Solução Lista 01 - Divisão e Conquista - CANA 2025.1

Cornélio A. de S.

Maio 2025

Q1

Podemos refatorar a soma $1 + c + c^2 + ... + c^n$ usando a fórmula da soma dos $n + 1$ primeiros termos de uma P.G.:

$$g(n) = 1 + c + c^2 + ... + c^n = \frac{1 - c^{n+1}}{1 - c}$$

Agora temos que analisar o crescimento assintótico de $g(n)$. Lembrando que para provar que $f(n) = \Theta(g(n))$ precisamos provar que $f(n) = \Omega(g(n))$ e $f(n) = O(g(n))$.

(a) $c < 1$

Para provar $g(n) = O(1)$ quando $c < 1$, precisamos achar constantes k e n_0 de modo que:

$$0 \leq k \cdot g(n) \leq 1 \quad \forall \quad n \geq n_0$$

Para tal, podemos escolher $k = 1 - c$, de modo que $k \cdot g(n) = 1 - c^{n+1}$ que é < 1 para valores de $n \geq 0$ (escolher $n_0 = 0$).

Agora, para provar $g(n) = \Omega(1)$, precisamos achar outras duas constantes $k^{'}$ e $n_0^{'}$ de modo que:

$$0 \leq 1 \leq k^{'} \cdot g(n) \quad \forall \quad n \geq n_0^{'}$$

Ou seja:

$$k^{'} \geq g(n)^{-1} = \frac{1 - c}{1 - c^{n+1}}$$

Para $c < 1$ temos $c^b \leq c$ quando $b > 0$ o que implica em $1 - c \leq 1 - c^b$. Portanto $\frac{1-c}{1-c^{n+1}} \leq 1$ quando $n \geq 0$. Dessa forma, podemos escolher qualquer valor de $k^{'} \geq 1$ e escolher $n_0^{'} \geq 0$ para provar $g(n) = O(1)$.

Dessa forma, $g(n) = \Theta(1)$ quando $c < 1$. ■

(b) $c = 1$

Quando $c = 1$ ficamos com $g(n) = 1 + 1 + ... + 1 = n + 1 = \Theta(n)$. ■

(c) $c > 1$

Primeiramente vamos mostrar que $g(n) = \frac{1-c^{n+1}}{1-c} = O(c^n)$:

$$0 \leq k \cdot \frac{1-c^{n+1}}{1-c} \leq c^n \quad \forall \quad n \geq n_0$$

$$\Rightarrow 0 \leq k \leq \frac{c^n \cdot (1-c)}{1-c^{n+1}}$$

Precisamos escolher um valor de k que mantenha essa inequação verdadeira quando n cresce idefinidamente:

$$k \leq \lim_{n\to\infty} \frac{c^n \cdot (1-c)}{1-c^{n+1}} = \lim_{n\to\infty} \frac{c^n \cdot (1-c)}{c^n \cdot (\frac{1}{c^n} - c)} = \frac{c-1}{c}$$

Escolhendo $k = (c-1)/c$:

$$0 \leq k \cdot \frac{1-c^{n+1}}{1-c} = \frac{c-1}{c} \cdot \frac{1-c^{n+1}}{1-c} = \frac{c^{n+1}-1}{c} = c^n - \frac{1}{c} \leq c^n \quad \forall \quad n \geq n_0 = 0$$

Mostremos agora que $g(n) = \Omega(c^n)$, ou seja:

$$0 \leq c^n \leq k \cdot \frac{1-c^{n+1}}{1-c} \quad \forall \quad n \geq n_0$$

Nossa inequação passa a ser:

$$k \geq \frac{c^n \cdot (1-c)}{1-c^{n+1}}$$

Quando n cresce indefinidamente:

$$k \geq \frac{c-1}{c}$$

Podemos simplesmente escolher $k = c - 1$:

$$0 \leq c^n \leq k \cdot \frac{1-c^{n+1}}{1-c} = (c-1) \cdot \frac{1-c^{n+1}}{1-c} = c^{n+1} - 1$$

Obviamente a prova acima supõem que $c^{n+1} - 1 \geq c^n$, o que é obvio para valores de $c >> 1$ e $n > 0$. Porém, para valores de c muito proximos de 1 (mas ainda $c > 1$), precisamos determinar com cuidado um valor para n_0. Mas, para qualquer valor de $c > 1$, podemos determinar um valor de n_0 suficientemente grande que faça $c^{n+1} - 1 \geq c^n$ verdadeiro. ■

Q2

(a)

Os primeiros termos da sequência de Fibonacci são: [0, 1, 1, 2, 3, 5, 8, 13...]. Ou seja $F_6 = 8$ e $F_7 = 13$. Ambos serão o nossa caso base:

$$F_6 = 8 \geq 2^{0.5\times 6} = 2^{6/2} = 2^3 = 8 \quad \text{e}$$

$$F_7 = 13 \geq 2^{7/2} \approx 11.3$$

Provado o caso base, a nossa hipótese indutiva é que $F_n \geq 2^{n/2}$ e $F_{n-1} \geq 2^{(n-1)/2}$. O passo indutivo consiste em provar $F_{n+1} \geq 2^{(n+1)/2}$ a partir dessa hipótese:

$$F_{n+1} = F_n + F_{n-1} \geq 2^{n/2} + 2^{(n-1)/2} = 2^{n/2} + 2^{(n/2)-(1/2)}$$
$$\Rightarrow F_{n+1} \geq 2^{n/2} + \frac{2^{n/2}}{2^{1/2}} = 2^{n/2} \cdot \left(1 + \frac{1}{\sqrt{2}}\right)$$

Decompondo $2^{(n+1)/2}$ obtemos $2^{n/2} \cdot \sqrt{2}$.
Facilmente verificamos que $1 + \frac{1}{\sqrt{2}} \approx 1.71 > \sqrt{2} \approx 1.41$. Portanto:

$$F_{n+1} \geq 2^{n/2} \cdot \left(1 + \frac{1}{\sqrt{2}}\right) \geq 2^{n/2} \cdot \sqrt{2} = 2^{(n+1)/2}$$

O que prova o passo indutivo e consequentemente prova que $F_n \geq 2^{n/2}$ para $n \geq 6$. ■

(b)

Com base na prova da alternativa (a) acima, temos:

$$2^{n/2} \leq F_n < 2^n$$

$F_n < 2^n$ é facilmente verificado.
Pontanto, nosso intervalo de procura é $c \in (\frac{1}{2}, 1)$, de modo que:

$$F_n \leq 2^{cn}$$

Usando as mesmas ideias da alternativa anterior, vamos supor que c é conhecido e que queremos provar $F_n \leq 2^{cn}$ usando indução. Vamos deixar o caso base para o fim e considerar logo o passo indutiva: queremos partir de $F_n \leq 2^{cn}$ e $F_{n-1} \leq 2^{c(n-1)}$ (hipótese indutiva) e mostrar que $F_{n+1} \leq 2^{c(n+1)}$. O começo dessa prova parte da decomposição de F_{n+1}:

$$F_{n+1} = F_n + F_{n-1} \leq 2^{cn} + 2^{c(n-1)} = 2^{cn} \cdot \left(1 + \frac{1}{2^c}\right)$$

Para "provar" $F_n \leq 2^{cn}$ a partir da equação acima, temos que mostrar que:

$$2^{cn} \cdot \left(1 + \frac{1}{2^c}\right) \leq 2^{c(n+1)} = 2^{cn} \cdot 2^c$$

Ou seja, provar que:

$$\left(1 + \frac{1}{2^c}\right) \leq 2^c$$

Portanto, qualquer valor de c que obdecer a inequação acima (e também for verdade para os casos base F_0 e F_1) resulta em $F_n \leq 2^{cn}$ por indução. Vemos que $c = 1/2$ não obedece, pois $1 + \frac{1}{\sqrt{2}} \approx 1.71 \nleq \sqrt{2} \approx 1.41$. Já sabemos disso, pois provamos na alternativa (a) que $F_n \geq 2^{n/2}$.
Fazendo alguns cálculos (transformar a inequação acima em uma função quadrática $x^2 - x - 1 \geq 0$, onde $x = 2^c$) achamos que $c = 3/4$ obedece a inequação:

$$\left(1 + \frac{1}{2^{3/4}}\right) \approx 1.595 \leq 2^{3/4} \approx 1.682$$

Portanto $c = 0.75$ é uma solução que faz $F_n \leq 2^{cn}$, uma vez que segue pela hipótese indutiva e vale para os casos base:

$$F_0 = 0 \leq 2^{0.75 \cdot 0} = 1$$
$$F_1 = 1 \leq 2^{0.75 \cdot 1} = 2^{0.75} \approx 1.682$$

■

(c)

$F_n = \Omega(2^{cn})$ equivale à:

$$0 \leq k \cdot 2^{cn} \leq F_n$$

Podemos simplesmente desconsiderar k, pois:

$$k \cdot 2^{cn} = 2^w \cdot 2^{cn} = 2^{(w+cn)} = 2^{n(\frac{w}{n}+c)} = 2^{c'n}$$

Portanto, queremos achar o maior valor de c em que $2^{cn} \leq F_n$ para todo $n \geq n_0$, semelhante a alternatava (a) (mas mais difícil). A diferença para a alternativa (a) é que estamos trabalhando com notação assintótica, então podemos escolher n_0 como qualquer valor finito não negativo (não temos um ponto de partida definido, como o F_6 da alternativa (a)).

Vamos esquecer o "ponto de partida" por enquanto (que seria o caso base da indução) e focar no passo indutivo: dado como verdade que $F_n \geq 2^{cn}$ e $F_{n-1} \geq 2^{c(n-1)}$, quais valores de c permitem que partamos dessas verdades e provemos $F_{n+1} \geq 2^{c(n+1)}$? Vamos aplicar as mesmas ideias da alternativa (b):

$$F_{n+1} = F_n + F_{n-1} \geq 2^{cn} + 2^{c(n-1)} = 2^{cn} \cdot \left(1 + \frac{1}{2^c}\right)$$

Dessa forma, para aplicar o passo de indução, c deve obdecer a seguinte inequação:

$$F_{n+1} \geq 2^{cn} \cdot \left(1 + \frac{1}{2^c}\right) \geq 2^{c(n+1)} = 2^{cn} \cdot 2^c$$

Ou seja:

$$\left(1 + \frac{1}{2^c}\right) \geq 2^c \quad (\times 2^c)$$
$$\Rightarrow 2^c + 1 \geq (2^c)^2$$
$$\Rightarrow x^2 - x - 1 \leq 0 \quad \text{onde} \quad x = 2^c, c > 0, x > 1$$

O maior valor de x que obedece essa inequação é:

$$x = \frac{1+\sqrt{5}}{2} = \varphi \approx 1.6180$$

O que nos dá um valor de $c = \log_2(\frac{1+\sqrt{5}}{2}) \approx 0.6942$. Ou seja, $c = \log_2(\frac{1+\sqrt{5}}{2})$ é o maior valor que permite provamos o passo indutivo a partir da hipótese indutiva:

$$2^{cn} = 2^{n \cdot \log_2(\frac{1+\sqrt{5}}{2})} = \varphi^n$$

Sabe-se que a taxa de crescimento da sequência de Fibonacci se aproxima da taxa de crescimento de φ^n (exponencial do número de ouro) para valores de n suficientemente grandes, ou seja, F_n e φ^n possuem uma taxa de crescimento assintótico semelhante. Dessa forma, se escolhermos um valor de c suficientemente menor que $\log_2 \varphi$, a taxa de crescimento da sequência de Fibonacci será ligeiramente maior que a taxa de crescimento de 2^{cn}, o que implicaria que a partir de algum valor de n suficientemente grande teriamos $F_n \geq 2^{cn}$, que é basicamente dizer que $F_n = \Omega(2^{cn})$.

Por exemplo, se escolhermos $c = 0.6$, $2^{0.6 \cdot n}$ supera F_n para valores pequenos de n. Por exemplo, $F_6 = 8 < 2^{0.6 \cdot 6} \approx 12.13$. Porém, para valores maiores de n, F_n supera $2^{0.6 \cdot n}$. Por exemplo, para $n = 100$, temos $F_{100} = 3,736,710,778,780,434,432 > 2^{0.6 \cdot 100} = 2^{60} \approx 1,152,921,504,606,846,976$.

Q3

```python
"""Merge sort."""

def merge(arr, i, k, j):
    """
    Inplace merge of ordered arrays 'arr[i:k+1]' and
    'arr[k+1:j+1]' into a fully ordered array 'arr[i:j+1]'.
    """
    if not i <= k <= j:
        raise ValueError("Must have 'i' <= 'k' <= 'j'")
    cp = list(arr[i:j+1]) # Copy of arr from i to j
    stop = (k-i, j-i)
    lft_idx = 0
    rgt_idx = stop[0] + 1
    ptr = i
    while lft_idx <= stop[0] and rgt_idx <= stop[1]:
        if cp[lft_idx] <= cp[rgt_idx]:
            arr[ptr] = cp[lft_idx]
            lft_idx += 1
        else:
            arr[ptr] = cp[rgt_idx]
            rgt_idx += 1
        ptr += 1
    if lft_idx > stop[0]:
        arr[ptr:j+1] = cp[rgt_idx:stop[1]+1]
    else:
        arr[ptr:j+1] = cp[lft_idx:stop[0]+1]
    return

def merge_sort(arr, i, j, merge_routine=merge):
```

```
    """
    Inplace sorting of 'arr' from 'i' to 'j',
    both inclusive.
    """
    if i == j:
        return arr
    if i > j:
        raise ValueError("'j' must be >= 'i'")
    mid = (i + j) // 2
    merge_sort(arr, i, mid, merge_routine)
    merge_sort(arr, mid+1, j, merge_routine)
    merge_routine(arr, i, mid, j)
    return arr

if __name__ == "__main__":
    import sys
    if len(sys.argv) != 4:
        print("Example usage: python msort.py 45,87,42,50 1 3")
        sys.exit(1)
    try:
        arr = [int(i) for i in sys.argv[1].split(",")]
        i = int(sys.argv[2])
        j = int(sys.argv[3])
    except ValueError:
        print("Use only base 10 integers!")
        print("Example usage: python msort.py 45,87,42,50 1 3")
        sys.exit(2)
    if not 0 <= i <= j < len(arr):
        print("Bad 0-based indexing!")
        print("Example usage: python msort.py 45,87,42,50 1 3")
        sys.exit(4)
    print(merge_sort(arr, i, j))
```

Q4

```
"""Fibonacci sequence using matrix multiplication."""

import numpy as np
from math import log2

num_muls = 0

def matrix_mul(A, B):
    """Matrix multiplication with a counter."""
    global num_muls
    num_muls += 1
    return A @ B
```

```python
def fib_pow(n):
    """Raise 'A' to a power of 'n'."""
    A = np.array([[0, 1], [1, 1]], dtype="uint64")
    if n == 1:
        return A
    if n < 1:
        raise ValueError("'n' must be positive")
    is_odd = bool(n % 2)
    half = fib_pow(n // 2)
    if is_odd:
        return matrix_mul(matrix_mul(half, half), A)
    else:
        return matrix_mul(half, half)

def fib(n):
    """Calculate the 'n'-th Fibonacci number."""
    if n == 0:
        return 0
    # '@' as a dot product
    return int(fib_pow(n)[0] @ [0, 1])

if __name__ == "__main__":
    import sys
    if len(sys.argv) != 2:
        print("Usage: python fib.py <base 10 integer>")
        sys.exit(1)
    try:
        n = int(sys.argv[1])
    except ValueError:
        print("Usage: python fib.py <base 10 integer>")
        sys.exit(2)
    print(f"Fib_{n} = {fib(n)}")
    print(f"> {num_muls} matrix multiplications total")
    if n: print(f"> log2({n}) ~= {log2(n):.3f}")
```

Q5

Nós podemos utilizar o algoritmo *merge-sort* para resolver este problema. A modificação que deve ser feita é na rotina *merge*, que é a rotina que faz as operações de ordenação propriamente dita e, consequentemente, é a operação que resolve as inversões presentes na lista a ser ordenada. Na rotina *merge*, temos a *array* da "esquerda" sendo comparada com a *array* da "direita". Essas *arrays* recebem esses nomes porque justamente uma está em uma posição à esquerda da outra dentro da *array* completa. Durante o *merge*, quando um elemento da *array* da direita é dito como menor que o elemento da *array* da esquerda, todos os elementos na *array* da esquerda a partir do "atual" (o que está sendo comparado) são inversões desse menor elemento. Quando esse elemento da direita é colocado no seu devido lugar, todas essas inversões são resolvidas. A operação *merge* modificada fica:

```python
"""Merge sort with inversions counter."""

from msort import merge_sort

num_inversions = 0

def inv_merge(arr, i, k, j):
    """
    Inplace merge of ordered arrays 'arr[i:k+1]' and
    'arr[k+1:j+1]' into a fully ordered array 'arr[i:j+1]'.

    Additionally, count the number of inversions encountered.
    """
    global num_inversions # <<< Mod
    if not i <= k <= j:
        raise ValueError("Must have 'i' <= 'k' <= 'j'")
    cp = list(arr[i:j+1]) # Copy of arr from i to j
    stop = (k-i, j-i)
    lft_idx = 0
    rgt_idx = stop[0]+1
    ptr = i
    while lft_idx <= stop[0] and rgt_idx <= stop[1]:
        if cp[lft_idx] <= cp[rgt_idx]:
            arr[ptr] = cp[lft_idx]
            lft_idx += 1
        else:
            # This modification add just a couple more
            # constant time operations, keeping merge O(n)
            # and merge_sort O(n.log(n))
            num_inversions += (1 + stop[0] - lft_idx) # <<< Mod
            arr[ptr] = cp[rgt_idx]
            rgt_idx += 1
        ptr += 1
    if lft_idx > stop[0]:
        arr[ptr:j+1] = cp[rgt_idx:stop[1]+1]
    else:
        arr[ptr:j+1] = cp[lft_idx:stop[0]+1]
    return arr

if __name__ == "__main__":
    import sys
    if len(sys.argv) != 4:
        print("Example usage: python msort.py 45,87,42,50 1 3")
        sys.exit(1)
    try:
        arr = [int(i) for i in sys.argv[1].split(",")]
        i = int(sys.argv[2])
        j = int(sys.argv[3])
    except ValueError:
        print("Use only base 10 integers!")
```

```
        print("Example usage: python msort.py 45,87,42,50 1 3")
        sys.exit(2)
    if not 0 <= i <= j < len(arr):
        print("Bad 0-based indexing!")
        print("Example usage: python msort.py 45,87,42,50 1 3")
        sys.exit(4)
    print(merge_sort(arr, i, j, merge_routine=inv_merge))
    print("Number of inversions:", num_inversions)
```

Essa modificação adiciona algumas operações de tempo constante (apenas somas), mantendo a operação *merge* como $O(n)$ e o *merge-sort* $O(n \log n)$.

Q6

Para resolver este problema em $O(n \log n)$ é necessário perceber que, dado que uma *array* de n elementos possui um elemento majoritário, digamos que esse elemento seja k, se dividirmos essa *array* em duas partes, pelo menos uma das partes também terá k como elemento majoritário.

Para provar isso, chamemos a *array* de tamanho n de A_T e digamos que ela tenha k como elemento majoritário. Para k ser majoritário, precisamos que A_T tenha pelo menos $\lfloor n/2 \rfloor + 1$ elementos k. Vamos dividir A_T em duas *arrays* de tamanhos d e $n - d$, chamemos elas de A_L e A_R, respectivamente. A ideia é que A_L tenha o maior número de elementos k possível sem que ele se torne majoritário. Queremos fazer isso para tirar o máximo de elementos k de A_R e tentar fazer com que k não seja elemento majoritário em nenhuma delas.

(1º caso) d par e n par: Com d par, podemos ter $d/2$ elementos k em A_L sem que ele se torne majoritário. Com n também par, temos que ter pelo menos $\lfloor n/2 \rfloor + 1 = n/2 + 1$ elementos k no total. Precisamos considerar apenas o pior caso, onde temos exatamente $n/2+1$ elementos k em A_T. Dessa forma, o número máximo de elementos k em A_R sem que ele se torne majoritário é $(n - d)/2$, mas nos temos $n/2 + 1 - d/2 = (n - d)/2 + 1$. Portanto, para d e n par, A_R obrigatoriamente tem k como elemento majoritário.

(2º caso) d par e n ímpar: Ainda temos o limite de $d/2$ elementos k em A_L e agora temos $\lfloor n/2 \rfloor + 1 = (n+1)/2$ elementos k no total. O número máximo de elementos k em A_R sem que ele se torne majoritário é $(n - d - 1)/2$, porém nos temos $(n + 1)/2 - d/2 = (n - d + 1)/2$, o que faz de k um elemento majoritário de A_R.

Para d ímpar a prova não muda basicamente nada. ■

Dessa forma, temos o algoritmo:

```
"""Computes the majority element of an array."""

def check_majority(arr, lft_candidate, rgt_candidate):
    lft_count = 0
    rgt_count = 0
    for elem in arr:
        if elem == lft_candidate:
            lft_count += 1
        if elem == rgt_candidate:
            rgt_count += 1
    limit = len(arr) // 2
    if lft_count > limit:
        return lft_candidate
```

```
    elif rgt_count > limit:
        return rgt_candidate
    else:
        return None

def get_majority(arr):
    if not arr:
        raise ValueError("'arr' can't be empty.")
    if len(arr) == 1:
        return arr[0]
    mid = len(arr) // 2
    lft_candidate = get_majority(arr[:mid])
    rgt_candidate = get_majority(arr[mid:])
    return check_majority(arr, lft_candidate, rgt_candidate)

if __name__ == "__main__":
    import sys
    if len(sys.argv) != 2:
        print("Example usage: python majority.py 10,4,10,7,10")
        sys.exit(1)
    try:
        arr = [int(i) for i in sys.argv[1].split(",")]
    except ValueError:
        print("Use only base 10 integers!")
        print("Example usage: python majority.py 10,4,10,7,10")
        sys.exit(2)
    majority = get_majority(arr)
    if majority is None:
        print("There's no majority.")
    else:
        print("Majority:", majority)
```

A função *get-majority* realiza um número constante de operações simples (complexidade $O(1)$), dividindo o problema em dois subproblemas semelhantes, mas com metade do tamanho. Os resultados desses subproblemas são depois mesclados usando a função *check-majority*, que possui complexidade $O(n)$ e que utiliza apenas comparações de igualdade. A equação de recurssão é:

$$T(n) = 2 \cdot T(n/2) + O(n)$$

Resolvendo a recurssão usando o Teorema Mestre chegamos numa complexidade $O(n \log n)$.