Técnicas de projeto de algoritmos: Indução

ACH2002 - Introdução à Ciência da Computação II

Delano M. Beder

Escola de Artes, Ciências e Humanidades (EACH)
Universidade de São Paulo
dbeder@usp.br

08/2008

Material baseado em slides dos professores Marcos Chaim, Cid de Souza e Cândida da Silva

Indução Matemática

- Técnica matemática muito poderosa para provar asserções sobre números naturais.
- Seja T um teorema que desejamos provar. Suponha que T tenha como parâmetro um número natural n.
- Ao invés de provar diretamente que T é válido para todos os valores de n, basta provar as duas condições a seguir:
  1. T é válido para $n = 1$ (**passo base**)
  2. Para todo $n > 1$, se T é válido para n - 1, então T é válido para n (**hipótese da indução ou passo indutivo**).

Indução Matemática

- Normalmente, provar a condição 1 é relativamente fácil.
- Provar a condição 2 é mais fácil do que provar o teorema, pois pode-se utilizar do fato de que T é válido para n - 1.
- Por que a indução funciona? Por que as duas condições são suficientes?
  - As condições 1 e 2 implicam que T é válido para $n = 2$.
  - Se válido T é válido para $n = 2$, então pela condição 2 implica que T também é válido para 3, e assim por diante.
- O princípio da indução é um axioma dos números naturais.

Indução Matemática

Exemplo 1

Considere a expressão de soma dos primeiros números naturais n, isto é, $S(n) = 1 + 2 + \cdots + n$.
Provar por indução que $S(n) = \frac{n*(n+1)}{2}$.
Prova:
Passo base: Para $n = 1$, S(1) = 1 (trivial).
Passo indutivo:
Pelo princípio da indução matemática podemos assumir $S(n-1) = \frac{(n-1)*(n-1+1)}{2}$ como válido.
Mas $S(n) = S(n-1) + n = \frac{(n-1)*(n-1+1)}{2} + n = \frac{n*(n+1)}{2}$.

Assim, provamos o passo indutivo.
Logo, $S(n) = \frac{n*(n+1)}{2}$ para todo $n \geq 1$.

Indução Matemática

Exemplo 2

Prove por indução que $2^n > 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^0$, $n \geq 1$.
Prova:
Passo base: Para $n = 1$, $2^1 > 2^0$ (trivial).
Passo indutivo: Pelo princípio da indução matemática

$$2^{n-1} > 2^{n-2} + 2^{n-3} + \ldots + 2^0 \text{ é verdadeiro}$$

Somando 2^{n-1} nos dois lados da inequação (lembrando que 2^{n-1} é positivo e por isso não altera o sinal da inequação) obtemos:

$$2.2^{n-1} > 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^0.$$

Portanto, $2^n > 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^0$. O passo indutivo está provado.

Logo, $2^n > 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^0$ para todo $n \geq 1$.

Variações

- Variação 1
  - Caso base: n = 1
  - Provar que para $\forall n \geq 2$, se a propriedade é válida para n, ela é valida para $n + 1$.
- Variação 2
  - Caso base: n = 1, 2 e 3
  - Provar que para $\forall n \geq 4$, se a propriedade é válida para n, ela é valida para $n + 1$.
- Variação 3 (Indução forte)
  - Caso base: n = 1
  - Provar que para $\forall n \geq 2$, se a propriedade é válida para $\forall 1 \geq m \geq n$, ela é valida para $n + 1$.

Exercícios - Indução Matemática

1. Prove que $1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{2n^3+3n^2+n}{6}, \forall n \geq 1$
2. Prove que $1 + 3 + 5 + \cdots + 2n - 1 = n^2, \forall n \geq 1$
3. Prove que $1^3 + 2^3 + 3^3 + \cdots + n^3 = \frac{n^4+2n^3+n^2}{4}, \forall n \geq 1$
4. Prove que $1^3 + 3^3 + 5^3 + \cdots + (2n-1)^3 = 2n^4 - n^2, \forall n \geq 1$
5. Prove que $1 + 2 + 2^2 + 2^3 + \cdots + 2^n = 2^{n+1} - 1, \forall n \geq 0$
6. Prove que $2^n \geq n^2, \forall n \geq 4$.
7. Prove que $\frac{1}{1} - \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{2n-1} - \frac{1}{2n} = \frac{1}{n+1} + \frac{1}{n+2} + \cdots + \frac{1}{2n}$
8. Prove a soma dos cubos de três numeros naturais positivos sucessivos é divisível por 9.
9. Prove que todo número natural $n > 1$ pode ser escrito como o produto de primos (indução forte).
10. Prove que todo número natural positivo pode ser escrito como a soma de diferentes potências de 2 (indução forte).

Princípio da Recursão

- Definições recursivas de métodos são baseadas no *princípio matemático da indução*.
- A idéia é que a solução de um problema pode ser apresentada da seguinte forma:
  - Primeiramente, definimos a solução para os casos básicos;
  - Em seguida, definimos como resolver o problema para os demais casos, porém, de uma forma mais simples.

Usando Indução em Programação

- Problema: definir a multiplicação de dois números inteiros não negativos m e n, em termos da operação de adição
- Qual o caso base ?

Usando Indução em Programação

- Problema: definir a multiplicação de dois números inteiros não negativos m e n, em termos da operação de adição
- Qual o caso base ?
  - Se n é igual a 0, então a multiplicação é 0.
- Qual seria o passo indutivo

Usando Indução em Programação

- Problema: definir a multiplicação de dois números inteiros não negativos m e n, em termos da operação de adição
- Qual o caso base ?
  - Se n é igual a 0, então a multiplicação é 0.
- Qual seria o passo indutivo
  - Temos que expressar a solução para $n > 0$, supondo que já sabemos a solução para algum caso mais simples.
  - $m * n = m + (m * (n - 1))$.

Usando Indução em Programação

- Problema: definir a multiplicação de dois números inteiros não negativos m e n, em termos da operação de adição
- Qual o caso base ?
  - Se n é igual a 0, então a multiplicação é 0.
- Qual seria o passo indutivo
  - Temos que expressar a solução para $n > 0$, supondo que já sabemos a solução para algum caso mais simples.
  - $m * n = m + (m * (n - 1))$.
- Portanto, a solução do problema pode ser expressa:
  - $m * 0 = 0$
  - $m * n = m + (m * (n - 1))$
- Como programar esta solução em Java?

Usando Indução em Programação

```
class Aritr {

   static int multr (int m, int n) {
    if(n == 0) {
      return 0;
    }
    else {
      return (m + multr(m, n-1));
    }
  }
}
```


Utilizando Recursão

Comando/ Expressão	Resultado (expressão)	Estado (após execução/avaliação)		
`multr (3,2)` ...		m —> 3 n —> 2		
n == 0	`false`	m —> 3 n —> 2		
return m + `multr (m,n-1)`	...	m —> 3 n —> 2	m —> 3 n —> 1	
n == 0	`false`	m —> 3 n —> 2	m —> 3 n —> 1	
return m + `multr (m,n-1)`	...	m —> 3 n —> 2	m —> 3 n —> 1	m —> 3 n —> 0
n == 0	`true`	m —> 3 n —> 2	m —> 3 n —> 1	m —> 3 n —> 0
`return 0`		m —> 3 n —> 2	m —> 3 n —> 1	
`return m + 0`		m —> 3 n —> 2		
`return m + 3`				
`multr (3,2)`	6			

Recursão

- Para solucionar o problema, é feita uma outra chamada para o próprio método, por isso, este método é chamado *recursivo*.
- Recursividade geralmente permite uma descrição mais clara e concisa dos algoritmos, especialmente quando o problema é recursivo por natureza.
- Cada chamada do método `multr` cria novas variáveis de mesmo nome `m` e `n`.
- Portanto, várias variáveis `m` e `n` podem existir em um dado momento.
- Em um dado instante, o nome (`m` ou `n`) refere-se à variável local ao corpo do método que está sendo executado.

Recursão

- As execuções das chamadas de métodos são feitas em uma *estrutura de pilha*.
- Pilha: estrutura na qual a inserção (ou alocação) e a retirada (ou liberação) de elementos é feita de maneira que o último elemento inserido é o primeiro a ser retirado.
- Assim, o último conjunto de variáveis alocadas na pilha corresponde às variáveis e aos parâmetros do último método chamado.
- O espaço de variáveis e parâmetros alocado para um método é chamado de *registro de ativação* desse método.
- O registro de ativação é desalocado quando termina a execução de um método.

Recursão

- Variáveis que podem ser usadas no corpo de um método:
  - variáveis ou atributos de classe (*static*): criados uma única vez;
  - variáveis ou atributos de instância: criados quando é criado um novo objeto (*new*);
  - parâmetros e variáveis locais: criados cada vez que é invocado o método.
- Na criação de uma variável local a um método, se não for especificado um valor inicial, o valor armazenado será *indeterminado*.
  - Indeterminado = valor existente nos bytes alocados para essa variável na *pilha* de chamada dos métodos.

Iteração

```
class Aritr {

   static int multr (int m, int n) {
    int r = 0;

    for (int i = 0; i <= n; i++) {
      r += m;
    }

    return r;
  }
}
```


Iteração

Comando/ Expressão	**Resultado** (expressão)	**Estado** (após execução/avaliação)
`multr (3,2)`	...	m —> 3, n —> 2
`int r = 0`	...	m —> 3, n —> 2, r —> 0
`int i = 1`		m —> 3, n —> 2, r —> 0, i —> 1
`i <= n`	`true`	m —> 3, n —> 2, r —> 0, i —> 1
`r+=m`	3	m —> 3, n —> 2, r —> 3, i —> 1
`i++`	2	m —> 3, n —> 2, r —> 3, i —> 2
`i <= n`	`true`	m —> 3, n —> 2, r —> 3, i —> 2
`r+=m`	6	m —> 3, n —> 2, r —> 6, i —> 2
`i++`	3	m —> 3, n —> 2, r —> 6, i —> 3
`i <= n`	`false`	m —> 3, n —> 2, r —> 6, i —> 3
`for ...`		m —> 3, n —> 2, r —> 6
`return r`	6	m —> 3, n —> 2, r —> 6
`mult(3,2)`	6	

Recursão x Iteração

- Soluções recursivas são geralmente mais concisas que as iterativas. Programas mais simples.
- Soluções iterativas em geral têm a memória limitada enquanto as recursivas, não.
- Cópia dos parâmetros a cada chamada recursiva é um custo adicional para as soluções recursivas.
- Programas recursivos que possuem chamadas no final do código são ditos terem *recursividade de cauda*. São facilmente transformáveis em uma versão não recursiva.
- Projetista de algoritmos deve levar consideração a complexidade (temporal e espacial), bem como os outros custos (e.g., facilidade de manutenção) para decidir por qual solução utilizar.

Exercícios

1. Forneça soluções recursivas para os problemas abaixo
   - cálculo do fatorial de um número.
   - cálculo do elemento n da série de Fibonacci.
     - $f_0 = 0$, $f_1 = 1$,
     - $f_n = f_{n-1} + f_{n-2}$ para $n >= 2$.
   - busca binária.
2. Como faço para calcular a complexidade (temporal ou espacial) de um algoritmo recursivo?
3. Escreva um método recursivo que calcule a soma dos elementos positivos do vetor de inteiros $v[0..n-1]$. O problema faz sentido quando n é igual a 0? Quanto deve valer a soma nesse caso? (Retirado de [1])
4. Escreva um método recursivo `maxmin` que calcule o valor de um elemento máximo e o valor de um elemento mínimo de um vetor $v[0..n-1]$. Quantas comparações envolvendo os elementos do vetor a sua função faz? (Retirado de [1])

Exercícios

5. Escreva um método recursivo que calcule a soma dos elementos positivos do vetor $v[ini..fim-1]$. O problema faz sentido quando ini é igual a fim? Quanto deve valer a soma nesse caso? (Retirado de [1])
6. Escreva um método recursivo que calcule a soma dos dígitos de um inteiro positivo n. A soma dos dígitos de 132, por exemplo, é 6. (Retirado de [1])
7. Escreva um método recursivo `onde()`. Ao receber um inteiro x, um vetor v e um inteiro n, o método deve devolver j no intervalo fechado $0 \ldots n-1$ tal que $v[j] == x$. Se tal j não existe, o método deve devolver -1. (Retirado de [1])
8. Escreva um método recursivo que recebe um inteiro x, um vetor v e inteiros ini e fim e devolve j tal que $ini \leq j \leq fim - 1$ e $v[j] == x$. Se tal j não existe então devolve ini-1. (Retirado de [1])

Referências

Referências utilizadas: [1] (Capítulo 5) e [2] (páginas 35-42).

[1] C. Camarão & L. Figueiredo. *Programação de Computadores em Java*. Livros Técnicos e Científicos Editora, 2003.

[2] N. Ziviani. *Projeto de Algoritmos com implementações em C e Pascal*. Editora Thomson, 2a. Edição, 2004.