M2L: Análise de complexidade

Murilo Dantas

PROBLEMAS DE REVISÃO

1. **Cronometrar um algoritmo com diferentes tamanhos de problemas:**
   a. **Pode dar uma ideia geral do comportamento do algoritmo em tempo de execução.**
   b. **Pode dar uma ideia do comportamento do algoritmo em tempo de execução em uma plataforma de hardware específica e em uma plataforma de software específica.**

2. **Instruções de contagem:**
   a. **Fornecem os mesmos dados em diferentes plataformas de hardware e software.**
   b. **Podem demonstrar a impraticabilidade dos algoritmos exponenciais com grandes tamanhos de problema.**

3. **As expressões O(n), O(n^2) e O(k^n) são, respectivamente:**
   a. **Exponencial, linear e quadrática.**
   b. **Linear, quadrática e exponencial.**
   c. **Logarítmica, linear e quadrática.**

4. **De um modo geral, é melhor:**
   a. **Ajustar um algoritmo para reduzir o tempo de execução em alguns segundos.**
   b. **Escolher um algoritmo com a ordem mais baixa de complexidade computacional.**

5. **A função Fibonacci recursiva faz aproximadamente:**
   a. **n^2 chamadas recursivas para problemas de tamanho n grande.**
   b. **2^n chamadas recursivas para problemas de tamanho n grande.**

6. **Dois algoritmos A e B possuem complexidade n^5 e 2^n, respectivamente. Você utilizaria o algoritmo B ao invés do A, em qual caso? Explique.**

7. **Um algoritmo tem complexidade O($3m^3 + 2mn^2 + n^2 + 10m + m^2$). Uma maneira simplificada de representar a complexidade desse algoritmo é:**
   a. **O($m^3 + mn^2$).**
   b. **O(m^3).**
   c. **O(m^2).**
   d. **O(mn^2).**
   e. **O($m^3 + n^2$).**

PROBLEMAS DE AVALIAÇÃO

1. **Uma pesquisa sequencial de uma lista ordenada pode ser interrompida quando o alvo é menor que determinado elemento na lista. Defina uma versão modificada desse algoritmo e indique a complexidade computacional, usando a notação big-O, do desempenho nos casos melhor, pior e médio.**

2. **O método de lista *reverse* inverte os elementos da lista. Defina uma função chamada *reverse* que inverte os elementos no argumento de lista (sem usar o método *reverse*). Tente tornar essa função a mais eficiente possível e indique sua complexidade computacional usando a notação big-O.**

3. **A função *pow* retorna o resultado de elevar um número a determinada potência. Defina uma função *expo* que execute essa tarefa e indique a complexidade computacional usando a notação big-O. O primeiro argumento dessa função é o número e o segundo argumento é o expoente (apenas números não negativos). Você pode usar um laço em sua implementação, mas não use o operador ** do Python ou a função *pow* do Python.**

4. **Uma estratégia alternativa para a função *expo* usa a seguinte definição recursiva:**
   expo(número, expoente)
   = 1, quando expoente = 0.
   = número * expo(número, expoente – 1), quando o expoente é ímpar.
   = (expo(número, expoente / 2))^2, quando o expoente é par.
   Defina uma função recursiva expo que usa essa estratégia e indique sua complexidade computacional usando a notação big-O.

5. **Dada a função abaixo, determine a complexidade:**

```
def tem_duplicacao(array, n):
    for i=0 to n-1:
        val = array[i]

        for j=(i+1) to n-1:
            if array[j] == val:
                return true
    return false
```

8. **Dada a função abaixo, determine a complexidade:**

```
def ache_min(array, n):
    min = array[0]

    for i=0 to n-1:
        if array[i] < min:
            min = array[i]
    return min
```

9. Do fragmento de código abaixo, determine a complexidade:

```
for i=0 to n-1:
    for j=0 to n-1:
        mat[i][j] = 0
        for k=0 to n-1:
            mat[i][j] += A[i][k] * B[k][j]
```

10. Do fragmento de código abaixo, determine a complexidade:

```
for i=0 to n-2:
    for j=i+1 to n-1:
        for k=1 to j-1:
            s=1
```

11. Calcule a complexidade, no pior caso, do fragmento de código abaixo:

```
s = 0
for i=0 to n-2:
    for j=1 to 2*N:
        s = s+1
```

12. Calcule a complexidade, no pior e no melhor caso, dos fragmentos de código abaixo:

```
for i=0 to n-1:
    print(i)
```

```
for i in range(0,n,2):
    print(i)
```

```
for i in range(0,n,2):
    print(i)
    i -= 1
```

13. Calcule a complexidade, no pior caso, do fragmento de código abaixo:

```
for i in range(0,n,2):
    for j in range(n,-1,-1):
        if V[i] < V[j]:
            print(i)
```

14. Escreva um algoritmo que receba valores em um vetor e imprima "ORDENADO" se o vetor estiver em ordem crescente. Qual é a complexidade deste seu algoritmo?

15. Escreva um algoritmo que receba um vetor ordenado e um número extra e insira esse número na sua posição correta no vetor ordenado, deslocando os outros números, se necessário. Qual é a complexidade no melhor e no pior caso deste algoritmo?