MC458 — Projeto e Análise de Algoritmos I

C.C. de Souza C.N. da Silva O. Lee P.J. de Rezende

1º. Semestre de 2018

Antes de mais nada. . .

- Estes slides são fruto de autoria colaborativa de vários docentes do IC (*).
- Várias pessoas colaboraram direta ou indiretamente para a preparação deste material, cedendo gentilmente seus arquivos digitais ou o seu tempo para fazer correções e dar sugestões. Entre elas:
  - Cândida N. da Silva (*)
  - Cid C. de Souza (*)
  - Célia P. de Mello
  - José Coelho de Pina
  - Orlando Lee (*)
  - Paulo Feofiloff
  - Pedro J. de Rezende (*)
  - Ricardo Dahab
  - Zanoni Dias
- Enfatizemos: este material é apenas um guia e não deve ser usado como única fonte de estudo. Para isso use a bibliografia da disciplina.

Introdução

O que veremos nesta disciplina?

- Como provar a "corretude" de um algoritmo
- Estimar a quantidade de recursos (temp, memória) de um algoritmo = análise de complexidade
- Técnicas e idéias gerais de projeto de algoritmos: divisão-e-conquista, programação dinâmica, algoritmos gulosos etc
- Tema recorrente: natureza recursiva de vários problemas
- A dificuldade intrínseca de vários problemas: inexistência de soluções eficientes

Algoritmos

O que é um algoritmo?

Informalmente, um algoritmo é um procedimento computacional bem definido que:

- recebe um conjunto de valores como entrada e
- produz um conjunto de valores como saída.

Equivalentemente, um algoritmo é uma ferramenta para resolver um problema computacional. Este problema define a relação precisa que deve existir entre a entrada e a saída do algoritmo.

Exemplos de problemas: teste de primalidade

Problema: determinar se um dado número é primo.

Exemplo:

Entrada: 9411461

Saída: É primo.

Exemplo:

Entrada: 8411461

Saída: Não é primo.

Exemplos de problemas: ordenação

Definição: um vetor $A[1 \ldots n]$ é crescente se $A[1] \leq \ldots \leq A[n]$.

Problema: rearranjar um vetor $A[1 \ldots n]$ de modo que fique crescente.

Entrada:

1										n
33	55	33	44	33	22	11	99	22	55	77

Saída:

1										n
11	22	22	33	33	33	44	55	55	77	99

Instância de um problema

Uma instância de um problema é um conjunto de valores que serve de entrada para esse.

Exemplo:
Os números 9411461 e 8411461 são instâncias do problema de primalidade.

Exemplo:
O vetor

1										n
33	55	33	44	33	22	11	99	22	55	77

é uma instância do problema de ordenação.

A importância dos algoritmos para a computação

- Onde se encontram aplicações para o uso/desenvolvimento de algoritmos “eficientes”?
  - projetos de genoma de seres vivos
  - rede mundial de computadores
  - comércio eletrônico
  - planejamento da produção de indústrias
  - logística de distribuição
  - *games* e filmes
  - ...

Dificuldade intrínseca de problemas

- Infelizmente, existem certos problemas para os quais **não se conhece** algoritmos eficientes capazes de resolvê-los. Eles são chamados problemas $\mathbb{NP}$-completos. Curiosamente, **não foi provado** que tais algoritmos não existem!
- Esses problemas tem a característica notável de que se um deles admitir um algoritmo "eficiente" então todos admitem algoritmos "eficientes".
- Por que devo me preocupar com problemas $\mathbb{NP}$-difíceis Problemas dessa classe surgem em inúmeras situações práticas!

Dificuldade intrínseca de problemas

Exemplos:

- calcular as rotas dos caminhões de entrega de uma distribuidora de bebidas em São Paulo, minimizando a distância percorrida. (vehicle routing)
- calcular o número mínimo de *containers* para transportar um conjunto de caixas com produtos. (bin packing 3D)
- calcular a localização e o número mínimo de antenas de celulares para garantir a cobertura de uma certa região geográfica. (facility location)
- e muito mais...

É importante saber indentificar quando estamos lidando com um problema $\mathbb{NP}$-completo!

Algoritmos e tecnologia

- O mundo ideal: os computadores têm velocidade de processamento e memória infinita. Neste caso, qualquer algoritmo é igualmente bom e esta disciplina é inútil! Porém...
- O mundo real: computadores têm velocidade de processamento e memória limitadas.

Neste caso, faz **muita** diferença ter um **bom** algoritmo.

Algoritmos e tecnologia

Exemplo: **ordenação de um vetor de n elementos**

- Suponha que os computadores A e B executam $1G$ e $10M$ instruções por segundo, respectivamente. Ou seja, A é **100 vezes mais rápido** que B.
- Algoritmo 1: implementado em A por um excelente programador em linguagem de máquina (ultra-rápida). Executa $2n^2$ instruções.
- Algoritmo 2: implementado na máquina B por um programador mediano em linguagem de alto nível dispondo de um compilador "mais-ou-menos". Executa $50n \log n$ instruções.

Algoritmos e tecnologia

- O que acontece quando ordenamos um vetor de um milhão de elementos? **Qual algoritmo é mais rápido?**
- Algoritmo 1 na máquina A:
  $\frac{2.(10^6)^2 \text{ instruções}}{10^9 \text{ instruções/segundo}} \approx 2000$ segundos
- Algoritmo 2 na máquina B:
  $\frac{50.(10^6 \log 10^6) \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 100$ segundos
- Ou seja, B foi **VINTE VEZES** mais rápido do que A!
- Se o vetor tiver 10 milhões de elementos, esta razão será de 2.3 dias para 20 minutos!

Algoritmos e tecnologia – Conclusões

- O uso de um algoritmo adequado pode levar a ganhos extraordinários de desempenho.
- Isso pode ser tão importante quanto o projeto de *hardware*.
- A melhora obtida pode ser tão significativa que não poderia ser obtida simplesmente com o avanço da tecnologia.
- As melhorias nos algoritmos produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, etc).

Descrição de algoritmos

Podemos descrever um algoritmo de várias maneiras:

- usando uma linguagem de programação de alto nível: `C`, `Pascal`, `Java`, etc.
- implementando-o em linguagem de máquina diretamente executável em *hardware*
- em português
- em um pseudo-código de alto nível, como no livro [CLRS]

Usaremos, essencialmente, as duas últimas alternativas nesta disciplina.

Exemplo de pseudo-código

Algoritmo ORDENA-POR-INSERÇÃO: rearranja um vetor $A[1 \ldots n]$ de modo que fique crescente.

```
ORDENA-POR-INSERÇÃO(A, n)
1  para j ← 2 até n faça
2      chave ← A[j]
3      ▷ Insere A[j] no subvetor ordenado A[1 . . . j−1]
4      i ← j − 1
5      enquanto i ≥ 1  e A[i] > chave faça
6          A[i + 1] ← A[i]
7          i ← i − 1
8      A[i + 1] ← chave
```

Corretude de algoritmos

- Um algoritmo (que resolve um determinado problema) está correto se, para toda instância do problema, ele pára e devolve uma resposta correta.
- Algoritmos incorretos também têm sua utilidade, se soubermos prever a sua probabilidade de erro.
- Nesta disciplina, vamos trabalhar apenas com algoritmos corretos.

Complexidade de algoritmos

- Em geral, não basta saber que um dado algoritmo pára. Se ele for muuuuuuuito leeeeeeento, terá pouca utilidade.
- Queremos projetar/desenvolver algoritmos eficientes (rápidos).
- Mas o que seria uma boa medida de eficiência de um algoritmo?
- Não estamos interessados em quem programou, em que linguagem foi escrito e nem em qual a máquina executado!
- Queremos um critério uniforme para comparar algoritmos.

Modelo Computacional

- Uma possibilidade é definir um modelo computacional de um máquina.
- O modelo computacional estabelece quais os recursos disponíveis, as instruções básicas e quanto elas custam (= tempo).
- Dentro desse modelo, podemos estimar através de uma análise matemática o tempo que um algoritmo gasta em função do tamanho da entrada (= análise de complexidade).
- A análise de complexidade depende sempre do modelo computacional adotado.

Máquinas RAM

Salvo mencionado o contrário, usaremos o Modelo Abstrato RAM (Random Access Machine):

- simula máquinas convencionais (de verdade),
- possui um único processador que executa instruções seqüencialmente,
- tipos básicos são números inteiros e reais,
- há um limite no tamanho de cada *palavra de memória*: se a entrada tem "tamanho" n, então cada inteiro/real é representado por $c \log n$ bits onde $c \geq 1$ é uma contante.

Máquinas RAM

- executa operações aritméticas (soma, subtração, multiplicação, divisão, piso, teto), comparações, movimentação de dados de tipo básico e fluxo de controle (teste *if/else*, chamada e retorno de rotinas) em **tempo constante**,
- Certas operações caem ficam em uma **zona cinza**, (por exemplo, exponenciação).
- Veja mais detalhes do modelo RAM no [CLRS].

Tamanho da entrada

Problema: Primalidade

Entrada: inteiro n

Tamanho: (número de bits de n) $\approx \lg n = \log_2 n$

Problema: Ordenação

Entrada: vetor $A[1 \ldots n]$

Tamanho: $n \lg U$ onde U é o maior número em $A[1 \ldots n]$

Medida de complexidade e eficiência de algoritmos

- A complexidade de tempo (= eficiência) de um algoritmo é o número de instruções básicas que ele executa em função do tamanho da entrada.
- Adota-se uma "atitude pessimista" e faz-se uma análise de pior caso.
  Determina-se o tempo máximo necessário para resolver uma instância de um certo tamanho.
- Além disso, a análise concentra-se no comportamento do algoritmo para entradas de tamanho GRANDE = análise assintótica.

Medida de complexidade e eficiência de algoritmos

- Um algoritmo é chamado eficiente se a função que mede sua complexidade de tempo é limitada por um **polinômio** no tamanho da entrada.
  Por exemplo: n, $3n-7$, $4n^2$, $143n^2-4n+2$, n^5.
- Mas por que polinômios?
  Polinômios são funções bem "comportadas".

Vantagens do método de análise proposto

- O modelo RAM é robusto e permite prever o comportamento de um algoritmo para instâncias GRANDES.
- O modelo permite comparar algoritmos que resolvem um mesmo problema.
- A análise é mais robusta em relação às evoluções tecnológicas.

Desvantagens do método de análise proposto

- Fornece um limite de complexidade pessimista sempre considerando o pior caso.
- Em uma aplicação real, nem todas as instâncias ocorrem com a mesma freqüência e é possível que as "instâncias ruins" ocorram raramente.
- Não fornece nenhuma informação sobre o comportamento do algoritmo no caso médio.
- A análise de complexidade de algoritmos no caso médio é bastante difícil, principalmente, porque muitas vezes não é claro o que é o "caso médio".

Começando a trabalhar

Complexidade assintótica de algoritmos

- Como dito anteriormente, na maior parte desta disciplina, estaremos nos concentrando na **análise de pior caso** e no **comportamento assintótico** dos algoritmos (instâncias de tamanho grande).
- Considere um algoritmo que tem como complexidade (de pior caso) uma função quadrática $an^2 + bn + c$, onde a, b, c são constantes absolutas que dependem apenas dos custos c_i.
- O estudo assintótico nos permite "jogar para debaixo do tapete" os valores destas constantes, i.e., aquilo que independe do tamanho da entrada (neste caso os valores de a, b e c).
- **Por que podemos fazer isso ?**

Análise assintótica de funções quadráticas

Considere a função quadrática $3n^2 + 10n + 50$:

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

Como se vê, $3n^2$ é o termo dominante quando n é grande.

De um modo geral, podemos nos concentrar nos termos dominantes e esquecer os demais.

Notação assintótica

- Usando notação assintótica, dizemos que o algoritmo Ordena-Por-Inserção tem complexidade de tempo de pior caso $\Theta(n^2)$.
- Isto quer dizer duas coisas:
  - a complexidade de tempo é limitada (superiormente) assintoticamente por algum polinômio da forma an^2 para alguma constante a,
  - para todo n suficientemente grande, existe alguma instância de tamanho n que consome tempo pelo menos dn^2, para alguma contante positiva d.
- Mais adiante discutiremos em detalhes o uso da notação assintótica em análise de algoritmos.

Algoritmos recursivos

- O que é o paradigma de **divisão-e-conquista**?
- Como mostrar a corretude de um algoritmo recursivo?
- Como analisar o consumo de tempo de um algoritmo recursivo?
- O que é uma **fórmula de recorrência**?
- O que significa *resolver* uma fórmula de recorrência?

Recursão e o paradigma de divisão-e-conquista

- Um algoritmo recursivo encontra a saída para uma instância de entrada de um problema chamando a si mesmo para resolver instâncias menores deste mesmo problema.
- Algoritmos de divisão-e-conquista possuem três etapas em cada nível de recursão:
  1. **Divisão:** o problema é dividido em subproblemas semelhantes ao problema original, porém tendo como entrada instâncias de tamanho menor.
  2. **Conquista:** cada subproblema é resolvido recursivamente a menos que o tamanho de sua entrada seja suficientemente "pequeno", quando este é resolvido diretamente.
  3. **Combinação:** as soluções dos subproblemas são combinadas para obter uma solução do problema original.

Exemplo de divisão-e-conquista: *Mergesort*

- Mergesort é um algoritmo para resolver o problema de ordenação e um exemplo clássico do uso do paradigma de divisão-e-conquista. (*to merge* = intercalar)
- Descrição do Mergesort em alto nível:
  1. **Divisão**: divida o vetor com n elementos em dois subvetores de tamanho $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$, respectivamente.
  2. **Conquista**: ordene os dois vetores recursivamente usando o Mergesort;
  3. **Combinação**: intercale os dois subvetores para obter um vetor ordenado usando o algoritmo Intercala.

Mergesort

Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

	p				q				r
A	66	33	55	44	99	11	77	22	88

Mergesort

Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

	p				q				r
A	33	44	55	66	99	11	77	22	88

Mergesort

Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

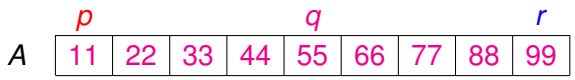
```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

	p				q				r
A	33	44	55	66	99	11	22	77	88

Mergesort

Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```



Corretude do Mergesort

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

O algoritmo está correto?

A corretude do algoritmo Mergesort apoia-se na corretude do algoritmo Intercala e pode ser demonstrada **por indução** em $n := r - p + 1$.

Aprendemos como fazer provas por indução em MC358.

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

Qual é a complexidade de MERGESORT?

Seja $T(n) :=$ o consumo de tempo máximo (pior caso) em função de $n = r - p + 1$

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

linha	consumo de tempo
1	?
2	?
3	?
4	?
5	?
	$T(n) =?$

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

linha	consumo de tempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$T(\lceil n/2 \rceil)$
4	$T(\lfloor n/2 \rfloor)$
5	$\Theta(n)$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) + \Theta(2)$$

Complexidade do Mergesort

- Obtemos o que chamamos de fórmula de recorrência (i.e., uma fórmula definida em termos de si mesma).

$$T(1) = \Theta(1)$$
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \quad \text{para } n = 2, 3, 4, \ldots$$

- Em geral, ao aplicar o paradigma de divisão-e-conquista, chega-se a um algoritmo recursivo cuja complexidade $T(n)$ é uma fórmula de recorrência.
- É necessário então resolver a recorrência! Mas, o que significa resolver uma recorrência?
- Significa encontrar uma "fórmula fechada" para $T(n)$.
- No caso, $T(n) = \Theta(n \lg n)$. Assim, o consumo de tempo do Mergesort é $\Theta(n \lg n)$ no pior caso.
- Veremos mais tarde como resolver recorrências.

Crescimento de funções

Notação Assintótica

- Vamos expressar complexidade através de funções em variáveis que descrevam o tamanho de instâncias do problema. Exemplos:
  - Problemas de aritmética de precisão arbitrária: número de bits (ou bytes) dos inteiros.
  - Problemas em grafos: número de vértices e/ou arestas
  - Problemas de ordenação de vetores: tamanho do vetor.
  - Busca em textos: número de caracteres do texto ou padrão de busca.
- Vamos supor que funções que expressam complexidade são sempre positivas, já que estamos medindo número de operações.

Comparação de Funções

- Vamos comparar funções assintoticamente, ou seja, para valores grandes, desprezando constantes multiplicativas e termos de menor ordem.

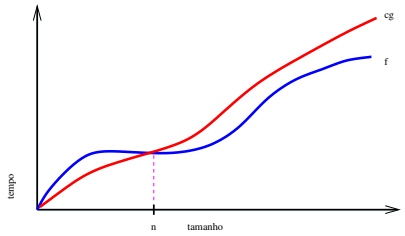
	$n = 100$	$n = 1000$	$n = 10^4$	$n = 10^6$	$n = 10^9$
$\log n$	2	3	4	6	9
n	100	1000	10^4	10^6	10^9
$n \log n$	200	3000	$4 \cdot 10^4$	$6 \cdot 10^6$	$9 \cdot 10^9$
n^2	10^4	10^6	10^8	10^{12}	10^{18}
$100n^2 + 15n$	$1,0015 \cdot 10^6$	$1,00015 \cdot 10^8$	$\approx 10^{10}$	$\approx 10^{14}$	$\approx 10^{20}$
2^n	$\approx 1,26 \cdot 10^{30}$	$\approx 1,07 \cdot 10^{301}$	?	?	?

Classe O

Definição:

$O(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in O(g(n))$, então $f(n)$ cresce no máximo tão rapidamente quanto $g(n)$.



Classe O

Definição:

$O(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in O(g(n))$, então $f(n)$ cresce no máximo tão rapidamente quanto $g(n)$.

Exemplo:

$\frac{1}{2}n^2 - 3n \in O(n^2)$
Valores de c e n_0 que satisfazem a definição são

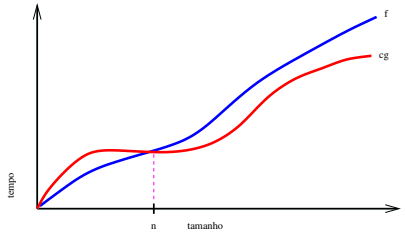
$$c = \frac{1}{2} \text{ e } n_0 = 7.$$

Classe Ω

Definição:

$\Omega(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq cg(n) \leq f(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in \Omega(g(n))$, então $f(n)$ cresce no mínimo tão rapidamente quanto $g(n)$.



Classe Ω

Definição:

$\Omega(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq cg(n) \leq f(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in \Omega(g(n))$, então $f(n)$ cresce no mínimo tão rapidamente quanto $g(n)$.

Exemplo:

$\frac{1}{2}n^2 - 3n \in \Omega(n^2)$
Valores de c e n_0 que satisfazem a definição são

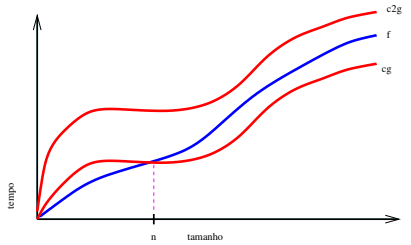
$$c = \frac{1}{14} \text{ e } n_0 = 7.$$

Classe Θ

Definição:

$\Theta(g(n)) = \{f(n) :$ existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in \Theta(g(n))$, então $f(n)$ cresce tão rapidamente quanto $g(n)$.



Classe Θ

Definição:

$\Theta(g(n)) = \{f(n) :$ existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in \Theta(g(n))$, então $f(n)$ cresce tão rapidamente quanto $g(n)$.

Exemplo:

$\frac{1}{2}n^2 - 3n \in \Theta(n^2)$
Valores de c_1, c_2 e n_0 que satisfazem a definição são

$$c_1 = \frac{1}{14}, c_2 = \frac{1}{2} \text{ e } n_0 = 7.$$

Classe o

Definição:

$o(g(n)) = \{f(n) :$ para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in o(g(n))$, então $f(n)$ cresce mais lentamente que $g(n)$.

Exemplo:

$1000n^2 \in o(n^3)$
Para todo valor de c, um n_0 que satisfaz a definição é

$$n_0 = \left\lceil \frac{1000}{c} \right\rceil + 1.$$

Classe ω

Definição:

$\omega(g(n)) = \{f(n) :$ para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq cg(n) < f(n)$, para todo $n \geq n_0.\}$

Informalmente, dizemos que, se $f(n) \in \omega(g(n))$, então $f(n)$ cresce mais rapidamente que $g(n)$.

Exemplo:

$\frac{1}{1000}n^2 \in \omega(n)$
Para todo valor de c, um n_0 que satisfaz a definição é

$$n_0 = \lceil 1000c \rceil + 1.$$

Verificação alternativa

$f(n) \in o(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$
$f(n) \in O(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$
$f(n) \in \Theta(g(n))$ se $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$
$f(n) \in \Omega(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0.$
$f(n) \in \omega(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$

Mas há ainda funções incomparáveis!

Propriedades das Classes

Transitividade:
Se $f(n) \in O(g(n))$ e $g(n) \in O(h(n))$, então $f(n) \in O(h(n))$.
Se $f(n) \in \Omega(g(n))$ e $g(n) \in \Omega(h(n))$, então $f(n) \in \Omega(h(n))$.
Se $f(n) \in \Theta(g(n))$ e $g(n) \in \Theta(h(n))$, então $f(n) \in \Theta(h(n))$.
Se $f(n) \in o(g(n))$ e $g(n) \in o(h(n))$, então $f(n) \in o(h(n))$.
Se $f(n) \in \omega(g(n))$ e $g(n) \in \omega(h(n))$, então $f(n) \in \omega(h(n))$.

Propriedades das Classes

Reflexividade:
$f(n) \in O(f(n))$.
$f(n) \in \Omega(f(n))$.
$f(n) \in \Theta(f(n))$.
Simetria:
$f(n) \in \Theta(g(n))$ se, e somente se, $g(n) \in \Theta(f(n))$.
Simetria Transposta:
$f(n) \in O(g(n))$ se, e somente se, $g(n) \in \Omega(f(n))$.
$f(n) \in o(g(n))$ se, e somente se, $g(n) \in \omega(f(n))$.

Exemplos

Quais as relações de comparação assintótica das funções:

- 2^{π}
- $\log n$
- n
- $n \log n$
- n^2
- $100n^2 + 15n$
- 2^n

Recorrências

Resolução de Recorrências

- Relações de recorrência expressam a complexidade de algoritmos recursivos como, por exemplo, os algoritmos de divisão e conquista.
- É preciso saber resolver as recorrências para que possamos efetivamente determinar a complexidade dos algoritmos recursivos.

Mergesort

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

Qual é a complexidade de MERGESORT?

Seja $T(n) :=$ o consumo de tempo máximo (pior caso) em função de $n = r - p + 1$

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

linha	consumo de tempo
1	?
2	?
3	?
4	?
5	?

$T(n) = ?$

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1  se p < r
2     então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)
```

linha	consumo de tempo
1	b_0
2	b_1
3	$T(\lceil n/2 \rceil)$
4	$T(\lfloor n/2 \rfloor)$
5	an

$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + an + (b_0 + b_1)$

Resolução de recorrências

- Queremos resolver a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + an + b \quad \text{para } n \geq 2. \end{aligned}$$

- Resolver uma recorrência significa encontrar uma fórmula fechada para $T(n)$.
- Não é necessário achar uma solução exata. Basta encontrar uma função $f(n)$ tal que $T(n) \in \Theta(f(n))$.

Resolução de recorrências

Alguns métodos para resolução de recorrências:

- substituição
- iteração
- árvore de recorrência

Veremos também um resultado bem geral que permite resolver várias recorrências: Master theorem.

Método da substituição

- Idéia básica: "adivinhe" qual é a solução e prove por indução que ela funciona!
- Método poderoso mas nem sempre aplicável (obviamente).
- Com prática e experiência fica mais fácil de usar!

Exemplo

Considere a recorrência:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \quad \text{para } n \geq 2. \end{aligned}$$

Chuto que $T(n) \in O(n \lg n)$.

Mais precisamente, chuto que $T(n) \leq 3n \lg n$.

(Lembre que $\lg n = \log_2 n$.)

Exemplo

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \\ &\leq 3 \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + 3 \left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq 3 \left\lceil \frac{n}{2} \right\rceil \lg n + 3 \left\lfloor \frac{n}{2} \right\rfloor (\lg n - 1) + n \\ &= 3 \left(\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor \right) \lg n - 3 \left\lfloor \frac{n}{2} \right\rfloor + n \\ &= 3n \lg n - 3 \left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq 3n \lg n. \end{aligned}$$

(Yeeeeeeesssssss!)

Exemplo

- Mas espere um pouco!
- $T(1) = 1$ e $3.1. \lg 1 = 0$ e a base da indução não funciona!
- Certo, mas lembre-se da definição da classe $O(\ )$.

  Só preciso provar que $T(n) \leq 3n \lg n$ para $n \geq n_0$ onde n_0 é alguma constante.

  Vamos tentar com $n_0 = 2$. Nesse caso

  $$T(2) = T(1) + T(1) + 2 = 4 \leq 3.2. \lg 2 = 6,$$

  e estamos feitos.

Exemplo

- Certo, funcionou para $T(1) = 1$.
- Mas e se por exemplo $T(1) = 8$?

  Então $T(2) = 8 + 8 + 2 = 18$ e $3.2.\lg 2 = 6$.
  Não deu certo...
- Certo, mas aí basta escolher uma constante maior.
  Mostra-se do mesmo jeito que $T(n) \leq 10n \lg n$ e para esta escolha $T(2) = 18 \leq 10.2.\lg 2 = 20$.
- De modo geral, se o passo de indução funciona ($T(n) \leq cn \lg n$), é possível escolher c e a base da indução (n_0) de modo conveniente!

Como achar as constantes?

- Tudo bem. Dá até para chutar que $T(n)$ pertence a classe $O(n \lg n)$.
- Mas como descobrir que $T(n) \leq 3n \lg n$? Como achar a constante 3?
- Eis um método simples: suponha como hipótese de indução que $T(n) \leq cn \lg n$ para $n \geq n_0$ onde c e n_0 são constantes que vou tentar determinar.

Primeira tentativa

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \\ &\leq c \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + c \left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq c \left\lceil \frac{n}{2} \right\rceil \lg n + c \left\lfloor \frac{n}{2} \right\rfloor \lg n + n \\ &= c \left(\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor \right) \lg n + n \\ &= cn \lg n + n \end{aligned}$$

(Hummm, não deu certo...)

Segunda tentativa

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \\ &\leq c \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + c \left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq c \left\lceil \frac{n}{2} \right\rceil \lg n + c \left\lfloor \frac{n}{2} \right\rfloor (\lg n - 1) + n \\ &= c \left(\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor \right) \lg n - c \left\lfloor \frac{n}{2} \right\rfloor + n \\ &= cn \lg n - c \left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq cn \lg n. \end{aligned}$$

Para garantir a última desigualdade basta que $-c\lceil n/2 \rceil + n \leq 0$ e $c = 3$ funciona. (Yeeeeeeesssssss!)

Completando o exemplo

Mostramos que a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \quad \text{para } n \geq 2. \end{aligned}$$

satisfaz $T(n) \in O(n \lg n)$.

Mas quem garante que $T(n)$ não é "menor"?

O melhor é mostrar que $T(n) \in \Theta(n \lg n)$.

Resta então mostrar que $T(n) \in \Omega(n \lg n)$. A prova é similar. (Exercício!)

Como chutar?

Não há nenhuma receita genérica para adivinhar soluções de recorrências. A experiência é o fator mais importante.

Felizmente, há várias idéias que podem ajudar.

Considere a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(\lfloor n/2 \rfloor) + n \quad \text{para } n \geq 2. \end{aligned}$$

Ela é quase idêntica à anterior e podemos chutar que $T(n) \in \Theta(n \lg n)$. Isto de fato é verdade. (Exercício ou consulte o CLRS)

Como chutar?

Considere agora a recorrência

$$\begin{aligned} T(n) &= c \qquad\qquad\qquad\quad \text{para } n \leq 34. \\ T(n) &= 2T(\lfloor n/2 \rfloor + 17) + n \quad \text{para } n \geq 35. \end{aligned}$$

Ela parece bem mais difícil por causa do "17" no lado direito.

Intuitivamente, porém, isto não deveria afetar a solução. Para n **grande** a diferença entre $T(\lfloor n/2 \rfloor)$ e $T(\lfloor n/2 \rfloor + 17)$ não é tanta.

Chuto então que $T(n) \in \Theta(n \lg n)$. (Exercício!)

Truques e sutilezas

Algumas vezes adivinhamos corretamente a solução de uma recorrência, mas as contas aparentemente não funcionam! Em geral, o que é necessário é fortalecer a hipótese de indução.

Considere a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \quad \text{para } n \geq 2. \end{aligned}$$

Chutamos que $T(n) \in O(n)$ e tentamos mostrar que $T(n) \leq cn$ para alguma constante c.

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \\ &\leq c\lceil n/2 \rceil + c\lfloor n/2 \rfloor + 1 \\ &= cn + 1. \end{aligned}$$

(Humm, falhou...)

E agora? Será que erramos o chute? Será que $T(n) \in \Theta(n^2)$?

Truques e sutilezas

Na verdade, adivinhamos corretamente. Para provar isso, é preciso usar uma hipótese de indução mais forte.

Vamos mostrar que $T(n) \leq cn - b$ onde $b > 0$ é uma constante.

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \\ &\leq c\lceil n/2 \rceil - b + c\lfloor n/2 \rfloor - b + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b \end{aligned}$$

onde a última desigualdade vale se $b \geq 1$.
(Yeeeessss!)

Método da iteração

- Não é necessário adivinhar a resposta!
- Precisa fazer mais contas!
- Idéia: expandir (iterar) a recorrência e escrevê-la como uma somatória de termos que dependem apenas de n e das condições iniciais.
- Precisa conhecer limitantes para várias somatórias.

Método da iteração

Considere a recorrência

$$\begin{aligned} T(n) &= b && \text{para } n \leq 3, \\ T(n) &= 3T(\lfloor n/4 \rfloor) + n && \text{para } n \geq 4. \end{aligned}$$

Iterando a recorrência obtemos

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) \\ &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor). \end{aligned}$$

Certo, mas quando devo parar?
O i-ésimo termo da série é $3^i\lfloor n/4^i \rfloor$. Ela termina quando $\lfloor n/4^i \rfloor \leq 3$, ou seja, $i \geq \log_4 n$.

Método da iteração

Como $\lfloor n/4^i \rfloor \leq n/4^i$ temos que

$$\begin{aligned} T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \cdots + 3^i b \\ T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \cdots + d.3^{\log_4 n} \\ &\leq n.(1 + 3/4 + 9/16 + 27/64 + \cdots) + dn^{\log_4 3} \\ &= n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + dn^{\log_4 3} \\ &= 4n + dn^{\log_4 3} \end{aligned}$$

pois $3^{\log_4 n} = n^{\log_4 3}$ e $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$ para $0 < q < 1$.

Como $\log_4 3 < 1$ segue que $n^{\log_4 3} \in o(n)$ e logo, $T(n) \in O(n)$.

Método de iteração

- As contas ficam mais simples se supormos que a recorrência está definida apenas para potências de um número, por exemplo, $n = 4^i$.
- Note, entretanto, que a recorrência deve ser provada para todo natural suficientemente grande.
- Muitas vezes, é possível depois de iterar a recorrência, adivinhar a solução e usar o método da substituição!

Método de iteração

$$\begin{aligned} T(n) &= b && \text{para } n \leq 3, \\ T(n) &= 3T(\lfloor n/4 \rfloor) + n && \text{para } n \geq 4. \end{aligned}$$

Chuto que $T(n) \leq cn$.

$$\begin{aligned} T(n) &= 3T(\lfloor n/4 \rfloor) + n \\ &\leq 3c\lfloor n/4 \rfloor + n \\ &\leq 3c(n/4) + n \\ &\leq cn \end{aligned}$$

onde a última desigualdade vale se $c \geq 4$.
(Yeeessss!)

Árvore de recorrência

- Permite visualizar melhor o que acontece quando a recorrência é iterada.
- É mais fácil organizar as contas.
- Útil para recorrências de algoritmos de divisão-e-conquista.

Árvore de recorrência

Considere a recorrência

$$\begin{aligned} T(n) &= \Theta(1) && \text{para } n = 1, 2, 3, \\ T(n) &= 3T(\lfloor n/4 \rfloor) + cn^2 && \text{para } n \geq 4, \end{aligned}$$

onde $c > 0$ é uma constante.

Costuma-se (CLRS) usar a notação $T(n) = \Theta(1)$ para indicar que $T(n)$ é uma constante.

Árvore de recorrência

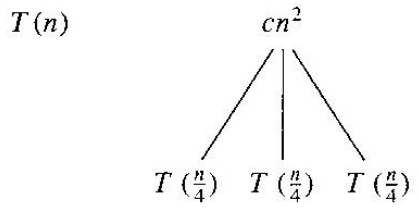
Simplificação

Vamos supor que a recorrência está definida apenas para potências de 4

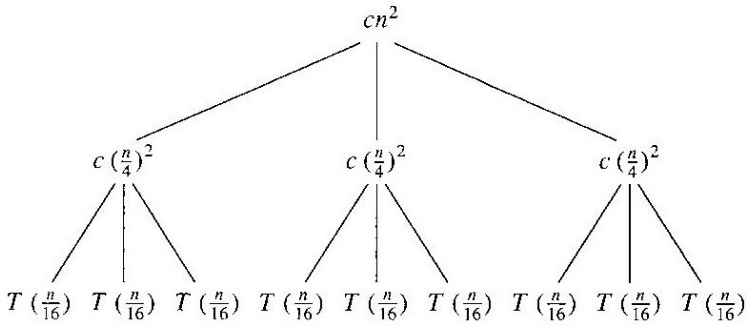
$$\begin{aligned} T(n) &= \Theta(1) && \text{para } n = 1, \\ T(n) &= 3T(n/4) + cn^2 && \text{para } n = 4, 16, \ldots, 4^i, \ldots. \end{aligned}$$

Isto permite descobrir mais facilmente a solução. Depois usamos o método da substituição para formalizar.

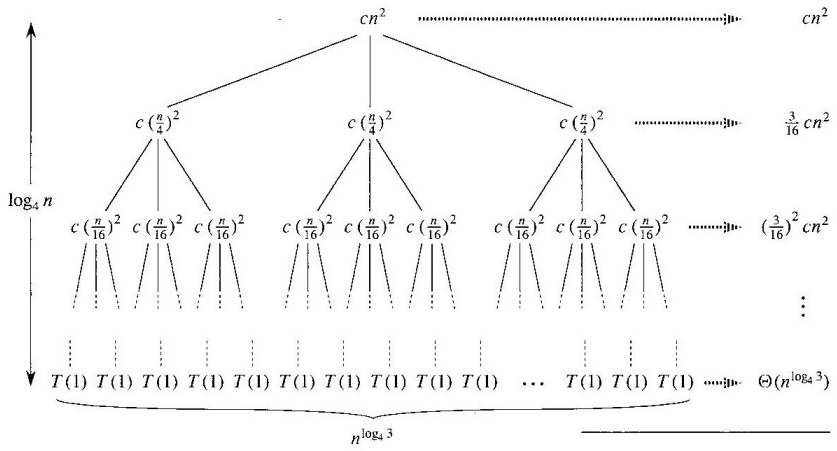
Árvore de recorrência



Árvore de recorrência



Árvore de recorrência



Total: $O(n^2)$

Árvore de recorrência

- O número de níveis é $\log_4 n + 1$.
- No nível i o tempo gasto (sem contar as chamadas recursivas) é $(3/16)^i cn^2$.
- No último nível há $3^{\log_4 n} = n^{\log_4 3}$ folhas. Como $T(1) = \Theta(1)$ o tempo gasto é $\Theta(n^{\log_4 3})$.

Árvore de recorrência

Logo,

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \cdots + \\ &\quad + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &\leq \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13}cn^2 + \Theta(n^{\log_4 3}), \end{aligned}$$

e $T(n) \in O(n^2)$.

Árvore de recorrência

Mas $T(n) \in O(n^2)$ é realmente a solução da recorrência original?

Com base na árvore de recorrência, chutamos que $T(n) \leq dn^2$ para alguma constante $d > 0$.

$$\begin{aligned} T(n) &= 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2 \end{aligned}$$

onde a última desigualdade vale se $d \geq (16/13)c$.
(Yeeesssss!)

Árvore de recorrência

Resumo

- O número de nós em cada nível da árvore é o número de chamadas recursivas.
- Em cada nó indicamos o "tempo" ou "trabalho" gasto naquele nó que **não** corresponde a chamadas recursivas.
- Na coluna mais à direita indicamos o tempo total naquele nível que **não** corresponde a chamadas recursivas.
- Somando ao longo da coluna determina-se a solução da recorrência.

Recorrências com O à direita (CLRS)

Uma "recorrência"

$$T(n) = \Theta(1) \qquad \text{para } n = 1, 2,$$
$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \qquad \text{para } n \geq 3$$

representa todas as recorrências da forma

$$T(n) = a \qquad \text{para } n = 1, 2,$$
$$T(n) = 3T(\lfloor n/4 \rfloor) + bn^2 \qquad \text{para } n \geq 3$$

onde a e $b > 0$ são constantes.

As soluções exatas dependem dos valores de a e b, mas estão todas na mesma classe Θ.

A "solução" é $T(n) = \Theta(n^2)$, ou seja, $T(n) \in \Theta(n^2)$.

As mesmas observações valem para as classes O, Ω, o, ω.

Recorrência do Mergesort

Podemos escrever a recorrência de tempo do Mergesort da seguinte forma

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \qquad \text{para } n \geq 2. \end{aligned}$$

A solução da recorrência é $T(n) = \Theta(n \lg n)$.

A prova é essencialmente a mesma do primeiro exemplo. (Exercício!)

Cuidados com a notação assintótica

A notação assintótica é muito versátil e expressiva. Entretanto, deve-se tomar alguns cuidados.

Considere a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(\lfloor n/2 \rfloor) + n \qquad \text{para } n \geq 2. \end{aligned}$$

É similar a recorrência do Mergesort!

Mas eu vou "provar" que $T(n) = O(n)$.

Cuidados com a notação assintótica

Vou mostrar que $T(n) \leq cn$ para alguma constante $c > 0$.

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2c\lfloor n/2 \rfloor + n \\ &\leq cn + n \\ &= O(n) \qquad \Longleftarrow \textbf{ERRADO!!!} \end{aligned}$$

Por quê?

Não foi feito o passo indutivo, ou seja, não foi mostrado que $T(n) \leq cn$.

Teorema Master

- Veremos agora um resultado que descreve soluções para recorrências da forma

$$T(n) = aT(n/b) + f(n),$$

  onde $a \geq 1$ e $b > 1$ são constantes.
- O caso base é omitido na definição e convenciona-se que é uma constante para valores pequenos.
- A expressão n/b pode indicar tanto $\lfloor n/b \rfloor$ quanto $\lceil n/b \rceil$.
- O Teorema Master **não** fornece a resposta para todas as recorrências da forma acima.

Teorema Master

Teorema (Teorema Master (CLRS))

Sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida para os inteiros não-negativos pela relação de recorrência

$$T(n) = aT(n/b) + f(n).$$

Então $T(n)$ pode ser limitada assintoticamente da seguinte maneira:

1. *Se $f(n) \in O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $T(n) \in \Theta(n^{\log_b a})$*
2. *Se $f(n) \in \Theta(n^{\log_b a})$, então $T(n) \in \Theta(n^{\log_b a} \log n)$*
3. *Se $f(n) \in \Omega(n^{\log_b a + \epsilon})$, para alguma constante $\epsilon > 0$ e se $af(n/b) \leq cf(n)$, para alguma constante $c < 1$ e para n suficientemente grande, então $T(n) \in \Theta(f(n))$*

Exemplos de Recorrências

Exemplos onde o Teorema Master se aplica:

- Caso 1:
  $T(n) = 9T(n/3) + n$
  $T(n) = 4T(n/2) + n \log n$
- Caso 2:
  $T(n) = T(2n/3) + 1$
  $T(n) = 2T(n/2) + (n + \log n)$
- Caso 3:
  $T(n) = T(3n/4) + n \log n$

Exemplos de Recorrências

Exemplos onde o Teorema Master **não se aplica**:

- $T(n) = T(n-1) + n$
- $T(n) = T(n-a) + T(a) + n$, ($a \geq 1$ inteiro)
- $T(n) = T(\alpha n) + T((1-\alpha)n) + n$, ($0 < \alpha < 1$)
- $T(n) = T(n-1) + \log n$
- $T(n) = 2T(\frac{n}{2}) + n \log n$

Demonstração Direta

A *demonstração direta* de uma implicação $p \Rightarrow q$ é uma sequência de passos lógicos (implicações):

$$p \Rightarrow p_1 \Rightarrow p_2 \Rightarrow \cdots \Rightarrow p_n \Rightarrow q,$$

que resultam, por transitividade, na implicação desejada.
Cada passo da demonstração é um axioma ou um teorema demonstrado previamente.

Exemplo:

Provar que $\sum_{i=1}^{k} 2i - 1 = k^2$.

Demonstração pela Contrapositiva

A *contrapositiva* de $p \Rightarrow q$ é $\neg q \Rightarrow \neg p$.
A contrapositiva é equivalente à implicação original. A veracidade de $\neg q \Rightarrow \neg p$ implica a veracidade de $p \Rightarrow q$, e vice-versa.
A técnica é útil quando é mais fácil demonstrar a contrapositiva que a implicação original.
Para demonstrarmos a contrapositiva de uma implicação, podemos utilizar qualquer técnica de demonstração.

Exemplo:

Provar que se $2 \mid 3m$, então $2 \mid m$.

Demonstração por Contradição

A *Demonstração por contradição* envolve supor absurdamente que a afirmação a ser demonstrada é falsa e obter, através de implicações válidas, uma conclusão contraditória.
A contradição obtida implica que a hipótese absurda é falsa e, portanto, a afirmação é de fato verdadeira.
No caso de uma implicação $p \Rightarrow q$, equivalente a $\neg p \vee q$, a negação é $p \wedge \neg q$.

Exemplo:

Dados os inteiros positivos n e $n+1$, provar que o maior inteiro que divide ambos n e $n+1$ é 1.

Demonstração por Casos

Na *Demonstração por Casos*, particionamos o universo de possibilidades em um conjunto finito de casos e demonstramos a veracidade da implicação para cada caso.
Para demonstrar cada caso individual, qualquer técnica de demonstração pode ser utilizada.

Exemplo:

Provar que a soma de dois inteiros x e y de mesma paridade é sempre par.

Demonstração por Indução

Na *Demonstração por Indução*, queremos demonstrar a validade de $P(n)$, uma propriedade P com um parâmetro natural n associado, para todo valor de n.
Há um número infinito de casos a serem considerados, um para cada valor de n. Demonstramos os infinitos casos de uma só vez:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução:** Supomos que $P(n)$ é verdadeiro.
- **Passo de Indução:** Provamos que $P(n+1)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Provar que $\sum_{i=1}^{k} 2i - 1 = k^2$, agora por indução.

Indução Fraca × Indução Forte

A *indução forte* difere da *indução fraca* (ou *simples*) apenas na suposição da hipótese.
No caso da indução forte, devemos supor que a propriedade vale para todos os casos anteriores, não somente para o anterior, ou seja:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução Forte:** Supomos que $P(k)$ é verdadeiro, para todo $k \leq n$.
- **Passo de Indução:** Provamos que $P(n+1)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Prove que todo inteiro n pode ser escrito como a soma de diferentes potências de 2.

Indução matemática

Demonstração por Indução

Na *Demonstração por Indução*, queremos demonstrar a validade de $P(n)$, uma propriedade P com um parâmetro natural n associado, para todo valor de n.
Há um número infinito de casos a serem considerados, um para cada valor de n. Demonstramos os infinitos casos de uma só vez:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução:** Supomos que $P(n)$ é verdadeiro.
- **Passo de Indução:** Provamos que $P(n+1)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Prove que a soma dos n primeiros naturais ímpares é n^2.

Demonstração por Indução

Outra forma equivalente:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução:** Supomos que $P(n-1)$ é verdadeiro.
- **Passo de Indução:** Provamos que $P(n)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Prove que a soma dos n primeiros naturais ímpares é n^2.

Demonstração por Indução

Às vezes queremos provar que uma proposição $P(n)$ vale para $n \geq n_0$ para algum n_0.

- **Base da Indução:** Demonstramos $P(n_0)$.
- **Hipótese de Indução:** Supomos que $P(n-1)$ é verdadeiro.
- **Passo de Indução:** Provamos que $P(n)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Prove que todo inteiro $n \geq 2$ pode ser fatorado como um produto de primos.

Indução Fraca $\times$ Indução Forte

A *indução forte* difere da *indução fraca* (ou *simples*) apenas na suposição da hipótese.
No caso da indução forte, devemos supor que a propriedade vale para todos os casos anteriores, não somente para o anterior, ou seja:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução Forte:** Supomos que $P(k)$ é verdadeiro, para todo $1 \leq k < n$.
- **Passo de Indução:** Provamos que $P(n)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Prove que todo inteiro $n \geq 2$ pode ser fatorado como um produto de primos.

Exemplo 1

Demonstre que, para naturais $x \geq 1$ e n, $x^n - 1$ é divisível por $x - 1$.

Demonstração:

- A **base da indução** é, naturalmente, o caso $n = 1$. Temos que $x^n - 1 = x - 1$, que é obviamente divisível por $x - 1$. Isso encerra a demonstração da base da indução.

Exemplo 1 (cont.)

- A **hipótese de indução** é: *Suponha que $x^n - 1$ seja divisível por $x - 1$ para todo natural x.*
- O **passo de indução** é: *Supondo a h.i., vamos mostrar $x^{n+1} - 1$ é divisível por $x - 1$, para todo natural x.* Primeiro reescrevemos $x^{n+1} - 1$ como

$$x^{n+1} - 1 = x(x^n - 1) + (x - 1).$$

Pela h.i., $x^n - 1$ é divisível por $x - 1$. Portanto, o lado direito da equação acima é, de fato, divisível por $x - 1$.

A demonstração por indução está completa. ■

Exemplo 2

Demonstre que a equação

$$\sum_{i=1}^{n}(3 + 5i) = 2.5n^2 + 5.5n$$

vale para todo inteiro $n \geq 1$.

Demonstração:

- A **base da indução** é, naturalmente, o caso $n = 1$. Temos

$$\sum_{i=1}^{1}(3 + 5i) = 8 = 2.5 \times 1^2 + 5.5 \times 1.$$

Portanto, a somatória tem o valor previsto pela fórmula fechada, demonstrando que a equação vale para $n = 1$.

Exemplo 2 (cont.)

- A **hipótese de indução** é: *Suponha que a equação vale para n.*
- O **passo de indução é:** *Supondo a h.i., vamos mostrar que a equação vale para o valor $n + 1$.* O caminho é simples:

$$\begin{aligned}
\sum_{i=1}^{n+1}(3 + 5i) &= \sum_{i=1}^{n}(3 + 5i) + (3 + 5(n + 1)) \\
&= 2.5n^2 + 5.5n + (3 + 5(n + 1)) \text{ (pela h.i.)} \\
&= 2.5n^2 + 5.5n + 5n + 8 \\
&= 2.5n^2 + 5n + 2.5 + 5.5n + 5.5 \\
&= 2.5(n + 1)^2 + 5.5(n + 1).
\end{aligned}$$

A última linha da dedução mostra que a fórmula vale para $n + 1$. A demonstração por indução está completa. ■

Exemplo 3

Demonstre que a inequação

$$(1 + x)^n \geq 1 + nx$$

vale para todo natural n e real x tal que $(1 + x) > 0$.

Demonstração:

- A **base da indução** é, novamente, $n = 1$. Nesse caso ambos os lados da inequação são iguais a $1 + x$, mostrando a sua validade. Isto encerra a prova do caso base.

Exemplo 3 (cont.)

- A **hipótese de indução** é: *Suponha que a inequação vale para n, isto é,* $(1+x)^n \geq 1+nx$ *para todo real* x *tal que* $(1+x) > 0$.
- O **passo de indução é:** *Supondo a h.i., vamos mostrar que a inequação vale para o valor* $n+1$*, isto é,* $(1+x)^{n+1} \geq 1+(n+1)x$ *para todo* x *tal que* $(1+x) > 0$. Novamente, a dedução é simples:

$$\begin{aligned}(1+x)^{n+1} &= (1+x)^n(1+x)\\ &\geq (1+nx)(1+x) \text{ (pela h.i. e } (1+x)>0)\\ &= 1+(n+1)x+nx^2\\ &\geq 1+(n+1)x \text{ (já que } nx^2 \geq 0)\end{aligned}$$

A última linha mostra que a inequação vale para $n+1$, completando a demonstração. ■

Exemplo 4

Demonstre que o número T_n de regiões no plano criadas por n retas em posição geral é igual a

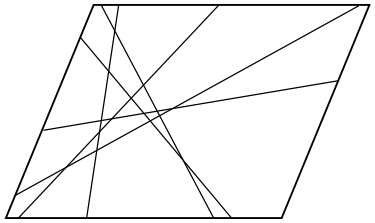
$$T_n = \frac{n(n+1)}{2} + 1.$$

Um conjunto de retas está em posição geral no plano se

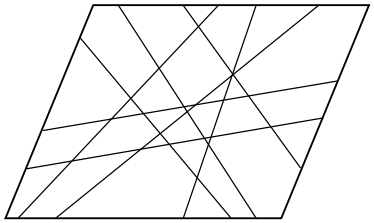
- todas as retas são concorrentes, isto é, não há retas paralelas e
- não há três retas interceptando-se no mesmo ponto.

Exemplo 4 (cont.)

Antes de prosseguirmos com a demonstração vejamos exemplos de um conjunto de retas que está em posição geral e outro que não está.



Em posição geral



Não estão em posição geral

Exemplo 4 (cont.)

Demonstração: A idéia que queremos explorar para o passo de indução é a seguinte: supondo que a fórmula vale para n, adicionar uma nova reta em posição geral e tentar assim obter a validade de $n+1$.

- A **base da indução** é, naturalmente, $n = 1$. Uma reta sozinha divide o plano em duas regiões. De fato,

$$T_1 = (1 \times 2)/2 + 1 = 2.$$

Isto conclui a prova para $n = 1$.

Exemplo 4 (cont.)

- A **hipótese de indução** é: *Suponha que* $T_n = (n(n+1)/2) + 1$ *para n.*
- O **passo de indução é:** *Supondo a h.i., vamos mostrar que para* $n+1$ *retas em posição geral vale que*

$$T_{n+1} = \frac{(n+1)(n+2)}{2} + 1.$$

Considere um conjunto L de $n+1$ retas em posição geral no plano e seja r uma dessas retas. Então, as retas do conjunto $L' = L \setminus \{r\}$ obedecem à hipótese de indução e, portanto, o número de regiões distintas do plano definidas por elas é $(n(n+1))/2 + 1$.

Exemplo 4 (cont.)

- Além disso, r intersecta as outras n retas em n pontos distintos. O que significa que, saindo de uma ponta de r no infinito e após cruzar as n retas de L', a reta r terá cruzado $n+1$ regiões, dividindo cada uma destas em duas outras.
- Assim, podemos escrever que

$$\begin{aligned} T_{n+1} &= T_n + n + 1 \\ &= \frac{n(n+1)}{2} + 1 + n + 1 \text{ (pela h.i.)} \\ &= \frac{(n+1)(n+2)}{2} + 1. \end{aligned}$$

Isso conclui a demonstração. ■

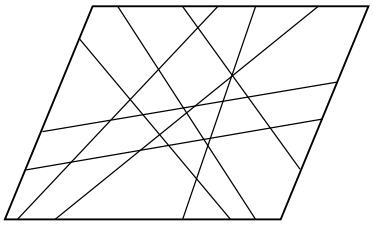
Exemplo 5

Definição:

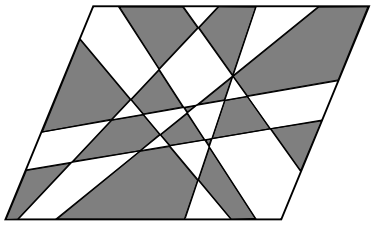
Um conjunto de *n* retas no plano define regiões convexas cujas bordas são segmentos das *n* retas. Duas dessas regiões são *adjacentes* se as suas bordas se intersectam em algum segmento de reta não trivial, isto é contendo mais que um ponto.
Uma *k-coloração* dessas regiões é uma atribuição de uma de *k* cores a cada uma das regiões, de forma que regiões adjacentes recebam cores distintas.

Exemplo 5 (cont.)

Veja exemplos dessas definições:



As regiões convexas



Uma 2-coloração do plano

Exemplo 5 (cont.)

Demonstre que para todo $n \geq 1$, existe uma 2-coloração das regiões formadas por n retas no plano.

Demonstração:

- A **base da indução** é, naturalmente, $n = 1$. Uma reta sozinha divide o plano em duas regiões. Atribuindo-se cores diferentes a essas regiões obtemos o resultado desejado.

  Isto conclui a prova para $n = 1$.

Exemplo 5 (cont.)

- A **hipótese de indução** é: *Suponha que sempre existe uma 2-coloração das regiões formadas por n retas no plano.*
- O **passo de indução é:** *Supondo a h.i., vamos exibir uma 2-coloração para as regiões formadas por $n + 1$ retas no plano.*

  A demonstração do passo consiste em observar que a adição de uma nova reta r divide cada região atravessada por r em duas, e definir a nova 2-coloração da seguinte forma: as regiões em um lado de r mantém a cor herdada da hipótese de indução; as regiões no outro lado de r têm suas cores trocadas.

  Você é capaz de demonstrar que a 2-coloração obtida nesse processo obedece à definição?

Exemplo 6

Vejamos agora um exemplo onde a indução é aplicada de forma um pouco diferente.

Demonstre que a série S_n definida abaixo satisfaz

$$S_n = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^n} < 1,$$

para todo inteiro $n \geq 1$.

Demonstração: A base é $n = 1$, para a qual a inequação se reduz a $\frac{1}{2} < 1$, obviamente verdadeira.
Como **hipótese de indução**, supomos que $S_n < 1$ para um valor $n \geq 1$. Vamos mostrar que $S_{n+1} < 1$.

Exemplo 6 (cont.)

Pela definição de S_n, temos $S_{n+1} = S_n + \frac{1}{2^{n+1}}$.
Pela hipótese de indução, $S_n < 1$. Entretanto, nada podemos dizer acerca de S_{n+1} em conseqüência da hipótese, já que não há nada que impeça que $S_{n+1} \geq 1$.
A idéia aqui é manipular S_{n+1} um pouco mais:

$$\begin{aligned} S_{n+1} &= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^{n+1}} \\ &= \frac{1}{2} + \frac{1}{2}\left[\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^n}\right] \\ &< \frac{1}{2} + \frac{1}{2} \times 1 \text{ (pela h.i.)} \\ &= 1. \end{aligned}$$

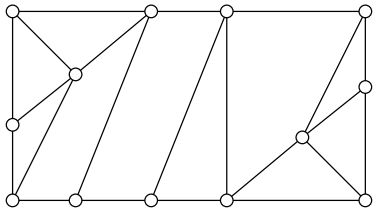
Isto conclui a demonstração. ■

Exemplo 7

Veremos a seguir um exemplo da aplicação de indução em Teoria dos Grafos.

Definição:

Um *grafo planar* é um grafo que pode ser desenhado no plano sem que suas arestas se cruzem. Um *grafo plano* é um desenho de grafo planar no plano, sem cruzamento de arestas (há inúmeros desenhos possíveis). Veja um exemplo de um grafo planar e um desenho possível dele no plano.



Exemplo 7 (cont.)

Definição:

- Um grafo plano define um conjunto F de *faces* no plano, que são as regiões contínuas **maximais** do desenho, livre de segmentos de retas ou pontos.
- Os *componentes* de um grafo são seus subgrafos maximais para os quais existe um caminho entre quaisquer dois de seus vértices.
- Dado um grafo plano G, com v vértices, e arestas, f faces e c componentes, a *Fórmula de Euler* (F.E.) é a equação

$$v - e + f = 1 + c.$$

Queremos demonstrar a Fórmula de Euler por indução.

Exemplo 7 (cont.)

- Há várias possibilidades para se fazer indução neste caso. No livro de U. Manber encontra-se uma indução em f cuja base é provada por indução em v. Além disso, lá a Fórmula de Euler está descrita simplificadamente, assumindo que o grafo é conexo. Isso torna a indução um pouco mais complicada.
- Nossa formulação aqui é mais geral simplificando a demonstração. Esse um fenômeno não é incomum em matemática: formulações mais poderosas muitas vezes resultam em demonstrações mais simples.
- Vamos demonstrar a F.E. por indução em e, o número de arestas do grafo plano G.

Exemplo 7 (cont.)

Demonstração:

- A base da indução é $e = 0$. Temos $f = 1$ e $c = v$ e

$$v - e + f = v + 1 = 1 + c$$

  como desejado. Isso demonstra a base.
- A hipótese de indução é: *Suponha que a F.E. vale para todo grafo com $e - 1$ arestas.*
- Seja G um grafo plano com e arestas, v vértices, f faces e c componentes. Seja a uma aresta qualquer de G. A remoção de a de G cria um novo grafo plano G' com $v' = v$ vértices e $e' = e - 1$ arestas, f' faces e c' componentes.
  A remoção de a de G pode ou não ter desconectado um componente de G. Caso tenha, $c' = c + 1$ e $f' = f$ (por quê?). Caso contrário, teremos $c' = c$ e $f' = f - 1$ (por quê?).

Exemplo 7 (cont.)

- No caso em que houve a criação de novo componente, temos

$$\begin{aligned} v - e + f &= v' - (e' + 1) + f' \\ &= 1 + c' - 1 \text{ (pela h.i.)} \\ &= c' \\ &= 1 + c. \end{aligned}$$

- Caso contrário obtemos

$$\begin{aligned} v - e + f &= v' - (e' + 1) + (f' + 1) \\ &= v' - e' + f' \\ &= 1 + c' \text{ (pela h.i.)} \\ &= 1 + c. \end{aligned}$$

Em ambos casos obtemos o resultado desejado. ■

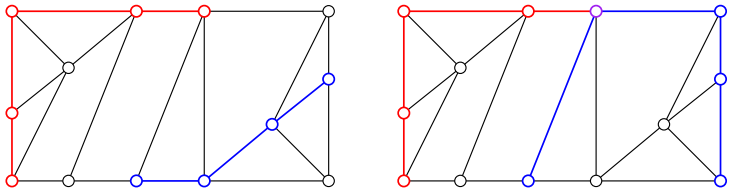
Exemplo 8

Este é um exemplo de indução forte. Antes algumas definições:

- Seja G um grafo não-orientado. O **grau** de um vértice v de G é o número de arestas incidentes a v, onde laços (arestas cujos extremos coincidem) são contados duas vezes.
- Um vértice é **ímpar** (**par**) se o seu grau é ímpar (par). O número de vértices ímpares em um grafo é sempre par (por quê?).

Exemplo 8 (cont.)

Dois caminhos em G são **aresta-disjuntos** se não têm arestas em comum. Veja exemplos de caminhos aresta-disjuntos em um grafo:



Exemplo 8 (cont.)

Teorema:
Seja G um grafo (não-orientado) conexo e I o conjunto de vértices ímpares de G. Então é possível encontrar alguma partição de I em $|I|/2$ pares e caminhos aresta-disjuntos cujos extremos são os vértices de cada par.

Demonstração: Se $I = \emptyset$, o resultado é vacuosamente verdadeiro. A demonstração é por indução no número de arestas de G. Seja e esse número.

- A base da indução trata do caso $e = 1$, ou seja, de um grafo com uma única aresta e com exatamente dois vértices (pois G é conexo). Neste caso, $|I| = 2$ e o teorema é trivialmente verdadeiro.

Exemplo 8 (cont.)

- A hipótese de indução (forte) é:
  Suponha que para todos os grafos conexos com menos que e arestas vale o resultado do enunciado do teorema.

  Vamos mostrar que o resultado vale para todo grafo conexo com e arestas.
- Seja então G um grafo qualquer com e arestas. Se $I = \emptyset$ não há nada que provar. Caso contrário existe pelo menos um par u, v de vértices de I. Como G é conexo, existe um caminho π em G cujos extremos são u, v.
- Seja G' o grafo obtido removendo-se de G as arestas de π. O grafo G' tem menos que e arestas (e dois vértices ímpares a menos).
- Embora seja tentador aplicar a h.i. a G', nada garante que G' seja conexo. Se não for, a h.i. não se aplica.

Exemplo 8 (cont.)

- É possível consertar a situação mudando a hipótese para:
  Suponha que para todos os grafos com menos que e arestas vale o seguinte: é possível encontrar alguma partição de I em $|I|/2$ pares, cada par na mesma componente, e os caminhos entre esses pares.

  Veja que removemos a restrição de conexidade, fortalecendo a hipótese.
- Com essa nova hipótese, a demonstração é a mesma. Aqui escolhemos x e y na mesma componente para garantir a existência do caminho π. Agora podemos aplicar a h.i. a G': os caminhos de G' e π são todos aresta-disjuntos e formam o conjunto de caminhos desejados para G. ■

Exercício: Você consegue demonstrar esse mesmo teorema usando indução fraca ?

Algumas armadilhas - redução $\times$ expansão

- A demonstração do passo da indução simples supõe a proposição válida para um $n-1$ e mostra que é válida para n.
- Portanto, devemos sempre partir de um caso geral n e **reduzi-lo** ao caso $n-1$. Às vezes porém, parece mais fácil pensar no caso $n-1$ e **expandi-lo** para o caso geral n.
- O perigo do procedimento de expansão é que ele não seja suficientemente geral, de forma que obtenhamos a implicação, a partir do caso $n-1$, para um caso **geral** n.
- As conseqüências de um lapso como esse podem ser a obtenção de uma estrutura de tamanho n fora da hipótese de indução, ou a prova da proposição para casos particulares de estruturas de tamanho n e não todos, como se deveria fazer.

Algumas armadilhas - redução $\times$ expansão

Por exemplo, se na demonstração da Fórmula de Euler tivéssemos optado por adicionar uma aresta $a = (i, j)$ a um grafo plano de $e-1$ arestas, deveríamos:

- garantir que i, j estivessem na mesma face; e
- considerar os casos em que i, j estivessem em componentes iguais e diferentes.

Caso contrário, ou produziríamos um cruzamento de arestas, ou provaríamos o teorema para uma **subclasse** de grafos planos apenas.

Mesmo com cuidados extras, a sensação de estar esquecendo algum caso é maior na expansão do que na redução por estarmos aumentando o universo de possibilidades.

Algumas armadilhas - outros passos mal dados

O que há de errado com a demonstração da seguinte proposição, claramente falsa ?

Proposição:

Considere n retas no plano, concorrentes duas a duas. Então existe um ponto comum a todas as n retas.

Demonstração:

- A base da indução é o caso $n = 1$, claramente verdadeiro.
- Para o caso $n = 2$, também é fácil ver que a proposição é verdadeira.
- Considere a proposição válida para $n - 1$, $n > 2$, e considere n retas no plano concorrentes duas a duas.

Algumas armadilhas - outros passos mal dados

Pela h.i., todo subconjunto de $n - 1$ das n retas têm um ponto em comum. Sejam S_1, S_2 dois desses subconjuntos, distintos entre si.

A interseção $S_1 \cap S_2$ contém $n - 2$ retas. Portanto, o ponto em comum às retas de S_1 tem que ser igual ao ponto em comum às retas de S_2, senão duas retas *distintas* de $S_1 \cap S_2$ teriam mais de um ponto em comum, o que não é possível.

Portanto, a asserção vale para n, completando a demonstração. *Certo?*

Errado!

O argumento no passo de indução funciona para todo $n > 2$, exceto $n = 3$. pois nesse caso $S_1 \cap S_2$ contém apenas uma reta. Não é possível concluir a validade para $n = 3$. De fato, a afirmação não vale para $n \geq 3$.

Indução matemática no projeto de algoritmos

Projeto de algoritmos por indução

- A seguir, usaremos a técnica de indução para desenvolver algoritmos para certos problemas.
- Isto é, a formulação do algoritmo vai ser análoga ao desenvolvimento de uma demonstração por indução.
- Assim, para resolver o problema P falamos do projeto de um algoritmo em dois passos:
  1. Exibir a resolução de uma ou mais instâncias pequenas de P (casos base);
  2. Exibir como a solução de toda instância de P pode ser obtida a partir da solução de uma ou mais instâncias menores de P.

Projeto de algoritmos por indução

Este processo indutivo resulta em algoritmos recursivos, em que:

- a base da indução corresponde à resolução dos casos base da recursão;
- a aplicação da hipótese de indução corresponde a uma ou mais chamadas recursivas; e
- o passo da indução corresponde ao processo de obtenção da resposta para o caso geral a partir daquelas devolvidas pelas chamadas recursivas.

Projeto de algoritmos por indução

- Um benefício imediato é que o uso (correto) da técnica nos dá uma prova da corretude do algoritmo.
- A complexidade do algoritmo resultante é expressa numa recorrência.
- Freqüentemente o algoritmo é eficiente, embora existam exemplos simples em que isso não acontece.
- Iniciaremos com dois exemplos que usam indução:
  1. cálculo do valor de polinômios e
  2. obtenção de subgrafos maximais com restrições de grau.

Exemplo 1 - Cálculo de polinômios

Problema:

Dada uma seqüência de números reais $a_n, a_{n-1}, \ldots, a_1, a_0$, e um número real x, calcular o valor do polinômio

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0.$$

Naturalmente este é um problema bem simples.
Estamos interessados em projetar um algoritmo que faça o menor número de operações aritméticas (multiplicações, principalmente).

Exemplo 1 - Cálculo de polinômios

Problema:

Dados uma seqüência de números reais $a_n, a_{n-1}, \ldots, a_1, a_0$, e um número real x, calcular o valor do polinômio

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0.$$

Hipótese de indução: (primeira tentativa)

Dados uma seqüência de números reais $a_{n-1}, \ldots, a_1, a_0$, e um número real x, sabemos calcular o valor de $P_{n-1}(x) = a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$.

- **Caso base**: $n = 0$. A solução é a_0.
- Para calcular $P_n(x)$, basta calcular x^n, multiplicar o resultado por a_n e somar o resultado com $P_{n-1}(x)$.

Exemplo 1 - Solução 1 - Algoritmo

CálculoPolinômio(A,x)

▷ **Entrada:** Coeficientes $A = a_n, a_{n-1}, \ldots, a_1, a_0$ e real x.
▷ **Saída:** O valor de $P_n(x)$.

1. **se** $n = 0$ **então** $P \leftarrow a_0$
2. **senão**
3. $\quad A' \leftarrow a_{n-1}, \ldots, a_1, a_0$
4. $\quad P' \leftarrow$ CálculoPolinômio(A', x)
5. $\quad xn \leftarrow 1$
6. $\quad$ **para** $i \leftarrow 1$ **até** n **faça** $xn \leftarrow xn * x$
7. $\quad P \leftarrow P' + a_n * xn$
8. **devolva** (P)

Exemplo 1 - Solução 1 - Complexidade

Chamando de $T(n)$ o número de operações aritméticas realizadas pelo algoritmo, temos a seguinte recorrência para $T(n)$:

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + (n+1) \text{ multiplicações} + 1 \text{ adição}, & n > 0. \end{cases}$$

Não é difícil ver que

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} [(i+1) \text{ multiplicações} + 1 \text{ adição}] \\ &= (n+3)n/2 \text{ multiplicações} + n \text{ adições}. \end{aligned}$$

Nota: o número de multiplicações pode ser diminuído calculando x^n com o algoritmo de exponenciação rápida que veremos mais tarde.

Segunda solução indutiva

- Desperdício cometido na primeira solução: recálculo de potências de x.
- Alternativa: eliminar essa computação desnecessária trazendo o cálculo de x^{n-1} para dentro da hipótese de indução.

Hipótese de indução reforçada:

Sabemos calcular o valor de $P_{n-1}(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ e também o valor de x^{n-1}.

- Então, no passo de indução, primeiro calculamos x^n multiplicando x por x^{n-1}, conforme exigido na hipótese. Em seguida, calculamos $P_n(x)$ multiplicando x^n por a_n e somando o valor obtido com $P_{n-1}(x)$.
- Note que para o caso base $n = 0$, a solução agora é $(a_0, 1)$.

Exemplo 1 - Solução 2 - Algoritmo

CálculoPolinômio(A,x)

▷ **Entrada:** Coeficientes $A = a_n, a_{n-1}, \ldots, a_1, a_0$ e real x.
▷ **Saída:** O valor de $P_n(x)$ e o valor de x^n.

1. **se** $n = 0$ **então** $P \leftarrow a_0$; $xn \leftarrow 1$
2. **senão**
3. $\quad A' \leftarrow a_{n-1}, \ldots, a_1, a_0$
4. $\quad P', x' \leftarrow$ CálculoPolinômio(A', x)
5. $\quad xn \leftarrow x * x'$
6. $\quad P \leftarrow P' + a_n * xn$
7. **devolva** (P, xn)

Exemplo 1 - Solução 2 - Complexidade

Novamente, se $T(n)$ é o número de operações aritméticas realizadas pelo algoritmo, $T(n)$ é dado por:

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 2 \text{ multiplicações} + 1 \text{ adição}, & n > 0. \end{cases}$$

A solução da recorrência é

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} (2 \text{ multiplicações} + 1 \text{ adição}) \\ &= 2n \text{ multiplicações} + n \text{ adições}. \end{aligned}$$

Terceira solução indutiva

- A escolha de considerar o polinômio $P_{n-1}(x)$ na hipótese de indução não é a única possível.
- Podemos **reforçar** ainda mais a h.i. e ter um ganho de complexidade:

Hipótese de indução mais reforçada:

Sabemos calcular o valor do polinômio $P'_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{n-2} \ldots + a_1$.

- Note que $P_n(x) = xP'_{n-1}(x) + a_0$. Assim, bastam uma multiplicação e uma adição para obtermos $P_n(x)$ a partir de $P'_{n-1}(x)$.
- O caso base é trivial pois, para $n = 0$, a solução é a_0.

Exemplo 1 - Solução 3 - Algoritmo

CálculoPolinômio(A,x)

▷ **Entrada:** Coeficientes $A = a_n, a_{n-1}, \ldots, a_1, a_0$ e real x.
▷ **Saída:** O valor de $P_n(x)$.
1. **se** $n = 0$ **então** $P \leftarrow a_0$
2. **senão**
3. $\quad A' \leftarrow a_n, a_{n-1}, \ldots, a_1$
4. $\quad P' \leftarrow$ CálculoPolinômio(A', x)
5. $\quad P \leftarrow x * P' + a_0$
6. **devolva** (P)

Exemplo 1 - Solução 3 - Complexidade

Temos que $T(n)$ é dado por

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 1 \text{ multiplicação} + 1 \text{ adição}, & n > 0. \end{cases}$$

A solução é

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} (1 \text{ multiplicação} + 1 \text{ adição}) \\ &= n \text{ multiplicações} + n \text{ adições}. \end{aligned}$$

Essa forma de calcular $P_n(x)$ é chamada de **regra de Horner**.

Projeto por indução - Exemplo 2
Subgrafos Maximais

- Suponha que você esteja planejando uma festa onde queira maximizar as interações entre as pessoas. Uma festa animada, mas sem recursos ilícitos para aumentar a animação.
- Uma das maneiras de conseguir isso é fazer uma lista dos possíveis convidados e, para cada um desses, a lista dos convidados com quem ele(a) interagiria durante a festa.
- Em seguida, é só localizar o maior subgrupo de convidados que interagiriam com, no mínimo, digamos, k outros convidados dentro do subgrupo.

Exemplo 2 - Subgrafos Maximais

Formulação do problema usando grafos:

Dado um grafo simples G (não-orientado) com n vértices e um inteiro $k \leq n$, encontrar em G um subgrafo induzido H com o número máximo de vértices, tal que o grau de cada vértice de H seja $\geq k$. Se um tal subgrafo H não existir, o algoritmo deve identificar esse fato.

Definições:

H é um subgrafo induzido de um grafo G, uma aresta de G está em H se, e somente se, tem ambos os seus extremos em H.

Um subgrafo qualquer G' de G é maximal com relação a uma propriedade P se G' satisfizer P e não existir em G outro subgrafo próprio que contenha G' e satisfaça P.

Exemplo 2 (cont.)

- Exemplo de um subgrafo que é induzido e de outro subgrafo que não é induzido em um grafo G:

INDUZIDO NÃO INDUZIDO

- Como exemplo de subgrafos maximais (propriedade relativa à continência) que não são máximos (propriedade relativa ao tamanho) em um grafo G, veja definição de clique em grafos.
- No nosso exemplo, o conceito de *maximal* é equivalente ao de *máximo* (número de vértices).
  Você pode provar esta afirmativa ?

Exemplo 2 - Indução

Usando o método indutivo:

Hipótese de indução:

Dados inteiros n, k e um grafo G com menos que n vértices, sabemos como encontrar um subgrafo maximal H de G com grau mínimo $\geq k$.

Caso base: o primeiro valor de n para o qual faz sentido buscarmos tais subgrafos H é $n = k + 1$, caso contrário não há como satisfazer a restrição de grau mínimo.

Assim, quando $n = k + 1$, a única forma de existir em G um subgrafo induzido com grau mínimo k é que **todos** os vértices tenham grau igual a k.

Se isso ocorrer, a resposta é $H = G$; caso contrário $H = \emptyset$.

Exemplo 2 - Indução (cont.)

- Seja então G um grafo com n vértices e $k \geq 0$ um inteiro tal que $n > k + 1$.
- Se todos os vértices de G têm grau $\geq k$ não há nada mais a fazer. Devolva $H = G$.
- Caso contrário, seja v um vértice com grau $< k$. É fácil ver que nenhum subgrafo que é solução para o problema conterá v. Assim, v pode ser removido e a hipótese de indução aplicada ao grafo resultante G'.
- Seja H' o subgrafo devolvido pela aplicação da h.i. em G'. Então H' também é resposta para G. ■

Exemplo 2 - Algoritmo

SubgrafoMaximal(G, k)

▷ **Entrada:** Grafo G com n vértices e um inteiro $k \geq 0$.
▷ **Saída:** Subgrafo induzido H de G com grau mínimo k.

```
1.  se (n < k + 1) então H ← ∅
2.  senão se todo vértice de G tem grau ≥ k
3.      então H ← G
4.      senão
5.          seja v um vértice de G com grau < k
6.          H ← SubgrafoMaximal(G − v, k)
7.  devolva H
```


Projeto por indução - Exemplo 3
Fatores de balanceamento em árvores binárias

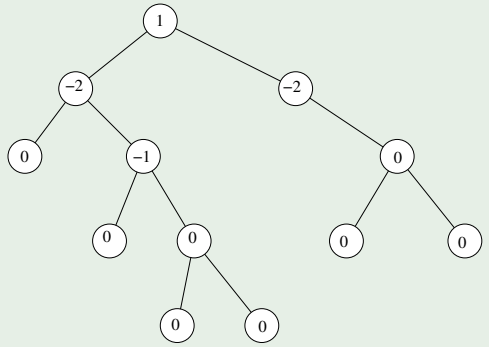
Definição:

Árvores binárias balanceadas são estruturas de dados que minimizam o tempo de busca de informações nela armazenadas. A idéia é que, para todo nó v da árvore, o *fator de balanceamento (f.b.)* de v, isto é, a diferença entre a altura da subárvore esquerda e a altura da subárvore direita de v não desvie muito de zero.

- Convenciona-se que a árvore vazia tem fator de balanceamento *zero* e altura igual a -1.
- Árvores AVL são exemplos de árvores binárias balanceadas, em que o f.b. de cada nó é $-1, 0$ ou $+1$.

Exemplo 3 (cont.)

Exemplo de uma árvore binária e os fatores de balanceamento de seus nós.



Exemplo 3 (cont.)

Problema:

Dada uma árvore binária A com n nós, calcular os fatores de balanceamento de cada nó de A.

- Vamos projetar o algoritmo indutivamente.

Hipótese de indução:

Sabemos como calcular fatores de balanceamento de árvores com menos que n nós.

- **Caso base**: quando $n = 0$, convencionamos que o f.b. é igual a zero.
- Vamos mostrar agora como usar a hipótese para calcular f.b.s de uma árvore A com exatamente n nós.

Exemplo 3 - indução (cont.)

A idéia é aplicar a h.i. às subárvores esquerda e direita da raiz e, em seguida, calcular o f.b. da raiz.

Dificuldade: o f.b. da raiz depende das alturas das subárvores esquerda e direita e não dos seus f.b.s.

Conclusão: é necessário uma h.i. mais forte!

Nova hipótese de indução:

Para um $n > 0$ qualquer, sabemos como calcular fatores de balanceamento e alturas de árvores com menos que n nós.

Exemplo 3 - indução (cont.)

- Novamente, o caso base $n = 0$ é fácil pois, por convenção, o f.b. é nulo e a altura igual a -1.
- Seja A uma árvore binária com n nós, para um $n > 0$ qualquer.

  Sejam (f_e, h_e) e (f_d, h_d) os f.b.s e alturas das subárvores esquerda (A_e) e direita (A_d) de A que, por h.i., sabemos como calcular.

  Então, o f.b. da raiz de A é $h_e - h_d$ e a altura da árvore é $\max(h_e, h_d) + 1$.

  Isto completa o cálculo dos f.b.s e da altura de A. ■

De novo, o fortalecimento da hipótese tornou a resolução do problema mais fácil.

Exemplo 3 - Algoritmo

FatorAltura (A)

▷ **Entrada:** Uma árvore binária A com $n \geq 0$ nós
▷ **Saída:** A árvore A os f.b.s nos seus nós e a altura de A

```
1.  se n = 0 então h ← −1
2.  senão
3.      seja r a raiz de A
3.      h_e ← FatorAltura(A_e)
4.      h_d ← FatorAltura(A_d)
5.      ▷ armazena o f.b. na raiz em r.f
6.      r.f ← h_e − h_d
7.      h ← max(h_e, h_d) + 1
8.  devolva h
```

Exemplo 3 - Complexidade

Seja $T(n)$ o número de operações executadas pelo algoritmo para calcular os f.b.s e a altura de uma árvore A de n nós. Então

$$T(n) = \begin{cases} \Theta(1), & n = 0 \\ T(n_e) + T(n_d) + \Theta(1), & n > 0, \end{cases}$$

onde n_e, n_d são os números de nós das subárvores esquerda e direita.

Exemplo 3 - Complexidade

O pior caso da recorrência parece ser quando, ao longo da recursão, um de n_e, n_d é sempre igual a zero (e o outro é sempre igual a $n-1$).

Chega-se então a:

$$T(n) = T(1) + \sum_{i=2}^{n} \Theta(1) = (n+1)\Theta(1) + n\Theta(1) \in \Theta(n).$$

Exercício:

Há algum ganho em complexidade quando ambos n_e e n_d são aproximadamente $n/2$ ao longo da recursão?

Projeto por Indução - Exemplo 4: o problema da celebridade

Definição

Num conjunto S de n pessoas, uma *celebridade* é alguém que é conhecido por todas as pessoas de S mas que não conhece ninguém. (Celebridades são pessoas de difícil convívio...).

Note que pode existir no máximo uma celebridade em S!

Problema:

Queremos saber se existe uma celebridade em S.

Projeto por Indução - Exemplo 4: o problema da celebridade

Vamos formalizar melhor: para um conjunto de n pessoas, associamos uma matriz $n \times n$ M tal que $M[i,j] = 1$ se a pessoa i conhece a pessoa j e $M[i,j] = 0$ caso contrário.

Problema:

Dado um conjunto de n pessoas e a matriz associada M encontrar (se existir) uma celebridade no conjunto.
Isto é, determinar um k tal que todos os elementos da coluna k (exceto $M[k,k]$) são 1s e todos os elementos da linha k (exceto $M[k,k]$) são 0s.

Existe uma solução simples mas laboriosa: para cada pessoa i, verifique todos os outros elementos da linha i e da coluna i. O custo dessa solução é $2n(n-1)$.

Exemplo 4 - Indução

Um argumento indutivo que parece ser mais eficiente é o seguinte:

Hipótese de Indução:

Sabemos encontrar uma celebridade (se existir) em um conjunto de $n-1$ pessoas.

- Se $n = 1$, podemos considerar que o único elemento é uma celebridade.
- Outra opção seria considerarmos o caso base como $n = 2$, o primeiro caso interessante.
  A solução é simples: existe uma celebridade se, e somente se, $M[1,2] \oplus M[2,1] = 1$. Mais uma comparação define a celebridade: se $M[1,2] = 0$, então a celebridade é a pessoa 1; se não, é a pessoa 2.

Exemplo 4 - Indução (cont.)

Tome então um conjunto $S = \{1, 2, \ldots, n\}, n > 2$, de pessoas e a matriz M associada. Considere o conjunto $S' = S \setminus \{n\}$;

Há dois casos possíveis:

1. Existe uma celebridade em S', digamos a pessoa k; então, k é celebridade em S se, e somente se, $M[n,k] = 1$ e $M[k,n] = 0$.
2. Se k não for celebridade em S ou não existir celebridade em S', então a pessoa n é celebridade em S se $M[n,j] = 0$ e $M[j,n] = 1, \forall j < n$; caso contrário não há celebridade em S.

Essa primeira tentativa, infelizmente, também conduz a um algoritmo quadrático em n. Por quê?

Obs: quadrático em n mas linear no tamanho da entrada!!!

Exemplo 4 - Segunda tentativa

A segunda tentativa baseia-se em um fato muito simples:

*Dadas duas pessoas i e j, é possível determinar se uma delas **não** é uma celebridade com apenas uma comparação: se $M[i,j] = 1$, então i não é celebridade; caso contrário j não é celebridade.*

Vamos usar esse argumento aplicando a hipótese de indução sobre o conjunto de $n-1$ pessoas obtidas removendo dentre as n uma pessoa que sabemos não ser celebridade.

- O caso base e a hipótese de indução são os mesmos que anteriormente.

Exemplo 4 - Segunda tentativa

Tome então um conjunto arbitrário de $n > 2$ pessoas e a matriz M associada.

Sejam i e j quaisquer duas pessoas e suponha que, usando o argumento acima, determinemos que j não é celebridade.

Seja $S' = S \setminus \{j\}$ e considere os dois casos possíveis:

1. Existe uma celebridade em S', digamos a pessoa k. Se $M[j,k] = 1$ e $M[k,j] = 0$, então k é celebridade em S; caso contrário não há uma celebridade em S.
2. Não existe uma celebridade em S'; então não existe uma celebridade em S.

Exemplo 4 - Algoritmo

Celebridade(S, M)

▷ **Entrada:** conjunto de pessoas $S = \{1, 2, \ldots, n\}$;
M, a matriz que define quem conhece quem em S.
▷ **Saída:** Um inteiro $k \leq n$ que é celebridade em S ou $k = 0$

```
1.  se |S| = 1 então k ← elemento em S
2.  senão
3.      sejam i, j quaisquer duas pessoas em S
4.      se M[i, j] = 1 então s ← i senão s ← j
5.      S' ← S \ {s}
6.      k ← Celebridade(S', M)
7.      se k > 0 então
8.          se (M[s, k] ≠ 1) ou (M[k, s] ≠ 0) então k ← 0
9.  devolva k
```


Exemplo 4 - Complexidade

O algoritmo resultante tem complexidade linear em n.

A recorrência $T(n)$ para o número de operações executadas pelo algoritmo é:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ T(n-1) + \Theta(1), & n > 1. \end{cases}$$

A solução desta recorrência é

$$\sum_{1}^{n} \Theta(1) = n\Theta(1) = \Theta(n).$$

Projeto por indução - Exemplo 5
Subseqüência consecutiva máxima (SCM)

Problema:

Dada uma seqüência $X = x_1, x_2, \ldots, x_n$ de números reais (não necessariamente positivos), encontrar uma ***subseqüência consecutiva*** *$Y = x_i, x_{i+1}, \ldots, x_j$ de X, onde $1 \leq i, j \leq n$, cuja soma seja máxima dentre todas as subseqüências consecutivas.*

Exemplos:

$X = [4, 2, -7, 3, 0, -2, 1, 5, -2]$	Resp: $Y = [3, 0, -2, 1, 5]$
$X = [-1, -2, 0]$	Resp: $Y = [0]$ ou $Y = [\,]$
$X = [-3, -1]$	Resp: $Y = [\,]$

Exemplo 5 - Indução

Como antes, vamos examinar o que podemos obter de uma **hipótese de indução** simples:

Hipótese de indução:

Sabemos calcular a SCM *de seqüências de comprimento $n - 1$.*

- Seja então $X = x_1, x_2, \ldots, x_n$ uma seqüência qualquer de comprimento $n > 1$.
- Considere a seqüência X' obtida de X removendo-se x_n.
- Seja $Y' = x_i, x_{i+1}, \ldots, x_j$ a SCM de X', obtida aplicando-se a h.i.

Exemplo 5 - Indução

Há três casos a examinar:

1. $Y' = [\quad]$. Neste caso, $Y = x_n$ se $x_n \geq 0$ ou $Y = [\quad]$ se $x_n < 0$.
2. $j = n - 1$. Como no caso anterior, temos $Y = Y'||x_n$ se $x_n \geq 0$ ou $Y = Y'$ se $x_n < 0$.
3. $j < n - 1$. Aqui há dois subcasos a considerar:
   1. Y' também é SCM de X; isto é, $Y = Y'$.
   2. Y' não é a SCM de X. Isto significa que x_n é parte de uma SCM Y de X. Esta tem que ser da forma $x_k, x_{k+1}, \ldots, x_{n-1}, x_n$, para algum $k \leq n - 1$.

Exemplo 5 - Indução

- A hipótese de indução nos permite resolver todos os casos anteriores, exceto o último.
  Não há informação suficiente na h.i. para permitir a resolução deste caso.
  O que falta na h.i.?
- É evidente que, quando

$$Y = x_k, x_{k+1}, \ldots, x_{n-1}, x_n,$$

  então $x_k, x_{k+1}, \ldots, x_{n-1}$ é um *sufixo* de X' de soma máxima entre os *sufixos* de X'.
- Assim, se conhecermos o sufixo máximo de X', além da SCM, teremos resolvido o problema completamente para X.

Exemplo 5 - Indução

Parece então natural enunciar a seguinte h.i. fortalecida:

Hipótese de indução reforçada:

Sabemos calcular a SCM *e o sufixo máximo de seqüências de comprimento* $n - 1$.

É clara desta discussão também, a base da indução: para $n = 1$, a SCM de $X = x_1$ é x_1 caso $x_1 \geq 0$, e a seqüência vazia caso contrário. Nesse caso, o sufixo máximo é igual a SCM.

Exemplo 5 - Algoritmo

SCM(X, n)

- **Entrada:** um inteiro n e uma seqüência de n números reais $X = [x_1, x_2, \ldots, x_n]$.
- **Saída:** Inteiros i, j, k e reais *MaxSeq*, *MaxSuf* tais que:
  - x_i, x_j são o primeiro e último elementos da SCM de X, cujo valor é *MaxSeq*; e
  - x_k é o primeiro elemento do sufixo máximo de X, cujo valor é *MaxSuf*.
  - O valor $j = 0$ significa que X é composta de negativos somente. Neste caso, convencionamos *MaxSeq* $= 0$.
  - O valor $k = 0$ significa que o sufixo máximo de X é vazio. Neste caso, *MaxSuf* $= 0$.

Exemplo 5 - Algoritmo (cont.)

SCM(X, n): (cont.)

```
1.  se n = 1
2.  então
3.      se x_1 < 0
4.          então i, j, k ← 0; MaxSeq, MaxSuf ← 0
5.          senão i, j, k ← 1; MaxSeq, MaxSuf ← x_1
6.  senão
7.      (i, j, k, MaxSeq, MaxSuf) ← SCM(X, n − 1)
8.      se MaxSuf = 0 então k ← n
9.      MaxSuf ← MaxSuf + x_n
10.     se MaxSuf > MaxSeq
11.         então i ← k; j ← n; MaxSeq ← MaxSuf
12.         senão se MaxSuf < 0 então MaxSuf ← 0; k ← 0
13.  devolva (i, j, k, MaxSeq, MaxSuf)
```


Exemplo 5 - Complexidade

A complexidade $T(n)$ de SCM é simples de ser calculada.
Como no último exemplo,

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ T(n-1) + \Theta(1), & n > 1. \end{cases}$$

A solução desta recorrência é

$$\sum_{1}^{n} \Theta(1) = n\Theta(1) = \Theta(n).$$

Reforçar a hipótese de indução: preciso lembrar disso ...

Projeto por indução - Erros comuns

Os erros discutidos nas provas por indução, naturalmente, traduzem-se em erros no projeto de um algoritmo por indução.

Exemplo:

Problema: *Dado um grafo conexo G, verificar se G é bipartido ou não. Caso seja, devolver a partição dos vértices.*

Definição:

Um grafo é *bipartido* se seu conjunto de vértices pode ser particionado em dois conjuntos, de forma que toda aresta de G tenha extremos em conjuntos diferentes.

Teorema:

Se G é conexo e bipartido, então a bipartição é única.

Projeto por indução - Erros comuns

O que há de **errado** com o seguinte (esboço de um) algoritmo recursivo para verificar se um grafo conexo é bipartido?

- Sejam G um grafo conexo, v um vértice de G e considere o grafo $G' = G - \{v\}$.
- Se G' não for bipartido, então G também não é. Caso contrário, sejam A e B os dois conjuntos da bipartição de G' obtidos recursivamente.
- Considere agora o vértice v e sua relação com os vértices de A e B.
- Se v tiver um vizinho em A e outro em B, então G não é bipartido (já que a bipartição, se existir, deve ser única).
- Caso contrário, adicione v a A ou B, o conjunto no qual v não tem vizinhos. A bipartição está completa.