Estruturas de Dados

Victor Machado da Silva, MSc
victor.silva@professores.ibmec.edu.br

ibmec

Índice

Apresentação do curso

Apresentação do curso

- Contato: victor.silva@professores.ibmec.edu.br
- Grupo no Whatsapp: https://chat.whatsapp.com/IqnSl5PTiVBI1WNwMpXxSS
- Material: www.victor0machado.github.io

Apresentação do curso

- Avaliação
  - Proporção
    - AC (20%): atividades em sala
    - AP1 (40%): projeto + prova objetiva (sem consulta)
    - AP2 (40%): projeto + prova objetiva (sem consulta)
  - Detalhes das entregas
    - Atividades da AC individuais ou em dupla
    - Projetos da AP1 e AP2 em grupos, mínimo 2 e máximo 3 pessoas

Algoritmos

Introdução

Um algoritmo é um processo sistemático para a resolução de um problema. O desenvolvimento de algoritmos é particularmente importante para problemas a serem solucionados em um computador, pela própria natureza do instrumento utilizado.

Um algoritmo computa uma saída, o resultado do problema, a partir de uma entrada, as informações inicialmente conhecidas e que permitem encontrar a solução do problema. Durante o processo de computação o algoritmo manipula dados, gerados a partir da sua entrada.

O estudo de estruturas de dados não pode ser desvinculado de seus aspectos algorítmicos. A escolha correta da estrutura adequada a cada caso depende diretamente do conhecimento de algoritmos para manipular a estrutura de maneira eficiente.

Apresentação dos algoritmos

As convenções seguintes serão utilizadas com respeito à linguagem:

- O início e o final de cada bloco são determinados por indentação, isto é, pela posição da margem esquerda. Se uma certa linha do algoritmo inicia um bloco, ele se estende até a última linha seguinte, cuja margem esquerda se localiza mais à direita do que a primeira do bloco;
- A declaração de atribuição é indicada pelo símbolo :=;
- As declarações seguintes são empregadas com significado semelhante ao usual:

```
se... então
se... então... senão
enquanto... faça
para... faça
pare
```

- Variáveis simples, vetores, matrizes e registro são considerados como tradicionalmente em linguagens de programação. Os elementos de vetores e matrizes são identificados por índices entre colchetes.

```
para i := 1, ..., |__n/2__|
    temp := S[i]
    S[i] := S[n - i + 1]
    S[n - i + 1] := temp
```

Aplicações

- Escreva os algoritmos, em pseudocódigo, para os problemas abaixo:
  - https://br.spoj.com/problems/TOMADA13/
  - https://br.spoj.com/problems/METEORO/
  - https://br.spoj.com/problems/JDESAF12/
  - https://br.spoj.com/problems/CARTAS14/
  - https://br.spoj.com/problems/ENCOTEL/

Recursividade

Um tipo especial de procedimento será utilizado, algumas vezes, ao longo do curso. É aquele que contém, em sua descrição, uma ou mais chamadas a si mesmo. Um procedimento dessa natureza é denominado **recursivo**.

Naturalmente, todo procedimento, recursivo ou não, deve possuir pelo menos uma chamada proveniente de um local exterior a ele. Essa chamada é denominada **externa**. Um procedimento não recursivo é, pois, aquele em que todas as chamadas são externas.

O exemplo clássico mais simples de recursividade é o cálculo do fatorial de um inteiro $n \geq 0$:

```
função fat(i)
    fat(i) := se i <= 1 então 1 senão i * fat(i - 1)
```

```
fat[0] := 1
para j := 1, ..., n faça
    fat[j] := j * fat[j - 1]
```

Aplicações

- Escreva os algoritmos, em pseudocódigo, para os problemas abaixo:
  - https://br.spoj.com/problems/F91/
  - https://br.spoj.com/problems/RUMO9S/
  - https://br.spoj.com/problems/PARIDADE/

Recursividade

Um exemplo conhecido, onde a solução recursiva é comum e extremamente mais simples que a solução não-recursiva, é o do Problema da Torre de Hanói.

Recursividade

A solução do problema é descrita a seguir. Para $n > 1$, o pino-trabalho deve ser utilizado como área de armazenamento temporário. O raciocínio utilizado para resolver o problema é semelhante ao de uma prova matemática por indução. Suponha que se saiba como resolver o problema até $n - 1$ discos, $n > 1$, de forma recursiva. A extensão para n discos pode ser obtida pela realização dos seguintes passos:

- Resolver o problema da Torre de Hanói para os $n - 1$ discos do topo do pino-origem A, supondo que o pino-destino seja C e o trabalho seja B;
- Mover o n-ésimo pino (maior de todos) de A para B;
- Resolver o problema da Torre de Hanói para os $n - 1$ discos localizados no pino C, suposto origem, considerando os pinos A e B como trabalho e destino, respectivamente.

```
procedimento hanoi(n, A, B, C)
se n > 0 então
    hanoi(n - 1, A, C, B)
    mover o disco do topo de A para B
    hanoi(n - 1, C, B, A)
```

Complexidade de algoritmos

Introdução

Conforme já mencionado, uma característica muito importante de qualquer algoritmo é o seu tempo de execução. Naturalmente, é possível determiná-lo através de métodos empíricos, isto é, obter o tempo de execução através da execução propriamente dita do algoritmo, considerando-se entradas diversas.

Ao contrário do método empírico, o método analítico visa aferir o tempo de execução de forma independente do computador utilizado, da linguagem e dos compiladores empregados e das condições locais de processamento.

Introdução

As seguintes simplificações serão introduzidas para o modelo proposto:

- Suponha que a quantidade de dados a serem manipulados pelo algoritmo seja suficientemente grande. Somente o comportamento assintótico será avaliado.
- Não serão consideradas constantes aditivas ou multiplicativas na expressão matemática obtida. Isto é, a expressão matemática obtida será válida, a menos de tais constantes.

O processo de execução de um algoritmo pode ser dividido em etapas elementares, denominadas *passos*. Cada passo consiste na execução de um número fixo de operações básicas cujos tempos de execução são considerados constantes.

Cálculo de passos

São considerados passos:

- Operações aritméticas, relacionais e lógicas
- Atribuições
- Acesso a elementos em vetores e matrizes
- Acesso a campos de uma estrutura (struct)
- Obtenção do endereço de uma variável (incluindo declarações de variáveis)
- Alteração/obtenção de conteúdo através de ponteiros
- Retorno de valores
- Instruções de alocação de memória
- Chamada de procedimento, função, método, etc.

Cálculo de passos

```
func main() int {
    x, y, media float64

    x = 1
    y = 2

    media = (x + y) / 2

    return 0
}
```

```
func main() int {
    x := 1
    y := 2

    media := (x + y) / 2

    return 0
}
```

```
func main() int {
    media float64

    media = (1 + 2) / 2

    return 0
}
```

Cálculo de passos

```
func main() int {
    x, y, media float64

    x = 1
    y = 2

    media = (x + y) / 2

    return 0
}
```

9

```
func main() int {
    x := 1
    y := 2

    media := (x + y) / 2

    return 0
}
```

9

```
func main() int {
    media float64

    media = (1 + 2) / 2

    return 0
}
```

5

Cálculo de passos

Nem sempre mudar ou melhorar o código vai causar diferenças de desempenho no algoritmo! Algumas podem simplesmente atrapalhar a legibilidade ou até provocar defeitos no software.

Prática: https://br.spoj.com/problems/JBUSCA12/

- Submeta um programa ao JBUSCA12 do SPOJ e avalie o tempo de execução
- Traga algumas melhorias para o programa e submeta novamente
- As melhorias reduziram o tempo de execução?

Cálculo de passos

Alguns algoritmos possuem um número variável de passos. Considere, por exemplo, um programa que calcule o n-ésimo número da série de Fibonacci:

```
func fibo(n int) int {
    penultimo, ultimo, proximo, i int

    i = 1
    penultimo = 1
    ultimo = 1

    for i < n {
        proximo = penultimo + ultimo
        penultimo = ultimo
        ultimo = proximo
        i++
    }

    return penultimo
}
```

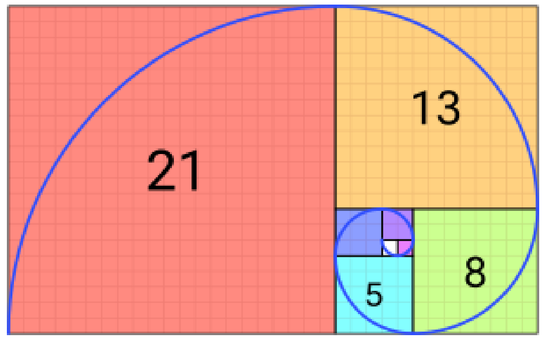
4

3

1...n

6 ——————→ (n – 1) vezes!

1



8 + n + (n – 1) * 6
7n + 2

Cálculo de passos

Com base nesse exemplo, podemos dizer que o custo de execução de um algoritmo (em termos de números de passos) depende, principalmente, do tamanho da entrada dos dados.

Logo, é comum considerar o tempo de execução de um programa como uma **função do tamanho da entrada**.

```
func soma(v []int, n int) int {
    soma := 0

    for i := 0; i < n; i++ {
        soma += v[i]

    return soma
}
```

2

2 / 0...n / 2 3 → Esses passos são executados n vezes

1

$$5 + (n + 1) + n * 5$$
$$6n + 6$$

Cálculo de passos

Com base nesse exemplo, podemos dizer que o custo de execução de um algoritmo (em termos de números de passos) depende, principalmente, do tamanho da entrada dos dados.

Logo, é comum considerar o tempo de execução de um programa como uma **função do tamanho da entrada**.

```
func conta() {
    i := 0

    for i < 30 {
        i++
}
```

2

0...30

2

$$2 + (30 + 1) + 30 * 2$$

$$93$$

Cálculo de passos

Com base nesse exemplo, podemos dizer que o custo de execução de um algoritmo (em termos de números de passos) depende, principalmente, do tamanho da entrada dos dados.

Logo, é comum considerar o tempo de execução de um programa como uma **função do tamanho da entrada**.

```
func busca(v []int, n int, k int) int {
    i := 0

    for i < n {
        if v[i] == k {
            return i
        }
        i++
    }

    return -1
}
```

2

0...n
2
1

2

1

A quantidade de vezes que o loop vai executar depende:

- Do tamanho do vetor
- Se k existe no vetor
- Da posição de k no vetor

Melhor caso: chave em v[0] → **6**

Pior caso: k não está no vetor →
2 + (n + 1) + n * (2 + 2) + 1
5n + 4

Cálculo de passos

Noção de complexidade:

- Seja A um algoritmo, $\{E_1, \dots, E_m\}$, o conjunto de todas as entradas possíveis de A. Denote por t_i o número de passos efetuados por A, quando a entrada for E_i. Definem-se, com p_i sendo a probabilidade de ocorrência da entrada E_i:
  - Complexidade do pior caso: $\max_{E_{i \in E}}\{t_i\}$;
  - Complexidade do melhor caso: $min_{E_{i \in E}}\{t_i\}$;
  - Complexidade do caso médio: $\sum_{1 \leq i \leq m}(p_i \times t_i)$.
- As complexidades têm por objetivo avaliar a eficiência de tempo ou espaço. A complexidade de tempo de pior caso corresponde ao número de passos que o algoritmo efetua no seu pior caso de execução, isto é, para a entrada mais desfavorável. De certa forma, a complexidade de pior caso é a mais importante das três mencionadas.

Cálculo de passos

Exercício: Dada uma matriz n x n de valores inteiros, implemente uma função que localize um dado valor x. A função deve retornar VERDADEIRO se houver achado, e FALSO caso contrário.

Cálculo de passos

Exercício: Dada uma matriz n x n de valores inteiros, implemente uma função que localize um dado valor x. A função deve retornar VERDADEIRO se houver achado, e FALSO caso contrário.

```
func busca(matriz [][]int, n, x int) int {
    i, j int
    i = 0

    for i < n {
        j = 0
        for j < n {
            if (matriz[i][j] == x) {
                return true // achou
            }
            j++
        }
        i++
    }
    return false // não achou
}
```

Qual o número de passos no melhor caso? E no pior caso?

Funções de tempo

As funções de tempo dos principais exemplos analisados anteriormente são:

Algoritmo	Função de tempo
Média	$f(n) = 9$
Fibonacci	$f(n) = 7n + 2$
Somatório de vetor	$f(n) = 6n + 6$
Busca em vetor	$f(n) = 5n + 4$
Busca em matriz	$f(n) = 5n^2 + 5n + 5$

Note que temos uma função **constante**, três funções **lineares** e uma função **quadrática**

Funções de tempo

Comparando o número de passos com base no valor n de entrada:

Valor de Entrada	Média de X e Y	N Fibonacci	Soma Vetor	Busca Vetor	Busca Matriz
Função	9	7n + 2	5n + 6	5n + 4	$5n^2$ +5n + 5
1	9	9	11	9	15
2	9	16	16	14	35
4	9	30	26	24	105
8	9	58	46	44	365
16	9	114	86	84	1365
32	9	226	166	164	5285
64	9	450	326	324	20805
128	9	898	646	644	82565
256	9	1794	1286	1284	328965
512	9	3586	2566	2564	1313285
1024	9	7170	5126	5124	5248005
2048	9	14338	10246	10244	20981765
4096	9	28674	20486	20484	83906565
8192	9	57346	40966	40964	335585285
16384	9	114690	81926	81924	1342259205
32768	9	229378	163846	163844	5368872965
65536	9	458754	327686	327684	21475164165
131072	9	917506	655366	655364	85900001285
262144	9	1835010	1310726	1310724	3.43599E+11

A notação O

Quando se considera o número de passos efetuados por um algoritmo, podem-se desprezar constantes aditivas ou multiplicativas.

Por exemplo, um valor de número de passos igual a 3n será aproximado para n.

Além disso, como o interesse é restrito a valores assintóticos, termos de menor grau também podem ser desprezados. Assim, um valor de número de passos igual a $n^2 + n$ será aproximado para n^2. O valor $6n^3 + 4n - 9$ será transformado em n^3.

Torna-se útil, portanto, descrever operadores matemáticos que sejam capazes de representar situações como essas. A notação O será utilizada com essa finalidade.

A notação O

Sejam f, h funções reais positivas de variável inteira n. Diz-se que f é $O(h)$, escrevendo-se $f = O(h)$, quando existir uma constante $c > 0$ e um valor inteiro n_o, tal que:

$$n > n_o \Rightarrow f(n) \leq c \times h(n)$$

Ou seja, a função h atua como um limite superior para valores assintóticos da função f. Em seguida são apresentados alguns exemplos da notação O.

$$f = n^2 - 1 \Rightarrow f = O(n^2)$$
$$f = n^3 - 1 \Rightarrow f = O(n^3)$$
$$f = 403 \Rightarrow f = O(1)$$
$$f = 5 + 2\log n + 3\log^2 n \Rightarrow f = O(\log^2 n)$$

A notação O

Propriedades da notação O

- $f(n) = O(f(n))$
- $c \cdot O(f(n)) = O(c \cdot f(n)) = O(f(n))$ (c = constante)
- $O(f(n)) + O(f(n)) = O(f(n))$
- $O(O(f(n))) = O(f(n))$
- $f1(n) \cdot O(f2(n)) = O(f1(n) \cdot f2(n))$
- $O(f1(n)) \cdot O(f2(n)) = O(f1(n) \cdot f2(n))$
  - Isto significa que a complexidade de um algoritmo com dois trechos aninhados, em que o segundo é repetidamente executado pelo primeiro, é dada como o produto da complexidade do trecho mais interno pela complexidade do trecho mais externo.
- $O(f1(n)) + O(f2(n)) = O(max(f1(n), f2(n)))$
  - Isto significa que a complexidade de um algoritmo com dois trechos em sequência com tempos de execução diferentes é dada como a complexidade do trecho de maior complexidade.

Aplicações

Escreva um algoritmo em pseudocódigo e implemente em Go os problemas abaixo:

- Dado um array de números inteiros positivos e um valor alvo, encontre um par de números no array cuja soma seja igual ao valor alvo. Se nenhum par for encontrado, retorne um valor (-1, -1) indicando que nenhum par foi encontrado.
- https://br.spoj.com/problems/POPULAR/
- Dado um array de números inteiros positivos, encontre o comprimento da maior subsequência crescente contígua. Uma subsequência crescente é uma sequência de elementos em que cada elemento subsequente é estritamente maior do que o anterior.

Desafios

Escreva um algoritmo em pseudocódigo e implemente em Go os problemas abaixo:

- Dado um array de números inteiros positivos, considerado ordenado crescentemente, e um valor alvo, encontre um par de números no array cuja soma seja igual ao valor alvo. Se nenhum par for encontrado, retorne um valor (-1, -1) indicando que nenhum par foi encontrado. Resolva esse problema com um algoritmo cuja complexidade é O(n).

IBMEC.BR

/IBMEC

IBMEC

@IBMEC_OFICIAL

@IBMEC

Ibmec