Estruturas de Dados e Básicas I - IMD0029

Selan R. dos Santos

DIMAp – Departamento de Informática e Matemática Aplicada
Sala 25, ramal 239, selan@dimap.ufrn.br
UFRN

2014.2

Introdução — Conteúdo

Motivação e Objetivos

- **Motivação**
  - Para apresentarmos o conteúdo da disciplina precisamos de um conjunto mínimo de conceitos e definições sobre o objeto de estudo da disciplina: estruturas de dados.
- **Objetivos**
  - Definir formalmente conceitos como algoritmo, problema computacional, tipo abstrato de dados, estruturas de dados, corretude e análise de complexidade.
  - Apresentar, de maneira geral, a ferramenta de análise de complexidade e sua importância.

Algoritmos e problemas computacionais

- O que é um **algoritmo**?
  - *R: É um processo sistemático para a resolução de um problema computacional.*
- O que é um **problema computacional**?

O problema de ordenação

- **Entrada**: uma sequência, $\langle a_1, \ldots, a_n \rangle$, de n objetos que aceitam uma ordenação total (por exemplo, números inteiros).
- **Saída**: uma permutação, $\langle a_{\pi_1}, \ldots, a_{\pi_n} \rangle$, da sequência de entrada tal que $a_{\pi_1} \leq a_{\pi_2} \leq \cdots \leq a_{\pi_n}$.

Algoritmos e problemas computacionais

- Por exemplo, dada a sequência de entrada $\langle 58, 41, 59, 26, 41, 31 \rangle$, para se obter uma ordem não decrescente[1] de elemento a saída correspondente é
  - $\langle 26, 31, 41, 41, 58, 59 \rangle$
- A sequência de entrada $\langle 58, 41, 59, 26, 41, 31 \rangle$ é uma instância do problema da ordenação.
- Um problema computacional é, portanto, uma coleção (em geral, infinita) de instâncias e suas respectivas soluções (ou seja, as saídas).

[1] *Ordem crescente com possível repetição.*

Algoritmos e problemas computacionais

- (Novamente) O que é um **algoritmo**?
  - *R (versão 2): É um processo sistemático para a resolução de um problema computacional que especifica uma sequência de ações executáveis que produz* (quando termina) *uma saída a partir de uma dada instância do problema*.
- Algoritmo: **E**ntrada (dados) $\Rightarrow$ **P**rocessamento $\Rightarrow$ **S**aída (solução)
- Um algoritmo é dito correto se, para qualquer instância dada, ele sempre **termina** e **produz a saída esperada** (ou seja, aquela associada à instância).
- Estudo de algoritmos envolve 2 aspectos básicos: corretude e complexidade.
  - Corretude: o algoritmo está **correto**?
  - Complexidade: o algoritmo é eficiente em termos dos recursos (**tempo de execução** e **uso de memória**) por ele utilizados?

Tipos de dados

- Em linguagens de programação é importante classificar constantes, variáveis, expressões e funções de acordo com certas características que indicam o que denominamos de **tipo de dados**.
- O tipo de dados caracteriza o conjunto de valores:
  - a que uma constante pertence, ou
  - que podem ser assumidos por uma variável ou expressão, ou
  - que podem ser gerados por uma função.
- Tipos de dados simples são grupos de valores indivisíveis, tais como os tipos básicos `int`, `float`, `double`, `char` e `bool` da linguagem C++.
- Os **tipos estruturados** em geral definem uma coleção de valores simples ou um agregado de valores de tipos de dados diferentes.
- A linguagem C++, por exemplo, oferece uma grande variedade de tipos de dados simples e estruturados ou compostos.

Tipos Abstratos de Dados *vs* Estruturas de Dados

Tipo Abstrato de Dados (TAD)

Pode ser definido como um modelo matemático acompanhado das operações definidas sobre o modelo.

- Ex.: o conjunto dos inteiros munido das operações de adição, subtração e multiplicação é um exemplo de um tipo abstrato de dados.

- São exemplos de TADs:
  - Lista
  - Pilhas, Filas, Deques
  - Conjuntos
  - Listas de prioridade
  - Grafos
  - Árvores
- Em geral, o projeto de algoritmos para problemas computacionais complexos utilizam TAD extensivamente.
- A implementação de um algoritmo em uma linguagem de programação específica requer que encontremos alguma forma de representar TADs em termos dos tipos de dados e operadores suportados pela linguagem.

Tipos Abstratos de Dados *vs* Estruturas de Dados (cont.)

- Portanto, uma **estrutura de dados** é uma representação concreta de um TAD, ou seja, uma implementação de um modelo definido por TAD.
- Por exemplo, uma TAD pilha (com operações definidas como push, pop e peek) pode ser implementada por meio de um arranjo ou lista encadeada simples.
- Outro exemplo, uma TAD mapa (associa chaves a valores) pode ser implementada por meio de um tabela de dispersão ou lista encadeada simples.

Estruturas de Dados

- Estruturas de dados caracterizam-se pela disposição e manipulação de seus dados.
  - ⋆ **disposição** está diretamente ligado à maneira usada para organizar os dados na memória — modelo matemático.
  - ⋆ **manipulação** está diretamente ligado à ideia de algoritmos — operações.
- Em geral, os dados devem ser dispostos de forma a tornar eficientes o acesso e a modificação dos mesmos pelos algoritmos.
- Não há uma estrutura de dados melhor do que todas as demais para todos os algoritmos.

Estruturas de Dados (cont.)

- Cada problema a ser modelado reque uma análise para identificar a estrutura de dados mais adequada, de acordo com possíveis restrições de recursos e imposições do problema.
- Por isso é importante conhecer muito bem várias estrutura de dados para poder saber qual delas utilizar com um algoritmo ou propósito específico.
- Pelo exposto até então é possível perceber que os conceitos de algoritmo, *TAD* e estruturas de dados estão intimamente relacionados.

Exemplo: verificar interseção de retângulos

Interseção de retângulos

Denominamos de retângulo xy-alinhado um retângulo R cujos lados são paralelos aos eixos Cartesianos x e y. Tal retângulo é caracterizado por seu canto inferior esquerdo (R_x, R_y), e sua largura R_w e altura R_h.

- **Problema:** Sejam R e S dois retângulos xy-alinhados no plano Cartesiano. Escreva uma função que testa se R e S possui uma interseção não-vazia. Se a interseção for não-vazia, retorne o retângulo formado por sua interseção.

Exemplo: verificar interseção de retângulos

Solução

Interseção de retângulos - *Solução*

Vamos considerar os retângulos dados como sendo $R = ((R_x, R_y), R_w, R_h)$ e $S = ((S_x, S_y), S_w, S_h)$. Observe que os retângulos não se intersectam se $I_x = [R_x, R_x + R_w] \cap [S_x, S_x + S_w] = \emptyset$; similarmente, não há interseção se $I_y = [R_y, R_y + R_h] \cap [S_y, S_y + S_h] = \emptyset$.

Reciprocamente, qualquer ponto $p = (x, y)$ tal que $x \in I_x$ e $y \in I_y$, pertence tanto a R quanto a S. Suponha que $I_x = [a_x, b_x]$ e $I_y = [a_y, b_y]$; então o retângulo desejado é $((a_x, a_y), b_x - a_x, b_y - a_y)$.

Exemplo: verificar interseção de retângulos

Solução (cont.)

```
1  struct Rectangle {
2      int x, y, width, height;
3  }
4  bool isIntersect( const Rectangle &R, const Rectangle &S ) {
5      return R.x <= S.x + S.width  && R.x + R.width >= S.x &&
6             R.y <= S.y + S.height && R.y + R.height >= R.y;
7  }
8  Rectangle intersectRect( const Rectangle &R, const Rectangle &S ) {
9      if ( isIntersect( R, S ) ) {
10         return { max(R.x, S.x), max(R.y,S.y),
11                  min(R.x+R.width,  S.x_S.width)  - max(R.x,S.x),
12                  min(R.y+R.height, S.y_S.height) - max(R.y,S.y) };
13     } else {
14         return { 0, 0, -1, -1 }; // Empty rectangle
15     }
16 }
```

- Variação: dado quatro pontos no plano, como você verificaria se eles constituem os vértices de um retângulo xy-alinhado?

Analisando Algoritmos

Corretude

- *Relembrando*... para poder decidir se uma determinada estrutura é adequada ou não para um dado problema precisamos saber avaliar dois aspectos básicos de algoritmos — quais são ele?
  - ⋆ corretude e complexidade
- OK, mas como podemos nos certificar que um dado algoritmo está correto?
- Uma possível estratégia é executar o algoritmo para todas as instâncias do problema e verificar se ele termina e produz a saída esperada para todas elas
- Quais são as desvantagens desta estratégia?
  - ⋆ Nem sempre é preciso testar todas as instâncias e, atualmente, existem ferramentas de software que automatizam a tarefa de teste

Analisando Algoritmos

Corretude

- Uma outra estratégia é provar matematicamente que o algoritmo esta correto
- Tal prova matemática pode ser realizada com o uso de certas ferramentas, tais como triplas de Hoare, invariantes e indução matemática
- Quando estudarmos algoritmos de ordenação, estudaremos também como provar matematicamente a corretude de certos algoritmos

Analisando Algoritmos
Complexidade

- De uma forma mais genérica devemos identificar critérios pra medir a qualidade de um *software*:
  - Lado do usuário ou cliente:
    - interface
    - robustez
    - compatibilidade
    - desempenho (rapidez)
    - consumo de recursos (ex. memória)
  - Lado do desenvolvedor ou fornecedor:
    - portabilidade
    - clareza
    - reuso

Analisando Algoritmos
Complexidade

- A **análise de complexidade de algoritmos** ou, simplesmente, **análise de complexidade**, é um mecanismo para entender e avaliar um algoritmo em relação aos critérios destacados anteriormente, bem como saber aplicá-los a problemas práticos
- Por que nos preocuparmos com desempenho e quantidade de memória utilizada?
  - Computadores podem ser rápidos, mas não são infinitamente rápidos — algumas aplicações, se implementadas com uma má escolha de algoritmos, podem levar dias para completar sua execução, enquanto que a escolha do algoritmo eficiente pode resolver o problema em alguns milissegundos
  - Memória pode ser barata, mas não é gratuita — considere também as limitações de dispositivos móveis

Analisando Algoritmos
Complexidade – exemplo

- Para se ter uma ideia da influência da eficiência de um algoritmo em termos de tempo computacional, vamos a um exemplo.
- Para resolver o problema de ordenação, é possível usar o algoritmo por inserção ou por intercalação.
- Vamos comparar os dois algoritmos em termos de seus tempos computacionais, determinados em função do número de operações e no tamanho n da entrada de dados a ser processada.
- O algoritmo de inserção realiza aproximadamente $c_1 \cdot n^2$ operações para ordenar uma sequência com n números.
- O algoritmo de ordenação por intercalação realiza aproximadamente $c_2 \cdot n \cdot \log_2 n$ operações para ordenar uma sequência com n números.
- Em ambos os casos, c_1 e c_2 representam constantes que não dependem de n, mas sim dos ambientes de software e hardware em que o algoritmo é implementado e executado.

Analisando Algoritmos
Complexidade – exemplo

- Suponha que executemos o algoritmo de inserção em um computador A e o algoritmo de intercalação em um computador B.
- Suponha que A seja 100 vezes mais rápido do que B.
- Para ser mais exato, suponha que A execute 1 bilhão de operações por segundo e que B execute 10 milhões de operações por segundo.
- Suponha também que o algoritmo de inserção tenha sido implementado por um exímio programador, tal que o código resultante requer $2 \cdot n^2$ (isto é, $c_1 = 2$) operações para ordenar uma sequência de n números.
- Por sua vez, suponha que o algoritmo de ordenação por intercalação tenha sido implementado por um programador iniciante, tal que o código resultante requer $50 \cdot n \cdot \log_2 n$ (isto é, $c_2 = 50$) operações para ordenar uma sequência de n números.

Analisando Algoritmos
Complexidade – exemplo

- Qual é o tempo de execução dos algoritmos para ordenar 1 milhão de elementos, ou seja $n = 10^6$?
- O tempo que o computador A leva para produzir a saída é

$$\frac{2 \cdot (10^6)^2 \text{ instruções}}{10^9 \text{ instruções por segundo}} = 2.000 \text{ segundos (33min20s)}$$

- O tempo que o computador B leva para produzir a saída é

$$\frac{50 \cdot (10^6) \cdot \log_2 10^6 \text{ instruções}}{10^7 \text{ instruções por segundo}} = 100 \text{ segundos (1min40s)}$$

Analisando Algoritmos
Complexidade – exemplo

- Embora A seja 100 vezes mais rápido do que B e o programador do algoritmo de inserção seja bem melhor do que o do algoritmo de ordenação por intercalação, o tempo que A levou para produzir uma saída foi 20 vezes maior do que o tempo de B.
- O que aconteceu?
  - ⋆ Na verdade, o que ocorre é que, para valores suficientemente grandes de n, o valor de
$$c_1 \cdot n^2$$
se torna muito maior do que o valor de
$$c_2 \cdot n \cdot \log_2 n$$
- O mais interessante é que a afirmação acima é verdadeira para **quaisquer** valores (positivos) de c_1 e c_2 — ela não depende de c_1 e c_2.

Analisando Algoritmos
Complexidade – exemplo

- ou seja, o algoritmo de intercalação sempre terá melhor desempenho do que o de inserção, para valores suficientemente grandes de n.
- Isto é independente de quão bem implementamos cada algoritmo e do poder computacional da máquina em que eles são executados.
- O exemplo anterior é bastante simples, mas ilustra alguns elementos fundamentais da análise da complexidade de um algoritmo.
- Um deles são as "fórmulas" que descrevem a quantidade de operações dos algoritmos em função do tamanho, n, da sequência de entrada.
- Em breve aprenderemos, de forma superficial, a derivar tais fórmulas, que é a operação básica de uma análise de complexidade.

Princípios de Análise de Algoritmo
Análise Empírica

- Uma das formas mais simples de avaliar um algoritmo é através da análise empírica: *"rodar" 2 algoritmos e verificar qual o mais rápido!*
- Desafios da análise empírica:
  1. desenvolver uma implementação correta e *completa*.
  2. determinar a natureza dos dados de entrada e de outros fatores que têm influência no experimento.
- Tipicamente temos 3 escolhas básicas de dados:
  - ⋆ **reais**: similar as entradas normais para o algoritmo; realmente mede o custo do programa em uso.
  - ⋆ **randômicos**: gerados aleatoriamente sem se preocupar se são dados reais; testa o algoritmo em si.
  - ⋆ **problemáticos**: dados manipulados para simular situações anômalas; garante que o programa sabe lidar com qualquer entrada.

Princípios de Análise de Algoritmo
Análise Empírica

- É necessário levar em considerações algumas variáveis na hora de comparar algoritmos empiricamente: máquinas, compiladores, sistemas e entradas problemáticas.
- Um dos possíveis problemas em se comparar algoritmos empiricamente é que uma implementação pode ter sido realizada com mais cuidado (i.e. otimização) do que a outra.
- Assim, em alguns casos a análise matemática é necessária:
  - se a análise experimental começar a consumir uma quantidade significando de tempo então é o caso de realizar análise matemática.
  - é necessário alguma indicação de eficiência antes de qualquer investimento de desenvolvimento.
- Veremos como conduzir uma análise empírica de algoritmos em aula posterior.

Princípios de Análise de Algoritmo
Análise Matemática – motivação

- É uma forma de avaliar um algoritmo que pode ser mais informativa e menos cara de se realizar.
- Razões para realizar análise matemática:
  - comparação de diferentes algoritmos para a mesma tarefa.
  - previsão de performance em um novo ambiente.
  - configurar valores de parâmetros para algoritmos.

Princípios de Análise de Algoritmo
Análise Matemática – princípios

- A maioria dos algoritmos possui um parâmetro primário n, que afeta o tempo de execução significativamente. Normalmente n é diretamente proporcional ao tamanho dos dados a serem processados.
- O objetivo da análise matemática é expressar a necessidade de recursos de um programa (ex. tempo de execução) em termos de n, usando expressões algébricas mais simples possíveis mas que são precisas para valores elevados de n.
- A ideia é oferecer uma análise independente da máquina, compilador ou sistema.

Princípios de Análise de Algoritmo
Análise Matemática – princípios

- Algumas simplificações são necessárias para facilitar o processo de análise:
  - quantidades de dados manipulados pelos algoritmos será suficientemente grande $\Rightarrow$ avaliação do **comportamento assintótico** (i.e. no limite).
  - não serão consideradas constantes aditivas ou multiplicativas na expressão matemática obtida.
  - termos de menor grau da expressão são desprezados.
- A análise de um algoritmo leva em consideração:
  - um algoritmo pode ser dividido em etapas elementares ou passos.
  - cada passo envolve um número fixo de operações básicas cujos tempos de execução são considerados constantes.
  - a operação básica de maior frequência é a operação dominante.
  - devido a simplificação mencionada anteriormente o número de passos do algoritmo será o número de execuções da operação dominante.

Princípios de Análise de Algoritmo

Exemplo 1

(*Abstração de) Algoritmo para acessar um elemento em um vetor

```
1 função acesso(V: arranjo de inteiro; idx: inteiro): inteiro
2     var N: inteiro ← tam V                          # recuperar o tamanho do vetor
3     se idx ≥ N ou idx < 0 então    # evitar acesso além dos limites do arranjo
4         erro ("Acesso fora dos limites do vetor!")  # imprime msg e aborta programa
5     senão
6         retorna V[idx]
7     fim
8 fim
```

- ▷ O tempo T de execução é calculado com base nos valores idx e N:
  - se $idx \geq N$ ou $idx < 0$, $T = 1$ comparação + chamada de erro
  - senão, $T = 1$ comparação + 1 acesso ao vetor V + 1 atribuição (implícita)
- ▷ Note que o tempo de execução é limitado por um número constante de operações, independente do tamanho da entrada
- ▷ Portanto a complexidade é dita **constante**

Princípios de Análise de Algoritmo

Exemplo 1

Função em C++ para acessar um elemento em um vetor

```
1 int acesso( int V[], int idx, int N ) {
2     if ( idx >= N || idx < 0 ) {  // Evitar acesso alem do vetor.
3         cerr << "\n>>> Acesso fora dos limites!\n";
4         exit( EXIT_FAILURE );
5     } else {
6         return V[ idx ];
7     }
8 }
```

- ▷ O tempo T de execução é calculado com base nos valores idx e N:
  - se $idx \geq N$ ou $idx < 0$, $T = 1$ comparação + chamada de erro
  - senão, $T = 1$ comparação + 1 acesso ao vetor V + 1 atribuição (implícita)
- ▷ Note que o tempo de execução é limitado por um número constante de operações, independente do tamanho da entrada
- ▷ Portanto a complexidade é dita **constante**

Princípios de Análise de Algoritmo

Exemplo 2

Algoritmo para achar o máximo elemento de um vetor

```
1:  função máximo(V: arranjo de inteiro): inteiro
2:      var N: inteiro ← tam V                                  # recuperar o tamanho do vetor
3:      var i: inteiro                                          # controlador do laço
4:      var max: inteiro ← ∅  # armazena maior elemento da i-ésima iteração; inicializado com "indefinido"
5:      se N = 0 então # c1
6:          escreva ("Função chamada com arranjo vazio!")
7:      senão
8:          max ← V[0] # c2, assumindo que o 1º elemento é o máximo.
9:          para i ← 1 até N − 1 faça # c3, percorrer os restantes n − 1 elementos do vetor.
10:             se V[i] > max então # c4, verificar se elemento atual é o maior.
11:                 max ← V[i]# c5
12:     retorna max # c6
```

- ▷ O tempo T de execução seria: $T \leq (c_1 + c_2) + (n-1)(c_3 + c_4 + c_5) + c_6 \Rightarrow T \leq n(c_3 + c_4 + c_5) + (c_1 + c_2 - c_3 - c_4 - c_5 + c_6)$.
- ▷ $T \leq c \cdot n$, onde c é uma constante que depende do sistema.
- ▷ Portanto a complexidade do algoritmo `máximo` é **linear**.

Princípios de Análise de Algoritmo

Exemplo 3: análise de um programa

- ▷ Faça o "teste de mesa" para o seguinte trecho de um programa em C:

```
1 for( i = 0; i < n; i++ ) {
2     for( j = 1, sum = a[0]; j <= i; j++ )
3         sum += a[j];
4     cout << "XXX from 0 through " << i << " is " << sum << endl;
5 }
```

- ▷ O que ele faz?
  - ⋆ *imprime a soma de todas os subvetores iniciados na posição 0*
- ▷ Analisando o programa:
  - ⋆ antes do laço externo (linha 1) ser executado o `i` é inicializado.
  - ⋆ o laço externo é acionado n vezes e em cada iteração são executados: o laço interno (linha 2), um comando de impressão, atribuição para `i` e inicializações para `j` e `sum`.
  - ⋆ o laço mais interno é executado i vezes para cada $i \in \{1, \ldots, n-1\}$ com duas atribuições em cada iteração: `sum` e `j`.
  - ⋆ portanto, temos $T \leq 1 + 3n + \sum_{i=1}^{n-1} 2i = 1 + 3n + 2(1 + 2 + \cdots + n - 1) = 1 + 3n + n(n-1) = O(n) + O(n^2) = O(n^2)$.
  - ⋆ assim temos um programa com complexidade **quadrática**.

Princípios de Análise de Algoritmo
Exemplo 4

- Nem sempre a análise é trivial, pois o número de iterações de laços depende dos dados de entrada.
- Considere o seguinte problema: Dados um vetor `A` de tamanho n, qual o comprimento do maior subvetor ordenado? Ex. $(1, 8, 1, 2, 5, 0, 11, 12)$.
- Veja o programa:

```
1 int n = 8, A[]={ 1, 8, 1, 2, 5, 0, 11, 12 };
2 int i=0, k=0, length=0, idxS=0, idxE=0;
3 for ( i=0, length=1; i < n-1; i++ ) { // gerar todas subseq. possíveis.
4     for ( idxS=idxE=k=i; k < n-1 && A[k]<A[k+1]; k++, idxE++ );
5     // Comp. da maior subseq. (até aqui) é < que a subseq. atual?
6     if ( length < (idxEnd - idxStart + 1) )
7         length = idxEnd - idxStart + 1; // Atualizar maior subseq.
8 }
9 cout << "Comp. do maior subvetor ordenado: " << length << endl;
```

- Se `A` estiver em ordem decrescente o laço externo é executado $n-1$ vezes, mas em cada iteração o laço interno (linha 5) é executado apenas 1 vez $\Rightarrow O(n)$ ou **linear**.
- O algoritmo é menos eficiente se `A` estiver em ordem crescente; **por que?**
- R: laço externo é executado $n-1$ vezes, e o laço interno é executado i vezes para cada $i \in \{n-1, \ldots, 1\} \Rightarrow O(n^2)$ ou **quadrático**.

Princípios de Análise de Algoritmo
Exercício #1

- 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.

```
1 void transpor( int M[ SIZE ][ SIZE ], int s ) {
2     int aux, i, j;
3
4     for ( i = 0; i < s-1; i++ )
5         for ( j = i+1; j < s; j++ ) {
6             printf ("> Trocando %2d com %2d\n", M[i][j], M[j][i]);
7             aux = M[ i ][ j ];
8             M[ i ][ j ] = M[ j ][ i ];
9             M[ j ][ i ] = aux;
10        }
11 }
```

- $T = (n-1)(c_1 + L)$, onde $L = (n-i)(c_2 + c_3 + c_4 + c_5)$. Desenvolvendo teremos: $T = k_0 n^2 + k_1 n + k_3 \Rightarrow O(n^2)$.

Princípios de Análise de Algoritmo
Exercício #1

- 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.

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

- $T = (n-1)(c_1 + L)$, onde $L = (n-i)(c_2 + c_3 + c_4 + c_5)$. Desenvolvendo teremos: $T = k_0 n^2 + k_1 n + k_3 \Rightarrow O(n^2)$.

Princípios de Análise de Algoritmo
Exercício #2

- Paulo Cintura 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()` que representa este processo. Utilize os seguintes procedimentos auxiliares `subirDegrau()`, `descerDegrau()` e `fazerUmaFlexao()`, todos com complexidade constante.
  Pré-condição: $n > 0$.
- Determinar a complexidade do algoritmo em função de n.

Solução

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

- $T = (n)(c_1 + c_2 + L) + (n)(c_5 + c_6)$, onde $L = (1000)(c_3 + c_4)$. Desenvolvendo teremos: $T = k_0 n + k_1 n \Rightarrow O(n)$.

Algoritmo para somar os elementos de uma lista

```
1 função soma(L: arranjo de inteiro): inteiro
2     var N: inteiro ← tam L                    #recuperar o tamanho do vetor
3     var resposta: inteiro                     #guarda resultado da soma
4     se N = 0 então # c1
5         resposta ← 0 # c2
6     senão
7         resposta ← L[0]+soma(subVetor(L, 1, N − 1)) # c3
8     retorna resposta # c4
```

- Cada chamada recursiva da soma decrementa o tamanho da lista. Exceção quando a lista é zero (caso base da recursão).
- Se n é o tamanho inicial, então o número total de chamadas será $n + 1$.
- O custo de cada chamada é: $c_1 + c_2 + c_4$ (última chamada) e $(c_1 + c_3 + c_4)$ (demais chamadas).
- O tempo total $T = n \cdot (c_1 + c_3 + c_4) + c_1 + c_2 + c_4$, ou seja, $T \leq n \cdot c$, onde c é constante $\Rightarrow$ a complexidade é dita **linear**.

- Para avaliar a complexidade em relação a memória necessária, é preciso compreender como a memória de microprocessadores é organizada.
- Cada chamada de função ocupa um espaço na pilha de execução, onde é reservado espaço para variáveis locais e parâmetros da função chamada.
- No caso do exemplo existem $n + 1$ chamadas de função. Portanto o consumo de memória é o somatório do comprimento das listas

$$n + (n-1) + (n-2) + \cdots + 1 + 0 =$$
$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \leq c \cdot n^2$$

- Portanto a **complexidade espacial** é uma função quadrática da entrada n.

- Escreva uma função para obter o k-ésimo menor elemento de uma lista sequencial L com n elementos (pré-condição: $1 \leq k \leq n$). 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.

Desafio de programação

Exercício #4

- 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.

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.