Recursividade, recorrências e técnicas de resolução de recorrências

7 de outubro de 2011

1 Introdução

O projeto de algoritmos para a resolução de problemas de maneira eficiente é grandemente facilitado quando se tem em mente algumas abordagens comuns para tal fim. Duas dessas técnicas são a recursão e a divisão e conquista.

2 Recursão

Um algoritmo é recursivo se chama a si mesmo para fazer parte de seu trabalho. Para esta abordagem ser bem sucedida, a chamada a si mesma deve ser para um problema menor que o original. Em geral, um algoritmo recursivo deve ter duas partes: o caso base, que trabalha uma entrada simples que pode ser resolvida sem necessitar de uma chamada recursiva, e a parte recursiva que contém uma ou mais chamadas recursivas do algoritmo onde os parâmetros estão, de certa forma, mais perto do caso base que aqueles da chamada original. Veja abaixo a função recursiva para calcular o fatorial de um inteiro n. A linha 2 é o caso base do problema e a linha 4 é a parte recursiva.

```
1: function Fat(n: integer):integer;
2:     if n = 1 then return 1;
3:     else return n * Fat(n - 1);
```

O uso da recursividade geralmente permite uma descrição mais clara e concisa dos algoritmos, especialmente quando o problema a ser resolvido é

recursivo por natureza ou utiliza estruturas recursivas como árvores. A forma usual de se implementar um procedimento recursivo é por meio de uma pilha, na qual são armazenados os parâmetros para cada chamada de uma subrotina que ainda não terminou de processar. Todos os dados não globais vão para a pilha, pois o estado corrente da computação deve ser registrado para que possa ser recuperado de uma nova ativação de uma subrotina recursiva, quando a ativação anterior deverá prosseguir.

A técnica básica para demonstrar que uma repetição termina é definir uma função $f(x)$, onde x é o conjunto de variáveis do programa, tal que:

1. $f(x) \leq 0$ implica a condição de término;
2. $f(x)$ é decrementada a cada iteração.

É necessário mostrar que o nível mais profundo de recursão seja não apenas finito, mas também possa ser mantido pequeno, pois, na ocasião de cada ativação recursiva de uma subrotina P, uma parcela de memória é necessária para acomodar variáveis a cada chamada.

Algoritmos recursivos são apropriados quando o problema a ser resolvido ou os dados a serem tratados são definidos em termos recursivos (pode citar exemplo de busca em árvores, como pré-ordem). Entretanto, isso não garante que um algoritmo recursivo é o melhor caminho para resolver o problema. Por exemplo, no cálculo dos números de Fibonacci, o programa recursivo é extremamente ineficiente pois recalcula o mesmo valor várias vezes. Veremos logo adiante que a programação dinâmica é mais eficaz para resolver esse problema. Assim, devemos evitar o uso de recursividade quando existe uma solução óbvia por iteração. Programas recursivos que possuem chamadas ao final do código são ditos terem recursividade de cauda e são facilmente transformáveis em uma versão não recursiva.

O tempo de execução dos algoritmos recursivos pode ser descrito por uma recorrência, que é uma equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores. Podemos então utilizar ferramentas matemáticas para resolver a recorrência e fornecer limites da performance do algoritmo.

A seguir, introduzimos o paradigma conhecido como divisão e conquista, que geralmente aparece combinado com recursividade na resolução de problemas. Então falaremos um pouco a respeito da complexidade de algoritmos recursivos.

3 Divisão e conquista

O paradigma divisão e conquista consiste em dividir o problema a ser resolvido em partes menores (subproblemas), encontrar soluções para os subproblemas e então combinar as soluções obtidas em uma solução global. É utilizada para resolver diversos problemas como, por exemplo, na ordenação de um conjunto de elementos. Exemplos são os algoritmos *Quicksort* e *Mergesort* que utilizam recursividade e divisão e conquista.

Considere o funcionamento do algoritmo de ordenação por intercalação (*Mergesort*): divida recursivamente o vetor a ser ordenado em dois vetores até obter n vetores de um único elemento. Intercale dois vetores de um elemento, formando um vetor ordenado de dois elementos. Repita este processo formando vetores ordenados cada vez maiores até que todo o vetor esteja ordenado (veja o pseudocódigo na Figura 1).

```
1: procedure MergeSort(var A: array[1..n] of integer; i, j: integer);
2:     se i < j então
3:         m := (i + j − 1)/2;
4:         MergeSort(A,i,m);
5:         MergeSort(A,m + 1,j);
6:         Merge(A,i,m,j);
```

Figura 1: Algoritmo de ordenação por intercalação

Considere $T(n)$ o tempo de execução de um problema de tamanho n e suponha que o problema seja dividido em a subproblemas, cada um dos quais com tamanho n/b. Para n suficientemente pequeno, menor do que uma constante c, diremos que o tempo de execução é constante: $\Theta(1)$. Se levarmos o tempo $D(n)$ para dividir o problema em subproblemas e o tempo $C(n)$ para combinar as soluções dadas aos subproblemas na solução para o problema original, temos

$$T(n) = \begin{cases} \Theta(1), \text{se } n \leq c, \\ aT(n/b) + D(n) + C(n) \text{ caso contrário.} \end{cases} \quad (1)$$

Para o algoritmo *Mergesort*, temos $a = b = 2$, ou seja, cada instância é dividida em dois subproblemas cujo tamanho é a metade do problema original. Para facilitar a análise, vamos supor que o tamanho do problema é uma potência de dois (o que não afeta a ordem de crescimento da solução para a recorrência). Podemos desmembrar o tempo de execução em:

- **Dividir:** simplesmente calcula o ponto médio do subarranjo, o que demora um tempo constante. Portanto, $D(n) = \Theta(1)$.
- **Conquistar:** resolvemos recursivamente dois subproblemas; cada um tem o tamanho $n/2$ e contribui com $T(n/2)$ para o tempo da execução.
- **Combinar:** o procedimento *MERGE* em um subarranjo de n elementos leva o tempo $\Theta(n)$ para intercalar os elementos. Assim, $C(n) = \Theta(n)$.

Desta forma, a recorrência para o tempo de execução $T(n)$ do pior caso do *Mergesort* é:

$$T(n) = \begin{cases} \Theta(1), \text{se } n = 1, \\ 2T(n/2) + \Theta(n) \text{ se } n > 1. \end{cases} \quad (2)$$

4 Técnicas de Análise de Algoritmos

Para exemplificar algumas das técnicas de análise de algoritmos que veremos a seguir, utilizaremos a recorrência do tempo de execução $T(n)$ do *Mergesort* dada na Equação 2, no pior caso.

4.1 Método da Substituição

Em geral, o método da substituição envolve duas etapas:

1. pressupor um limite hipotético;
2. usar indução matemática para provar que a suposição está correta.

O método é empregado em casos fáceis de pressupor a forma da resposta e pode ser usado para estabelecer limites superiores ou inferiores.

Para fazer um bom palpite, podemos usar resultados de recorrências semelhantes. Também podemos provar limites superiores (ou inferiores) e reduzir o intervalo de incerteza. Por exemplo, para equação do *Mergesort*, poderíamos primeiro provar que $T(n) = O(n^2)$. E depois provar que $T(n) = O(n \log n)$ que é de fato a complexidade de pior caso do *Mergesort*. Além disso, podemos utilizar o resultado da árvore de recursão, que veremos a seguir, como limite hipotético para o método de substituição.

4.2 Método de Árvore de Recursão

Uma árvore de recursão apresenta uma forma bem intuitiva para a análise de complexidade de algoritmos recursivos. Numa árvore de recursão cada nó representa o custo de um único subproblema da respectiva chamada recursiva. Primeiro, somamos os custos de todos os nós de um mesmo nível, para obter o custo daquele nível. Então, somamos os custos de todos os níveis para obter o custo da árvore.

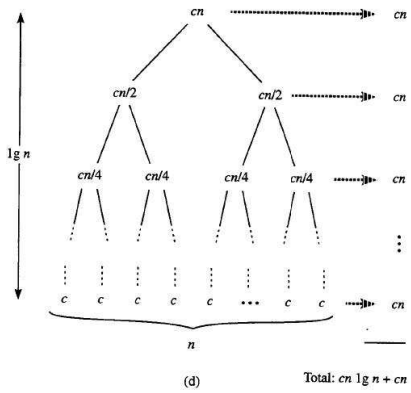
Para analisar a recorrência do *Mergesort* utilizando árvore de recursão, vamos reescrever a recorrência 2 como:

$$T(n) = \begin{cases} c, \text{se } n = 1, \\ 2T(n/2) + c.n \text{ se } n > 1. \end{cases} \tag{3}$$

onde a constante c representa o tempo exigido para resolver problemas de tamanho 1, como também o tempo por elemento do arranjo para as etapas de dividir e combinar. A figura abaixo mostra como podemos resolver a recorrência. Por conveniência, supomos que n é uma potência exata de 2. A figura mostra $T(n)$ que é progressivamente expandido para formar a árvore de recursão. O termo $c.n$ é a raiz (o custo no nível superior da recursão), e as duas subárvores da raiz são as duas recorrências menores $T(\frac{n}{2}$ que também são expandidos na sequência. O custo para cada um dos dois nós no segundo nível da recursão é $\frac{c.n}{2}$. Continuamos a expandir cada nó da árvore, desmembrando-o em suas partes constituintes como determina a recorrência, até os tamanhos de problemas se reduzirem a 1, cada qual com o custo c. A árvore completamente expandida tem $\lg n + 1$ níveis (isto é, altura $\lg n$) e cada nível contribui com o custo total de $c.n$. Então o custo total é $c.n \lg n + c.n$, que é $\Theta(n \log n)$.

4.3 Método Mestre

O Teorema Mestre é um método de resolução de recorrências no formato $T(n) = aT(n/b) + f(n)$. Este formato de recorrência descreve o tempo de execução de um algoritmo que divide um problema de tamanho n em a subproblemas, cada um de tamanho n/b, onde $a \geq 1$ e $b > 1$ são constantes positivas. Os a subproblemas são resolvidos recursivamente, cada um no tempo $T(n/b)$. O custo de dividir o problema e combinar os resultados dos subproblemas é descrito pela função $f(n)$, que deve ser uma função assintoticamente positiva para que o Método Mestre possa ser utilizado.



Assim, se uma recorrência estiver no formato descrito acima, então $T(n)$ é limitada assintoticamente como:

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para algum $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$.
3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para algum $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para $c < 1$ e para todo n suficientemente grande, então $T(n) = \Theta(f(n))$.

Observe que os três casos acima não abrangem todas as possibilidades, por isso, mesmo que uma recorrência esteja no formato descrito anteriormente, pode ser que tenhamos que recorrer a outro método para analisá-la. Aplicando o método sobre a recorrência do tempo de execução $T(n)$ do pior caso do *Mergesort*, obtemos $a = 2$, $b = 2$ e $f(n) = \Theta(n)$. Pelo segundo caso, chegamos à $\Theta(n \lg n)$.