SCC-0223 - Capítulo 2
Análise de Algoritmos

João Rosa[1]

[1]Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo - São Carlos
joaoluis@icmc.usp.br

2023

Sumário

Sumário

Algoritmo: noção geral

- **Algoritmo**[1] é um conjunto de instruções que devem ser seguidas para solucionar um determinado problema.
- Cormen *et al.* [2]:
  - Qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores de **entrada** e produz algum valor ou conjunto de valores de **saída**;
  - Ferramenta para resolver um problema computacional bem especificado;
  - Assim como o hardware de um computador, constitui uma tecnologia, pois o desempenho total do sistema depende da escolha de um algoritmo eficiente tanto quanto da escolha de um hardware rápido.

[1] A palavra "algoritmo" vem do nome de um matemático persa (825 d.C.), Abu Ja'far Mohammed ibn Musa al Khowarizmi.

Algoritmo: noção geral

- Cormen *et al.* [2]:
  - Deseja-se que um algoritmo termine e seja correto.
- Perguntas:
  - Mas um algoritmo correto vai terminar, não vai?
  - A afirmação está redundante?

Eficiência e Problemas Difíceis

- Além de um algoritmo correto, busca-se também um algoritmo **eficiente** para resolver um determinado problema
- Pergunta: como 'medir' eficiência de um algoritmo?

- Obs. Existem problemas para os quais não se conhece nenhum algoritmo eficiente para obter a solução: $\mathcal{NP}$-completos.

Recursos de um algoritmo

- Uma vez que um algoritmo está pronto/disponível, é importante determinar os **recursos necessários** para sua execução:
  - Tempo
  - Memória
- Qual o principal quesito? Por que?

Análise de algoritmos

- Um algoritmo que soluciona um determinado problema, mas requer o processamento de **um ano**, não deve ser usado.
- O que dizer de uma afirmação como a abaixo?
  "Desenvolvi um novo algoritmo chamado TripleX que leva 14,2 segundos para processar 1.000 números, enquanto o método SimpleX leva 42,1 segundos."
- Você trocaria o *SimpleX* que roda em sua empresa pelo *TripleX*?

Análise de algoritmos

- A afirmação tem que ser examinada, pois há diversos fatores envolvidos:
  - Características da máquina em que o algoritmo foi testado:
    - Quantidade de memória.
  - Linguagem de programação:
    - Compilada vs. interpretada,
    - Alto vs. baixo nível.
  - Implementação pouco cuidadosa do algoritmo *SimpleX* vs. "super" implementação do algoritmo *TripleX*.
  - Quantidade de dados processados:
    - Se o *TripleX* é mais rápido para processar 1.000 números, ele também é mais rápido para processar quantidades maiores de números, certo?

Análise de algoritmos

- A comunidade de computação começou a pesquisar formas de comparar algoritmos de forma independente de
  - Hardware,
  - Linguagem de programação,
  - Habilidade do programador.
- Portanto, quer-se comparar **algoritmos** e não **programas**:
  - Área conhecida como "análise/complexidade de algoritmos".

Eficiência de algoritmos

- Sabe-se que:
  - Processar 10.000 números leva mais tempo do que 1.000 números,
  - Cadastrar 10 pessoas em um sistema leva mais tempo do que cadastrar 5,
  - Etc.
- Então, pode ser uma boa idéia estimar a **eficiência** de um algoritmo em função do **tamanho** do problema:
  - Em geral, assume-se que n é o tamanho do problema, ou número de elementos que serão processados,
  - E calcula-se o número de operações que serão realizadas sobre os n elementos.

Eficiência de algoritmos

- O melhor algoritmo é aquele que requer menos operações sobre a entrada, pois é o mais rápido:
  - O tempo de execução do algoritmo pode variar em diferentes máquinas, mas o número de operações é uma boa medida de desempenho de um algoritmo.
- De que operações estamos falando?
- Toda operação leva o mesmo tempo?

Exemplo: *TripleX* vs. *SimpleX*

- *TripleX*: para uma entrada de tamanho n, o algoritmo realiza $n^2 + n$ operações:
  - Pensando em termos de função: $f(n) = n^2 + n$.
- *SimpleX*: para uma entrada de tamanho n, o algoritmo realiza $1.000n$ operações:
  - $g(n) = 1.000n$.

Exemplo: *TripleX* vs. *SimpleX*

- Faça os cálculos do desempenho de cada algoritmo para cada tamanho de entrada:

tamanho da entrada n	1	10	100	1.000	10.000
$f(n) = n^2 + n$					
$g(n) = 1.000n$					

Exemplo: *TripleX* vs. *SimpleX*

- Faça os cálculos do desempenho de cada algoritmo para cada tamanho de entrada:

tamanho da entrada n	1	10	100	1.000	10.000
$f(n) = n^2 + n$	2	110	10.100	**1.001.000**	**100.010.000**
$g(n) = 1.000n$	**1.000**	**10.000**	100.000	1.000.000	10.000.000

- A partir de $n = 1.000$, $f(n)$ mantém-se maior e cada vez mais distante de $g(n)$:
  - Diz-se que $f(n)$ cresce mais rápido do que $g(n)$.

Análise assintótica

- Devemos nos preocupar com a eficiência de algoritmos quando o tamanho de n for **grande**.
- *Definição*: a **eficiência assintótica** de um algoritmo descreve a sua eficiência relativa quando n torna-se grande.
- Portanto, para comparar 2 algoritmos, determinam-se as taxas de crescimento de cada um: o algoritmo com menor taxa de crescimento rodará mais rápido quando o tamanho do problema for grande.

Análise assintótica

- **Atenção**:
  - Algumas funções podem não crescer com o valor de n:
    - Quais?
  - Também se pode aplicar os conceitos de análise assintótica para a quantidade de memória usada por um algoritmo:
    - Mas não é tão útil, pois é difícil estimar os detalhes exatos do uso de memória e o impacto disso.

Sumário

Relembrando um pouco de matemática...

- Expoentes:
  - $x^a x^b = x^{a+b}$
  - $x^a / x^b = x^{a-b}$
  - $(x^a)^b = x^{ab}$
  - $x^n + x^n = 2x^n$ (diferente de x^{2n})
  - $2^n + 2^n = 2^{n+1}$
- Logaritmos (usaremos a base 2, a menos que seja dito o contrário):
  - $x^a = b \Rightarrow log_x b = a$
  - $log_a b = log_c b / log_c a$, se $c > 0$
  - $log\ ab = log\ a + log\ b$
  - $log\ a/b = log\ a - log\ b$
  - $log(a^b) = b\ log\ a$
  - E o mais importante:
    - $log\ x < x$ para todo $x > 0$.

Função exponencial vs. logarítmica

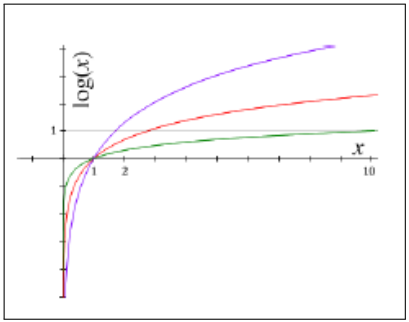


Figure 1: Exemplos de logaritmos para várias bases.

Figure 2: Na palma da mão direita.

Relembrando um pouco de matemática...

- Séries:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

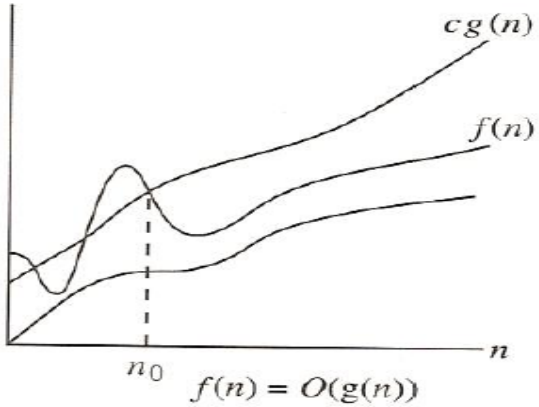
$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

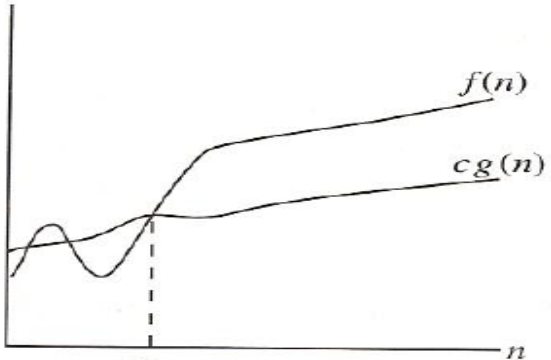
Algumas notações

- Dadas duas funções, $f(n)$ e $g(n)$,
  - diz-se que $f(n)$ é **da ordem de** (**big-oh**) $g(n)$ ou que $f(n)$ é $\mathcal{O}(g(n))$, se existirem constantes c e n_0 tais que $f(n) \leq c * g(n)$ para todo $n \geq n_0$.
    - A taxa de crescimento de $f(n)$ é menor ou igual à taxa de $g(n)$.
  - diz-se que $f(n)$ é **ômega** $g(n)$ ou que $f(n) = \Omega(g(n))$, se existirem constantes c e n_0 tais que $f(n) \geq c * g(n)$ para todo $n \geq n_0$.
    - A taxa de crescimento de $f(n)$ é maior ou igual à taxa de $g(n)$.
  - diz-se que $f(n)$ é **theta** $g(n)$ ou que $f(n) = \Theta(g(n))$, se e somente se $f(n) = \mathcal{O}(g(n))$ e $f(n) = \Omega(g(n))$.
    - A taxa de crescimento de $f(n)$ é igual à taxa de $g(n)$.
  - diz-se que $f(n)$ é **little-oh** $g(n)$ ou que $f(n) = o(g(n))$, se e somente se $f(n) = \mathcal{O}(g(n))$ e $f(n) \neq \Theta(g(n))$.
    - A taxa de crescimento de $f(n)$ é menor do que a taxa de $g(n)$.
  - diz-se que $f(n)$ é **little ômega** $g(n)$ ou que $f(n) = \omega(g(n))$, se e somente se $f(n) = \Omega(g(n))$ e $f(n) \neq \Theta(g(n))$.
    - A taxa de crescimento de $f(n)$ é maior do que a taxa de $g(n)$.

Algumas notações

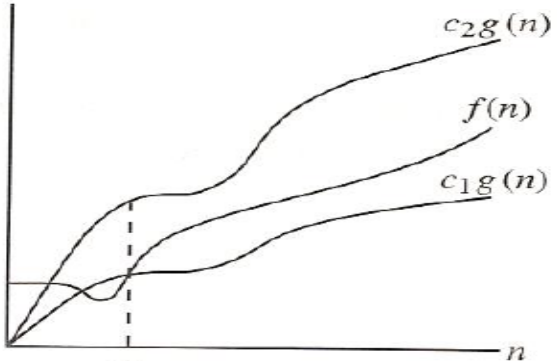


Algumas notações



$f(n) = \Omega(g(n))$

Algumas notações



$f(n) = \Theta(g(n))$

Algumas considerações

- O uso das notações permite **comparar a taxa de crescimento** das funções correspondentes aos algoritmos:
  - **Não faz sentido comparar pontos isolados das funções**, já que podem não corresponder ao comportamento assintótico.
- Ao dizer que $g(n) = \mathcal{O}(f(n))$, garante-se que $g(n)$ cresce numa taxa não maior do que $f(n)$, ou seja, $f(n)$ é seu limite superior.
- Ao dizer que $f(n) = \Omega(g(n))$, tem-se que $g(n)$ é o limite inferior de $f(n)$.

Exemplo

- Para 2 algoritmos quaisquer, considere as funções de eficiência correspondentes $1.000n$ e n^2:
  - A primeira é maior do que a segunda para valores pequenos de n,
  - A segunda cresce mais rapidamente e finalmente será uma função maior, sendo que o ponto de mudança é $n = 1.000$,
  - Segundo as notações anteriores, se existe um ponto n_0 a partir do qual $c * f(n)$ é sempre pelo menos tão grande quanto $g(n)$, então, ignorados os fatores constantes $f(n)$ é pelo menos tão grande quanto $g(n)$:
    - No nosso caso, $g(n) = 1.000n$, $f(n) = n^2$, $n_0 = 1.000$ e $c = 1$ (ou, ainda, $n_0 = 10$ e $c = 100$): Dizemos que $1.000n = \mathcal{O}(n^2)$.

Outros exemplos

1. A função n^3 cresce mais rapidamente que n^2:
   - $n^2 = \mathcal{O}(n^3)$
   - $n^3 = \Omega(n^2)$
2. Se $f(n) = n^2$ e $g(n) = 2n^2$, então essas duas funções têm taxas de crescimento iguais:
   - Portanto, $f(n) = \mathcal{O}(g(n))$ e $f(n) = \Omega(g(n))$.

Taxas de crescimento

- Algumas regras:
  - Se $T_1(n) = \mathcal{O}(f(n))$ e $T_2(n) = \mathcal{O}(g(n))$, então:
    - $T_1(n) + T_2(n) = max(\mathcal{O}(f(n)), \mathcal{O}(g(n)))$.
    - $T_1(n) * T_2(n) = \mathcal{O}(f(n) * g(n))$.
  - Se $T(x)$ é um polinômio de grau n, então:
    - $T(x) = \Theta(x^n)$.
  - Relembrando: um polinômio de grau n é uma função que possui a forma abaixo:

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

    seguindo a seguinte classificação em função do grau:
    - Grau 0: polinômio constante
    - Grau 1: função afim (polinômio linear, caso $a_0 = 0$)
    - Grau 2: polinômio quadrático
    - Grau 3: polinômio cúbico
  - $log^k n = \mathcal{O}(n)$ para qualquer constante k, pois logaritmos crescem muito vagarosamente.

Funções e taxas de crescimento

- As mais comuns:

c	constante
$log\ n$	logarítmica
$log^2\ n$	logarítmica ao quadrado
n	linear
$n\ log\ n$	
n^2	quadrática
n^3	cúbica
2^n	
a^n	exponencial

Funções e taxas de crescimento

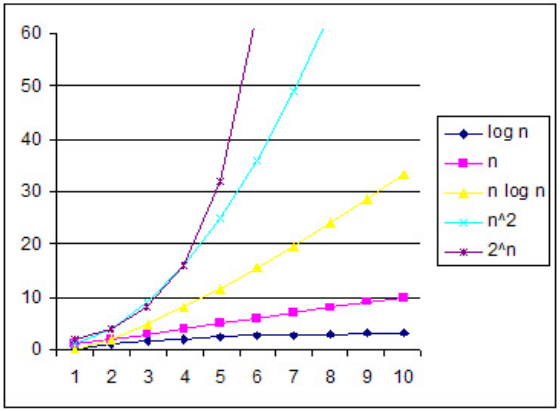


Figure 3: Crescimentos de algumas funções.

Taxas de crescimento

- Apesar de às vezes ser importante, não é comum incluir constantes ou termos de menor ordem em taxas de crescimento:
  - Queremos medir a taxa de crescimento da função, o que torna os "termos menores" irrelevantes,
  - As constantes também dependem do tempo exato de cada operação; como ignoramos os custos reais das operações, ignoramos também as constantes.
- Não se diz que $T(n) = \mathcal{O}(2n^2)$ ou que $T(n) = \mathcal{O}(n^2 + n)$:
  - Diz-se apenas $T(n) = \mathcal{O}(n^2)$.

Exercício

- Um algoritmo tradicional e muito utilizado é da ordem de $n^{1,5}$, enquanto um algoritmo novo proposto recentemente é da ordem de $n \log n$:
  - $f(n) = n^{1,5}$,
  - $g(n) = n \log n$.
- Qual algoritmo você adotaria na empresa que está fundando?
  - Lembre-se que a eficiência desse algoritmo pode determinar o sucesso ou o fracasso de sua empresa!

Exercício

- Uma possível solução:

$$f(n) = n^{1,5} \quad \Rightarrow \quad \frac{n^{1,5}}{n} = n^{0,5} \quad \Rightarrow \quad (n^{0,5})^2 = n$$

$$g(n) = n\ log\ n \quad \Rightarrow \quad \frac{(n\ log\ n)}{n} = log\ n \quad \Rightarrow \quad (log\ n)^2 = log^2 n$$

- Como n cresce mais rapidamente do que qualquer potência de log, temos que o **algoritmo novo** é mais eficiente e, portanto, deve ser o adotado pela empresa no momento.

Análise de algoritmos

- Para proceder a uma análise de algoritmos e determinar as taxas de crescimento, necessitamos de um modelo de computador e das operações que executa.
- Assume-se o uso de um computador tradicional, em que as instruções de um programa são executadas sequencialmente,
  - com memória infinita, por simplicidade.

Análise de algoritmos

- Repertório de instruções simples: soma, multiplicação, comparação, atribuição, etc.
  - Por simplicidade e viabilidade da análise, assume-se que cada instrução demora exatamente uma unidade de tempo para ser executada,
    - Obviamente, em situações reais, isso pode não ser verdade: a leitura de um dado em disco pode demorar mais do que uma soma.
  - Operações complexas, como inversão de matrizes e ordenação de valores, não são realizadas em uma única unidade de tempo, obviamente: devem ser analisadas em partes.

Análise de algoritmos

- Considera-se somente o algoritmo e suas entradas (de tamanho n).
- Para uma entrada de tamanho n, pode-se calcular $T_{melhor}(n)$, $T_{media}(n)$ e $T_{pior}(n)$, ou seja, o **melhor** tempo de execução, o tempo **médio** e o **pior**, respectivamente:
  - Obviamente, $T_{melhor}(n) \leq T_{media}(n) \leq T_{pior}(n)$.
- Atenção: para mais de uma entrada, essas funções teriam mais de um argumento.

Análise de algoritmos

- Geralmente, utiliza-se somente a análise do pior caso $T_{pior}(n)$, pois ela fornece os **limites** para todas as entradas, incluindo particularmente as entradas ruins:
  - Logicamente, muitas vezes, **o tempo médio pode ser útil**, principalmente em sistemas executados rotineiramente:
    - Por exemplo: em um sistema de cadastro de alunos como usuários de uma biblioteca, o trabalho difícil de cadastrar uma quantidade enorme de pessoas é feito somente uma vez; depois, cadastros são feitos de vez em quando apenas.
  - Dá mais trabalho calcular o tempo médio,
  - O melhor tempo não tem muita utilidade.

Análise de algoritmos

- Idealmente, para um algoritmo qualquer de ordenação de vetores com n elementos:
  - Qual a configuração do vetor que você imagina que provavelmente resultaria no melhor tempo de execução?
  - E qual resultaria no pior tempo?
- Exemplo:
  - Soma da subsequência máxima:
    - Dada uma sequência de inteiros (possivelmente negativos) a_1, a_2, ..., a_n, encontre o valor da máxima soma de quaisquer números de elementos consecutivos; se todos os inteiros forem negativos, o algoritmo deve retornar 0 como resultado da maior soma,
    - Por exemplo, para a entrada -2, 11, -4, 13, -5 e -2, a resposta é 20 (soma de a_2 a a_4).

Soma da subsequência máxima

- Há muitos algoritmos propostos para resolver esse problema:
  - Alguns são mostrados abaixo juntamente com seus tempos de execução (n é o tamanho da entrada):

algoritmo	1	2	3	4
tempo	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
$n = 10$	0,00103	0,00045	0,00066	0,00034
$n = 100$	0,47015	0,01112	0,00486	0,00063
$n = 1.000$	448,77	1,1233	0,05843	0,00333
$n = 10.000$	ND[2]	111,13	0,68631	0,03042
$n = 100.000$	ND	ND	8,0113	0,29832

[2] Não Disponível.

Soma da subsequência máxima

- Deve-se notar que:
  - Para entradas pequenas, todas as implementações rodam num piscar de olhos:
    - Portanto, se somente entradas pequenas são esperadas, não devemos gastar nosso tempo para projetar melhores algoritmos.
  - Para entradas grandes, o melhor algoritmo é o 4.
  - Os tempos não incluem o tempo requerido para leitura dos dados de entrada:
    - Para o algoritmo 4, o tempo de leitura é provavelmente maior do que o tempo para resolver o problema: característica típica de algoritmos eficientes.

Taxas de crescimento

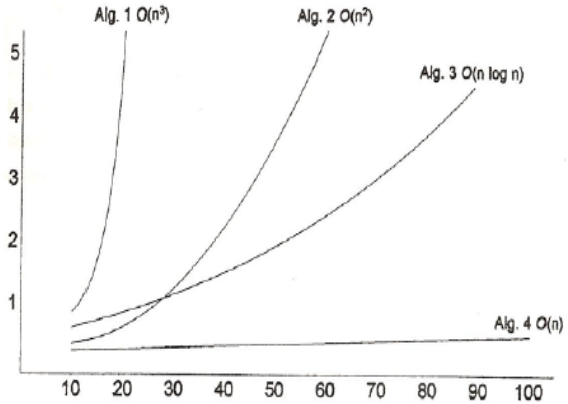


Figure 4: Gráfico (n vs. milisegundos) das taxas de crescimentos dos quatro algoritmos com entradas entre 10 e 100.

Taxas de crescimento

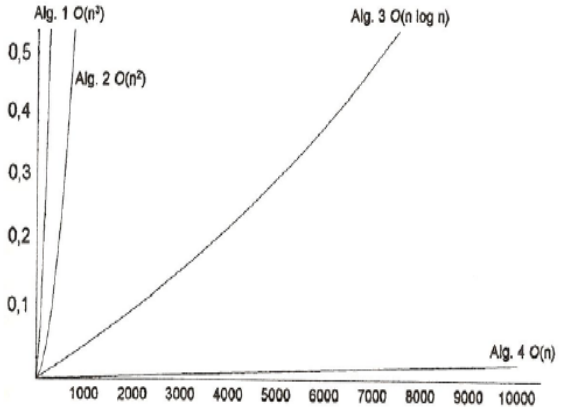


Figure 5: Gráfico (n vs. segundos) das taxas de crescimentos dos quatro algoritmos para entradas maiores.

Sumário

Análise de algoritmos

- Existem basicamente 2 formas de estimar o tempo de execução de programas e decidir quais são os melhores:
  - empiricamente,
  - teoricamente.
- É desejável e possível estimar qual o melhor algoritmo sem ter que executá-los:
  - Função da análise de algoritmos.

Calculando o tempo de execução

- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de

$$\sum_{i=1}^{n} i^3$$

1. Início
2. declare soma_parcial numérico;
3. soma_parcial ← 0;
4. para $i \leftarrow 1$ até n faça
5. soma_parcial ← soma_parcial+i*i*i;
6. escreva(soma_parcial);
7. Fim

Calculando o tempo de execução

$$\sum_{i=1}^{n} i^3$$

3. 1 unidade de tempo
4. 1 unidade para iniciação de i, $n+1$ unidades para testar se $i = n$ e n unidades para incrementar $i = 2n + 2$
5. 4 unidades (1 da soma, 2 das multiplicações e 1 da atribuição) executada n vezes (pelo comando "para") $= 4n$ unidades
6. 1 unidade para escrita

- **Custo total**: somando tudo, tem-se $6n + 4$ unidades de tempo, ou seja, a função é $\mathcal{O}(n)$!

Calculando o tempo de execução

- Ter que realizar todos esses passos para cada algoritmo (principalmente algoritmos grandes) pode se tornar uma tarefa cansativa
- Em geral, como se dá a resposta em termos do *big-oh*, costuma-se desconsiderar as constantes e elementos menores dos cálculos:
  - No exemplo anterior:
    - A linha **3** soma_parcial $\leftarrow 0$ é insignificante em termos de tempo,
    - É desnecessário ficar contando 2, 3 ou 4 unidades de tempo na linha **5** soma_parcial $\leftarrow$ soma_parcial+i*i*i,
    - O que realmente dá a grandeza de tempo desejada é a repetição na linha **4** "para $i \leftarrow 1$ até n faça".

Regras para o cálculo

- Repetições:
  - O tempo de execução de uma repetição é pelo menos o tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada.
- Repetições aninhadas:
  - A análise é feita de dentro para fora,
  - O tempo total de comandos dentro de um grupo de repetições aninhadas é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições.
  - O exemplo abaixo é $\mathcal{O}(n^2)$:

```
para i ← 0 até n faça
 para j ← 0 até n faça
   faça k ← k + 1;
```

Regras para o cálculo

- Comandos consecutivos:
  - É a soma dos tempos de cada um, o que pode significar o máximo entre eles,
  - O exemplo abaixo é $\mathcal{O}(n^2)$, apesar da primeira repetição ser $\mathcal{O}(n)$:

```
para i ← 0 até n faça
  k ← 0;
para i ← 0 até n faça
  para j ← 0 até n faça
    faça k ← k + 1;
```

Regras para o cálculo

- Se... então... senão:
  - Para uma cláusula condicional, o tempo de execução nunca é maior do que o tempo do teste mais o tempo do maior entre os comandos relativos ao então e os comandos relativos ao senão,
  - O exemplo abaixo é $\mathcal{O}(n)$:

```
se i < j
 então i ← i + 1
 senão para k ← 1 até n faça
   i ← i * k;
```

- Chamadas a sub-rotinas:
  - Uma sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/sub-rotina que a chamou.

Regras para o cálculo

- **Exercício**: Estime quantas unidades de tempo são necessárias para rodar o algoritmo abaixo:

```
1  Início
2  declare i e j numéricos;
3  declare A vetor numérico de n posições;
4  i ← 1;
5  enquanto i ≤ n faça
6     A[i] ← 0;
7     i ← i + 1;
8  para i ← 1 até n faça
9   para j ← 1 até n faça
10     A[i] ← A[i] + i + j;
11 Fim
```

Sumário

Exercício

- Analise a sub-rotina recursiva abaixo:

```
1 sub-rotina fatorial(n: numérico)
2 início
3   declare aux numérico;
4   se n = 1
5     então aux ← 1
6     senão aux ← n * fatorial(n − 1);
7   fatorial ← aux;
8 fim
```

Regras para o cálculo

- Sub-rotinas recursivas:
  - Se a recursão é um "disfarce" da repetição (e, portanto, a recursão está mal empregada, em geral), basta analisá-la como tal,
  - O exemplo anterior é obviamente $\mathcal{O}(n)$.
- Eliminando a recursão:

```
1 sub-rotina fatorial(n: numérico)
2 início
3   declare aux numérico;
4   aux ← 1
5   enquanto n > 1 faça
6     aux ← aux * n;
7     n ← n − 1;
8   fatorial ← aux;
9 fim
```

Regras para o cálculo

- Sub-rotinas recursivas:
  - Em muitos casos (incluindo casos em que a recursividade é bem empregada), é difícil transformá-la em repetição:
    - Nesses casos, para fazer a análise do algoritmo, pode ser necessário usar a **análise de recorrência**.
    - **Recorrência**: equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores.
    - Caso típico: algoritmos de **divisão-e-conquista**, ou seja, algoritmos que desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

Regras para o cálculo

- Exemplo de uso de recorrência:
  - Números de Fibonacci:
    - 0,1,1,2,3,5,8,13...
    - $fib(0) = 0$, $fib(1) = 1$, $fib(i) = fib(i-1) + fib(i-2)$.
  - A rotina:

```
1 sub-rotina fib(n: numérico)
2 início
3 declare aux numérico;
4 se n <= 1
5   então aux ← n
6   senão aux ← fib(n − 1) + fib(n − 2);
7 fib ← aux;
8 fim
```

Regras para o cálculo

- Seja $T(n)$ o tempo de execução da função:
  - **Caso 1**: Se $n = 0$ ou $n = 1$, o tempo de execução é constante, que é o tempo de testar o valor de n no comando *se*, mais atribuir o valor 1 à variável *aux*, mais atribuir o valor de *aux* ao nome da função; ou seja, $T(0) = T(1) = 3$.
  - **Caso 2**: Se $n > 2$, o tempo consiste em testar o valor de n no comando *se*, mais o trabalho a ser executado no *senão* (que é uma soma, uma atribuição e 2 chamadas recursivas), mais a atribuição de *aux* ao nome da função; ou seja, a recorrência $T(n) = T(n-1) + T(n-2) + 4$, para $n > 2$.

Regras para o cálculo

- Muitas vezes, a recorrência pode ser resolvida com base na prática e experiência do analista,
- Alguns métodos para resolver recorrências:
  - Método da substituição,
  - Método mestre,
  - Método da árvore de recursão,
  - Método da iteração.

Sumário

Resolução de recorrências

- Método da substituição:
  - Supõe-se (aleatoriamente ou com base na experiência) um limite superior para a função e verifica-se se ela não extrapola este limite:
    - Uso de indução matemática.
  - O nome do método vem da "substituição" da resposta adequada pelo palpite,
  - Pode-se "apertar" o palpite para achar funções mais exatas.

Resolução de recorrências

- Método mestre:
  - Fornece limites para recorrências da forma $T(n) = aT(\frac{n}{b}) + f(n)$, em que $a \geq 1$, $b > 1$ e $f(n)$ é uma função dada,
  - Envolve a memorização de alguns casos básicos que podem ser aplicados para muitas recorrências simples.

Resolução de recorrências

- Método da árvore de recursão:
  - Traça-se uma árvore que, nível a nível, representa as recursões sendo chamadas,
  - Em seguida, em cada nível/nó da árvore, são acumulados os tempos necessários para o processamento:
    - No final, tem-se a estimativa de tempo do problema.
  - Este método pode ser utilizado para se fazer uma suposição mais informada no método da substituição.

Resolução de recorrências

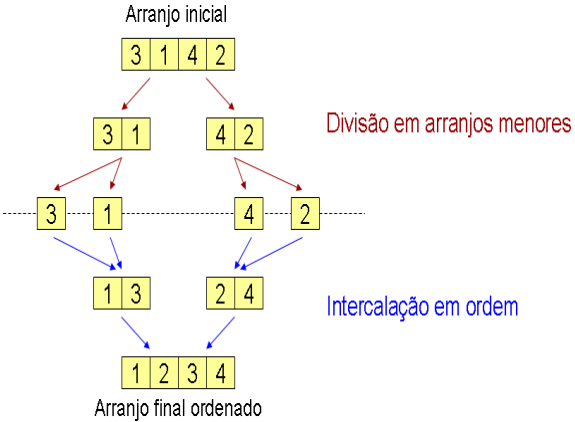
- Método da iteração:
  - Este método repetidamente faz substituições para cada ocorrência da função T do lado direito até que todas as ocorrências desapareçam.

Resolução de recorrências

- Método da árvore de recursão:
  - **Exemplo**: algoritmo de ordenação de arranjos por intercalação:
    - Passo 1: divide-se um arranjo não ordenado em dois subarranjos,
    - Passo 2: se os subarranjos não são unitários, cada subarranjo é submetido ao passo 1 anterior; caso contrário, eles são ordenados por intercalação dos elementos e isso é propagado para os subarranjos anteriores.

Ordenação por intercalação

- Exemplo com arranjo de 4 elementos:



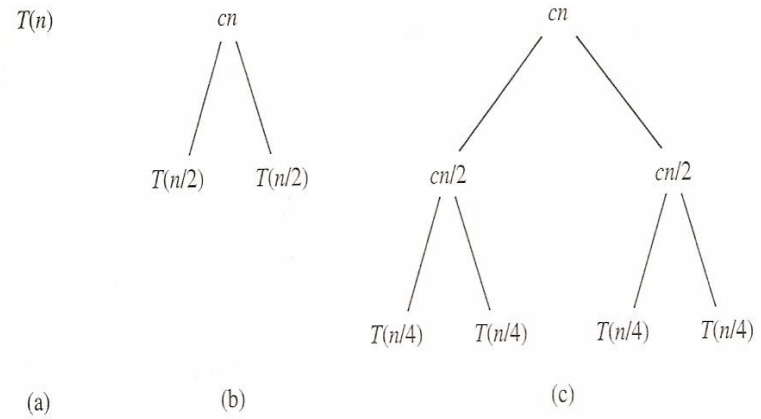
- Implemente a(s) sub-rotina(s) e calcule sua complexidade.

Resolução de recorrências

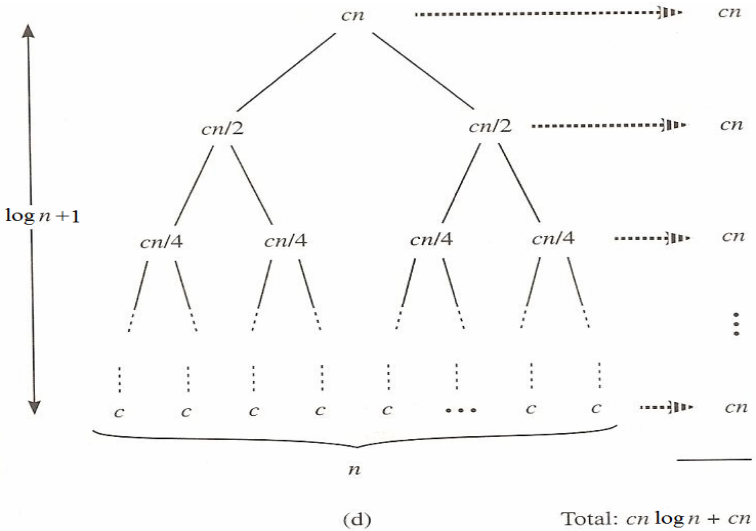
- Método da árvore de recursão:
  - Considere o tempo do algoritmo (que envolve recorrência):

$$T(n) = c, \text{ se } n = 1$$
$$T(n) = 2T(\tfrac{n}{2}) + cn, \text{ se } n > 1$$

Resolução de recorrências



Resolução de recorrências



Resolução de recorrências

- Tem-se que:
  - Na parte (a), há $T(n)$ ainda não expandido,
  - Na parte (b), $T(n)$ foi dividido em árvores equivalentes representando a recorrência com custos divididos ($T(\frac{n}{2})$ cada uma), sendo cn o custo no nível superior da recursão (fora da recursão e, portanto, associado ao nó raiz),
  - ...
  - No fim, nota-se que o tamanho da árvore corresponde a $(\log\ n)+1$, o qual multiplica os valores obtidos em cada nível da árvore, os quais, nesse caso, são iguais:
    - Como resultado, tem-se $cn\ \log\ n + cn$, ou seja, $\mathcal{O}(n\ \log\ n)$.

Resolução de recorrências

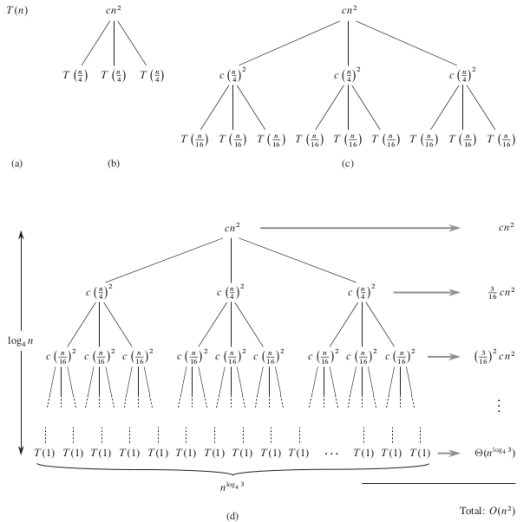
- Alguns dizem que a expressão correta é "$f(n)$ é $\mathcal{O}(g(n))$":
  - Seria considerado redundante e inadequado dizer "$f(n) \leq \mathcal{O}(g(n))$" ou (ainda pior) "$f(n) = \mathcal{O}(g(n))$",
  - Não é incorreto (embora não seja usual) dizer "$f(n) \in \mathcal{O}(g(n))$", já que o operador *Big-oh* representa todo um conjunto de funções.

Sumário

Exemplo 1

- O próximo slide [2] mostra a árvore de recursão para $T(n) = 3T(n/4) + cn^2$:
  - Assume-se que n é uma potência de 4 para que os tamanhos dos sub-problemas sejam inteiros,
  - Parte (a) mostra $T(n)$, que é expandido na parte (b) em uma árvore equivalente representando a recorrência,
  - O termo cn^2 na raiz representa o custo no nível superior da recursão e as três sub-árvores da raiz representam os custos dos sub-problemas de tamanho $n/4$,
  - A parte (c) mostra esse processo um passo a frente expandindo cada nó com custo $T(n/4)$ a partir da parte (b),
  - O custo de cada um dos três sucessores da raiz é $c(n/4)^2$,
  - Continua-se expandindo cada nó da árvore, quebrando-o em suas partes constituintes, como determinado pela recorrência.

Exemplo 1


T(n)
cn²
T(n/4) T(n/4) T(n/4)
(a)
(b)
cn²
c(n/4)² c(n/4)² c(n/4)²
T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16)
(c)
cn²
cn²
c(n/4)² c(n/4)² c(n/4)²
3/16 cn²
log₄ n
c(n/16)² c(n/16)² c(n/16)² c(n/16)² c(n/16)² c(n/16)² c(n/16)² c(n/16)² c(n/16)²
(3/16)² cn²
T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) ⋯ T(1) T(1) T(1)
Θ(n^log₄ 3)
n^log₄ 3
Total: O(n²)
(d)

Exemplo 1

- Como os tamanhos dos sub-problemas descrescem por um fator de 4 cada vez que descemos um nível, deve-se alcançar uma condição limite.
- Quão longe da raiz alcançaremos?
- O tamanho do sub-problema para um nó na profundidade i é $n/4^i$.
- Portanto, o tamanho do sub-problema alcança $n = 1$ quando $n/4^i = 1$, ou equivalentemente, quando $i = \log_4 n$.
- Ou seja, a árvore tem $\log_4 n + 1$ níveis (em profundidades 0, 1, 2, ..., $\log_4 n$).

Exemplo 1

- Depois, determina-se o custo em cada nível da árvore.
- Cada nível tem três vezes mais nós que o nível acima, e portanto o número de nós na profundidade i é 3^i.
- Como os tamanhos dos sub-problemas reduzem por um fator de 4 para cada nível a partir da raiz, cada nó na profundidade i, para $i = 0, 1, 2, ..., \log_4 n - 1$, tem um custo de $c(n/4^i)^2$.
- Multiplicando, o custo total de todos os nós na profundidade i é $3^i c(n/4^i)^2 = (3/16)^i cn^2$.
- O nível do fundo, na profundidade $\log_4 n$, tem $3^{\log_4 n} = n^{\log_4 3}$ nós, cada um com custo $T(1)$, para um custo total de $n^{\log_4 3} T(1)$, que é $\Theta(n^{\log_4 3})$, assumindo $T(1)$ como uma constante.

Exemplo 1

- Custo da árvore inteira:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) = \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})$$

pois

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

Exemplo 1

Quando a soma é infinita e $|x| < 1$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Logo

$$T(n) = \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

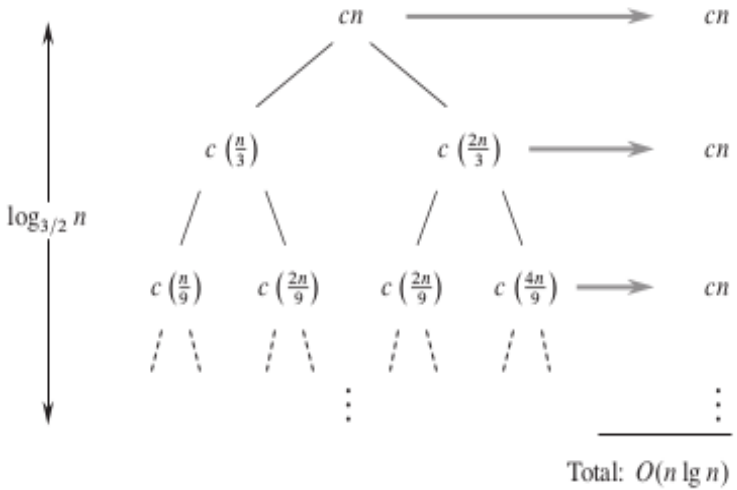
$$= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = \mathcal{O}(n^2).$$

pois $\log_4 3 = 0.79$.

Exemplo 2

- Outro exemplo [2]: árvore de recursão para $T(n) = T(n/3) + T(2n/3) + \mathcal{O}(n)$.
- c representa o fator constante no termo $\mathcal{O}(n)$.
- Quando se adiciona os valores ao longo dos níveis da árvore de recursão, chega-se ao valor de cn para cada nível.
- O caminho mais longo da raiz a uma folha é $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$.
- Como $(2/3)^k n = 1$ quando $k = \log_{3/2} n$, a altura da árvore é $\log_{3/2} n$.

Exemplo 2



Exemplo 2

- Intuitivamente, espera-se que a solução para a recorrência seja no máximo o número de níveis vezes o custo de cada nível, ou $\mathcal{O}(cn \log_{3/2} n) = \mathcal{O}(n \log n)$.
- A figura mostra apenas os níveis superiores da árvore de recursão e no entanto, nem todo nível contribui com um custo de cn.
- Considere o custo das folhas.
- Se a árvore for uma árvore binária completa de altura $\log_{3/2} n$, então haveria $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ folhas.
- Como o custo de cada folha é uma constante, o custo total de todas as folhas seria $\Theta(n^{\log_{3/2} 2}) = \omega(n \log n)$, já que $\log_{3/2} 2$ é uma constante maior que 1.

Exemplo 2

- Esta árvore de recursão não é uma árvore binária completa, portanto ela tem menos de $n^{\log_{3/2} 2}$ folhas.
- Consequentemente, níveis próximos do fundo contribuem menos que cn para o custo total.
- Como estamos preocupados apenas em adivinhar (método da substituição), não precisamos nos preocupar com o custo exato.
- Portanto, nosso "chute" para o limite superior será $\mathcal{O}(n \log n)$.
- Usaremos então o método da substituição para verificar que $\mathcal{O}(n \log n)$ é um limite superior para a solução da recorrência.
- Mostramos que $T(n) \leq d\ n\ \log n$, onde d é uma constante positiva.

Exemplo 2

- Tem-se

$$
\begin{aligned}
T(n) &\leq T(n/3) + T(2n/3) + cn \\
&\leq d(n/3)\log(n/3) + d(2n/3)\log(2n/3) + cn \\
&= (d(n/3)\log n - d(n/3)\log 3) \\
&+(d(2n/3)\log n - d(2n/3)\log(3/2)) + cn \\
&= d\ n\ \log n - d((n/3)\log 3 + (2n/3)\log(3/2)) + cn \\
&= d\ n\ \log n - d((n/3)\log 3 + (2n/3)\log 3 - (2n/3)\log 2) + cn \\
&= d\ n\ \log n - d\ n(\log 3 - 2/3) + cn \\
&\leq d\ n\ \log n
\end{aligned}
$$

- desde que $d \geq c/(\log 3 - (2/3))$.

Problema das 8 rainhas

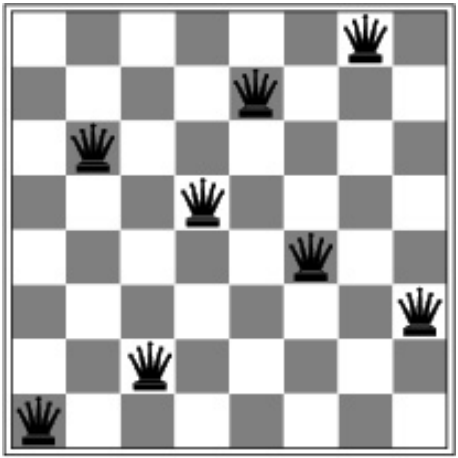
Exercício proposto

Resolver, através da recursão, o **problema das 8 rainhas**. Fazer a análise do algoritmo em termos de *big-Oh* (equação de recorrência). O problema das 8 rainhas consiste em colocar num tabuleiro de xadrez (matriz 8×8), 8 rainhas, uma em cada linha, de tal forma que uma rainha não "coma" outra. Lembre-se de que a rainha é a peça do xadrez que se movimenta qualquer número de casas na vertical, na horizontal e nas diagonais.

Obs.: O problema tem 92 soluções distintas.

Problema das 8 rainhas

- Veja uma "quase" solução:



Problema das 8 rainhas

92 Soluções:

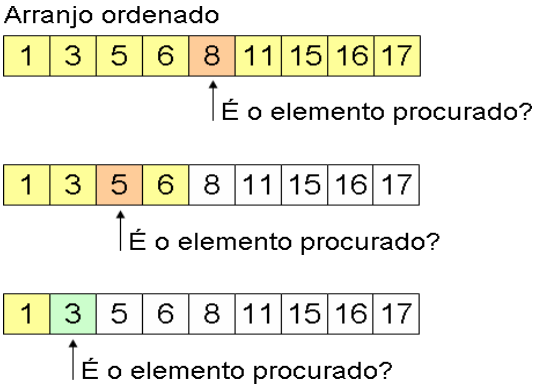
1 5 8 6 3 7 2 4	1 6 8 3 7 4 2 5	1 7 4 6 8 2 5 3	1 7 5 8 2 4 6 3	2 4 6 8 3 1 7 5
2 5 7 1 3 8 6 4	2 5 7 4 1 8 6 3	2 6 1 7 4 8 3 5	2 6 8 3 1 4 7 5	2 7 3 6 8 5 1 4
2 7 5 8 1 4 6 3	2 8 6 1 3 5 7 4	3 1 7 5 8 2 4 6	3 5 2 8 1 7 4 6	3 5 2 8 6 4 7 1
3 5 7 1 4 2 8 6	3 5 8 4 1 7 2 6	3 6 2 5 8 1 7 4	3 6 2 7 1 4 8 5	3 6 2 7 5 1 8 4
3 6 4 1 8 5 7 2	3 6 4 2 8 5 7 1	3 6 8 1 4 7 5 2	3 6 8 1 5 7 2 4	3 6 8 2 4 1 7 5
3 7 2 8 5 1 4 6	3 7 2 8 6 4 1 5	3 8 4 7 1 6 2 5	4 1 5 8 2 7 3 6	4 1 5 8 6 3 7 2
4 2 5 8 6 1 3 7	4 2 7 3 6 8 1 5	4 2 7 3 6 8 5 1	4 2 7 5 1 8 6 3	4 2 8 5 7 1 3 6
4 2 8 6 1 3 5 7	4 6 1 5 2 8 3 7	4 6 8 2 7 1 3 5	4 6 8 3 1 7 5 2	4 7 1 8 5 2 6 3
4 7 3 8 2 5 1 6	4 7 5 2 6 1 3 8	4 7 5 3 1 6 8 2	4 8 1 3 6 2 7 5	4 8 1 5 7 2 6 3
4 8 5 3 1 7 2 6	5 1 4 6 8 2 7 3	5 1 8 4 2 7 3 6	5 1 8 6 3 7 2 4	5 2 4 6 8 3 1 7
5 2 4 7 3 8 6 1	5 2 6 1 7 4 8 3	5 2 8 1 4 7 3 6	5 3 1 6 8 2 4 7	5 3 1 7 2 8 6 4
5 3 8 4 7 1 6 2	5 7 1 3 8 6 4 2	5 7 1 4 2 8 6 3	5 7 2 4 8 1 3 6	5 7 2 6 3 1 4 8
5 7 2 6 3 1 8 4	5 7 4 1 3 8 6 2	5 8 4 1 3 6 2 7	5 8 4 1 7 2 6 3	6 1 5 2 8 3 7 4
6 2 7 1 3 5 8 4	6 2 7 1 4 8 5 3	6 3 1 7 5 8 2 4	6 3 1 8 4 2 7 5	6 3 1 8 5 2 4 7
6 3 5 7 1 4 2 8	6 3 5 8 1 4 2 7	6 3 7 2 4 8 1 5	6 3 7 2 8 5 1 4	6 3 7 4 1 8 2 5
6 4 1 5 8 2 7 3	6 4 2 8 5 7 1 3	6 4 7 1 3 5 2 8	6 4 7 1 8 2 5 3	6 8 2 4 1 7 5 3
7 1 3 8 6 4 2 5	7 2 4 1 8 5 3 6	7 2 6 3 1 4 8 5	7 3 1 6 8 5 2 4	7 3 8 2 5 1 6 4
7 4 2 5 8 1 3 6	7 4 2 8 6 1 3 5	7 5 3 1 6 8 2 4	8 2 4 1 7 5 3 6	8 2 5 3 1 7 4 6
8 3 1 6 2 5 7 4	8 4 1 3 6 2 7 5			

Cuidado!

- A análise assintótica é uma ferramenta fundamental ao projeto, análise ou escolha de um algoritmo específico para uma dada aplicação,
- No entanto, deve-se ter sempre em mente que essa análise "esconde" fatores assintoticamente irrelevantes, mas que em alguns casos podem ser relevantes na prática, particularmente se o problema de interesse se limitar a entradas (relativamente) pequenas:
  - Por exemplo, um algoritmo com tempo de execução da ordem de $10^{100}n$ é $\mathcal{O}(n)$, assintoticamente melhor do que outro com tempo $10\ n\ \log\ n$, o que nos faria, em princípio, preferir o primeiro,
  - No entanto, 10^{100} é o número estimado por alguns astrônomos como um limite superior para a quantidade de átomos existente no universo observável!

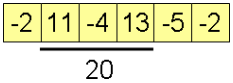
Análise de algoritmos recursivos

- Muitas vezes, temos que resolver recorrências:
  - **Exemplo**: pesquisa binária pelo número 3:



Exercícios propostos

1. Implemente o algoritmo da busca binária em um arranjo ordenado, teste e analise o algoritmo,
2. Faça um algoritmo para resolver o problema da maior soma de subsequência em um arranjo e analise-o:



3. Implemente o algoritmo de Euclides para calcular o máximo divisor comum para 2 números, e faça a análise de recorrência do mesmo,
4. Escreva e analise um algoritmo recursivo para calcular x^n.

Sumário

Estratégia

- Dada uma função para computar n entradas, a estratégia *divisão-e-conquista* separa as entradas em k subconjuntos distintos, $1 < k \leq n$, levando a k subproblemas,
- Estes subproblemas devem ser resolvidos, e então um método deve ser encontrado para combinar subsoluções em uma solução do todo,
- Se os subproblemas ainda forem muito grandes, então a estratégia divisão-e-conquista pode ser reaplicada,
- Os subproblemas resultantes são do **mesmo tipo** do problema original.

Estratégia

- A reaplicação do princípio é naturalmente expressa por um algoritmo recursivo,
- Subproblemas do mesmo tipo cada vez menores são gerados até que finalmente subproblemas que são pequenos o suficiente para serem resolvidos sem a separação são produzidos.

Estratégia

- Considere a estratégia divisão-e-conquista com a separação da entrada em dois subproblemas do mesmo tipo,
- Pode-se escrever uma abstração de controle que espelha a forma como um algoritmo baseado na estratégia parecerá,
- Por *abstração de controle* entende-se um procedimento cujo fluxo de controle é claro mas cujas operações primárias são especificadas por outros procedimentos cujos significados precisos são indefinidos,
- dec é inicialmente chamado como dec(p), onde p é o problema a ser resolvido.

dec(p)

- pequeno(p) é uma função booleana que determina se o tamanho da entrada é pequeno o suficiente para que a resposta possa ser computada sem dividir a entrada,
- Se isso for verdade, a função s é chamada,
- De outra forma, o problema p é dividido em subproblemas menores,
- Estes subproblemas p_1,p_2,...p_k, são resolvidos por aplicações recursivas de dec,
- combine é uma função que determina a solução para p usando as soluções para os k subproblemas,

dec(p)

```
Algoritmo dec(p)
{
 Se pequeno(p) então retorne s(p);
 senão
 {
  divida p em instâncias menores p_1, p_2, ... p_k,  k ≥ 1;
  aplique dec a cada um destes subproblemas;
  retorne combine(dec(p_1), dec(p_2), ..., dec(p_k));
 }
}
```

dec(p)

- Se o tamanho de p é n e os tamanhos dos k subproblemas são n_1, n_2, ..., n_k, respectivamente, então o tempo de computação de dec(p) é descrito pela **relação (equação) de recorrência**:

$$T(n) = \begin{cases} g(n) & n \text{ pequeno} \\ T(n_1) + T(n_2) + ... + T(n_k) + f(n) & n \text{ grande} \end{cases}$$

- onde $T(n)$ é o tempo para dec em qualquer entrada de tamanho n e $g(n)$ é o tempo para computar a resposta para entradas pequenas,
- A função $f(n)$ é o tempo para dividir p e combinar as soluções para os subproblemas.

Recorrências

- Para algoritmos baseados em divisão-e-conquista que produzem subproblemas do mesmo tipo do problema original, é muito natural descrever estes algoritmos usando **recursão**.
- A complexidade de muitos algoritmos divisão-e-conquista é dada por recorrências da forma:

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

- onde a e b são constantes conhecidas. Assume-se que $T(1)$ é conhecido e n é uma potência de b (isto é, $n = b^k$).

Método da iteração

- Um dos métodos para resolver tal relação de recorrência é chamado de *método da iteração*,
- Este método repetidamente faz substituições para cada ocorrência da função T do lado direito até que todas as ocorrências desapareçam,
- **Exemplo**: Considere o caso no qual $a = 2$ e $b = 2$. Seja $T(1) = 2$ e $f(n) = n$. Tem-se:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\dots \end{aligned}$$

Método da iteração

- Em geral, nota-se que $T(n) = 2^i T(n/2^i) + in$, para $i \geq 1$,
- Em particular, então, $T(n) = 2^{\log\ n} T(n/2^{\log\ n}) + n\ \log\ n$, correspondente a escolha de $i = \log\ n$. Portanto, $T(n) = nT(1) + n\ \log\ n = n\ \log\ n + 2n$.

Sumário

Princípio da indução matemática (forte) [4]

- Seja $P(n)$ um predicado que é definido para inteiros n, e sejam a e b inteiros fixos, com $a \leq b$. Suponha que as duas afirmações seguintes sejam verdadeiras:
  1. $P(a)$, $P(a+1)$, ..., $P(b)$ são V (**passo base**)
  2. Para qualquer inteiro $k \geq b$, se $P(i)$ é V para $a \leq i < k$ então $P(k)$ é V.

  Logo, a afirmação "para todos inteiros $n \geq a$, $P(n)$ é V.
- (A suposição que $P(i)$ é V para $a \leq i < k$ é chamada de **hipótese indutiva**.)

Princípio da indução matemática (forte): Exemplo [4]

- Seja a sequência $a_1, a_2, a_3, \ldots$ definida como

$$a_1 = 0$$
$$a_2 = 2$$
$$a_k = 3 \cdot a_{\lfloor k/2 \rfloor} + 2, k \geq 3$$

  Prove que a_n é par, para $n \geq 1$.

- **Prova** (por indução matemática):
  1. Passo base: Para $n = 1$ e $n = 2$ a propriedade é válida já que $a_1 = 0$ e $a_2 = 2$.
  2. Passo indutivo: Vamos supor que a_i é par para todos inteiros i, $0 \leq i < k$ (**hipótese indutiva**)

Princípio da indução matemática (forte): Exemplo [4]

- Se a propriedade é válida para $0 \leq i < k$, então é válida para k, ou seja, a_k é par (*o que deve ser mostrado*).
- Pela definição de $a_1, a_2, a_3, ...$

$$a_k = 3 \cdot a_{\lfloor k/2 \rfloor} + 2, k \geq 3$$

- O termo $a_{\lfloor k/2 \rfloor}$ é par pela hipótese indutiva já que $k > 2$ e $0 \leq \lfloor k/2 \rfloor < k$.
- Desta forma, $3 \cdot a_{\lfloor k/2 \rfloor}$ é par e $3 \cdot a_{\lfloor k/2 \rfloor} + 2$ também é par, o que mostra que a_k é par.

Indução matemática e algoritmos [4]

- É útil para provar asserções sobre a correção e a eficiência de algoritmos,
- Consiste em inferir uma lei geral a partir de instâncias particulares,
- Seja T um teorema que tenha como parâmetro um número natural n,
- Para provar que T é válido para todos os valores de n, prova-se que:
  1. T é válido para $n = 1$;
  2. Para todo $n > 1$, se T é válido para $n - 1$, então T é válido para n.

Indução matemática e algoritmos [4]

- A condição 1 é chamada de **passo base**,
- Provar a condição 2 é geralmente mais fácil que provar o teorema diretamente (pode-se usar a asserção de que T é válido para $n-1$,
- Esta afirmativa é chamada de **hipótese de indução** ou **passo indutivo**,
- As condições 1 e 2 implicam T válido para $n=2$, o que junto com a condição 2 implica T também válido para $n=3$, e assim por diante.

Indução matemática e algoritmos [4]

- $S(n) = 1 + 2 + \cdots + n = n(n+1)/2$:
  - Para $n = 1$ a asserção é verdadeira, pois $S(1) = 1 = 1 \times (1+1)/2$ (**passo base**),
  - Assume-se que a soma dos primeiros n números naturais $S(n)$ é $n(n+1)/2$ (**hipótese de indução**),
  - Pela definição de $S(n)$ sabe-se que $S(n+1) = S(n) + n + 1$,
  - Usando a hipótese de indução, $S(n+1) = n(n+1)/2 + n + 1 = (n+1)(n+2)/2$, que é exatamente o que se quer provar.

Limite superior de equações de recorrência [4, 8]

- A solução de uma equação de recorrência pode ser **difícil** de ser obtida,
- Nestes casos, pode ser mais fácil tentar **advinhar a solução** ou **obter um limite superior** para a ordem de complexidade,
- Advinhar a solução funciona bem quando se está interessado apenas em um limite superior, ao invés da solução exata,
- Mostrar que um certo limite existe é mais fácil do que obter o limite,
- Ex.: $T(2n) \leq 2T(n) + 2n - 1$, $T(2) = 1$, definida para valores de n que são potências de 2:
  - O objetivo é encontrar um limite superior na notação $\mathcal{O}$, onde o lado direito da desigualdade representa o pior caso.

Indução matemática para resolver equação de recorrência [8, 4]

- $T(2n) \leq 2T(n) + 2n - 1,\ T(2) = 1$, definida para valores de n que são potências de 2:
  - Procura-se $f(n)$ tal que $T(n) = \mathcal{O}(f(n))$, mas fazendo com que $f(n)$ seja o mais próximo possível da solução real para $T(n)$ (**limite assintótico firme**),
  - Considera-se o palpite $f(n) = n^2$,
  - Quer-se provar que $T(n) \leq f(n) = \mathcal{O}(f(n))$ utilizando **indução matemática** em n:
    - **Passo base**: $T(2) = 1 \leq f(2) = 4$, portanto verdadeiro.
    - **Passo de indução**: se a recorrência é verdadeira para n então deve ser verdadeira para $2n$, i.e., $T(n) \rightarrow T(2n)$ (lembre-se de que n é uma potência de 2; consequentemente o "número seguinte" a n é $2n$).

Indução matemática para resolver equação de recorrência [8, 4]

- Reescrevendo o **passo de indução** temos:

$$\text{Predicado}(n) \rightarrow \text{Predicado}(2n) \; (T(n) \leq f(n)) \rightarrow (T(2n) \leq f(2n))$$

$$\begin{array}{rcll} T(2n) & \leq & 2T(n) + 2n - 1, & \text{(def. da recorrência)} \\ & \leq & 2n^2 + 2n - 1, & \text{(hipótese de indução,} \\ & & & \text{pode-se substituir } T(n)) \\ & \leq & 2n^2 + 2n - 1 <^{?} (2n)^2 & \text{(a conclusão é verdadeira?)} \\ & \leq & 2n^2 + 2n - 1 < 4n^2 & \textbf{(sim!)} \end{array}$$

que é exatamente o que se quer provar. Logo, $T(n) = \mathcal{O}(n^2)$.

Indução matemática para resolver equação de recorrência [8, 4]

- Vai-se tentar um palpite menor, $f(n) = cn$, para alguma constante c,
- Quer-se provar que $T(n) \leq f(n) = cn = \mathcal{O}(f(n))$ utilizando **indução matemática** em n.
  - **Passo base**: $T(2) = 1 \leq f(2) = 2c$, portanto verdadeiro.
  - **Passo de indução**: se a recorrência é verdadeira para n então deve ser verdadeira para $2n$, i.e., $T(n) \rightarrow T(2n)$.

Indução matemática para resolver equação de recorrência [8, 4]

- Reescrevendo o **passo de indução** temos:

$$\text{Predicado}(n) \rightarrow \text{Predicado}(2n)\ (T(n) \leq f(n)) \rightarrow (T(2n) \leq f(2n))$$
$$(T(n) \leq cn)) \rightarrow (T(2n) \leq 2cn)$$

$$\begin{array}{rcll} T(2n) & \leq & 2T(n) + 2n - 1, & \text{(def. da recorrência)} \\ & \leq & 2cn + 2n - 1, & \text{(hipótese de indução)} \\ & \leq & 2cn + (2n - 1) & \\ & \leq & 2cn + (2n - 1) > 2cn & \text{(a conclusão } (T(2n) \leq 2cn) \\ & & & \textbf{não} \text{ é válida)} \end{array}$$

- Conclusão:
  - cn cresce mais lentamente que $T(n)$,
  - $T(n)$ está entre cn e n^2 e $T(n) \nleq f(n) = cn$.

Indução matemática para resolver equação de recorrência [8, 4]

- Vai-se então tentar $f(n) = n \log n$, uma função entre n e n^2.
- Quer-se provar que $T(n) \leq f(n) = n \log n = \mathcal{O}(f(n))$ utilizando **indução matemática** em n.
  - **Passo base**: $T(2) = 1 \leq f(2) = 2 \log 2$, portanto verdadeiro.
  - **Passo de indução**: se a recorrência é verdadeira para n então deve ser verdadeira para $2n$, i.e., $T(n) \rightarrow T(2n)$.

Indução matemática para resolver equação de recorrência [8, 4]

- Reescrevendo o **passo de indução** temos:

$$\text{Predicado}(n) \rightarrow \text{Predicado}(2n)\ (T(n) \leq f(n)) \rightarrow (T(2n) \leq f(2n))$$
$$(T(n) \leq n\ log\ n)) \rightarrow (T(2n) \leq 2n\ log\ 2n)$$

$$\begin{array}{rcll} T(2n) & \leq & 2T(n) + 2n - 1, & \text{(def. da recorrência)} \\ & \leq & 2n\ log\ n + 2n - 1, & \text{(hipótese de indução)} \\ & \leq & 2n\ log\ n + (2n-1) <^{?} 2n\ log\ 2n & \text{(a conclusão} \\ & & & \text{é verdadeira?)} \\ & \leq & 2n\ log\ n + (2n-1) < 2n\ log\ n + 2n & \text{(sim!)} \end{array}$$

- Conclusão:
  - A diferença entre as fórmulas agora é de apenas 1,
  - De fato, $T(n) = n\ log\ n - n + 1$ é a solução exata de $T(n) = 2T(n/2) + n - 1$, $T(1) = 0$, que descreve o comportamento do algoritmo de ordenação *mergesort*.

Outro exemplo

- Seja $T(n) = T(n-1) + t_2$.
- Pode-se escrever $T(n-1) = T(n-1-1) + t_2$, desde que $n > 1$.
- Como $T(n-1)$ aparece no lado direito da primeira equação, pode-se substituir o lado direito inteiro da última equação,
- Repetindo o processo, chega-se a:

$$\begin{aligned} T(n) &= T(n-1) + t_2 \\ &= (T(n-2) + t_2) + t_2 \\ &= T(n-2) + 2t_2 \\ &= (T(n-3) + t_2) + 2t_2 \\ &= T(n-3) + 3t_2 \\ &\ldots \end{aligned}$$

Indução

- O próximo passo requer certa intuição. Pode-se tentar obter o padrão emergente. Neste caso, é óbvio: $T(n) = T(n-k) + kt_2$, onde $1 \leq k \leq n$.
- Se houver dúvidas sobre nossa intuição, sempre pode-se provar por indução:
  - **Caso Base**: Para $k = 1$, a fórmula é correta: $T(n) = T(n-1) + t_2$.
  - **Hipótese Indutiva**: Assuma que $T(n) = T(n-k) + kt_2$ para $k = 1, 2, \ldots$. Assim, $T(n) = T(n-l) + lt_2$.
- Note que usando a relação de recorrência original pode-se escrever $T(n-l) = T(n-l-1) + t_2$, para $l \leq n$.

Indução

- Logo:

$$\begin{aligned} T(n) &= T(n-l-1)+t_2+lt_2 \\ &= T(n-(l+1))+(l+1)t_2 \end{aligned}$$

- Portanto, por indução em l, a fórmula está correta para todo $0 \leq k \leq n$.
- Portanto, mostrou-se que $T(n) = T(n-k)+kt_2$, para $1 \leq k \leq n$. Agora, se n é conhecido, pode-se repetir o processo até que se tenha $T(0)$ do lado direito.
- O fato de que n é desconhecido não deve ser impeditivo: consegue-se $T(0)$ do lado direito quando $n-k=0$. Isto é, $k=n$. Fazendo $k=n$ tem-se

$$\begin{aligned} T(n) &= T(n-k)+kt_2 \\ &= T(0)+nt_2 \\ &= t_1+nt_2 \end{aligned}$$

Sumário

Big-Oh

- Como as relações de recorrência podem ser usadas para ajudar a determinar o tempo de execução (big-Oh) de funções recursivas,
- Uma função com, características similares: qual é a complexidade assintótica da função FacaCoisa mostrada abaixo? Por que?
- Assuma que a função Combine roda no tempo $\mathcal{O}(n)$ quando $|left - right| = n$, i.e., quando Combine é usada para combinar n elementos no vetor a.

Big-Oh

```
void FacaCoisa(int a[], int left, int right)
// póscondição:  a[left] <= ...  <= a[right]
{
 int mid = (left+right)/2;
 if (left < right)
 {
  FacaCoisa(a,left,mid);
  FacaCoisa(a,mid+1,right);
  Combine(a,left,mid,right);
 }
}
```

- Esta função é uma implementação do algoritmo de ordenação *merge sort*.
- A complexidade do *merge sort* é $\mathcal{O}(n \log n)$ para um vetor de n elementos.

A relação de recorrência

- Seja $T(n)$ o tempo para FacaCoisa executar em um vetor de n elementos, i.e., quando $|left - right| = n$.
- Veja que o tempo para executar um vetor de um elemento é $\mathcal{O}(1)$, tempo constante.
- Tem-se então o seguinte relacionamento:

$$T(n) = \begin{cases} 2T(n/2) + \mathcal{O}(n) & \text{o } \mathcal{O}(n) \text{ é para Combine} \\ \mathcal{O}(1) \end{cases}$$

- Este relacionamento é chamado de *relação de recorrência* porque a função $T(...)$ ocorre em ambos os lados de "=".
- Esta relação de recorrência descreve completamente descreve a função FacaCoisa, tal que se se resolver a relação de recorrência pode-se saber a complexidade de FacaCoisa já que $T(n)$ é o tempo para executar FacaCoisa.

Caso base

- Quando se escreve uma relação de recorrência, deve-se escrever duas equações: uma para o caso geral e uma para o caso base.
- As equações referem-se à função recursiva a qual a recorrência se aplica.
- O caso base é normalmente uma operação $\mathcal{O}(1)$, apesar de poder ser diferente.
- Em algumas relações de recorrência o caso base envolve entrada de tamanho um, tal que escreve-se $T(1) = \mathcal{O}(1)$.
- Entretanto há casos em que o caso base tem tamanho zero.
- Em tais casos, a base poderia ser $T(0) = \mathcal{O}(1)$.

Resolvendo relações de recorrência

- Pode-se realmente resolver a relação de recorrência dada no slide anterior:
  - Escreve-se n em vez de $\mathcal{O}(n)$ na primeira linha para simplificar.

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2[2T(n/4) + n/2] + n \\
&= 4T(n/4) + 2n \\
&= 4[2T(n/8) + n/4] + 2n \\
&= 8T(n/8) + 3n \\
&\ldots \\
&= 2^k T(n/2^k) + kn
\end{aligned}
$$

Resolvendo relações de recorrência

- Note que a última linha é derivada observando um padrão – esta é a "sacada" – generalização de padrões matemáticos como parte do problema.
- Sabe-se que $T(1) = 1$ e esta é uma forma de terminar a derivação. Na verdade, deseja-se que $T(1)$ apareça do lado direito do sinal de "=".
- Isto significa que se quer $n/2^k = 1$ ou $n = 2^k$ ou $\log\ n = k$.
- Continuando com a derivação anterior, tem-se o seguinte já que $k = \log\ n$

$$\begin{aligned} &= 2^k T(n/2^k) + kn \\ &= 2^{\log\ n} T(1) + (\log\ n)n \\ &= n + n\ \log\ n \qquad \text{[lembre-se que } T(1) = 1\text{]} \\ &= \mathcal{O}(n\ \log\ n) \end{aligned}$$

Resolvendo relações de recorrência

- Resolveu-se a relação de recorrência e sua solução é o que se esperava.
- Para tornar isto uma prova formal, seria necessário usar indução para mostrar que $\mathcal{O}(n \log n)$ é a solução para a dada relação de recorrência,
- Mas o método "rápido" mostrado acima mostra como derivar a solução,
- A verificação subsequente que esta é a solução é deixado para algoritmos mais avançados.

Resolvendo relações de recorrência: RESUMO

- Vimos quatro métodos para resolução de recorrência:
  1. **Método da Substituição**: 1. Adivinhe a solução; 2. Use indução para achar as constantes e mostrar que a solução funciona.
  2. **Árvore de Recursão**: 1. Use para gerar um palpite (*chute*); 2. Verifique pelo método da substituição.
  3. **Método Mestre**: Usado por muitas recorrências divisão-e-conquista da forma $T(n) = aT(n/b) + f(n)$, onde $a \geq 1$, $b > 1$ e $f(n) > 0$. Baseado no **Teorema Mestre**.
  4. **Método da Iteração**: Aplica-se linha a linha as substituições para valores posteriores da entrada, até atingir uma forma geral, da qual pode ser obtida uma expressão que contenha o caso base.

Teorema Mestre

- **Teorema Mestre** [2]: Sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida no domínio dos números inteiros não negativos pela recorrência

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

Então $T(n)$ tem os seguintes limites assintóticos:

1. Se $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$ ($f(n)$ é polinomialmente menor que $n^{\log_b a}$. Intuitivamente, o custo é dominado pelas folhas);
2. Se $f(n) = \Theta(n^{\log_b a} \log^k n)$, onde $k \geq 0$, então $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$ ($f(n)$ está dentro de um fator polilog de $n^{\log_b a}$ mas não menor. Intuitivamente, o custo é $n^{\log_b a} \log^k n$ em cada nível e há $\Theta(\log n)$ níveis. **Caso simples**: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$);
3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguma constante $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todos os n suficientemente grandes, então $T(n) = \Theta(f(n))$ ($f(n)$ é polinomialmente maior que $n^{\log_b a}$. Intuitivamente, o custo é dominado pela raiz).

Método Mestre - Exemplo 1

- Seja a seguinte relação de recorrência:

$$T(n) = 9T(\frac{n}{3}) + n$$

- Resolução utilizando o método Mestre:
  - $a = 9$
  - $b = 3$
  - $f(n) = n$
  - $n^{\log_3 9} = n^2$
  - $f(n) = \mathcal{O}(n^{\log_3 9 - \epsilon})$, caso **1** quando $\epsilon = 1$
  - Logo, $\Theta(n^{\log_3 9}) = \Theta(n^2)$

Método Mestre - Exemplo 2

- Seja a seguinte relação de recorrência:

$$T(n) = 2\,T(\frac{n}{3}) + 3n + 1$$

- Resolução utilizando o método Mestre:
  - $a = 2$
  - $b = 3$
  - $f(n) = 3n + 1$
  - $n^{\log_3 2}$
  - $f(n) = \Omega(n^{\log_3 2+\epsilon})$, caso **3** quando $\epsilon = 0.37$, pois $\log_3 2 = 0.63$ e $af(n/b) \leq cf(n) \Rightarrow 2f(n/3) \leq cf(n) \Rightarrow 2(\frac{3n}{3}+1) \leq c(3n+1) \Rightarrow 2n+2 \leq c(3n+1)$, para, no limite, $c = \frac{2n+2}{3n+1}$, que será sempre menor que 1 para n grande.
  - Logo, $\Theta(f(n)) = \Theta(n)$

Referências I

[1] Astrachan, O. L.
Big-Oh for Recursive Functions: Recurrence Relations.
http://www.cs.duke.edu/~ola/ap/recurrence.html

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Introduction to Algorithms.
Third edition, The MIT Press, 2009.

[3] Horowitz, E., Sahni, S. Rajasekaran, S.
Computer Algorithms.
Computer Science Press, 1998.

[4] Loureiro, A. A. F.
Paradigmas de Projeto de Algoritmos.
UFMG/ICEx/DCC:
http://www.decom.ufop.br/menotti/paa101/slides/aula-Paradigma.pdf

Referências II

[5] Pardo, T. A. S.
Análise de Algoritmos. SCE-181 Introdução à Ciência da Computação II. *Slides*. Ciência de Computação. ICMC/USP, 2008.

[6] Preiss, B. R.
Data Structures and Algorithms with Object-Oriented Design Patterns in C++. 1999.
http://www.brpreiss.com/books/opus4/html/page41.html\
#SECTION003151000000000000000

[7] Rosa, J. L. G.
Análise de Algoritmos - parte 2 e Divisão e Conquista. SCC-201 Introdução à Ciência da Computação II (capítulo 3).
Slides. Ciência de Computação. ICMC/USP, 2009.

Referências III

[8] Ziviani, N.
Projeto de Algoritmos - com implementações em Java e C++.
Thomson, 2007.