Recursão

Vinicius A. Matias

May 10, 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 de recursão

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.

A busca binária utiliza o fato de usar um arranjo ordenado para encontrar um valor, tendo complexidade $\mathcal{O}(\log n)$. Uma implementação iterativa da busca binária iterativa é exibida a seguir.

Listing 6: Busca binária iterativa

```
1 def busca_binaria(A, valor, n):
2   ini = 0
3   fim = n−1
4   while ini <= fim:
5     meio = int((ini+fim)/2)
6         if valor > A[meio]: ini = meio+1
7         elif valor < A[meio]: fim = meio−1
8         else: return meio
9   return −1
```

A busca binária usa um arranjo A de tamanho n (índices variando no intervalo $[0, n-1]$) para encontrar um valor. A estratégia que reduz significativamente a complexidade da busca é de se

procurar elementos pelo meio do arranjo, verificando se o elemento buscado é maior ou menor que este valor, fazendo uma busca no meio no "subarranjo" maior ou menor que o índice até encontrar o índice que indica um elemento igual ao buscado, ou terminando a busca sem encontrar o elemento.

A implementação recursiva da busca binária tem como caso base a verificação de que não há mais nenhum elemento à ser buscado no arranjo (retorno -1) e quando o elemento é encontrado (retornando o índice do arranjo A). Nota-se que o passo base é o mesmo da busca sequencial.

Por Hipótese de Indução, sabemos calcular busca_binaria(A, valor, ini, meio-1) e busca_binaria(A, valor, meio+1, fim)

O algoritmo recursivo pode ser implementado como:

Listing 7: Busca binária recursiva

```
1  def busca_binaria(A, valor, ini, fim):
2    if fim <= ini: return -1
3    meio = int((ini+fim)/2)
4
5    if valor > A[meio]: ,
6      return busca_binaria(A, valor, meio+1, fim)
7    elif valor < A[meio]:
8      return busca_binaria(A, valor, ini, meio-1)
9    else:
10     return meio
```

5 Equações de Recorrência

Algoritmos recursivos não seguem os padrões já estudados para análise de complexidade. Para estudar o custo da recursão, deve-se utilizar uma equação de recorrência, que se baseia essencialmente nas chamadas que um método recursivo tende a fazer.

5.1 Exemplos

Mostraremos a definição das equações de recorrência para os exercícios da seção 4.

Fatorial: Uma equação de recorrência é divida em dois casos: um para passo base e outro para o indutivo. Primeiro se encontra a operação mais relevante para o algoritmo e desenvolvemos a equação de recorrência com base nessa operação.

Para o cálculo da fatorial a operação mais importante é a multiplicação, isso implica que nossa equação de recorrência contará quantas multiplicações o algoritmo desenvolvido faz.

Para o caso base ($n = 0$ ou $n = 1$) o algoritmo não realiza nenhuma multiplicação, então o custo é 0 para esses valore de n.

Para o passo indutivo, notamos que é realizada uma multiplicação no método, além de uma chamada à $n - 1$, implicado em um custo de $T(n-1)+1$. A equação de recorrência pode ser definida como:

$$T(n) = \begin{cases} 0, n \leq 1 \\ T(n-1) + 1,\ n > 1 \end{cases}$$

Fibonacci: A operação relevante para o algoritmo recursivo de fibonacci é a adição. No caso base o custo desta operação é 0. Para o passo indutivo à uma adição e duas chamadas recursivas. Assim, a equação de recorrência pode ser definida como:

$$T(n) = \begin{cases} 0, n \leq 1 \\ T(n-1) + T(n-2) + 1,\ n > 1 \end{cases}$$

Busca Sequencial: Na busca sequencial recursiva a operação de interesse é a comparação entre elementos do arranjo. Para o caso base não há comparações entre elementos. No passo indutivo o algoritmo faz uma comparação, e no **pior caso** (não acha o elemento no arranjo) ocorrem n-1 chamadas recursivas, podendo determinar a equação de recorrência como:

$$T(n) = \begin{cases} 0, n = 0 \\ T(n-1) + 1,\ n \geq 1 \end{cases}$$

Busca binária: Assim como na busca sequencial, o custo da busca binária é definido pela quantidade de comparações. No caso base não há elementos disponíveis para se procurar, portanto o custo da operação é 0. No **pior caso** do passo indutivo são realizadas sempre duas comparações ($valor > A[meio]$ seguido de $valor < A[meio]$), além da quantidade de chamadas recursivas igual à $(n-1)/2$ (metade dos índices sendo considerados válidos em cada chamada da função, com índices que varia no intervalo $[0, n-1]$). A equação de recorrência é definida por:

$$T(n) = \begin{cases} 0, n = 0 \\ T(\frac{n-1}{2}) + 2,\ n \geq 1 \end{cases}$$

5.2 Resolução de Equações de Recorrência

Comum uma equação de recorrência definida, podemos contar o número de operações que serão feitas no pior caso por meio de algumas estratégias. Esta seção abordará diferentes equações de recorrência que podem ser resolvidas por uma abordagem metódica.

5.3 Exercícios de resolução de equações de recorrência

I - $T(n) = \begin{cases} 1, n = 1 \\ 2 * T(n-1),\ n > 1 \end{cases}$

Procuraremos uma maneira de $T(n)$ para n maior que 1 ($2*T(n-1)$) ser reduzido à um valor conhecido, dado por $T(1)$. Primeiro devemos notar que $2*T(n-1)$ diz que a recursão funciona indo de 1 em 1 passo (n-1).

Tentaremos então encontrar um padrão para os próximos passo da equação de recorrência (T(n-1), T(n-2) etc):

$T(n-1) = 2T(n-1-1) = 2T(n-2)$
$T(n-2) = 2T(n-2-1) = 2T(n-3)$
$T(n-3) = 2T(n-3-1) = 2T(n-4)$
[...]

Já podemos identificar um padrão acontecendo por aqui, mas para resolver a equação devemos substituir esses valores encontrados na equação de recorrência $T(n)$ com $n > 1$:

$T(n) = 2T(n-1)$
$T(n) = 2(2T(n-2)) = 2^2T(n-2)$
$T(n) = 2^2(2T(n-3)) = 2^3T(n-3)$
$T(n) = 2^3(2T(n-4)) = 2^4T(n-4)$
[...]
$T(n) = 2^iT(n-i)$
Lembre que $T(1) = 1$, e perceba que $T(n-i) = T(1)$ se $i = n-1$:
$T(n) = 2^iT(n-i)$ (mesma equação)
$T(n) = 2^{n-1}T(n-(n-1))$ (substituindo i por n-1)
$T(n) = 2^{n-1}T(1))$
$T(n) = 2^{n-1}$

Assim, demonstramos que :
$T(n) = 2T(n-1) = 2^{n-1}$ para $n > 1$

II - $T(n) = \begin{cases} 1, n = 1 \\ 2 * T(n-1) + 1,\ n > 1 \end{cases}$

Notando que $2*T(n-1)+1$ varia os passos de 1 em 1, manipularemos a equação de recorrência para diferentes dependências de n:
$T(n) = 2 * T(n-1) + 1$
$T(n-1) = 2*T(n-1-1)+1 = 2T(n-2)+1$
$T(n-2) = 2*T(n-2-1)+1 = 2T(n-3)+1$
$T(n-3) = 2*T(n-3-1)+1 = 2T(n-4)+1$
[...]

Aplicaremos estes valores em $T(n), n > 1$
$T(n) = 2T(n-1) + 1$
$T(n) = 2(2T(n-2) + 1) + 1$
$T(n) = 2^2T(n-2) + 2^1 + 1$
$T(n) = 2^2(2T(n-3) + 1) + 2^1 + 1$
$T(n) = 2^3(T(n-3)) + 2^2 + 2^1 + 1$
$T(n) = 2^3(2T(n-4) + 1) + 2^2 + 2^1 + 1$
$T(n) = 2^4(T(n-4)) + 2^3 + 2^2 + 2^1 + 1$
$T(n) = 2^4(T(n-4)) + 2^4 - 1$ (Soma de PG)
[...]
$T(n) = 2^i(T(n-i)) + 2^i - 1$
Assim, para $T(n-i) = T(1)$, i deve ser igual à $n-1$:
$T(n) = 2^i(T(n-i)) + 2^i - 1$
$T(n) = 2^{n-1}(T(n-n+1)) + 2^{n-1} - 1$
$T(n) = 2^{n-1}(T(1)) + 2^{n-1} - 1$
$T(n) = 2^{n-1} + 2^{n-1} - 1$
$T(n) = 2 * 2^{n-1} - 1$
$T(n) = 2^n - 1$

Assim, demonstramos que :
$T(n) = 2 * T(n-1) + 1 = 2^n - 1$ para $n > 1$

III - $T(n) = \begin{cases} 1, n = 1 \\ 2 * T(\frac{n}{2}),\ n > 1 \end{cases}$

Note que a equação de recorrência para $n > 1$ decai em frações de 2 (1/2, 1/4, 1/8 etc). Manipulando algumas equações de recorrência para n maior que 1, observamos um padrão:
$T(n) = 2 * T(\frac{n}{2})$
$T(n/2) = 2 * T(\frac{n}{2}/2) = 2 * T(\frac{n}{2^2})$
$T(n/4) = 2 * T(\frac{n}{2}/4) = 2 * T(\frac{n}{2^3})$
$T(n/8) = 2 * T(\frac{n}{2}/8) = 2 * T(\frac{n}{2^4})$

Manipulando agora a equação de recorrência $T(n)$:
$T(n) = 2 * T(\frac{n}{2})$
$T(n) = 2(2 * T(\frac{n}{2^2})) = 2^2 * T(\frac{n}{2^2})$
$T(n) = 2^2(2 * T(\frac{n}{2^3})) = 2^3 * T(\frac{n}{2^3})$
$T(n) = 2^3(2 * T(\frac{n}{2^4})) = 2^4 * T(\frac{n}{2^4})$
[...]
$T(n) = 2^i * T(\frac{n}{2^i})$

Como queremos $T(\frac{n}{2^i}) = 1$:

$\frac{n}{2^i} = 1$
$n = 2^i$
$\log_2 n = i$
Aplicando na equação de recorrência:
$T(n) = 2^i * T(\frac{n}{2^i})$
$T(n) = 2^{\log_2 n} * T(1)$
$T(n) = 2^{\log_2 n}$
$T(n) = n$ (por propriedade $b^{\log_b M} = M$)

Assim, demonstramos que :
$T(n) = 2 * T(\frac{n}{2}) = n$ para $n > 1$

IV - $T(n) = \begin{cases} 1, n = 1 \\ T(n-1) + 1,\ n > 1 \end{cases}$

Note que a equação de recorrência para $n > 1$ segue passos de 1 em 1 ($T(n-1)$). Começaremos manipulando a equação $T(n)$ para n maior que 1 com parâmetros n-1, n-2 e n-3.
$T(n) = T(n-1) + 1$
$T(n-1) = T(n-1-1) + 1 = T(n-2) + 1$
$T(n-2) = T(n-2-1) + 1 = T(n-3) + 1$
$T(n-3) = T(n-3-1) + 1 = T(n-4) + 1$
[...]

Conhecendo esses valores, manipularemos $T(n)$ buscando remover a dependência da recorrência para chegarmos em uma equação mais simples.
$T(n) = T(n-1) + 1$
$T(n) = T(n-2) + 1 + 1 = T(n-2) + 2$
$T(n) = T(n-3) + 1 + 2 = T(n-3) + 3$
$T(n) = T(n-4) + 1 + 3 = T(n-4) + 4$
[...]

$T(n) = T(n-i) + i$
Para igualar $T(n-i)$ à 1, i deve ser igual à $n-1$
$T(n) = T(n-i) + i)$
$T(n) = T(n-n+1) + n - 1$
$T(n) = T(1) + n - 1$
$T(n) = 1 + n - 1$
$T(n) = n$
Assim, demonstramos que :
$T(n) = T(n-1) + 1 = n$ para $n > 1$

V - $T(n) = \begin{cases} 0, n = 1 \\ T(n-1) + n - 1,\ n > 1 \end{cases}$

Note que a equação de recorrência para $n > 1$ segue passos de 1 em 1 ($T(n-1)$). Começaremos manipulando a equação $T(n)$ para n maior que 1 com parâmetros n-1, n-2 e n-3.
$T(n) = T(n-1) + n - 1$
$T(n-1) = T(n-1-1)+n-1 = T(n-2)+n-1$
$T(n-2) = T(n-2-1)+n-1 = T(n-3)+n-1$
$T(n-3) = T(n-3-1)+n-1 = T(n-4)+n-1$
[...]

Conhecendo esses valores, manipularemos $T(n)$ buscando remover a dependência da recorrência para chegarmos em uma equação mais simples.
$T(n) = T(n-1) + n - 1$
$T(n) = (T(n-2) + n - 1) + n - 1$
$T(n) = T(n-2) + 2n - 2$
$T(n) = T(n-3) + n - 1 + 2n - 2$
$T(n) = T(n-3) + 3n - 3$
$T(n) = T(n-4) + n - 1 + 3n - 3$
$T(n) = T(n-4) + 4n - 4$
[...]

$T(n) = T(n-i) + in - i$
Como queremos eliminar $T(n-i)$, $n-i$ deve ser igual à 0, pois conhecemos $T(0)$. Obviamente, n deve ser igual à i para a subtração resultar em zero.
$T(n) = T(n-i) + in - i$
$T(n) = T(n-n) + n * n - n$
$T(n) = 0 + n * n - n$
$T(n) = n^2 - n$