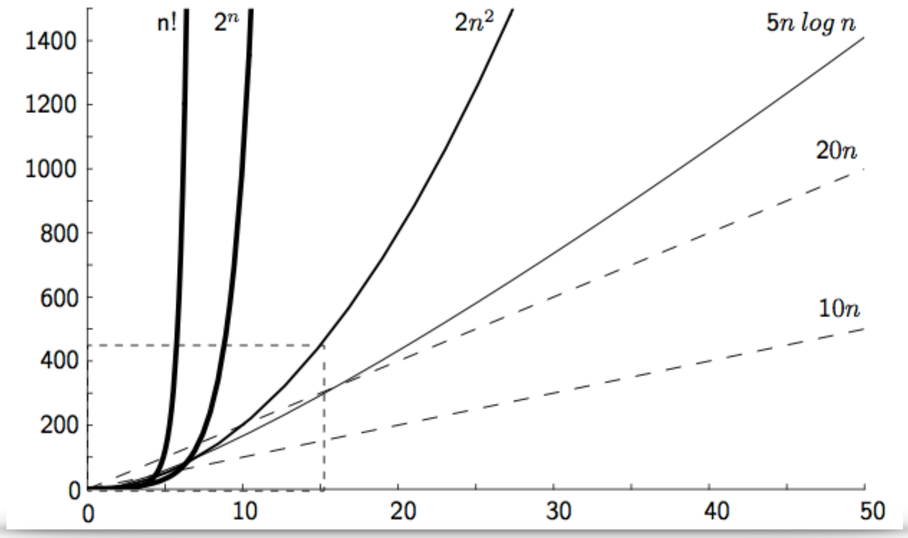
Profa. Dra. Raquel C. de Melo-Minardi
Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade Federal de Minas Gerais



MÓDULO 3
COMPLEXIDADE DE ALGORITMOS
Classes de complexidade

CLASSES DE COMPLEXIDADE

- Se *f(n)* é uma função de complexidade para um determinado algoritmo, então dizemos que *O(f(n))* é a sua **complexidade assintótica**

- Diversos algoritmos pertencem a uma mesma classe de complexidade *O(f(n))* e podemos dizer que eles são **equivalentes** em termos de custo computacional

  - Podemos usar essas **classes de equivalência** para comparar dois ou mais algoritmos

    - Por exemplo, suponha que um algoritmo é $f(n) = 2n^2$ e o outro é $g(n) = 100n$ Qual deles é melhor?

CLASSES DE COMPLEXIDADE

- A resposta é: depende.
  - Para $n < 50$, o programa quadrático $(f(n) = 2n^2)$ se comporta melhor que o linear $(g(n) = 100n)$
  - Para problemas com entradas pequenas, esse programa $O(n^2)$ se comporta melhor
  - Entretanto, quando os dados de entrada crescem, o programa $O(n)$ leva menos tempo

- $O(n^2)$ domina assintoticamente $O(n)$, o que nos levaria a escolher o programa da classe $O(n)$ que se comportaria melhor para entradas de tamanhos cada vez maiores
- Contudo, para entradas pequenas o programa da classe $O(n^2)$ pode apresentar um comportamento melhor

O(1)

- São algoritmos de complexidade **constante**, ou seja, o **custo independe** do tamanho da entrada pois as instruções são executadas um número fixo de vezes

- Esse tipo de algoritmo **não tem grande utilidade** pois não realiza nenhum tipo de processamento sobre a entrada

  - Exemplo: um algoritmo que imprime uma mensagem fixa na tela

O(LOG(N))

- São algoritmos de complexidade **logarítmica**
- **Dividem** um problema em **subproblemas menores**
- Um logaritmo gera um tempo de exceção que pode ser **menor que uma constante grande**
- Quando $n = 1000$, $log_2 n \approx 10$, quando n é *1* milhão, $log_2 n \approx 20$. Para que $log_2 n$ seja dobrado, é preciso tomar n ao quadrado!

  - Exemplo: não conheço nenhum problema particular da bioinformática que seja resolvido em complexidade logarítmica, infelizmente, mas um exemplo clássico de algoritmo logarítmico que acabamos de vez é o de pesquisa binária que é usado em diversas áreas, inclusive em bioinformática

O(N)

- São algoritmos de complexidade **linear**
- Realiza um pequeno trabalho sobre cada entrada
  - Exemplo: um algoritmo que processa uma sequência para calcular o conteúdo GC ou mesmo a freqüência de cada um dos 4 nucleotídeos terá complexidade linear

O(N LOG(N))

- Tipicamente um algoritmo pertence a essa classe quando **divide um problema em subproblemas menores** e **posteriormente ajunta as soluções** para gerar a solução final

- É bastante eficiente pois está entre $O(n)$ e $O(n^2)$. Quando n é *1* milhão, $n \log_2 n$ é cerca de *20* milhões. Quando n é *2* milhões, $n \log_2 n$ é cerca de *42* milhões, pouco mais do que o dobro

  - Exemplo: também não conheço um problema típico de bioinformática que pertença a essa classe mas um exemplo clássico de algoritmos dessa classe são os algoritmos de ordenação que recebem um arranjo, o dividem em sub-arranjos menores, ordenam-os e ajuntam sucessivamente para construir o arranjo final ordenado

    - Em caso de interesse por esses algoritmos, procurar no capítulo sobre Ordenação em [Ziviani, 2004].

$O(N^2)$

- São algoritmos de complexidade **quadrática**
- Algoritmos que processam elementos aos pares, muitas vezes um laços aninhados dentro de outros
- Observe que quando *n* é *1.000*, *f(n)* será da ordem de *1* milhão. Quando *n* dobra, o tempo é multiplicado por *4*
- São úteis apenas para resolver pequenos problemas
- Exemplo: o famoso algoritmo de Smith-Waterman [Waterman et al., 1981] que é base para os algoritmos de alinhamento de sequência par-a-par. Ele é quadrático sobre *n* onde *n* é o tamanho de uma das sequências. Caso as duas sequências alinhadas tenham tamanhos ordens de grandeza diferentes podemos dizer que ele é *O(mn)*, sendo *m* e *n* os comprimentos das sequências em termos de nucleotídeos ou aminoácidos

$O(N^3)$

- Tem complexidade chamada **cúbica** sendo útil apenas em problemas muito pequenos
- Quando *n* é *100*, o número de operações é da ordem de *1* milhão e que quando *n* dobra, o tempo de execução fica multiplicado por *8*

- Exemplo: um exemplo clássico de problema cúbico é a multiplicação de matrizes. Mais uma vez, caso as três dimensões envolvidas nas duas matrizes (matrizes de dimensão *m x n e n x l* resulta em uma matriz de dimensão *m x l)* sejam ordens de grandeza diferentes podemos dizer que o algoritmo é O(*mnl*)

$O(2^N)$

- São chamados **exponenciais** e não são úteis na prática
- Tipicamente ocorrem quando se usa a **força bruta** para resolver um problema
  - Esse jargão força bruta é muito utilizado em Ciência da Computação e significa que o algoritmo tenta todas as possibilidades de solução
- Quando *n* é *20*, faz-se cerca de *1* milhão de operações e quando *n* dobra, o tempo é elevando ao quadrado
- Exemplo: um problema de bioinformática que pertence a essa classe é o famoso Problema do Enovelamento de Proteínas (*PFP*, do inglês *Protein Folding Problem*). A base é o número de aminoácidos da proteína e o expoente é o número de conformações que um aminoácido pode assumir
  - Você pode calcular quão complexo esse problema é?

$O(2^N)$

- Outro exemplo interessante que pertence a essa classe é o *docking* de pequenas moléculas se considerar que a base é o número de ligações rotacionáveis da molécula e o expoente, o número de possível graus de liberdade.

O(N!)

- São algoritmos que tem complexidade **fatorial**
- Também ocorre quando se usa **força bruta** e são ainda piores que os da clases 2^n. Quando *n = 20*, *20! = 2.432.902.008.176.640.000*, um número com *19* dígitos. Quando *n = 40*, *40!* é um número com *48* dígitos!
  - Exemplo: um problema em bioinformática tipicamente fatorial é o problema da montagem de um genoma a partir dos fragmentos onde *n* é o número de fragmentos. A montagem de um genoma nada mais é do que a tentativa de se ordenar os fragmentos da forma como eles deveriam estar encadeados no genoma e um algoritmo que tentasse todas as possibilidades afim de encontrar a melhor ou mais correta tentaria n! possibilidades ou
    todas as permutações possíveis.
- Todo problema que envolver **permutações**, **combinações** ou **arranjos** (problemas de otimização combinatória) pertencerão a essa classe

COMPARANDO OS TEMPOS DE EXECUÇÃO

- Veja nessa tabela extraída de [Ziviani, 2004], quanto tempo programas das diversas classes de complexidade levariam para executar com entradas de tamanhos que variam entre $n = 10$ e 60:

Função de custo	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0035 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0,316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc
3^n	0,059 s	58 min	6,5 anos	3.855 séc.	10^8 séc.	10^{13} séc.

COMPARANDO OS TEMPOS DE EXECUÇÃO

- É assustador notar que problemas exponenciais são intratáveis mesmo para entradas tão pequena quanto $n = 50$
- Na prática, isso nos mostra que o problema do enovelamento de proteínas, por exemplo, não é possível de ser resolvido de forma ótima usando força bruta (ou seja, tentando todas as possibilidades), mesmo para proteínas com poucas dezenas de aminoácidos
- Há inúmeros problemas em Bioinformática que são exponenciais ou fatoriais
  - A maioria dos problemas interessantes em boinformática são intratáveis

COMPUTADORES MAIS POTENTES?

- Você pode estar se perguntando se temos boas perspectivas com o desenvolvimento de hardware mais avançado que processe os programas mais rapidamente ou em paralelo?

- Veja mais nessa tabela extraída de [Ziviani, 2004] que faz a estimativa do tamanho dos problemas que poderiam ser resolvidos se tivéssemos computadores mais rápidos

Função de custo de tempo	Computador atual	Computador 100x mais rápido	Computador 1.000x mais rápido
n	t_1	100 t_1	1.000 t_1
n^2	t_2	10 t_2	31,6 t_2
n^3	t_3	4,6 t_3	10 t_3
2^n	t_4	t_4 + 6,6	t_4 + 10

COMPUTADORES MAIS POTENTES?

- Note que o problema não é o poder computacional mas, de fato, a complexidade dos problemas

- Um problema exponencial não será possível de ser resolvido mesmo com um computador que seja 1.000 vezes mais rápido que o atual ou que tenha *1.000* núcleos e se possa resolver o problema em paralelo (supondo que isso possa ser feito)

- Note que, no caso de um problema exponencial, mesmo que o computador fosse *1.000* vezes mais rápido, poderíamos resolver um problema quase do mesmo tamanho que t_4 ou um problema de tamanho *10* unidades maior *(t_4 + 10)*

- Pensando no problema do enovelamento de proteínas, se atualmente fossemos capazes de enovelar computacionalmente uma proteína de t_4 aminoácidos, um computador *1.000* vezes mais rápido conseguia fazê-lo para uma proteína de t_4 + *10* aminoácidos.

Sumário

Há problemas cujos algoritmos pertencem a classes de funções **polinomiais** e outros que pertencem a classes **exponenciais**.

Comumente, dizemos que os **fatoriais** pertencem à classe **exponencial** também. A distinção entre essas duas grandes classes é bastante significativa quando o tamanho do problema cresce.

Os algoritmos **polinomiais** são **úteis** na prática, enquanto os **exponenciais não**.

Assim, um problema é considerado **intratável** quando **não existe um algoritmo polinomial para resolvê-lo** e **bem resolvido** quando o mesmo **existe**.