Exame de CAL 2020.2 – Vinícius Takeo

Questão 1: Relação de Recorrência

$$T(N) = 2T(n/2) + n^2$$

$$T(1) = 1$$

$$T(N) = 2T(n/2) + n^2$$

$$2T(n/2) = 2.2T(n/2^2) + 2.(n/2)^2$$

$$2^2T(n/2^2) = 2^3 T(n/2^3) + 2^2 (n/2^2)^2$$

$$2^3 T(n/2^3) = 2^4 T(n/2^4) + 2^3 (n/2^3)^2$$

$$T(N) = 2^l T(n/2^l) + 2^{l-1}(n/2^{l-1})^2 + 2^{l-2}(n/2^{l-2})^2 + \dots + n^2$$

$$T(N) = 2^l \underbrace{\cancel{T(1)}}_{1} + \sum_{i=0}^{l-1} 2^i \left(\frac{n}{2^i}\right)^2$$

$$\frac{n}{2^l} = 1 \Rightarrow 2^l = n \Rightarrow l = \log_2 n$$

$$T(N) = 2^{\log_2 n} + \sum_{i=0}^{l-1} \frac{\cancel{2^i} . n^2}{\cancel{2^i} . 2^i}$$

$$T(N) = n + \sum_{i=0}^{l-1} \frac{n^2}{2^i}$$

$$T(N) = n + n^2 \sum_{i=0}^{l-1} \frac{1}{2^i}$$

Resolvendo o somatório:

$$\sum_{i=0}^{l-1} \frac{1}{2^i} \Rightarrow \text{substituindo } l-1 \text{ por } k:$$

$$\sum_{i=0}^{k} \frac{1}{2^i} = \frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \dots + \frac{1}{2^k} + \frac{1}{2^{k+1}}$$

$$\sum_{i=0}^{k} \frac{1}{2^i} + \frac{1}{2^{k+1}} = \frac{1}{2^0} + \frac{1}{2^1} + \dots + \frac{1}{2^k} + \frac{1}{2^{k+1}}$$

$$\sum_{i=0}^{k} \frac{1}{2^i} + \frac{1}{2^{k+1}} = \frac{1}{2^0} + \sum_{i=0}^{k} \frac{1}{2^{i+1}}$$

$$\sum_{i=0}^{k} \frac{1}{2^i} + \frac{1}{2^{k+1}} = \frac{1}{1} + \sum_{i=0}^{k} \frac{1}{2.2^i}$$

$$\sum_{i=0}^{k} \frac{1}{2^i} + \frac{1}{2.2^k} = 1 + \frac{1}{2} \sum_{i=0}^{k} \frac{1}{2^i}$$

$$\frac{1}{2} \sum_{i=0}^{k} \frac{1}{2^i} = 1 - \frac{1}{2.2^k}$$

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 \left[1 - \frac{1}{2.2^k} \right]$$

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 - \frac{2}{2.2^k}$$

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 - \frac{1}{2^k}$$

Voltando a substituição: $k = l - 1$

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 - \frac{1}{2^k} \Rightarrow \sum_{i=0}^{l-1} \frac{1}{2^i} = 2 - \frac{1}{2^{l-1}}$$

$$\sum_{i=0}^{l-1} \frac{1}{2^i} = 2 - \frac{1}{\frac{2^l}{2}} \Rightarrow \sum_{i=0}^{l-1} \frac{1}{2^i} = 2 - \frac{2}{2^l}$$

Voltando a relação de recorrência:

$$T(N) = n + n^2 \sum_{i=0}^{l-1} \frac{1}{2^i}$$

$$T(N) = n + n^2 \left[2 - \frac{2}{2^l} \right]$$

Sabendo que $l = \log_2 n$ então $2^l \Rightarrow 2^{\log_2 n} \Rightarrow n$

logo:

$$T(N) = n + n^2 \left[2 - \frac{2}{n} \right]$$

$$T(N) = n + n^2 \left[\frac{2n-2}{n} \right]$$

$$T(N) = n + \frac{2n^3 - 2n^2}{n}$$

$$T(N) = n + 2n^2 - 2n$$

$$T(N) = 2n^2 - n$$

Questão 2: Complexidade da Multiplicação

- Letra a: Quando o número de bits de m e a são limitados pelo processador

Quando eles são limitados pelo número de bits do processador, a grande maioria das operações ocorrem em tempo constante. Veja o cálculo na imagem a seguir:

MUL(m, a)
Se (*m* = *1*) O(1)
  Retorne a
res ← MUL (*m* >> *1*, *a* + *a*) O(1) O(1)
Se (*impar*(*m*)) O(1)
  res ← *res* + *a* O(1)
Retorne *res*

T(n) = T(n-1) + 1
T(n-1) = T(n-2) + 1
...
T(n) = T(1) + n*1
T(n) = 1 + n => O(n)

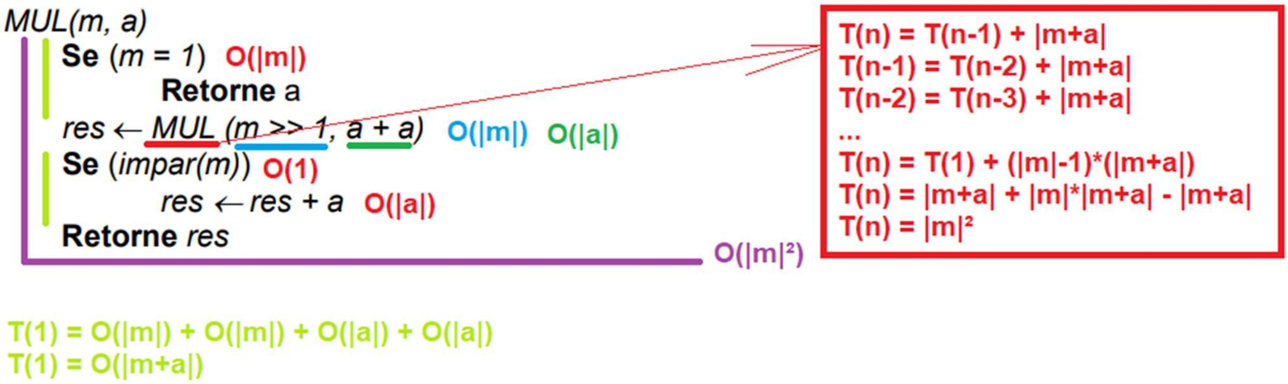
Como ocorrem O(|m|) chamadas recursivas, a complexidade de espaço é O(|m|) , mas como |m| é constante, a complexidade de espaço é constante O(1)

O(|m|) como o tamanho da palavra é constante, então a complexidade de tempo é O(1)

Como o número de bits é limitado, verificar se o valor é 1 acontece em tempo constante, tal como o right shift e a soma de a. A verificação do valor ser ímpar também (basta analisar o último bit) e a soma de res com a também. Logo, para resolver a relação de recorrência, que chama a função n-1 vezes onde n é o número de bits, basta levar em consideração o caso base que tem tempo 1 e suas n chamadas. Ao final tem-se que a complexidade é O(n) onde n é o tamanho da palavra do processador. Como tal valor é constante, a complexidade de tempo também é, resultando na **complexidade de tempo total do algoritmo de O(1).** Já na complexidade de espaço, por ocorrerem n-1 chamadas recursivas (novamente n é o tamanho da palavra do processador), a complexidade espaço é O(|m|) e como |m| é constante, temos que a **complexidade de espaço é O(1)**.

Letra b: Quando o tamanho de m e a são ilimitados

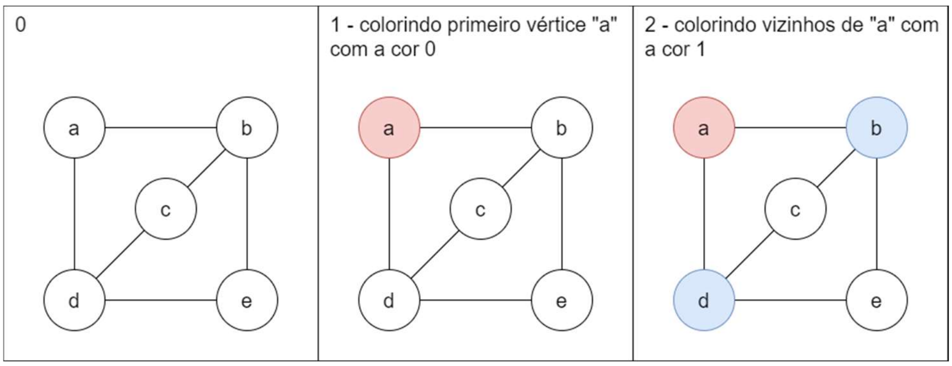
A complexidade de tempo do caso base é O(|m+a|), já que o maior valor dominará o menor nessa adição. A verificação se o valor é igual a 1, possui tempo O(|m|), a verificação se o valor é ímpar possui tempo constante, basta verificar o valor do último bit. Já a soma de res com a, possui tempo O(|a|). O deslocamento de bits possui complexidade O(|m|) e o cálculo do dobro do valor de a possui complexidade O(|a|), o que faz com que o caso base tenha complexidade igual a O(|m+a|). Por fim, resolvendo a relação de recorrência, temos que a **complexidade de tempo total do algoritmo é O(|m|²)**.

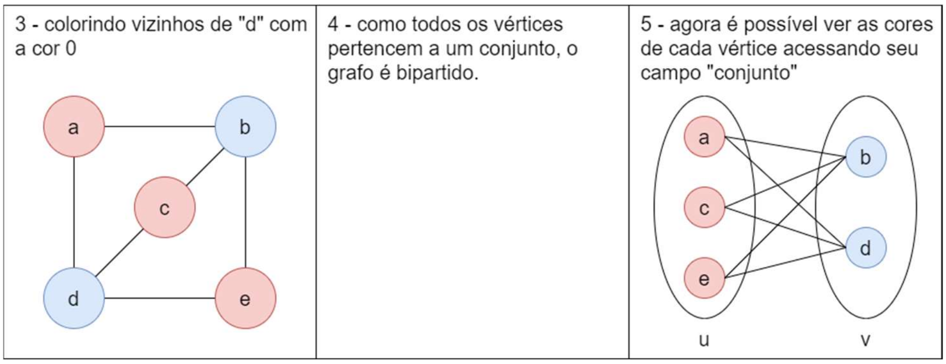


Como ocorrem |m|- 1 chamadas recursivas, logo, a **complexidade espaço é O(|m|)**.

Questão 3: 2-Graph-Coloring reduzido a Verificação de bipartidade de um grafo

Utilizando o algoritmo descrito abaixo, apresenta-se uma redução do problema de 2-coloração ao de determinar se um grafo é bipartido:





O algoritmo utilizado para resolver o problema possui o seguinte pseudocódigo:

Algorithm 1: Algorithm to check the Bipartiteness of a Graph

```
Data: G(V, E), S
Result: Bipartite or Not Bipartite
r= NULL; O(1)
S = G.firstVertex O(1)
r.enqueue(S); O(1)
while r is not empty do O(|V|²)
    n1 = r.dequeue(); O(1)
    for each n2 in n1.Adj() do O(|V|)
        if color.n2 == NIL then O(1)
            if color.n1 == RED then O(1)
                color.n2 = BLUE; O(1)
            else
                color.n2 = RED; O(1)
                r.enqueue(n2) ; O(1)
            end
            if color.n2 == color.n1 then O(1)
                retorne FALSE O(1)
            end
        end
    end
end
retorne TRUE
```

Complexidade do algoritmo 1: T(n) = O(|V|²)

Porém, tal redução não prova nada, visto que ambos os problemas pertencem a classe P (note que o algoritmo resolve o problema em tempo polinomial, com complexidade quadrática). Essa redução não prova nada a respeito de estar ou não na classe NP-Difícil, visto que para isso seria necessário reduzir todos os problemas NP a ele, porém essa redução não apresentou nenhum problema NP.

Questão 4: Fatoração de números inteiros

Para provar que a fatoração de números inteiros de qualquer tamanho em números primos pertence a classe NP, é necessário apresentar um algoritmo verificador de tempo polinomial. Para isso, apresenta-se o procedimento abaixo, utilizando do algoritmo AKS (NEMANA; VENKAIAH, 2016) com sua respectiva complexidade:

```
verifica(n,p,q)
        se ehPrimoAKS(p) == false:
                retorne false
        se ehPrimoAKS(q) == false:
                retorne false
        mult = p * q
        se mult for diferente de n:
                retorne false
        retorne true
```

O(log^12 |p|)

O(log^12 |q|)

O((|p|+|q|) ²)

O(|n|+|p+q|²)

T(n) = O(log^12 |p|) + O(log^12 |q|) + O((|p| + |q|)²) + O(|n|+|p+q|²)

T(n) = O ((log|p|)^12) + (log|q|)^12))

Logo, a complexidade de tempo do verificador é polinomial

NEMANA, L. K.; VENKAIAH, V. C. **An empirical study towards refiningthe aks primality testing algorithm**. IACR Cryptol. ePrint Arch., v. 2016,p. 362, 2016.