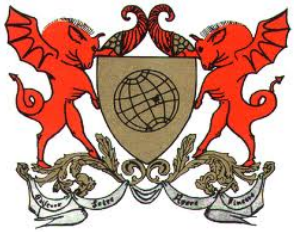
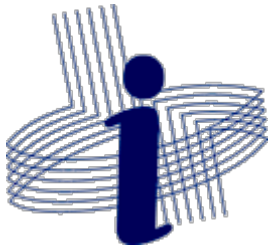


Universidade Federal de Viçosa
Departamento de Informática
Centro de Ciências Exatas e Tecnológicas



INF101 – Introdução à Programação II

Funções Recursivas

Recursividade

- A recursividade é um mecanismo poderoso de expressão em matemática
- Na realidade, a ideia de computabilidade atualmente aceita é fundamentada na recursividade
- Em programação imperativa, a recursividade está implícita na iteração
- Mesmo assim, certos problemas têm solução expressa de forma mais simples por meio da recursividade do que por meio da iteração

Funções Recursivas

- A recursividade em programação imperativa é expressa explicitamente por meio de funções recursivas
- Uma função é recursiva se ela chama a si própria dentro do corpo de sua definição
- Em geral, toda função explicitamente recursiva tem uma forma equivalente não recursiva usando iteração
- Nos computadores atuais, devido a sua arquitetura, em geral, as funções apenas iterativas são implementadas de maneira mais eficiente

Funções Recursivas

- Mesmo assim, para implementarmos a solução computacional de certos problemas, vamos preferir usar funções recursivas
- Isto por causa da simplicidade de apresentar a solução e por não haver perda de eficiência
- Em particular, os algoritmos de manipulação de árvore binária são bem implementados por meio de recursividade
- Problemas cuja solução usa a técnica de retrocesso (*backtracking*) também têm solução recursiva mais simples


Exemplos

- A função fatorial
  - A definição do fatorial é um exemplo clássico do uso de recursividade
  - Com efeito:

$$0! = 1$$

$$n! = n(n-1)!, \text{ se } n > 0$$

Exemplos

- Implementação do fatorial por meio de uma função recursiva

```
def fat(n):
    if n <= 1:
        return 1
    else:
        return n * fat(n-1)
```


Exemplos

- A seqüência de Fibonacci
  - Em INF100, vimos a implementação do cálculo dos números de Fibonacci por meio iterativo
  - Por definição, os elementos da sequência são dados por:

$$f_0 = 0, \quad f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}, \quad \text{se } n > 1$$

Exemplos

- Implementação do cálculo dos números de Fibonacci por meio de uma função recursiva

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```


Exemplos

- Máximo divisor comum
  - O mdc tem uma definição recursiva muito interessante
  - De fato:

$$\text{mdc}(x, y) = \text{mdc}(y \bmod x, x), \text{ se } x > 0$$

$$\text{mdc}(0, y) = y$$

Exemplos

- Implementação do cálculo do mdc como uma função recursiva

```
def mdc(x, y):
    if x > 0:
        return mdc(y % x, x)
    else:
        return y
```


Exemplos

- Torre de Hanoi
  - Diz a lenda que, em um templo no extremo oriente, monges estão tentando mover uma pilha de discos de marfim de um pino de ouro a outro também de ouro, usando um terceiro pino como auxiliar
  - A pilha inicial tinha 64 discos, todos de tamanhos diferentes, colocados em um dos pinos de ouro e dispostos do fundo para o topo em tamanhos decrescentes
  - Os monges estão movendo os discos desse pino para um segundo sob a condição de somente um pino ser movido de cada vez e, hora nenhuma, um disco maior pode ser colocado sobre um menor

Exemplos

- Torre de Hanoi (cont.)
  - Um terceiro pino é usado como auxiliar
  - Ainda, segundo a lenda, o mundo vai acabar quando os monges completarem a tarefa, portanto ninguém é incentivado a ajudar os monges nessa tarefa

Exemplos

- Torre de Hanoi (cont.)
  - Para 3 discos apenas, os movimentos necessários para passar do pino 1 para o 3 usando o pino 2 como auxiliar são no mínimo:
    - 1 ➔ 3
    - 1 ➔ 2
    - 3 ➔ 2
    - 1 ➔ 3
    - 2 ➔ 1
    - 2 ➔ 3
    - 1 ➔ 3


Torre de Hanoi

- Vamos refinar o pseudocomando:

  Determine os movimentos necessários para transferir n *discos do pino* 1 *para o pino* 3*, usando o pino* 2 *como intermediário.*

  que resolve inteiramente o problema da Torre de Hanoi

- Como sugerido, vamos usar a recursividade como técnica de solução do problema

Torre de Hanoi

- Base
  - Se *n* = 0 disco, o problema já está resolvido; não precisa fazer nada.
- Indução
  - Para *n* > 0 discos,
    - Mova *n* – 1 discos do pino de *origem* para o pino *auxiliar*, usando o pino de *destino* como área de armazenamento temporário.
    - Mova o *n*-ésimo disco do pino de *origem* para o pino de *destino*.
    - Mova *n* – 1 discos do pino *auxiliar* para o pino de *destino*, usando o pino de *origem* como área de armazenamento temporário.

Torre de Hanoi

- Vamos implementar o algoritmo como uma função recursiva em Python:

```
def mova(n, origem, destino, aux):
    if n > 0:
        mova(n-1, origem, aux, destino)
        print(origem, '->', destino)
        mova(n-1, aux, destino, origem)
```


Torre de Hanoi

```
# hanoi.py
# 23/06/2016
# Calcula os movimentos necessários para o jogo Torre de Hanoi,
# com n discos, n >= 0, e 3 pinos: origem, destino e aux.

def mova(n, origem, destino, aux):
    if n > 0:
        mova(n-1, origem, aux, destino)
        print(origem, "->", destino)
        mova(n-1, aux, destino, origem)

num_discos = int(input("Forneça o número de discos a ser movidos " +
                       "na Torre de Hanoi: "))
if num_discos >= 0:
    print("Para", num_discos, "discos, os movimentos necessários " +
          "são:")
    mova(num_discos, 1, 3, 2)
else:
    print("Número inválido de discos!")
```

Torre de Hanoi

- Análise do desempenho do algoritmo
- Vamos considerar como operação preponderante a de mover um disco. Logo o número de movimentos M_n é dado pela seguinte equação recorrente:

$$M_n = M_{n-1} + 1 + M_{n-1}$$
$$= 2M_{n-1} + 1$$

- Resolvendo a equação recorrente, sabendo-se que $M_1 = 1$, obtemos:

$$M_n = 2^n - 1$$


Torre de Hanoi

- Análise do desempenho do algoritmo (cont.)
  - Para *n* = 3 discos, o número de movimentos é 7
  - Para *n* = 5 discos, o número de movimentos é 31
  - Para *n* = 10 discos, o número de movimentos é 1023
  - Para *n* = 64 discos, segundo a lenda, o número de movimentos é $2^{64} - 1$. Supondo que os monges são rápidos e conseguem movimentar cada disco em 1 seg. Qual é o tempo que gastarão? Lembre-se de que, após terminarem os movimentos, o mundo vai acabar!!!

Torre de Hanoi

- Análise do desempenho do algoritmo (cont.)
  - $2^{64} - 1 = 18.446.744.073.709.551.615$
  - $(2^{64} - 1)$ seg $\approx$ 584.942.417.355 anos
  - Ufa!!! Ainda bem!!!
  - Até um computador demoraria muito tempo:
    - $(2^{64} - 1) \times 10^{-8}$ seg $\approx$ 5849,42 anos

Ordenação por Inserção Direta

- Outro algoritmo simples de ordenação de listas é o de *inserção direta*
- Ele é o método preferido dos jogadores de cartas de baralho quando necessitam ordenar uma mão de cartas: cada carta é inserida no lugar apropriado para manter a ordenação
- Vamos considerar a ordenação crescente para apresentar o método em Python

Ordenação por Inserção Direta

```
def ordene(L):
    if len(L) == 0:
        return []
    else:
        return insira(L[0], ordene(L[1:]))

def insira(x, L):
    if len(L) == 0:
        return [x]
    elif x < L[0]:  # ordem crescente
        return [x] + L
    else:
        return [L[0]] + insira(x, L[1:])
```

Ordenação por Inserção Direta

- O algoritmo não recursivo de inserção direta pode ser implementado assim em Python:

```
def ordene(L):
    for h in range(1, len(L)):
        insira(L, h)

def insira(L, k):
    pos = k - 1
    # Procura o primeiro elemento de L a partir de k-1 <= L[k]
    while pos >= 0 and L[k] < L[pos]:
        pos = pos - 1
    pos = pos + 1
    # Movimenta L[pos:k] para colocar L[k]
    temp = L[k]
    for i in range(k-1, pos-1, -1):
        L[i+1] = L[i]
    L[pos] = temp
```

Mergesort

- O *mergesort* é um dos métodos de ordenação mais eficientes
- Ele baseia-se em dividir a lista a ser ordenada em duas partes: digamos dividi-la ao meio
- Depois ordenamos cada uma das partes
- E, finalmente, intercalamo-las mantendo a ordenação


Mergesort

- O método do mergesort é claramente recursivo; assim sendo podemos escrevê-lo como uma função recursiva em Python:

```
def mergesort(L, inicio, fim):
    if inicio < fim:
        meio = (inicio + fim)//2
        mergesort(L, inicio, meio)
        mergesort(L, meio+1, fim)
        intercale(L, inicio, meio, fim)
```


Mergesort

- Como determinamos o modo de ordenar: crescente ou decrescentemente?
- É pelo algoritmo de intercalação
- Portanto, para fixar as ideias, suponhamos ordem crescente
- A seguir, apresentamos o algoritmo de intercalação para manter ordem crescente
- Fica, de antemão, como exercício, modificá-lo para obter ordem decrescente


Intercalação *(Merging)*

- Em alto nível (de abstração), podemos escrever o algoritmo de intercalação das listas L1 e L2 ordenadas (crescentemente) produzindo a lista final L da seguinte maneira:

1. Inicialize os índices i, j, k. O índice i percorrerá L1, j percorrerá L2 e k percorrerá L.
2. Enquanto houver elementos em L1 e L2, faça:
   2.1. Se L1[i] <= L2[j],
   copie L1[i] em L[k];
   avance L1
   2.2. senão
   copie L2[j] em L[k];
   avance L2
   fim_se;
   2.3. avance L
   fim_enquanto.
3. Verifique qual das duas, L1 ou L2, ainda não acabou;
   e copie o resto da que não acabou em L.

Intercalação *(Merging)*

- Algoritmo de intercalação para ordem crescente já escrito como uma função em Python:

```
def intercale(L, inicio, meio, fim):
    L1 = L[inicio:meio+1]
    L2 = L[meio+1:fim+1]
    i, j, k = 0, 0, inicio
    while i < len(L1) and j < len(L2):
        if L1[i] <= L2[j]:   # determina ordem crescente
            L[k] = L1[i]
            i = i + 1
        else:
            L[k] = L2[j]
            j = j + 1
        k = k + 1
    if i < len(L1):   # L1 ainda não acabou
        while i < len(L1):
            L[k] = L1[i]
            i, k = i + 1, k + 1
    else:   # L2 é que ainda não acabou
        while j < len(L2):
            L[k] = L2[j]
            j, k = j + 1, k + 1
```


Análise do Mergesort

- O tempo médio de execução do mergesort é proporcional a $n\log_2 n$
- Portanto, em relação aos métodos de seleção direta, inserção direta ou o da bolha que são proporcionais a n^2, o mergesort é muito eficiente
- Pode ser usado na prática para grandes quantidades de dados

Quicksort

- O *quicksort* é outro método de ordenação eficiente no tempo de execução
- Ele é recursivo
- Baseia-se em escolher um elemento qualquer da lista como pivô
- E então dividir a lista em duas partes:
  - Uma com os elementos menores que o pivô
  - E a outra com os elementos maiores que o pivô
- Claro, isto para ordem crescente
- Para ordem decrescente, é só partir ao contrário
- Em seguida, aplicamos o mesmo método a cada uma das partes

Quicksort

- O algoritmo pode ser descrito assim em Python:

```
def quicksort(L):
    if len(L) == 0:
        return []
    else:
        # o primeiro elemento de L será o pivô
        L1 = [ x for x in L[1:] if x <= L[0] ]
        L2 = [ y for y in L[1:] if y >  L[0] ]
        return quicksort(L1) + [L[0]] + quicksort(L2)
```

Quicksort

- Outra versão do algoritmo com mais eficiência do espaço de memória ocupado durante a execução:

```
def quicksort(L, inicio, fim):
    i, j = particione(L, inicio, fim)
    if inicio < j: quicksort(L, inicio, j)
    if i < fim:    quicksort(L, i, fim)
```


Quicksort

- O algoritmo de partição da seção da lista do índice `i` ao índice `j` pode ser escrito assim em Python:

```
def particione(L, i, j):
    pivot = L[(i+j)//2] # elemento do meio; pode ser
                        # qualquer um entre i e j inclus.
    while True:
        while L[i] < pivot: i = i + 1
        while L[j] > pivot: j = j - 1
        if i <= j:
            L[i], L[j] = L[j], L[i]
            i, j = i + 1, j - 1
        else: break
    return (i, j)
```

Análise do Quicksort

- A partição feita pelo algoritmo dado está considerando ordem crescente
- Fica, como exercício, modificá-lo para considerar ordem decrescente
- O tempo médio de execução do quicksort é proporcional a $n\log_2 n$
- Portanto o quicksort também é muito eficiente e pode ser usado na prática para listas muito grandes


Considerações Finais

- O método de ordenação do Python
  - O tipo lista do Python tem um método de ordenação *builtin* denominado `sort`
  - Por exemplo, para uma lista L de números inteiros, ele pode ser usado assim para ordem crescente:

```
L.sort()
```

  - Ou assim para ordem decrescente:

```
L.sort(reverse=True)
```

  - Na prática, devemos utilizar esse método, porque ele é “turbinado” para o uso em Python


Exercícios

- Reescreva o algoritmo do quicksort de modo que considere o primeiro elemento da lista como pivô
- Faça um experimento para comparar os tempos de execução dos algoritmos de ordenação dados, em sua (de você) plataforma computacional
- Para tanto, crie uma lista grande de números inteiros aleatórios, digamos 1 milhão de elementos
- Teste o método de inserção direta, o de seleção direta, o mergesort e o quicksort


Resultados de um Experimento

Tamanho	Seleção Dir.	Troca Dir. (Bolha)	Inserção Dir.	Mergesort	Quicksort
10.000	4,453129 seg	9,46271 seg	4,92912 seg	0,089958 seg	0,049536 seg
100.000	7,369705 min	15,8989 min	8,32233 min	0,639951 seg	0,344059 seg
1.000.000	16:23:58,06 h	?????	20:19:54,78 h	7,917561 seg	3,888776 seg

Feito em um Linux Ubuntu 16.04 64 bit, proc. Intel Core i5-2500 3,30 GHz x 4, mem. 8 GB, Python 3.5.2

Resultados de um Experimento

Tamanho	Seleção Dir.	Troca Dir. (Bolha)	Inserção Dir.	Mergesort	Quicksort
10.000	0,062008 seg	0,17988 seg	0,093546 seg	0,005849 seg	0,001881 seg
100.000	5,86376 seg	17,6680 seg	9,10255 seg	0,039879 seg	0,010544 seg
1.000.000	9,77938 min	28,9075 min	15,01452 min	0,397259 seg	0,114966 seg

Feito em um MacBook Air de meados de 2012, proc. Intel Core i5 1,7 GHz, mem. 4 GB DDR3 1600 MHz, comp. GNU C++ 4.2.1 Apple LLVM 7.0.0.