Estruturas de Dados - Análise de Complexidade

Análise de complexidade

- Esquema de um modelo de computação:
    - memória:
        - armazena o programa e os dados
    - unidade de processamento:
        - é capaz de realizar operações lógicas e aritméticas
    - contador de programa
        - dita o ritmo da execução de um programa
        - faz chamada à unidade de processamento com uma operação a ser realizada
- Comunicação entre esses três elementos

Análise de complexidade

- Como medir o tempo de execução de um algoritmo nesse modelo?
- Ideia: passo de um algoritmo
  - é uma sequência de operações de um algoritmo cujo tempo de execução não depende da instância do problema
- Assim, basta-nos contar o número de passos executados

Análise de complexidade

Algoritmo: Fatorial(n)

Entrada: natural n

Saída: valor de n!

```
1 nn = n
2 enquanto n > 1 faça
3     n = n − 1
4     nn = nn · n
5 retorne nn
```

Note que:

- as linhas 3 e 4 podem ser consideradas um só passo
- o loop das linhas 2 a 4 não pode ser um passo, pois depende do valor de n

Análise de complexidade: fatorial

Algoritmo: Fatorial(n)

Entrada: natural n

Saída: valor de n!

```
1 nn = n
2 enquanto n > 1 faça
3     n = n − 1
4     nn = nn · n
5 retorne nn
```

passo	1	2	3	4	5
excuções	1	n	n-1	n-1	1

Número total de passos: $1 + n + (n - 1) \cdot 2 + 1 = 3n$

Análise de complexidade: mínimo de um vetor

Algoritmo: MinimoVetor(S, p, q)

Entrada: vetor S, índices p e q

Saída: índice, entre p e $q-1$, do menor elemento em $S[p..q-1]$

$min = S[p]$
$i = p$
$imin = p$
enquanto $i < q$ **faça**
  se $S[i] < min$ **então**
    $min = S[i]$
    $imin = i$
  $i = i + 1$
retorne $imin$

Análise de complexidade: mínimo de um vetor

Algoritmo: MinimoVetor(S, p, q)

Entrada: vetor S, índices p e q

Saída: índice, entre p e $q-1$, do menor elemento em $S[p..q-1]$

```
1 min = S[p]
2 i = p
3 imin = p
4 enquanto i < q faça
5     se S[i] < min então
6         min = S[i]
7         imin = i
8     i = i + 1
9 retorne imin
```

passo	(1,2,3)	(4)	(5)	(6,7)	(8)	(9)
excuções	3	n	1	2	1	1

Número total de passos (com $n = q - p$): $3 + n \cdot (1 + 2 + 1) + 1 = 4(n + 1)$

Análise de complexidade: Torre de Hanoi

Algoritmo: Hanoi(n, O, D, T)

Entrada: tamanho n do disco a ser movido, torres de origem O, de trabalho T, e de destino D

Saída: passos para mover n discos de O para D usando T

```
se n == 1 então
    mover disco n de O para D
senão
    Hanoi(n − 1, O, T, D)
    mover disco n de O para D
    Hanoi(n − 1, T, D, O)
```

Análise de complexidade: Torre de Hanoi

Algoritmo: Hanoi(n, O, D, T)

Entrada: tamanho n do disco a ser movido, torres de origem O, de trabalho T, e de destino D

Saída: passos para mover n discos de O para D usando T

```
se n == 1 então
    mover disco n de O para D
senão
    Hanoi(n − 1, O, T, D)
    mover disco n de O para D
    Hanoi(n − 1, T, D, O)
```

Número total de passos:

$$T(n) = \begin{cases} 2, & \text{se } n = 1 \\ 2 + 2T(n-1), & \text{caso contrário} \end{cases}$$

Análise de complexidade: Torre de Hanoi

$$T(n) = \begin{cases} 2, & \text{se } n = 1 \\ 2 + 2T(n-1), & \text{caso contrário} \end{cases}$$

$$\begin{aligned} T(n) &= 2 + 2T(n-1) \\ &= 2 + 2 \cdot (2 + 2T(n-2)) = 6 + 4T(n-2) \\ &= 6 + 4 \cdot (2 + 2T(n-3)) = 14 + 8T(n-3) \\ &= 14 + 8 \cdot (2 + 2T(n-4)) = 30 + 16T(n-4) \\ &\vdots \\ &= 2^i T(n-i) + \sum_{j=1}^{i} 2^j \rightarrow T(n) = 2^n + \sum_{j=1}^{i} 2^j \end{aligned}$$

Temos uma quantidade de passos da ordem 2^n

Análise de complexidade: busca em um vetor

Algoritmo: BuscaVetor(S, p, q, x)

Entrada: vetor S, índices p e q, natural x

Saída: índice de x em $S[p..q-1]$, se existir, e q caso contrário

```
i = p
enquanto S[i] ≠ x e i < q faça
    i = i + 1
retorne i
```

- Pior caso: número máximo de operações realizadas para todas as instâncias de mesmo tamanho
  - n + 2
- Melhor caso: número mínimo de operações
  - 3

Notação assintótica

Notação O

- Definição:
  - seja $f : \mathbb{N} \rightarrow \mathbb{R}$ uma função
  - $O(f(n)) = \{g(x) \mid g$ é uma função, $g : \mathbb{N} \rightarrow \mathbb{R}$, e existem duas constantes c e n_0 tais que $g(n) \leq c \cdot f(n)$ para todo $n \geq n_0$ $\}$
- Exemplos:
  - $n + 3 \in O(n)$?
  - $40000n + 380400 \in O(n)$?
  - $n^2 \in O(n)$?
  - $n^2 + n \in O(n^2)$?
  - $g(n) \in O(n^2) \rightarrow g(n) \in O(n^2 + n)$?
  - $n^{1000} \in O(2^n)$?

Notação assintótica

- $n + 3 \in O(n)$? Sim
  - $c = 20, n_0 = 30000$
- $40000n + 380400 \in O(n)$? Sim
- $n^2 \in O(n)$? Não
  - não existe n_0 tal que $n^2 \leq cn$ para todo $n \geq n_0$
- $n^2 + n \in O(n^2)$? Sim
  - $n^2 + n \leq cn^2 \rightarrow n^2 + n \leq c_1 n^2 + c_2 n^2$
- $g(n) \in O(n^2) \rightarrow g(n) \in O(n^2 + n)$? Sim
- $n^{1000} \in O(2^n)$? Sim
  - obs: $2^n \notin O(n^{1000})$

Complexidade de tempo

- Função que determina o número de passos de um algoritmo, no pior caso, tendo como parâmetro o tamanho ou valor da instância, e expressa em notação assintótica
  - complexidade de Fatorial: $O(n)$
  - complexidade de MinimoVetor: $O(n)$
  - complexidade de BuscaVetor: $O(n)$
  - complexidade de Hanoi: $O(2^n)$
- A complexidade expressa em notação assintótica "esconde" as aproximações associadas à nossa definição de passo

Complexidade de tempo

- Dois algoritmos para um mesmo problema só podem ser considerados diferentes, no que se refere ao tempo de execução, se o número de passos que eles realizam são expressos por funções cujos crescimentos diferem por mais de um valor constante
  - se dois algoritmos resolvem o mesmo problema com $3n + 1$ e $3n + 1000$ passos, consideramos os dois iguais, pois os dois tem complexidade $O(n)$
- A complexidade de um algoritmo é determinada pelo passo que é utilizado o maior número de vezes

Complexidade de tempo

- Exemplos de ordens de complexidade:
  - $O(1)$: constante
    - ex: verificar se um número é par ou ímpar
  - $O(\log n)$: logarítmico
    - ex: busca binária em um vetor
  - $O(n)$: linear
    - ex: busca linear em um vetor
  - $O(n^2)$: quadrático
    - ex: soma de matrizes bidimensionais
  - $O(n^k)$: polinomial
    - ...
  - $O(k^n)$: exponencial
    - ex: verificar se duas expressões lógicas são equivalentes por força bruta
  - $O(n!)$: fatorial
    - ex: resolver o problema do caixeiro viajante por força bruta