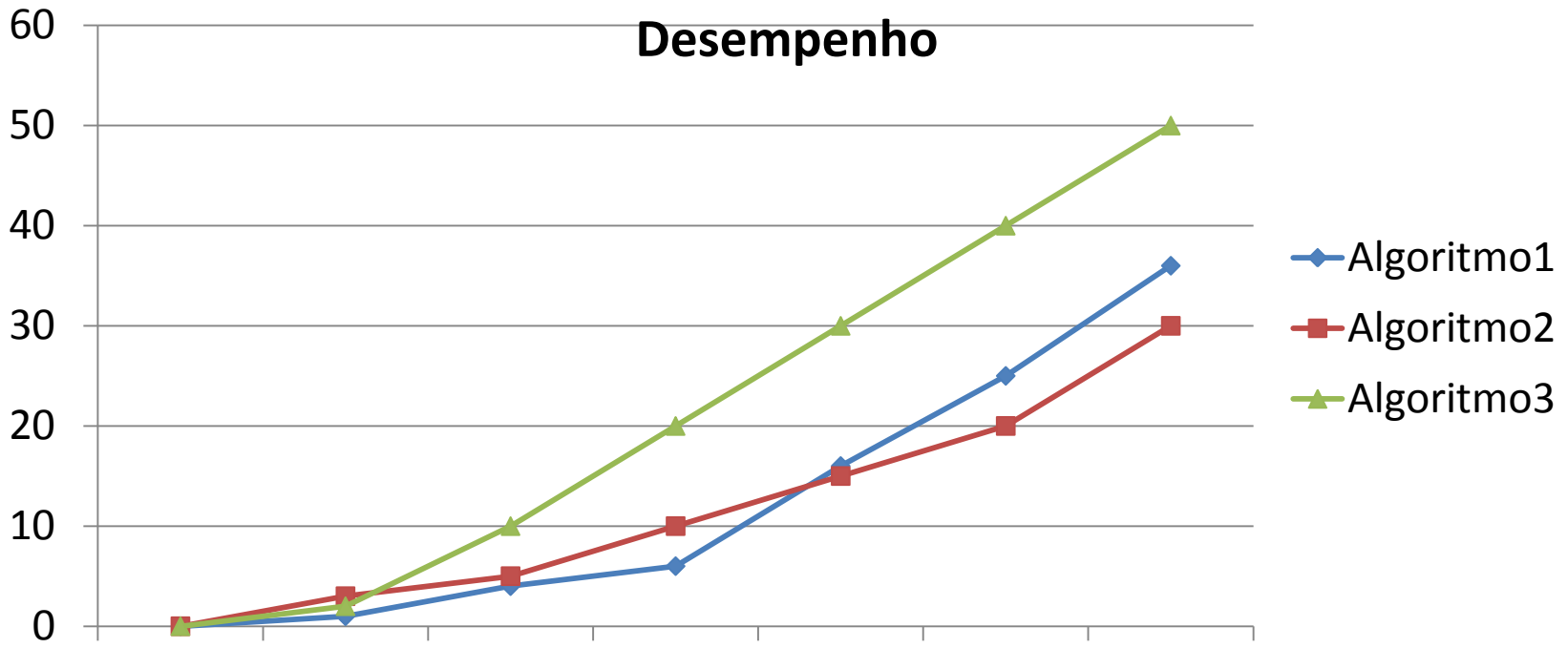


Complexidade de Algoritmos

Prof. Demétrios Coutinho

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
RIO GRANDE DO NORTE

Campus Pau dos Ferros
Disciplina de Algoritmos
Demetrios.coutinho@ifrn.edu.br

Introdução

Algoritmo é um processo sistemático para a resolução de um problema.

Estudo de algoritmos envolve 2 aspectos básicos: **correção e análise.**

- Correção: exatidão do método empregado (prova matemática).
- Análise: avaliar a eficiência do algoritmo em termos dos recursos (memória e tempo de execução) utilizados.

Algoritmo: Entrada (dados) ➡ Processamento ➡ Saída

Os princípios básicos de **Complexidade** é uma ferramenta útil para escolha e/ou desenvolvimento do melhor algoritmo a ser utilizado para resolver determinado problema.

INTRODUÇÃO

De uma forma mais genérica devemos identificar critérios pra medir a **qualidade** de um software:

- Lado do usuário ou cliente:
  - interface
  - robustez
  - compatibilidade
  - desempenho (rapidez)
  - consumo de recursos (ex. memória)
- Lado do desenvolvedor ou fornecedor:
  - portabilidade
  - clareza
  - reuso

A análise de algoritmos (ou análise de complexidade) é um mecanismo para entender e avaliar um algoritmo em relação aos critérios destacados, bem como saber aplica-los à problemas práticos.

Uma das formas mais simples de avaliar um algoritmo é através da análise empírica: rodar 2 ou mais algoritmos e verificar qual o mais rápido.

Desafios da análise empírica:

- Desenvolver uma implementação correta e completa.
- Determinar a natureza dos dados de entrada e de outros fatores que têm influência no experimento.

Tipicamente temos 3 escolhas básicas de dados:

- **reais**: Similar as entradas normais para o algoritmo; realmente mede o custo do programa em uso.
- **randômicos**: Gerados aleatoriamente sem se preocupar se são dados reais; testa o algoritmo em si.
- **problemáticos**: Dados manipulados para simular situações Anômalas; garante que o programa sabe lidar com qualquer entrada.

Complexidade de Algoritmos

Príncipios de Análise de Algoritmo ➡ Análise Empírica

É necessário levar em considerações algumas variáveis na hora de comparar algoritmos empiricamente: máquinas, compiladores, sistemas e entradas problemáticas.

Um dos possíveis problemas em se comparar algoritmos empiricamente é que uma implementação pode ter sido realizada com mais cuidado (otimizada) do que a outra.

Contudo em alguns casos a análise matemática é necessária:

- Se a análise experimental começar a consumir uma quantidade significando de tempo então é o caso de realizar análise matemática.
- É necessário alguma indicação de eficiência antes de qualquer investimento de desenvolvimento.

Análise Matemática é uma forma de avaliar um algoritmo que pode ser mais **informativa** e menos **cara** de se executar.

Razões para realizar análise matemática:

- **Comparação** de diferentes algoritmos para a mesma tarefa.
- **previsão** de performance em um novo ambiente.
- **configurar** valores de parâmetros para algoritmos.

Complexidade de Algoritmos

Príncipios de Análise de Algoritmo ➡ **Análise Matemática**

A maioria dos algoritmos possui um **parâmetro primário N**, que afeta o tempo de execução significativamente. Normalmente N é diretamente proporcional ao tamanho dos dados a serem processados.

O **objetivo** da análise matemática é expressar a necessidade de recursos de um programa (ex. tempo de execução) em termos de N, usando fórmulas matemáticas mais simples possíveis, mas que são precisas para valores elevados de N.

A idéia é oferecer uma análise **independente** da máquina, compilador ou sistema.

Algumas simplificações são necessárias para facilitar o processo de análise:

- Quantidades de dados manipulados pelos algoritmos será **suficientemente grande (avaliação do comportamento assintótico** (no limite))
- Não serão consideradas **constantes** aditivas ou multiplicativas na expressão matemática obtida.
- Termos de **menor grau** são desprezados.

A análise de um algoritmo leva em consideração:

- Um algoritmo pode ser dividido em etapas elementares ou **passos**.
- Cada passo envolve um número fixo de **operações básicas** cujos tempos de execução são considerados **constantes**.
- A operação básica de maior freqüência é a **operação dominante**.
- Devido a simplificação mencionada anteriormente o número de passos do algoritmo será o número de **execuções** da operação dominante.

Algoritmo para acessar um elemento em um vetor

```
01: Função acesso ( v: Vetor(N) Inteiro;  i,N: Inteiro ) : Inteiro
02:         Se i > N então
03:           erro("Acesso fora dos limites do vetor!");
04:         Senão
05:           retorne v[i];
06:         Fim-Se.
```

O tempo T de execução depende dos valores de **i** e **N**:

- Se i > N -> T = uma comparação + chamada de erro.
- Se i $\leq$ N -> T = uma comparação + um acesso ao vetor v + uma atribuição (implícita).

O tempo de execução é limitado por um número constante de operações, independente do tamanho da entrada.

Portanto a complexidade é dita **constante**

Algoritmo para achar o máximo elemento de um vetor

```
01: Função máximo (v: Vetor(N) Inteiro; N: Inteiro): Inteiro
02: var i, max: Inteiro;
03: Início
04:    Se N = 0 Então % c1
05:       erro("máximo chamado com vetor vazio!");
06:    Senão
07:       max := v[1]; % c2
08:       Para i := 2 Até N Faça % c3
09:          Se v[i] > max Então % c4
10:             max := v[i]; % c5
11:       Fim-Para
12:    Fim-Se
13:    Retorne max; % c6
14: Fim.
```

- O tempo T de execução seria:

$$T \leq (c1 + c2) + (n - 1)(c3 + c4 + c5) + c6)$$

$$T \leq n(c3 + c4 + c5) + (c1 + c2 - c3 - c4 - c5 + c6)$$

- $T \leq c * n$, onde c é uma constante que depende do sistema.
- Portanto a complexidade do algoritmo máximo é **linear.**

Algoritmo para achar o máximo elemento de um vetor

```
01:  for ( i = 0; i < n; i++ ) {
02:     for ( j = 1, sum = a[0]; j <= i; j++ )
03:        sum += a[j];
04:     cout « "A soma do sub-vetores de 0 até " « i «" é "« sum « endl;
05: }
```

- O que ele faz?
  imprime a soma de todas os sub-vetores iniciados na posição 0.
- Analisando o programa:
  - Antes do laço externo (linha 1) ser executado o i é inicializado.
  - O laço externo é acionado **n** vezes e em cada iteração são executados: o laço interno (linha 2), um comando de impressão e atribuições para **i, j e sum.**
  - O laço mais interno é executado **i** vezes para cada i ϵ {1, . . . , n − 1} com duas atribuições em cada iteração: **sum e j**

Algoritmo para achar o máximo elemento de um vetor

```
01:   for ( i = 0; i < n; i++ ) {
02:      for ( j = 1, sum = a[0]; j <= i; j++ )
03:         sum += a[j];
04:      cout « “A soma do sub-vetores de 0 até “ « i «” é ”« sum « endl;
05: }
```

- Portanto, temos n(p1 + p2 + p3 + p4 + p5) -> n(p1 +(n – 1)(p2 + p3) + p4 + p5) -> np1 + n (n – 1)(p2 + p3) + np4 + np5 -> np1 + (n^2 – n)(p2 + p3) + np4 + np5 -> n^2 – n -> n^2

- Assim temos um programa com complexidade **quadrática.**

Nem sempre a análise é trivial, pois o número de iterações de laços **depende** dos dados de entrada

Considere o seguinte trecho de um programa que dado um vetor A, determina o comprimento do maior sub-vetor ordenado:

```
01:  int A[]={ 1, 8, 1, 2, 5, 0, 11, 12 };
02:  int n=8;
03:  int i=0, k=0, length=0, idxStart=0, idxEnd=0;
04:  for ( i=0, length=1; i < n-1; i++ ) {
05:     for ( idxStart = idxEnd = k = i; k < n-1 && A[k]<A[k+1]; k++, idxEnd++ );
06:
07:     if ( length < (idxEnd - idxStart + 1) )
08:        length = idxEnd - idxStart + 1;
09:  }
10:   cout « ”\n Comprimento do maior sub-vetor ordenado: ” « length « endl;
```

Se **A** estiver em ordem decrescente o laço externo é executado **n – 1 vezes**, mas em cada iteração o laço interno é executado apenas 1 vez -> O(n) ou **linear**

O algoritmo é menos eficiente se A estiver em ordem crescente; por que?

R: laço externo é executado **n – 1 vezes**, e o laço interno é executado i vezes para cada i 2 ϵ {n − 1, . . . , 1}) -> O(n^2) ou **quadrático**

```
01:  int A[]={ 1, 8, 1, 2, 5, 0, 11, 12 };
02:  int n=8;
03:  int i=0, k=0, length=0, idxStart=0, idxEnd=0;
04:  for ( i=0, length=1; i < n-1; i++ ) {
05:     for ( idxStart = idxEnd = k = i; k < n-1 && A[k]<A[k+1]; k++, idxEnd++ );
06:
07:     if ( length < (idxEnd - idxStart + 1) )
08:        length = idxEnd - idxStart + 1;
09:  }
10:   cout « ”\n Comprimento do maior sub-vetor ordenado: ” « length « endl;
```

Desenvolver um algoritmo para transpor uma matriz quadrada M. Os parâmetros do algoritmo são a matriz M, de tamanho n × n. Não utilize matriz ou vetor auxiliar na solução.

Determinar a complexidade do algoritmo em função de n.

```
01: Função transpor(M: Ref Matriz[n,n] Inteiro; n: Inteiro)
02: Var aux, i, j: Inteiro;
03: Início
04:   Para i := 1 Até n-1 Faça % c1
05:     Para j := i+1 Até n Faça % c2
06:         aux := M[i][j]; % c3
07:         M[i][j] := M[j][i]; % c4
08:         M[j][i] := aux; % c5
09:     Fim-Para
10:   Fim-Para
11: Fim.
```

$T = (n - 1)(c1 + L)$, onde $L = (n - j)(c2 + c3 + c4 + c5)$.

Desenvolvendo teremos: $T = k2n^2 + k1n + k3 \rightarrow O(n^2)$

Algoritmo para somar os elementos de uma lista

```
01:  Função soma(L: Ref Vetor(N) Inteiro; N: Inteiro) : inteiro
02:  Var resposta: Inteiro;
03:  Início
04:     Se N = 0 Então % c1
05:        resposta := 0; % c2
06:     Senão
07:        resposta := (L[1] + soma(L[2..N], N-1)); % c3
08:     Fim-Se
09:     Retorne resposta; % c4
10: Fim.
```

- Cada chamada recursiva da soma decrementa o tamanho da lista. Exceção quando a lista é zero (caso base da recursão).
- Se N é o tamanho inicial, então o número total de chamadas será N + 1.
- Custo de cada chamada é: c1 + c2 + c4 (última chamada) e (c1 + c3 + c4) (demais chamadas).
- O tempo total T = N · (c1 +c3 +c4)+c1 +c2 +c4, ou seja, T N · c, onde c é constante -> a complexidade é dita **linear**

Para avaliar a complexidade em relação a memória necessária, é preciso compreender como a memória de micro-processadores é organizada.

Cada chamada de função ocupa um espaço na pilha de execução, onde é reservado espaço para variáveis locais e parâmetros da função chamada.

No caso do exemplo existem N + 1 chamadas de função. Portanto o consumo de memória é o somatório do comprimento das listas.

$$N + (N-1) + (N-2) + \cdots + 1 + 0 =$$
$$\sum_{i=0}^{N} i = \frac{N \times N+1}{2} \leq c \cdot N^2$$

Portanto a **complexidade espacial** é uma função **quadrática da** entrada N

Complexidade de Algoritmos

Crescimento de Funções

Em geral os algoritmos que iremos estudar possuem tempo de execução proporcional a uma das funções abaixo:

- **1**: maior parte das instruções são executadas apenas uma ou algumas vezes, independente de N – tempo de execução **constante.**
- **Log N**: ocorre em programas que resolve um problema maior transformado-o em uma série de subproblemas menores, assim reduzindo o tamanho do problema por uma certa constante fracionária a cada passo – tempo de execução **logarítmico**. Sempre que N dobra, log N aumenta de uma certa constante, mas log N não dobra até que N tenha sido aumentado para N2.
- **N**: acontece quando uma pequena quantidade de processamento deve ser feito para cada elemento da entrada – tempo de execução **linear**. Esta é a situação **ótima** para um algoritmo que deve processar N entradas (ou gerar N saídas)
- **N logN**: ocorre em algoritmos que quebra o problema principal em subproblemas menores, resolvendo-os e combinando as soluções –tempo de execução **"linearítmico"** ou **N logN**. Quando N dobra o tempo de execução torna-se um pouco mais do que o dobro

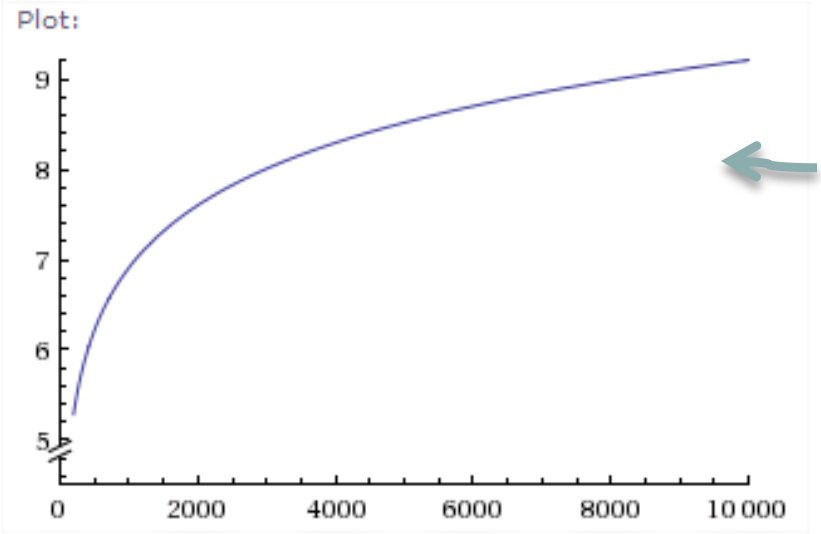
Crescimento de Funções

Em geral os algoritmos que iremos estudar possuem tempo de execução proporcional a uma das funções abaixo:

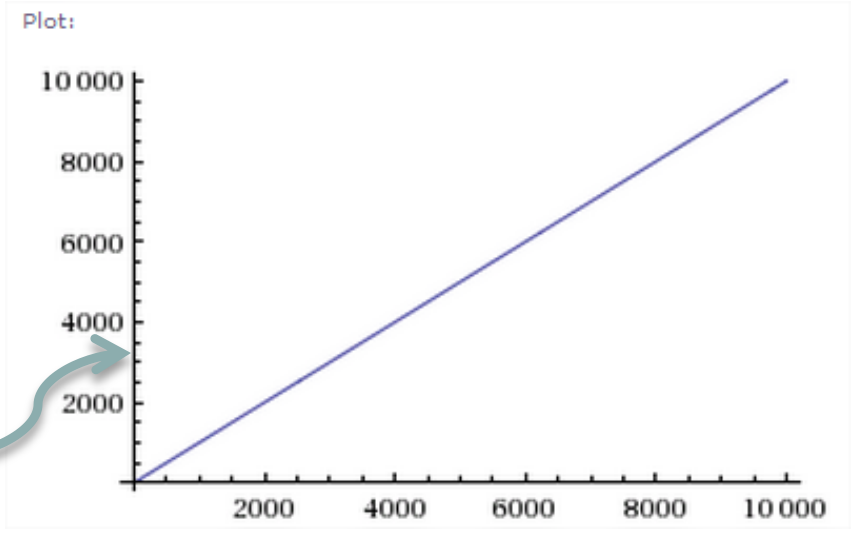
- **N2**: tipicamente representa algoritmos que processa todos os pares de itens de dados – tempo de execução **quadrático**. Este tipo de complexidade é aceitável apenas para problemas relativamente pequenos. Quando N dobra o tempo de execução aumenta 4 vezes.
- **N3**: similarmente refere-se a algoritmos que processam todos os triplas de itens de dados – tempo de execução **cúbico**. Quando N dobra o tempo de execução aumenta 8 vezes.
- **2^N**: corresponde a algoritmos que utilizam força-bruta na solução de problemas – tempo de execução **exponencial**. Algoritmos com esta performance são impraticáveis. Sempre que N dobra o tempo de execução é elevado ao quadrado!

Complexidade de Algoritmos

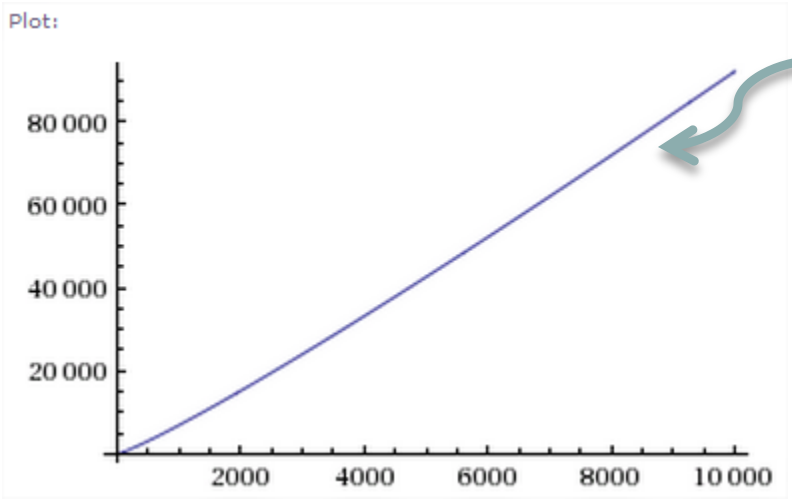
Crescimento de Funções



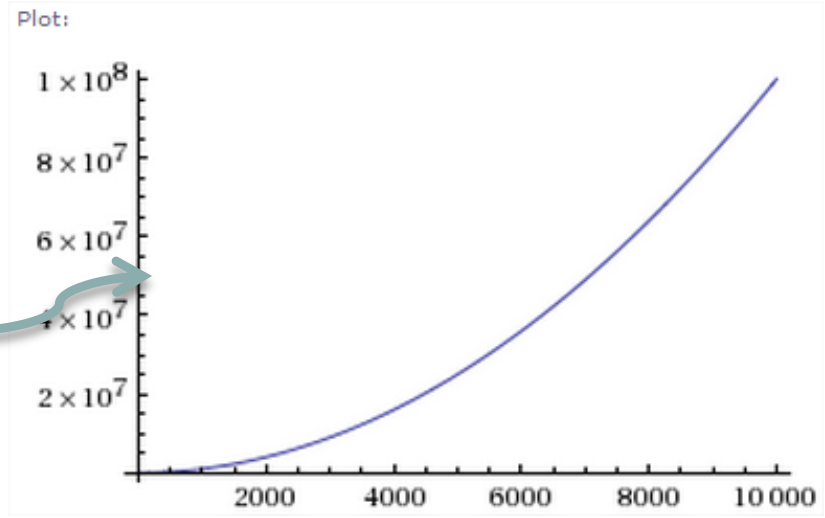
Log N



N



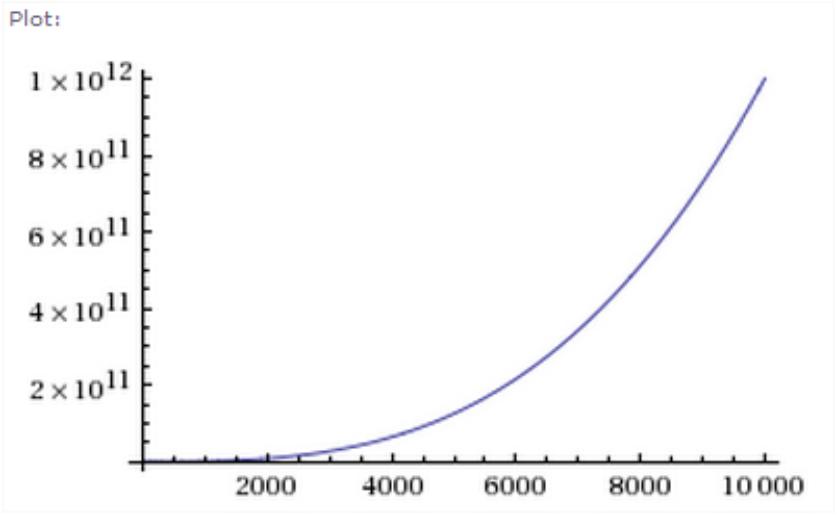
N Log N



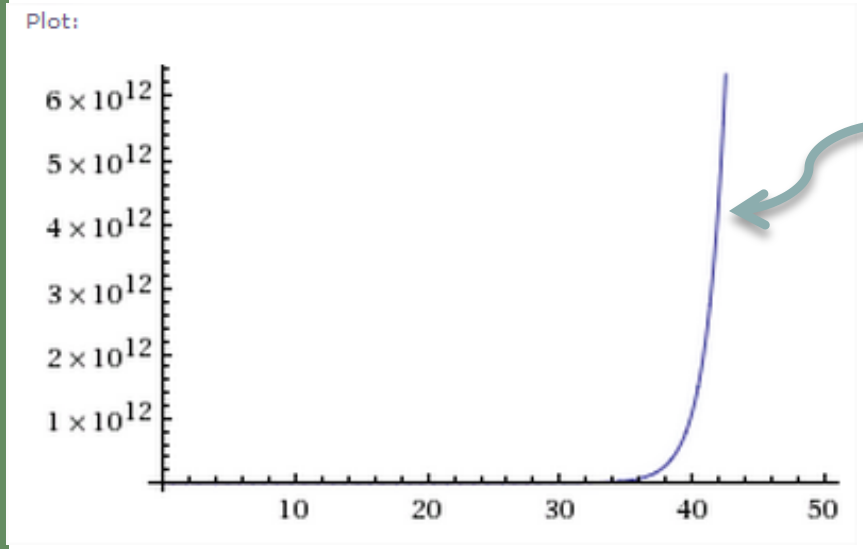
N^2

Complexidade de Algoritmos

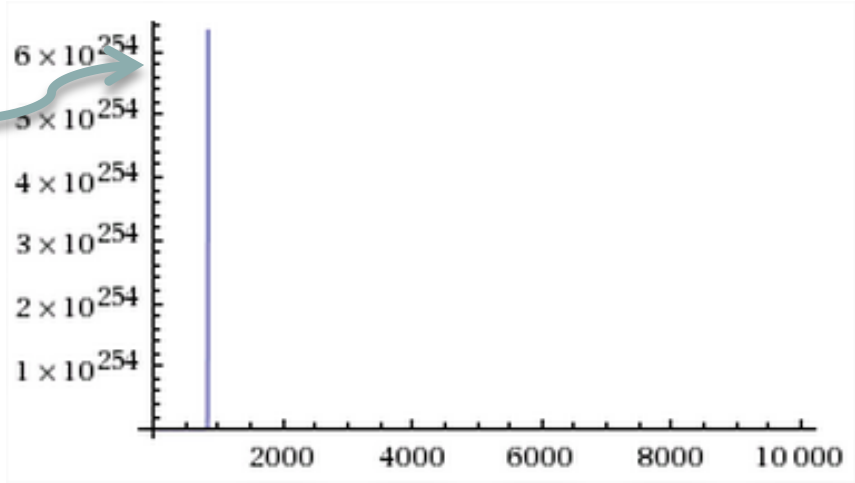
Crescimento de Funções



N^3



2^N



Complexidade de Algoritmos

Complexidade de Tempo

Exemplo de Tempo Necessário para Solução de Problemas

Veja abaixo a relação entre tempo de execução e tamanho de N para três máquinas com diferentes capacidades de processamento.

Operações por seg	Tamanho problema: 1 milhão			Tamanho problema: 1 bilhão		
	N	**N Log N**	**N^2**	**N**	**N Log N**	**N^2**
10^12	instante	instante	segundos	instante	instante	semanas
10^9	instante	instante	horas	segundos	segundos	décadas
10^6	segundos	segundos	semanas	horas	horas	nunca

Complexidade de Algoritmos

Pior, Melhor e Casos Médios

A complexidade do pior caso corresponde ao número de passos que o algoritmo efetua para a entrada mais desfavorável. É uma das mais importantes (porque?)

- Fornece um limite superior para o número de passos que o algoritmo pode efetuar, em qualquer caso.

Análises também podem ser feitas para o melhor caso e o caso médio. Neste último, supõe-se conhecida uma certa distribuição da entrada.

Exemplo: Busca sequencial de um dado elemento em um vetor armazenando n elementos de forma aleatória. Discuta o pior caso, melhor caso e o caso médio. No caso médio suponha a distribuição uniforme e que o dado buscado está dentro do vetor.

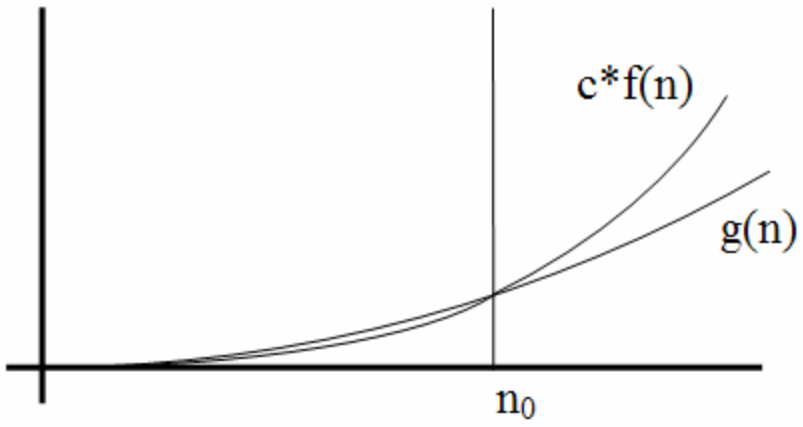
Limite Superior

A notação ***O*** mais comumente usada para medir algoritmos é O (big-Oh), que permite expressar um limite superior da complexidade.

Definição *O*: A função g(n) é O(f(n)), g = O(f), se existir c0 > 0 e um certo inteiro n0, tais que:

$$0 <= g(n) <= c^* f(n)$$

quando n > n0.



- A partir de n0, a função g fica sempre menor que a função f , com o fator multiplicativo c0.
- Se g é O(f) então g não cresce mais rapidamente que f.
- Uma fórmula com um termo O é denominada de expressão assintótica.

Complexidade de Algoritmos

Limite Superior

A idéia destas definições é estabelecer uma ordem relativa entre funções.

Portanto estamos interessados nas taxas relativas de crescimento.

Por exemplo:

Comparando 1000N e N^2. Quem cresce a uma taxa maior? Qual é o ponto de virada? N = 1000.

A definição diz que eventualmente existe um ponto n0 após o qual c0*f (n) é sempre, pelo menos, tão alta quanto g(n), de forma que, se as constantes podem ser ignoradas, f(n) é pelo menos tão grande quanto g(n).

No exemplo g(n) = 1000n, f (n) = n^2, n0 = 1000 e c0 = 1.

Portanto 1000N = O(N^2)

Limite Inferior

A notação Ω (big Omega) permite expressar um limite inferior da complexidade.

Definição Ω : A função g(n) é Ω(f (n)), g = Ω (f), se existir c0 > 0 e um certo inteiro n0, tais que:

$$0 <= c^* f(n) <= g(n)$$

quando n > n0.

- A partir de n0, a função g fica sempre maior que a função f , com o fator multiplicativo c0.
- Se g é Ω(f) então g cresce mais rapidamente que f.
- Problemas podem ter limite inferior: a ordenação com um único processador é Ω (n log n). Portanto não há algoritmos sequenciais mais eficientes que n log n para ordenação.

Complexidade de Algoritmos

Limite Exato

A notação Θ (big Theta) permite expressar uma estimativa precisa da complexidade de um algoritmo.

Definição Θ : A função g(n) é Θ (f (n)), g = Θ (f), se g(n) for O(f) e Ω(f) simultaneamente.

Se g é Θ (f) então g não cresce tanto quanto f.

Comparação de Algoritmos

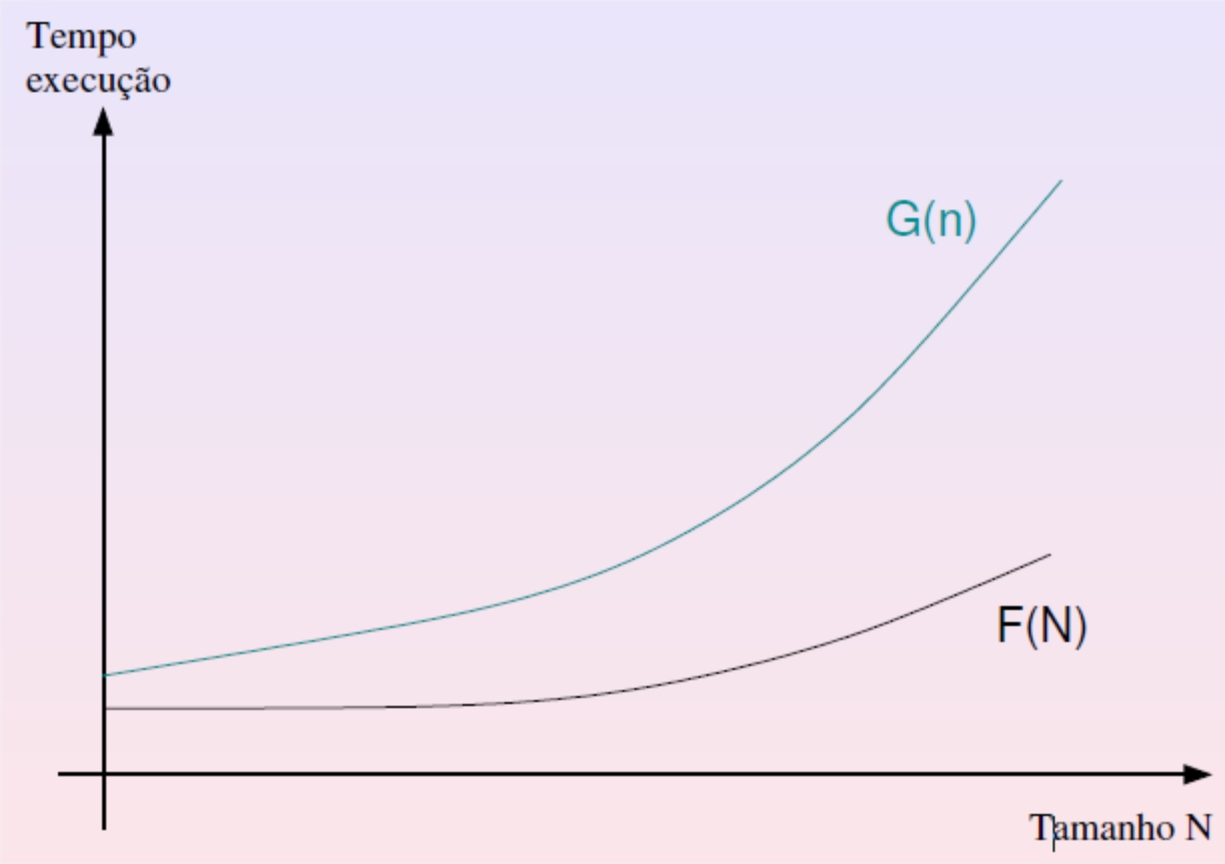
Para se comparar dois algoritmos devemos primeiramente calcular suas complexidades temporal e espacial.

Supondo que as funções F e G sejam expressões assintóticas de dois algoritmos diferentes, podemos compara-las da seguinte forma:

- Se F é sempre inferior a G, ou seja, o gráfico de F está sempre abaixo do gráfico de G, então escolhemos o algoritmo correspondente a F.
- Se F as vezes é inferior a G, e vice-versa, e os gráficos de F e G se interceptam em um número infinito de pontos. Neste caso, temos um empate, ou seja, a função de custo não é suficiente para escolher entre os dois algoritmos.
- Se F as vezes é inferior a G, e vice-versa, e os gráficos de F e G se interceptam em um número finito de pontos. Desta forma, a partir de um certo valor n, F é sempre superior a G, ou é sempre inferior. Neste caso, escolhemos o algoritmo cuja função é inferior para grandes valores de n execução.

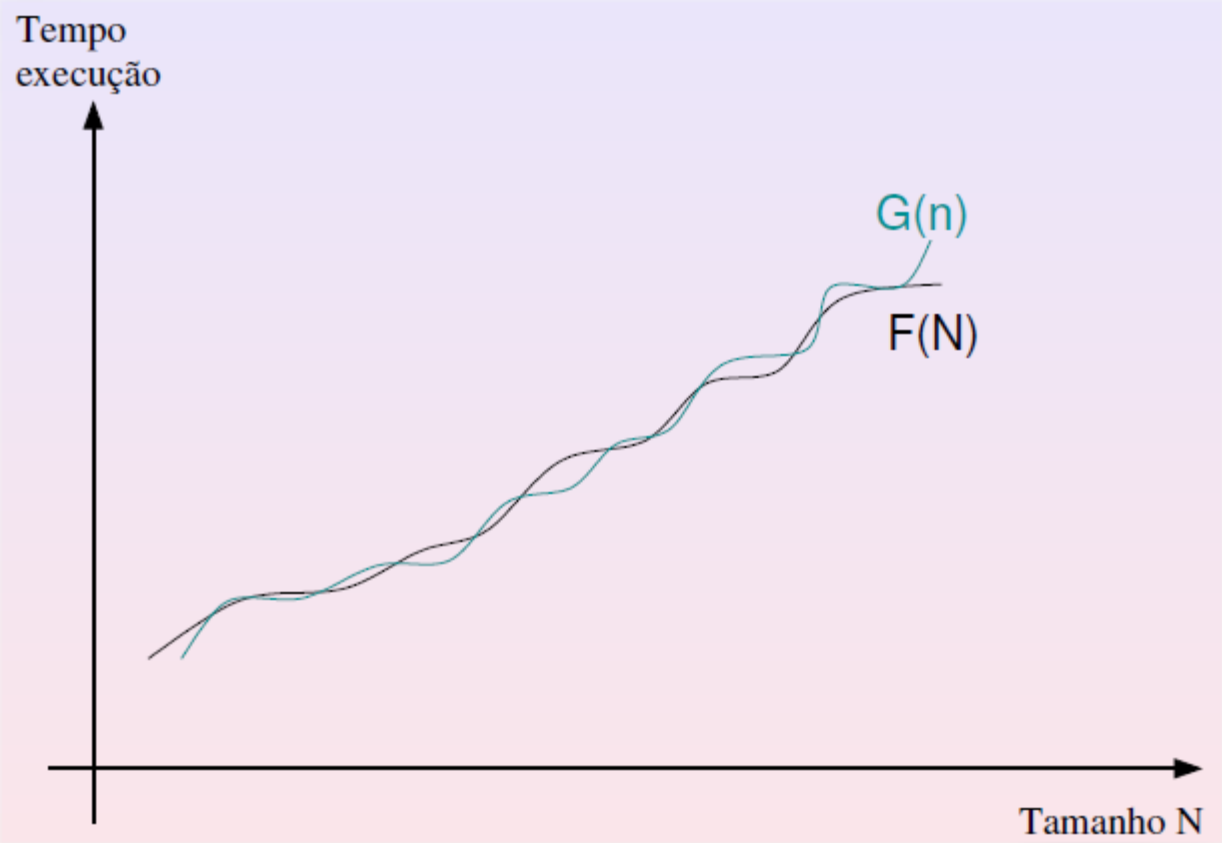
Complexidade de Algoritmos

Comparação de Algoritmos



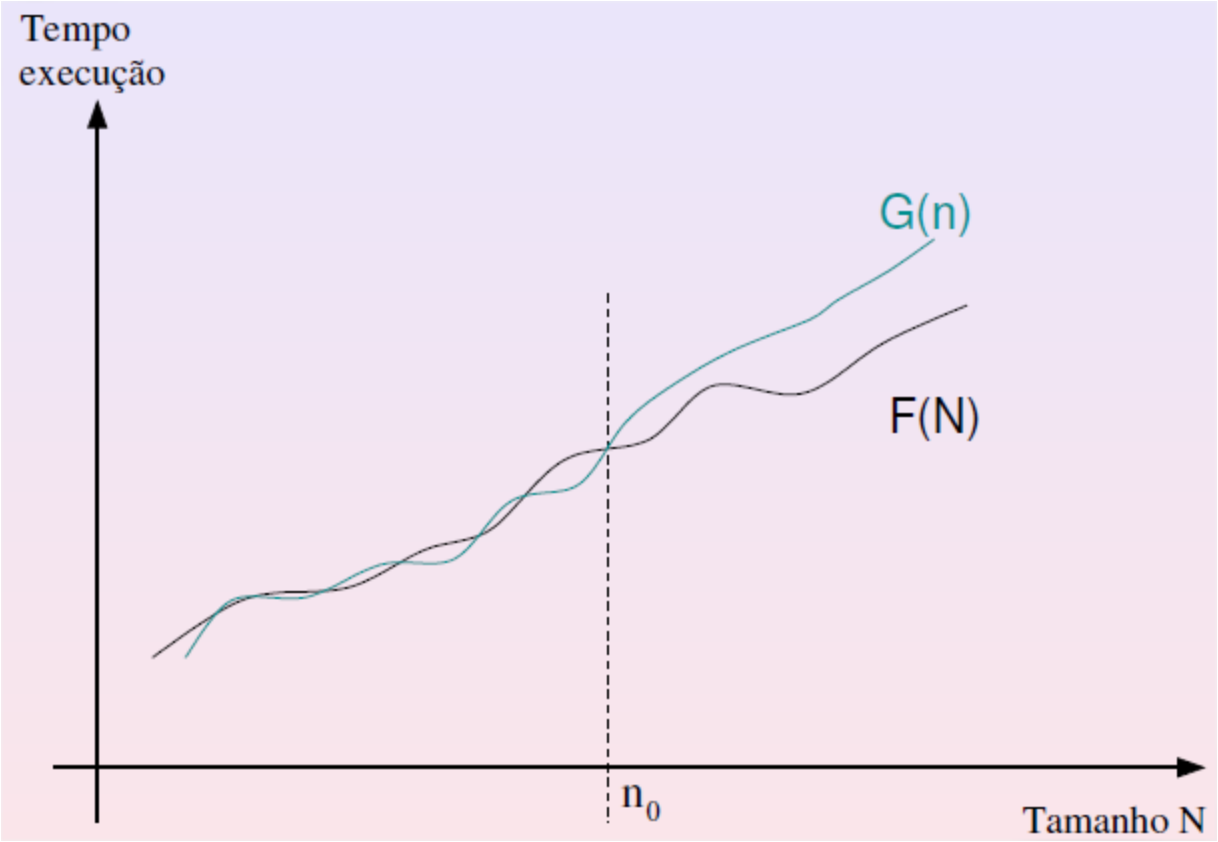
Complexidade de Algoritmos

Comparação de Algoritmos



Complexidade de Algoritmos

Comparação de Algoritmos



Resumindo

- Conceito de comportamento assintótico de função.
- Definição das notações de comparação de comportamento assintótico.
- Aplicação dessas notações para documentar a complexidade de algoritmos.
- Saber determinar a complexidade de algoritmos.
- Saber que existem problemas para os quais existe uma complexidade inerente.

Algoritmo Ótimo

Definição : Seja P um problema. Um limite inferior para P é uma função h, tal que a complexidade do pior caso de qualquer algoritmo que resolva P é $\Omega(h)$. Assim:

1. Todo algoritmo que resolve P efetua, pelo menos, $\Omega(h)$ passos.
2. Se existir um algoritmo A, cuja complexidade seja O(h), então A é denominado de algoritmo ótimo para P.
3. Neste caso, o limite $\Omega(h)$ é o melhor (maior) possível.

Algoritmo Ótimo

Considere o seguinte algoritmo para inverter um arranjo:

```
1: função inversão(V: Ref Vetor[n] inteiro; n: inteiro)
2:    var i, aux: inteiro;
3:    Início
4:    Para i := 1 até n/2 faça
5:        aux := V[i];
6:        V[i] := V[n-i+1];
7:        V[n-i+1] := aux;
8:    Fim-Para
9: Fim.
```

Todo algoritmo tem que efetuar a leitura dos dados -> o problema de inversão de sequencia é $\Omega(N)$. Como a complexidade do algoritmo é O(N), conclui-se que trata-se de um **algoritmo ótimo**.

Regras Gerais para Análise de Algoritmos

Regra 1: Laços **For**

- O tempo de execução para um laço For é, no máximo, o tempo de execução dos comandos internos do laço (incluindo os testes) vezes o número de iterações.

Regra 2: Laços aninhados

- Analise laços aninhados de dentro pra fora. O tempo de execução total de um comando dentro de um grupo de laços aninhados é o tempo de execução do comando multiplicado pelo produto do tamanho de todos os laços.

```
Para i := 1 até N faça
   Para j := 1 até N faça
      Temp := Temp + 1;
   Fim-Para
Fim-Para
```

Regra 3: Comandos Consecutivos

- Comandos consecutivos simplesmente são adicionados. Isto na prática significa que o tempo total é o do comando com maior tempo.

Regra #4: Condicional Composto Se/Senão

- Para um comando condicional composto o tempo total de execução não é nunca maior do que o tempo necessário para realizar o teste mais o maior dos tempos de execução entre os dois blocos do condicional (S1 e S2).

```
Se ( expressão ) então
    bloco S1
Senão
    bloco S2
Fim-Se
```

Exercicio 1

Problema: busca seqüencial em um vetor

- Dado um conjunto de valores previamente armazenados em um vetor A, nas posições A[l],A[l + 1], . . . ,A[n], verificar se um número v está entre este conjunto de valores. Se o elemento procurado v não for encontrado a função deve retornar −1. Caso contrário deve retornar o índice do vetor A que contém o elemento v.
- Desenvolver o algoritmo, calcular a complexidade para o melhor caso e pior caso para o algoritmo proposto.

Exercicio 2

Problema: busca binária em um vetor

- Dado um conjunto de valores previamente armazenados em um vetor A, nas posições A[l],A[l + 1], . . . ,A[n], em ordem crescente, verificar se um número v está entre este conjunto de valores. Se o elemento procurado v não for encontrado a função deve retornar −1. Caso contrário deve retornar o índice do vetor A que contém o elemento v.
- Desenvolver o algoritmo, calcular a complexidade para o melhor caso e pior caso para o algoritmo proposto.

Complexidade de Algoritmos

Exercicio 2

Melhor caso: o elemento procurado está no meio do vetor -> O(1).

Pior caso: o elemento não está no vetor

- Inicialmente procuramos em um vetor de tamanho n, então em um de tamanho n/2 , depois em um que é metade disso, n/2^2 , e assim por diante, até atingirmos tamanho 1.
- Portanto temos a sequencia n, n/2 , n/2^2 , . . . , n/2^m e queremos saber o valor de *m* (quantidade de passos).
- Porém, sabemos que o último termo da sequencia, n/2^m , é 1.
- Logo temos que m = lg n -> O(lg n).

BONS ESTUDOS :)

Algoritmos

Prof. Demétrios Coutinho

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
RIO GRANDE DO NORTE

Campus Pau dos Ferros
Disciplina de Algoritmos