SCC5900 Projeto de Algoritmos

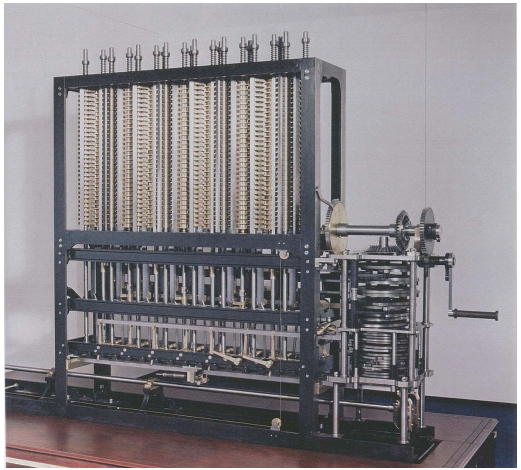
Complexidade de Algoritmos

Joao Batista
ICMC - USP

Por que se preocupar com complexidade?

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? ”

Charles Babbage (1864)



O que é um algoritmo Eficiente?

- Todos queremos um programa que seja **rápido**.
- Além disso, alguns podem se preocupar com a quantidade de memória

- Como afinal, medir a eficiência de um algoritmo? Qual o principal quesito

- Obs: Existem problemas para os quais não se conhece nenhum algoritmo eficiente para obter a solução: *NP*-Completos > não trataremos disso aqui

O que é um algoritmo Eficiente?

- Def. 1: um algoritmo é eficiente se ele roda rapidamente em situações reais (entradas reais)

- Def. 2: Um algoritmo é eficiente se tem, qualitativamente, melhor desempenho, **no pior caso**, se comparado à qq solução força bruta !

- Def. 3: Um algoritmo é eficiente se ele tem tempo de execução polinomial.
  - n, $n \log n$, n^2, n^3.

O que é um algoritmo Eficiente?

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

O que é um algoritmo Eficiente?

- Vamos analisar a afirmação 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 que usa SimpleX o trocaria por TriploX ???

Análise de Algoritmos

- Há vários fatores a se considerar nesta afirmação:
  - Linguagem de programação utilizada
    - Compilada/Interpretada
    - Alto nível/ Baixo Nível
  - A implementação do SImpleX foi mal feita, enquanto TripleX é muito elaborada
  - Qual máquina em que testei os algoritmos
    - O que dizer quanto à quantidade de memória que possui?
  - Quantidade de dados processador, acima de tudo:
    - Se TripleX é mais rápido para 1000 números, este continua sendo mais rápido para uma quantidade maior???????

Complexidade de Algoritmos

- Em função destas questões, uma forma justa de predizer a eficiência de um algoritmo é desconsiderar:
  - Hardware
  - Linguagem
  - Habilidade do Programador.
- O que devemos é comparar algoritmos e não programas
  - A esta “arte” dá-se o nome de análise/complexidade de algoritmos.

SimpleX vs. TripleX

- Dos argumentos citados anteriormente, o número de operações é uma boa medida de eficiência.

- 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. Em termos de função, $g(n) = 1.000n$.

SimpleX vs. TripleX

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

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

SimpleX vs. TripleX

- 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

- Se entendemos eficiência como algo relacionado a quão rápido um algoritmo executa, então devemos considerar apenas quando o tamanho de n for **grande**.

- 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.

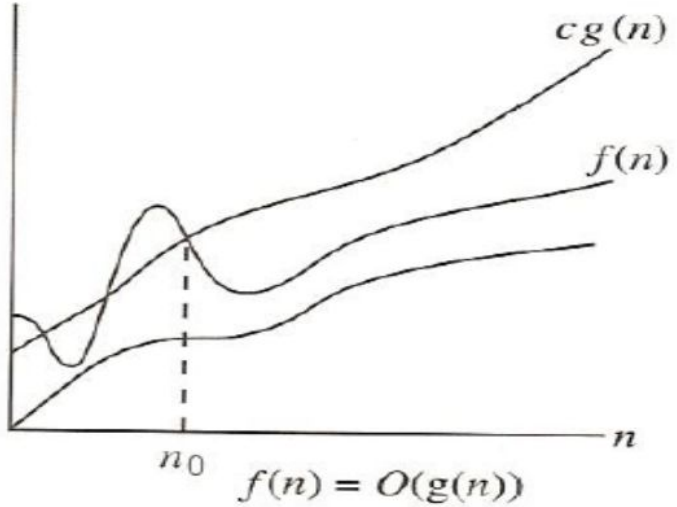
Análise Assintótica: **big-oh, omega, theta**

- Sejam 2 algoritmos com as seguintes funções de eficiência: $1000n$ 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, a partir de um certo ponto.

  - Qual valor é este??

Análise Assintótica: **big-oh, omega, theta**

- Dadas duas funções, *f(n)* e *g(n)*.
- diz-se que f(n) é 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).

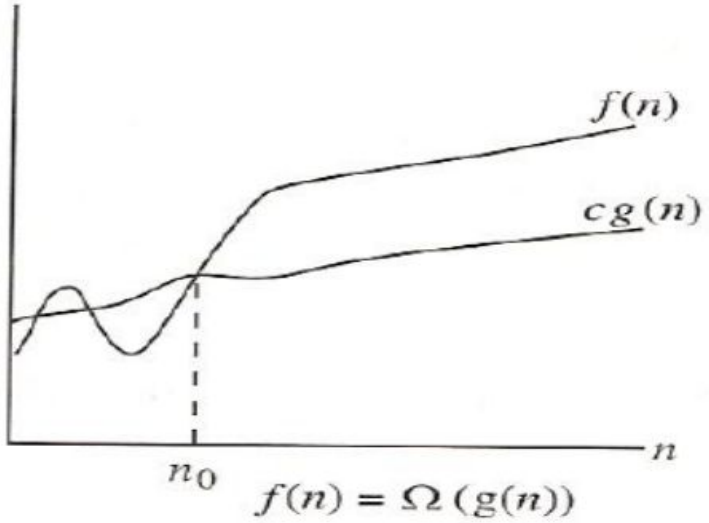


Análise Assintótica: **big-oh, omega, theta**

- Sejam 2 algoritmos com as seguintes funções de eficiência: 1000n 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,
  - seja g(n) = 1000n e f(n) = n^2. 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 men
  - Neste 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 = O(n^2).

Análise Assintótica: **big-oh, omega, theta**

- Dadas duas funções, *f(n)* e *g(n)*.
- diz-se que f(n) é **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).



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) = 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) = Ω(g(n)), tem-se que g(n) é o limite inferior de f(n).

Algumas considerações

- A função n^3 cresce mais rapidamente que n^2 :
  - $n^2 = O(n^3)$
  - $n^3 = \Omega(n^2)$

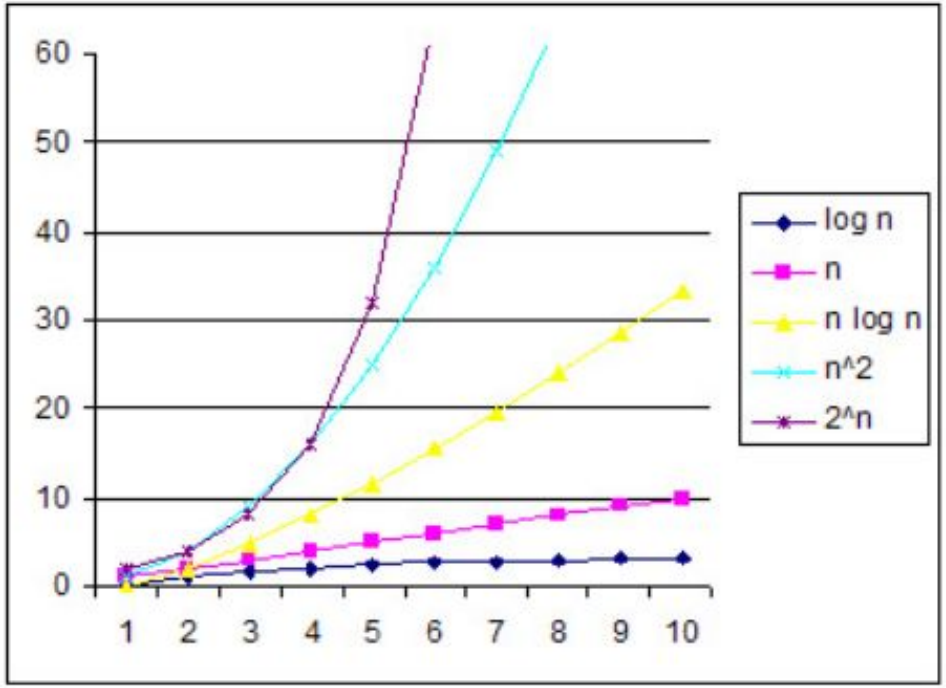
- Se $f(n) = n^2$ e $g(n) = 2n^2$, então essas duas funções têm taxas de crescimento iguais:
  - Portanto, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

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



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 $T(n) = O(2n^2)$ ou que $T(n) = O(n^2 + n)$:
  - Diz-se apenas $T(n) = O(n^2)$.

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:

Análise de Algoritmos

- Um 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 Subsequência Máxima

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

Exercício:

- Encontre exemplos comuns de problemas que possuam o seguinte tempo de processamento (big-oh):
  - linear
  - n log n
  - log n (sub-linear)
  - n^2
  - n3
  - polinomial (n^K)
  - exponencial (2^n)