Semana 5

Algoritmos recursivos: especificação, corretude e análise

5.1 Especificação e corretude

Veja os vídeos “projeto e corretude de algoritmos recursivos” e “projeto e corretude de recursivos (exemplos)”.

5.2 Análise de complexidade

Na determinação da complexidade de algoritmo recursivos precisamos determinar e resolver a recorrência que fornece o total de instruções executadas. Representar a função de complexidade $T(n)$ como uma **recorrência** significa que vamos expressar $T(n)$ através da própria função T aplicada em valores menores que n. Por exemplo, $T(n) = T(n-1) + 1$ é uma recorrência, pois para definir $T(n)$ usamos a função T aplica em $n-1$, que é menor do que n.

Para que a definição de uma recorrência esteja completa, precisamos de uma definição não recursiva para valores suficientemente pequenos de n (caso base). Nas funções de complexidade assumimos que $T(n)$ está definida para valores positivos de n, e que $T(c) \in \Theta(1)$ para qualquer constante positiva c, ou seja, **para qualquer tamanho de entrada que não dependa de n, o tempo de execução será constante**.

Uma vez determinada a recorrência que fornece o número de instruções executadas, precisamos encontrar uma **solução não recursiva para a recorrência**, para poder determinar seu crescimento. Temos várias técnicas para solução de recorrências, mas em muitos casos a determinação da solução é uma tarefa complexa. Felizmente, como na análise de algoritmos estamos **interessados apenas no crescimento** da solução, podemos resolver as recorrências mais comuns utilizando **regras simples**, como veremos na Seção “Aproximação assintótica de recorrências”. Antes disso, vejamos como extrair a recorrência que define a complexidade de um algoritmo recursivo.

5.2.1 Complexidade representada por recorrência

Como extrair a função de complexidade a partir de uma algoritmo recursivo? Vamos considerar os dois casos mais comuns.

Divisão e conquista

A entrada de tamanho n é dividida em b partes com aproximadamente o mesmo tamanho, e o algoritmo é chamado recursivamente para a destas partes. Além disso, o algoritmo gasta tempo $f(n)$ nas instruções que não são chamadas recursivas. Concluímos que a complexidade do algoritmo vale $T(n) = a \cdot T(n/b) + f(n)$.

Como exemplo, considere o algoritmo de ordenação **Mergesort**. Este algoritmo divide a lista de entrada de tamanho n em duas listas com aproximadamente $n/2$ elementos. Em seguida, o algoritmo é chamada recursivamente para as duas listas menores. Finalmente, um procedimento de tempo $\Theta(n)$ é executado para intercalar as duas listas ordenadas retornadas pelas chamadas recursivas. Concluímos que a complexidade deste algoritmo vale $T(n) = 2 \cdot T(n/2) + \Theta(n)$. Veremos que a solução desta recorrência é $T(n) \in \Theta(n \log n)$.

Redução e conquista

Em a iterações retiramos b elementos da entrada, onde b é positivo e não depende de n, e chamamos o algoritmo recursivamente para os $n-b$ elementos restantes. As instruções que não são chamadas recursivas gastam tempo $f(n)$. Portanto, a complexidade vale $T(n) = a \cdot T(n-b) + f(n)$.

Como exemplo, considere uma implementação recursiva do algoritmo de **ordenação por inserção**. Vamos retirar o último elemento e recursivamente ordenar os $n-1$ primeiros elementos da lista. Quando a chamada recursiva terminar teremos os $n-1$ primeiros elementos em ordem crescente, restando apenas posicionar o último elemento. Para colocar o último elemento na posição correta vamos precisar deslocar cada elemento maior que o último uma posição para a direita. No pior caso este deslocamento vai gastar tempo $\Theta(n)$. Concluímos que a complexidade deste algoritmo vale $T(n) = T(n-1) + \Theta(n)$. Veremos que a solução desta recorrência é $T(n) \in \Theta(n^2)$.

O problema das **Torres de Hanoi** faz 2 chamadas recursivas, cada uma com $n-1$ discos, e as instruções que não são chamadas recursivas gastam tempo constante. Portanto, a complexidade vale $T(n) = 2 \cdot T(n-1) + \Theta(1)$, cuja solução veremos que vale $T(n) \in \Theta(2^n)$.

A implementação recursiva de algoritmos de redução e conquista geralmente provocam **estouro de pilha**, pois o número de stack frames empilhados é da ordem do tamanho da entrada. Portanto, é necessário **transformar o algoritmo recursivo em iterativo**, mas isso geralmente é uma tarefa simples. Este problema de estouro de pilha **não ocorre em algoritmos de divisão e conquista**, pois lá o número de stack frames empilhados é da ordem do logaritmo do tamanho da entrada.

5.2.2 Aproximação assintótica de recorrências

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

Vou resumir a seguir os principais resultados apresentados no vídeo. Em cada caso assuma que a, b, d e e são constantes, ou seja, não dependem do valor de n.

Recorrências de divisão e conquista

Se $T(n) = a \cdot T(n/b) + f(n)$, com $f(n) = \Theta(n^d \log^e n)$,

$\log_b a$ **versus** d	e	Solução
$<$	qualquer	$\Theta(f(n))$
$=$	> -1	$\Theta(f(n) \cdot \log n)$
$=$	$= -1$	$\Theta(n^d \cdot \log \log n)$
$=$	< -1	$\Theta(n^{\log_b a})$
$>$	qualquer	$\Theta(n^{\log_b a})$

Exemplos:

1. $T(n) = 9T(n/3) + n^3 \in \Theta(n^3)$.
2. $T(n) = 9T(n/3) + n^2 \in \Theta(n^2 \log n)$.
3. $T(n) = 9T(n/3) + n^2/\log n \in \Theta(n^2 \log \log n)$.
4. $T(n) = 9T(n/3) + n^2/\log^2 n \in \Theta(n^2)$.
5. $T(n) = 9T(n/3) + n \in \Theta(n^2)$.
6. Mergesort: $T(n) = 2T(n/2) + \Theta(n) \in \Theta(n \log n)$.

Recorrências de reduzir e conquistar

Se $T(n) = a \cdot T(n-b) + f(n)$, com $f(n) = \Theta(n^d \log^e n)$,

a	Solução
> 1	$\Theta(a^{n/b})$
$= 1$	$\Theta(n \cdot f(n))$

Exemplos:

1. $T(n) = 8T(n-3) + f(n) \in \Theta(8^{n/3}) = \Theta(2^n)$.
2. $T(n) = T(n-3) + n \log n \in \Theta(n \cdot (n \log n)) = \Theta(n^2 \log n)$.
3. Ordenação por inserção: $T(n) = T(n-1) + \Theta(n) \in \Theta(n^2)$.
4. Torres de Hanoi: $T(n) = 2T(n-1) + \Theta(1) \in \Theta(2^n)$.

De todas as soluções apresentadas, a única que produz **crescimento maior que polinômio** ocorre quando $T(n) = a \cdot T(n-b) + f(n)$, e $a > 1$. Portanto, precisamos reduzir a complexidade de algoritmos que fazem mais de uma chamada recursiva removendo apenas uma quantidade constante de elementos. Uma estratégia para isso é usar programação dinâmica.

Leitura complementar

Divisão e conquista, e estratégias gerais para solução de recorrências: Capítulo 4 do Cormen, Capítulo 2 do Vazirani (exceto Seção 2.6), Seção 4.10 do Skiena. O livro *Matemática concreta* do Knuth faz uma apresentação detalhada de várias técnicas para solução de recorrências.

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