Estruturas de Dados e Básicas I - DIM0119

Selan R. dos Santos

DIMAp – Departamento de Informática e Matemática Aplicada
Sala 231, ramal 231, selan@dimap.ufrn.br
UFRN

2018.1

Introdução — Conteúdo

1. Exercícios propostos
2. Plano geral análise algoritmos iterativos
3. Referências

Exercício #1: menor distância

Descrição

- Desenvolver um algoritmo que recebe um arranjo de inteiros como entrada, calcula e retorna a **menor** distância entre dois elementos quaisquer do arranjo.
- Define-se como distância o valor absoluto da diferença entre dois elementos. Por exemplo, dado o vetor $A = (2, 9, -13, 5, 26)$, a menor distância entre elementos do arranjo é 3 ($|2-5| = 3$).
- Determinar a complexidade temporal do algoritmo desenvolvido em função do tamanho do vetor de entrada.
- Qual o parâmetro primário?
- A organização dos dados influencia a complexidade do algoritmo? Isto é, existe situação de melhor ou pior caso para o algoritmo proposto?

Exercício #1: menor distância

Solução e Análise de Complexidade

```
função minDist(A: arranjo de inteiro):inteiro
    var dmin: inteiro ← ∞                                   #maior valor inteiro possível.
    var i, j: inteiro                                        #controladores de laços.
    var n: inteiro ← tam A# c1
    para i ← 0 até n − 1 faça # c2
        para j ← 0 até n − 1 faça # c3
            se i ≠ j e |A[i] − A[j]| < dmin então # c4
                dmin ← |A[i] − A[j]|# c5
    retorna dmin# c6
```

- $T(n) = c_1 + c_2 + n \cdot L + c_6$ e ...
- $L = c_3 + n(c_4 + c_5)$.
- Substituindo: $T(n) = c_1 + c_2 + n(c_3 + n(c_4 + c_5)) + c_6$.
- Agrupando: $T(n) = (c_4 + c_5)n^2 + c_3 n + (c_1 + c_2 + c_6) = n^2$.
- Simplificando: $T(n) = \Theta(n^2)$ ou quadrático.
- É possível melhorar a *complexidade*?

Exercício #2: Filter (1ª versão)
Descrição

- Vamos analisar a primeira versão do algoritmo filter.

```
função filter_v1(V: arranjo de inteiro): inteiro
    var n: inteiro ← tam V                          #recuperar o tamanho do vetor
    var i: inteiro ← 0                              #contador laço
    enquanto i < n faça #c1                         #testar cada elemento do arranjo
        se V[i] ≤ 0 então #c2                       #critério (ser <=0) satisfeito, filtrar...
            para j ← i até n − 2 faça #c3           #deslocar elemts. p/ esquerda
                V[j] ← V[j + 1] #c4                 #copiar o seguinte sobre o atual
            n ← n − 1 #c5                           #diminuir tamanho lógico
        senão                                       #Elemento permanece, processar o próximo
            i ← i + 1 #c6
    retorna n #c7                                   #Retornar o novo tamanho lógico do vetor
```

- Qual é a operação dominante?
- Em que situação a operação dominante é executada? Qual o pior caso?

Exercício #2: Filter (1ª versão)
Análise de Complexidade

```
função Filter_v1(V: arranjo de inteiro): inteiro
    var n: inteiro ← tam V                          #recuperar o tamanho do vetor
    var i: inteiro ← 0                              #contador laço
    enquanto i < n faça #c1                         #testar cada elemento do arranjo
        se V[i] ≤ 0 então #c2                       #critério (ser <=0) satisfeito, filtrar...
            para j ← i até n − 2 faça #c3           #deslocar elemts. p/ esquerda
                V[j] ← V[j + 1] #c4                 #copiar o seguinte sobre o atual
            n ← n − 1 #c5                           #diminuir tamanho lógico
        senão                                       #Elemento permanece, processar o próximo
            i ← i + 1 #c6
    retorna n #c7                                   #Retornar o novo tamanho lógico do vetor
```

- Podem acontecer duas situações, gerando 2 casos a serem analisados.

$$T(n) \leq \begin{cases} n(c_1 + c_2 + L) + c_7, & \text{se } V[i] \leq 0 \\ n(c_1 + c_2 + c_6) + c_7, & \text{se } V[i] > 0 \end{cases}$$

Exercício #2: Filter (1ª versão)
Análise do pior caso

```
enquanto i < n faça #c1
    se V[i] ≤ 0 então #c2
        para j ← i até n − 2 faça #c3
            V[j] ← V[j + 1] #c4
        n ← n − 1 #c5
    senão
        i ← i + 1 #c6
retorna n #c7
```

$T(n)_{\text{pior}} = n(c_1{+}c_2{+}L){+}c_7,\ i{\in}[0;n{-}1] \Rightarrow T(n)_{\text{pior}} = nc_1{+}nc_2{+}nL{+}c_7$

Desenvolvendo $nL{=}((n{-}2){-}i{+}1)(c_3{+}c_4){=}(n{-}1{-}i)(c_3{+}c_4)$ mas é expressão com variável i!

$$\left.\begin{array}{lll} i{=}0, & j{\in}[0;n{-}2], & (n{-}1{-}0)(c_3{+}c_4) \text{ ou } (n{-}1)\cdot(c_3{+}c_4) \\ i{=}1, & j{\in}[1;n{-}2], & (n{-}1{-}1)(c_3{+}c_4) \text{ ou } (n{-}2)\cdot(c_3{+}c_4) \\ i{=}2, & j{\in}[2;n{-}2], & (n{-}1{-}2)(c_3{+}c_4) \text{ ou } (n{-}3)\cdot(c_3{+}c_4) \\ \vdots & \vdots & \\ i{=}n{-}3, & j{\in}[n{-}3;n{-}2], & (n{-}1{-}(n{-}3))(c_3{+}c_4) \text{ ou } (2)\cdot(c_3{+}c_4) \\ i{=}n{-}2, & j{\in}[n{-}2;n{-}2], & (n{-}1{-}(n{-}2))(c_3{+}c_4) \text{ ou } (1)\cdot(c_3{+}c_4) \end{array}\right\} \left.\begin{array}{ll} i{=}0, & (n{-}1)\cdot(c_3{+}c_4) \\ i{=}1, & (n{-}2)\cdot(c_3{+}c_4) \\ i{=}2, & (n{-}3)\cdot(c_3{+}c_4) \\ \vdots & \vdots \\ i{=}n{-}3, & (2)\cdot(c_3{+}c_4) \\ i{=}n{-}2, & (1)\cdot(c_3{+}c_4) \end{array}\right\}$$

$1{\cdot}(c_3{-}$

$(1{+}2{-}$

$\sum_{j=}^{n-}$

$\frac{n(n-1}{2}$

Portai

Exercício #2: Filter (1ª versão)
Análise do pior caso

```
enquanto i < n faça #c1
    se V[i] ≤ 0 então #c2
        para j ← i até n − 2 faça #c3
            V[j] ← V[j + 1] #c4
        n ← n − 1 #c5
    senão
        i ← i + 1 #c6
retorna n #c7
```

$$\begin{array}{lll} T(n)_{\text{pior}} & \leq & nc_1 + nc_2 + nL + c_7, \quad \text{mas } nL = c_8 \cdot \frac{n(n-1)}{2} \\ T(n)_{\text{pior}} & \leq & nc_1 + nc_2 + c_8 \cdot \frac{n(n-1)}{2} + c_7 \\ T(n)_{\text{pior}} & \leq & c_1 n + c_2 n + \frac{c_8}{2} n^2 - \frac{c_8}{2} n + c_7 \\ T(n)_{\text{pior}} & \leq & nc_1 + nc_2 + c_9 n^2 - c_9 n + c_7 \\ T(n)_{\text{pior}} & \leq & c_9 \cdot n^2 + (c_1 + c_2 - c_9) \cdot n + c_7 \leq n^2. \\ T(n)_{\text{pior}} & \leq & n^2 \text{ ou } \textbf{quadrático}. \end{array}$$

Exercício #2: Filter (1ª versão)

Análise do melhor caso

```
enquanto i < n faça # c1
    se V[i] ≤ 0 então # c2
        para j ← i até n − 2 faça # c3
            V[j] ← V[j + 1]  # c4
        n ← n − 1 # c5
    senão
        i ← i + 1 # c6
retorna n # c7
```

$$T(n)_{\text{melhor}} \leq n(c_1 + c_2 + c_6) + c_7$$
$$T(n)_{\text{melhor}} \leq c_9 \cdot n + c_7$$
$$T(n)_{\text{melhor}} \leq n \text{ ou } \textbf{linear}.$$

Exercício #3: Filter (2ª versão)

Descrição

▷ Vamos analisar a **segunda versão** do algoritmo filter.

```
função filter_v2(V: arranjo de inteiro): inteiro
    var n: inteiro ← tam V                          # recuperar o tamanho do vetor
    var slow: inteiro ← 0                           # índice lento, avança eventualmente
    var fast: inteiro ← 0                           # índice rápido, controla o laço
    enquanto fast < n faça # c1                     # testar cada elemento do arranjo
        se V[i] > 0 então # c2                      # critério inclusão (ser >0) satisfeito
            V[slow] ← V[fast]                       # elemento vai pra parte filtrada
            slow ← slow + 1                         # região filtrada cresce
        fast ← fast + 1                             # avanço regular
    retorna slow                                    # Retornar o novo tamanho lógico do vetor
```

▷ Existe pior/melhor casos?

▷ Qual é a operação dominante?

▷ Qual a complexidade?

Exercício #4: Dois vetores

▷ Para os exercícios seguintes, considere dois vetores de inteiros `A` e `B`, ambos com tamanho n:

1) Determine se um elemento inteiro `target` está presente em qualquer um dos dois vetores.
   O que caracteriza o pior caso e qual a sua complexidade?
2) Determine se os vetores tem um elemento em comum.
   O que caracteriza o pior caso e qual a sua complexidade?
3) Determine se existe algum elemento duplicado em `A` e `B`.
   O que caracteriza o pior caso e qual a sua complexidade?

Exercício #5: Transposição de matriz

Descrição

▷ Desenvolver um algoritmo para transpor uma matriz quadrada M. Os parâmetros do algoritmo são a matriz M, de tamanho $n \times n$. Não utilize matriz ou vetor auxiliar na solução.

▷ Determinar a complexidade do algoritmo em função de n.

Exercício #5: Transposição de matriz

Solução e Análise de Complexidade

```
procedimento transpor(M: arranjo de ref inteiro)
    var aux: inteiro                       # ajudar a troca de elementos
    var i, j: inteiro                      # controladores de laço para linha e coluna
    var n: inteiro ← tam M                 # recupera a 1ª dimensão de M
    para i ← 0 até n − 2 faça # c1
        para j ← i + 1 até n − 1 faça # c2
            aux ← M[i, j]  # c3
            M[i, j] ← M[j, i]  # c4
            M[j, i] ← aux  # c5
```

- $T(n) = c_1 + (n - 2 - 0 + 1)L = c_1 + (n - 1)L$.
- Temos que $L = c_2 + [(n - 1) - (i + 1) + 1](c_3 + c_4 + c_5) =$
  $L = c_2 + ((n - 1) - i) \cdot c_6$.

Exercício #5: Transposição de matriz

Análise de Complexidade (cont.)

```
para i ← 0 até n − 2 faça # c1
    para j ← i + 1 até n − 1 faça # c2
        aux ← M[i, j]  # c3
        M[i, j] ← M[j, i]  # c4
        M[j, i] ← aux  # c5
```

$T(n) = c_1 + (n-1)L, \; L = c_2 + ((n-1)-i) \cdot c_6, \; i \in [0; n-2]$

Desenvolvendo $(n-1)L$

$$\left.\begin{array}{lll} i=0, & j\in[1;n-1] & \text{ou } c_2+(n-1)\cdot c_6 \\ i=1, & j\in[2;n-1] & \text{ou } c_2+(n-2)\cdot c_6 \\ i=2, & j\in[3;n-1] & \text{ou } c_2+(n-3)\cdot c_6 \\ \vdots & \vdots & \\ i=n-3, & j\in[n-2;n-1] & \text{ou } c_2+2\cdot c_6 \\ i=n-2, & j\in[n-1;n-1] & \text{ou } c_2+1\cdot c_6 \end{array}\right\}$$

$c_2 + c_2 + \cdots + c_2 = c_2(1+1+1+\cdots+1) = c_2(n-1)$

$1\cdot c_6 + 2\cdot c_6 + 3\cdot c_6 + \cdots + (n-1)\cdot c_6 =$
$(1+2+3+\cdots+n-1)\cdot c_6 = \sum_{j=1}^{n-1} j\cdot c_6 =$
$c_6 \cdot \sum_{j=1}^{n-1} j = c_6 \cdot \frac{n(n-1)}{2}$.

Portanto, $(n-1)L = c_2(n-1) + c_6 \cdot \frac{n(n-1)}{2}$.

Exercício #5: Transposição de matriz

Análise de Complexidade (cont.)

```
para i ← 0 até n − 2 faça # c1
    para j ← i + 1 até n − 1 faça # c2
        aux ← M[i, j]  # c3
        M[i, j] ← M[j, i]  # c4
        M[j, i] ← aux  # c5
```

$$\begin{aligned} T(n) &\leq c_1 + (n-1)L, \quad (n-1)L = c_2 \cdot (n-1) + c_6 \cdot \tfrac{n(n-1)}{2} \\ T(n) &\leq c_1 + c_2 \cdot (n-1) + c_6 \cdot \tfrac{n(n-1)}{2} \\ T(n) &\leq \tfrac{c_6}{2} \cdot n(n-1) + c_2 \cdot n + (c_1 - c_2) \\ T(n) &\leq c_7 \cdot n^2 - c_7 \cdot n + c_2 \cdot n + c_8 \\ T(n) &\leq c_7 \cdot n^2 + c_9 n + c_8 \\ T(n) &\leq n^2 \text{ ou } \textbf{quadrático}. \end{aligned}$$

Exercício #6: Malhação 1000 flexões

Descrição

- Paulo gosta de se exercitar. Desta forma, Paulo deseja subir uma escada com n degraus, fazendo 1000 flexões em cada degrau, para depois descer a escada sem fazer flexões até o ponto inicial de onde ele iniciou a subida.
- Escreva o procedimento `subirEscada1000Flexoes(n)` que representa este processo, sendo `n` o número de degraus.
- Utilize os seguintes procedimentos auxiliares `subirDegrau()`, `descerDegrau()` e `fazerUmaFlexao()`, todos com complexidade constante.
  Precondição: $n > 0$.
- Determinar a complexidade do algoritmo em função de n.

Exercício #6: Malhação 1000 flexões

Solução e Análise de Complexidade

Solução

```
procedimento subirEscada1000Flexoes(n: inteiro)
    var i, j: inteiro                # controladores de laço para escada e flexões
    para i ← 1 até n faça # c1
        subirDegrau() # c2
        para j ← 1 até 1000 faça # c3
            fazerUmaFlexao() # c4
    para i ← n até 1 com passo −1 faça # c5
        descerDegrau() # c6
```

- $T(n) \leq n(c_1 + c_2 + L) + n(c_5 + c_6)$, onde $L = 1000(c_3 + c_4)$.
- $T(n) \leq n(c_1 + c_2 + 1000(c_3 + c_4)) + n(c_5 + c_6)$.
- $T(n) \leq k_0 n + k_1 n \Rightarrow T \in O(n)$ ou **linear**.

Exercício #7: Malhação dobra flexões

Descrição

- Paulo agora quer subir a escada de n degraus dobrando o número de flexões a cada degrau. No primeiro degrau ele faz 1 flexão, depois 2, no próximo 4, no seguinte 8 e assim por diante.
- Escreva o procedimento `subirEscadaFazendoFlexoes(n)` que representa este processo, sendo `n` o número de degraus.
- Utilize os seguintes procedimentos auxiliares `subirDegrau()`, `descerDegrau()` e `fazerUmaFlexao()`, todos com complexidade constante.
  Precondição: $n > 0$.
- Determinar a complexidade do algoritmo em função de n.

Exercício #7: Malhação dobra flexões

Solução e Análise de Complexidade

Solução

```
procedimento subirEscadaFazendoFlexoes(n: inteiro)
    var i, j: inteiro                # controladores de laço para escada e flexões
    var f ← 1: inteiro                                        # número de flexões
    para i ← 1 até n faça # c1
        subirDegrau() # c2
        para j ← 1 até f faça # c3
            fazerUmaFlexao() # c4
        f ← 2 * f # c5
```

- $T(n) \leq n(c_1 + c_2 + L + c_5)$, onde $L = f(c_3 + c_4)$.
- $T(n) \leq c_6 \cdot n + L \cdot n$, onde $c_6 = (c_1 + c_2 + c_5)$, $L = c_7 \cdot f$ e $c_7 = (c_3 + c_4)$, com $i \in [1; n]$ e $j \in [1; f = 2^{i-1}]$.

Exercício #7: Malhação dobra flexões

Análise de Complexidade (cont.)

```
procedimento subirEscadaFazendoFlexoes(n: inteiro)
    var i, j: inteiro                # controladores de laço para escada e flexões
    var f ← 1: inteiro                                        # número de flexões
    para i ← 1 até n faça # c1
        subirDegrau() # c2
        para j ← 1 até f faça # c3
            fazerUmaFlexao() # c4
        f ← 2 * f # c5
```

- $T(n) = c_6 \cdot n + L \cdot n$, onde $L = c_7 \cdot f$, com $i \in [1; n]$ e $j \in [1; f = 2^{i-1}] \Rightarrow j \in [1; f = 2^{n-1}]$.

$$\left.\begin{array}{ll} i=1, & j\in[1;1] \text{ ou } 1\cdot c_7 \\ i=2, & j\in[1;2] \text{ ou } 2\cdot c_7 \\ i=3, & j\in[1;4] \text{ ou } 4\cdot c_7 \\ \vdots & \vdots \\ i=n, & j\in[1;2^{n-1}] \text{ ou } 2^{n-1}\cdot c_7 \end{array}\right\} L\cdot n = 1\cdot c_7 + 2\cdot c_7 + 4\cdot c_7 + \cdots + 2^{n-1}\cdot c_7$$

$$\sum_{j=1}^{n-1} 2^{j-1}\cdot c_7 = c_7\cdot\sum_{j=1}^{n-1} 2^{j-1} = \frac{c_7}{2}\cdot\sum_{j=1}^{n-1} 2^{j} = c_8\cdot\sum_{j=1}^{n-1} 2^{j}.$$

Exercício #7: Malhação dobra flexões

Solução e Análise de Complexidade

```
procedimento subirEscadaFazendoFlexoes(n: inteiro)
    var i, j: inteiro                    # controladores de laço para escada e flexões
    var f ← 1: inteiro                   # número de flexões
    para i ← 1 até n faça # c1
        subirDegrau() # c2
        para j ← 1 até f faça # c3
            fazerUmaFlexao() # c4
        f ← 2 * f # c5
```

- ▷ $T(n){=}c_6{\cdot}n{+}L{\cdot}n$, onde $L{\cdot}n{=}c_8{\cdot}\sum_{j=1}^{n-1} 2^j$, com $n{\in}[1;n]$ e $f{\in}[1;2^{n-1}]$.
- ▷ $L \cdot n + c_8 \cdot 2^0 = (c_8 \cdot \sum_{j=1}^{n-1} 2^j) + c_8 \cdot 2^0$.
- ▷ $L \cdot n + c_8 \cdot 2^0 = (c_8 \cdot \sum_{j=0}^{n-1} 2^j) \Rightarrow L \cdot n = (c_8 \cdot \sum_{j=0}^{n-1} 2^j) - c_8 \cdot 2^0$.
- ▷ Note que, $\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1$.
- ▷ $L \cdot n = c_8 \cdot (2^{n+1} - 1 - 2^n) - c_8 \cdot 2^0$
- ▷ $L \cdot n = c_8 \cdot (2^n - 1) - c_8 = c_8 \cdot 2^n - 2 \cdot c_8 = c_8 \cdot 2^n + c_9$.
- ▷ $T(n) = c_6 \cdot n + c_8 \cdot 2^n + c_9 \Rightarrow \Theta(2^n)$ ou exponencial.

Exercício #8: Unidade em Arranjo

Descrição

Problema: Unicidade em Arranjo de tamanho n

Dado um conjunto de valores previamente armazenados em um vetor A, nas posições $A[0], A[1], \ldots, A[n-1]$, verificar se todos os elementos de A são distintos (i.e. são únicos). Retornar *verdadeiro* em caso afirmativo, ou *falso*, caso contrário.

Desenvolver o algoritmo, calcular a complexidade para o pior caso para o algoritmo proposto.

Exercício #8: Unidade em Arranjo

Solução

Problema: Unicidade em Arranjo de tamanho n

```
função uniqueElements(A arranjo de inteiro): booleano
    var n: inteiro ← tam V                       # recupera o tamanho de A
    var i, j: inteiro                            # controle de laços
    para i ← 0 até n − 2 faça                    # percorrer vetor até antes do final
        para j ← i + 1 até n − 1 faça            # compara com elementos a frente
            se A[i] = A[j] então                 # são iguais?
                retorna falso
    retorna verdadeiro                           # arranjo é único
```


Exercício #8: Unidade em Arranjo

Análise de Complexidade

- ▷ Existem dois tipos de **pior caso**, ou seja, entradas para o qual o algoritmo não finaliza prematuramente:
  1. arranjos sem elementos iguais; e
  2. arranjo cujo os últimos dois elementos são iguais.

$$\begin{aligned} T_{pior}(n) = {} & \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \\ & \sum_{i=0}^{n-2}(n-1-i) = \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = \\ & (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} = \\ & (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in O(n^2). \end{aligned}$$

- ▷ Complexidade de pior caso é quadrática.

Exercício #9: Encontrar o k-ésimo Menor Elemento
Descrição

- Escreva uma função para obter o k-ésimo menor elemento de uma lista sequencial L com n elementos (precondição: $1 \leq k \leq n$). Podem haver elementos repetidos em L.
- Elabore sua função de modo a minimizar a complexidade de pior caso e determine esta complexidade.
- Por fim, descreva a situação correspondente ao pior caso considerado e forneça um exemplo ilustrativo com, pelo menos, $n = 5$ elementos.
- **Sugestão**: Quando encontrar o 1º menor, troque-o com o elemento na 1ª posição de L, quando encontrar o 2º menor, troque-o com o elemento na 2ª posição de L, e assim por diante. Ao final, a solução estará na k-ésima posição do vetor.

Exercício #9: Encontrar o k-ésimo Menor Elemento
Solução

Solução

```
1  função FindKSmallest(L: arranjo de inteiro; k inteiro): inteiro
2      var menor, aux, i, j: inteiro
3      var n: inteiro ← tam L                         # obter o tamanho do arranjo
4      para i ← 0 até k − 1 faça                # realizar busca pelo menor apenas k vezes
5          menor ← i               # na i-ésima busca o elemento i é o (candidato) menor
6          para j ← i + 1 até tam L faça                  # buscar menor no resto da lista
7              se L[j] < L[menor] então                     # achamos um elemento menor?
8                  menor ← j                              # atualizar índice do novo menor
9          aux ← L[i]                            # realizar a operação de troca ou swap
10         L[i] ← L[menor]                                     # mover o menor pra frente
11         L[menor] ← aux                                            # finalizar a troca
12     retorna L[k]                                   # o k-ésimo menor está na posição i
```


Exercício #9: Encontrar o k-ésimo Menor Elemento
Análise de Complexidade

- O algoritmo tem por objetivo selecionar qual o k-ésimo menor elemento em um arranjo unidimensional não ordenado.
- Inicialmente o algoritmo assume que o primeiro elemento é o menor e tenta encontrar outro candidato menor no restante do vetor.
- Se encontrar um novo menor este é trocado com o antigo menor. Desta forma o algoritmo prossegue isolando, a cada passo, os menores elementos no início do vetor, classificados em ordem não decrescente.
- A cada passo a quantidade de elementos a vasculhar diminui em 1 posição. Ao final o algoritmo retorna o elemento na k-ésima posição do vetor, que é o k-ésimo menor elemento.

Exercício #9: Encontrar o k-ésimo Menor Elemento
Análise de Complexidade (cont.)

- O pior caso ocorre quando $k = n$ e o vetor encontra-se classificado em ordem decrescente. O fato de estar em ordem decrescente implica que o algoritmo sempre executará a linha 07. Nesta situação temos que linhas $04 - 11$ é $O(k)$ e linha $06 - 08$ é $O(n)$. Como $k = n$ então o algoritmo é $O(n^2)$. Interessante notar que nesta situação o algoritmo ordena todo vetor.
- Na verdade não há melhor caso visto que o algoritmo sempre precisa percorrer o subvetor para achar o menor elemento, para então realizar a troca; Em outras palavras, independente da organização dos dados, o tempo é sempre proporcional ao quadrado da entrada. Neste caso afirmamos que o algoritmo é $\Theta(n)$.

Desafio de programação

Exercício #10

- Desenvolva um algoritmo (ou programa) que **recebe** como entrada as coordenadas Cartesianas (x, y) do pontos que definem dois segmentos de reta P_1Q_1 e P_2Q_2 e **determina** se os segmentos têm ou não um ponto em comum.
- Faça a análise de complexidade.

Eficiência Temporal de Algoritmos Não-recursivos

Plano Geral para Análise de Algoritmos Não-recursivos

- Decidir sobre o parâmetro n associado a entrada dos dados;
- Identificar a operação básica do algoritmo;
- Determinar as situações de entrada n que correspondem ao pior e melhor casos;
- Montar o somatório (expressão) que representa o número de vezes que a operação básica é executada; e
- Simplificar o somatório com base em fórmulas padrão e regras.

Fórmulas e Regras Úteis Envolvendo Somatórios

- $\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1 \in \Theta(n)$, onde $l \leq u$ são inteiros correspondendo aos limites superior (u) e inferior (l).
- $\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)$.
- $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3} \in \Theta(n^3)$.
- $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1}-1}{a-1}, \forall a \neq 1$
  - ⋆ Em particular, $\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$.
- $\sum (a_i \pm b_i) = \sum a_i \pm \sum b_i$.
- $\sum c a_i = c \sum a_i$.
- $\sum_{i=l}^{u} a_i = \sum_{l}^{m} a_i + \sum_{m+1}^{u} a_i$.
- $\sum_{i=0}^{n} a_i = a_1 + a_2 + \cdots + a_n = \frac{n}{2}(a_1 + a_n)$.
- $\sum_{i=0}^{n} = a_1 + a_1 r + a_1 r^2 + \cdots + a_n r^n = \begin{cases} \frac{a_1(r^n-1)}{r-1} & r > 1 \\ \frac{a_1(1-r^n)}{1-r} & r < 1 \end{cases}$
- Soma de uma PG inifita: $S_\infty = \sum_{n=1}^{\infty} a r^{n-1} = \frac{a_1}{1-r}, \quad -1 < r < 1$.

Referências

J. Szwarcfiter and L. Markenzon
Estruturas de Dados e Seus Algoritmos, 2ª edição, **Cap. 1**.
Editora LTC, 1994.

R. Sedgewick
Algorithms in C, Parts 1-4, 3rd edition. **Cap. 2**
Addision Wesley, 2004.

A. Drozdeck
Data Structures and Algorithms in C++, 2nd edition. **Cap. 2**
Brooks/Cole, Thomsom Learning, 2001.

D. Deharbe
Slides de Aula. **aula 2**
DIMAp, UFRN, 2006.

M. Siqueira
Slides de Aula. **aula 1**
DIMAp, UFRN, 2009.