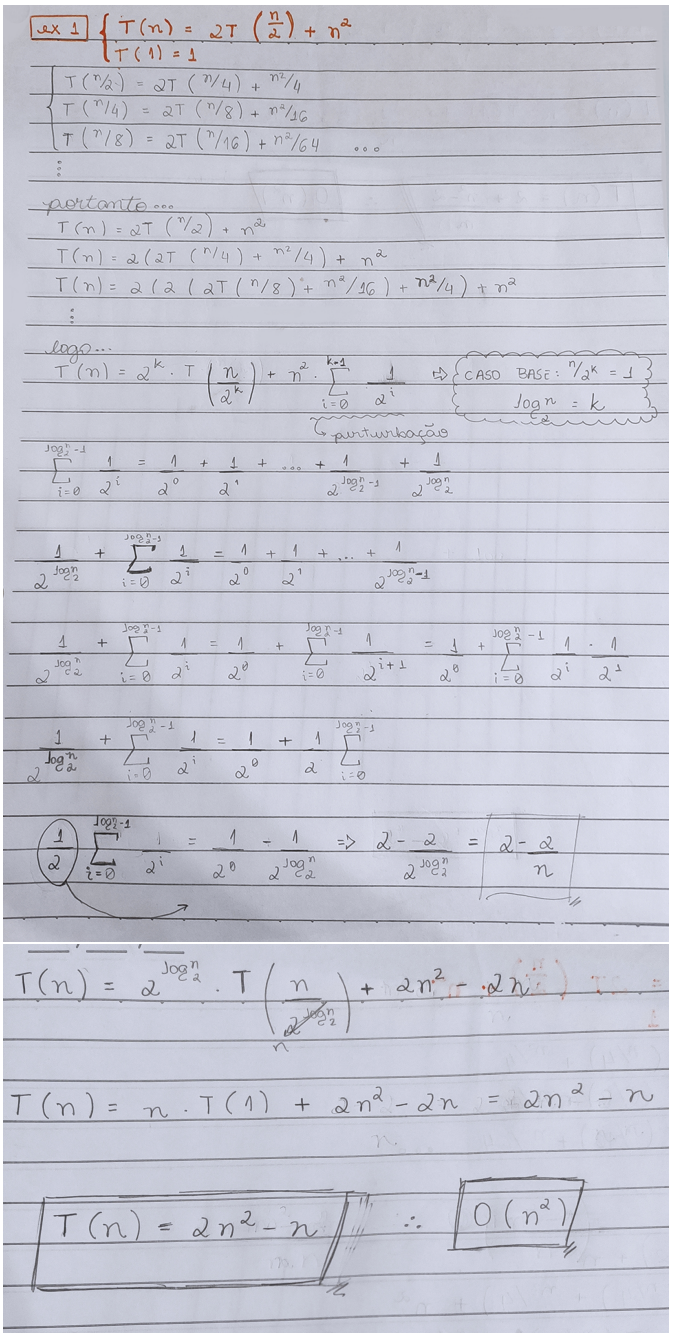

Figure 1. Resposta da Questão 1.

2. Questão 2

Abaixo está o código a ser analisado junto com a enumeração das linhas para melhor entendimento.

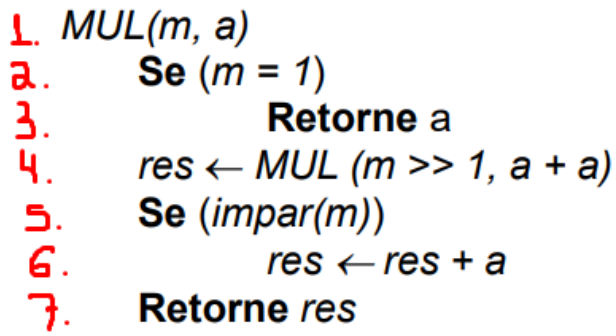

Figure 2. Detalhamento de linhas do código mostrado na questão 2.

2.1. A. Número de bits de *m* e *a* são limitados ao tamanho da palavra do processador

As linhas 3 e 7 possuem complexidade constante, porque retornamos o valor das variáveis.

Na linha 5, uma única operação precisa ser feita: Conferir o último bit do número para saber se ímpar (dígito final 1), o que também resulta em complexidade constante.

Na linha 6, observamos um comportamento constante já que qualquer operação de soma em um número fixo do processador é constante.

Na linha 2, verificamos se um número é igual a 1, então é necessário conferir todos os bits à esquerda do último digito: Se todos os dígitos forem 0, e o último 1, então isso significa que m = 1; se qualquer outro dígito à esquerda for diferente de 0, então m é diferente de 1. Neste caso, como o tamanho das variáveis é limitado ao tamanho da palavra do processador, então isso significa que essa complexidade é de mesmo tamanho que o tamanho da palavra; que, como é constante, também resulta em complexidade constante.

Agora, na linha 4, precisamos prestar mais atenção. Em $m >> 1$, executaremos o mesmo número de chamadas recursivas tal qual o tamanho de m; que, como é constante, torna a complexidade nesta parte igualmente constante. Tal raciocínio também pode ser aplicado à $a + a$. Por fim, podemos executar o seguinte cálculo para a chamada recursiva:

$$T(n) = T(n-1) + 1$$
$$T(n-1) = T(n-2) + 1$$
$$T(n-2) = T(n-3) + 1$$

...Logo...

$$T(n) = T(n-k) + k * 1, onde : n - k = 1, portanto : k = n - 1$$
$$T(n) = T(1) + (n-1) * 1$$
$$T(n) = 1 + n - 1$$
$$T(n) = n$$

Chegando à mesma conclusão que também possui complexidade de tempo O(n). Novamente, considerando o tamanho como constante, a complexidade assume tamanho constante de maneira análoga.

2.1.1. *Conclusão*

Analisando todas as linhas do código, podemos concluir que a complexidade de tempo é O(tamanho da palavra); como sabemos que o tamanho da palavra é constante, então concluímos que a **complexidade de tempo é O(1)**. Da mesma forma, a **complexidade de espaço é O(1)**, porque o programa faz um número de chamadas recursivas igual ao tamanho da palavra, que, nesse caso, é constante.

2.2. B. Não há limite para o número de bits de *m* e *a*

Por não possuir um limite de bits, todas as afirmações feitas sobre complexidade constante nos últimos tópicos não possui mais validade nesta situação. Na verdade, onde foi afirmado que seria constante por conta da constância do tamanho da variável, irá mudar para, justamente, o tamanho da variável. Ou seja, nas linhas 2 (O($|m|^2$)), 4 (O($|m+a|^2$)), 5 (O($|m|^2$)) e 6 (O($|a|^2$), uma vez que agora a soma não tem mais um valor fixo e precisará ser computada bit a bit; ou seja, justamente o tamanho da variável *a*).

Assumimos T(1) como $|m|$, pois no caso base o programa percorrerá toda a extensão de *m*.

$$T(n) = T(n-1) + |m+a|$$
$$T(n-1) = T(n-2) + |m+a|$$
$$T(n-2) = T(n-3) + |m+a|$$

...Logo...

$$T(n) = T(n-k) + k * |m+a|, onde : n - k = 1, portanto : k = n - 1$$
$$T(n) = T(1) + (n-1) * |m+a|$$
$$T(n) = |m| + n * |m+a| - |m+a|$$

2.2.1. *Conclusão*

Como a recursão está sempre deslocando um bit de m a cada iteração, podemos afirmar que o programa fará $|m|$ chamadas recursivas, e, para tanto, $n = |m|$. Substituindo n por $|m|$ na última linha da relação de recorrência passada, chegamos à conclusão que **complexidade de tempo é O($|\boldsymbol{m}|^2$)**, o pior caso da multiplicação de $|\boldsymbol{m}| * |\boldsymbol{m} + \boldsymbol{a}|$. Analogamente, a **complexidade de tempo é O($|\boldsymbol{m}|$)**, justamente por ser o número de chamadas recursivas a serem realizadas.

3. Questão 3

Um problema x é NP-difícil se todos os problemas NP-Completos podem ser reduzidos a ele. Tendo essa definição em mente, o seguinte algoritmo feito em C++ confere se um determinado grafo admite bicoloração.

```
int n;
vector<vector<int>> adj;

vector<int> side(n, -1);
bool is_bipartite = true;
queue<int> q;
for (int st = 0; st < n; ++st) {
    if (side[st] == -1) {
        q.push(st);
        side[st] = 0;
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int u : adj[v]) {
                if (side[u] == -1) {
                    side[u] = side[v] ^ 1;
                    q.push(u);
                } else {
                    is_bipartite &= side[u] != side[v];
                }
            }
        }
    }
}

cout << (is_bipartite ? "YES" : "NO") << endl;
```

O problema foi resolvido, mas isso não prova que a redução é um problema NP-Difícil; afinal, o algoritmo acima é executável em tempo polinomial. Ou seja, não há qualquer problema NP-Completo existente dentro do algoritmo que pudesse ser reduzido a um NP-Difícil.

4. Questão 4

Abaixo, para a prova, utilizaremos o algoritmo ***Pollar-Rho***, cujo objetivo é fatorar um número n em dois outros primos, p e q. Sua **complexidade é O($n^{\frac{1}{4}} * \log n$)**. Dentro da prova, também usaremos o algoritmo de ***Euclides (MDC ou GCD)*** para encontrar o maior divisor comum entre dois números; a complexidade para esse algoritmo é **O($\log n$)**.

Algoritmo 1: Pollar-Rho

```
i = 1
x_1 = random(0, n-1)
while True do
    i = i+1
    x_i = (x_{i-1}^2 - 1) mod n
    d = MDC(y - x_i, n)
    if d ≠ 1 e d ≠ n then
        print d
    end
    if i == k then
        y = x_i
        k = 2k
    end
end
```

Como estamos usando um algoritmo que é uma heurística, nem seu tempo de execução nem seu sucesso é garantido (embora o procedimento seja muito eficiente na prática). Dessa forma estaremos usando na prova um algoritmo não determinístico (ou seja, um algoritmo cujo comportamento não é previsível). Sendo a classe NP um acrônimo de *Non-Deterministic Polynomial Time*, então podemos inferir que o algoritmo de fatoração de um inteiro é pertencente à classe NP.