Complexidade de Algoritmos

Rômulo César Silva

Unioeste

Janeiro de 2020

unioeste
Universidade Estadual do Oeste do Paraná

Sumário

1. Conceitos Básicos
2. Análise Detalhada
3. Análise Simplificada
4. Estruturas Algoritmicas
5. Complexidade de Algoritmos Recursivos
6. Bibliografia

Lembrete

Estes *slides* são o resultado de diversos estudos realizados e experiências didáticas nas edições da disciplina **Projeto e Análise de Algoritmos** na Universidade Estadual do Oeste do Paraná (Unioeste) desde 2008. Durante as aulas não permaneça invariavelmente calado. Pergunte! Participe! E caso encontre incorreções, erros ou omissões neste material, favor avise enviando e-mail para:

romulocesarsilva@gmail.com

Grato!

Algoritmo e Problema

Algoritmo

Um **algoritmo** é uma sequência de passos para resolução de um *problema*.

Mas, o que é um problema dentro da Teoria da Computação?

Problema

Um **problema** é uma questão geral que pode ser descrita em termos de seus dados ou parâmetros de entrada e da caracterização de sua solução (saída).
Ex.: ordenação de números inteiros.
Dados de entrada: N números inteiros
Saída: Os N números inteiros ordenados crescentemente.

Problema e Instância

Instância

Uma **instância** de um problema corresponde a um exemplo particular dos dados de entrada do problema.
Ex.: considerando o problema da ordenação de número inteiros, os valores 7,2,4,9,11,6 corresponde a uma instância.

Podemos definir formalmente a relação entre um conjunto de instâncias I, problema P e conjunto solução S da seguinte maneira: um problema P é uma função que mapeia instâncias de I em S:

$$P : I \rightarrow S$$

O algoritmo é exatamente a sequência de passos que efetua esse mapeamento.

Análise de Algoritmos

A análise de algoritmos compreende 2 aspectos:

1. **Correção**: verificar se o algoritmos faz o que realmente se propõe a fazer, considerando as diferentes possibilidades de dados de entrada.
2. **Eficiência**: calcular o gasto de tempo e memória envolvidos na execução do algoritmo

Análise de Algoritmos

Geralmente um programa ou *software* é composto de vários algoritmos. Do ponto de vista da Engenharia de Software, a avalição da qualidade de um *software* inclui avaliar entre outros aspectos:

- interface: amigável?
- robustez: capaz de lidar com situações (entradas) adversas?
- compatibilidade: com versões anteriores e/ou com outros softwares.
- flexibilidade: fácil alterar?
- portabilidade: fácil executar em outro ambiente (hardware, SO, *browser*, ...)?
- clareza: código é legível?
- reusabilidade: fácil reutilizar o código implementado?
- **desempenho e consumo de recursos $\Rightarrow$ Análise de Algoritmos**

Análise de Algoritmos

A análise de algoritmos pode ser efetuada de 2 maneiras:

- **Empírica**: consiste em implementar o algoritmo em um computador específico, e testá-lo com diferentes dados de entrada, medindo seu desempenho.
- **Teórica**: consiste em fazer uma análise matemática, procurando obter uma fórmula que expresse seu desempenho em função do tamanho dos dados de entrada.

Análise Empírica de Algoritmos

A **análise empírica** de algoritmos analisa o desempenho medindo o tempo de execução e memória gasta sobre diferentes conjuntos de dados de entrada, que podem ser:

- reais: dados com padrões esperados pelo algoritmo
- randômicas: dados gerados aleatoriamente que satisfaçam os pré-requisitos do algoritmo
- maliciosas: dados que correspondem a situações anormais ou não esperadas pelo algoritmo

A análise empírica é útil na detecção de diferenças de desempenho relacionadas a otimizações de código.

Análise Teórica de Algoritmos

O propósito geral da análise teórica de um algoritmo é prever seu comportamento, sem a necessidade de implementá-lo em um computador específico.
Algumas vantagens dessa abordagem:

- é mais conveniente ter algum tipo de "medida" para a eficiência do algoritmo que implementá-lo e testar a eficiência toda vez que algum parâmetro relacionado ao hardware se altera.
- é importante conseguir demonstrar a corretude do algoritmo antes de implementá-lo, ainda que informalmente, visando garantir o funcionamento correto.
- permite comparar o desempenho de diferentes algoritmos para o mesmo problema, facilitando assim qual escolher.

Análise Teórica de Algoritmos

A análise teórica de complexidade é feita em função do tamanho dos dados de entrada de uma instância genérica.
Por exemplo: Considere o problema de ordenação de números inteiros. Uma instância genérica tem n números a serem ordenados. Assim n corresponde ao tamanho da entrada.

Complexidade de Tempo e Espaço

A **complexidade de tempo** corresponde ao tempo necessário para execução do algoritmo em função do tamanho da entrada.
A **complexidade de espaço** corresponde à memória necessária para execução do algoritmo em função do tamanho da entrada.

Análise Teórica de Algoritmos

A complexidade de tempo geralmente é o fator mais crítico na execução de um algoritmo, pois ao contrário da memória, não pode ser reaproveitado. Assim,a ênfase maior a ser dada neste curso é a análise de tempo.

Análise Teórica de Algoritmos

Para ver isto, considere hipoteticamente 5 algoritmos diferentes para um mesmo problema, cujas complexidades de tempo $T_k(n)$ correspondam ao número de operações realizadas em função do tamanho n da entrada, e considerando que cada operação gasta 1ms.

n	A_1 $T_1(n) = n$	A_2 $T_2(n) = n \lg n$	A_3 $T_3 = n^2$	A_4 $T_4 = n^3$	A_5 $T_5(n) = 2^n$
16	$0.016s$	$0.064s$	$0.256s$	$4s$	$1m4s$
32	$0.032s$	$0.16s$	$1s$	$33s$	46 dias
64	$0.064s$	$0.384s$	$4s$	$4m22s$	5×10^6 séculos
512	$0.512s$	$9s$	$4m22s$	1 dia $13h$	10^{137} séculos

Análise Teórica de Algoritmos

Suponha 2 algoritmos A_1 e A_2 para o mesmo problema com complexidades de tempo $T_1(n) = 3n^2$ e $T_2(n) = 500n + 10$, respectivamente. Quando usar A_1? E quando usar A_2? A rigor A_1 deveria ser usado enquanto $T_1(n) \leq T_2(n)$, o que ocorre até $n = 165$. Ou seja, para entradas com tamanho menor que 165, $T_1(n)$ tem melhor desempenho.

Porém, em situações como essa, na prática escolhe-se A_2, pois na maioria das vezes $n = 165$ ainda é considerada uma entrada relativamente pequena e provavelmente não é o caso da maioria das entradas do problema.

Por isso, geralmente ao se comparar complexidades de algoritmos, as comparações são feitas assintoticamente.

Análise Teórica de Algoritmos

Um mesmo algoritmo pode executar mais rápido para certos dados de entrada que outros. Por isso, na análise de complexidade de algoritmos, procura-se analisar o comportamento do algoritmo para 3 situações diferentes:

- **melhor caso**: situação na qual o algoritmo possui o melhor desempenho, ou seja, corresponde ao menor tempo de execução sobre todas as possíveis entradas.
- **pior caso**: situação na qual o algoritmo possui o pior desempenho, ou seja, corresponde ao maior tempo de execução sobre todas as possíveis entradas.
- **caso médio** ou **esperado**: corresponde à média dos tempos de execução de todas as entradas.

Análise Teórica de Algoritmos

Na análise do **caso esperado** (médio) é feita uma suposição de distribuição de probabilidades sobre o conjunto de entradas, e o custo médio é obtido com base nessa distribuição. Geralmente é suposto que todas entradas são igualmente provavéis, o que nem sempre é verdade na prática.
Observações: As análises do **pior caso** e do **melhor caso** permite-nos fazer as seguintes asserções:

- Se um algoritmo é eficiente para o pior caso, então o é para todos os casos.
- Se um algoritmo é ineficiente para o melhor caso, então o é para todos os casos, e provavelmente deve ser descartado seu uso.

Análise Teórica de Algoritmos

Veremos aqui 2 formas de fazer análise teórica:

- Análise detalhada: leva em conta que os custos das operações (atribuição, soma, subtração, multiplicação, divisão, etc.) são diferentes
- Análise simplificada: considera que os custos das operações escalares são iguais

Em ambos tipos de análise, o objetivo final é obter uma expressão matemática em função do tamanho n da entrada.

Análise Detalhada

Considere o algoritmo de *busca sequencial* de um elemento em um vetor não-ordenado, mostrado no pseudocódigo abaixo:

```
1:  function BUSCA_SEQUENCIAL(A[1..n], x)     ▷ Retorna a posição de x no vetor A,
2:                                             ▷ ou −1 caso não esteja no vetor
3:      i ← 1
4:      while i ≤ n and A[i] ≠ x do
5:          i ← i + 1
6:      end while
7:      if i ≤ n then
8:          return i                           ▷ x está na posição i do vetor A
9:      else
10:         return −1                          ▷ x não está no vetor A
11:     end if
12: end function
```

Análise Detalhada

Assumindo os seguintes custos para as operações:

```
1:  function BUSCA_SEQUENCIAL(A[1..n], x)
2:      i ← 1                                  ▷ c1: custo da atribuição
3:      while i ≤ n and A[i] ≠ x do            ▷ c2: custo do teste do While
4:          i ← i + 1                          ▷ c3: custo do incremento e atribuição
5:      end while
6:      if i ≤ n then                          ▷ c4: custo do teste do If
7:          return i                           ▷ c5: custo do retorno de i
8:      else
9:          return −1                          ▷ c6: custo do retorno de −1
10:     end if
11: end function
```

- **melhor caso**: o elemento x está na primeira posição do vetor A. Neste caso, a linha 4 nunca é executada porque o teste da linha 3 resulta em falso na primeira vez. Assim o custo de execução será: $T(n) = c_1 + c_2 + c_4 + c_5$. Por se tratar de uma soma de constantes, $T(n) \in O(1)$.

Análise Detalhada

```
1:  function BUSCA_SEQUENCIAL(A[1..n], x)
2:      i ← 1                                  ▷ c1: custo da atribuição
3:      while i ≤ n and A[i] ≠ x do            ▷ c2: custo do teste do While
4:          i ← i + 1                          ▷ c3: custo do incremento e atribuição
5:      end while
6:      if i ≤ n then                          ▷ c4: custo do teste do If
7:          return i                           ▷ c5: custo do retorno de i
8:      else
9:          return −1                          ▷ c6: custo do retorno de −1
10:     end if
11: end function
```

- **pior caso**: o elemento x não está no vetor A ou está na última posição. Neste caso, o teste da linha 3 é executado $n+1$ vezes, e a linha 4 é executada n vezes. Assim o custo de execução será: $T(n) = c_1 + c_2(n+1) + c_3 n + c_4 + c_6$, isto é: $T(n) = (c_2 + c_3)n + c_1 + c_2 + c_4 + c_6$. Agrupando as constantes, pode ser reescrita como $T(n) = c_a n + c_b$, que por se tratar de uma função linear, portanto $T(n) \in O(n)$.

Análise Detalhada

```
1:  function BUSCA_SEQUENCIAL(A[1..n], x)
2:      i ← 1                                  ▷ c1: custo da atribuição
3:      while i ≤ n and A[i] ≠ x do            ▷ c2: custo do teste do While
4:          i ← i + 1                          ▷ c3: custo do incremento e atribuição
5:      end while
6:      if i ≤ n then                          ▷ c4: custo do teste do If
7:          return i                           ▷ c5: custo do retorno de i
8:      else
9:          return −1                          ▷ c6: custo do retorno de −1
10:     end if
11: end function
```

- **caso médio**: considerando que toda pequisa tem sucesso, se p_i é a probabilidade de que o i-ésimo elemento seja o procurado, e considerando que para recuperar o i-ésimo elemento são necessárias i execuções da linha 3, temos que o número de execuções da linha 3 é:
  $f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + ... + n \times p_n$.

Análise Detalhada

```
1:  function BUSCA_SEQUENCIAL(A[1..n], x)
2:      i ← 1                                  ▷ c1: custo da atribuição
3:      while i ≤ n and A[i] ≠ x do            ▷ c2: custo do teste do While
4:          i ← i + 1                          ▷ c3: custo do incremento e atribuição
5:      end while
6:      if i ≤ n then                          ▷ c4: custo do teste do If
7:          return i                           ▷ c5: custo do retorno de i
8:      else
9:          return −1                          ▷ c6: custo do retorno de −1
10:     end if
11: end function
```

- **caso médio**: Para calcular $f(n)$, precisa-se conhecer a distribuição de probabilidades p_i. Supondo que cada elemento tenha exatamente a mesma probabilidade de ser o buscado, teremos que $f(n) = \frac{1}{n}(1+2+3+...+n) = \frac{1}{n}(\frac{n(n+1)}{2}) = \frac{n+1}{2}$. Assim, o teste da linha 3 é executado $\frac{n+1}{2}$ vezes, e a linha 4 é executada $\frac{n+1}{2} - 1$ vezes. Portanto o custo de execução é: $T(n) = c_1 + c_2(\frac{n+1}{2}) + c_3(\frac{n+1}{2} - 1) + c_5$.

Análise Detalhada

```
1:  function BUSCA_SEQUENCIAL(A[1..n], x)
2:      i ← 1                                  ▷ c1: custo da atribuição
3:      while i ≤ n and A[i] ≠ x do            ▷ c2: custo do teste do While
4:          i ← i + 1                          ▷ c3: custo do incremento e atribuição
5:      end while
6:      if i ≤ n then                          ▷ c4: custo do teste do If
7:          return i                           ▷ c5: custo do retorno de i
8:      else
9:          return −1                          ▷ c6: custo do retorno de −1
10:     end if
11: end function
```

- **caso médio**:
  $T(n) = c_1 + c_2(\frac{n+1}{2}) + c_3(\frac{n+1}{2} - 1) + c_5$, isto é:
  $T(n) = (c_2 + c_3)\frac{n}{2} + c_1 + \frac{c_2 - c_3}{2} + c_5$. Pode ser reescrita como:
  $T(n) = c_a n + c_b$, tal que $c_a = \frac{(c_2 + c_3)}{2}$ e
  $c_b = c_1 + \frac{c_2 - c_3}{2} + c_5$, e portanto $T(n) \in O(n)$.

Análise Detalhada

Considere o algoritmo *Insertion Sort* que ordena os n elementos de um vetor e os custos da operações escalares envolvidas.

```
 1: procedure InsertionSort(A[1..n])
 2:     for j ← 2, ..., n do                                   ▷ c1
 3:         aux ← A[j]                                         ▷ c2
 4:         ▷ insere A[j] no vetor ordenado A[1..j − 1]
 5:         i ← j − 1                                          ▷ c3
 6:         while i > 0 and A[i] > aux do                      ▷ c4
 7:             A[i + 1] ← A[i]                                ▷ c5
 8:             i ← i − 1                                      ▷ c6
 9:         end while
10:         A[i + 1] ← aux                                     ▷ c7
11:     end for
12: end procedure
```

- **melhor caso**: ocorre quando o vetor já está ordenado. Neste caso, o teste do *While* na linha 6 é executado apenas uma vez para cada $j = 2, ..., n$, encontrando sempre que $A[i] \leq aux$.

Análise Detalhada

```
1:  procedure InsertionSort(A[1..n])
2:      for j ← 2, ..., n do                                   ▷ c1
3:          aux ← A[j]                                         ▷ c2
4:          ▷ insere A[j] no vetor ordenado A[1..j − 1]
5:          i ← j − 1                                          ▷ c3
6:          while i > 0 and A[i] > aux do                      ▷ c4
7:              A[i + 1] ← A[i]                                ▷ c5
8:              i ← i − 1                                      ▷ c6
9:          end while
10:         A[i + 1] ← aux                                     ▷ c7
11:     end for
12: end procedure
```

- **melhor caso**:
  $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$
  Logo, $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$, que pode ser reescrita como: $T(n) = c_a n + c_b$, uma função linear, e portanto $T(n) \in O(n)$.

Análise Detalhada

```
1:  procedure InsertionSort(A[1..n])
2:      for j ← 2, ..., n do                                   ▷ c1
3:          aux ← A[j]                                         ▷ c2
4:          ▷ insere A[j] no vetor ordenado A[1..j − 1]
5:          i ← j − 1                                          ▷ c3
6:          while i > 0 and A[i] > aux do                      ▷ c4
7:              A[i + 1] ← A[i]                                ▷ c5
8:              i ← i − 1                                      ▷ c6
9:          end while
10:         A[i + 1] ← aux                                     ▷ c7
11:     end for
12: end procedure
```

- **pior caso**: ocorre quando os elementos estão ordenados decrescentemente no vetor. Nessa situação, cada elemento $A[j]$ será comparado com cada elemento no subvetor $A[1..j-1]$. Assim, o número de vezes da linha 6 corresponde à somatória $\sum_{2}^{n} j = \frac{n(n+1)}{2}$.

Análise Detalhada

```
1:  procedure InsertionSort(A[1..n])
2:      for j ← 2, ..., n do                                   ▷ c1
3:          aux ← A[j]                                         ▷ c2
4:          ▷ insere A[j] no vetor ordenado A[1..j − 1]
5:          i ← j − 1                                          ▷ c3
6:          while i > 0 and A[i] > aux do                      ▷ c4
7:              A[i + 1] ← A[i]                                ▷ c5
8:              i ← i − 1                                      ▷ c6
9:          end while
10:         A[i + 1] ← aux                                     ▷ c7
11:     end for
12: end procedure
```

- **pior caso**:
  $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n-1)}{2} - 1) + c_5(\frac{n(n-1)}{2}) + c_6(\frac{n(n-1)}{2}) + c_7(n-1)$
  $T(n) = (\frac{c_4+c_5+c_6}{2})n^2 + (c_1+c_2+c_3+(\frac{c_4-c_5-c_6}{2})+c_7)n - (c_2+c_3+c_4+c_7)$,
  que pode ser reescrita como: $T(n) = c_a n^2 + c_b n + c_c$, uma função quadrática, e portanto $T(n) \in O(n^2)$.

Análise Detalhada

Observações:

- o **caso médio** é aproximadamente tão ruim quanto o **pior caso**, por isso geralmente sua análise raramente é feita. Por exemplo, no caso do *Insertion Sort*, o caso médio corresponde a checar metade dos elementos do subvertor $A[1..j-1]$ na linha 6, que calculando $T(n)$ para esta situação obtém-se igualmente uma função quadrática, ou seja, o algoritmo ainda seria $O(n^2)$.
- é comum o uso das notações assintóticas para expressar a complexidade, pois quando o tamanho n da entrada cresce muito, os termos de ordem mais baixa e as constantes envolvidas tornam-se irrelevantes quando comparados ao termo de maior ordem.

Análise Detalhada

Exemplos:

- considere um algoritmo de complexidade $T(n) = n^2 + 5n + 100$. Por ser uma função quadrática, o termo n^2 torna-se consideravelmente maior que os termos $5n$ e 100. Por isso, usa-se $T(n) \in O(n^2)$.
- considere um algoritmo de complexidade $T(n) = 2^n + 50n^2 + 10n$. O termo exponencial 2^n cresce muito mais rapidamente que os termos $50n^2$ e $10n$. Assim, usa-se $T(n) \in O(2^n)$.

Análise Simplificada

- assume-se custo unitário para todas as operações escalares (soma, subtração, multiplicação, divisão).
- assume-se custo unitário para todas as atribuições entre escalares.
- assume-se custo unitário para teste de condições envolvendo escalares e operações booleanas
- o número de repetições em laços definidos (*for*) é obtido diretamente do indíce usado, sendo expresso em somatório, e é contado apenas seu conteúdo interno
- declarações de variáveis e comentários têm custo zero.

Análise Simplificada

Exemplo: considere o pseudocódigo abaixo, que soma os elementos de um vetor:

```
1: function SOMA(A[1..n], n)
2:     aux ← 0
3:     for i ← 1, ..., n do
4:         aux ← aux + A[i]
5:     end for
6:     return aux
7: end function
```

- custo da linha 2: 1
- custo da linha 4: 1 (adição) + 1 (atribuição) = 2 por execução
- custo da linha 6: 1
- custo do *for* (linhas 3 a 5): $\sum_{i=1}^{n} 2$

Assim, $T(n) = 1 + \sum_{i=1}^{n} 2 + 1 = 1 + 2(n - 1 + 1) + 1 = 2n + 2$.

Logo $T(n) \in O(n)$.

Análise Simplificada

Considere o pseudocódigo abaixo, que soma os elementos de uma matriz:

```
1: function SomaMatriz(A[1..n][1..n], n)
2:     aux ← 0
3:     for i ← 1, ..., n do
4:         for j ← 1, ..., n do
5:             aux ← aux + A[i][j]
6:         end for
7:     end for
8:     return aux
9: end function
```

- custo da linha 2: 1
- custo da linha 5: 1 (adição) + 1 (atribuição) = 2 por execução
- custo da linha 8: 1
- custo dos laços (linhas 3 a 7): $\sum_{i=1}^{n} \sum_{j=1}^{n} 2$

Análise Simplificada

```
1: function SomaMatriz(A[1..n][1..n], n)
2:     aux ← 0
3:     for i ← 1, ..., n do
4:         for j ← 1, ..., n do
5:             aux ← aux + A[i][j]
6:         end for
7:     end for
8:     return aux
9: end function
```

Assim:

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n} \sum_{j=1}^{n} 2 + 1 \\ &= 1 + \sum_{i=1}^{n} 2n + 1 \\ &= 1 + 2n(n - 1 + 1) + 1 \\ &= 2n^2 + 2 \end{aligned}$$

Logo $T(n) \in O(n^2)$.

Análise Simplificada

Considere o pseudocódigo abaixo:

```
1: function PROG(n)
2:     aux ← 0
3:     for i ← 0, ..., n − 1 do
4:         for j ← i, ..., n do
5:             aux ← aux + i × j
6:         end for
7:     end for
8:     return aux
9: end function
```

- custo da linha 2: 1
- custo da linha 5: 1 (adição) + 1 (multiplicação) + 1 (atribuição) = 3 por execução
- custo da linha 8: 1
- custo dos laços (linhas 3 a 7): $\sum_{i=0}^{n-1} \sum_{j=i}^{n} 3$

Análise Simplificada

```
1: function PROG(n)
2:     aux ← 0
3:     for i ← 0, ..., n − 1 do
4:         for j ← i, ..., n do
5:             aux ← aux + i × j
6:         end for
7:     end for
8:     return aux
9: end function
```

Assim:

$$
\begin{aligned}
T(n) &= 1 + \sum_{i=0}^{n-1} \sum_{j=i}^{n} 3 + 1 \\
&= 1 + \sum_{i=0}^{n-1} 3(n - i + 1) + 1 \\
&= 1 + 3\left(\sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 1\right) + 1
\end{aligned}
$$

Análise Simplificada

```
1: function PROG(n)
2:     aux ← 0
3:     for i ← 0, ..., n − 1 do
4:         for j ← i, ..., n do
5:             aux ← aux + i × j
6:         end for
7:     end for
8:     return aux
9: end function
```

Assim:

$$
\begin{aligned}
T(n) &= 1 + 3\Big(\sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 1\Big) + 1 \\
&= 1 + 3(n-1-0+1)n + \frac{3(0+(n-1))(n-1-0+1)}{2} + \\
&\qquad 3(n-1-0+1) + 1 \\
&= 1 + 3n^2 + \frac{3(n^2-n)}{2} + 3n + 1 \\
&= \frac{2+6n^2+3n^2-3n+6n+2}{2} \\
&= \frac{9n^2+3n+4}{2}
\end{aligned}
$$

Logo $T(n) \in O(n^2)$.

Estruturas Algoritmicas

Atribuição:

- se sobre escalares tem custo unitário ou $O(1)$:

  $aux \leftarrow 0$ ▷ custo: $O(1)$
  $i \leftarrow j$ ▷ custo: $O(1)$

- se sobre funções, tem custo da função:

  $aux \leftarrow Funcao(A)$ ▷ custo:$O(Funcao)$

Estruturas Algoritmicas

Condicionais:

if b **then**
    S
else
    T
end if

ou

if b **then**
    S
end if

- $C_{If} = O(C_b + C_S)$ quando a condição b é verdadeira
- $C_{If} = O(C_b + C_T)$ no primeiro caso ou $C_{If} = O(C_b)$ no segundo caso quando a condição b é falsa

Em geral, adota-se o pior caso. Assim no primeiro tipo de condicional, caso $C_T > C_S$ usa-se $C_{If} = O(C_b + C_T)$, senão usa-se $C_{If} = O(C_b + C_S)$. Ou seja, $C_{If} = O(C_b + max(C_S, C_T))$.

Estruturas Algoritmicas

Iteração Definida:

for $i \leftarrow j, .., n$ **do**
    S
end for

$C_{For} = O(\sum_{i=j}^{n} C_S)$, considerando que S preserva o tamanho em todas as iterações.

Estruturas Algoritmicas

Iteração Indefinida:

while b **do**
  S
end while

- Considerando entrada d:
  - se d satifaz b, então S é executada sobre d gerando $S(d)$
  - se $S(d)$ satisfaz b, então S é executada sobre $S(d)$ gerando $S^2(d)$
  - ...
  - b não é satifeita para $S^{H(d)}(d)$
- calcular o custo de S para $S(d), S^2(d), S^3(d), ..., S^{H(d)-1}(d), S^{H(d)}(d)$
- $H(d)$ representa o número de vezes que S precisa ser executado para b ser falso

$C_{While} = O((\sum_{i=0}^{H(d)-1} C_S + C_b) + C_b)$, considerando que S preserva o tamanho em todas as iterações.

Estruturas Algoritmicas

Composição ou Sequência:

S;
T;

$$C_{seq} = O(C_S + C_T)$$

Complexidade de Algoritmos Recursivos

Algoritmos Recursivos

Utilizando os mesmos princípios adotados na análise simplificada, expressa-se sua complexidade através de uma recorrência, que é resolvida através de algum dos métodos vistos neste curso.

Complexidade de Algoritmos Recursivos

Considere o pseudocódigo abaixo que calcular o fatorial do inteiro n:

```
1: function FAT(n)
2:     if n = 0 then                          ▷ custo: 1
3:         return 1                           ▷ custo: 1
4:     else
5:         return n × Fat(n − 1)
6:     end if
7: end function
```

- custo da linha 5: 1(multiplicação)+1(retorno) + $T(n-1)$(chamada recursiva) ou 1(multiplicação) + $T(n-1)$(chamada recursiva) caso se despreze o tempo gasto no retorno.

A complexidade de tempo pode ser expressa pela recorrência

$$T(n) = \begin{cases} 1 & \text{se} \quad n = 0 \\ T(n-1) + 1 & \text{se} \quad n > 0 \end{cases}$$

Complexidade de Algoritmos Recursivos

```
1: function FAT(n)
2:     if n = 0 then                          ▷ custo: 1
3:         return 1                           ▷ custo: 1
4:     else
5:         return n × Fat(n − 1)
6:     end if
7: end function
```

A complexidade de tempo pode ser expressa pelas recorrências:

$$T(n) = \begin{cases} 2 & \text{se} \quad n = 0 \\ T(n-1) + 2 & \text{se} \quad n > 0 \end{cases} \text{(detalhada) ou}$$

$$T(n) = \begin{cases} 1 & \text{se} \quad n = 0 \\ T(n-1) + 1 & \text{se} \quad n > 0 \end{cases} \text{(simplificada)}$$

que resolvendo pelo método da iteração chega-se $T(n) = 2(n+1)$ ou $T(n) = n + 1$. Em ambas, $T(n) \in O(n)$.

Complexidade de Algoritmos Recursivos

Considere o pseudocódigo abaixo que calcular o n-ésimo número de Fibonacci:

```
1: function FIB(n)
2:     if n = 0 or n = 1 then
3:         return 1
4:     else
5:         return Fib(n − 1) + Fib(n − 2)
6:     end if
7: end function
```

A complexidade de tempo pode ser expressa pela recorrência

$$T(n) = \begin{cases} 1 & \text{se} \quad n = 0 \quad or \quad n = 1 \\ T(n-1) + T(n-2) & \text{se} \quad n > 1 \end{cases}$$ que

resolvendo pelo método do polinômio característico obtém-se:

$T(n) = (\frac{\sqrt{5}+1}{2\sqrt{5}})[\frac{1+\sqrt{5}}{2}]^n + (\frac{\sqrt{5}-1}{2\sqrt{5}})[\frac{1-\sqrt{5}}{2}]^n$, e portanto

$T(n) \in O(2^n)$ (exponencial).

Complexidade de Algoritmos Recursivos

Considere o pseudocódigo da busca binária abaixo:

```
1:  function BuscaBinaria(A[0..n − 1], x, e, d)
2:      ▷ parâmetros: vetor, valor x procurado, índice da esquerda e índice da direita
3:      if e = d − 1 then
4:          return d
5:      else
6:          m ← (e+d)/2                          ▷ m: índice da posição central
7:          if A[m] < x then
8:              return BuscaBinaria(A, x, m, d)   ▷ recursão para 2ª metade do vetor
9:          else
10:             return BuscaBinaria(A, x, e, m)   ▷ recursão para 1ª metade do vetor
11:         end if
12:     end if
13: end function
```

É importante observar que sempre apenas uma das 2 chamadas recursivas é executada, dependendo se $A[m] < x$ ou não. E em qualquer uma das chamadas recursivas, a faixa do vetor a ser analisada reduze-se à metade da original.

Complexidade de Algoritmos Recursivos

```
1:  function BuscaBinaria(A[0..n − 1], x, e, d)
2:      ▷ parâmetros: vetor, valor x procurado, índice da esquerda e índice da direita
3:      if e = d − 1 then
4:          return d
5:      else
6:          m ← (e+d)/2                          ▷ m: índice da posição central
7:          if A[m] < x then
8:              return BuscaBinaria(A, x, m, d)   ▷ recursão para 2ª metade do vetor
9:          else
10:             return BuscaBinaria(A, x, e, m)   ▷ recursão para 1ª metade do vetor
11:         end if
12:     end if
13: end function
```

Assim, supondo que a chamada inicial seja $BuscaBinaria(A, x, -1, n)$, a recorrência que expressa a complexidade de tempo é: $T(n) = T(\lfloor \frac{n}{2} \rfloor) + 1$, que resolvendo pelo método mestre, obtém-se que $T(n) \in \Theta(lg\ n)$.

Bibliografia I

[Manber 1989] Udi Manber.
Introduction to Algorithms: A Creative Approach.
Addison-Wesley, 1989.

[Ziviani 2011] Nivio Ziviani.
Projeto de Algoritmos: com implementação em Java e C++.
Cengage Learning, São Paulo, 2011.

[Cormen 1997] Cormen,T.; Leiserson, C.; Rivest, R.
Introduction to Algorithms. McGrawHill, New York, 1997.

[Sedgewick 2011] Robert Sedgewick; Kevin Wayne.
Algorithms. Addison-Wesley, New York, 2011.

[Weiss 1997] Mark Allen Weiss.
Data Structures and Algorithm Analysis in C. Addison-Wesley, New York, 1997.

Bibliografia II

[Feofiloff 2009] Paulo Feofiloff.
Algoritmos em linguagem C. Elsivier, Rio de Janeiro, 2009.