Python: Recursão

Claudio Esperança

Recursão

- É um princípio muito poderoso para construção de algoritmos
- A solução de um problema é dividido em
  - Casos simples:
    - São aqueles que podem ser resolvidos trivialmente
  - Casos gerais:
    - São aqueles que podem ser resolvidos compondo soluções de casos mais simples
- Semelhante à prova de teoremas por indução
  - Casos simples: O teorema é verdadeiro trivialmente
  - Casos genéricos: são provados assumindo-se que todos os casos mais simples também são verdadeiros

Função recursiva

- Implementa um algoritmos recursivo onde a solução dos casos genéricos requerem chamadas à própria função
- Uma função recursiva é a maneira mais direta (mas não necessariamente a melhor) de se resolver problemas de natureza recursiva ou para implementar estruturas de dados recursivas
- Considere, por exemplo, a definição da seqüência de Fibonacci:
  - O primeiro e o segundo termo valem 0 e 1, respectivamente
  - O *i*-ésimo termo é a soma do (*i*-1)-ésimo e o (*i*-2)-ésimo termo

```
>>> def fib(i):
        if i==1: return 0
        elif i==2: return 1
        else: return fib(i-1)+fib(i-2)
>>> for i in range(1,11):
        print fib(i),
0 1 1 2 3 5 8 13 21 34
```

Exemplo: Busca binária

- Um exemplo clássico de recursão é o algoritmo conhecido como busca binária que é usado para pesquisar um *valor* em uma *lista* ordenada
- Chamemos de `imin` e `imax` os índices mínimo e máximo da lista onde a busca será feita
  - Inicialmente, `imin = 0` e `imax = len(`*lista*`)-1`
- O caso base corresponde a `imin == imax`
  - Então, ou o *valor* é igual a *lista* `[imin]` ou não está na lista
- Senão, podemos dividir o intervalo de busca em dois
  - Seja `meio = (imin+imax)/2`
  - Se o *valor* é maior que *lista* `[meio]`, então ele se encontra em algum dos índices entre `meio+1` e `imax`
  - Caso contrário, deve se encontrar em algum dos índices

Busca binária: implementação

```
def testa(lista,valor):
    def busca_binaria(imin,imax):
        if imin==imax: return imin
        else:
            meio=(imax+imin)/2
            if valor>lista[meio]:
                return busca_binaria(meio+1,imax)
            else:
                return busca_binaria(imin,meio)
    i = busca_binaria(0,len(lista)-1)
    if lista[i]==valor:
        print valor,"encontrado na posicao",i
    else:
        print valor,"nao encontrado"

>>> testa([1,2,5,6,9,12],3)
3 nao encontrado
>>> testa([1,2,5,6,9,12],5)
5 encontrado na posicao 2
```

Recursão infinita

- Assim como nos casos dos laços de repetição, é preciso cuidado para não escrever funções infinitamente recursivas
  - Ex.:

```
def recursiva(x):
    if f(x): return True
    else: return recursiva(x)
```

- Uma função recursiva tem que
  - Tratar *todos* os casos básicos
  - Usar recursão apenas para tratar casos *garantidamente mais simples* do que o caso corrente
  - Ex.:

```
def recursiva(x):
    if f(x): return True
    elif x==0: return False
    else: return recursiva(x-1)
```

Eficiência de funções recursivas

- Quando uma função é chamada, um pouco de memória é usado para guardar o ponto de retorno, os argumentos e variáveis locais
- Assim, soluções iterativas são normalmente mais eficientes do que soluções recursivas equivalentes
- Isto não quer dizer que soluções iterativas sempre sejam preferíveis a soluções recursivas
- Se o problema é recursivo por natureza, uma solução recursiva é mais clara, mais fácil de programar e, freqüentemente, mais eficiente

Pensando recursivamente

- Ao invés de pensar construtivamente para para obter uma solução, às vezes é mais simples pensar em termos de uma prova indutiva
- Considere o problema de testar se uma lista *a* é uma permutação da lista *b*
  - Caso básico: *a* é uma lista vazia
    - Então *a* é permutação de *b* se *b* também é uma lista vazia
  - Caso básico: *a*[0] não aparece em *b*
    - Então *a* não é uma permutação de *b*
  - Caso genérico: *a*[0] aparece em *b* na posição *i*
    - Então *a* é permutação de *b* se *a*[1:] é uma permutação de *b* do qual foi removido o elemento na posição *i*

Exemplo: Testa permutações

```
def e_permutacao(a,b):
    """
    Retorna True sse a lista a é uma
    permutação da lista b
    """
    if len(a) == 0 : return len(b)==0
    if a[0] in b:
        i = b.index(a[0])
        return e_permutacao(a[1:],b[0:i]+b[i+1:])
    return False

>>> e_permutacao([1,2,3],[3,2,1])
True
>>> e_permutacao([1,2,3],[3,3,1])
False
>>> e_permutacao([1,2,3],[1,1,2,3])
False
>>> e_permutacao([1,1,2,3],[1,2,3])
False
```

Estruturas de dados recursivas

- Há estruturas de dados que são inerentemente recursivas, já que sua própria definição é recursiva
- Por exemplo, uma lista pode ser definida recursivamente:
  - [] é uma lista (vazia)
  - Se A é uma lista e x é um valor, então A+[x] é uma lista com x como seu último elemento
- Esta é uma definição construtiva, que pode ser usada para escrever funções que criam listas
- Uma outra definição que pode ser usada para analisar listas é:
  - Se L é uma lista, então:
    - L == [] , ou seja, L é uma lista vazia, ou
    - x = L.pop() torna L uma lista sem seu último elemento x
  - Esta definição não é tão útil em Python já que o comando

Exemplo: Subseqüência

```
def e_subseq(a,b):
    """ Retorna True sse a é subseqüência de b,
    isto é, se todos os elementos a[0..n-1] de a
    aparecem em b[j(0)], b[j(1)]... b[j(n-1)]
    onde j(i)<j(i+1) """
    if a == []:
        # Lista vazia é subseqüência de qq lista
        return True
    if a[0] not in b:
        return False
    return e_subseq (a[1:], b[b.index(a[0])+1:])
```

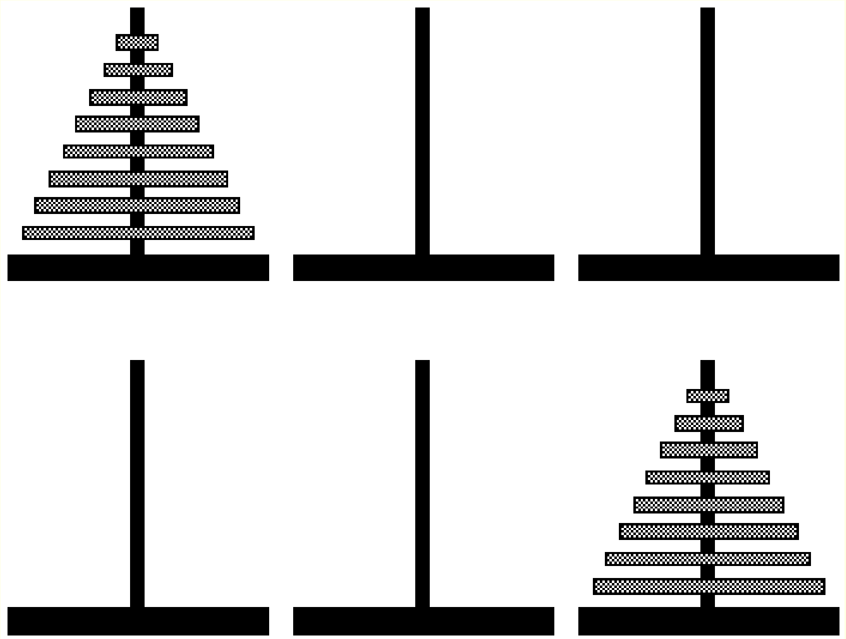
Encontrando a recorrência

- Alguns problemas não se apresentam naturalmente como recursivos, mas pensar recursivamente provê a solução
- Tome o problema de computar todas as permutações de uma lista
  - Assumamos que sabemos computar todas as permutações de uma lista sem seu primeiro elemento *x*
    - Seja perm uma dessas permutações
    - Então, a solução do global contém todas as listas obtidas inserindo x em todas as possíveis posições de perm

Exemplo: computar todas as permutações de uma lista

```
def permutacoes(lista):
    """ Dada uma lista, retorna uma lista de listas,
    onde cada elemento é uma permutação da lista
    original """
    if len(lista) == 1:  # Caso base
        return [lista]
    primeiro = lista[0]
    resto = lista [1:]
    resultado = []
    for perm in permutacoes(resto):
        for i in range(len(perm)+1):
            resultado += \
               [perm[:i]+[primeiro]+perm[i:]]
    return resultado
```

Torres de Hanói



- Jogo que é um exemplo clássico de problema recursivo
- Consiste de um tabuleiro com 3 pinos no qual são encaixados discos de tamanho decrescente
- A idéia é mover os discos de um pino para outro sendo que:
  - Só um disco é movimentado por vez
  - Um disco maior nunca pode ser posto sobre um menor

Torres de Hanói: Algoritmo

- A solução é simples se supusermos existir um algoritmo capaz de mover todos os discos menos um do pino de origem para o pino sobressalente
- O algoritmo completo para mover *n* discos do pino de origem *A* para o pino de destino *B* usando o pino sobressalente *C* é
  - Se *n* é 1, então a solução é trivial
  - Caso contrário,
    - Usa-se o algoritmo para mover *n*-1 discos de *A* para *C* usando *B* como sobressalente
    - Move-se o disco restante de *A* para *B*
    - Usa-se o algoritmo para mover *n*-1 discos de C para *B* usando *A* como sobressalente

Torres de Hanói: Implementação

```
def hanoi(n,origem,destino,temp):
    if n>1: hanoi(n-1,origem,temp,destino)
    mover(origem,destino)
    if n>1: hanoi(n-1,temp,destino,origem)

def mover(origem,destino):
     print “Mover de“, origem, “para”,\
      “destino”
```

- Com um pouco mais de trabalho, podemos redefinir a função mover para que ela nos dê uma representação “gráfica” do movimento dos discos

Torres de Hanói: Exemplo

```
   |       |       |
   *       |       |
  ***      |       |
 *****     |       |
=======================
   |       |       |
   |       |       |
  ***      |       |
 *****     *       |
=======================
   |       |       |
   |       |       |
   |       |       |
 *****     *      ***
=======================
   |       |       |
   |       |       |
   |       |       *
 *****     |      ***
=======================

   |       |       |
   |       |       |
   |       |       *
   |     *****    ***
=======================
   |       |       |
   |       |       |
   |       |       |
   *     *****    ***
=======================
   |       |       |
   |       |       |
   |      ***      |
   *     *****     |
=======================
   |       |       |
   |       *       |
   |      ***      |
   |     *****     |
=======================
```