Recursividade e divisão e conquista

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. Apresentamos, neste texto, algumas dessas técnicas – a saber: recursão e divisão e conquista – com as suas respectivas abordagens para determinarmos as complexidades temporais de execução dos algoritmos obtidos com essas técnicas.

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 3 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 como a seguir:

- **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:

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

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 e b 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 $f(n)$.

A recorrência que surge do *Mergesort* tem $a = 2$, $b = 2$ e $f(n) = \Theta(n)$. De acordo com o método, se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \lg n)$. Como $f(n) = \Theta(n)$, a resolução desta recorrência é $\Theta(n \lg n)$. Tecnicamente, na realidade a recorrência do *Mergesort* não está bem definida, porque n/b poderia não ser um inteiro. Porém, a substituição de cada um dos a termos $T(n/b)$ por $T(\lfloor n/b \rfloor)$ ou $T(\lceil n/b \rceil)$ não afeta o comportamento assintótico da recorrência. Por esta razão omitimos as funções piso e teto.

Uma das vantagens dos algoritmos de divisão e conquista é que seus tempos de execução são, em geral, facilmente determinados por técnicas de análise de algoritmos tais como o Método Mestre utilizado acima, o Método de Substituição e o Método de Árvore de Recursão.

4. Conclusão

Neste texto, abordamos algumas técnicas importantes de projeto e análise de algoritmos eficientes. Discutimos as técnicas de divisão e conquista e recursividade, bem como a combinação de ambas. Nesse contexto, quando efetuamos a decomposição de um problema em subproblemas, procuramos fazê-la de modo que o número de problemas seja tão pequeno quanto possível para que a decomposição não se torne onerada (sobrecarregada). Ao mesmo tempo, porém, os subproblemas obtidos devem ser de tamanhos tão próximos quanto possível. Neste tópico, poderíamos ainda discutir a técnica de balanceamento que corresponde à decomposição de um problema em subproblemas com a preocupação do número e do tamanho dos subproblemas.