Algoritmos, Análise

Prof. Ricardo Reis
Universidade Federal do Ceará
Sistemas de Informação
Estruturas de Dados

2 de maio de 2013

1 Introdução

Um **algoritmo** é um procedimento expresso em forma lógica não ambígua para resolução de um problema e cuja construção independe de linguagem de programação ou hardware utilizados. *Implementar* um algoritmo significa escrevê-lo numa linguagem de programação para posterior execução. O tempo de execução, numa dada máquina, da implementação de um algoritmo varia em função do número de instruções que os processadores presentes são capazes de efetuar por unidade de tempo e também das condições de processamento (memória livre, total de processos em andamento, número de processadores envolvidos, condições de tráfico em rede e etc).

Para evitar análises dependentes de tempo considera-se que um algoritmo é subdividido, ao invés de em milissegundos, em uma quantidade finita de *passos* onde um passo pode ser interpretado como uma instrução indivisível e de *tempo constante*, ou seja, independente de condições de entrada ou processamento.

A quantidade de passos necessários ao cumprimento de um algoritmo é denominada de **complexidade do algoritmo**. A complexidade é em geral sensível a alguma característica da entrada do algoritmo e por essa razão é frequentemente expressa como uma função matemática $f(n)$ onde n é a característica (por exemplo, o valor de uma entrada numérica, o comprimento de um vetor e etc).

2 Medindo a Complexidade de Algoritmos

As ilustrações dessa sessão apresentam alguns algoritmos clássicos e a determinação de suas respectivas complexidades.

ILUSTRAÇÃO-1 (Soma de Matrizes): Sejam A e B duas matrizes quadradas de ordem n que devem ser somadas. O Algoritmo-1 ilustra a resolução do problema. Haja vista que o tempo total da soma é sensível ao número de elementos envolvidos então é conveniente expressar a complexidade deste algoritmo em função de n. Além disso, sendo a soma (linha-4) a operação que predomina durante a execução então nada mais coerente que expressar a complexidade como sendo o total de vezes que elas acontecem. Note que ocorre uma soma para cada uma das n iterações do laço mais interno (linha-3) o qual, por sua vez, é repetido n vezes pelo laço mais externo (diz-se neste caso que os laços são invariantes). Associando os laços contabilizam-se um total de $n \cdot n = n^2$ iterações e logo a complexidade se expressa por, $f(n) = n^2$.

Algoritmo 1 Soma de duas matrizes quadradas

1: **Função** SOMA-MATRIZES(**ref** A, **ref** B, n)
2:     **Para** $i \leftarrow 1$ **até** n **faça**
3:         **Para** $j \leftarrow 1$ **até** n **faça**
4:             $C[i,j] \leftarrow A[i,j] + B[i,j]$
5:     **Retorne** C

ILUSTRAÇÃO-2 (Multiplicação de Matrizes): Sejam A e B duas matrizes quadradas a serem multiplicadas entre si. Na matriz-produto, C, que também é quadrada e de ordem n, cada elemento $C_{i,j}$, com i representando linha e j a coluna, é obtido pelo processamento da i-ésima linha de A com a j-ésima coluna de B. Matematicamente estes processamentos equivalem a somatória,

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

O Algoritmo-2 implementa o produto de duas matrizes. Note que para cada processamento linha-de-A $\times$ coluna-de-B são realizadas n multiplicações e $n-1$ somas. Para medir então a complexidade tem-se algumas alternativas: contabilizar somas, contabilizar multiplicações ou ainda contabilizar operações aritméticas (somas + multiplicações). Na prática, como se verá mais adiante, qualquer uma delas corresponde a uma medida coerente de complexidade. Mas, devido a multiplicação ser uma operação computacionalmente mais custosa que a soma e também de tratar-se de um problema de multiplicação então a medida da complexidade será a quantidade de multiplicações efetuadas.

Notemos que, no Algoritmo-2, similar ao que ocorre na soma (Algoritmo-1), a operação de multiplicação, $\times$, aparece uma vez dentro de um aninhamento de três laços invariantes, ou seja, todos de n iterações. Assim o total de multiplicações realizadas, que corresponde a complexidade do algoritmo, vale $f(n) = n \cdot n \cdot n = n^3$.

Algoritmo 2 Multiplicação de duas matrizes quadradas

Função MULT-MATRIZES(**ref** A, **ref** B, n)
  Para $i \leftarrow 1$ **até** n **faça**
    Para $j \leftarrow 1$ **até** n **faça**
      $C[i,j] \leftarrow 0$
      Para $k \leftarrow 1$ **até** n **faça**
        $C[i,j] \leftarrow C[i,j] + A[i,k] \times B[k,j]$
  Retorne C

ILUSTRAÇÃO-3 (Valores Extremos de um Vetor): Seja L um vetor de inteiros do qual se deseja obter os valores extremos, ou seja, mínimo e máximo. Uma proposta para isso é apresentada pelo Algoritmo-3. A ideia é manter duas variáveis auxiliares, i e j, que deverão conter, ao final do processamento, os índices de L onde estão respectivamente o menor e maior valores. Inicialmente i e j contêm 1 (linha-2) como primeira estimativa (mínimo e máximo na primeira posição). O laço na linha-3 processa o restante do vetor fazendo i e j serem, sempre que necessário, modificados. i muda quando um elemento menor que $L[i]$ é encontrado (linha-4) e j muda quando um elemento maior que $L[j]$ é encontrado (linha-6). Tais ações são desconexas.

Algoritmo 3 Valores extremos de um vetor

1: **Função** EXTREMOS(**ref** L, n)
2:   $i \leftarrow j \leftarrow 1$
3:   **Para** $k \leftarrow 2$ **até** n **faça**
4:     **Se** $L[k] < L[i]$ **então**
5:       $i \leftarrow k$
6:     **Se** $L[k] > L[j]$ **então**
7:       $j \leftarrow k$
8:   **Retorne** $L[i]$, $L[j]$

No Algoritmo-3, uma medida coerente de complexidade é dada pelo total de testes das linhas 4 e 6. Eles ocorrem sempre aos pares em cada iteração do laço principal o qual possui $n-1$ iterações. Assim a função de complexidade é dada por, $f(n) = 2n - 2$.

3 Complexidades de Melhor e Pior Caso

Na maior parte dos algoritmos a função de complexidade $f(n)$ não é geral, ou seja, pode mudar conforme características da entrada. Nestes casos a análise de complexidade consiste em avaliar o algoritmo em situações extremas, ou seja, determinar que funções de complexidade descrevem os casos de melhor e de pior entrada.

A *complexidade de pior caso* é aquela que revela o máximo de passos que o algoritmo pode efetuar para uma dada entrada denominada *entrada do pior caso*. Analogicamente *a complexidade de melhor caso* equivale a quantidade mínima de passos que o algoritmo efetua para uma dada entrada denominada *entrada de melhor caso*.

Matematicamente, seja $E = \{e_1, e_2, e_3, \cdots, e_n\}$ o conjunto de todas as entradas de um algoritmo A e t_i a complexidade de A quando recebe a entrada e_i, com $i \in \{1, 2, 3, 4, \cdots, n\}$. Então se $t_m = \min\limits_{1 \leq i \leq n} t_i$ então t_m é a complexidade de melhor caso e e_m a entrada de melhor caso. Analogicamente se $t_p = \max\limits_{1 \leq i \leq n} t_i$ então t_p é a complexidade de pior caso e e_p a entrada de pior caso.

ILUSTRAÇÃO-4 (Valores Extremos de um Vetor, Versão 2): Para ilustrar as complexidades de melhor e pior caso, tomemos o Algoritmo-3 e o modifiquemos ligeiramente para o Algoritmo-4. Esta nova versão utiliza um **senão** entre-testes (linha-6) e o motivo é explicado a seguir. Para todos os valores que i e j assumem durante o laço **Para** então sempre $L[i] \leq L[j]$. Além do mais quando i e j mudam então os novos valores de $L[i]$ e $L[j]$ são respectivamente menor e maior que os anteriores. Consequentemente cada vez que i muda então j não deve mudar e vice-versa. **senão** provoca este comportamento. Nesta situação podemos falar em pior e melhor caso. No melhor caso todos os testes da linha-4 obtêm êxito impedindo a execução dos testes da linha-6 e fazendo assim um total de $n-1$ testes. No pior caso todos os testes da linha-4 devem falhar exigindo assim que todos os testes da linha-6 ocorram contabilizando $2n-2$ testes. Assim a complexidade de melhor caso é $f_m(n) = n-1$ e de pior caso é $f_p(n) = 2n-2$.

ILUSTRAÇÃO-5 (Valores Extremos de um Vetor, Versão 3): Uma terceira forma de determinar os valores extremos de um vetor de inteiros L de comprimento n é implementada pelo Algoritmo-5. Como nos casos anteriores ainda existem contadores i e j para apontarem em L, no final do processo,

Algoritmo 4 Valores extremos de um vetor (Versão 2)

1: **Função** EXTREMOS(**ref** L, n)
2: $i \leftarrow j \leftarrow 1$
3: **Para** $k \leftarrow 2$ **até** n **faça**
4: **Se** $L[k] < L[i]$ **então**
5: $i \leftarrow k$
6: **senão Se** $L[k] > L[j]$ **então**
7: $j \leftarrow k$
8: **Retorne** $L[i]$, $L[j]$

os respectivos índices onde se encontram os valores mínimo e máximo. Entretanto a ideia principal de busca é diferente: O vetor de entrada é dividido em pares de elementos vizinhos, $\{L[k], L[k+1]\}$, com $k \in \left\{1, 3, 5, 7, ..., \left(2\left\lfloor \frac{n}{2} \right\rfloor - 1\right)\right\}$, que são processados no laço principal, linha-3. Cada par é processado como explicado a seguir. Se o primeiro elemento for menor que o segundo (linha-4) significa que o primeiro elemento poderá ser menor que o $L[i]$ corrente *e/ou* que o segundo é maior que $L[j]$ corrente. Caso uma ou ambas sejam verdadeiras então i e/ou j mudarão por ação das linhas 5 a 7. Similarmente se o primeiro elemento do par for maior que o segundo então este primeiro poderá ser maior que o $L[j]$ corrente *e/ou* o segundo menor que o $L[i]$ corrente. Caso uma ou ambas sejam verdadeiras então i e/ou j mudarão por ação das linhas 10 a 12. Se n for impar (linha-15) então o último elemento $L[n]$ não será processado pelo laço principal pois não forma par. Consequentemente existirá a possibilidade de $L[n]$ ser menor ou maior respectivamente que os valores correntes de $L[i]$ e $L[j]$. Quando este for o caso procederão as linhas 16 a 18 similares ao Algoritmo-4.

No Algoritmo-5 o laço principal efetua $\lfloor n/2 \rfloor$ iterações. Em cada iteração deste laço ocorrem sempre *três* testes (o da linha-4 mais os das linhas 5 e 7 *ou* mais os das linhas 10 e 12). Além disso, entre as linhas 15 e 19, pode ocorrer apenas o teste da linha-15 e nenhum, um ou ambos os testes das linhas 16 e 18. Isso contabiliza, no melhor caso, $3\lfloor n/2 \rfloor + 1$ testes e, no pior caso, $3\lfloor n/2 \rfloor + 3$ testes.

Algoritmo 5 Valores extremos de um vetor (Versão 3)

```
Função EXTREMOS(ref L, n)
    i, j, k ← 1, 2, 1
    Enquanto k < n faça
        Se L[k] < L[k + 1] então
            Se L[k] < L[i] então
                i ← k
            Se L[k + 1] > L[j] então
                j ← k + 1
        senão
            Se L[k] > L[j] então
                j ← k
            Se L[k + 1] < L[i] então
                i ← k + 1
        k ← k + 2                          ▷ Passo igual a 2
    Se n mod 2 ≠ 0 então                   ▷ n é ímpar?
        Se L[n] < L[i] então
            i ← n
        senão Se L[n] > L[j] então
            j ← n
    Retorne L[i], L[j]
```

4 Notação Assintótica

Denomina-se *notação assintótica* a forma matemática de representação simplificada de uma função $f(n)$ levando em conta as componentes de f que *crescem mais rapidamente quando n cresce*. Apresentaremos sucintamente duas destas importantes notações. A notação O (O-grande) e notação Ω (ômega).

Diz-se que $h(n)$ é **limite superior** de $f(n)$, representado por $f(n) = O(h(n))$ (f é O-grande de h), quando existem contantes positivas m e c tal que para todo $n \geq m$ então $c \times h(n) \geq f(n)$ (Figura-1).

Em outras palavras um limite superior denota um *teto* para $f(n)$, ou seja, toda imagem de $f(n)$ ficará abaixo de $c \times h(n)$ para valores de n a partir de m. O cruzamento entre $f(n)$ e $c \times h(n)$ na Figura-1 é único acusando que esta segunda função, a partir de m, estará sempre acima da primeira.

ILUSTRAÇÃO-6 (Verificação de limite superior): Propor um limite superior para

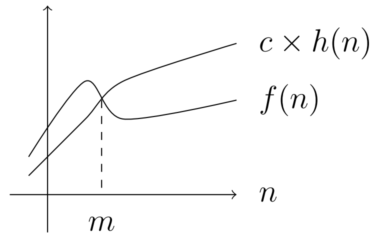

Figura 1: Representação gráfica de um limite superior.

$f(n) = 3n^2 + 18$ juntamente com constantes m e c válidas.

Propomos $h(n) = n^2$ e como constantes válidas citamos $m = 3$ e $c = 5$. Verifiquemos,

$$\begin{aligned} c \times h(n) &\geq f(n) \\ 5(n^2) &\geq 3n^2 + 18 \\ 2(n^2) &\geq 18 \\ n^2 &\geq 9 \Rightarrow \{n \leq -3 \cup n \geq 3\} \end{aligned}$$

Como $m = 3$ e $n \geq 3$ então $3n^2 + 18 = O(n^2)$.

Se $f(n) = O(h(n))$ então existe o limite,

$$\lim_{n\to\infty} \frac{f(n)}{h(n)}$$

ILUSTRAÇÃO-7 (Verificação de limite superior usando limites): Mostre que $5n^3 + 6n - 11 = O(n^3)$.

Fazendo o limite,

$$\begin{aligned} &= \lim_{n\to\infty} \frac{5n^3 + 6n - 11}{n^3} \\ &= \lim_{n\to\infty} \frac{5 + 6\frac{1}{n^2} - \frac{11}{n^3}}{1} \\ &= \frac{5 + 0 + 0}{1} = 5 \end{aligned}$$

Como o limite existe então de fato $5n^3 + 6n - 11 = O(n^3)$.

ILUSTRAÇÃO-8 (Outros exemplos de determinação de limite superior): A soma de matrizes, Algoritmo-1, possui complexidade $O(n^2)$, a multiplicação, Algoritmo-2, é $O(n^3)$ e todas as versões de determinação de extremos de um vetor (Algoritmo-3, Algoritmo-4 e Algoritmo-5) são $O(n)$ (Mostre!).

A determinação do limite superior de uma função $f(n)$ pode ser feita pela seguinte análise,

- Eliminar constantes aditivas e multiplicativas o que dividirá $f(n)$ em componentes para análise.
- Identificar a componente de $f(n)$ que cresce mais rapidamente que as demais (um gráfico poderá auxiliar a tarefa). Esta componente será o limite superior procurado.
- Quando necessário, aplicar o teste $\lim_{n\to\infty} \frac{f(n)}{h(n)}$ onde $h(n)$ é a componente separada.

Ilustração-9 (Determinação de limite superior por análise): Determinar a complexidade assintótica de um algoritmo cuja função de complexidade é $f(n) = 3n^5 + 2^n + 45$.

Eliminando constantes aditivas e multiplicativas encontramos as componentes $\{n^5, 2^n\}$. Como funções exponenciais crescem mais rapidamente que as polinomiais então $f(n) = O(2^n)$.

Ilustração-10 (Outro exemplo de determinação de limite superior por análise): Determinar limite superior para função $f(n) = 5n\log n + 8\log^2 n - 11$.

Eliminando constantes aditivas e multiplicativas encontramos as componentes $\{n\log n, \log^2 n\}$. Quem cresce mais rapidamente? Observe o gráfico destas componentes na Figura-2.

Pela Figura-2 o limite superior é $n\log n$. Verifiquemos,

$$\begin{aligned}
&= \lim_{n\to\infty} \frac{5n\log n + 8\log^2 n - 11}{n\log n} \\
&= \lim_{n\to\infty} 5 + 8\frac{\log n}{n} - 11\frac{1}{n\log n} \\
&= 5 + 8(0) - 11(0) = 5
\end{aligned}$$

Logo, como o limite existe, então $5n\log n + 8\log^2 n - 11 = O(n\log n)$.

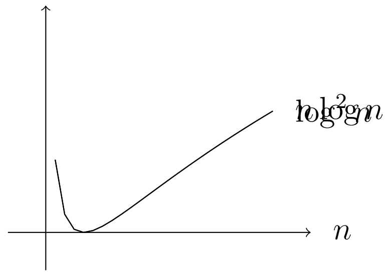

Figura 2: Crescimento das componentes $n\log n$ e $\log^2 n$.

Diz-se que $g(n)$ é **limite inferior** de $f(n)$, representado por $f(n) = \Omega(g(n))$ (f é ômega de g), quando existem contantes positivas m e c tal que para todo $n \geq m$ então $c \times g(n) \leq f(n)$ (Figura-3).

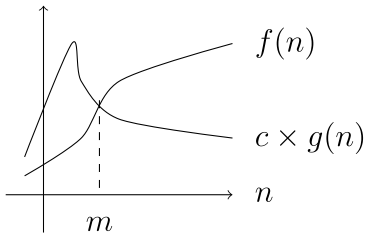

Figura 3: Representação gráfica de um limite inferior.

De forma similar ao limite superior, um limite inferior denota um *piso* para $f(n)$, ou seja, toda imagem de $f(n)$ ficará acima de $c \times g(n)$ para valores de n a partir de m. O cruzamento entre $f(n)$ e $c \times g(n)$ na Figura-3 é único indicando que esta segunda função, a partir de m, estará sempre abaixo da primeira.

Ilustração-11 (Verificação de limite inferior): Propor um limite inferior para $f(n) = n^2 - 3n$ juntamente com constantes m e c válidas.

Propomos $g(n) = n$ e como constantes válidas citamos $m = 4$ e $c = 1$. Verifiquemos,

$$\begin{aligned}
c \times g(n) &\leq f(n) \\
1(n) &\leq n^2 - 3n \\
n^2 - 4n \geq 0 &\Rightarrow \{n \leq -2 \cup n \geq 2\}
\end{aligned}$$

Como $m = 4$ e $n \geq 2$ então $n^2 - 3n = \Omega(n)$.

Se $f(n) = \Omega(g(n))$ então existe o limite,

$$\lim_{n\to\infty} \frac{g(n)}{f(n)}$$

ILUSTRAÇÃO-12 (Verificação de limite inferior usando limites): Verificar que $g(n) = n$ é limite inferior de $f(n) = n^2 - 3n$.

Aplicando limite,

$$\begin{aligned} &= \lim_{n\to\infty} \frac{n}{n^2 - 3n} \\ &= \lim_{n\to\infty} \frac{\frac{1}{n}}{1 - \frac{3}{n}} \\ &= \frac{0}{1-0} = 0 \end{aligned}$$

Como o limite existe então $n^2 - 3n = \Omega(n)$.

5 Complexidade e Recursividade

É prática comum expressar a complexidade de um algoritmo recursivo como a quantidade de chamadas recursivas que ocorrem até sua finalização. Se uma lei de recorrência $T(n)$ (veja artigo sobre recursividade) descreve a medida de chamadas recursivas de um dado algoritmo A então a resolução desta lei implica também na determinação da complexidade de A (em geral de pior caso).

As ilustrações a seguir determinam a lei de recorrência e a complexidade assintótica de algoritmos recursivos.

ILUSTRAÇÃO-13 (Busca Binária): A busca binária é um procedimento de busca em um vetor ordenado que, fazendo uso desta ordenação, faz-se substancialmente mais rápida que a busca linear [1]. A busca binária por um elemento x em um vetor L ordenado segue as etapas,

1. Definir os contadores p e q e iniciá-los respectivamente com as posições inicial e final de L.
2. Se p for menor que q a busca se encerra sem sucesso.
3. Determinar a posição m média entre as posições p e q.
4. Se o elemento $L[m]$ for igual a x a busca termina com sucesso.
5. Se $L[m]$ for maior que x então q recebe $m - 1$ e volta-se a etapa-2. Do contrário se $L[m]$ for menor que x então p recebe $m+1$ e volta-se a etapa-2 (Note que este procedimento divide pela metade o intervalo de busca da próxima iteração).

O Algoritmo-6 implementa a versão recursiva da busca binária. A função BUSCABIN recebe o vetor ordenado L, a chave x que se deseja determinar a posição em L e a faixa de índices $p..q$ onde a busca ocorrerá (para cobrir todo o vetor deve-se fazer a chamada externa repassando-se 1 e n respectivamente para p e q). Quando $p > q$ então BUSCABIN retorna -1 para indicar busca sem sucesso (linha-3). Na busca com sucesso o índice m (linha-4) contendo x é retornado (linha-6). A chamada recursiva para quando $x < L[m]$ ocorre na linha-8. A chamada recursiva para quando $x > L[m]$ ocorre na linha-10. Apenas uma delas procederá numa dada iteração quando x ainda não tiver sido encontrado.

Para medir a quantidade de passos recursivos do Algoritmo-6 consideremos o pior caso, ou seja, a busca sem sucesso. Nesta situação a lei de recorrência característica deve ser,

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + 1 & n > 0 \\ 1 & n = 1 \end{cases}$$

onde $T(n)$ denota a quantidade de passos recursivos quando L é avaliado com comprimento n e o $+1$ denota a contabilização

[1] A *busca linear* investiga um vetor sequencialmente partindo da primeira posição até encontrar o elemento procurado (busca com sucesso) ou até extrapolar a posição final (busca sem sucesso). O pior caso é naturalmente a busca sem sucesso cuja complexidade é $O(n)$ (Mostre!)

Algoritmo 6 Busca binária

```
Função BUSCABIN(ref L, p, q, x)
    Se p > q então
        Retorne −1
    m ← ⌊(p+q)/2⌋
    Se x = L[m] então
        Retorne m
    senão Se x < L[m] então
        Retorne BUSCABIN(L, p, m − 1, x)
    senão
        Retorne BUSCABIN(L, m + 1, q, x)
```

de um passo. Quando p extrapola q significa que na última chamada a BUSCABIN a faixa avaliada possuía comprimento 1 ($n = 1$) e logo é de se esperar que $T(1)$ seja 1, condição esta necessária a parada da recorrência da equação anterior.

Devido ao operador piso $\lfloor\,\rfloor$ na equação anterior, faremos uma simplificação para permitir sus resolução. Consideraremos n uma potência de 2 (ou seja, $n = 2^\alpha$, onde α é um inteiro positivo). Esta simplificação ao mesmo tempo permite a remoção do operador piso (pois toda recorrência tratará um número divisível por 2) como mantém a generalidade da resposta. Assim a lei de recorrência a ser tratada terá forma,

$$T(n) = \begin{cases} T(n/2) + 1 & n > 0 \\ 1 & n = 1 \end{cases}$$

Resolvendo esta lei de recorrência temos,

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= T(n/2/2) + 1 + 1 = T(n/2^2) + 2 \\ &= T(n/2^2/2) + 2 + 1 = T(n/2^3) + 3 \\ &\dots \\ &= T(n/2^k) + k \end{aligned}$$

Dado que $T(1) = 1$ fazemos $n/2^k = 1 \Rightarrow k = \log_2 n$. Utilizando este valor de k obtemos,

$$\begin{aligned} T(n) &= T(n/2^k) + k \\ &= T(1) + \log_2 n \\ &= 1 + \log_2 n \end{aligned}$$

Assim a complexidade do Algoritmo-6 é $f(n) = 1 + \log_2 n$. Seque ainda que,

$$\begin{aligned} f(n) &= 1 + \log_2 n \\ &= 1 + \frac{\log n}{\log 2} \\ &= 1 + \frac{1}{\log 2} \cdot \log n \end{aligned}$$

Eliminando os termos aditivas e multiplicativos obtemos complexidade assintótica $O(\log n)$ [2].

ILUSTRAÇÃO-14 (Potência Rápida): No artigo sobre recursividade foi apresentado um algoritmo que permite de forma eficiente determinar o valor da potência a^n onde $a \in \mathbb{R}$ e $n \in \mathbb{N}$. Ele está aqui novamente reproduzido no Algoritmo-7.

Algoritmo 7 Potência rápida

```
1: Função QPOT(a, n)
2:     Se n > 0 então
3:         x ← QPOT(a, ⌊n/2⌋)
4:         Se n mod 2 = 0 então
5:             Retorne x · x
6:         senão
7:             Retorne x · x · a
8:     senão
9:         Retorne 1
```

Se $T(n)$ representa o número de passos para finalização do Algoritmo-7 então podemos escrever a seguinte lei de recorrência,

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + 1 & n > 0 \\ 1 & n = 1 \end{cases}$$

onde a condição de parada da recorrência refere-se a potência a^1 que requer apenas um passo. Note que esta equação de fato não contabiliza a operação dominante (multiplicação). Entretanto no melhor caso é efetuada uma multiplicação em cada uma das iterações (linha-5) e no pior caso duas multiplicações (linha-7). Essas quantidades contabilizam respectivamente

[2] Este processo, conhecido como mudança de base do logaritmo, foi utilizado para expressar a complexidade na base logarítmica convencional. Entretanto isso não é regra e é correto dizer também que a complexidade do Algoritmo-6 vale $O(\log_2 n)$

$T(n)$ e $2T(n)$ e logo genericamente o total de multiplicações efetuadas deve ser um valor q tal que $T(n) \leq q \leq 2T(n)$. Dado que esta lei de recorrência é a mesma da ilustração anterior então o Algoritmo-7 tem complexidade $O(\log n)$.

6 Exercícios

Nos casos a seguir proponha $g(n)$ e $h(n)$ tal que $f(n) = O(g(n))$ e $f(n) = \Omega(h(n))$. Encontre também m e c válidos conforme definições,

1. $f(n) = 3n^2 + 2n$
2. $f(n) = \log(n^2) + 11$
3. $f(n) = n \log(n + 1)$
4. $f(n) = 3^{2n} + 5^n$
5. $f(n) = (n - 3)!$

Demonstre as proposições a seguir,

6. $g(n) = 7n^2 + 2n^3 - 1$ é $O(n^3)$
7. $g(n) = \frac{n^2}{23} + \frac{1}{n^5}$ é $O(n^2)$
8. $g(n) = \ln(\frac{n}{6})$ é $O(\log(n))$
9. $g(n) = n^2 + 2\log(n)$ é $O(n^2)$
10. $g(n) = 10n\log(n^3) + 9n$ é $O(n\log(n))$
11. $g(n) = 2^{3n-11}$ é $O(2^n)$
12. $g(n) = 2^n + 3^n + 3$ é $O(e^n)$
13. $g(n) = \ln(n^2 + 11n + 6)$ é $O(log(n))$
14. $g(n) = 2^{2n}$ não é $O(2^n)$
15. $g(n) = 3^{n+1}$ é $O(3^n)$

Determine a complexidade de pior caso dos algoritmos a seguir,

16. **Função** F(L, n)
    $s \leftarrow 0$
    Para $i \leftarrow 1$ **até** $n - 1$ **faça**
    Para $j \leftarrow i + 1$ **até** n **faça**
    Se $L[i] > L[j]$ **então**
    $s \leftarrow s + 1$
    Retorne s

17. **Função** G(n)
    $k \leftarrow 0$
    Enquanto $n > 0$ **faça**
    $n \leftarrow n/2$
    $k \leftarrow k + 1$
    Retorne k

18. **Função** H(L, n)
    Se $n > 1$ **então**
    $x \leftarrow \mathrm{H}(L, n - 1)$
    Se $x > L[n]$ **então**
    Retorne x
    senão
    Retorne $L[n]$
    senão Se $n = 1$ **então**
    Retorne $L[1]$

19. **Função** P(n)
    Se $n > 2$ **então**
    $r \leftarrow 0$
    $x \leftarrow \mathrm{P}(n/3)$
    Para $j \leftarrow 1$ **até** n **faça**
    $r \leftarrow r + x$
    Retorne r
    senão
    Retorne 1

20. **Função** Z(n)
    Se $n > 1$ **então**
    $x \leftarrow 0$
    $y \leftarrow \mathrm{Z}(n - 1)$
    Para $i \leftarrow 1$ **até** $n - 1$ **faça**
    Para $j \leftarrow i + 1$ **até** n **faça**
    $x \leftarrow x + y$
    Retorne x
    senão
    Retorne 0

Resolva os problemas a seguir,

21. Sejam as funções de complexidade $a(n) = n^2 - n + 549$ e $b(n) = 49n + 49$ referentes aos algoritmos A e B. Para que valores de n é melhor aplicar o algoritmo A?

22. Considere o seguinte algoritmo recursivo de determinação da maior chave de um vetor. Divide-se o vetor de entrada em três partes e recursivamente determina-se o maior elemento de cada parte. O maior entre os três representará a chave procurada. Construa lei de recorrência, algoritmo e resolva a lei de recorrência encontrada para este algoritmo.

23. A busca ternária é uma variação da busca binária que divide o vetor em três em vez

de duas partes. Proponha o algoritmo e a lei de recorrência inerentes ao problema e em seguida resolva a lei de recorrência.

24. Particionar um vetor L em relação a um elemento $k \in L$ significa rearranjar seus elementos de forma que todos os elementos anteriores e posteriores a k sejam respectivamente menores e maiores que k. Construa algoritmo de complexidade $O(n)$ para particionar um vetor de comprimento n e que ainda retorne a posição final de k em L após o particionamento.

25. Proponha e demonstre algoritmo de complexidade $O(n^2)$ para ordenação de um vetor de números de comprimento n.