Programação Script

Busca Binária e Recursão

Aula 14

Prof. Felipe A. Louza

UFU

Roteiro

1. Busca Binária
2. Custo computacional
3. Recursão
4. Exemplos
5. Referências

Roteiro

1. Busca Binária
2. Custo computacional
3. Recursão
4. Exemplos
5. Referências

O Problema da Busca

Relembrando o problema da busca:

- Dada uma `lista` de `inteiros`, queremos encontrar um **valor** x nessa lista.

0	1	2	3	4	5	6	7	8	9
20	5	15	24	67	5	1	76	17	5

- Na aula passada vimos a **busca sequencial**.
  - Operações: `find(x)`, `count(x)` e `locate(x)`.

Agora, vamos assumir que a nossa lista **está ordenada**[1].

- Podemos fazer **melhor** que a busca sequencial?

[1] Vimos também algoritmos de ordenação: `Selection Sort` e `Bubble Sort`.

O Problema da Busca

Relembrando o problema da busca:

- Dada uma `lista` de `inteiros`, queremos encontrar um **valor** x nessa lista.

0	1	2	3	4	5	6	7	8	9
20	5	15	24	67	5	1	76	17	5

- Na aula passada vimos a **busca sequencial**.
  - Operações: `find(x)`, `count(x)` e `locate(x)`.

Agora, vamos assumir que a nossa lista **está ordenada**[1].

- Podemos fazer **melhor** que a busca sequencial?

[1] Vimos também algoritmos de ordenação: Selection Sort e Bubble Sort.

O Problema da Busca

Relembrando o problema da busca:

- Dada uma `lista` de `inteiros`, queremos encontrar um **valor** x nessa lista.

1	5	5	5	15	17	20	24	67	76
0	1	2	3	4	5	6	7	8	9

- Na aula passada vimos a **busca sequencial**.
  - Operações: `find(x)`, `count(x)` e `locate(x)`.

Agora, vamos assumir que a nossa lista **está ordenada**[1].

- Podemos fazer **melhor** que a busca sequencial?

[1]Vimos também algoritmos de ordenação: `Selection Sort` e `Bubble Sort`.

O Problema da Busca

Relembrando o problema da busca:

- Dada uma `lista` de `inteiros`, queremos encontrar um **valor** x nessa lista.

1	5	5	5	15	17	20	24	67	76
0	1	2	3	4	5	6	7	8	9

- Na aula passada vimos a **busca sequencial**.
  - Operações: `find(x)`, `count(x)` e `locate(x)`.

Agora, vamos assumir que a nossa lista **está ordenada**[1].

- Podemos fazer **melhor** que a busca sequencial?

[1] Vimos também algoritmos de ordenação: `Selection Sort` e `Bubble Sort`.

Busca Binária

A busca binária é um algoritmo um pouco mais **sofisticado**.

- Assumindo que a lista está ordenada, começamos com `l = 0` e `r = len(lista)-1`:

1	5	5	5	15	17	20	24	67	76
0	1	2	3	4	5	6	7	8	9

- Compare a **chave de busca** x com o valor da posição do meio da lista, isto é, `lista[m]` com `m=(l+r)//2`.
  1. Se `lista[m] == x`, encontramos a chave ✓.
  2. Se `lista[m] > x`, então repita o processo mas considere a primeira metade da lista.
  3. Se `lista[m] < x`, conside a segunda metade para a próxima etapa.

Considere $x = 67$.

Busca Binária

A busca binária é um algoritmo um pouco mais **sofisticado**.

- Assumindo que a lista está ordenada, começamos com `l = 0` e `r = len(lista)-1`:

1	5	5	5	15	17	20	24	67	76
0	1	2	3	4	5	6	7	8	9

- Compare a **chave de busca** x com o valor da posição do meio da lista, isto é, `lista[m]` com `m=(l+r)//2`.
  1. Se `lista[m] == x`, encontramos a chave ✓.
  2. Se `lista[m] > x`, então repita o processo mas considere a primeira metade da lista.
  3. Se `lista[m] < x`, conside a segunda metade para a próxima etapa.

Considere $x = 67$.

Busca Binária

1	5	5	5	15	17	20	24	67	76
0	1	2	3	4	5	6	7	8	9

```python
def busca_binaria_find(lista, x):
  l = 0
  r = len(lista)-1

  while(l <= r): #enquanto a lista tiver pelo menos 1 elemento
    m = (l + r) // 2
    if(lista[m] == x):
      return True
    elif (lista[m] > x):
      r = m - 1
    else:
      l = m + 1
  #não encontrou
  return False
```

Busca Binária

```
>>> for item in lista:
        print(item, busca_binaria_find(lista,item))
```

1	5	5	5	15	17	20	24	67	76
0	1	2	3	4	5	6	7	8	9

- Como implementar:
  1. `count(x)`: quantas vezes x ocorre na lista?
  2. `locate(x)`: em quais posições x ocorre na lista?

Busca Binária

```
>>> for item in lista:
        print(item, busca_binaria_find(lista,item))
```

1	5	5	5	15	17	20	24	67	76
0	1	2	3	4	5	6	7	8	9

- Como implementar:
  1. `count(x):` quantas vezes x ocorre na lista?
  2. `locate(x):` em quais posições x ocorre na lista?

Busca Binária

- Operação `count(x)`:

```python
def busca_binaria_count(lista, x):
  l = 0
  r = len(lista)-1
  total = 0
  while(l <= r): #enquanto a lista tiver pelo menos 1 elemento
    m = (l + r) // 2
    if(lista[m] == x):
      total += 1
      i = m + 1
      while(i < len(lista) and lista[i] == x):
        total += 1
        i += 1
      i = m - 1
      while(i >= 0 and lista[i] == x):
        total += 1
        i -= 1
      return total
    elif (lista[m] > x): r = m - 1
    else: l = m + 1
  #não encontrou
  return total
```

Busca Binária

- Operação `locate(x)`:

```python
def busca_binaria_locate(lista, x):
  l = 0
  r = len(lista)-1
  res = []
  while(l <= r): #enquanto a lista tiver pelo menos 1 elemento
    m = (l + r) // 2
    if(lista[m] == x):
      res.append(m)
      i = m+1
      while(i<len(lista) and lista[i]==x):
        res.append(i)
        i+=1
      i = m-1
      while(i>=0 and lista[i]==x):
        res.append(i)
        i-=1
      return res
    elif (lista[m] > x): r = m - 1
    else: l = m + 1
  #não encontrou
  return False
```

O Problema da Busca

Busca Binária **vs.** busca sequencial:

- Qual solução é a mais eficiente?

Roteiro

Custo computacional

Podemos dizer que algoritmo de busca binária é mais eficiente do que a **busca sequencial**?

- Na aula passada vimos como estimar o tempo de execução de um algoritmo (**análise simplificada**).
    1. quantas operações o algoritmo executa?
    2. quanto tempo cada operação demora?

Eficiência dos Algoritmos

No caso da **busca binária** temos três possibilidades para `find()`:

0	1	2	3	4	5	6	7	8	9	10	11	12

- No melhor caso, a chave de busca estará na posição do meio. Portanto teremos uma única comparação.
- No pior caso, a busca divide a **lista** em sublistas de tamanhos:

$$n/2, n/4, n/8, \ldots, ??$$

  - só paramos quando a (sub)lista tem 1 elemento
    - Ou seja, após $\log_2 n$ chamadas $\leftarrow$ $\log_2 n$ comparações
- No caso médio, se uma chave qualquer pode ser requisitada com a mesma probabilidade, o número de comparações também será

$$\approx \log_2 n$$

Eficiência dos Algoritmos

No caso da **busca binária** temos três possibilidades para `find()`:

0	1	2	3	4	5	6	7	8	9	10	11	12

- No melhor caso, a chave de busca estará na posição do meio. Portanto teremos uma única comparação.
- No pior caso, a busca divide a **lista** em sublistas de tamanhos:

$$n/2, n/4, n/8, \dots, ??$$

  - só paramos quando a (sub)lista tem 1 elemento
    - Ou seja, após $\log_2 n$ chamadas $\leftarrow$ $\log_2 n$ comparações
- No caso médio, se uma chave qualquer pode ser requisitada com a mesma probabilidade, o número de comparações também será

$$\approx \log_2 n$$

No caso da **busca binária** temos três possibilidades para `find()`:

0	1	2	3	4	5	6	7	8	9	10	11	12

- No melhor caso, a chave de busca estará na posição do meio. Portanto teremos uma única comparação.
- No pior caso, a busca divide a **lista** em sublistas de tamanhos:

$$n/2, n/4, n/8, \ldots, ??$$

  - só paramos quando a (sub)lista tem 1 elemento
  - Ou seja, após $\log_2 n$ chamadas $\leftarrow$ $\log_2 n$ comparações

- No caso médio, se uma chave qualquer pode ser requisitada com a mesma probabilidade, o número de comparações também será

$$\approx \log_2 n$$

Eficiência dos Algoritmos

No caso da **busca binária** temos três possibilidades para `find()`:

0	1	2	3	4	5	6	7	8	9	10	11	12

- No melhor caso, a chave de busca estará na posição do meio. Portanto teremos uma única comparação.
- No pior caso, a busca divide a **lista** em sublistas de tamanhos:

$$n/2, n/4, n/8, \ldots, ??$$

  - só paramos quando a (sub)lista tem 1 elemento
    - Ou seja, após $\log_2 n$ chamadas $\leftarrow$ $\log_2 n$ comparações
- No caso médio, se uma chave qualquer pode ser requisitada com a mesma probabilidade, o número de comparações também será

$$\approx \log_2 n$$

Eficiência dos Algoritmos

No caso da **busca binária** temos três possibilidades para `find()`:

0	1	2	3	4	5	6	7	8	9	10	11	12

- No melhor caso, a chave de busca estará na posição do meio. Portanto teremos uma única comparação.
- No pior caso, a busca divide a **lista** em sublistas de tamanhos:

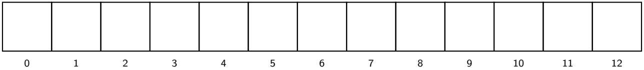
$$n/2, n/4, n/8, \ldots, ??$$

  - só paramos quando a (sub)lista tem 1 elemento
    - Ou seja, após $\log_2 n$ chamadas $\leftarrow$ $\log_2 n$ comparações
- No caso médio, se uma chave qualquer pode ser requisitada com a mesma probabilidade, o número de comparações também será

$$\approx \log_2 n$$

Custo computacional

No exemplo da lista telefonica com **2 milhões** de registros do tipo `(nome, telefone)`.



Assumindo cada comparação feita em 1 milisegundo (10^{-3}).

- Com a **busca sequencial**:

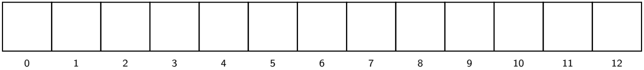
  2000 s $\approx$ 33 minutos no pior caso
  1000 s $\approx$ 16 minutos no caso médio

- Com a **busca binária**: $\log_2$(2 milhões) $\approx$ 21 comparações

  $\approx$ 21 milisegundos nos dois casos

Custo computacional

No exemplo da lista telefonica com **2 milhões** de registros do tipo `(nome, telefone)`.



Assumindo cada comparação feita em 1 milisegundo (10^{-3}).

- Com a **busca sequencial**:

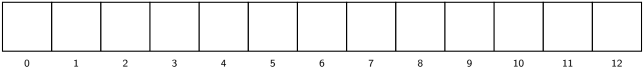
$$\begin{aligned} 2000 \text{ s} &\quad \approx 33 \text{ minutos no pior caso} \\ 1000 \text{ s} &\quad \approx 16 \text{ minutos no caso médio} \end{aligned}$$

- Com a **busca binária**: $\log_2(2 \text{ milhões}) \approx 21$ comparações

$$\approx 21 \text{ milisegundos nos dois casos}$$

Custo computacional

No exemplo da lista telefonica com **2 milhões** de registros do tipo `(nome, telefone)`.



Assumindo cada comparação feita em 1 milisegundo (10^{-3}).

- Com a **busca sequencial**:

$$\begin{aligned} 2000 \text{ s} &\quad \approx 33 \text{ minutos no pior caso} \\ 1000 \text{ s} &\quad \approx 16 \text{ minutos no caso médio} \end{aligned}$$

- Com a **busca binária**: $\log_2(2 \text{ milhões}) \approx 21$ comparações

$$\approx 21 \text{ milisegundos nos dois casos}$$

Eficiência dos Algoritmos

Mas uma ressalva deve ser feita: para utilizar a busca binária, a **precisa estar ordenada!**

- Se você tiver um cadastro onde vários itens são atualizados com **frequência**, a busca binária pode não ser a melhor opção, já que você precisará manter a lista ordenada.

Eficiência dos Algoritmos

Mas uma ressalva deve ser feita: para utilizar a busca binária, a **precisa estar ordenada!**

- Se você tiver um cadastro onde vários itens são atualizados com **frequência**, a busca binária pode não ser a melhor opção, já que você precisará manter a lista ordenada.

Roteiro

Chamada de funções:

```
def funcao1():
   #cmd C
   funcao2();
   #cmd K

   ...

def funcao2():
   #cmd D
   funcao3();
   #cmd J
```

```
def funcao3():
   #cmd E
   funcao4()
   #cmd I

   ...

def funcao4():
   #cmd F
   funcao4()
   #cmd H
```

```
def main():
   # cmd A
   # cmd B
   funcao1()

main()
```

- Vimos que qualquer função pode chamar outra função (inclusive ela mesma).
  - Recursão!

Recursão

Relembrando que sempre temos que definir um critério de parada.

```python
def main():
  x = 5
  y = misterio(x)
  print("y =", y)

def misterio(x, a=1):
  c = 0
  if(a < x):
    c = misterio(x, a+1)
  return x + c

main()
```

x	a	c	return

- A função `misterio()` chama `misterio()` enquanto `a < x`.
- Na verdade `misterio()` calcula:

$$x + (x + (x + (x + \cdots + (x)))) = \underbrace{x + x + x + \cdots + x}_{x \text{ vezes}} = x^2$$

Recursão

Relembrando que sempre temos que definir um critério de parada.

```python
def main():
  x = 5
  y = misterio(x)
  print("y =", y)

def misterio(x, a=1):
  c = 0
  if(a < x):
    c = misterio(x, a+1)
  return x + c

main()
```

x	a	c	return

- A função `misterio()` chama `misterio()` enquanto `a < x`.
- Na verdade `misterio()` calcula:

$$x + (x + (x + (x + \cdots + (x)))) = \underbrace{x + x + x + \cdots + x}_{x \text{ vezes}} = x^2$$

Recursão

Relembrando que sempre temos que definir um critério de parada.

```
def main():
  x = 5
  y = misterio(x)
  print("y =", y)

def misterio(x, a=1):
  c = 0
  if(a < x):
    c = misterio(x, a+1)
  return x + c

main()
```

x	a	c	return

- A função `misterio()` chama `misterio()` enquanto `a < x`.
- Na verdade `misterio()` calcula:

$$x + (x + (x + (x + \cdots + (x)))) = \underbrace{x + x + x + \cdots + x}_{x \text{ vezes}} = x^2$$

Recursão

Relembrando que sempre temos que definir um critério de parada.

```python
def main():
  x = 5
  y = misterio(x)
  print("y =", y)

def misterio(x, a=1):
  c = 0
  if(a < x):
    c = misterio(x, a+1)
  return x + c

main()
```

x	a	c	return

- A função `misterio()` chama `misterio()` enquanto `a < x`.
- Na verdade `misterio()` calcula:

$$x + (x + (x + (x + \cdots + (x)))) = \underbrace{x + x + x + \cdots + x}_{x \text{ vezes}} = x^2$$

Recursão

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, etc...
- ou podem ser vistos do ponto de vista recursivo:
  - soma, quadrado de um número, exponenciação, busca binária, etc...

Recursão

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, etc...
- ou podem ser vistos do ponto de vista recursivo:
  - soma, quadrado de um número, exponenciação, busca binária, etc...

Recursão

Recursão é um método de resolução de problemas que envolve:

- Primeiro, definir as soluções para casos básicos
- Em seguida, reduzir o problema para instâncias menores do mesmo problema.
- Finalmente, combinar o resultado das instâncias menores para obter um resultado do problema original

Roteiro

Exemplo 1

O fatorial de um número natural n, representado por $n!$, pode ser escrito como:

$$fat(n) = \begin{cases} 1 & \text{se } n = 0 \\ \underline{n \times fat(n-1)} & \text{se } n > 0 \end{cases}$$

```
def main():
  x = 5
  print("x! =", fat(x))

def fat(n):
  if(n == 0): # caso base
    return 1
  else:
    return n * fat(n-1)

main()
```

n	return

- Calcule `fatorial(5)`:

Exemplo 1

O fatorial de um número natural n, representado por $n!$, pode ser escrito como:

$$fat(n) = \begin{cases} 1 & \text{se } n = 0 \\ \underline{n \times fat(n-1)} & \text{se } n > 0 \end{cases}$$

```python
def main():
  x = 5
  print("x! =", fat(x))

def fat(n):
  if(n == 0): # caso base
    return 1
  else:
    return n * fat(n-1)

main()
```

n	return

- Calcule `fatorial(5)`:

Exemplo 1

Observe que também podemos calcular o fatorial de forma iterativa:

$$fat(n) = 1 \times 2 \times \cdots \times n$$

```python
def main():
  x = 5
  print("x! =", fat2(x))

def fat2(n):
  total = 1
  for i in range(2, n+1):
    total *= i
  return total

main()
```

- Calcule `fat2(5)`:

Comparando recursão e algoritmos iterativos

Qual solução é mais eficiente?

$$fat(n) = \begin{cases} 1 & \text{se } n = 0 \\ \underline{n \times fat(n-1)} & \text{se } n > 0 \end{cases}$$

ou

$$\underline{fat(n) = 1 \times 2 \times \cdots \times n}$$

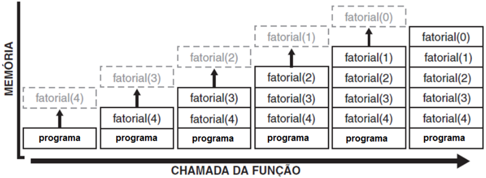
As duas vão ser executadas n vezes, porém a segunda deve ser mais rápida.

- Para responder melhor, vamos entender o que acontece na memória.

Comparando recursão e algoritmos iterativos

Toda vez que uma função chama outra (recursivamente ou não), suas variáveis locais precisam ser **armazenadas na memória**.

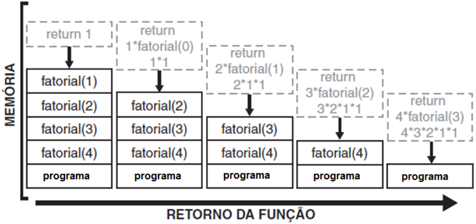
- Dessa forma, podemos recuperar esses valores quando houver o retorno da função chamada.
- No caso do fatorial:



Comparando recursão e algoritmos iterativos

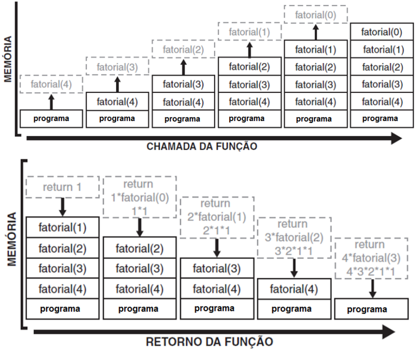
Quando uma função termina, suas variáveis locais **são removidas** da memória.

- Pode ser que algum valor seja retornado, e
- Os valores das chamadas anteriores são recuperados.



Comparando recursão e algoritmos iterativos

- Tudo isso, além de **ocupar memória**, tem um custo adicional de tempo a cada chamada recursiva.



  - Por isso que, em algumas situações o algoritmo recursivo é menos eficiente.

Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais "elegantes"

Mas algumas vezes podem ser

- muito ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

1. encontrar algoritmo recursivo para o problema
2. reescrevê-lo como um algoritmo iterativo

Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais "elegantes"

Mas algumas vezes podem ser

- muito ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

1. encontrar algoritmo recursivo para o problema
2. reescrevê-lo como um algoritmo iterativo

Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais "elegantes"

Mas algumas vezes podem ser

- muito ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

1. encontrar algoritmo recursivo para o problema
2. reescrevê-lo como um algoritmo iterativo

Exemplo 2

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

$$fibo(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \underline{fibo(n-1)} + \underline{fibo(n-2)} & \text{se } n > 2 \end{cases}$$

```
def main():
  n = 5
  print("fibo(n) =", fibo(n))

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

main()
```


- Calcule `fibo(5)`

Exemplo 2

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

$$fibo(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \underline{fibo(n-1)} + \underline{fibo(n-2)} & \text{se } n > 2 \end{cases}$$

```
def main():
  n = 5
  print("fibo(n) =", fibo(n))

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

main()
```


- Calcule `fibo(5)`

Exemplo 2

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

$$fibo(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \underline{fibo(n-1)} + \underline{fibo(n-2)} & \text{se } n > 2 \end{cases}$$

```
def main():
  n = 5
  print("fibo(n) =", fibo(n))

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

main()
```


- Calcule `fibo(5)`

Exemplo 2

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

$$fibo(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \underline{fibo(n-1)} + \underline{fibo(n-2)} & \text{se } n > 2 \end{cases}$$

```python
def main():
  n = 5
  print("fibo(n) =", fibo(n))

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

main()
```


- Calcule `fibo(5)`

Exemplo 2

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

$$fibo(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \underline{fibo(n-1)} + \underline{fibo(n-2)} & \text{se } n > 2 \end{cases}$$

```
def main():
  n = 5
  print("fibo(n) =", fibo(n))

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

main()
```


- Calcule `fibo(5)`

Exemplo 2

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

$$fibo(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \underline{fibo(n-1)} + \underline{fibo(n-2)} & \text{se } n > 2 \end{cases}$$

```
def main():
  n = 5
  print("fibo(n) =", fibo(n))

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

main()
```


- Calcule `fibo(5)`

Exemplo 2

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

$$fibo(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \underline{fibo(n-1)} + \underline{fibo(n-2)} & \text{se } n > 2 \end{cases}$$

```
def main():
  n = 5
  print("fibo(n) =", fibo(n))

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

main()
```


- Calcule `fibo(5)`

Exemplo 2

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

$$fibo(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \underline{fibo(n-1)} + \underline{fibo(n-2)} & \text{se } n > 2 \end{cases}$$

```
def main():
  n = 5
  print("fibo(n) =", fibo(n))

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

main()
```


- Calcule `fibo(5)`

Exemplo 2

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, . . .

$$fibo(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \underline{fibo(n-1)} + \underline{fibo(n-2)} & \text{se } n > 2 \end{cases}$$

```python
def main():
  n = 5
  print("fibo(n) =", fibo(n))

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

main()
```


- Calcule `fibo(5)`

Exemplo 2

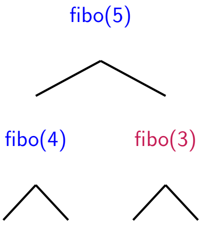
Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, . . .

$$fibo(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ \underline{fibo(n-1)} + \underline{fibo(n-2)} & \text{se } n > 2 \end{cases}$$

```python
def main():
  n = 5
  print("fibo(n) =", fibo(n))

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

main()
```



- Calcule `fibo(5)`

Exemplo 2

Solução iterativa: Sequência de Fibonacci.

$$fibo(n) = \underbrace{fibo(1) + fibo(2) + fibo(3) + \cdots + fibo(n)}_{n \text{ vezes}}$$

```python
def main():
  n = 5
  print("fibo2(n) =", fibo2(n))

def fibo2(n):
  ant = 1
  atual = 1
  for i in range(3, n+1):
    prox = ant + atual
    ant = atual
    atual = prox
  return atual

main()
```

ant	atual	prox

- Calcule fibo2(5)

Exemplo 2

Solução iterativa: Sequência de Fibonacci.

$$fibo(n) = \underbrace{fibo(1) + fibo(2) + fibo(3) + \cdots + fibo(n)}_{n \text{ vezes}}$$

```python
def main():
  n = 5
  print("fibo2(n) =", fibo2(n))

def fibo2(n):
  ant = 1
  atual = 1
  for i in range(3, n+1):
    prox = ant + atual
    ant = atual
    atual = prox
  return atual

main()
```

ant	atual	prox

- Calcule `fibo2(5)`

Exemplo 2

Número de operações:

- iterativo: $\approx n$
- recursivo: $\approx \texttt{fib}(n)$ (aproximadamente 1.6^n)

Exemplo 2

```
def tempo(algoritmo, lista):
  import time
  antes = time.time()
  algoritmo(lista)
  depois = time.time()
  return depois-antes
```

```
>>> n = 40
>>> print("{:.2f} segundos".format(tempo(fibo, n)))
21.74 segundos
>>>
>>> print("{} segundos".format(tempo(fibo2, n)))
3.337860107421875e-05 segundos
```

Exemplo 3

Exponenciação: a^n

Seja a é um número inteiro e n é um número inteiro não-negativo

$$a^n = \underbrace{a \times a \times \cdots \times a}_{n \text{ vezes}}$$

```python
def main():
  a, n = 2, 5
  print("a^n =", pot(a, n))

def pot(a, n):
  total = 1
  for i in range(0, n):
    total *= a
  return total

main()
```

- Calcule pot(2,5):

Exemplo 3

Podemos redefinir o problema de forma recursiva:

$$a^n = \begin{cases} 1 & \text{se } n = 0 \\ \underline{a \times a^{n-1}} & \text{se } n > 0 \end{cases}$$

```python
def main():
  a, n = 2, 5
  print("a^n =", pot2(a, n))

def pot2(a, n):
  if(n == 0): # caso base
    return 1
  else:
    return a * pot2(a, n-1)

main()
```

a	n	return

- Calcule pot2(2,5):

Exemplo 3

Podemos redefinir o problema de forma recursiva:

$$a^n = \begin{cases} 1 & \text{se } n = 0 \\ \underline{a \times a^{n-1}} & \text{se } n > 0 \end{cases}$$

```python
def main():
  a, n = 2, 5
  print("a^n =", pot2(a, n))

def pot2(a, n):
  if(n == 0): # caso base
    return 1
  else:
    return a * pot2(a, n-1)

main()
```

a	n	return

- Calcule `pot2(2,5)`:

Exemplo 3

Neste caso a solução iterativa **também** é mais eficiente.

```python
def pot(a, n):
  total = 1
  for i in range(0, n):
    total *= a
  return total
```

```python
def pot2(a, n):
  if(n == 0): # caso base
    return 1
  else:
    return a * pot2(a, n-1)
```

- 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 memória**.

Exemplo 3

Mas e se definirmos a potência de forma diferente?

$$a^n = \begin{cases} 1 & \text{se } n = 0 \\ (a^{n/2})^2 & \text{se } n > 0 \text{ e } n \text{ é par} \\ a \times (a^{(n-1)/2})^2 & \text{se } n > 0 \text{ e } n \text{ é ímpar} \end{cases}$$

```python
def pot3(a, n):
  if n == 0:
    return 1
  elif n%2 == 0:
    aux = pot3(a, n//2)
    return aux*aux
  else:
    aux = pot3(a, (n-1)//2)
    return aux*aux*a
```

a	aux	n	return

- Calcule `pot3(2,5)`:

Exemplo 3

Mas e se definirmos a potência de forma diferente?

$$a^n = \begin{cases} 1 & \text{se } n = 0 \\ (a^{n/2})^2 & \text{se } n > 0 \text{ e } n \text{ é par} \\ a \times (a^{(n-1)/2})^2 & \text{se } n > 0 \text{ e } n \text{ é ímpar} \end{cases}$$

```python
def pot3(a, n):
  if n == 0:
    return 1
  elif n%2 == 0:
    aux = pot3(a, n//2)
    return aux*aux
  else:
    aux = pot3(a, (n-1)//2)
    return aux*aux*a
```

a	aux	n	return

- Calcule `pot3(2,5)`:

Exemplo 3

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

```
def pot3(a, n):
  if n == 0:
    return 1
  elif n%2 == 0:
    aux = pot3(a, n//2)
    return aux*aux
  else:
    aux = pot3(a, (n-1)//2)
    return aux*aux*a
```

Exemplo 3

Número de operações:

- A cada chamada recursiva o valor de n **é dividido por 2**.

$$n/2, n/4, n/8, \ldots, ??$$

  - Ou seja, usando divisões inteiras faremos $\lceil(\log_2 n)\rceil + 1$ chamadas recursivas

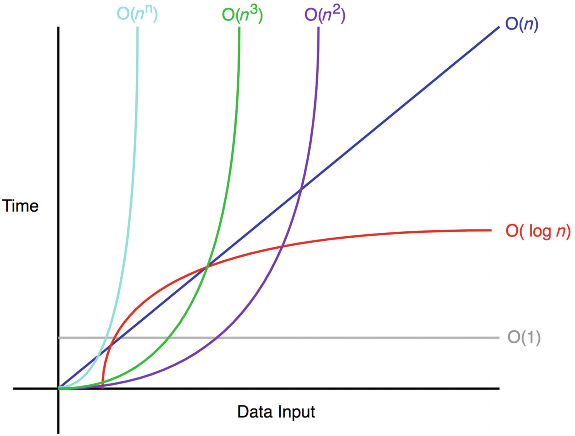
$$\approx \log_2 n \text{ operações}$$

- Enquanto isso, o algoritmo iterativo executa o laço n vezes.

$$n \text{ operações!}$$

Eficiência dos Algoritmos

Por que isso importa?



Fim

Dúvidas?

Roteiro

Referências

1. Materiais adaptados dos slides do Prof. Eduardo C. Xavier, da Universidade Estadual de Campinas.
2. panda.ime.usp.br/pensepy/static/pensepy/index.html