Recursão

Vinicius A. Matias

May 9, 2021

1 Introdução

Recursão é uma poderosa técnica de programação que tipicamente torna os códigos mais legíveis, ainda que com algumas consequências (principalmente no consumo de memória). Tal técnica utiliza os conceitos de indução matemática e podem ter sua complexidade estudada pelas funções de recorrência.

2 Indução matemática

A indução é uma técnica matemática para provar um teorema T para todos os valores de n a partir de um n_0 base. Para provar um teorema por indução devem ser obedecidas duas condições:

1. Passo base: T é válido para um n mínimo;
2. Passo indutivo: Para todo n a partir do n base, se T é válido para $n-1$, então T é válido para n.

2.1 Exemplo 1

Considerando a soma dos primeiros n números naturais como $S(n) = 1 + 2 + ... + n$; queremos provar por indução que:
$S(n) = \frac{n*(n+1)}{2}, \forall n \geq 1$

Passo base: $S(1) = 1$
$S(1) = \frac{1*(1+1)}{2} = 1$

Passo Indutivo:
Como o caso base é verdadeiro, assumimos:
$S(n-1) = \frac{(n-1)*((n-1)+1)}{2}$ verdadeiro por Hipótese de Indução
Então, para encontrarmos $S(n)$ conhecendo $S(n-1)$ devemos notar que precisamos adicionar n à $S(n-1)$, ou seja:
$S(n) = S(n-1) + n$
E isso é verdade pois:

$S(n) = S(n-1) + n$
$S(n) = \frac{(n-1)*((n-1)+1)}{2} + n$
$S(n) = \frac{(n-1)*n}{2} + n$
$S(n) = \frac{n^2-n}{2} + n$
$S(n) = \frac{n^2-n+2n}{2}$
$S(n) = \frac{n^2+n}{2}$
$S(n) = \frac{n(n+1)}{2}$

Assim, está demonstrado que $S(n) = \frac{n(n+1)}{2}$ é a fórmula para a soma dos primeiros n números naturais.

2.2 Exemplo 2

Provar por indução que:
$2^n = 2^{n-1} + 2^{n-2} + ... + 2^1 + 2^0 + 1$

Passo base: n = 1
$2^1 = 2^{1-1} + 1$
$2^1 = 2$

Passo indutivo:
$ST(n-1) = 2^{n-1}$ (verdadeiro por Hipótese de Indução)
Portanto, precisamos notar que dado $ST(n-1)$ podemos definir $ST(n)$ como:
$ST(n) = S(n-1) + 2^{n-1}$
$ST(n) = 2^{n-1} + 2^{n-1}$
$ST(n) = 2 * 2^{n-1}$
$ST(n) = 2^{n-1+1}$
$ST(n) = 2^n$
Assim, provamos pelo passo indutivo que se $ST(n-1) = 2^{n-1}$, $ST(n) = 2^n$.

3 Recursão

A recursão se baseia na indução matemática. Quando sabemos resolver um caso simples do problema, podemos fazer chamadas sequenciais do problema (computacionalmente, uma função) até que consigamos chegar em um ponto simples e de solução direta.

Implementações recursivas ocupam mais memória que um método iterativo, pois as chamadas aos métodos são empilhadas até se chegar ao caso base.

3.1 Exemplo

Problema: Usando apenas o operador de adição, definir a multiplicação de dois números inteiros não negativos m e n.

Caso Base: Se $m = 1$ e $n = 0$, temos $m*n = 1*0 = 0$

Mult(0) = 0

Mult(n-1) é verdadeiro por hipótese de indução.

Perceba que a multiplicação também pode ser expressa como:

$m*n = m*(n-1)+???$

$m*n = m*(n-1)+m$

Assim, podemos definir um código como o seguinte:

Listing 1: Multiplicação recursiva

```
1 def mult(m, n):
2         if n == 0: return 0
3         return mult(m, n−1) + m
```

Apenas para comparação, um código iterativo para a multiplicação é exibido na Listing 2.

Listing 2: Multiplicação iterativa

```
1 def mult(m, n):
2         soma = 0
3         for i in range(n):
4                 soma += m
5         return soma
```

4 Exercícios

4.1 Fatorial

Resolva recursivamente o cálculo da fatorial de um número.

Caso base: Fatorial de 0 é igual à 1

Passo indutivo: Como fat(n-1) = (n-1)! é verdade por Hipótese de Indução, temos que:

$fat(n) = fat(n-1)*n$

Listing 3: Fatorial recursiva

```
1 def fatorial(n):
2         if n == 0: return 1
3         return fatorial(n−1)∗n
```

4.2 Fibonacci

Sendo que por definição um elemento n da série de fibonacci é igual à soma de seus dois antecessores, temos que:

Caso base: $f_0 = 0$ e $f_1 = 1$

Passo indutivo: $f_n = f_{n-1} + f_{n-2}, \forall n \geq 2$ é verdadeiro por Hipótese de Indução.

Definindo um código recursivo para calcular o n-ésimo valor da sequência de fibonacci, podemos utilizar a implementação da Listing 4.

Listing 4: Fionacci recursivo

```
1 def fib(n):
2         if n == 0 or n == 1: return n
3         return fib(n−1) + fib(n−2)
```

Note que uma versão iterativa deste código é mais viável, pois é recorrente fazer a mesma ação duas vezes. Ainda, com um $n = 40$ o tempo de execução do código já fica inviável.

4.3 Busca sequencial

Definir recursivamente a busca sequencial.

Listing 5: Busca sequencial recursiva

```
1 def busca_seq(A, valor, n):
2         if n == 0: return −1
3         if A[n−1] == valor: return n−1
4         return busca_seq(A, valor, n−1)
```

4.4 Busca binária

Definir recursivamente a busca binária.

Listing 6: Busca binária recursiva

```
1 def busca_binaria(A, valor, n):
2         if n == 0: return −1
3         if A[n−1] == valor: return n−1
4         return busca_seq(A, valor, n−1)
```