MC102 – Aula 27
Recursão II

Prof. Luiz F. Bittencourt

Turmas QR

Instituto de Computação – Unicamp

2019

Conteúdo adaptado de slides fornecidos pelo Prof. Eduardo Xavier.

Roteiro

Recursão - Relembrando

- Definições recursivas de funções são baseadas no *princípio matemático da indução* que vimos anteriormente.
- A idéia é que a solução de um problema pode ser expressa da seguinte forma:
  - Definimos a solução para os casos básicos;
  - Definimos como resolver o problema geral utilizando soluções do mesmo problema só que para casos menores.

Cálculo de Potências

Suponha que temos que calcular x^n para n inteiro positivo. Como calcular de forma recursiva?
x^n é:

- 1 se $n = 0$.
- xx^{n-1} caso contrário.

Cálculo de Potências

```
def pot(x, n):
  if(n == 0):
    return 1
  else:
    return x*pot(x,n-1)
```

Cálculo de Potências

Neste caso a solução iterativa é mais eficiente.

```
def pot2(x, n):
  p=1
  for i in range(1, n+1):
    p = p * x
  return p
```

- O laço é executado *n* vezes.
- Na solução recursiva são feitas *n* chamadas recursivas, mas tem-se o custo adicional para criação/remoção de variáveis locais na pilha.

Cálculo de Potências

Mas e se definirmos a potência de forma diferente?

x^n é:

- Caso básico:
  - Se $n = 0$ então $x^n = 1$.
- Caso Geral:
  - Se $n > 0$ e é par, então $x^n = (x^{n/2})^2$.
  - Se $n > 0$ e é ímpar, então $x^n = x(x^{(n-1)/2})^2$.

Note que aqui também definimos a solução do caso maior em termos de casos menores.

Cálculo de Potências

Este algoritmo é mais eficiente do que o iterativo. Por que? Quantas chamadas recursivas o algoritmo pode fazer?

```
def pot3(x, n):
  if(n == 0):
    return 1

  elif(n%2 == 0): #se n é par
    aux = pot3(x, n//2)
    return aux * aux

  else: #se n é impar
    aux = pot3(x, (n-1)//2)
    return x * aux * aux
```

Cálculo de Potências

- No algoritmo anterior, a cada chamada recursiva o valor de n é dividido por 2. Ou seja, a cada chamada recursiva, o valor de n decai para pelo menos a metade.
- Usando divisões inteiras faremos no máximo $\lceil(\log_2 n)\rceil + 1$ chamadas recursivas.
- Enquanto isso, o algoritmo iterativo executa o laço n vezes.

Torres de Hanoi

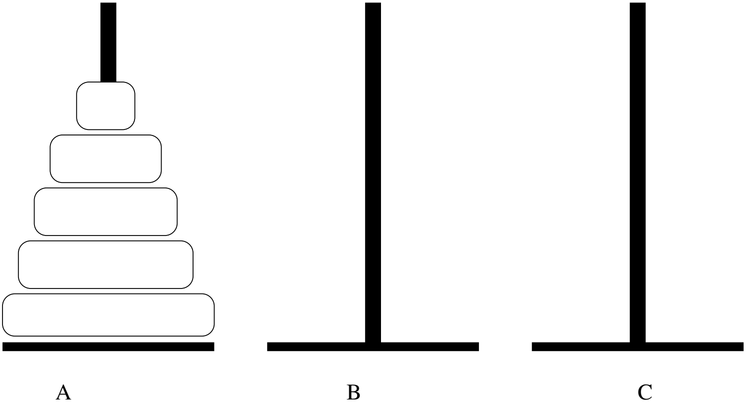
- Problema inventado pelo matemático francês Édouard Lucas em 1883.
- Também conhecido como Torres de Brahma.

No grande templo de Brahma em Benares, numa bandeja de metal sob a cúpula que marca o centro do mundo, três agulhas de diamante servem de pilar a sessenta e quatro discos de ouro puro. Incasavelmente, os sacerdotes transferem os discos um de cada vez, de agulha para agulha, obedecendo sempre à lei imutável de Brahma: nenhum disco se poderá sobrepor a um menor.

Torres de Hanoi

No início do mundo, todos os sessenta e quatro discos de ouro foram dispostos na primeira das três agulhas, constituindo a Torre de Brahma. No momento em que o menor dos discos for colocado de tal modo que se forme uma vez mais a Torre de Brahma numa agulha diferente da inicial, tanto a torre como o templo serão transformados em pó e o ribombar de um trovão assinalará o fim do mundo.

Torres de Hanoi



Torres de Hanoi

- Inicialmente temos 5 discos de diâmetros diferentes na estaca A.
- O problema das torres de Hanoi consiste em transferir os cinco discos da estaca A para a estaca C (pode-se usar a estaca B como auxiliar).
- Porém deve-se respeitar as seguintes regras:
  - Apenas o disco do topo de uma estaca pode ser movido.
  - Nunca um disco de diâmetro maior pode ficar sobre um disco de diâmetro menor.

Torres de Hanoi

- Vamos considerar o problema geral onde há n discos.
- Vamos usar indução para obtermos um algoritmo para este problema.

Torres de Hanoi

Teorema

É possível resolver o problema das torres de Hanoi com n discos.

Prova.

- Base da Indução: $n = 1$. Neste caso temos apenas um disco. Basta mover este disco da estaca A para a estaca C.
- Hipótese de Indução: Sabemos como resolver o problema quando há $n - 1$ discos.

$\square$

Torres de Hanoi

Prova.

- Passo de Indução: Devemos resolver o problema para n discos assumindo que sabemos resolver o problema com $n - 1$ discos.
  - Por hipótese de indução sabemos mover os $n - 1$ primeiros discos da estaca **A** para a estaca **B** usando a estaca **C** como auxiliar.
  - Depois de movermos estes $n - 1$ discos, movemos o maior disco (que continua na estaca **A**) para a estaca **C**.
  - Novamente pela hipótese de indução sabemos mover os $n - 1$ discos da estaca **B** para a estaca **C** usando a estaca **A** como auxiliar.
- Com isso temos uma solução para o caso onde há n discos.

$\square$

Torres de Hanoi: Passo de Indução

- A indução nos fornece um algoritmo e ainda por cima temos uma demonstração formal de que ele funciona!

Torres de Hanoi: Algoritmo

Problema: Mover n discos de **A** para **C**.

1. Se $n = 1$ então mova o único disco de **A** para **C** e pare.
2. Caso contrário ($n > 1$) desloque de forma recursiva os $n - 1$ primeiros discos de **A** para **B**, usando **C** como auxiliar.
3. Mova o último disco de **A** para **C**.
4. Mova, de forma recursiva, os $n - 1$ discos de **B** para **C**, usando **A** como auxiliar.

Torres de Hanoi: Algoritmo

- A função que computa a solução em Python terá o seguinte protótipo:

```
def hanoi(n, estacaIni, estacaFim, estacaAux):
```

- São passados como parâmetros:
  - o número de discos a ser movido (**n**),
  - um caracter indicando de onde os discos serão movidos (**estacaIni**)
  - um caracter indicando para onde devem ser movidos (**estacaFim**); e
  - um caracter indicando qual é a estaca auxiliar (**estacaAux**).

Torres de Hanoi: Algoritmo

A função que computa a solução é:

```
def hanoi(n, estacaIni, estacaFim, estacaAux):
  if(n==1):
    #Caso base. Move único disco do Ini para Fim
    print("Mova disco %d da estaca %c para %c." %(n, estacaIni, estacaFim))
  else:
    #Move n-1 discos de Ini para Aux com Fim como auxiliar
    hanoi(n-1, estacaIni, estacaAux, estacaFim)

    #Move maior disco para Fim
    print("Mova disco %d da estaca %c para %c." %(n, estacaIni, estacaFim))

    #Move n-1 discos de Aux para Fim com Ini como auxiliar
    hanoi(n-1, estacaAux, estacaFim, estacaIni)
```

Torres de Hanoi: Algoritmo

```python
def main():
  hanoi(4, 'A', 'C' , 'B')

#Discos são numerados de 1 até n

def hanoi(n, estacaIni, estacaFim, estacaAux):
  if(n==1):
    #Caso base. Move único disco do Ini para Fim
    print("Mova disco %d da estaca %c para %c." %(n, estacaIni, estacaFim))
  else:
    #Move n-1 discos de Ini para Aux com Fim como auxiliar
    hanoi(n-1, estacaIni, estacaAux, estacaFim)

    #Move maior disco para Fim
    print("Mova disco %d da estaca %c para %c." %(n, estacaIni, estacaFim))

    #Move n-1 discos de Aux para Fim com Ini como auxiliar
    hanoi(n-1, estacaAux, estacaFim, estacaIni)

main()
```

Torres de Hanoi

Seja $T(n)$ o número de movimentos necessários para resolver o problema com n discos.

Pelo algoritmo anterior, $T(n)$ é dado pela seguinte recursão:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T(n-1) + 1 & \text{se } n \geq 1 \end{cases}$$

Fazendo alguns cálculos, chegamos à solução seguinte:

$T(n) = 2^n - 1$, para $n \geq 1$.

Quanto tempo para resolver o problema com 64 discos?

Torres de Hanoi

Quanto tempo para resolver o problema com 64 discos? Suposições:

- Sacerdotes são semi-deuses que movem os discos na velocidade da luz (≈300.000.000 m/s) (lembre-se: só Chuck Norris consegue mover objetos mais rápido que a velocidade da luz.).
- 3 torres a 67.33 cm uma da outra. Distância média dos movimentos: $\frac{3*0.6733}{2} = 1.00995 \approx 1.0099$m.
- Discos com altura mínima, tendendo a zero $\Rightarrow$ tamanho da pilha tende a zero $\Rightarrow$ Tempo para tirar o disco da pilha tende a zero.

$R = (2^{64}) \times (\frac{1.0099}{300000000})/60/60/24/365$

Recursão e Backtracking

- Muitos problemas podem ser resolvidos enumerando-se de forma sistemática todas as possibilidades de arranjos que formam uma solução para um problema.
- Vimos em aulas anteriores o seguinte exemplo: Determinar todas as soluções inteiras de um sistema linear como:

$$x_1 + x_2 + x_3 = C$$

com $x_1 \geq 0$, $x_2 \geq 0$, $C \geq 0$ e todos inteiros.

```
Para cada possível valor de x1 entre 0 e C
    Para cada possível valor de x2 entre 0 e C-x1
                  Faça  x3 = C - (x1 + x2)
                  Imprima solução x1 + x2 + x3 = C
```

Recursão e Backtracking

Abaixo temos o código de uma solução para o problema com $n = 3$ variáveis e constante C passada como parâmetro.

```
def solution(C):
  for x1 in range(0, C+1):
    for x2 in range(0,C-x1+1):
      x3 = C -x1 -x2
      print("%d + %d + %d = %d" %(x1, x2, x3, C))
```

Recursão e Backtracking

Como resolver este problema para o caso geral, onde n e C são parâmetros?

$$x_1 + x_2 + \ldots + x_{n-1} + x_n = C$$

- A princípio deveríamos ter $n - 1$ laços encaixados.
- Mas não sabemos o valor de n. Só saberemos durante a execução do programa.

Recursão e Backtracking

- A técnica de recursão pode nos ajudar a lidar com este problema:
  - Construir uma função com um único laço e que recebe uma variável k como parâmetro.
  - A variável k indica que estamos setando os possíveis valores de x_k.
  - Para cada valor de x_k devemos setar o valor de x_{k+1} de forma recursiva!
  - Se $k == n$ basta setar o valor da última variável.

Recursão e Backtracking

```
função solution(n, C, k){
  Se k == n Então
     x_n = C - x_1 - ... - x_{n-1}
     Imprima solução
  Senão
     Para cada valor V entre 0 e (C - x_1 - ... - x_{k-1}) faça
        x_k = V
        solution(n, C, k+1) //Vamos setar os possíveis valores da var. x seguinte
}
```

Recursão e Backtracking

- Em Python teremos uma função com o seguinte protótipo:

```python
def solution(n, C, k, R, x):
```

- A variável R terá o valor da constante C menos os valores já setados para variáveis em chamadas recursivas anteriores, i.e, $R = C - x_1 - \ldots - x_{k-1}$.
- A lista x corresponde aos valores das variáveis.
  - Lembre-se que em Python a lista começa na posição 0, por isso as variáveis serão $x[0], \ldots, x[n-1]$.

Recursão e Backtracking

- Primeiramente temos o caso de parada (quando $k == n - 1$):

```
def solution(n, C, k, R, x):
  if(k == n-1):
    #imprimindo a solução
    for i in range(0, n-1):
      print("%d + " %x[i], end="")
    print("%d = %d" %(R, C))
  .
  .
  .
}
```

Recursão e Backtracking

- A função completa é:

```
def solution(n, C, k, R, x):
  if(k == n-1):
    #imprimindo a solução
    for i in range(0, n-1):
      print("%d + " %x[i], end="")
    print("%d = %d" %(R, C))

  else:
    #seta cada possível valor de x[k] e faz recursão
    for x[k] in range(0, R+1):
      solution(n, C, k+1, R-x[k], x)
```

A chamada inicial da função deve ter $k = 0$.

Recursão e Backtracking

```
import sys

def main():
  if(len(sys.argv) != 3):
    print("Execute informando n (num. de vari.) e C (constante int. positiva)")
  else:
    n = int(sys.argv[1])
    C = int(sys.argv[2])
    x = [0 for i in range(n)]
    solution(n, C, 0, C, x)

def solution(n, C, k, R, x):
  if(k == n-1):
    #imprimindo a solução
    for i in range(0, n-1):
      print("%d + " %x[i], end="")
    print("%d = %d" %(R, C))

  else:
    #seta cada possível valor de x[k] e faz recursão
    for x[k] in range(0, R+1):
      solution(n, C, k+1, R-x[k], x)

main()
```

Exercício

- Defina de forma recursiva a busca binária.
- Escreva um algoritmo recursivo para a busca binária.

Exercício

- Escreva um programa que lê uma string do teclado e então imprime todas as permutações desta palavra. Se por exemplo for digitado "abca" o seu programa deveria imprimir (não necessariamente nesta ordem):

  aabc aacb abac abca acab acba baac baca bcaa caab caba cbaa