UFABC

Complexidade de Algoritmos

O que é um algoritmo?

- Sequência bem definida e finita de cálculos que, para um dado valor de entrada, retorna uma saída desejada/esperada.

- Na computação:
  - Uma descrição de como resolver um problema proposto.

UFABC

Exemplos de Algoritmos

- Algoritmo de Euclides:
  - Definição: o maior divisor comum entre dois ou mais números é chamado de máximo divisor comum (mdc).
  - Problema: dados dois números inteiros maiores que zero, encontrar o maior denominador comum entre eles.
  - Entrada: dois números inteiros a, b > 0.

UFABC

Exemplos de Algoritmos

Algoritmo de Euclides		
1	Entrada: a, b	
2	Enquanto a ≠ b faça	
3	Se a > b então	
4	a = a – b	
5	Senão	
6	b = b – a	
7	Saída: a	

UFABC

Exemplos de Algoritmos

-
- Ordenação de Vetores por Inserção:
  - **Definição:** um vetor $V \in \mathbb{R}^n$ é crescente se $V_1 \leq \ldots \leq V_n$.
  - **Problema:** dados um vetor de tamanho n, rearranjar seus elementos de tal forma que ele se torne crescente.
  - **Entrada:** vetor V de tamanho *n*.
  - **Saída:** vetor V crescente de tamanho n.

UFABC

Exemplos de Algoritmos

Ordenação de Vetores por Inserção		
1	Entrada: V	
2	Para j=2 até n faça	
3	valor_atual = j	
4	i = j-1	
5	Enquanto i ≥ 1 e V_i > valor_atual faça	
6	$V_{i+1} = V_i$	
7	i = i – 1	
8	V_{i+1} = valor_atual	
9	Saída: V	

UFABC

Instância de um Problema

- Instância de um problema é um conjunto qualquer de valores que é uma entrada válida para esse:
  - No algoritmo de Euclides temos que a=124 e b=245 é uma instância do problema.
  - No algoritmo de ordenação uma instância do problema seria: V = [12, 32, 4, 25, 21, 44].

UFABC

Análise de Algoritmo

- Ao analisar um algoritmo 3 aspectos são de grande importância:
  - **Finitude:** o algoritmo é composto de passos finitos?
  - **Corretude:** o algoritmo resolve o problema proposto?
  - **Complexidade:** quantas instruções são necessárias para obter a solução de uma instância do problema?

UFABC

Análise de Algoritmo

- Finitude: o algoritmo é composto de passos finitos?
- Corretude: o algoritmo resolve o problema proposto?
- Complexidade: quantas instruções são necessárias para obter a solução de uma instância do problema?

UFABC

Finitude do Algoritmo

- O algoritmo de Euclides sempre tem fim?

- Uma vez que é garantido realizar a operação de subtração do maior número (*a)* pelo menor número (*b)*, o resultado será um valor *x*, tal que $a > x \geq 0$ e que substituirá o valor em *a*.

UFABC

Finitude do Algoritmo

- O algoritmo de Euclides sempre tem fim?

- Dado também que *a* e *b* são inteiros, e o maio valor entre esses dois diminui a cada iteração, eventualmente, em sucessivas subtrações, um deles atingirá o valor de 0.

UFABC

Finitude do Algoritmo

- O algoritmo de Euclides sempre tem fim?

- Porém, para resultar em 0 os valores de *a* e *b* na iteração anterior devem ser iguais, logo definindo o critério de parada como verdadeiro.

UFABC

Finitude do Algoritmo

- O algoritmo de ordenação por inserção sempre tem fim?
- O laço principal desse algoritmo é bem definido, repetindo a iteração n-1 vezes.

UFABC

Finitude do Algoritmo

- O algoritmo de ordenação por inserção sempre tem fim?
- O laço interno inicia sua operação com i = j-1 ≥ 1 e decrementa seu valor a cada iteração, eventualmente chegando ao valor i=0, saindo do laço.

UFABC

Análise de Algoritmo

- ~~Finitude: o algoritmo é composto de passos finitos?~~
- Corretude: o algoritmo resolve o problema proposto?
- Complexidade: quantas instruções são necessárias para obter a solução de uma instância do problema?

Corretude de um Algoritmo

- Um algoritmo é dito correto se:
  - Para qualquer instância do problema ele retorna a solução correta em tempo finito.
- Algoritmos incorretos também podem ser úteis.

UFABC

Corretude de um Algoritmo

- O algoritmo de Euclides está correto?
  - Dado que o mdc(a,b) = mdc(a-x,b) = mdc(a,b-y), com x < a e y < b então temos que:
    - mdc(a,b) = mdc(a-b,b), com a > b
  - Com essa propriedade do mdc garantimos que o resultado final não é alterado durante as sucessivas subtrações.

Corretude de um Algoritmo

- O algoritmo de Euclides está correto?
  - Dado que o mdc(a,a) = a, garantimos que a solução final, quando os dois valores são iguais, está correto.

Corretude de um Algoritmo

- O algoritmo de ordenação por inserção está correto?
- Para provar que sim utilizaremos uma invariante de um laço e técnica de indução matemática.

UFABC

Invariante de um Laço

- Invariante de um laço é um estado ou condição de uma variável que é invariante durante a execução das iterações.

- Com essa invariante é possível mostrar a corretude do algoritmo ao longo dos laços contidos nele.

Corretude de um Algoritmo

- O algoritmo de ordenação por inserção está correto?
- Para o laço principal do algoritmo podemos supor que a invariante é que o vetor V está ordenado para os índices de 1..j-1

UFABC

Corretude de um Algoritmo

- O algoritmo de ordenação por inserção está correto?
- Utilizando indução matemática precisamos primeiro mostrar que na primeira iteração (j=2) o subvetor V(1..j-1) está ordenado.
- Como esse subvetor é composto por apenas um elemento, então ele está ordenado.

Corretude de um Algoritmo

- O algoritmo de ordenação por inserção está correto?

- Agora, dado que o subvetor está ordenado na iteração em que j=k, precisamos mostrar que ele estará ordenado na iteração j=k+1.

Corretude de um Algoritmo

- O algoritmo de ordenação por inserção está correto?

- Em suma, o laço interno guarda o elemento j=k+1 e “empurra” os outros elementos em uma posição adiante até encontrar a posição correta para esse elemento, tendo ao final um subvetor ordenado V(1..k+1).

UFABC

Análise de Algoritmo

- ~~Finitude: o algoritmo é composto de passos finitos?~~
- ~~Corretude: o algoritmo resolve o problema proposto?~~
- Complexidade: quantas instruções são necessárias para obter a solução de uma instância do problema?

Complexidade de um Algoritmo

- Nos dias atuais, com o avanço crescente da capacidade de computação, o uso do algoritmo mais rápido ainda é importante?

- Suponha que temos dois computadores disponíveis:
  - computador C1 capaz de realizar 10^9 operações por segundo e;
  - computador C2 capaz de realizar 10^6 operações por segundo.

UFABC

Complexidade de um Algoritmo

- Suponha também que temos dois algoritmos para realizar uma operação:
  - algoritmo A1, que necessita executar n^2 operações e;
  - algoritmo A2, que necessita executar n.log(n) operações.

UFABC

Complexidade de um Algoritmo

- Executando A1 em C1 e A2 em C2 e sendo $n=10^6$, temos:
  - A1 em C1:
    - $(10^6)^2$ instruções / 10^9 instruções por seg. ≈ 10^3 segs.
  - A2 em C2:
    - $10^6.\log(10^6)$ instruções / 10^6 instruções por seg. ≈ 6 segs.
- A2 foi cerca de 160x mais rápido que A1, mesmo em uma máquina 1000x mais lenta.

UFABC

Complexidade de um Algoritmo

- Executando A1 em C1 e A2 em C2 e sendo $n=10^6$, temos:
  - A1 em C1:
    - $(10^6)^2$ instruções / 10^9 instruções por seg. ≈ 10^3 segs.
  - A2 em C2:
    - $10^6.\log(10^6)$ instruções / 10^6 instruções por seg. ≈ 6 segs.
- Se o tamanho do problema aumentar em 10x, A1 levará mais ou menos 1 dia enquanto A2 levará 70 segundos!!!

UFABC

Medida de Complexidade de Algoritmo

- A complexidade de tempo de um algoritmo, ou eficiência, é mensurada como o número de instruções básicas que ele executa em função do tamanho da entrada.

- Geralmente essa medida é efetuada levando em conta o pior caso do algoritmo: a instância daquele tamanho que leva o maior tempo.

UFABC

Medida de Complexidade de Algoritmo

- De forma a determinar a complexidade do algoritmo nas situações que realmente interessam, é feita uma análise assintótica da complexidade.

- Análise assintótica: análise feita em instâncias MUITO GRANDES do problema.

UFABC

Algoritmos Eficientes

- O algoritmo é dito eficiente se a complexidade dele é polinomial em função do tamanho da entrada:
  - n, n^2, n^3, n^4,…
- Eles são considerados eficientes pois são funções bem comportadas e não crescem tão rapidamente conforme o tamanho do problema.

Complexidade

- Para determinar a complexidade do algoritmo é necessário verificar a contagem de tempo das operações elementares do algoritmo em função do tamanho da entrada.

- O tamanho da entrada para o algoritmo de ordenação pode ser considerado o tamanho do vetor a ser ordenado.

- Para o algoritmo de Euclides o tamanho da entrada pode ser considerado o número de bits para representar cada um dos dois números que este recebe de entrada.

UFABC

Contando o tempo no problema de Ordenação

Ordenação de Vetores por Inserção		
		# execuções
1	Entrada: V	
2	Para j=2 até n faça	n
3	valor_atual = j	n-1
4	i = j-1	n-1
5	Enquanto i ≥ 1 e V_i > valor_atual faça	
6	$V_{i+1} = V_i$	
7	i = i – 1	
8	V_{i+1} = valor_atual	n-1
9	Saída: V	

UFABC

c_i é o número de repetições do laço interno para aquele valor de i

Contando o tempo no problema de Ordenação

- O número total de operações é então a somatória de cada item da coluna "# execução" da tabela:

$$4*n - 3 + 3*\sum_{i=2}^{n} c_i - 2*(n-1) = 2*n - 1 + 3*\sum_{i=2}^{n} c_i$$

  - O número de operações depende de c_i que, por sua vez, depende do estado inicial do vetor de entrada.

UFABC

Melhor Caso

- No melhor caso, temos como entrada um vetor já ordenado e, portanto, a condição do laço interno de que V_i > valor_atual sempre será falsa.

- Com isso, teremos que $c_i = 1$ para todo i=2..n, e o número de operações será:

$$2*n - 1 + 3*\sum_{i=2}^{n} 1 = 2*n - 1 + 3*(n-1) = 5*n - 4$$

- Portanto, no melhor caso, a execução do algoritmo é uma função linear no tamanho de entrada.

UFABC

Pior Caso

- No pior caso, temos como entrada um vetor com valores em ordem decrescente e, portanto, no laço interno será necessário percorrer até o início do vetor.

- Com isso, teremos que c_i = j para todo i=2..n, e o número de operações será:

$$2*n - 1 + 3*\sum_{i=2}^{n} j = 2*n - 1 + 3*(n(n+1)/2 - 1) = (3*n^2 + 7*n - 8)/2$$

- Portanto, no pior caso, a execução do algoritmo é uma função quadrática no tamanho de entrada.

Complexidade Assintótica

- Geralmente a análise de complexidade de um algoritmo é feita para o pior caso, embora não seja incomum fazer TAMBÉM uma análise de melhor caso.

- Além disso, na maioria dos casos interessa fazer uma análise de comportamento assintótico para os algoritmos (análise com instâncias MUITO grandes).

- Esse estudo tem a vantagem de simplificar a notação de complexidade utilizada para medir o desempenho do algoritmo.

UFABC

Complexidade Assintótica

- Levando em conta a complexidade do algoritmo de ordenação, $(3*n^2 + 7*n -8)/2$, vamos comparar essa complexidade com função $3*n^2$ para diversos valores de n:

n	$3*n^2 + 7*n - 8$	$3*n^2$
64	12.728	12.288
128	50.040	49.152
256	198.392	196.608
1024	3.152.888	3.145.728
2048	12.597.240	12.582.912
4096	50.360.312	50.331.648
8192	201.383.928	201.326.592

UFABC

Complexidade Assintótica

- Podemos perceber que a importância do termo linear da função diminui conforme n aumenta. Então, em uma análise assintótica em que n tende a infinito, podemos nos concentrar apenas no termo dominante da função de complexidade.

n	$3*n^2 + 7*n - 8$	$3*n^2$
64	12.728	12.288
128	50.040	49.152
256	198.392	196.608
1024	3.152.888	3.145.728
2048	12.597.240	12.582.912
4096	50.360.312	50.331.648
8192	201.383.928	201.326.592

UFABC

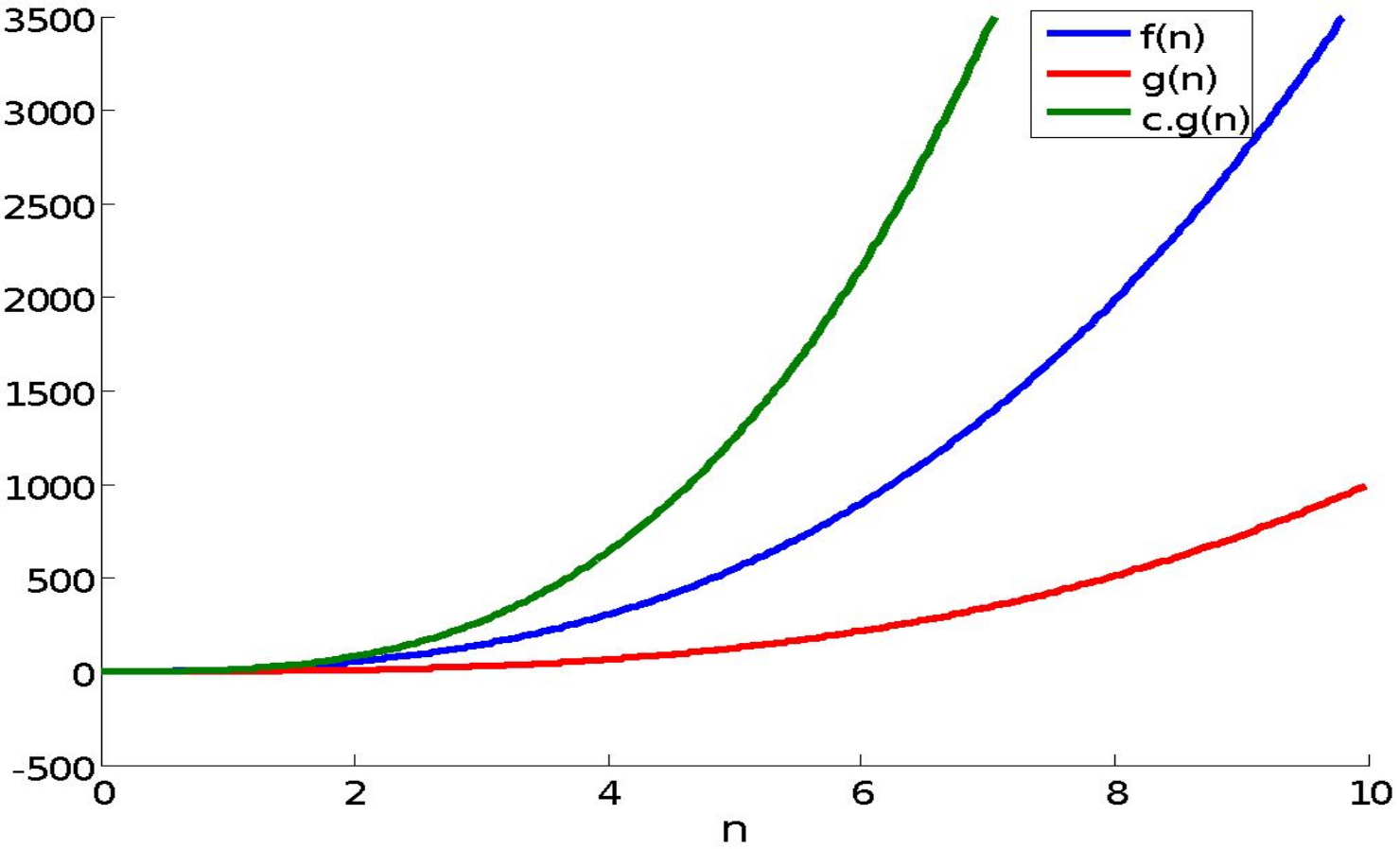
Notações Assintóticas: classe O

- Define o limitante superior da complexidade assintótica do algoritmo.

- Para uma função f(n) que define a complexidade de um algoritmo, ele é O(g(n)) se $0 \leq f(n) \leq c.g(n)$ para algum c e para todo $n \geq n_0$.

- Diz-se que se f(n) tem complexidade O(g(n)) então ele cresce no máximo tão rapidamente quanto g(n).

Notações Assintóticas: classe O



Notações Assintóticas: classe Ω

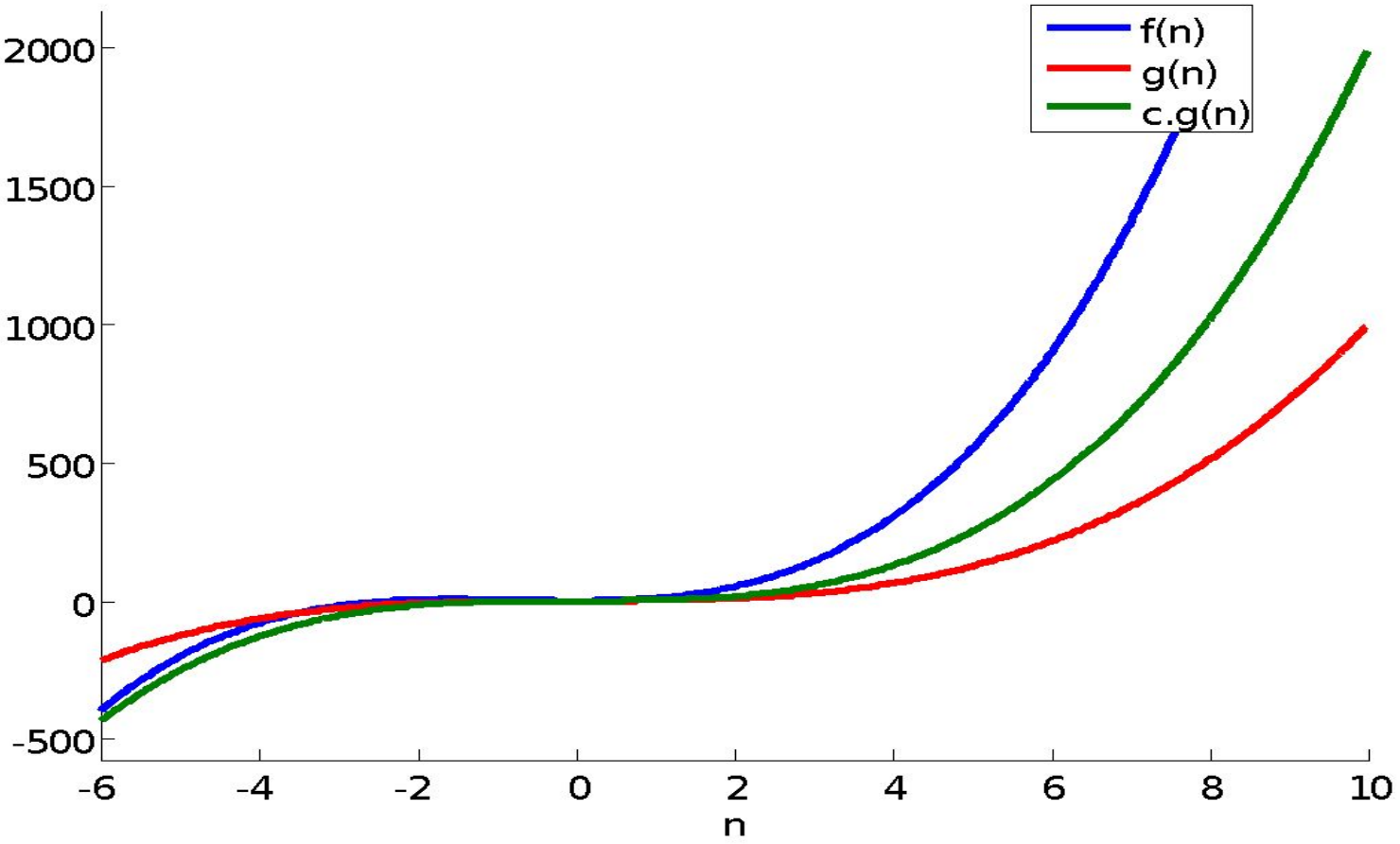
- Define o limitante inferior da complexidade assintótica do algoritmo.

- Para uma função f(n) que define a complexidade de um algoritmo, ele é $\Omega(g(n))$ se $0 \leq c.g(n) \leq f(n)$ para algum c e para todo $n \geq n_0$.

- Diz-se que se f(n) tem complexidade $\Omega(g(n))$ então ele cresce no mínimo tão lentamente quanto g(n).

UFABC

Notações Assintóticas: classe Ω



Notações Assintóticas: classe Θ

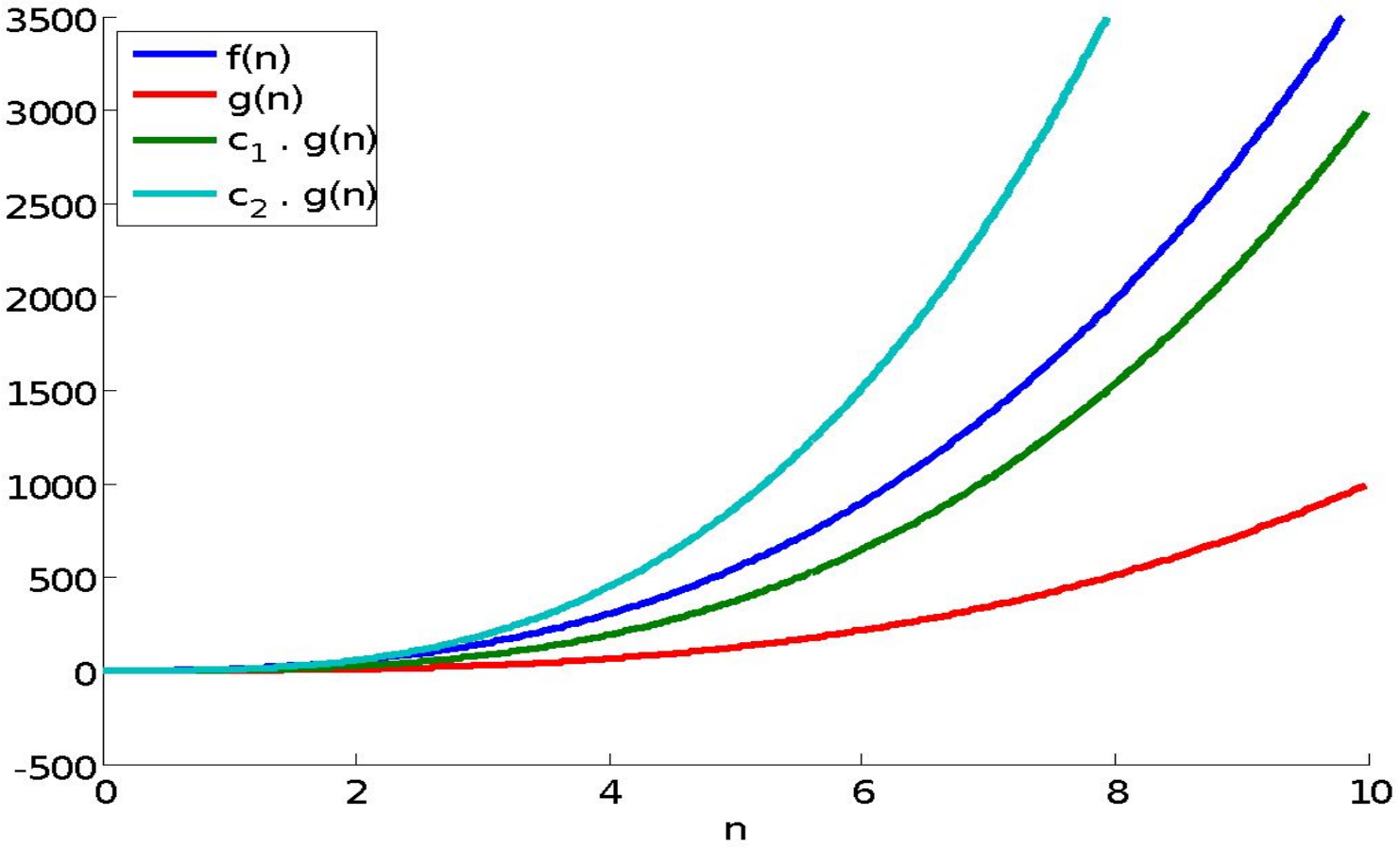
- Define um limitante estrito da complexidade assintótica do algoritmo.

- Para uma função f(n) que define a complexidade de um algoritmo, ele é $\Theta(g(n))$ se $c_1.g(n) \leq f(n) \leq c_2.g(n)$ para algum c_1, c_2 e para todo $n \geq n_0$.

- Diz-se que se f(n) tem complexidade $\Theta(g(n))$ então ele cresce tanto quanto g(n).

UFABC

Notações Assintóticas: classe Θ



Recorrência

- Mas e quanto a complexidade do algoritmo de Euclides?

- Para efetuar a contagem de operações necessárias nesse caso, vamos reescrever o algoritmo de uma forma recursiva.

UFABC

Recorrência

Algoritmo de Euclides Recursivo		
1	Entrada: a, b	
2	Se a ≠ b faça	
3	Se a > b então	
4	Retorne Euclides(a mod b, b)	
5	Senão	
6	Retorne Euclides(a, b mod a)	
7	Senão	
8	Retorne a	
9	Saída: a	

UFABC

Recorrência

Algoritmo de Euclides Recursivo		
		# execuções
1	Entrada: a, b com a > b	
2	Se a ≠ b faça	1
3	Retorne Euclides(b,a mod b)	T(b,a mod b)
7	Senão	1
8	Retorne a	1
9	Saída: a	

Utilizar divisão requer menor número de passos (subtração agrupada)
Dado que a > b, a mod b = r < b e o primeiro parâmetro sempre será maior que o segundo.

UFABC

Recorrência

Algoritmo de Euclides Recursivo		
		# execuções
1	Entrada: a, b com a > b	
2	Se a ≠ b faça	1
3	Retorne Euclides(b,a mod b)	T(b,a mod b)
7	Senão	1
8	Retorne a	1
9	Saída: a	

O número de execuções depende da função T(b, a mod b), que é uma função recursiva.

UFABC

Recorrência – Método da Iteração

- O número de execuções depende da função T(a, b), que é uma função recursiva.

- Denotando a mod b como r_0 e sabemos também que:
  - $T(a,b) = 1 + T(b, r_0) = 2 + T(r_0, r_1) = \ldots = N + T(r_{N-2}, r_{N-1}) = N + 1$
  - $T(a,0) = 0$

Recorrência – Método da Iteração

- Supondo que leva-se N passos para chegar ao final se fizermos N=1 temos que:

- T(a,b) = 1 => e b divide a com resto igual a 0.
- Os menores números em que isso é possível é a=2 e b=1*

*lembre-se: a > b sempre!!

UFABC

Recorrência – Método da Iteração

- Fazendo N=2 temos que:

- T(a,b) = 2 => e b divide a com resto diferente de 0 na primeira recursão.
- Os menores números em que isso é possível é a=3 e b=2*

*b=1 levaria a apenas uma recursão.

Recorrência – Método da Iteração

- Fazendo N=3 temos que:

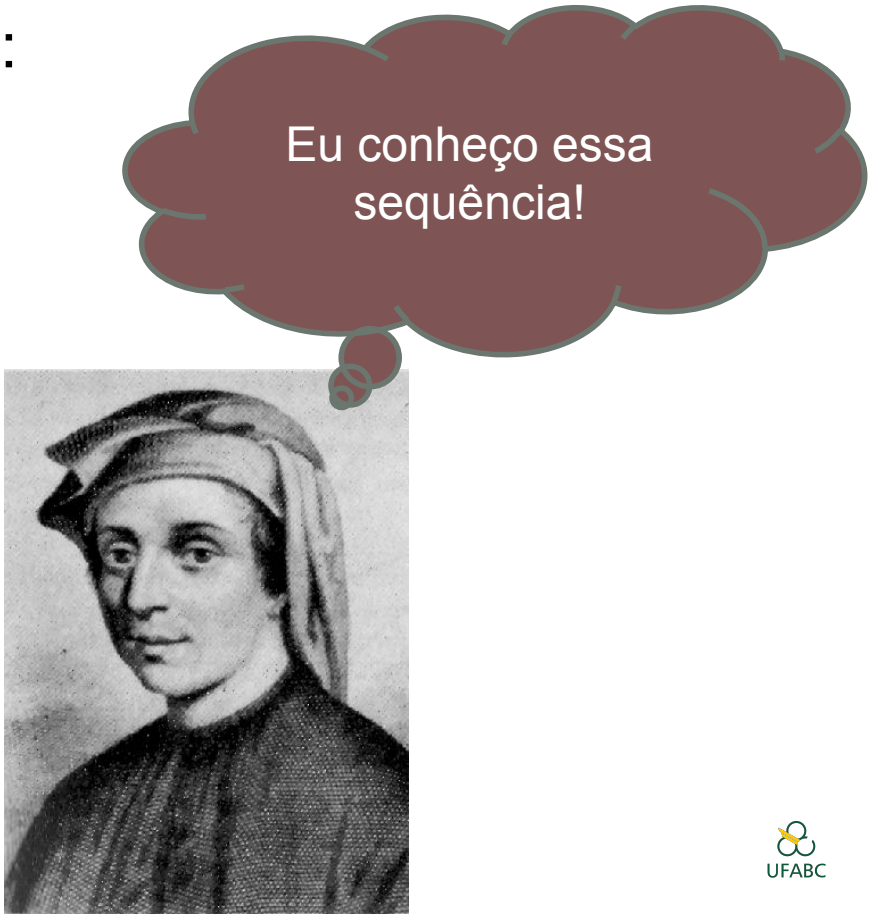
- T(a,b) = 3 => e b divide a com resto maior que 1 na primeira recursão.
- Os menores números em que isso é possível é a=5 e b=3*

*b=2 levaria a apenas duas recursões, pois todo resto da divisão por 2 é 1 e apenas com a=5 teríamos resto 2.

Recorrência – Método da Iteração

- Prosseguindo chegamos a:

N	a	b
1	2	1
2	3	2
3	5	3
4	8	5
5	13	8
6	21	13
7	34	21



UFABC

Recorrência – Método da Iteração

- Logo, dado uma quantidade N de recursões, os menores números a e b que levam a essa quantidade são os números de fibonacci F_{N+2} e F_{N+1}, respectivamente. (prova por indução)

UFABC

Recorrência – Método da Iteração

- Então, se o algoritmo requer N passos para completar, o valor de b deve ser maior ou igual a F_{N+1}.

- $F_{N+1} \geq \phi^{N-1}$ => $b \geq \phi^{N-1}$ => $\log(b) \geq (N-1)\log(\phi)$ => $\log(b)/(N-1) \geq \log(\phi)$

- Como $\log(\phi) > 1/5$:
  - $\log(b)/(N-1) > 1/5 \Rightarrow 5\log(b) > N-1 \Rightarrow 5\log(b) \geq N$

UFABC

Recorrência – Método da Iteração

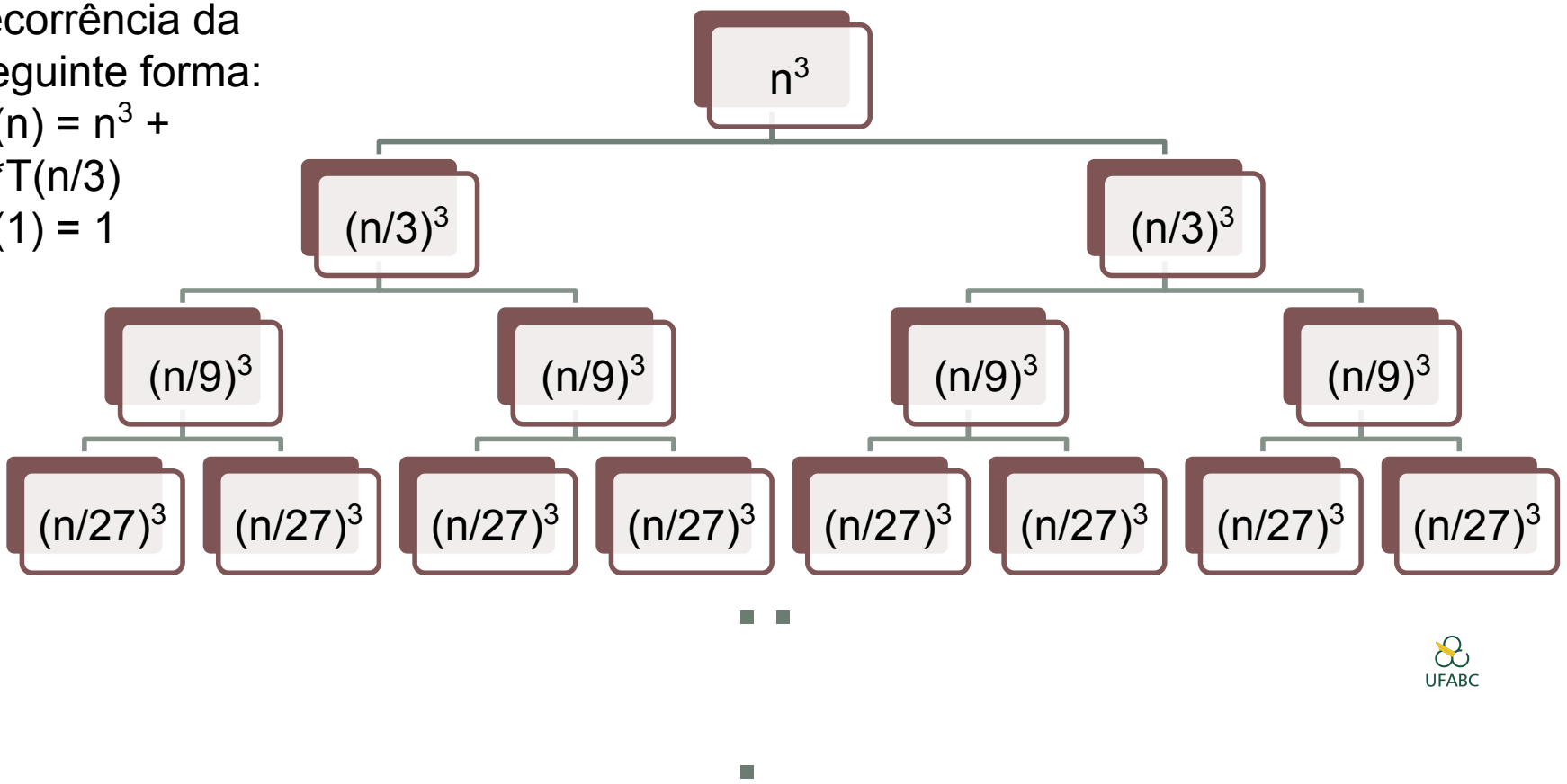
- Logo, o número máximo de recursão N é proporcional ao número de dígitos do número b: O(log(b)).

- Esse algoritmo é então O(logn) em relação ao número de dígitos do segundo parâmetro.

Recorrência – Árvore de Recorrência

Suponha uma função de recorrência da seguinte forma:
$T(n) = n^3 + 2*T(n/3)$
$T(1) = 1$

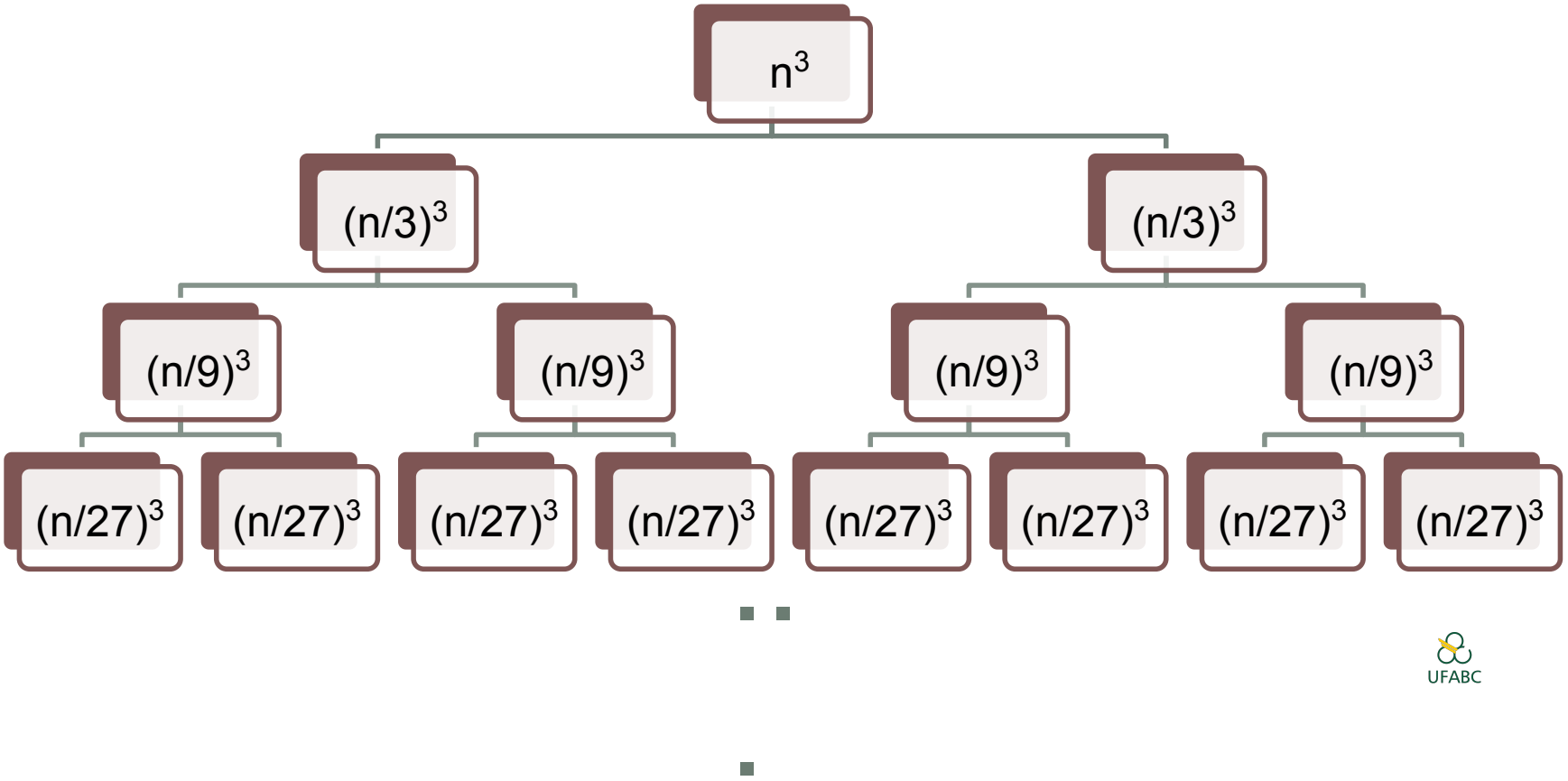
Se construírmos uma árvore desmembrando as recursões, teremos:



UFABC

Recorrência – Árvore de Recorrência

A altura dessa árvore será proporcional a $log_3 n$



UFABC

Recorrência – Árvore de Recorrência

A soma dos custos de operação vai ser:

$$n^3 + \frac{2}{9}n^3 + \frac{2}{3}\frac{2}{9}n^3 + \frac{2^2}{3}\frac{2}{9}n^3 + \frac{2^3}{3}\frac{2}{9}n^3 + \cdots + \frac{2^{log_3 n}}{3}\frac{2}{9}n^3 = O(n^3)$$