Algoritmos e Complexidade

Fabio Gagliardi Cozman
Thiago Martins

PMR2300/PMR3200
Escola Politécnica da Universidade de São Paulo

Algoritmos

- Um algoritmo é um procedimento descrito passo a passo para resolução de um problema *em tempo finito*.
- Formalização: máquinas de Turing.
- Algoritmos são julgados com base em vários fatores:
  - tempo de escrita;
  - complexidade de manutenção;
  - consumo de memória;
  - eficiência de execução.

Eficiência

Quanto à eficiência, vários fatores se inter-relacionam:

- qualidade do código;
- tipo do processador;
- qualidade do compilador;
- linguagem de programação.

Análise de Algoritmos

Análise de algoritmos O procedimento geral para se verificar se um algoritmo termina em tempo finito e efetivamente resolve o problema proposto...

Análise de algoritmos O procedimento geral para se verificar se um algoritmo termina em tempo finito e efetivamente resolve o problema proposto... não existe

Análise de Algoritmos

Análise de algoritmos O procedimento geral para se verificar se um algoritmo termina em tempo finito e efetivamente resolve o problema proposto... não existe e *nunca existirá* ! (Tese de Church-Turing)

Técnicas para Algoritmos Iterativos

- Identifique as pré-condições do algoritmo.
- Identifique os laços.
- Identifique as condições de permanência nos laços.
- Mostre que as condições são eventualmente atendidas (finitude).
- Identifique a lei de recorrência (ou regra de transição) em cada laço.
- Encontre um *invariante* adequado para o algoritmo. Invariantes de um algoritmo iterativo são propriedades, ou proposições lógicas, que permanecem inalteradas em todos os laços (ou seja, não são afetadas pelas regras de transição)
- Mostre que o invariante ao final do algoritmo leva ao resultado correto.

Exemplo: Busca Sequencial

Seja $A = \{a_1, \ldots, a_n\}$ uma sequência de inteiros, e x um número inteiro. Construa um algoritmo que ou retorna i tal que $a_i = x$ ou retorna `None` se o número x não existe na sequência.

Exemplo: Busca Sequencial

Seja $A = \{a_1, \ldots, a_n\}$ uma sequência de inteiros, e x um número inteiro. Construa um algoritmo que ou retorna i tal que $a_i = x$ ou retorna `None` se o número x não existe na sequência.

```python
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pré condições (em “matematiquês”):

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pré condições (em “matematiquês”):

$$A = \{a_1, \ldots, a_n\}, n \geq 0 \qquad (1)$$

Exemplo: Busca Sequencial

```python
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pré condições (em “matematiquês”):

$$A = \{a_1, \ldots, a_n\}, n \geq 0 \tag{1}$$

A é uma sequência finita de valores (pode ser nula!).

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pré condições (em “matematiquês”):

$$A = \{a_1, \ldots, a_n\}, n \geq 0 \tag{1}$$

A é uma sequência finita de valores (pode ser nula!).

$$a_i \in \mathbb{Z} \forall i \leq n \tag{2}$$

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pré condições (em “matematiquês”):

$$A = \{a_1, \ldots, a_n\}, n \geq 0 \quad (1)$$

A é uma sequência finita de valores (pode ser nula!).

$$a_i \in \mathbb{Z} \forall i \leq n \quad (2)$$

A é uma sequência de *inteiros*.

Exemplo: Busca Sequencial

```python
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pré condições (em “matematiquês”):

$$A = \{a_1, \ldots, a_n\}, n \geq 0 \tag{1}$$

A é uma sequência finita de valores (pode ser nula!).

$$a_i \in \mathbb{Z} \forall i \leq n \tag{2}$$

A é uma sequência de *inteiros*.

$$x \in \mathbb{Z} \tag{3}$$

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pré condições (em “matematiquês”):

$$A = \{a_1, \ldots, a_n\}, n \geq 0 \quad (1)$$

A é uma sequência finita de valores (pode ser nula!).

$$a_i \in \mathbb{Z} \forall i \leq n \quad (2)$$

A é uma sequência de *inteiros*.

$$x \in \mathbb{Z} \quad (3)$$

x é um *inteiro*

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pré condições (em “matematiquês”):

$$A = \{a_1, \ldots, a_n\}, n \geq 0 \tag{1}$$

A é uma sequência finita de valores (pode ser nula!).

$$a_i \in \mathbb{Z} \forall i \leq n \tag{2}$$

A é uma sequência de *inteiros*.

$$x \in \mathbb{Z} \tag{3}$$

x é um *inteiro*

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pós-condições (desejadas):

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pós-condições (desejadas):

$$\texttt{busca}(A, x) = r \tag{4}$$

$$r = \texttt{None} \implies a_k \neq x \forall 0 \leq k \leq n \tag{5}$$

$$r \neq \texttt{None} \implies a_r = x \tag{6}$$

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Pós-condições (desejadas):

$$\texttt{busca}(A, x) = r \tag{4}$$

$$r = \texttt{None} \implies a_k \neq x \forall 0 \leq k \leq n \tag{5}$$

$$r \neq \texttt{None} \implies a_r = x \tag{6}$$

Seja r o retornado pela função. Se r é `None` então *nenhum* número na sequência é igual a x. Senão, $a_r = x$.

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

Regras de transição:
Seja i_j o valor de i ao *final* da j-ésima iteração do laço `while`, com $i_0 = 0$.

$$i_{j+1} = i_j + 1 \tag{7}$$

Isso, naturalmente, implica em $i = j$, ou seja, o valor de i na j-gésima iteração é j (embora isso possa parecer trivial, há laços mais complexos).

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

É um algoritmo finito? A condição de permanência do laço implica que

$$i \leq n \wedge a_i \neq x \tag{8}$$

Ora, mas pela lei de recorrência, i_j é exatamente o número de iterações realizadas. Deste modo, são realizadas no máximo n iterações, e o algoritmo é finito.

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

É um algoritmo *correto*?

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

É um algoritmo *correto*? Invariante *ao final* do loop:

$$a_k \neq x \forall 0 \leq k \leq (i-1) \tag{9}$$

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

É um algoritmo *correto*? Invariante *ao final* do loop:

$$a_k \neq x \forall 0 \leq k \leq (i - 1) \tag{9}$$

Todos os valores de A de 0 a $i - 1$ são diferentes de x).
Isso é *sempre* verdade?

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

É um algoritmo *correto*? Invariante *ao final* do loop:

$$a_k \neq x \forall 0 \leq k \leq (i-1) \tag{9}$$

Todos os valores de A de 0 a $i-1$ são diferentes de x).
Isso é *sempre* verdade?
É evidentemente verdade *no final* da *primeira* iteração do loop, pois para se estar dentro do loop, $a_0 != x$.

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

É um algoritmo *correto*? Invariante *ao final* do loop:

$$a_k \neq x \forall 0 \leq k \leq (i-1) \tag{9}$$

Todos os valores de A de 0 a $i-1$ são diferentes de x).
Isso é *sempre* verdade?
É evidentemente verdade *no final* da *primeira* iteração do loop, pois para se estar dentro do loop, $a_0 != x$. Ora, mas se existe um valor de i para o qual o invariante é válido e se a condição de permanência continua válida na iteração seguinte, então

$$(a_k \neq x \forall 0 \leq k \leq (i-1)) \wedge a_i \neq x \wedge i \leq n \tag{10}$$

Exemplo: Busca Sequencial

```
def busca(a,x):
    i, n = 0, len(a)-1
    while i <= n and a[i] != x : i = i + 1
    return i if i <= n else None
```

É um algoritmo *correto*?
Invariante *ao final* do loop:

$$a_k \neq x \forall 0 \leq k \leq i - 1 \tag{11}$$

O término do loop significa que a condição de permanência foi violada, ou seja,

$$i = n + 1 \vee a_i = x \tag{12}$$

Ora, mas vale também o invariante! Assim, se valor retornado r for `None`, então $i = n + 1$ e vale $a_k \neq x \forall 0 \leq k \leq n$. Senão, então $a_r = x$.

Máximo divisor comum - Algoritmo de Euclides

- Algoritmo para Encontrar o máximo divisor comum de dois números.
- A seguinte função implementa esse algoritmo para $a > b > 1$:

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

Máximo divisor comum - Algoritmo de Euclides

- É um algoritmo?

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

Máximo divisor comum - Algoritmo de Euclides

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo? Seja a_i, b_i o valor das variáveis `a` e `b` no *início* da i-gésima iteração.

$$\begin{cases} a_{i+1} & = & b_i \\ b_{i+1} & = & a_i \mod b_i \end{cases}$$

Máximo divisor comum - Algoritmo de Euclides

```python
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo? Seja a_i, b_i o valor das variáveis `a` e `b` no *início* da i-gésima iteração.

$$\begin{cases} a_{i+1} = b_i \\ b_{i+1} = a_i \mod b_i \end{cases}$$

Veja que se $a_i > b_i > 0$ então $a_{i+1} > b_{i+1} \geq 0$

Máximo divisor comum - Algoritmo de Euclides

```python
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo? Seja a_i, b_i o valor das variáveis `a` e `b` no *início* da i-gésima iteração.

$$\begin{cases} a_{i+1} & = & b_i \\ b_{i+1} & = & a_i \mod b_i \end{cases}$$

Veja que se $a_i > b_i > 0$ então $a_{i+1} > b_{i+1} \geq 0$
Além disso, $a_{i+1} < a_i$ e $b_{i+1} < b_i$

Máximo divisor comum - Algoritmo de Euclides

```python
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo? Seja a_i, b_i o valor das variáveis `a` e `b` no *início* da i-gésima iteração.

$$\begin{cases} a_{i+1} & = & b_i \\ b_{i+1} & = & a_i \mod b_i \end{cases}$$

Veja que se $a_i > b_i > 0$ então $a_{i+1} > b_{i+1} \geq 0$
Além disso, $a_{i+1} < a_i$ e $b_{i+1} < b_i$
O algoritmo termina em tempo finito!

Máximo divisor comum - Algoritmo de Euclides

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo *correto*?

Máximo divisor comum - Algoritmo de Euclides

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo *correto*?

$$\begin{cases} a_{i+1} & = & b_i \\ b_{i+1} & = & a_i \mod b_i \end{cases}$$

Máximo divisor comum - Algoritmo de Euclides

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo *correto*?

$$\begin{cases} a_{i+1} & = & b_i \\ b_{i+1} & = & a_i \mod b_i \end{cases}$$

$$\exists q_i \in \mathbb{N}^* : b_{i+1} = a_i - q_i b_i$$

Máximo divisor comum - Algoritmo de Euclides

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo *correto*?

$$\begin{cases} a_{i+1} & = & b_i \\ b_{i+1} & = & a_i \mod b_i \end{cases}$$

$$\exists q_i \in \mathbb{N}^* : b_{i+1} = a_i - q_i b_i$$

$$\text{gcd}\left(a_{i+1}, b_{i+1}\right) = \text{gcd}\left(a_i, b_i\right)$$

Máximo divisor comum - Algoritmo de Euclides

- É um algoritmo *correto*?

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

Máximo divisor comum - Algoritmo de Euclides

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo *correto*?

$$\begin{cases} a_{i+1} &= b_i \\ b_{i+1} &= a_i \mod b_i \end{cases}$$

$$b_{n+1} = 0 \Rightarrow \gcd(a_n, b_n) = b_n = a_{n+1}$$

Máximo divisor comum - Algoritmo de Euclides

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo *correto*?

$$\begin{cases} a_{i+1} & = & b_i \\ b_{i+1} & = & a_i \mod b_i \end{cases}$$

$$b_{n+1} = 0 \Rightarrow \text{gcd}(a_n, b_n) = b_n = a_{n+1}$$

$$\begin{cases} \text{gcd}(a_1, b_1) = \text{gcd}(a_n, b_n) \\ \text{gcd}(a_n, b_n) = a_{n+1} \end{cases}$$

Máximo divisor comum - Algoritmo de Euclides

```
def gcd(a,b):
    """Calcula M.D.C.
    entre a e b.
    requer a>b>0"""
    while b!=0:
        a, b = b, a%b
    return a
```

- É um algoritmo *correto*?

$$\begin{cases} a_{i+1} & = & b_i \\ b_{i+1} & = & a_i \mod b_i \end{cases}$$

$$b_{n+1} = 0 \Rightarrow \text{gcd}(a_n, b_n) = b_n = a_{n+1}$$

$$\begin{cases} \text{gcd}(a_1, b_1) = \text{gcd}(a_n, b_n) \\ \text{gcd}(a_n, b_n) = a_{n+1} \end{cases}$$

$$\text{gcd}(a_1, b_1) = a_{n+1}$$

Máximo valor

- Conside um exemplo onde queremos encontrar o máximo valor em um arranjo de inteiros.
- Um algoritmo simples é varrer o arranjo, verificando se cada elemento é o maior até o momento (e caso seja, armazenando esse elemento).
- A seguinte função implementa esse algoritmo:

```
def encontra_max(a):
    i = a[0]
    for j in a[1:]:
        if j>i: i=j
    return i
```

Complexidade

- Consideremos um exemplo no qual temos dois métodos para resolver o mesmo problema, cada um dos quais com uma complexidade diferente.
- Suponha que tenhamos um arranjo `x` e que queiramos calcular outro arranjo `a` tal que:

$$a_i = \frac{\sum_{j=1}^{i} x_j}{i}.$$

Uma solução é:

```
def medias(x):
    a = [None]*len(x)
    for i in range(len(x)):
        temp = 0
        for j in range(i+1):
            temp = temp + x[j]
        a[i] = temp/(i+1)
    return a
```

Qual é o custo desta função?
Note que o loop interno roda n vezes, onde n é o tamanho de $\mathtt{x}$. Em cada uma dessas vezes, temos um número de operações proporcional a

$$1 + 2 + 3 + \cdots + (n - 1) + n,$$

ou seja,

$$\sum_{i=0}^{n-1} (i + 1) = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}.$$

Portanto o custo total deverá ser aproximadamente quadrático em relação ao tamanho de $\mathtt{x}$.

Complexidade

Uma outra solução para o mesmo problema é:

```
def medias_linear(x):
    a = [None]*len(x)
    temp = 0
    for i in range(len(x)):
        temp = temp + x[i]
        a[i] = temp/(i+1)
    return a
```

Essa solução envolve basicamente *n* operações (há um custo fixo e um custo para cada elemento de `x`); o custo total é proporcional a *n*.

Ordenação

- Ordenação é o ato de se colocar os elementos de uma sequência de informações, ou dados, em uma ordem predefinida.
- A ordenação de sequências é um dos mais importantes problemas em computação, dado o enorme número de aplicações que a empregam.

Ordenação por Seleção (em ordem decrescente)

```
def ordena(a):
    for i in range(len(a)):
        min_pos = i
        min_val = a[i]
        for j in range(i+1,len(a)):
            if min_val > a[j]:
                min_val = a[j]
                min_pos = j
        temp = a[i]
        a[i] = min_val
        a[min_pos] = temp
```

Ordenação por Seleção

A complexidade desse algoritmo é quadrática no tamanho n de `a`, pois é proporcional a

$$(n-1)+(n-2)+\cdots+2+1=\frac{(n-1)n}{2}=\frac{n^2-n}{2}.$$

Ordenação por Inserção

O seguinte sequência ilustra o funcionamento de ordenação por inserção em ordem crescente:

```
i
```

6	3	4	5	9	8

Ordenação por Inserção

O seguinte sequência ilustra o funcionamento de ordenação por inserção em ordem crescente:

i						
	6	3	4	5	9	8
0	[3]	6	4	5	9	8

Ordenação por Inserção

O seguinte sequência ilustra o funcionamento de ordenação por inserção em ordem crescente:

i						
	6	3	4	5	9	8
0	[3]	6	4	5	9	8
1	3	[4]	6	5	9	8

Ordenação por Inserção

O seguinte sequência ilustra o funcionamento de ordenação por inserção em ordem crescente:

i						
	6	3	4	5	9	8
0	[3]	6	4	5	9	8
1	3	[4]	6	5	9	8
2	3	4	[5]	6	9	8

O seguinte sequência ilustra o funcionamento de ordenação por inserção em ordem crescente:

i						
	6	3	4	5	9	8
0	[3]	6	4	5	9	8
1	3	[4]	6	5	9	8
2	3	4	[5]	6	9	8
3	3	4	5	[6]	9	8

Ordenação por Inserção

O seguinte sequência ilustra o funcionamento de ordenação por inserção em ordem crescente:

`i`						
	6	3	4	5	9	8
0	[3]	6	4	5	9	8
1	3	[4]	6	5	9	8
2	3	4	[5]	6	9	8
3	3	4	5	[6]	9	8
4	3	4	5	6	[8]	9

Ordenação por Inserção

O seguinte sequência ilustra o funcionamento de ordenação por inserção em ordem crescente:

`i`						
	6	3	4	5	9	8
0	[3]	6	4	5	9	8
1	3	[4]	6	5	9	8
2	3	4	[5]	6	9	8
3	3	4	5	[6]	9	8
4	3	4	5	6	[8]	9

Ordenação por Inserção (em ordem crescente)

```
def ordena_inserct(a):
    for i in range(1, len(a)):
        temp = a[i]
        j = i
        while j > 0 and temp < a[j-1]:
            a[j] = a[j-1]
            j -= 1
        a[j] = temp
```

Ordenação por Inserção

- Note que a complexidade de ordenação por inserção varia com a entrada!
- O pior caso é aquele em que o arranjo está ordenado em ordem decrescente.
- O melhor caso, aquele em que o arranjo já está ordenado em ordem crescente.
- No *melhor caso*, o custo é proporcional a n.
- No *pior caso*, o custo é proporcional a

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = \frac{n^2 - n}{2}.$$

Definição: análise assintótica

- Para quantificar a complexidade de um algoritmo, vamos usar a ordem de crescimento do tempo de processamento em função do tamanho da entrada.
- Vamos assumir que todo algoritmo tem uma única entrada crítica cujo tamanho é N (por exemplo, o comprimento do arranjo a ser ordenado).
- A notação para indicar a ordem de um algoritmo é denominada **“BigOh”**. Temos:
  - $\mathcal{O}(N)$: ordem linear;
  - $\mathcal{O}(N^2)$: ordem quadrática;
  - $\mathcal{O}(2^N)$: ordem exponencial;
  - $\mathcal{O}(\log N)$: ordem logarítmica.

Definição formal

- O custo $T(N)$ de um algoritmo com entrada de tamanho N é $\mathcal{O}(f(N))$ se existem constantes $K > 1$ e M tal que:

$$T(N) \leq K \cdot f(N), \qquad \forall N \geq M.$$

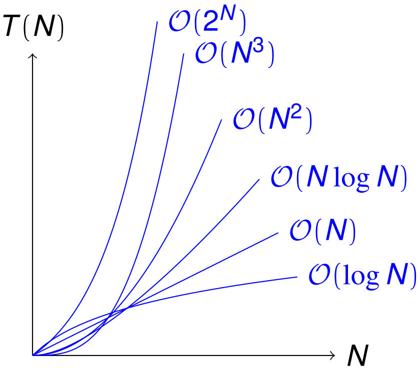
- Ou seja, se um algoritmo é $\mathcal{O}(f(N))$, então há um ponto M a partir do qual o desempenho do algoritmo é limitado a um múltiplo de $f(N)$.

Consequências da definição

- $\mathcal{O}(N^3) + \mathcal{O}(N^2) = \mathcal{O}(N^3)$;
- $\mathcal{O}(N^2) + N = \mathcal{O}(N^2)$;
- $\mathcal{O}(\sum_{i=1}^{k} a_i N^i) = \mathcal{O}(N^k)$.
- $\mathcal{O}(1)$ indica um trecho de programa com custo constante.

Crescimento do esforço computacional

As diversas ordens de algoritmos podem ser esquematizadas como segue:



Observação 1

- Um algoritmo A_1 pode ser mais rápido que outro algoritmo A_2 para pequenos valores de N, e no entanto ser pior para grandes valores de N.
- A análise por notação “BigOh” se preocupa com o comportamento para *grandes* valores de N, e é por isso denominada *análise assintótica*

Observação 2

- Um algoritmo pode ter comportamentos diferentes para diferentes tipos de entradas; por exemplo, ordenação por inserção depende da entrada estar ordenada ou não.
- Em geral a complexidade é considerada *no pior caso*. Usaremos sempre essa convenção neste curso.

Observação 3

- A análise assintótica ignora o comportamento para N pequeno e ignora também custos de programação e manutenção do programa, mas esses fatores podem ser importantes em um problema prático.

Observação 4

- O custo de um algoritmo é sempre “dominado” pelas suas partes com maior custo.
- Em geral, as partes de um algoritmo (ou programa) que consomem mais recursos são os laços, e é neles que a análise assintótica se concentra.
- Note que se dois ou mais laços se sucedem, aquele que tem maior custo “domina” os demais.

Exemplos 1 e 2

- ```
  for i in range(1, n):
      sum += 1
  ```
  Custo é $\mathcal{O}(N)$.

- ```
  for i in range(1, n):
      for j in range(1, n):
          sum += 1
  ```
  Custo é $\mathcal{O}(N^2)$.

Exemplo 3

Consideremos um exemplo, a procura da subsequência contígua máxima (o problema é encontrar uma subsequência contígua de um arranjo de inteiros, tal que a soma de elementos dessa subsequência seja máxima). Por exemplo:

$$-2, \underbrace{11, -4, 13}_{\text{subseq máx}}, -5, 2,$$

Podemos codificar soluções cúbicas, quadráticas e lineares para esse problema.

Exemplo 3

Definição do problema: Seja $A = \{a_1, \ldots, a_n\}$ uma sequência de inteiros com $n \geq 1$. O problema consiste em resolver o problema de otimização:

$$\begin{array}{cc} \max\limits_{l,r} & \sum_{i=l}^{r} a_i \\ \text{tal que} & 1 \leq l \leq r \leq n \end{array}$$

Esta definição exige que a máxima subsequência tenha ao menos um elemento.

Solução cúbica

```
def max_sub_sum(a):
    sequence = len(a)-1, len(a)-1
    maxSum =a[-1] # ultimo elemento
    for i in range(len(a)-1):
        for j in range(i, len(a)):
            thisSum = 0
            for k in range(i,j+1):
                thisSum += a[k]
                if thisSum > maxSum :
                    maxSum   = thisSum
                    sequence = i, j
    return sequence
```

Análise

Essa solução tem custo:

$$
\begin{aligned}
\mathcal{O}\left(\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}\sum_{k=i}^{j}1\right) &= \mathcal{O}\left(\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}(j-i+1)\right) \\
&= \mathcal{O}\left(\sum_{i=0}^{N-1}\frac{(N-i)(N+i-1)}{2}+i(i-N)+(N-i)\right) \\
&= \mathcal{O}(N^3)
\end{aligned}
$$

Somas

Para fazer esse tipo de análise, é importante nos lembrarmos de algumas fórmulas:

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \qquad \sum_{i=0}^{N-1} i = \frac{N(N-1)}{2}$$

$$\sum_{i=1}^{N} i^2 = \frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6} \qquad \sum_{i=0}^{N-1} i^2 = \frac{N^3}{3} - \frac{N^2}{2} + \frac{N}{6}$$

$$\sum_{i=1}^{N} i^3 = \frac{N^4}{4} + \frac{N^3}{2} + \frac{N^2}{4} \qquad \sum_{i=0}^{N-1} i^3 = \frac{N^4}{4} - \frac{N^3}{2} + \frac{N^2}{4}$$

Solução quadrática

```
def max_sub_sum(a):
    sequence = len(a)-1, len(a)-1
    maxSum =a[-1] # ultimo elemento
    for i in range(len(a)-1):
        thisSum = 0
        for j in range(i, len(a)):
            thisSum += a[j]
            if thisSum > maxSum :
                maxSum    = thisSum
                sequence = i, j
    return sequence
```

Solução Linear

Um problema diferente agora, “travando” r:

$$\begin{array}{ll} \max\limits_{l} & S_r = \sum_{i=l}^{r} a_i \\ \text{tal que} & 1 \leq l \leq r \end{array}$$

Evidentemente, a solução do problema original é max S_r.

Solução Linear

$$\begin{array}{ll} \max\limits_{l} & S_r = \sum_{i=l}^{r} a_i \\ \text{tal que} & 1 \leq l \leq r \end{array}$$

É simples resolver cada S_r com complexidade linear.
Do mesmo modo, pode-se varrer sequencialmente cada valor de r de modo a encontrar $\max S_r$, mas isso nos leva novamente a uma complexidade quadrática.

Solução Linear

$$\begin{array}{ll} \max\limits_{l} & S_r = \sum_{i=l}^{r} a_i \\ \text{tal que} & 1 \leq l \leq r \end{array}$$

Dado S_{r-1}, há alguma maneira fácil de se encontrar S_r?

Solução Linear

$$\max_{l} \quad S_r = \sum_{i=l}^{r} a_i$$
$$\text{tal que} \quad 1 \leq l \leq r$$

Dado S_{r-1}, há alguma maneira fácil de se encontrar S_r?
O problema para S_r é o problema para S_{r-1} "aumentado" pelo elemento a_r.

Solução Linear

$$\begin{array}{ll} \max\limits_{l} & S_r = \sum_{i=l}^{r} a_i \\ \text{tal que} & 1 \leq l \leq r \end{array}$$

Dado S_{r-1}, há alguma maneira fácil de se encontrar S_r?
O problema para S_r é o problema para S_{r-1} "aumentado" pelo elemento a_r. Se a soma em S_r inclui qualquer elemento da soma de S_{r-1}, então ela inclui *todos*, caso contrário, S_{r-1} não seria máximo.
Assim, $S_r = \max\{S_{r-1} + a_r, a_r\}$.
Isso pode ser calculado com complexidade *constante*.

Solução linear

```
def max_sub_sum(a):
    soma_r = a[0]
    max_soma = soma_r
    for r in range(1, len(a)):
        if soma_r < 0:
            soma_r = a[r]
        if max_soma < soma_r:
            max_soma = soma_r
    return max_soma
```

“Parêntese”: Programação Dinâmica

A idéia por trás da solução com complexidade linear é observar que para resolver o problema S_r pode-se reaproveitar cálculos de soluções anteriores.

Problemas que apresentam uma sobreposição de sub-problemas podem ser eficientemente abordados por programação dinâmica.

Nota: O termo “programação” aqui vem da área de pesquisa operacional, e *não* do ato de construir um programa de computador.

“Parêntese”: Programação Dinâmica

Encontre o tamanho da maior subsequência (não necessariamente contígua) comum entre duas sequências de caracteres.
Exemplo: a maior subsequência comum entre “`AABKDEQEHUI`” e “`WCBKLQU`” é “`BKQU`”, com 4 caracteres.

“Parêntese”: Programação Dinâmica

Encontre o tamanho da maior subsequência (não necessariamente contígua) comum entre duas sequências de caracteres.

Exemplo: a maior subsequência comum entre “`AABKDEQEHUI`” e “`WCBKLQU`” é “`BKQU`”, com 4 caracteres.

Uma possível solução é enumerar todas as possíveis subsequências de uma das sequências e verificar se ela está presente na outra.

Mas tal enumeração tem complexidade exponencial!

“Parêntese”: Programação Dinâmica

Sobreposição de sub-problemas:
Sejam $A_{1:n} = \{a_1, \ldots, a_n\}$ e $B_{1:m} = \{b_1, \ldots, b_m\}$ as sequências em questão.
$L[A_{1:n}, B_{1:m}]$ o comprimento da maior subsequência comum.

“Parêntese”: Programação Dinâmica

Sobreposição de sub-problemas:
Sejam $A_{1:n} = \{a_1, \ldots, a_n\}$ e $B_{1:m} = \{b_1, \ldots, b_m\}$ as sequências em questão.
$L[A_{1:n}, B_{1:m}]$ o comprimento da maior subsequência comum.
Se $a_n = b_m$, então

$$L[A_{1:n}, B_{1:m}] = 1 + L[A_{1:n-1}, B_{1:m-1}]$$

Se $a_n \neq b_m$, então

$$L[A_{1:n}, B_{1:m}] = \max\{L[A_{1:n}, B_{1:m-1}], L[A_{1:n-1}, B_{1:m}]\}$$

É possível usar soluções prévias de sub-problemas para resolver um problema maior.
Note que neste caso é necessário armazenar diversas soluções prévias.

“Parêntese”: Programação Dinâmica

```
def max_sub_seq(A, B):
    if(len(A)<len(B)): A, B = B, A
    n = len(A)
    m = len(B)
    L = [0]*(m+1)
    for i in range(n):
        for j in range(m):
            if A[i]==B[j]:
                L[j+1] = 1 + L[j]
            else:
                if L[j+1]<L[j]:
                    L[j+1] = L[j]
    return L[m]
```

“Parêntese”: Programação Dinâmica

Ao final de cada iteração do laço externo a variável L contém na posição j todas as soluções dos problemas $L\left[A_{1:i}, B_{1:j}\right]$.
O Laço interno atualiza esta variável usando as soluções de $L\left[A_{1:i-1}, B_{1:j}\right]$
Complexidade Quadrática: $\mathcal{O}(NM)$
Uso de memória: $\mathcal{O}(M)$

Complexidade logarítmica

- Vimos até agora exemplos de complexidade polinomial, ou seja, $\mathcal{O}(N^\alpha)$.
- Um tipo importante de algoritmo é o que tem complexidade logarítmica, ou seja, $\mathcal{O}(\log N)$.
- IMPORTANTE: a base do logaritmo não importa, pois

$$\mathcal{O}(\log_\alpha N) = \mathcal{O}\left(\frac{\log_\beta N}{\log_\beta \alpha}\right) = \mathcal{O}(\log_\beta N).$$

Exemplo 1

- Considere que uma variável x é inicializada com 1 e depois é multiplicada por 2 um certo número K de vezes.
- Qual é o número K^* tal que x é maior ou igual a N?
- Esse problema pode ser entendido como uma análise do seguinte laço:

```
x=1
while x<N:
        ...
        x*=2
        ...
```

Análise

- A questão é quantas iterações serão realizadas.
- Note que se o interior do laço tem custo constante $\mathcal{O}(1)$, então o custo total é $\mathcal{O}(K^*)$.
- A solução é simples: queremos encontrar K tal que:

$$2^K \geq N \Rightarrow K \geq \log_2 N,$$

e portanto $K^* = \lceil \log N \rceil$ garante que x é maior ou igual a N.

Exemplo 2

- Considere agora que uma variável $\mathtt{x}$ é inicializada com N e depois é dividida por 2 um certo número K de vezes.
- Qual é o número K^* tal que $\mathtt{x}$ é menor ou igual a 1?
- De novo podemos entender esse problema como o cálculo do número de iterações de um laço:

```
x=N
while x>1:
        ...
        x//=2
        ...
```

Análise

- A solução é dada por:

$$N\left(\frac{1}{2}\right)^K \leq 1 \Rightarrow N \leq 2^K \Rightarrow 2^K \geq N \Rightarrow K \geq \log_2 N.$$

- De novo, temos que $K^* = \lceil \log N \rceil$ é a solução.

Logaritmos

- Nessas duas situações a complexidade total é $\mathcal{O}\left(\lceil \log N \rceil\right)$, supondo que o custo de cada iteração é constante.
- Note que $\lceil \log N \rceil \leq (\log N) + 1$ e portanto:

$$\mathcal{O}\left(\lceil \log N \rceil\right) = \mathcal{O}\left((\log N) + 1\right) = \mathcal{O}\left(\log N\right).$$

Busca não-informada

- Considere um arranjo de tamanho N e suponha que queremos encontrar o índice de um elemento `x`.
- O algoritmo de *busca sequencial* simplesmente varre o arranjo do início ao fim, até encontrar o elemento procurado.

Busca sequencial

```
def busca_seq(a, chave):
    for i, v in enumerate(a):i
    if v==chave:
        return i
```

Custo

- O custo desse algoritmo no pior caso é $\mathcal{O}(N)$, onde N é o tamanho da sequência `a`.
- *Se* `x` ocorre *uma e somente uma* vez em `a` e sua posição está *uniformemente distribuída*, então podemos dizer que o custo médio é proporcional a $\frac{N}{2}$, ainda de ordem $\mathcal{O}(N)$ (note que as suposições aqui feitas são muito fortes!).

Busca binária

- Suponha agora que o arranjo está ordenado. Nesse caso podemos dividir o problema em dois a cada passo, verificando se o elemento está na metade superior ou inferior do arranjo em consideração.

Algoritmo

```
def busca_binaria(a, chave):
    low = 0
    high = len(a)-1
    while low <= high:
        meio = (low+high)//2;
        if a[meio]==chave:
            return meio
        elif a[meio]<chave:
            low = meio + 1
        else:
            high = meio -1
```

Custo

- Esse programa faz uma iteração que recebe um arranjo de tamanho N, depois *menor ou igual a* $\frac{N}{2}$ (verifique!), depois menor que $\frac{N}{4}$, e assim sucessivamente; no pior caso isso prossegue até que o tamanho seja 1.
- Portanto o número de iterações é $\mathcal{O}\left(\lceil \log N \rceil\right)$ e o custo total é $\mathcal{O}\left(\log N\right)$, já que o custo de cada iteração é constante.

Recursão

- Um procedimento recursivo é um procedimento que chama a si mesmo durante a execução.
- Recursão é uma técnica importante e poderosa; algoritmos recursivos devem ter sua complexidade assintótica analisada com cuidado.

Definição

Considere como exemplo um programa que calcula fatorial:

```
def fatorial(x): return 1 if x <= 1 else x*fatorial
```

Análise

- O método funciona para $x = 1$; além disso, funciona para $x \geq 1$ se funciona para $(x - 1)$. Por indução finita, o método calcula os fatoriais corretamente.
- O caso $x = 1$, em que ocorre a parada do algoritmo, é chamado *caso base* da recursão.

Árvore de recursão

$$\begin{array}{c} \text{fat. x} \\ \downarrow \\ \text{fat.(x-1)} \\ \downarrow \\ \text{fat.(x-2)} \\ \downarrow \\ \vdots \\ \downarrow \\ \text{fat. 2} \\ \downarrow \\ \text{fat. 1} \end{array}$$

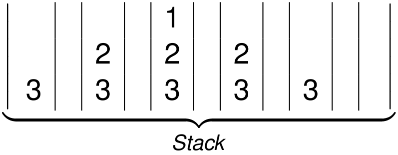
Busca binária recursiva

```
def busca_bin(a, x, low, high):
   if low>high : return
   mid=(low+high)//2
   if a[mid]<x : return busca_bin(a,x,(mid+1),high)
   elif a[mid]>x : return busca_bin(a,x,low,(mid-1))
   else : return mid
```

Controle via Stack

- Um procedimento recursivo cria várias “cópias” de suas suas variáveis locais no Stack à medida que suas chamadas são feitas.
- Sempre é necessário avaliar a complexidade de um procedimento recursivo com cuidado, pois um procedimento recursivo pode “esconder” um laço bastante complexo.
- Consideremos a função recursiva fatorial. Uma execução típica seria:

$$\text{fatorial}(3) \longrightarrow \text{fatorial}(2) \longrightarrow \text{fatorial}(1)$$



Custo de fatorial

O custo para uma chamada `fatorial(N)` é $\mathcal{O}(N)$. Uma solução iterativa equivalente seria:

```
def fatorial(n):
    fat=1;
    for i in range(2,n+1): fat *= i
    return fat
```

Essa solução tem claramente custo $\mathcal{O}(N)$.

Recursão de cauda (*tail call recursion*)

- Quando a *última* operação em um corpo de uma função é a chamada de outra função, o estado *não precisa ser preservado*.
- Em particular, recursões podem ser convertidas em laços simples.

Por exemplo,

function $F(\boldsymbol{X})$
    if $C(\boldsymbol{X})$ **then**
        return $E(\boldsymbol{X})$
    else
        return $F(G(\boldsymbol{X}))$
    end if
end function

Recursão de cauda (*tail call recursion*)

Quando a *última* operação em um corpo de uma função é a chamada de outra função, o estado *não precisa ser preservado*.

Em particular, recursões podem ser convertidas em laços simples.

Por exemplo,

function $F(\boldsymbol{X})$
    if $C(\boldsymbol{X})$ **then**
        return $E(\boldsymbol{X})$
    else
        return $F(G(\boldsymbol{X}))$
    end if
end function

É equivalente a:

function $F(\boldsymbol{X})$
    while NOT $C(\boldsymbol{X})$ **do**
        $\boldsymbol{X} \leftarrow G(\boldsymbol{X})$
    end while
    return $E(\boldsymbol{X})$
end function

Recursão de cauda (*tail call recursion*)

Quando a *última* operação em um corpo de uma função é a chamada de outra função, o estado *não precisa ser preservado*.

Em particular, recursões podem ser convertidas em laços simples.

Por exemplo,

function $F(\boldsymbol{X})$
  if $C\left(\boldsymbol{X}\right)$ **then**
    return $E\left(\boldsymbol{X}\right)$
  else
    return $F\left(G\left(\boldsymbol{X}\right)\right)$
  end if
end function

É equivalente a:

function $F(\boldsymbol{X})$
  while NOT $C\left(\boldsymbol{X}\right)$ **do**
    $\boldsymbol{X} \leftarrow G\left(\boldsymbol{X}\right)$
  end while
  return $E\left(\boldsymbol{X}\right)$
end function

Não há custo adicional de armazenamento!

Recursão de cauda (*tail call recursion*)

Atenção! Nem toda linguagem faz otimização de chamada de cauda (*taill call*)!

Linguagens que suportam otimização de chamada de cauda (em 2015!):

- C, C++ (compiladores usuais: GCC, msvc, ICC, Clang, etc.)
- Java (IBM J9!)
- Ruby (opcional, normalmente desativada)
- Funcionais em geral (Erlang, Haskell, F# Lisp-*like*, etc.)

Linguagens que *não* suportam otimização de chamada de cauda (em 2015!):

- Java (Oracle HotSpot, OpenJDK, Dalvik)
- Python (mas há progresso recente em *pypy*)
- C#, VB.net
- PHP

Recursão de cauda (*tail call recursion*)

Atenção! Nem toda linguagem faz otimização de chamada de cauda (*taill call*)!

Linguagens que suportam otimização de chamada de cauda (em 2015!):

- C, C++ (compiladores usuais: GCC, msvc, ICC, Clang, etc.)
- Java (IBM J9!)
- Ruby (opcional, normalmente desativada)
- Funcionais em geral (Erlang, Haskell, F# Lisp-*like*, etc.)

Linguagens que *não* suportam otimização de chamada de cauda (em 2015!):

- Java (Oracle HotSpot, OpenJDK, Dalvik)
- Python (mas há progresso recente em *pypy*)
- C#, VB.net
- PHP

Na dúvida otimize você mesmo!

Mergesort

- Consideremos o algoritmo de *ordenação por união (mergesort)*.
- Considere um arranjo `a` de tamanho N. Para ordená-lo, divida o arranjo em 2 partes, ordene cada uma e una as duas partes (com custo $\mathcal{O}(N)$). O algoritmo completo é mostado a seguir.

Algoritmo - parte 1

```
void mergesort(a) {
    temp = [None]*len(a)
    def mergesort_rec(left, right):
        if (left+1) < right:
            center = (left+right)//2
            mergesort_rec(left, center)
            mergesort_rec(center, right)
            merge(left, center, right)
```

Continua...

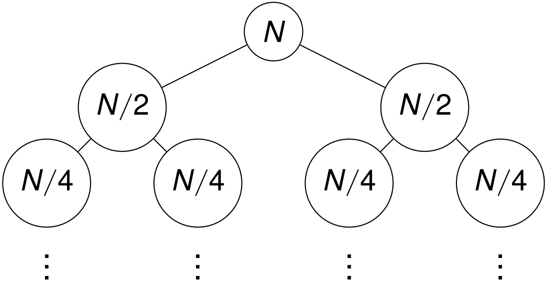
Algoritmo - parte 2

Esta função está definida *dentro* do escopo de `mergesort`:

```python
def merge(left, mid, right):
    i_left = left    # iterador da 1a metade do vetor de entrada
    i_right = mid    # iterador da 2a metade do vetor de entrada
    i_out = left     # iterador da posicao de saida
    while i_left < mid and i_right < right:
        if a[i_left] < a[i_right]:
            temp[i_out] = a[i_left]
            i_left += 1
        else:
            temp[i_out] = a[i_right]
            i_right += 1
        i_out += 1
    while i_left < mid:
        temp[i_out] = a[i_left]
        i_left += 1
        i_out += 1
    while i_right < right:
        temp[i_out] = a[i_right]
        i_right += 1
        i_out += 1
    # copia o conteudo em temp de volta
    for i in range(left, right):
        a[i] = temp[i]
```

Custo

Com um arranjo de tamanho N, temos a seguinte árvore de recursão (estamos assumindo que N é uma potência de 2):



$\Longleftarrow$ Custo $\mathcal{O}(N)$

$\Longleftarrow$ Custo $\mathcal{O}(N)$

$\Longleftarrow$ Custo $\mathcal{O}(N)$

Relação de recorrência

- O custo em cada *nível*, $\mathcal{O}(N)$, é o custo da combinação das soluções. Suponha que N seja uma potência de 2. Nesse caso teremos $\log_2 N$ níveis, cada um com custo $\mathcal{O}(N)$. O custo total é $\mathcal{O}(N \log N)$.
- Uma maneira geral de calcular esse custo é considerar que o custo total $T(N)$ é regulado pela seguinte relação de recorrência:

$$T(N) = 2 \cdot T\left(\frac{N}{2}\right) + C \cdot N.$$

Solução

Podemos escrever:

$$\begin{aligned} T\left(\frac{N}{2}\right) &= 2 \cdot T\left(\frac{N}{4}\right) + C \cdot \left(\frac{N}{2}\right) \\ \Longrightarrow T(N) &= 2 \cdot \left(2 \cdot T\left(\frac{N}{4}\right) + C \cdot \left(\frac{N}{2}\right)\right) + C \cdot N \\ &= 4 \cdot T\left(\frac{N}{4}\right) + 2 \cdot C \cdot N. \end{aligned}$$

Seguindo esse raciocínio, chegamos a

$$T(N) = 2^K \cdot T\left(\frac{N}{2^K}\right) + K \cdot C \cdot N$$

para uma recursão de K níveis.

Solução

Sabemos que o número total de níveis é $\log_2 N$ (para N potência de 2),

$$
\begin{aligned}
T(N) &= 2^{\log_2 N} \cdot T\left(\frac{N}{2^{\log_2 N}}\right) + C \cdot N \cdot \log_2 N \\
&= N \cdot T\left(\frac{N}{N}\right) + C \cdot N \cdot \log N \\
&= C' \cdot N + C \cdot N \cdot \log N = O(N \cdot \log N),
\end{aligned}
$$

pois temos $T(1) = \mathcal{O}(1)$ nesse algoritmo.

Solução

Se a suposição que N é uma potência de 2 for removida, teremos um custo de no máximo $\mathcal{O}(N)$ em cada nível, e um número máximo de $(1 + \log_2 N)$ níveis. Nesse pior caso,

$$
\begin{aligned}
T(N) &= 2^{1+\log_2 N} \cdot T\left(\frac{N}{2^{\log_2 N}}\right) + C \cdot N \cdot (1 + \log_2 N) \\
&= 2 \cdot N \cdot T(1) + C \cdot N \cdot \log N + C \cdot N \\
&= O(N \log N),
\end{aligned}
$$

se considerarmos que $T(1)$ é uma constante (já que esse custo nunca é realmente atingido, pois todas as recursões “param” quando $N = 1$).

Divisão e conquista

Existem muitos problemas que podem ser resolvidos pelo método genérico de “divisão-e-conquista”, no qual o problema é dividido em partes que são resolvidas independentemente e depois combinadas. Esquematicamente temos:

- Problema original dividido em A sub-problemas, cada um com entrada N/B.
- Cada sub-problema dividido em A sub-problemas, cada um com entrada $(N/B)/B$.
- etc etc
- Até que cada sub-problema seja resolvido.

Custo

- Podemos em geral obter uma relação de recorrência:

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + c \cdot f(N),$$

  onde $f(N)$ é o custo de combinar os subproblemas.
- Um resultado geral é o seguinte: A relação $T(N) = A \cdot T\left(\frac{N}{B}\right) + cN^L$, com $T(1) = \mathcal{O}(1)$, tem solução

$$T(N) = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L \end{cases}$$

Exemplo

- Suponha que uma recursão resolve a cada nível 3 subproblemas, cada um com metade do tamanho de chamada, e com custo linear para combinar subproblemas.
- Ou seja, $A = 3$, $B = 2$ e $L = 1$.
- Como $A > B^L$, sabemos que

$$T(N) = \mathcal{O}\left(N^{\log_2 3}\right) = \mathcal{O}(N^{1.59}).$$

Análise

- Supondo que N é potência de 2, temos no primeiro nível um custo maior ou igual que $c \cdot N$;
- no segundo nível um custo maior ou igual que $3 \cdot c \cdot \frac{N}{2}$;
- no terceiro nível um custo $\leq 3^2 \cdot c \cdot \frac{N}{2^2}$.

Análise detalhada

Com essas considerações, chegamos a

$$
\begin{aligned}
T(N) &= (\text{custo total das } 3^{\log_2 N}\text{folhas} = 3^{\log_2 N}) + \\
&\quad \left(c \cdot N + 3 \cdot c \cdot \frac{N}{2} + 3^2 \cdot c \cdot \frac{N}{2^2} + \cdots\right) \\
&= N^{\log_2 3} + c \cdot N \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i,
\end{aligned}
$$

onde k é o número de níveis da recursão, igual a $\log_2 N$.

Solução

$$
\begin{aligned}
T(N) &= N^{\log_2 3} + c \cdot N \cdot \sum_{i=0}^{\log_2 N-1} \left(\frac{3}{2}\right)^i \\
&= N^{\log_2 3} + cN \frac{\left(\frac{3}{2}\right)^{\log_2 N} - 1}{\frac{3}{2} - 1} \\
&= N^{\log_2 3} + (cN)\left(2\frac{3^{\log_2 N}}{2^{\log_2 N}} - 2\right) \\
&= N^{\log_2 3} + \frac{cN2}{N} N^{\log_2 3} - 2cN \\
&= N^{\log_2 3} + 2cN^{\log_2 3} - 2cN \\
&= O\left(N^{\log_2 3}\right),
\end{aligned}
$$

onde usamos

$$
\sum_{i=0,a>0,a\neq 1}^{k} a^i = \frac{a^{k+1} - 1}{a - 1}.
$$

Teorema

- $\mathcal{O}(N^a - N^b) = \mathcal{O}(N^a)$ se $a > b$.
  - Prova: $cN^a \geq c(N^a - N^b)$.
  - Além disso, $N^{a-\epsilon} < N^a - N^b$ para qualquer $\epsilon > 0$, quando N cresce.
    (Suponha $N^a/N^\epsilon \geq (N^a - N^b)$; então $N^\epsilon(1 - N^{b-a}) \leq 1$, o que é impossível para N grande.)

N qualquer

Essa solução assume que N é potência de 2; se isso não ocorre, o número de níveis é menor que $(\log_2 N + 1)$ e a complexidade ainda é $\mathcal{O}\left(N^{\log_2 3}\right)$.

Resumindo:

1. dado um programa recursivo, determine a relação de recorrência que rege o programa;
2. substitua os custos assintóticos em notação $\mathcal{O}(\cdot)$ por funções;
3. obtenha expressões para $T(N)$, resolvendo somatórios ou produtórios.

Resultado geral

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + cN^L = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L, \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L, \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L. \end{cases}$$

Temos:

$$T(N) = cN^L + Ac\frac{N^L}{B^L} + A^2 c\frac{N^L}{B^{2L}} + \cdots$$

Análise detalhada

$$
\begin{aligned}
T(N) &= AT(N/B) + cN^L \\
&= A(AT(N/B^2) + c(N/B)^L) + cN^L \\
&= A(A(AT(N/B^3) + c(N/B^2)^L) + c(N/B)^L) + cN^L \\
&= A^3T(N/B^3) + A^2c(N/B^2)^L + Ac(N/B)^L + cN^L \\
&= A^3T(N/B^3) + cN^L(1 + A/B^L + (A/B^L)^2).
\end{aligned}
$$

Portanto, para k níveis:

$$
\begin{aligned}
T(N) &= A^kT(N/B^k) + cN^L(1 + A/B^L + \cdots + (A/B^L)^{k-1}) \\
&= A^kT(N/B^k) + cN^L\sum_{i=0}^{k-1}(A/B^L)^i.
\end{aligned}
$$

Análise detalhada: Caso 1

Se $A/B^L = 1$, temos:

$$\begin{aligned} T(N) &= A^k T(N/B^k) + cN^L \sum_{i=0}^{k-1} 1 \\ &= A^k T(N/B^k) + cN^L k. \end{aligned}$$

Tomando $k = \log_B N$, temos:

$$\begin{aligned} T(N) &= A^{\log_B N} T(N/B^{\log_B N}) + cN^L \log_B N \\ &= N^{\log_B A} T(N/N) + cN^L \log_B N \\ &= N^{\log_B A} c' + cN^L \log_B N \\ &= c' N^L + cN^L \log_B N \end{aligned}$$

pois $\log_B A = L$, e portanto obtemos $\mathcal{O}(N^L \log N)$.

Análise detalhada: Caso 2

Se $A/B^L < 1$, temos:

$$\begin{aligned} T(N) &= A^k T(N/B^k) + cN^L \sum_{i=0}^{k-1} (A/B^L)^i \\ &= A^k T(N/B^k) + cN^L \left(\frac{(A/B^L)^k - 1}{A/B^L - 1} \right). \end{aligned}$$

Tomando $k = \log_B N$, temos:

$$\begin{aligned} T(N) &= A^{\log_B N} T(N/B^{\log_B N}) + cN^L \left(\frac{1 - (A/B^L)^{\log_B N}}{1 - A/B^L} \right) \\ &= N^{\log_B A} T(N/N) + c'' N^L (1 - (A/B^L)^{\log_B N}) \\ &= N^{\log_B A} c' + c'' N^L - c'' N^{\log_B A} \end{aligned}$$

e notando que $\log_B A < L$, obtemos $\mathcal{O}(N^L)$.

Análise detalhada: Caso 3

Se $A/B^L > 1$, temos:

$$\begin{aligned} T(N) &= A^k T(N/B^k) + cN^L \sum_{i=0}^{k-1} (A/B^L)^i \\ &= A^k T(N/B^k) + cN^L \left(\frac{1-(A/B^L)^k}{1-A/B^L} \right). \end{aligned}$$

Tomando $k = \log_B N$, temos:

$$\begin{aligned} T(N) &= A^{\log_B N} T(N/B^{\log_B N}) + cN^L \left(\frac{(A/B^L)^{\log_B N} - 1}{A/B^L - 1} \right) \\ &= N^{\log_B A} T(N/N) + c'' N^L ((A/B^L)^{\log_B N} - 1) \\ &= N^{\log_B A} c' + c'' N^{\log_B A} - c'' N^L \end{aligned}$$

e notando que $\log_B A > L$, obtemos $\mathcal{O}(N^{\log_B A})$.

Exemplo: Inversão de Matrizes

- O algoritmo de Strassen para inversão de matrizes $N \times N$ divide o número de linhas pela metade, mas realiza sete chamadas recursivas por vez, com custo $\mathcal{O}(N^2)$ de combinação.
- Portanto o custo de inversão de uma matriz é $\mathcal{O}(N^{\log_2 7})$.

Little oh

- $T(N)$ é $o(f(N))$ se existe M positivo tal que
$$T(N) \leq \epsilon |f(N)|$$
para todo $N \geq M$ e todo $\epsilon > 0$.
- Isto é, $\lim_{x \to \infty} f(x)/g(x) = 0$.
- Little oh não é muito usado (não vamos usar nesse curso).

Big Omega e Big Theta

- $T(N)$ é $\Omega(f(N))$ se existem k e M positivos tal que
$$T(N) \geq kf(N)$$
para todo $N \geq M$.
- $T(N)$ é $\Theta(f(N))$ se existem k_1, k_2 e M positivos tal que
$$k_1 f(N) \leq T(N) \leq k_2 f(N)$$
para todo $N \geq M$.

Exercício

Prove: se $f(N)$ é $\Theta(N^L)$, então

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + f(N) = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L, \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L, \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L. \end{cases}$$

Formulário

- $b^{\log_c a} = a^{\log_c b}$;
- $\log_b a = \frac{\log_c a}{\log_c b}$;
- $\sum_{i=1}^{n} f(i) - f(i-1) = f(n) - f(0)$;
- $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$;
- $\sum_{i=0}^{N-1} i = \frac{N(N-1)}{2}$;
- $\sum_{i=1}^{N} i^2 = \frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6}$;
- $\sum_{i=0}^{N-1} i^2 = \frac{N^3}{3} - \frac{N^2}{2} + \frac{N}{6}$;
- $\sum_{i=1}^{N} i^3 = \frac{N^4}{4} + \frac{N^3}{2} + \frac{N^2}{4}$;
- $\sum_{i=0}^{N-1} i^3 = \frac{N^4}{4} - \frac{N^3}{2} + \frac{N^2}{4}$;
- $\sum_{i=0,a>0,a\neq 1}^{k} a^i = \frac{a^{k+1}-1}{a-1}$
- $\sum_{i=0,a\in[0,1]}^{\infty} a^i = \frac{1}{1-a}$
- $\sum_{i=0,a>0,a\neq 1}^{n} i \cdot a^i = \frac{a-(n+1)a^{n+1}+na^{n+2}}{(1-a)^2}$

Quicksort

- O algoritmo mais usado para ordenação é o quicksort, que tem pior caso $\mathcal{O}(N^2)$!
- Seu caso médio, quando a entrada é uma permutação uniforme, é $\mathcal{O}(N \log N)$. Na prática o quicksort é muito rápido para arranjos não ordenados.

Algoritmo

```
def quicksort(s, a, b):
    if a>=b:
        return
    p = s[b]      # pivot
    l = a
    r = b-1
    while l<=r :
        while l<=r and s[l]<=p:
            l = l+1
        while l<=r and s[r]>=p:
            r = r - 1
        if l<r:
            temp = s[l]
            s[l] = s[r]
            s[r]=temp
    s[b] = s[l]
    s[l] = p
    quicksort(s, a, (l-1))
    quicksort(s, (l+1), b)
```

Análise no caso médio: quicksort

$$T(N) = C \cdot N + T(i) + T(N - i)$$

Onde i é a posição do pivô.

Análise no caso médio: quicksort

$$T(N) = C \cdot N + T(i) + T(N - i)$$

Onde i é a posição do pivô.

- No caso médio, a entrada pode ser vista como uma permutação aleatória do vetor ordenado.
- A posição final de cada partição tem distribuição *uniforme* no vetor, *independentemente* da posição original do pivô (verifique!).

Análise no caso médio: quicksort

$$T(N) = C \cdot N + T(i) + T(N - i)$$

Onde i é a posição do pivô.

- No caso médio, a entrada pode ser vista como uma permutação aleatória do vetor ordenado.
- A posição final de cada partição tem distribuição *uniforme* no vetor, *independentemente* da posição original do pivô (verifique!). Assim,

$$\mathbb{E}\left\langle T(N)\right\rangle = C \cdot N + \frac{2}{N}\sum_{i=0}^{N-1}\mathbb{E}\left\langle T(i)\right\rangle$$

Análise no caso médio: quicksort

Seja

$$A(N) = C \cdot N + \frac{2}{N} \sum_{i=0}^{N-1} A(i)$$

Análise no caso médio: quicksort

Seja

$$A(N) = C \cdot N + \frac{2}{N} \sum_{i=0}^{N-1} A(i)$$

$$N \cdot A(N) = C \cdot N^2 + 2 \sum_{i=0}^{N-1} A(i)$$

Análise no caso médio: quicksort

Seja

$$A(N) = C \cdot N + \frac{2}{N} \sum_{i=0}^{N-1} A(i)$$

$$N \cdot A(N) = C \cdot N^2 + 2 \sum_{i=0}^{N-1} A(i)$$

Logo,

$$(N-1) \cdot A(N-1) = C \cdot (N-1)^2 + 2 \sum_{i=0}^{N-2} A(i)$$

Análise no caso médio: quicksort

Seja

$$A(N) = C \cdot N + \frac{2}{N} \sum_{i=0}^{N-1} A(i)$$

$$N \cdot A(N) = C \cdot N^2 + 2 \sum_{i=0}^{N-1} A(i)$$

Logo,

$$(N-1) \cdot A(N-1) = C \cdot (N-1)^2 + 2 \sum_{i=0}^{N-2} A(i)$$

$$N \cdot A(N) = (N-1) \cdot A(N-1) + 2C \cdot N - C$$

Análise no caso médio: quicksort

$$N \cdot A(N) = (N-1) \cdot A(N-1) + 2C \cdot N - C$$

Dividindo-se ambos os lados por $N(N+1)$,

Análise no caso médio: quicksort

$$N \cdot A(N) = (N-1) \cdot A(N-1) + 2C \cdot N - C$$

Dividindo-se ambos os lados por $N(N+1)$,

$$\begin{aligned}
\frac{A(N)}{N+1} &= \frac{A(N-1)}{N} + \frac{2C}{N+1} - \frac{C}{N(N+1)} \\
\frac{A(N-1)}{N} &= \frac{A(N-2)}{N-1} + \frac{2C}{N} - \frac{C}{(N-1)N} \\
\frac{A(N-2)}{N-1} &= \frac{A(N-3)}{N-2} + \frac{2C}{N-1} - \frac{C}{(N-2)(N-1)}
\end{aligned}$$

Análise no caso médio: quicksort

$$N \cdot A(N) = (N-1) \cdot A(N-1) + 2C \cdot N - C$$

Dividindo-se ambos os lados por $N(N+1)$,

$$\begin{aligned}
\frac{A(N)}{N+1} &= \frac{A(N-1)}{N} + \frac{2C}{N+1} - \frac{C}{N(N+1)} \\
\frac{A(N-1)}{N} &= \frac{A(N-2)}{N-1} + \frac{2C}{N} - \frac{C}{(N-1)N} \\
\frac{A(N-2)}{N-1} &= \frac{A(N-3)}{N-2} + \frac{2C}{N-1} - \frac{C}{(N-2)(N-1)}
\end{aligned}$$

Somando-se e desprezando-se os termos $O(C/N^2)$,

$$\frac{A(N)}{N+1} = \frac{A(1)}{2} + O\left(2C \sum_{i=3}^{N+1} \frac{1}{i}\right)$$

Análise no caso médio: quicksort

$$\frac{A(N)}{N+1} = \frac{A(1)}{2} + O\left(2C\sum_{i=3}^{N+1}\frac{1}{i}\right)$$

Mas sabe-se que $\sum_{i=3}^{N+1} 1/i = O(\ln N)$ (Soma de Euler)

Análise no caso médio: quicksort

$$\frac{A(N)}{N+1} = \frac{A(1)}{2} + O\left(2C\sum_{i=3}^{N+1}\frac{1}{i}\right)$$

Mas sabe-se que $\sum_{i=3}^{N+1} 1/i = O(\ln N)$ (Soma de Euler)

$$\frac{A(N)}{N+1} = \frac{A(1)}{2} + O(\ln N)$$

Análise no caso médio: quicksort

$$\frac{A(N)}{N+1} = \frac{A(1)}{2} + O\left(2C\sum_{i=3}^{N+1}\frac{1}{i}\right)$$

Mas sabe-se que $\sum_{i=3}^{N+1} 1/i = O(\ln N)$ (Soma de Euler)

$$\frac{A(N)}{N+1} = \frac{A(1)}{2} + O(\ln N)$$

$$A(N) = O(N \ln N)$$

Multiplicação de grandes números: Algoritmo de Karatsuba

Sejam os números a serem multiplicados x e y, aproximadamente da mesma ordem (número de dígitos), expressos como

$$x = x_h \cdot B + x_l$$

$$y = y_h \cdot B + y_l$$

Onde x_h, x_l, y_h, y_l têm *metade* dos dígitos de x e y e B é uma potência apropriada da base do sistema de numeração. Assim,

$$x \cdot y = (x_h \cdot y_h)B^2 + (x_h \cdot y_l + x_l \cdot y_h)B + x_l \cdot y_l$$

Multiplicação de grandes números: Algoritmo de Karatsuba

Sejam os números a serem multiplicados x e y, aproximadamente da mesma ordem (número de dígitos), expressos como

$$x = x_h \cdot B + x_l$$

$$y = y_h \cdot B + y_l$$

Onde x_h, x_l, y_h, y_l têm *metade* dos dígitos de x e y e B é uma potência apropriada da base do sistema de numeração. Assim,

$$x \cdot y = (x_h \cdot y_h)B^2 + (x_h \cdot y_l + x_l \cdot y_h)B + x_l \cdot y_l$$

A multiplicação convencional ($O(N^2)$, onde N é o número de dígitos) pode ser decomposta em *quatro* multiplicações com *metade* do tamanho da original.

Multiplicação de grandes números: Algoritmo de Karatsuba

Anatoly Karatsuba:

$$x \cdot y = z_h \cdot B^2 + z_m \cdot B + z_l$$

onde

$$\begin{aligned} z_h &= x_h \cdot y_h \\ z_l &= x_l \cdot y_l \\ z_m &= (x_h + x_l) \cdot (y_h + y_l) - z_h - z_l \end{aligned}$$

A multiplicação por Karatsuba converte uma multiplicação em *três* sub-multiplicações, cada uma com *metade* do tamanho da original, e algumas somas e subtrações ($O(N)$).

$$T(N) = 3T(N/2) + C \cdot N$$

Multiplicação de grandes números: Algoritmo de Karatsuba

$$T(N) = 3T(N/2) + C \cdot N$$

Multiplicação de grandes números: Algoritmo de Karatsuba

$$T(N) = 3T(N/2) + C \cdot N$$

$$T(N) = O(N^{\log_2 3})$$

Note que $O(N^{\log_2 3})$ é melhor que $O(N^2)$.