Semana 4

Algoritmos iterativos: especificação, corretude e análise

Nesta aula vamos nos concentrar nos algoritmos iterativos, ou seja, nos algoritmos que podem usar **comandos de repetição**, mas que **não utilizam recursão**. Veremos como especificar e projetar já levando em conta a prova de corretude.

A vantagem de **combinar o projeto com a prova de corretude** é facilitar a demonstração de que o algoritmo é correto. A especificação com prova de corretude é bem **trabalhosa** e detalhada, o que inviabiliza seu uso em algoritmos grandes. Porém, quando o algoritmo é pequeno mas possui detalhes que precisam ser considerados com cuidado para evitar bugs, este procedimento de especificação com prova de corretude produz algoritmos com **chance muito menor de conter erros**.

No restante da disciplina não vou exigir especificações neste nível de detalhe, mas nas demonstrações e em alguns exercícios vou utilizar elementos do que vamos aprender aqui: pré-condições, pós-condições, medida de progresso, invariante do loop, condição de saída, código pré-loop, código do loop e código pós-loop.

Veremos também nesta aula como **simplificar a solução dos somatórios que aparecem na análise de complexidade** de algoritmos. Vimos na aula de análise assintótica que a complexidade de tempo é medida através da contagem do número de execuções das instruções. Neste contagem aparecem somatórios que precisamos resolver para obter a expressão da complexidade. A solução exata de somatórios pode exigir muita manipulação algébrica, mas como estamos interessados apenas na aproximação Θ destas expressões, esta tarefa fica bem mais simples, como veremos na seção sobre aproximação assintótica de somatórios.

4.1 Especificação e corretude

Veja os vídeos "projeto e corretude de algoritmos iterativos" e "projeto e corretude de algoritmos (exemplos)".

4.2 Análise de complexidade

4.2.1 Aproximação assintótica de somatórios

Veja o vídeo "aproximação de somatórios". Caso esteja interessado nas demonstrações, veja também o vídeo "demonstrações das regras de aproximação de somatórios", disponibilizado no material complementar.

Vou resumir a seguir os principais resultados apresentados no vídeo. Em cada caso vou apresentar a solução assumindo que $f(i)$ está na forma básica $c \cdot b^{ai} \cdot i^d \cdot \log^e i$, e cada somatório começa em um valor constante n_0 (ou seja, o valor de n_0 não depende de n).

Soma geométrica

Quando $b^a > 1$, o somatório é dominado pelo último termo.

$$\sum_{i=n_0}^{n} f(i) \ \in\ \Theta(f(n)) = \Theta(b^{an} \cdot n^d \cdot \log^e n).$$

Ex.: $\sum_{i=1}^{n} 2^i \log i \ \in\ \Theta(2^n \log n)$.

Esta regra também vale quando $f(i)$ cresce mais que exponencial.
Ex.: $\sum_{i=1}^{n} 2^{i^2} \ \in\ \Theta(f(n)) = \Theta(2^{n^2})$.

Soma aritmética

Quando $b^a = 1$ e $d > -1$, podemos agrupar os termos em pares cujas somas são aproximadamente iguais (primeiro com último, segundo com penúltimo, assim sucessivamente). Portanto, o somatório será da ordem do número de termos vezes o último termo.

$$\sum_{i=n_0}^{n} f(i) \ \in\ \Theta(n \cdot f(n)) = \Theta(n^{d+1} \cdot \log^e n).$$

Ex.: $\sum_{i=1}^{n} i^2 \log i \ \in\ \Theta(n^3 \log n)$.

Soma com cauda limitada

Quando $b^a < 1$ (geométrica decrescente) ou quando $b^a = 1$ e $d < -1$ (aritmética decrescente), o somatório é dominado pelo primeiro termo.

$$\sum_{i=n_0}^{n} f(i) \ \in\ \Theta(f(n_0)) = \Theta(1).$$

Ex.: $\sum_{i=3}^{n} (1/2)^i \log i \ \in\ \Theta((1/2)^3 \log 3) = \Theta(1)$.

Esta regra também vale quando $f(i)$ decresce mais rápido que exponencial.
Ex.: $\sum_{i=1}^{n} 1/2^{i^2} \ \in\ \Theta(1/2^{1^2}) = \Theta(1)$.

Soma harmônica

Quando $b^a = 1$, $d = -1$ e $e = 0$, temos

$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + \epsilon, \quad \text{com } 0 < \epsilon \leq 1.$$

Portanto,

$$\sum_{i=n_0}^{n} \frac{1}{i} \in \Theta(\log n).$$

Obs.: $\ln n$ é o logaritmo de n na base 2,718281828459... (número de Euler).

4.2.2 Contagem de instruções

Para determinar a complexidade de pior caso de um algoritmo iterativo podemos determinar a complexidade de cada linha do algoritmo em notação O ou Θ, e em seguida somar estas complexidades. A complexidade de cada linha é a complexidade da instrução multiplicada pelo número de vezes que a instrução é executada. A notação O é mais apropriada quando não sabemos se uma análise mais detalhada revelaria um crescimento assintótico menor. Considere o exemplo abaixo (`https://pt.wikipedia.org/wiki/Insertion_sort`).

Algoritmo: Ordenação_por_inserção($L[1..n]$)

Entrada: Lista $L[1..n]$ de números indexados de 1 até n.
Saída : Reorganização dos elementos de L em ordem não decrescente.

1 **for** j *de 2 até* n **do** // $O(n)$
2 chave $= L[j]$ // $O(n)$
3 $i = j - 1$ // $O(n)$
4 **while** $i \geq 1$ **and** $L[i] >$ chave **do**
5 $L[i+1] = L[i]$
6 $i = i - 1$
$\Bigg\}\ \sum_{j=2}^{n}\sum_{i=1}^{j-1} O(1)$
7 $L[i+1] =$ chave // $O(n)$
8

O for da linha 1 gasta $O(1)$ por execução para incrementar o j e para testar se $j \leq n$, e é executado $O(n)$ vezes, totalizando $O(n)$. As linhas 2, 3 e 7 também gastam $O(1)$ por execução, e são executados $O(n)$ vezes, totalizando $O(n)$. As linhas 4, 5 e 6 também gastam $O(1)$ por execução, restando apenas determinar quantas vezes estas linhas são executadas. No pior caso, temos uma iteração do while da linha 4 para todo i de 1 até $j-1$, e a variável j pode assumir valores de 2 até n (for da linha 1). Portanto, o número de execuções das linhas 4, 5 e 6 vale

$$\sum_{j=2}^{n}\sum_{i=1}^{j-1} O(1) = \sum_{j=2}^{n} (j-1)O(1) = \sum_{j=2}^{n} O(j) = O(n^2).$$

Concluímos que a complexidade do algoritmo é

$$4 \cdot O(n) + 3 \cdot O(n^2) = O(n^2).$$

Como na notação assintótica vamos **ignorar as instruções com menor crescimento**, podemos concentrar a contagem de instruções apenas na parte que sabemos que no pior caso provocará o maior crescimento. No caso da ordenação por inserção, uma avaliação rápida permite concluir que basta determinar a complexidade das linhas 4, 5 e 6.

Qual seria a complexidade do algoritmo se cada execução da linha 2 gastasse $O(n \log n)$? Como já sabemos que a linha 2 é executada $O(n)$ vezes, a complexidade desta linha seria $O(n) \cdot O(n \log n) = O(n^2 \log n)$. Neste caso esta linha iria determinar a complexidade do algoritmo, ao invés das linhas 4, 5 e 6 que possuem complexidade $O(n^2)$. Ou seja, o algoritmo passaria a ter complexidade $O(n^2 \log n)$.

Leitura complementar

Contagem de instruções: Seções 2.1 e 2.2 do Cormen, e a Seção 2.5 do Skiena.

A especificação de algoritmos iterativos e as regras para aproximação de somatórios na forma básica foram extraídas do livro *How to think abount algorithms* de Jeff Edmonds (não disponível na biblioteca).