UNIVERSIDADE FEDERAL DE PELOTAS
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO
TEORIA DA COMPUTAÇÃO

COMPLEXIDADE DE ESTRUTURAS ALGORÍTMICAS

Professora: Simone Costa

Autores: Matheus de Luna Dobke
Patrícia Teixeira Davet
Thiago Ferreira Pontes.

Pelotas, junho de 2014

Complexidade de Estruturas Algorítmicas

Entre as medidas de complexidade, a complexidade no pior caso é o critério de avaliação mais utilizado. Neste trabalho será apresentada uma metodologia de cálculo de complexidade pessimista, baseada em estruturas algorítmicas.

1. Equações de Complexidade Pessimista:

Analisando a estrutura do algoritmo é possível obter a sua complexidade, passo a passo, através da complexidade de suas componentes. É necessário, para tal, a complexidade de suas componentes básicas e saber como combinar complexidades de componentes.

Neste âmbito será realizado um estudo das principais estruturas algorítmicas: atribuição, sequência, condicional, for e while a fim de estabelecer equações de complexidade para cada uma dessas componentes básicas.

1.1 Atribuição

A atribuição é apresentada na forma:

a ←**b**;

A complexidade associada a esta estrutura depende do tipo dos dados atribuídos.

$c(\mathbf{a} \leftarrow \mathbf{b}) = c(\leftarrow) + c\,(\mathbf{b})$ (1.1-1)

Onde $c(\leftarrow)$ é a complexidade da atribuição para um tipo de dado **a**. Assim, $c(\leftarrow)$ pode ser a complexidade da inserção de um nodo num grafo se **a** é tipo grafo, a atualização de uma matriz se **a** é uma matriz, ou uma operação simples, como a atribuição de um valor inteiro a uma variável. A atribuição pode ainda requerer uma avaliação de b, que pode ser uma expressão ou uma função. Nestes casos, existe um esforço computacional associado à operação de atribuição $c(\leftarrow)$ propriamente dita e um esforço associado à avaliação de b.

Exemplos:

a) Para variáveis inteiras i e j:
   i ←0 {inicialização},
   j ← i {transferência}.

   Ambas tem complexidade constante: Θ(1).

b) Para lista v de inteiros e variável inteira m:
   m ←Max(v) {valor máximo}.

   Esta atribuição envolve (sendo n o comprimento da lista em v):
   - determinar o máximo da lista v, com complexidade Θ(n);
   - transferir este valor, com complexidade Θ(1).
   Portanto sua complexidade tem ordem linear, que é a de maior ordem: Θ(n).

c) Para listas u, v e w:
   u ← v {transfere lista},
   w← Reversa(v) {inverte lista}.

A atribuição de transferência transfere cada elemento da lista v, tendo complexidade Θ(n), para uma lista v com comprimento n.

A atribuição w ← Reversa(v) envolve (sendo n o comprimento da lista em v):

- inverter a lista, com complexidade Θ(n);
- transferir os elementos da lista invertida, com complexidade Θ(n).

Sua complexidade tem ordem n + n, isto é O(n).

1.2 Sequência

Esta estrutura tem a forma:

a; **b**;

A complexidade da sequência é a soma das complexidades de suas componentes. A execução de **a**, entretanto pode alterar o volume de dados para **b**, então:

c(**a** ; **b**) (n) = c(**a**) (n) + c(**b**) (t(n)) onde, t(n) é o tamanho da entrada após a execução de a, dado que antes da execução era n. (1.2-1)

A ordem de execução de uma sequência é importante no cálculo da complexidade, a qual pode variar sensivelmente com uma alteração dessa ordem. A complexidade varia quando a execução de **a** altera o tamanho do problema sobre qual **b** vai atuar.

Exemplos**:**

a) Considere o trecho de algoritmo:
v← Reversa(u); w ← Ordene(v);

onde Reversa e Ordene tem complexidades Θ(n) e O(n^2), respectivamente.
O algoritmo composto tem complexidade de ordem n + n^2, isto é O(n^2).

b) Dados algoritmos Prim(u) e Buscab(a,v), considere a sequência
v ← Prim(u); Buscab(a,v).

Vamos supor que:

- Prim(u) dá como saída a primeira metade da lista em u, com comprimento $\lfloor n/2 \rfloor$, e tem complexidade Θ(n) (sendo n o comprimento da lista em u);

Buscab(a,v) procura **a** na lista v, com complexidade O(logm), para lista v com comprimento m.

Assim, o algoritmo tem complexidade de ordem: n + log $\lfloor n/2 \rfloor$, ou seja, O(n).

1.3 Condicional

A estrutura condicional pode apresentar-se de diversas formas, sendo a mais usual:

if **a** then **b** else **c**

A complexidade desta estrutura é definida pela complexidade da avaliação da condição **a** mais a complexidade de **b** ou a complexidade de **c**, conforme o critério de complexidade a ser utilizado. Como está se tratando de complexidade no pior caso, a complexidade é definida como a complexidade de **a** mais a complexidade máxima entre **b** e **c**. Ou seja;

c(if **a** then **b** else **c**) = c(**a**) + max(c(**b**),c(**c**)) (1.3-1)

Ocorre, porém, que max não tem uma interpretação padrão nesse caso, que c(**b**) e c(**c**) não são simples naturais, mas sim funções de IN em IN.
Frequentemente, a partir de certo ponto, f(n) fica sempre maior do que g(n), ou vice-versa. Então, parece razoável tomar f ou g como max(f,g), conforme o caso. Entretanto, pode acontecer a não existência dessa dominância.
Para exemplificar esta situação, pode-se imaginar um algoritmo que manipula grafos e efetua duas operações, uma cuja complexidade depende exclusivamente do número de arestas e outra cuja complexidade varia exclusivamente com o número de nodos. A função da entrada do algoritmo tem duas componentes: número de arestas e número de nodos, combinados de alguma forma (somados, por exemplo). O algoritmo constitui-se de um condicional cujo ramo then efetua uma das operações, por exemplo aquela dependente do número de arestas, e o ramo else efetua a outra operação. Aumentando o número de arestas, aumenta somente a complexidade do ramo then e aumentando o número de nodos, somente aumenta a complexidade do ramo else. Desta forma, aumentando convenientemente a entrada, a complexidade de cada ramo pode superar a do outro. Nesse ponto, o máximo ponto a ponto pode ser usado, i.e., a função:

max(f,g)(n) := max(f(n),g(n)).

Uma solução simplista, mas muitas vezes usada, é utilizar como máximo a soma ponto a ponto das duas funções: max (f,g) = f + g, com

(f+g) (n) := f(n) + g(n).

Na verdade, há várias possíveis escolhas para máximo assintótico de funções de naturais. Para definir o máximo entre funções, é preciso ter uma relação de ordem entre elas.
A estrutura condicional também pode apresentar-se num modo mais simples sem a presença do else:

if **a** then **b** neste caso, a complexidade desta estrutura é simplificada:

c(if **a** then **b**) = c(**a**) + c(**b**) (1.3-2)

1.4 Iteração Não Condicional ou Definida (Estrutura For)

O caso mais simples de iteração não condicional (ou definida) é:

for **k** = **i** to **j** do **a**

A execução da iteração causa a execução de **a** (**j**-**i**+1) vezes, com o valor de **k** variando de **i** até **j**. Considerando-se que os valores de **i** e **j** não são alterados na execução de **a**, o número de iterações é determinado por (**j**-**i**+1). Pode ocorrer, entretanto, a situação onde a complexidade de execução de **a** varia a cada iteração, por exemplo, alterando o tamanho da entrada, então tem que ser considerada a complexidade de cada iteração executada. Por estas razões, a complexidade desta estrutura tem dois casos a serem considerados: Se a complexidade de execução de **a** não varia durante a iteração:

c(for **k** = **i** to **j** do **a**) = (**j**-**i**+1). c(a) (1.4-1)

Se a complexidade da execução de **a** varia durante a iteração, tem-se que:

c(for **k** = **i** to **j** do **a**) (n) = $\sum_{k=0}^{j-i} c(a)(t^k(n))$ (1.4-2) onde t(n) é o tamanho da entrada após a execução de **a** dado que antes da execução era n e t^k (n) = t(t^{k-1} (n)) se k ≥ 1 e t^0(n) = n e t^{k+1} (n) = t^k (t(n)).

Exemplos:

a) Para variável inteira m:
para k de 1 até 20 faça
m ←m+1
Sua complexidade tem ordem constante O(20) devido ao cálculo da equação de complexidade resultar na constante 20, pois o número de iterações (**j**-**i** +1=20) vezes 1 (complexidade de **a,** que é uma atribuição) =20.

b) Considere a iteração definida:
para k de 1 até n-1 faça
Troca(A[k], A[k+1]);

Onde Troca(A[p], A[q]) troca de posição os elementos A[p] e A[q] no vetor a[1..n], com complexidade constante O(1), digamos.
Assim, cada iteração mantém fixo o tamanho do vetor A[1..n] de entrada.

São executados (n-1) trocas sucessivamente:
Troca(A[1], A[2]), Troca(A[2], A[3]),........, Troca(A[n-1], A[n]);
Cada uma delas com mesma complexidade O(1).

A iteração definida tem complexidade de ordem:
1+1+1...+1 (n-1) vezes

Assim teremos complexidade:
c(para k de 1 até n-1 faça Troca(A[k], A[k+1])) =
= O((n-1).1) = O(n).

c) Considere a iteração definida:
para k de 1 até n faça **a** onde

c(a) é um polinômio de grau m, i.e. c(a) (n) = O(n^m); e
tam(a(d)) = tam(d) – 1 (portanto a(n)= n-1 e $a^k(n)$= n - k).
Neste caso, temos
c(para k de 1 até n faça **a**) (n) = O($\sum_{k=1}^{n}(n-1)^m$).

Pode-se mostrar que $\sum_{k=1}^{n}(n-1)^m$ = O(n^{m+1}). Portanto a complexidade para a iteração é O(n^{m+1}).

1.5 Iteração Condicional ou Indefinida (Estrutura While)

As estruturas de iteração condicional (ou indefinida) podem assumir várias formas. A forma vista a seguir será o while, mas o tratamento para as demais estruturas é similar.
while **a** do **b**

Neste tipo de iteração **b** será executado sucessivamente enquanto a condição **a** for satisfeita, possivelmente com alteração no volume dos dados.

c(while **a** do **b**)(n) = c(a) $(t^k(n))$ + $\sum_{i=0}^{k-1} c(a) + c(b)(t^i(n))$ (1.5-1)

Exemplos:

a) Fatorial de natural n:
f ←1; i←n {inicializações};
enquanto i >1 faça f ← $f.i$; i←i-1 fim enquanto {iteração}.

Sua complexidade tem ordem (n-1) . 1 : O(n).

b) Considere a seguinte iteração indefinida
enquanto 1≤i ≤n faça
Buscas(A[i], B[i..n]); i←i+1 fim enquanto

Onde Buscas procura **a** na parte [i..n] (de j até n) do vetor B, com complexidade O(m) (onde m:= n-i+1).
Assim, cada iteração consiste de uma busca em um vetor de dimensão m:= n-i+1.
Desprezando o teste, a iteração indefinida tem complexidade pessimista.
c(enquanto 1≤i ≤n faça Buscas(A[i], B[1..i]); i←i+1 fim enquanto) =
= O($\sum_{i=0}^{h(n)-1} c(\text{Buscas}(\text{A}[\text{i}], \text{B}[1..\text{i}]);\ \text{i} \leftarrow \text{i}+1)$) $(s^i(n))$.

Neste caso, temos:

c(Buscas(A[i], B[1..i]); i←i+1) (m)= O(m);
h(n)=n e s(m)= m-1 (dando $s^i(m) = m - i$).)

Logo c(Buscas(A[i], B[1..i]); i←i+1)) $(s^i(n))$ = O(n-1)
Como $\sum_{i=0}^{n-1}(n-i)$ =O(n^2).

Portanto, a ordem de complexidade desta iteração indefinida é quadrática O(n^2).

Referência.

[1] Toscani, Laira Vieira; Veloso, Paulo A.S.– *Complexidade de Algoritmos* – Série Livros Didáticos– Instituto de Informática da UFRGS, 2002.