

Estruturas de Dados 1

481440

Abril/2018

Mario Liziér
lazier@ufscar.br

Algumas perguntas ...

- Em quanto tempo terei a resposta?
- Porque o meu programa está tão lento?
- Quanto de memória vou precisar?
- Consigo executar este algoritmo com muitos dados?
- Se eu comprar um computador 2x mais rápido, terei minha resposta em tempo hábil?
- Vale a pena comprar mais memória?
- Em uma aplicação prática terei 10x mais dados, este algoritmo será útil?



*"People who **analyze algorithms** have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more **quickly** and more **economically**." - Donald Knuth*

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)
- ...

Exemplo

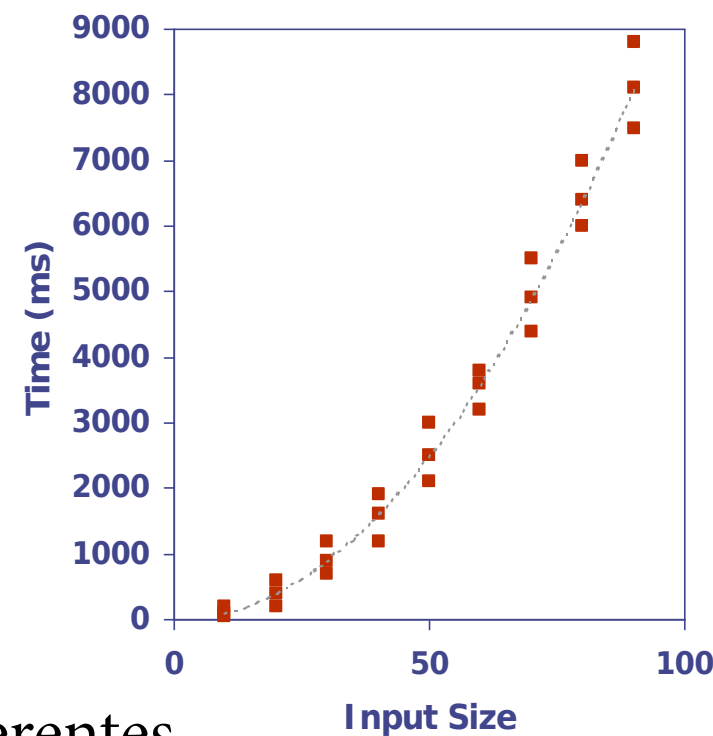
- Busca em uma sequência ordenada:
 - Entrada: uma sequência de n números ($a_0, a_1, a_2, \dots, a_{n-1}$) ordenados, ou seja, $a_0 \leq a_1 \leq a_2 \leq \dots \leq a_{n-1}$, e um número *key* a ser procurado;
 - Saída: índice da posição de *key* na sequência de entrada, ou o valor -1 caso *key* não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

Algumas questões

- Considerando o primeiro algoritmo:
 - Em quanto tempo encontramos um elemento em uma entrada com 10 números?
 - E se dobrarmos o tamanho desta entrada, executando com 20 números?
 - Qualquer entrada com 1000 números demoram X segundos?
 - O consumo de memória aumenta em que proporção? Varia de acordo com o tamanho da entrada ou os valores presentes?
 - Um computador 2x mais rápido levará metade do tempo para encontrar um elemento?
- O segundo algoritmo é melhor que o primeiro? Sempre?
- Para qual tamanho de entrada o primeiro algoritmo é mais adequado que o segundo?

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



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;
}
```

Busca binária

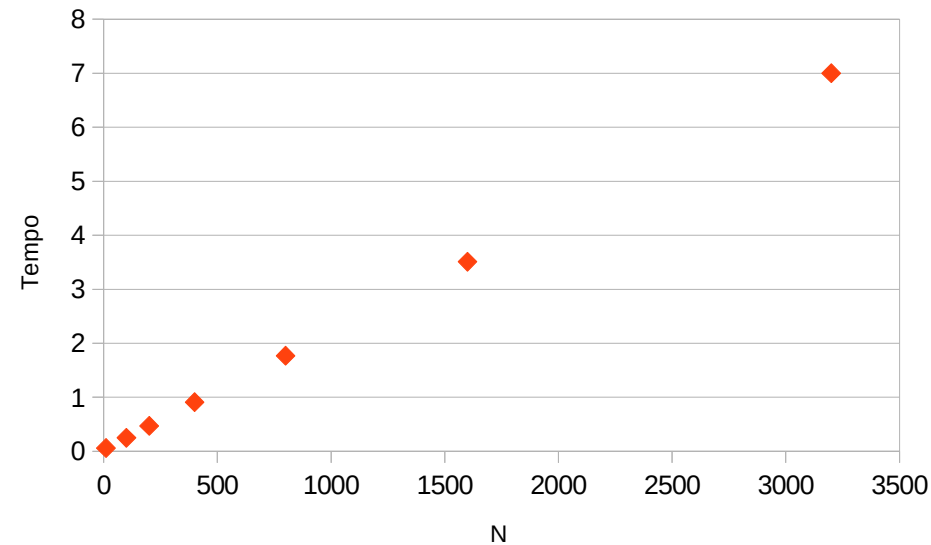
- Ideia base: comparar o elemento do meio para continuar a busca na metade certa

```
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;
}
```


Modelo experimental

- Busca sequencial
 - Espaço: não muda em relação as entradas (somar bytes)
 - Tempo: (código em C)

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



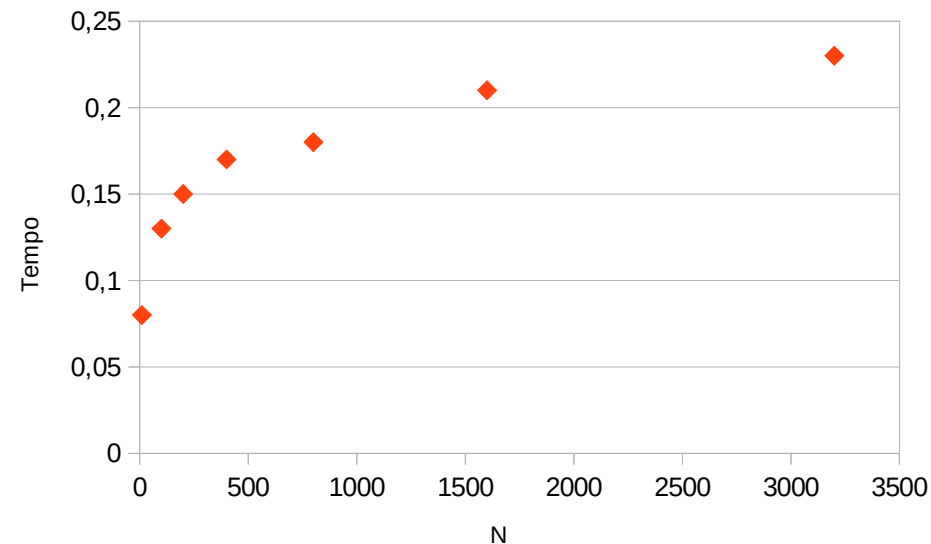
CPU: Intel Core i3-2310M – 2.1GHz

Modelo experimental

- Busca binária
 - Espaço: não muda
 - Tempo: (código em C)

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



Considerações importantes

Modelo experimental

- Método aparentemente “simples”
- **Difícil** conseguir bons testes
- Não fornece qualquer garantia teórica
- Necessidade de realizar todos as 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 o expoente do 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;  
}
```

costs (depend on machine, compiler)

$$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

A = array access

B = integer add

C = integer compare

D = increment

E = variable assignment

frequencies
(depend on algorithm, input)

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	<code>Math.sin(theta)</code>	91.3
arctangent	<code>Math.atan2(y, x)</code>	129.0
...

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

Cost of basic operations

operation	example	nanoseconds [†]
variable declaration	<code>int a</code>	C_1
assignment statement	<code>a = b</code>	C_2
integer compare	<code>a < b</code>	C_3
array element access	<code>a[i]</code>	C_4
array length	<code>a.length</code>	C_5
1D array allocation	<code>new int[N]</code>	$C_6 N$
2D array allocation	<code>new int[N][N]</code>	$C_7 N^2$
string length	<code>s.length()</code>	C_8
substring extraction	<code>s.substring(N/2, N)</code>	C_9
string concatenation	<code>s + t</code>	$C_{10} N$

Novice mistake. Abusive string concatenation.

Estimando o tempo de execução

- Define-se as operações primitivas que serão consideradas
 - Importante: operações mais frequentes
- Consideraremos constante o tempo de execução de cada operação primitiva
- Aproximamos por uma função conhecida estes valores
- Casos para avaliar:
 - Melhor caso
 - Pior caso
 - Caso médio

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!**
- Será mesmo que são necessárias?

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
- 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

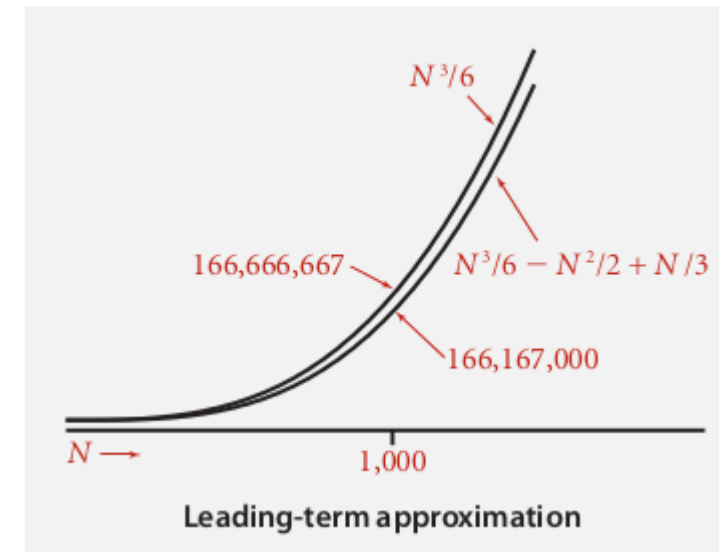
- Estimar o tempo de execução (ou memória) por uma **função** em relação a entrada
- Notação **til**
 - Ignorar os termos de menor ordem

Exemplos:

$$\frac{1}{6} N^3 + 20 N + 16 \quad \sim \frac{1}{6} N^3$$

$$3N^2 + 100 N + 3 \quad \sim 3 N^2$$

$$\frac{1}{6} N^3 - \frac{1}{2} N^2 + \frac{1}{3} N \quad \sim \frac{1}{6} N^3$$



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

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!

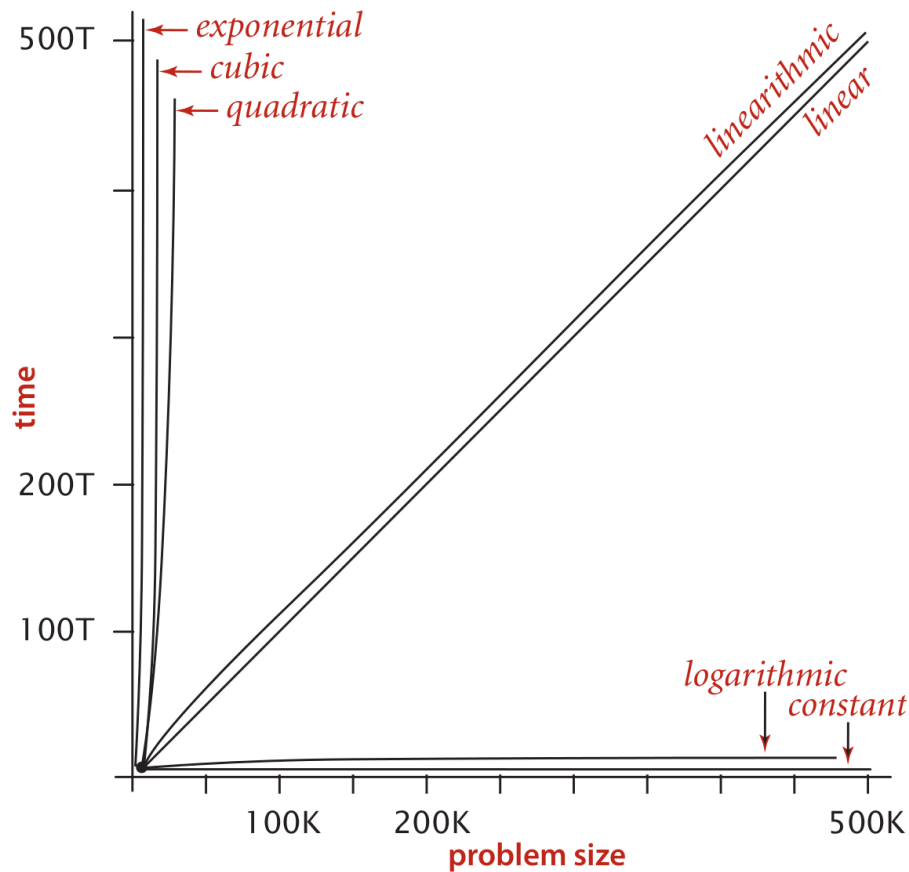
operation	frequency	tilde notation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2} (N + 1) (N + 2)$	$\sim \frac{1}{2} N^2$
equal to compare	$\frac{1}{2} N (N - 1)$	$\sim \frac{1}{2} N^2$
array access	$N (N - 1)$	$\sim N^2$
increment	$\frac{1}{2}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

Ordem de crescimento

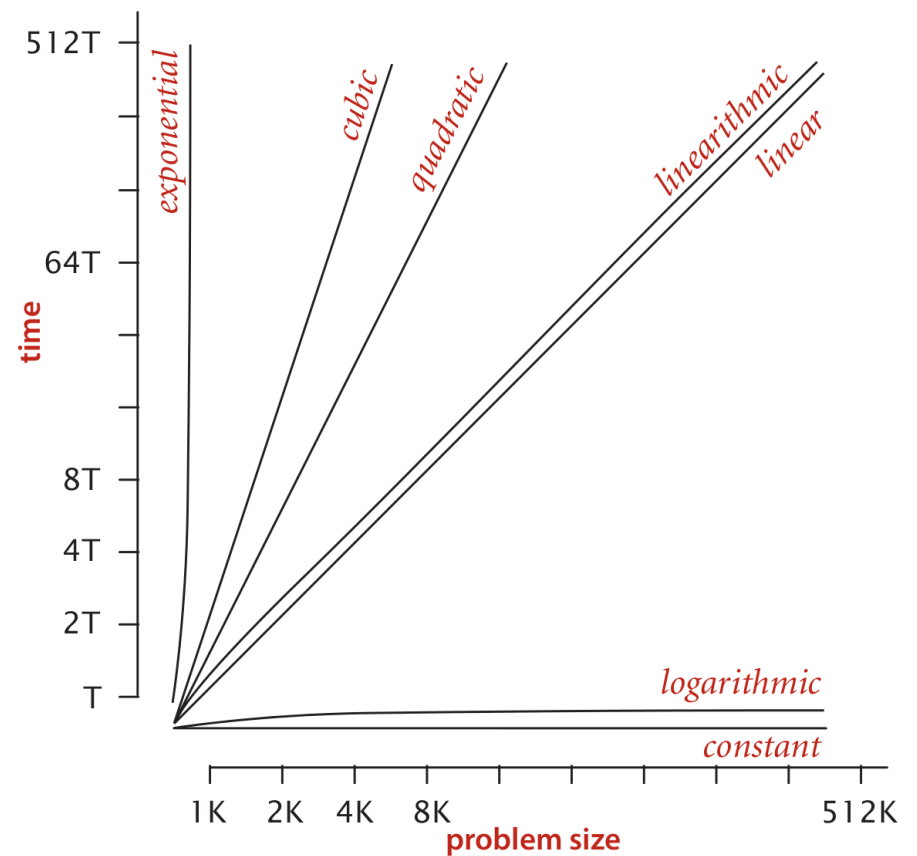
- Conjunto pequeno de funções
 - Constante ~ 1
 - Logarítmica $\sim \log N$
 - Linear $\sim N$
 - Linear-Logarítmica $\sim N^* \log N$
 - Quadrática $\sim N^2$
 - Cúbica $\sim N^3$
 - Exponencial $\sim 2^N$

Ordem de crescimento

standard plot



log-log plot



Common order-of-growth classifications

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

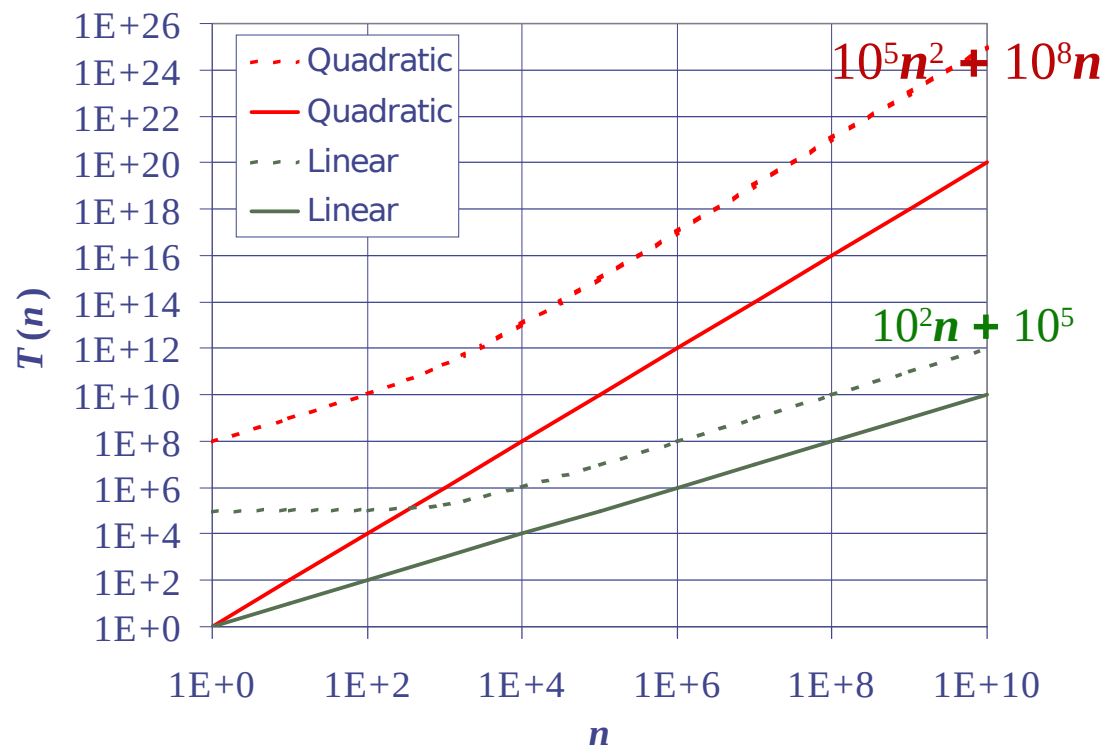
Exercícios ...

- 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

Análise assintótica

- Útil quando analisamos para N **muito** grande
- Todos os fatores constantes e termos de ordem inferior são eliminados
- Aproximamos a função do modelo de custo por uma das funções “básicas”
- Criação de famílias de funções: constantes, logarítmicas, lineares, quadráticas, exponenciais, etc ...

Notação O

Big-Oh

Obs: Operador “ ϵ ” e “=”
são utilizados como
Equivalentes!

- O conjunto de **funções** $O(g(n))$ é definido como:

$O(g(n)) = \{ f(n) : \text{existem } \mathbf{constantes} \text{ positivas } c \text{ e } n_0, \text{ tal que } 0 \leq f(n) \leq c g(n), \text{ para todo } n \geq n_0 \}$

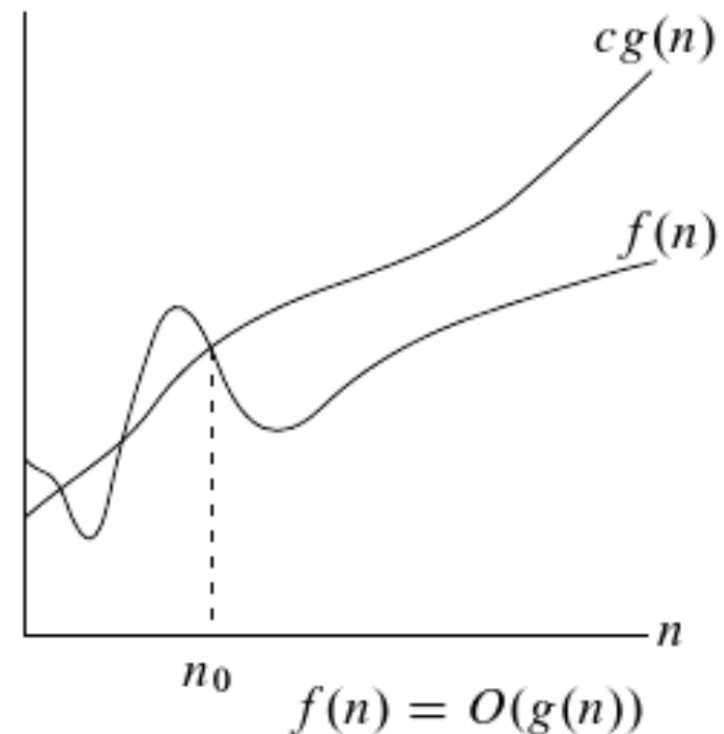
- Exemplos:

$$n^2 + n + 10 \in O(n^2)$$

$$3n^3 + n^2 \in O(n^3)$$

$$50 \log N + 30 \in O(\log N)$$

$$\frac{1}{3} N \log N + 4N \in O(N \log N)$$



Notação Ω

Big-Omega

- O conjunto de **funções** $\Omega(g(n))$ é definido como:

$\Omega(g(n)) = \{ f(n) : \text{existem } \mathbf{constantes} \text{ positivas } c \text{ e } n_0, \text{ tal que } 0 \leq c g(n) \leq f(n), \text{ para todo } n \geq n_0 \}$

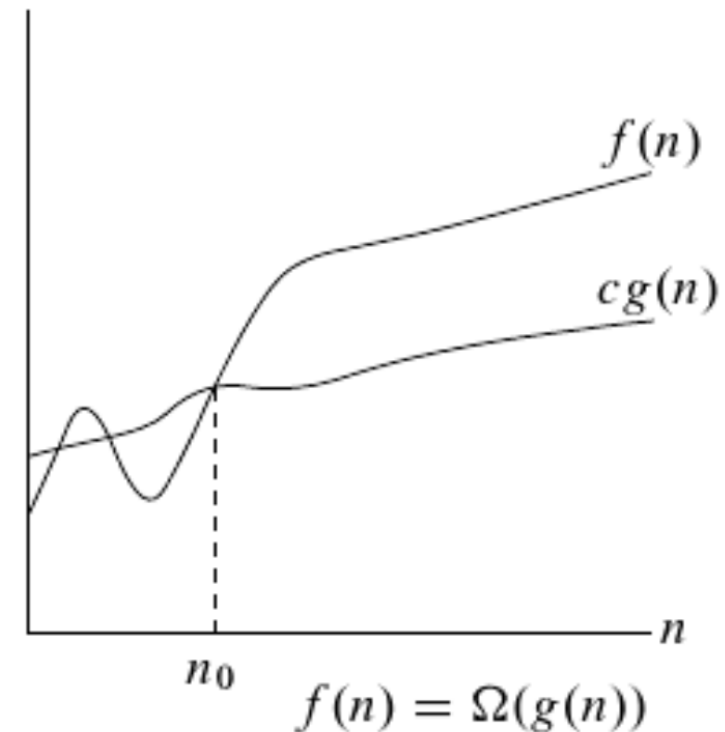
- Exemplos:

$$n^2 + n + 10 \in \Omega(n^2)$$

$$3n^3 + n^2 \in \Omega(n^3)$$

$$50 \log N + 30 \in \Omega(\log N)$$

$$\frac{1}{3} N \log N + 4N \in \Omega(N \log N)$$



Notação Θ

Theta

- O conjunto de **funções** $\Theta(g(n))$ é definido como:

$\Theta(g(n)) = \{ f(n) : \text{existem } \mathbf{constantes} \text{ positivas } c_1, c_2 \text{ e } n_0, \text{ tal que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ para todo } n \geq n_0 \}$

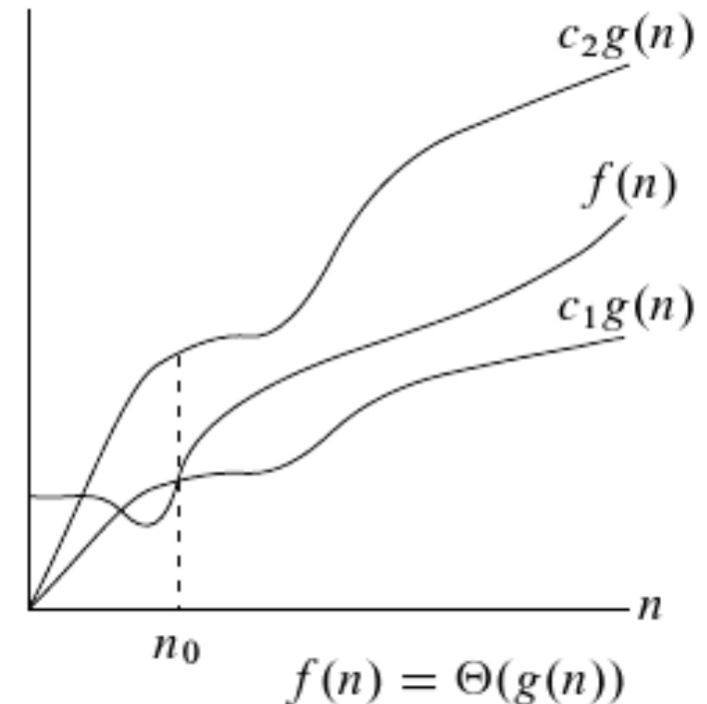
- Exemplos:

$$n^2 + n + 10 \in \Theta(n^2)$$

$$3n^3 + n^2 \in \Theta(n^3)$$

$$50 \log N + 30 \in \Theta(\log N)$$

$$\frac{1}{3} N \log N + 4N \in \Theta(N \log N)$$



Notação Θ

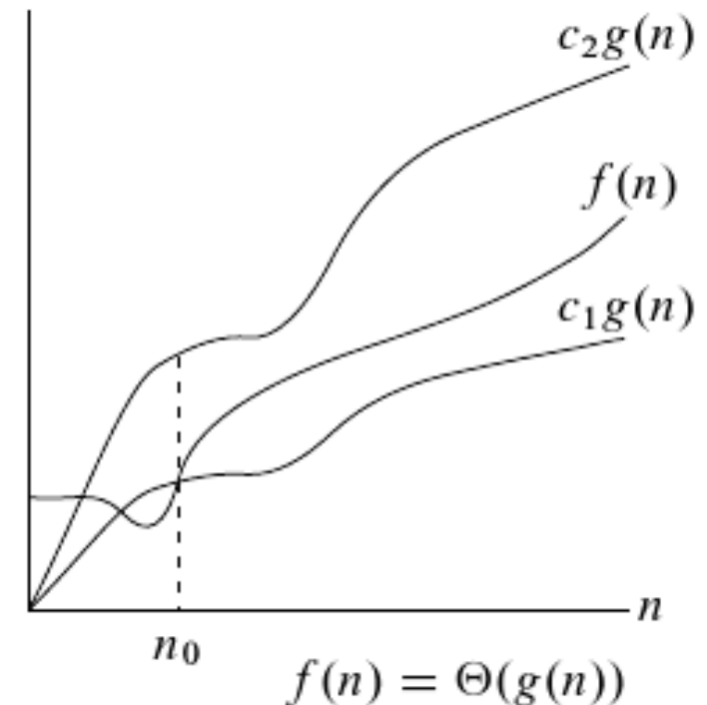
Theta

- Outra definição:

Para quaisquer funções $f(n)$ e $g(n)$,

temos $f(n) = \Theta(g(n))$, se e somente se,

$f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.



Exemplo:

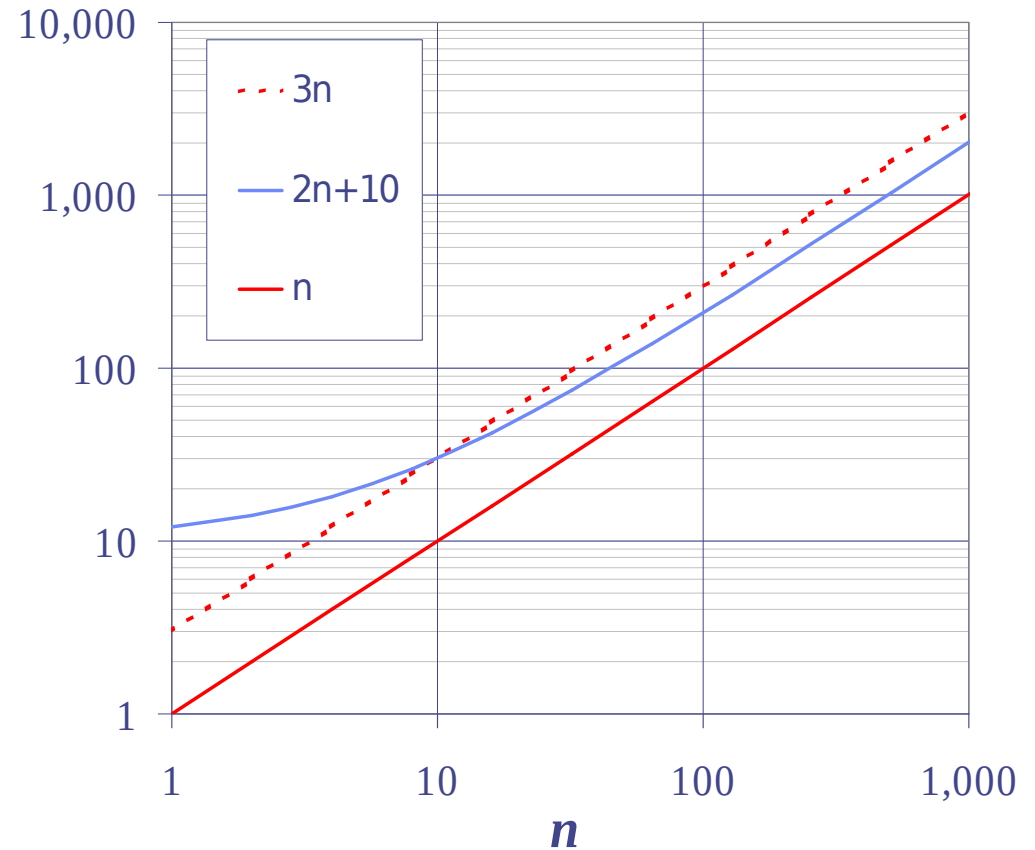
$$2^*n + 10 = O(n)$$

$$2^*n + 10 \leq c^*n$$

$$(c-2)^*n \geq 10$$

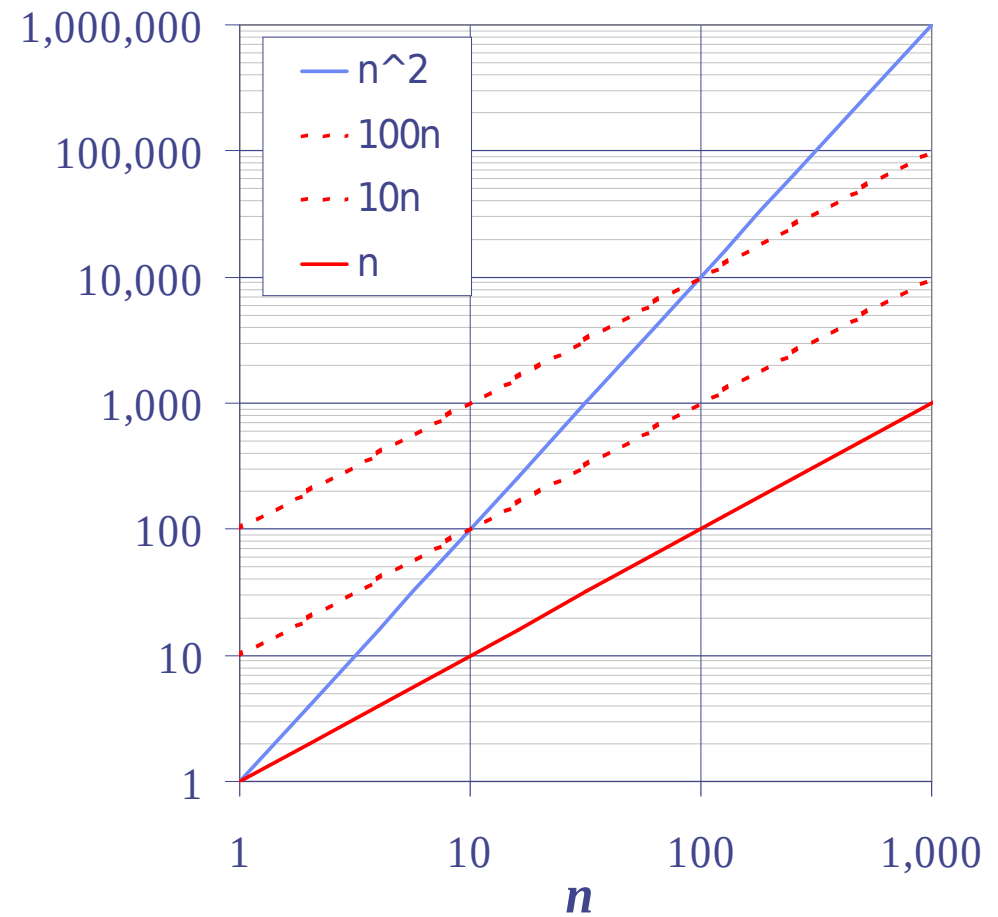
$$n \geq 10/(c-2)$$

$$c = 3 \text{ e } n_0 = 10$$



Exemplo:

$$n^2 \neq \Theta(n)$$



Observações sobre O , Ω e Θ

- Embora possam parecer “redundantes”, cada problema requer uma notação
- Ao avaliar o pior caso, a utilização de O geralmente é mais apropriada
- Ao avaliar o melhor caso, a utilização de Ω pode ser mais apropriada
- O uso de Θ já indica que conseguimos limitar “por baixo” e “por cima” a função
- As minúsculas o e ω indicam que uma aproximação muito fraca foi feita, para O e Ω respectivamente.

Exemplos o e ω

$$10N^2 + n + 5 = o(N^3) = O(N^2)$$

$$10N^2 + n + 5 = \omega(N) = \Omega(N^2)$$

- Reparem que pelas definições aqui apresentadas de O , Ω e Θ , *estas possibilidades de “péssima” aproximações não estavam sendo consideradas!*

Análise de algoritmos

- Devemos utilizar a família de função que mais se “ajusta” a função que devemos aproximar

Forte!

- Logo, é errado dizer que:

$$n^2 = O(n^3)$$

$$n^2 = \Omega(n)$$

Propriedades relacionais

Transitivity:

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$,
 $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$,
 $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$,
 $f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$,
 $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$.

Propriedades relacionais

Reflexivity:

$$f(n) = \Theta(f(n)) ,$$

$$f(n) = O(f(n)) ,$$

$$f(n) = \Omega(f(n)) .$$

Propriedades relacionais

Symmetry:

$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Transpose symmetry:

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$,

$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Modelo matemático

- Teoria: temos disponível uma modelagem matemática precisa, com maior rigor
- Prática:
 - As fórmulas podem se tornar muito complicadas
 - Exigirá conhecimento mais avançados de matemática
 - Utilizaremos **aproximações**

Considerações finais

- Os dois modelos são importantes!
 - O modelo matemático é independente do sistema; utiliza uma máquina que ainda não foi construída!
 - O modelo experimental (análise empírica) é necessária para validar o modelo matemático e fazer um prognóstico futuro

Exemplo: busca sequencial

- Aproximadamente quantas comparações são realizadas?

```
int bsequencial( T vetor[], T key, int n )
{
    int i = 0;
    while( (i < n) && (vetor[i] < key) )           1 até 2*N+1
        i++;
    if ( (i < n) && (vetor[i] == key) )             1 até 2
        return i;
    else
        return -1;
}
```

Total: 2 até 2*N+2 ~2*N

Exemplo: busca sequencial

- Melhor caso: quando **key** está na posição 0
2 comparações: $f(n)$ é constante!
- Pior caso: quando **key** não está presente
 $\sim 2 \cdot N$ comparações: $f(n)$ é linear!
- Caso médio: dependerá da aplicação, mas considerando algo totalmente randômico, teríamos $\sim N$ comparações, ou seja, linearmente dependente do tamanho do vetor
- Neste caso, é apropriado utilizar $O(N)$ para expressar a complexidade deste algoritmo

Exemplo: busca binária

- Aproximadamente quantas comparações são realizadas?

```
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: busca binária

- Melhor caso: o elemento **key** está no meio do vetor: com um número constante de comparações encontramos! (exatamente 3)
- Pior caso: quando não encontramos um elemento, o *loop* executa “por completo”, pois não é interrompido no *return imid*;

Exemplo: busca binária

- Pior caso:

Descemos toda a árvore binária, ou seja, um número de comparações proporcional a $\log_2 N$

Memória

- Além do tempo de processamento, o a função de custo pode modelar o consumo de memória
- Os dois algoritmos apresentados (busca sequencial e binária) não requerem memória proporcional as entradas (constante), excluindo o próprio vetor de dados, que cresce linearmente em relação ao número de elementos
- No exemplo a seguir temos uma versão recursiva da busca binária:

Busca binária: versão recursiva

- Qual a taxa de crescimento do uso de memória?

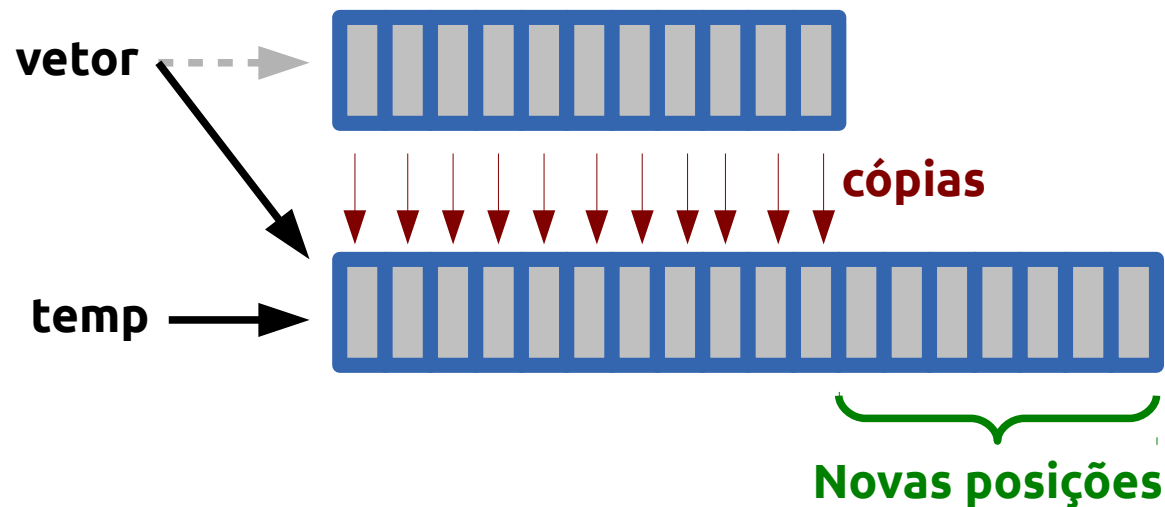
```
int bbinariarec( T vetor[], T key, int imin, int imax)
{
    if (imax >= imin)
    {
        int imid = imin + ((imax - imin) / 2);
        if ( key > vetor[imid] )
            return bbinariarec(vetor, key, imid+1, imax);
        else if ( key < vetor[imid] )
            return bbinariarec(vetor,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

Arrays: como ED

- Primeiro modo que aprendemos: *fácil*
- Alocação de uma região contígua da memória
- Definição do tamanho na compilação ou execução
- Desvantagem:
 - Tamanho fixo, redimensionar exige realocação
 - Posições fixas, reordenação exige trocas de posições
- Vantagem:
 - Acesso direto a todos os elementos (posições)
 - Alocação de memória apenas para armazenar os dados

Redimensionamento de *arrays*

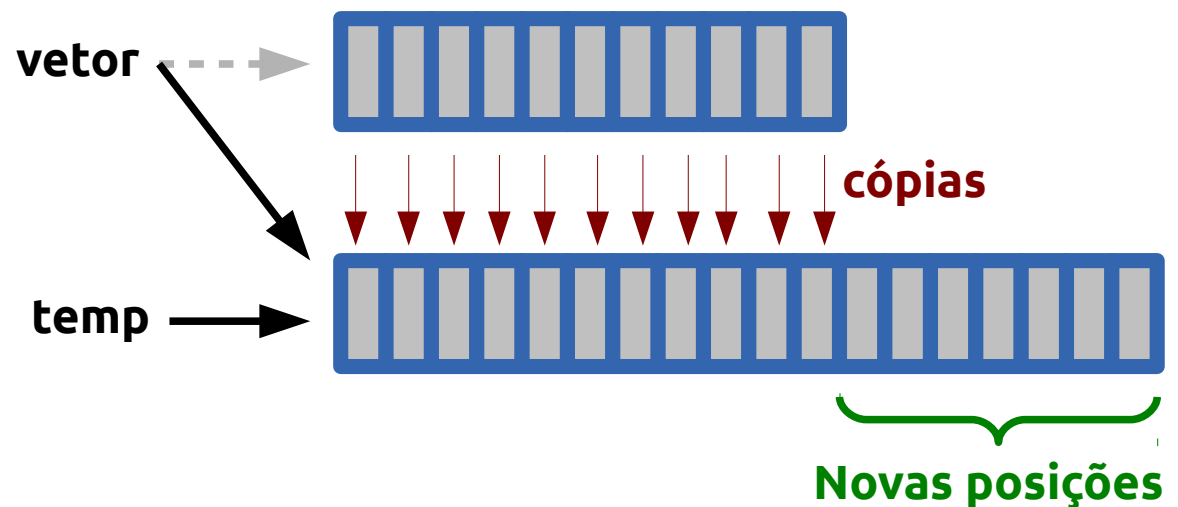
- Como aumentar um array para armazenar mais elementos?



- Como diminuir um array para liberar memória não utilizada?

Redimensionamento de *arrays*

- Solução:
 - Alocar um novo array, incluindo o espaço adicional necessário
 - Copiar os elementos do antigo array no novo
 - Liberar o espaço do antigo array
- Boa estratégia:
 - Vamos pensar em uma implementação de pilha em array



Redimensionamento de *array*

- Primeira implementação (“ingênua”):
 - Se a pilha estiver cheia: push aumenta o tamanho do *array* em 1
 - Custo de inserir N elementos na pilha:
 - Acessos ao vetor: $\sim N^2/2$
inserções: $1+1+1+1+1+\dots+1 = N$
realocações: $0+1+2+3+4+\dots+N-1 = \sim N^2/2$
 - A operação de redimensionar é cara: $\Theta(N)$
 - Desafio: garantir que ela não será sempre executada!

Redimensionamento de *array*

- Solução:
 - Se a pilha estiver cheia: push **dobra** o tamanho do *array*
 - Custo de inserir N elementos na pilha:
 - Acessos ao vetor: $\sim 3N$
inserções: $1+1+1+1+\dots+1 = N$
realocações: $0+1+2+0+4+0+\dots+8+\dots+16+\dots+32+\dots+N = \sim 2N$
 - A operação de redimensionar é executada apenas $\log N$ vezes!

Redimensionamento de *array*

```
template <class Item>
class Pilha
{
    private:
        Item *s; int N; int Nmax;

        void resize(int max)
        {
            Item[] temp = new Item[max];
            for (int i = 0; i < N; i++)
                temp[i] = s[i];
            delete s;
            s = temp;
            Nmax = max;
        }

    public:
        Pilha( ) { // tamanho inicial?
            s = new Item[2];
            N = 0;
            Nmax = 2;
        }
        // Construtor por cópia
        // Operador atribuição
        // Destrutor
};
```

```
    bool empty() const {
        return N == 0;
    }

    void push(Item item) {
        if( N == Nmax )
            resize(Nmax*2);
        s[N++] = item;
    }

    bool pop(Item &item) {
        // próximo slide
        ...
        item = s[--N];
        ...
    }
};
```

Redimensionamento de *array*

- Podemos também diminuir o *array* quando muitos elementos são removidos

– Má ideia:

```
bool pop(Item &item) {  
    if( N > 0 ) {  
        item = s[--N];  
        if( N>0 && N == Nmax/2)  
            resize(Nmax/2);  
        return true;  
    } else  
        return false;  
}
```

– Boa ideia:

Porque?

```
bool pop(Item &item) {  
    if ( N > 0 ) {  
        item = s[--N];  
        if( N>0 && N == Nmax/4)  
            resize(Nmax/2);  
        return true;  
    } else  
        return false;  
}
```

Análise amortizada de custo

- Muito útil em estruturas de dados
- Considera-se o custo total de uma sequência de operações, dividindo-se pelo número de operações executadas
- Nesta análise, uma operação pode ser muito custosa em um pior caso, mas se soubermos que em M chamadas desta operação o pior caso não acontecerá em todas as vezes, não podemos dizer que teríamos $(M \cdot \text{Pior caso})/M$
- Por exemplo: redimensionamento de *arrays*
 - Push:
 - Pior caso: $O(N)$
 - Melhor caso: $\Omega(1)$
 - Pop:
 - Pior caso: $O(N)$
 - Melhor caso: $\Omega(1)$

Análise amortizada de custo

- Por exemplo: redimensionamento de *arrays*
 - Push:
 - Pior caso: $O(N)$
 - Melhor caso: $\Omega(1)$
 - Pop:
 - Pior caso: $O(N)$
 - Melhor caso: $\Omega(1)$
- Qualquer sequência de chamadas **push** e/ou **pop** (em qualquer ordem ou alternância), temos:
 - Push: caso amortizado $\Theta(1)$
 - Pop: caso amortizado $\Theta(1)$

Pilhas - Implementações

- Encadeamento:
 - Cada operação utiliza um tempo **constante** no pior caso
 - Utiliza o **espaço extra** dos ponteiros
 - Alocação por nó: chamadas de sistema
- Array (sem redimensionamento):
 - Cada operação utilizará um tempo **constante** no pior caso
 - Não ocupa espaço extra (possui posições não utilizadas), mas **fixo**, podendo ser insuficiente ou desperdiçar muito espaço
- Array (com redimensionamento):
 - Cada operação utilizará um tempo constante **amortizado**
 - Não ocupa espaço extra (com ponteiros, mas o vetor terá posições não utilizadas)

Referências:

- Livro do Drozdek
- Livro do Cormen
- Livro do Robert Sedgewick
 - <https://algs4.cs.princeton.edu>