025089 – Projeto e Análise de Algoritmos

Aula 02

lizier@dc.ufscar.br

Análise de Algoritmos

- Determinar corretamente o uso de um algoritmo
- Permitir a comparação entre algoritmos
- Determinar o uso dos recursos computacionais (processamento e memória)
- Prever o crescimento do uso destes recursos
- Relacionar o uso de recursos com o tipo/tamanho da entrada
- Balancear o uso entre os recursos (espaço vs tempo)

- ...

Análise de Algoritmos

- Eficiência (ou complexidade) de tempo
  - Tempo de processamento para a execução de um programa
  - Quão mais lento ficará um programa a medida que a entrada aumenta de tamanho?

- Eficiência (ou complexidade) de espaço
  - Espaço de memória necessária para a execução de um programa, excluindo o espaço necessário para a entrada e saída do programa
  - Quanto mais de espaço precisamos ter disponível a medida que a entrada aumenta de tamanho?

Análise de Algoritmos

- Eficiência de tempo/espaço ou complexidade de tempo/espaço

- Modelos:
  - Experimental ou empírico:
    - Definição da métrica de interesse
    - Geração das entradas de testes
    - Execução dos testes
    - Análise dos dados obtidos
  - Matemático:
    - Definição do tamanho da entrada
    - Definição do modelo de custo
    - Escolha do tipo de análise
    - Aproximação da ordem de crescimento

Análise de Algoritmos

- Antes de tudo:

  (comum nos dois modelos: empírico e matemático)

  - Compreender o objetivo do algoritmo
  - Entender as possíveis entradas
  - Distinguir a influência da entrada na execução do algoritmo
  - Determinar o que é caracterizado como **“tamanho do problema”**

Análise de Algoritmos

- Definição de “tamanho” do problema

  - Exemplos de problemas:
    - Ordenação: número de elementos (tamanho do vetor)
    - Multiplicação de matrizes: tamanho de uma dimensão (quadrada) ou número de elementos
    - Avaliação de polinômio: grau ou número de coeficientes
    - Verificar se um número é primo: magnitude do número
    - Grafos: número de vértices e/ou arestas
    - Busca de uma palavra no texto: número de caracteres de ambos ou palavras do texto

Exemplo: buscas

- Busca em uma sequência ordenada:
  - Entrada: uma sequência de n números ($a_0, a_1, a_2, \ldots, a_{n-1}$) ordenados, ou seja, $a_0 \leq a_1 \leq a_2 \leq \ldots \leq a_{n-1}$, e um número ***key*** a ser procurado;
  - Saída: índice da posição de ***key*** na sequência de entrada, ou o valor ***-1*** caso ***key*** não esteja presente na sequência de entrada;
- Algoritmos propostos:
  - Busca sequencial
  - Busca binária

Algoritmo: Busca sequencial

- Ideia base: começar do primeiro elemento e ir comparando um a um até encontrar (ou encontrar um elemento maior)

```
int bsequencial( T vetor[], T key, int n )
{
    int i = 0;
    while( (i < n) && (vetor[i] < key) )
        i++;
    if ( (i < n) && (vetor[i] == key ) )
        return i;
    else
        return -1;
}
```

Algoritmo: Busca binária

- Ideia base: comparar o elemento do meio para continuar a busca na metade certa

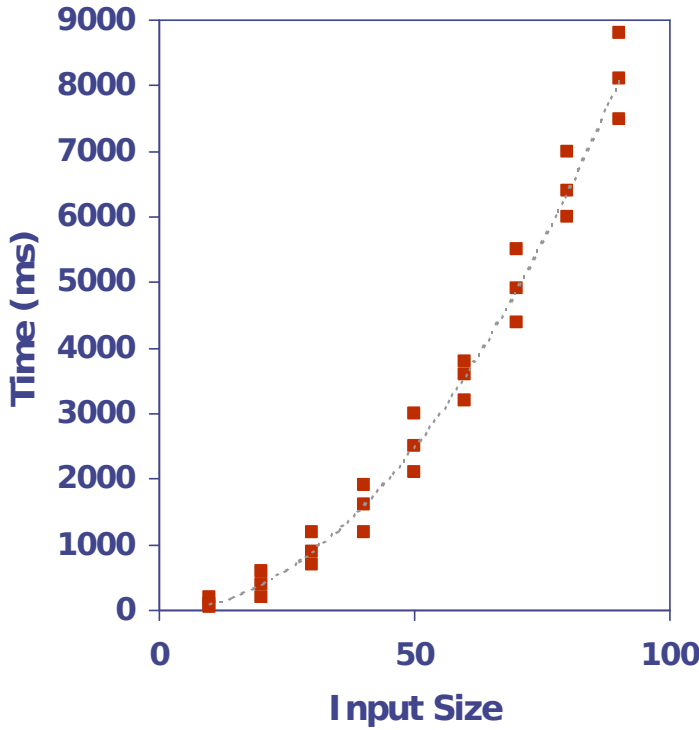
```
int bbinaria( T vetor[], T key, int n )
{
    int imax = n-1;
    int imin = 0;
    while( imax >= imin )
    {
        int imid = imin + ((imax - imin) / 2);
        if( key > vetor[imid] )
            imin = imid + 1;
        else if( key < vetor[imid])
            imax = imid - 1;
        else
            return imid;
    }
    return -1;
}
```

Exemplo: buscas

- Tamanho da entrada:
  - Tamanho do vetor, ou seja, o número de elementos pelos quais procuramos uma determinada chave
  - Os valores das entradas também influenciam o tempo de execução das buscas

- Vamos primeiro pensar no modelo experimental, escolhendo como modelo de custo o tempo total de execução
  - Poderíamos, por exemplo, ter escolhido a frequência de uma operação

Modelo experimental



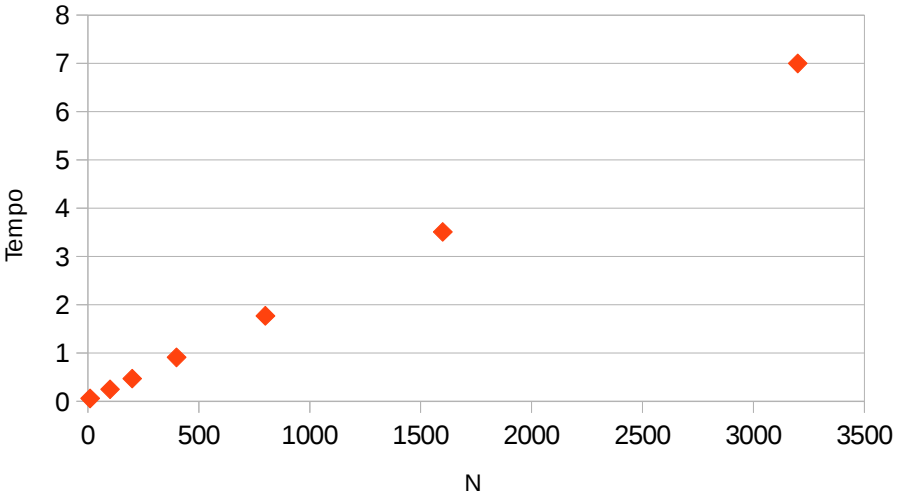
- Análise empírica:
  - Executar diversas vezes o algoritmo
  - Executar com entradas diferentes
  - Executar com entradas de tamanho diferentes
  - Medir o tempo e memória utilizados em cada execução

- Podemos responder várias questões com este modelo
- Em alguns casos pode ser muito difícil conseguir uma sequência boa de experimentos
- Muito dependente da máquina dos testes

Modelo experimental

- Busca sequencial
  - Espaço: não muda em relação as entradas (somar bytes)
  - Tempo: execução total do programa implementado em C
  - N: tamanho do vetor (número dos elementos)

N	Tempo em micro-segundos (10^{-6})
10	0.06
100	0.25
1000	2.21
10000	21.88
100000	219.79
1000000	?

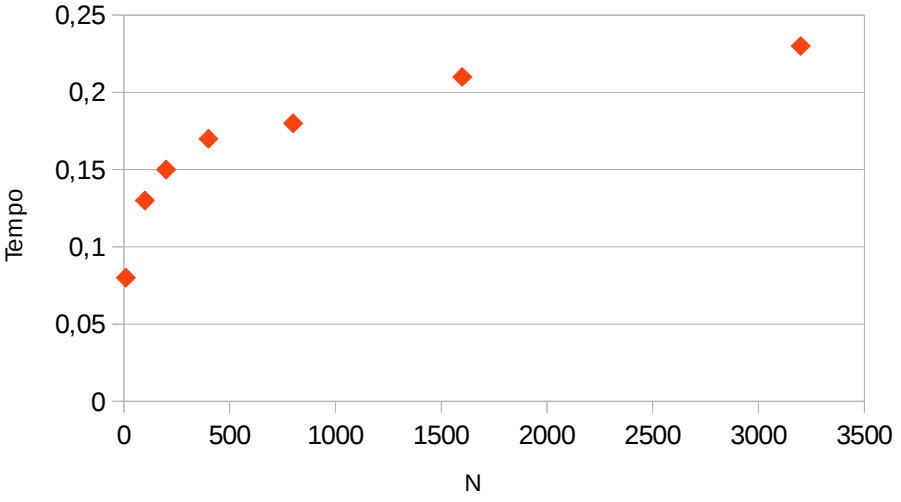


CPU: Intel Core i3-2310M – 2.1GHz

Modelo experimental

- Busca binária
  - Espaço: não muda
  - Tempo: execução total do programa implementado em C
  - N: tamanho do vetor (número dos elementos)

N	Tempo em micro-segundos (10^{-6})
10	0.08
100	0.13
1000	0.20
10000	0.27
100000	0.35
1000000	0.55



CPU: Intel Core i3-2310M – 2.1GHz

Modelo experimental

Considerações importantes

- Método aparentemente “simples”
- **Difícil** conseguir bons testes
- Não fornece qualquer garantia teórica
- Necessidade de realizar execuções
- Inclui variáveis da máquina (cache, concorrência, etc …)
- Inclui variáveis do compilador/linguagem
- Difícil realizar qualquer previsão

Modelo matemático

- Tempo total de execução:

  Somatório: **custo** x **frequência** de cada operação

- Precisamos analisar quais operações considerar e o número de operações
- **Custo** depende da máquina/compilador
- **Frequência** das operações dependem do algoritmo e dados de entrada
- Podemos analisar o consumo de memória, mas não vamos fazer isso agora

Modelo matemático

- Caracteriza o tempo de execução como uma **função** em relação ao tamanho/tipo da entrada
- Considera **todas** as possíveis entradas
- Baseada em uma descrição de alto nível (sem detalhes da linguagem/compilador)
- Independente da máquina (eliminando o custo e variáveis externas)
- Facilidade de fazer previsões
- Não é necessário realizar execuções

Modelo matemático

- Independente do sistema
  - Algoritmo
  - Dados de entrada

  } Determinam a ordem de crescimento

- Dependente do sistema
  - Hardware: CPU, memória, cache, …
  - Software: compilador, interpretador, *garbage collector*, …
  - Sistema: sistema operacional, rede, outros aplicativos, …

  } Determinam as constantes

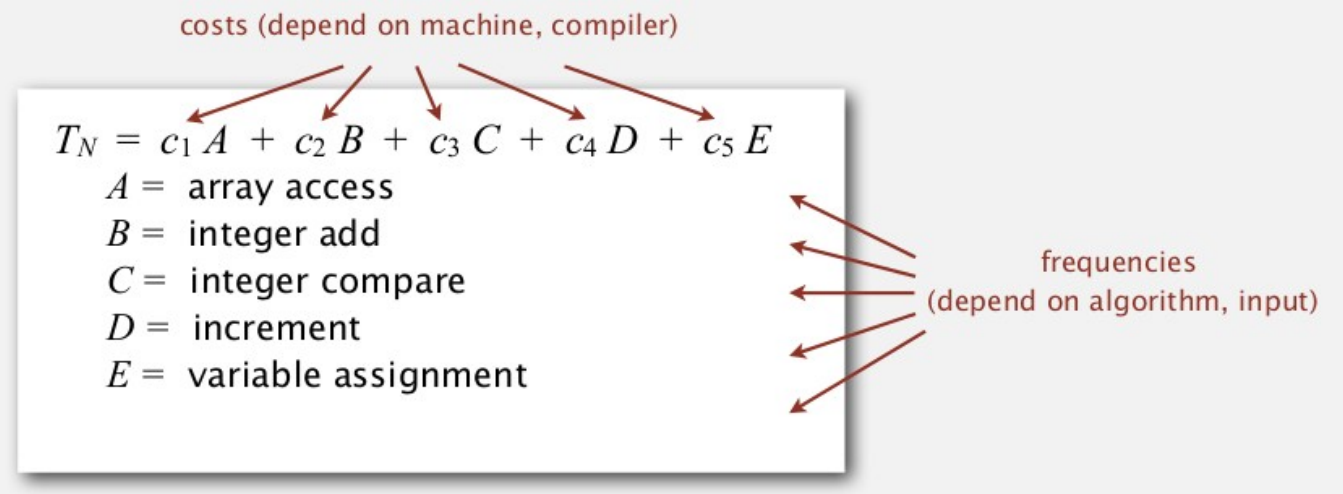
Modelo RAM
(Random Access Machine)

- Uma única **CPU**
- Memória “ilimitada” e de acesso constante e randômico (sem considerar *cache* ou tamanho de palavra)
- Realiza as operações básicas
- Independente de linguagem/compilador
- Em geral, assume-se tempo constante para cada operação

Algumas operações básicas

- Atribuição
- Comparação
- Acesso a vetor
- Acesso a variável
- Operação Arit.
- Operação Lógica
- Aloc. de variável

```
int bsequencial( T vetor[], T key, int n )
{
    int i = 0;
    while( (i < n) && (vetor[i] < key) )
        i++;
    if ( (i < n) && (vetor[i] == key ) )
        return i;
    else
        return -1;
}
```



Cost of basic operations

operation	example	nanoseconds †
integer add	`a + b`	2.1
integer multiply	`a * b`	2.4
integer divide	`a / b`	5.4
floating-point add	`a + b`	4.6
floating-point multiply	`a * b`	4.2
floating-point divide	`a / b`	13.5
sine	`Math.sin(theta)`	91.3
arctangent	`Math.atan2(y, x)`	129.0
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Estimando o tempo de execução

- Analisa como a entrada influencia a execução do programa (definindo ou alterando o número de operações que são executadas)

- Define-se as operações primitivas que serão consideradas
  - Importante: operações mais frequentes

- Aproximamos a frequência de chamadas dessas operações por uma função conhecida

Exemplo 1

- Em função dos dados de entrada, quantas instruções são executadas?

```
1. for( int i = 0; i < n; i++)
2.     vetor[i] = i + 1;
```

Operação	Frequência
Alocação de variável	1
Atribuição	N+1
Comparação menor	N+1
Incremento	N
Soma	N
Acesso de vetor	N

Exemplo 2

- Em função dos dados de entrada, quantas instruções são executadas?

```
1. int count = 0;
2. for(int i = 0; i < n; i++)
3.     if( vetor[i] == 0 )
4.         count++;
```

Operação	Frequência
Alocação de variável	2
Atribuição	2
Comparação menor	N+1
Incremento	N até 2*N
Comparação igual	N
Acesso de vetor	N

Considerando “+=” como sendo:
uma soma e uma atribuição

Exemplo 3

- Em função dos dados de entrada, quantas instruções são executadas?

```
1. int count = 0;
2. for( int i = 0; i < n; i++)
3.     for( int j = n; j > 0; j--)
4.         count += i + j;
```

Operação	Frequência
Alocação de variável	N+2
Atribuição	N*N+N+2
Comparação menor	N+1
Comparação maior	N*(N+1)
Incremento	N
Soma	2*N*N
Decremento	N*N

Exemplo 4

- Em função dos dados de entrada, quantas instruções são executadas?

```
1. int count = 0;
2. for(int i = 0; i < n; i++)
3.    for(int j=i+1; j < n; j++)
4.       if( vetor[i] + vetor[j] == 0 )
5.          count++;
```

Operação	Frequência
Alocação de variável	N+2
Atribuição	N+2
Comparação menor	((N+1)*(N+2))/2
Comparação igual	(N*(N-1))/2
Acesso de vetor	N*(N-1)
Incremento	(N*(N+1))/2 até N*N

Contando operações primitivas

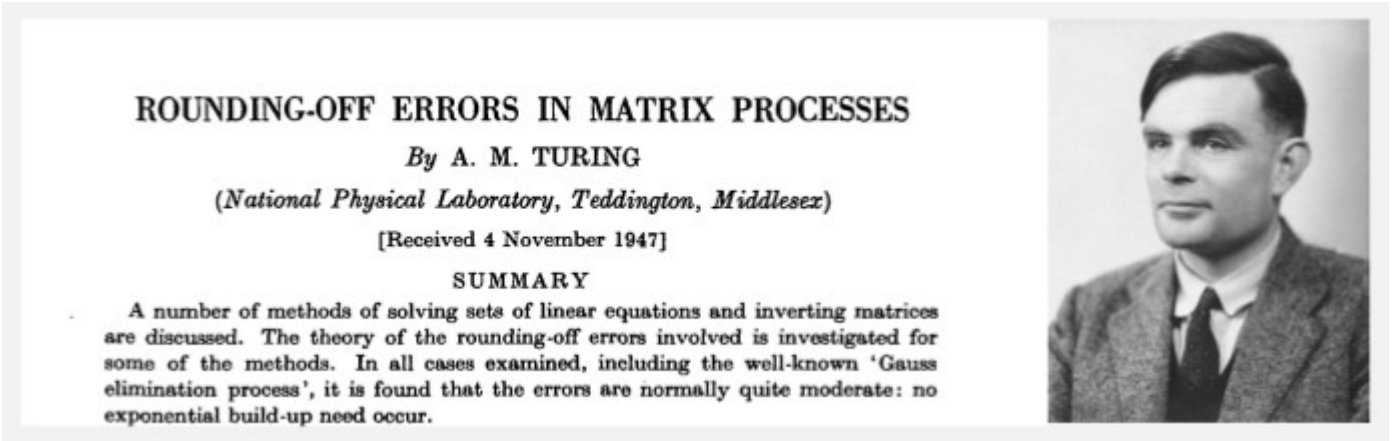
- Alternativamente, podemos contar o número total de operações primitivas por linha

- Em ambos os casos, estas estratégias de contagem são: **tediosas**!

- Vamos aproximar o custo ...

Simplificando

"It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings." - Alan Turing



ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

Primeira simplificação

- **Modelo de custo**: escolhemos apenas uma operação primitiva para calcular
  - A mais frequente!

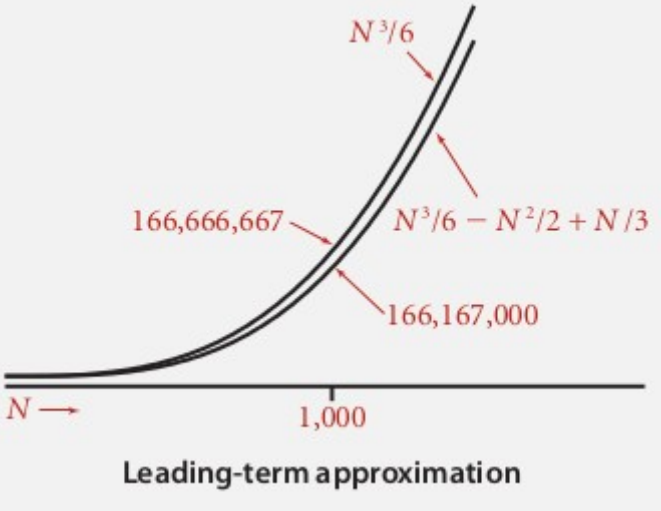
- Exemplos:
  - Busca: podemos considerar apenas as comparações, ou então, apenas os acessos ao vetor
  - Ordenação: considerar apenas as comparações, ou apenas acessos ao vetor, ou apenas o número de permutações, ...

Segunda simplificação

- Estimar o tempo de execução (ou memória) por uma **função** em relação ao tamanho da entrada
- Notação **til**
  - Ignorar os termos de menor ordem

    Exemplos:

$1/6\ N^3 + 20\ N + 16$	$\sim 1/6\ N^3$
$3N^2 + 100\ N + 3$	$\sim 3\ N^2$
$1/6\ N^3 - ½\ N^2 + 1/3\ N$	$\sim 1/6\ N^3$



Leading-term approximation

Technical definition. $f(N) \sim g(N)$ means $\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$

Notação **til**

- Quando o N é **grande**: os termos de menor ordem não mudam muito o resultado
- Quando o N é **pequeno**: o peso dos termos de menor ordem pode ser significativo, mas … a entrada é pequena!
- O interesse maior está na ordem de crescimento e não em valores absolutos!

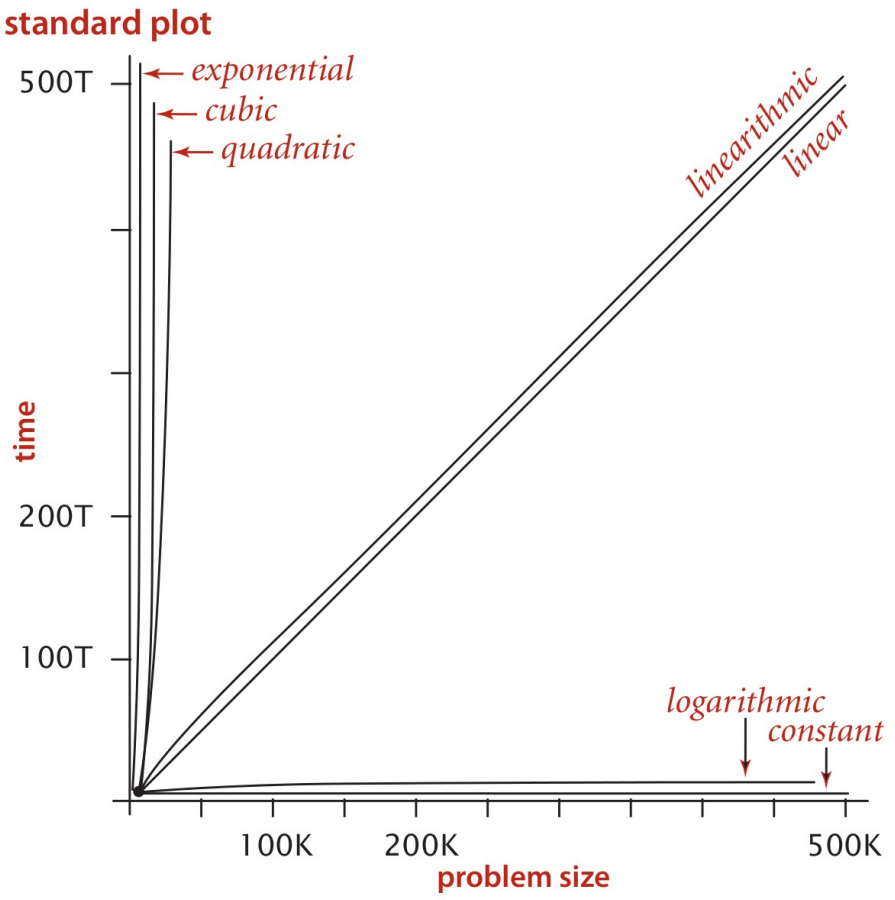
operation	frequency	tilde notation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2} (N + 1) (N + 2)$	$\sim \frac{1}{2} N^2$
equal to compare	$\frac{1}{2} N (N - 1)$	$\sim \frac{1}{2} N^2$
array access	$N (N - 1)$	$\sim N^2$
increment	$\frac{1}{2} N (N - 1)$ to $N (N - 1)$	$\sim \frac{1}{2} N^2$ to $\sim N^2$

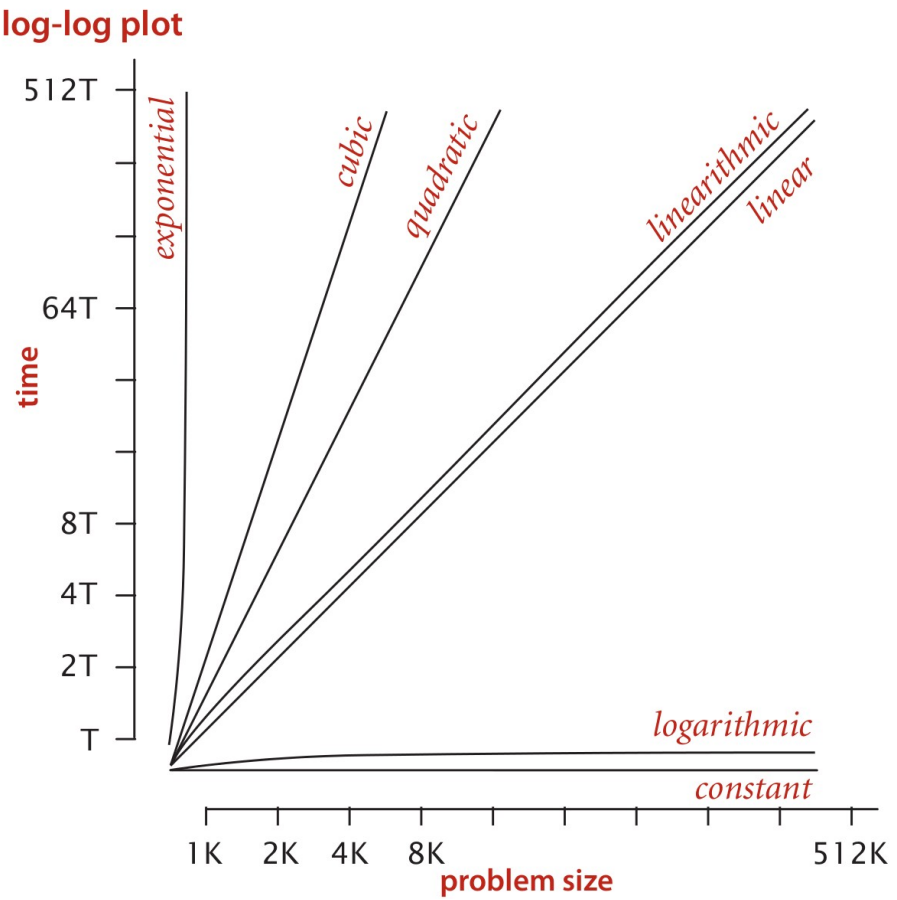
Ordem de crescimento

- Surpreendentemente, é frequente apenas um pequeno conjunto de funções:

  - Constante ~ 1
  - Logarítmica ~ logN
  - Linear ~ N
  - Linear-Logarítmica ~ N*logN
  - Quadrática ~ N2
  - Cúbica ~ N3
  - Exponencial ~ 2N

Ordem de crescimento





Common order-of-growth classifications

order of growth	name	typical code framework	description	example	T(2N) / T(N)
1	constant	`a = b + c;`	statement	add two numbers	1
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	T(N)

Practical implications of order-of-growth

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
log N	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
N log N	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

Bottom line. Need linear or linearithmic alg to keep pace with Moore's law.

Exercícios ...

- Considerando como modelo de custo a operação executada com maior frequência, forneça a ordem de crescimento do número de execuções em relação a variável ***n*** de cada código a seguir:

```
for (i=n; i>0; i=i-2)
  x = x + i;
```

Exercícios ...

- Considerando como modelo de custo a operação executada com maior frequência, forneça a ordem de crescimento do número de execuções em relação a variável ***n*** de cada código a seguir:

```
for (i=0; i<n; i++)
  for (j=n; j>0; j--)
    x = x + j;
```

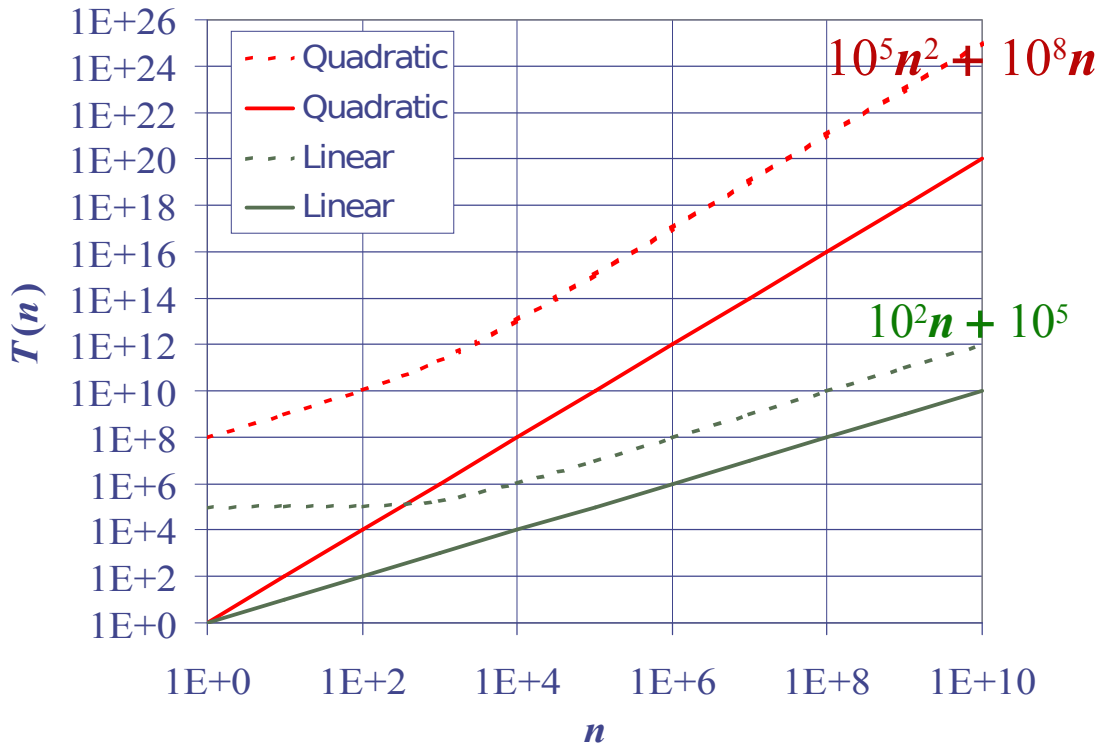
Exercícios ...

- Considerando como modelo de custo a operação executada com maior frequência, forneça a ordem de crescimento do número de execuções em relação a variável ***n*** de cada código a seguir:

```
for (i=1; i<n; i*=2)
  x = x + i;
```

Ordem de crescimento

- A taxa de crescimento não é afetada por:
  - Fatores constantes
  - Termos de ordem menor



Tipos de análises

- Melhor caso
  - Determinado pelos dados de entrada que levam ao menor número de passos
  - Provê um limitante inferior, ou seja, o menor número de passos que o algoritmo executará
- Pior caso
  - Determinado pelos dados de entrada que levam ao maior número de passos
  - Provê um limitante superior, ou seja, considerando qualquer entrada possível, este será o maior tempo necessário de processamento

Tipos de análises

- Caso médio
  - Custo esperado em médio (muitas vezes difícil de definir!)

- Abordagens:
  - Apenas a avaliação do pior caso é necessária
    - Busca sequencial: elemento não encontrado!
  - Apenas o caso médio é importante
    - Quicksort
  - Todos os os casos precisam ser avaliados