Algoritmos – Parte 2

1. Introdução

A primeira parte deste tópico foi abordada na realização de janeiro de 2020 (disponível em https://impa.br/ensino/programas-de-formacao/linha-do-tempo-dos-cursos/papmem-janeiro-de-2020/

Naquela realização, tratamos dos seguintes assuntos:

- Algoritmos
- Fluxogramas
- Pseudo-códigos
- Progamação em Python

Também falamos rapidamente sobre complexidade computacional, que abordaremos com mais profundidade agora.

2. Complexidade Computacional

A complexidade computacional de um algoritmo é uma medida abstrata de seu tempo de execução. O tempo real depende do computador, da linguagem de programação, etc. A complexidade computacional considera que cada instrução requer tempo unitário e exprime o número de instruções a menos de uma constante multiplicativa.

A complexidade computacional é usualmente expressa na forma $O(f(n))$, que significa que, no pior caso, o número de instruções executadas é menor ou igual a $kf(n)$, onde k é uma constante.

Em geral, f é escolhida como uma função simples (uma função potência, por exemplo). Algumas medidas usuais de complexidade são: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, $O(n!)$.

Uma dúvida que poderia haver neste ponto é se é necessário se preocupar com a complexidade computacional. Afinal, o poder computacional dos computadores aumenta rapidamente; assim, talvez possamos compensar o desempenho de um algoritmo simplesmente escolhendo uma máquina mais rápida. O exemplo a seguir mostra que não é bem assim...

Exemplo: Suponha que, com os computadores atuais, seja possível resolver, em um tempo aceitável, um certo problema para situações de tamanho $N = 100$. Com um computador duas vezes mais rápido, qual é o novo tamanho N' das situações que podem ser resolvidas se a complexidade do algoritmo é:

a) $O(n)$?
b) $O(n^2)$?
c) $O(2^n)$?

Solução: Como a velocidade de processamento do computador dobrou, no mesmo tempo poderemos processar situações cujo tempo de execução seja o dobro do tempo anterior.

a) Se a complexidade é $O(n)$, então o novo tamanho de problema N' que pode ser executado em um tempo aceitável é tal que $kN' = 2\,kN$, ou seja, $N' = 2N = 200$. Portanto, o tamanho da situação que pode ser executada dobra quando a velocidade de processamento dobra.

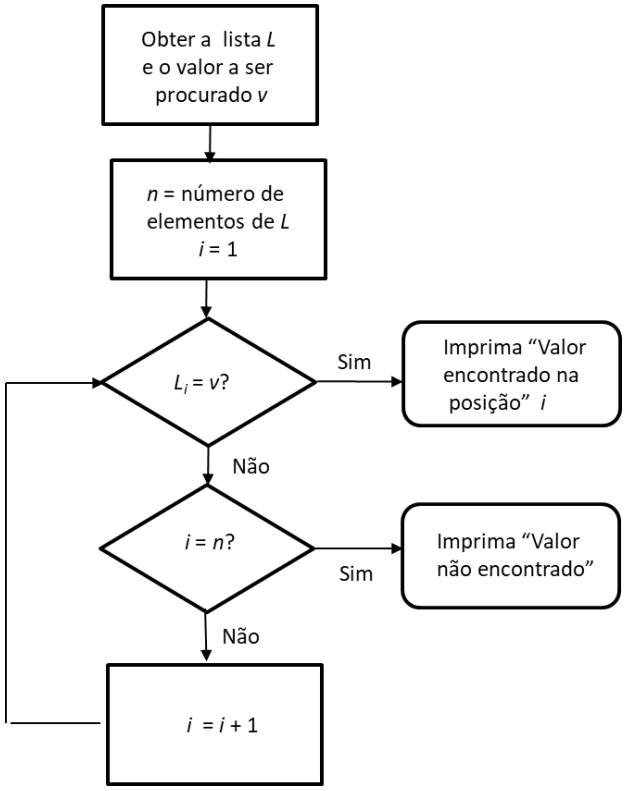
b) Se a complexidade é $O(n^2)$, então o novo tamanho de problema N' que pode ser executado em um tempo aceitável é tal que $kN'^2 = 2\,kN^2$, ou seja, $N' = \sqrt{2}\,N \approx 141$. Portanto, o tamanho da situação que pode ser executada aumenta cerca de 40% quando a velocidade de processamento dobra.

c) Se a complexidade é $O(2^n)$, então o novo tamanho de problema N' que pode ser executado em um tempo aceitável é tal que $k\,2^{N'} = 2\,k\,2^N$, ou seja, $2^{N'} = 2^{N+1}$, o que implica em $N'=N+1 = 101$. Portanto, o tamanho da situação que pode ser executada aumenta de apenas uma unidade quando a velocidade de processamento dobra.

O exemplo mostra que algoritmos que têm complexidade computacional da forma exponencial não são capazes de tirar partido, de modo eficiente, o aumento de velocidade de processamento dos computadores. Por este motivo, de modo geral, buscamos algoritmos de complexidade polinomial, ou seja, tenham complexidade da forma $O(n^p)$, onde, de preferência, p seja pequeno.

3. Procura de um valor em uma lista

Para ilustrar algoritmos de diferentes complexidades para uma situação, vamos considerar o problema de procurar um valor em uma lista. A ideia básica do primeiro algoritmo que vamos examinar é percorrer a lista, elemento por elemento, comparando-o com o valor procurado. Este algoritmo pode ser representado pelo fluxograma abaixo:



Conforme visto no curso de janeiro de 2020, embora fluxogramas sejam úteis para um primeiro contato com algoritmos, é mais conveniente, para representar um algoritmo que vai ser implementado em uma linguagem de programação, utilizar um pseudo-código, que é uma

descrição do algoritmo em Português, mas utilizando estruturas de controle presentes nas modernas linguagens de programação.

O pseudo-código a seguir descreve o algoritmo de procura em uma lista

```
Dados: lista L e valor a ser buscado v
n = número de elementos em L
i = 1
não_encontrei = Verdadeiro
enquanto i ≤ n  e nao_encontrei:
        se L_i = v:
                não_encontrei = Falso
                imprima (encontrado na posição i)
        i = i + 1
se nao_encontrei:
        imprima (“não encontrado”)
```

A partir do pseudo-código é relativamente simples escrever um programa, em alguma linguagem de programação, que implemente o algoritmo. Em Python, um possível programa é o dado abaixo.

```
L = [2, 7, 3, 5]
v = int(input("Valor a ser procurado: "))
i = 0
n = len(L)
nao_encontrei = True
while i < n and nao_encontrei:
   if (L[i] == v):
      print("Encontrado na posição: " + str(i))
      nao_encontrei = False
   i = i + 1

if  nao_encontrei:
   print("Não encontrado")
```

Se desejar, você pode executar o programa online em diversos endereços. Dois deles são https://www.onlinegdb.com/online_python_compiler e https://www.tutorialspoint.com/execute_python_online.php

Para determinar a complexidade computacional do algoritmo, observamos que ela é determinada pelo trabalho realizado no laço determinado pelo comando “enquanto”. No pior caso (que ocorre quando precisamos percorrer a lista toda para concluir que o elemento procurado não está na lista), as instruções dentro do laço são executadas uma vez para cada elemento da lista. Como há um número finito e fixo dessas instruções, concluímos que o algoritmo tem complexidade $O(n)$.

Se a lista na qual devemos buscar o valor não tem qualquer estrutura, não há nada melhor a ser feito: verificar se um elemento está na lista requer examinar cada elemento da lista, levando a um

algoritmo $O(n)$. Mas esta não é a nossa experiência, por exemplo, ao procurar uma palavra em um dicionário. Seria inviável procurar uma palavra em um dicionário como o "Aurélio", que tem cerca de 285.000 verbetes se precisássemos percorrer as palavras uma a uma. O que permite que possamos fazer uma busca mais eficiente é o fato de que as palavras em um dicionário estão em ordem alafabética. Como veremos a seguir, a procura em uma lista ordenada pode ser feita de modo muito mais eficiente.

4. Procura de um valor em uma lista ordenada

Neste caso, podemos localizar o valor (ou constatar que ele não está na lista) usando uma estratégia denominada *procura* (ou *busca*) *binária*. O procedimento tem este nome porque, em cada passo, reduzimos a porção da lista a ser processada ao meio. A ideia básica do algoritmo é a seguinte:

- Algoritmo de **procura (ou busca) binária**
  - Comparar o valor procurado v com o elemento central da lista L_m

    … … … … … … … … L_m … … … … … … … …
  - Se $v = L_m$, ótimo: encontrado!
  - Se $v < L_m$, o valor procurado só pode estar na metade **esquerda** da lista; repetir o processo para essa metade.

    … … … … … … … L_m … … … … … … … …
  - Se $v > L_m$, o valor procurado só pode estar na metade **direita** da lista; repetir o processo para essa metade.

    … … … … … … … … L_m … … … … … … … …

A partir dessa ideia, podemos elaborar o pseudo-código que se segue.

```
Dados: lista L, valor v
min = 1, max = número de elementos de L
nao_encontrei = Verdadeiro
enquanto max ≥ min e nao_encontrei:
        m = (min + max)/2
        se v = L_m:
                nao_encontrei = Falso
                imprima (encontrado na posição m)
        senão,  se v < L_m :
                max = m – 1
        senão:
                min = m + 1
se nao_encontrei:
        imprima (não encontrado)
```

Para determinar a complexidade computacional do algoritmo, observamos que, mais uma vez, ela é determinada pelo trabalho realizado no laço determinado pelo comando "enquanto". No caso da busca binária, cada vez que é feita uma comparação, metade da lista é eliminada. Deste modo, o número x de vezes que o procedimento é realizado é tal que $2^x = n$, ou seja, $x = \log_2 n$, onde n é o tamanho da lista. Portanto, o algoritmo de busca binária tem complexidade $O(\log n)$.

Voltando à busca de uma palavra em um dicionário com 285.000 verbetes, utilizando a busca binária é preciso comparar a palavra com apenas $\log_2 285.000 = 18$ palavras!

Concluímos, assim, que o fato de uma lista estar ordenada permite melhorar dramaticamente o desempenho do processo de buscar um elemento nessa lista.

Uma consequência disto é a importância do problema de colocar os elementos de uma lista em ordem. Na realização de janeiro de 2020, vimos um algoritmo simples para esta tarefa (o algoritmo "bubble-sort"), cuja complexidade é $O(n^2)$. Em uma outra realização do PAPMEM, veremos como podemos melhorar este desempenho.