Algoritmos e Estruturas de Dados

Análise de algoritmos – Capítulo 4

2019/20

Critérios de escolha

- Dado um problema, há vários programas (soluções) possíveis.

  Como escolher?

- Tempo de Execução
  - Complexidade temporal

- Memória: espaço requerido
  - Complexidade espacial

em função da dimensão do problema.

Complexidade

- Empírica
  - Medir o tempo/espaço com dados reais.
- Por Simulação
  - Medir o tempo/espaço com dados gerados de acordo com determinadas distribuições.
- Analítica
  - Determinar uma função que estima a ordem de grandeza do tempo/espaço

Complexidade Temporal
Análise Experimental (1)

- Estudar o tempo de execução de forma experimental:
  - Executar conjuntos de testes e registar o tempo em cada execução;
  - Determinar a dependência geral entre a dimensão do input e o tempo de execução;
  - Os resultados permitem depois executar análises estatísticas, de forma a adaptar os resultados a funções conhecidas (linear, quadrática, logaritmica, exponencial, …).

- Vejamos um exemplo “Concatenação de Strings” (capítulo 4, pag 152)

COMBINANDO
PALAVRAS

Complexidade Temporal

Análise Experimental (2)

```
/** Uses repeated concatenation to compose a String with n copies of character c. */
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}
```

```
/** Uses StringBuilder to compose a String with n copies of character c. */
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder();
  for (int j=0; j < n; j++)
    sb.append(c);
  return sb.toString();
}
```

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

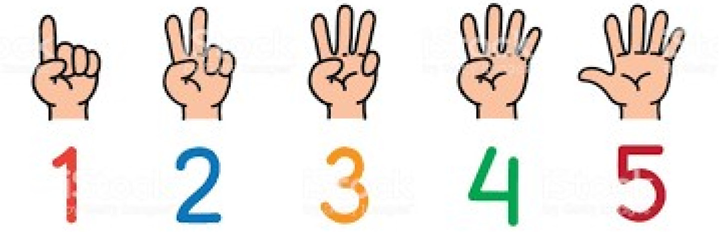
Table 4.1: Results of timing experiment on the methods from Code Fragment 4.2.

Complexidade Temporal
Análise Experimental (3)

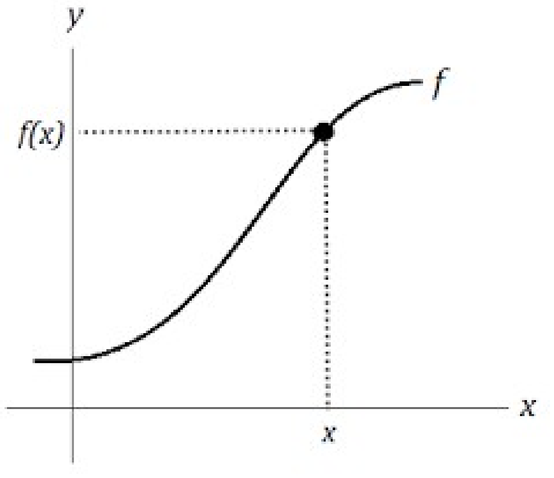
- Desvantagens:
  - Só considera um conjunto limitado de inputs de teste;
  - Dificulta a comparação entre resultados experimentais de algoritmos diferentes, se estes forem executados em hardware e software diferente;
  - O algoritmo tem de ser implementado para que o estudo experimental seja feito.

Complexidade Temporal

- Em vez de contar o tempo de execução, vamos:
  - estudar diretamente as instruções que compõem o algoritmo em análise. A base para isso é <u>contar</u> o número de ***Operações Primitivas*** que são executadas para um determinado input do algoritmo;



  - associar a cada algoritmo uma função **f(n)** que caracteriza o comportamento do mesmo (com base no número de operações primitivas realizadas), a partir da dimensão do input, dado pela variável n.



Complexidade Temporal
Algumas operações primitivas

- As operações primitivas são:
  - Afetacões;
  - Comparações;
  - Operações aritméticas;
  - Leituras e Escritas;
  - Acesso a vectores, campos de registos ou variáveis;
  - Chamadas e Retornos de funções/métodos

- Na análise do algoritmo, assumismo que:
  As operações primitivas custam 1 unidade de tempo.

- O tempo real de execução dum algoritmo será proporcional ao número de operações primitivas que este executa.

Complexidade Temporal
Contagem de operações primitivas

$$\left.\begin{array}{l}\text{Afectações, Comparações,}\\ \text{Operações aritméticas, Leituras e Escritas,}\\ \text{Acessos a campos de vectores ou a variáveis,}\\ \text{Chamadas e Retornos de métodos, } \ldots,\end{array}\right\}$$

1 unidade de tempo

if C **then** S_1 **else** S_2 $\qquad T_C + max(T_{S^1} ; T_{S^2})$

while C **do** S $\qquad K *(T_C + T_S)$ ***K número de vezes que o ciclo while é executado***

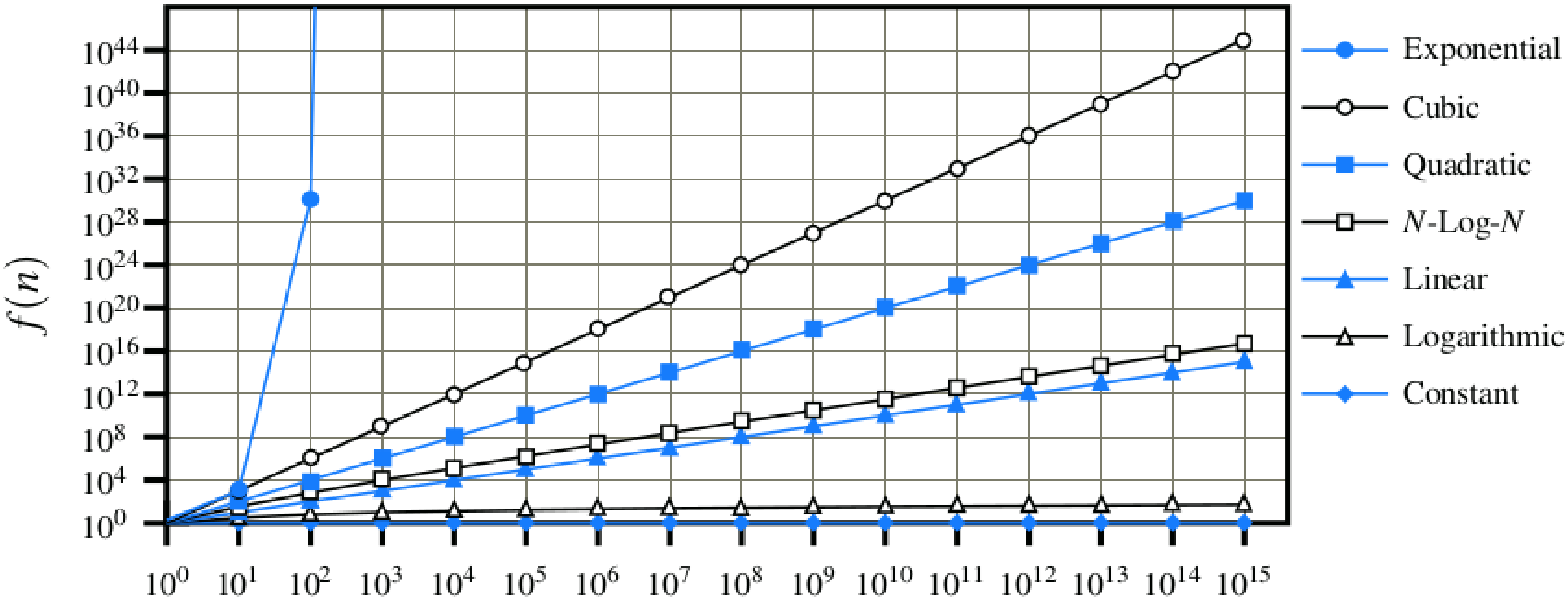
$$\left.\begin{array}{l}\textbf{switch } E\\ \textbf{case } V_1 : S_1\\ \ldots\\ \textbf{case } V_j : S_j\end{array}\right\} \qquad T_E + max(T_{S^1} ; T_{S^j})$$

Complexidade
Algumas funções típicas (1)

- ***T (n)*** — ordem de grandeza do tempo/espaço requerido para um problema de dimensão n.

$$\left.\begin{array}{ll} 1 & \text{constante} \\ \log n & \text{logarítmica} \\ n & \text{linear} \\ n \log n & \\ n^2 & \text{quadrática} \\ n^3 & \text{cúbica} \\ n^4 & \\ \vdots & \end{array}\right\} \text{Polinomial} \qquad \left.\begin{array}{l} 2^n \\ 3^n \\ n! \\ \vdots \end{array}\right\} \text{Exponencial}$$

Complexidade
Algumas funções típicas (2)



- <u>Idealmente</u>, gostaríamos ter funções: constante, logaritmo, linear ou n-log-n.
- Funções quadráticas ou cúbicas são menos desejadas, mas funções exponenciais são inviáveis, excepto se soubermos que a dimensão do input é pequena.

Complexidade Temporal

Contagem de operações primitivas (1)

Pesquisa Sequencial em vector ordenado (dimensão n)

```
@requires vector v está ordenado
public static int sequentialSearch(int[] v, int n, int aim){

    int  i=0;

    while ( i < n && v[i] < aim )
       i++;

    if ( i < n && v[i] == aim )
       return i;

    return -1;
}
```

(1) 2 *unidades de tempo*

(2) *K * (8 + 2) unidades de tempo,*
K número de iterações

(3) 8 + 1 *unidades de tempo*

(4) *1 unidade de tempo*

Complexidade Temporal
Contagem de operações primitivas (2)

- Para calcular a ordem de grandeza da complexidade, <u>ignoram-se as constantes</u>.

```
public static int sequentialSearch(int[] v, int n, int aim){
   int  i=0;
   while ( i < n && v[i] < aim )
      i++;
   if ( i < n && v[i] == aim )
      return i;
   return -1;
}
```

*K * (8 + 2) unidades de tempo,*
K número de iterações

Logo, temos que saber qual é o K ?

Ou seja contar o número de células/casas visitadas no vector

Complexidade Temporal
Contagem de operações primitivas (3)

- Pesquisa Sequencial em vector ordenado (dimensão n)

4	7	13	19	24	29	30	32	41
0	1	2	3	4	5	6	7	8

```
@requires vector v está ordenado
public static int sequentialSearch(int[] v, int n, int aim){

   int  i=0;
   while ( i < n && v[i] < aim )
      i++;
   if ( i < n && v[i] == aim )
      return i;
   return -1;
}
```

Elemento procurado	Células visitadas (comparações realizadas)
4	1
3	1
19	4
23	5
41	9
50	9

Complexidade Temporal

Contagem de operações primitivas (4)

- Pesquisa Sequencial em vector ordenado (dimensão n)

4	7	13	19	24	29	30	32	41
0	1	2	3	4	5	6	7	8

Elemento procurado	Células visitadas (comparações realizadas)
4	1
3	1
19	4
23	5
41	9
50	9

Melhor caso

No(s) caso(s) "de muita sorte".

1 - constante

Pior caso

No(s) caso(s) "de muito azar".

n - linear

$$\frac{(1+2+3+4+5+6+7+8+9)}{9} \approx \frac{n}{2}$$

Caso médio (esperado)

É a média de todos os casos possíveis.

Em geral, consideram-se todos os casos equiprováveis.

Complexidade Temporal
Contagem de operações primitivas (5)

- Pesquisa Binária em vector ordenado (dimensão n)

4	7	13	19	24	29	30	32	41
0	1	2	3	4	5	6	7	8

```
public static int binSearch(int[]v, int n, int aim){
   int low=0, high=n-1, mid, current;
   boolean found=false;
   while ( low <= high && !found) {
       mid = ( low + high ) / 2;
       current = v[mid];
       if ( aim == current )
         found=true;
       else if (aim < current)
               high = mid-1;
            else
               low = mid+1;
    }
   if (found)return mid;
   return -1;
}
```

Elemento procurado	Células visitadas (comparações realizadas)
4	3 (4;1;0)
3	3 (4;1;0)
19	4 (4;1;2;3)
23	4 (4;1;2;3)
41	4 (4;6;7;8)
24	1 (4)

Complexidade Temporal
Contagem de operações primitivas (6)

- Pesquisa Binária em vector ordenado (dimensão n)

4	7	13	19	24	29	30	32	41
0	1	2	3	4	5	6	7	8

Elemento procurado	Células visitadas (comparações realizadas)
4	3 (4;1;0)
3	3 (4;1;0)
19	4 (4;1;2;3)
23	4 (4;1;2;3)
41	4 (4;6;7;8)
24	1 (4)

Pior caso

No(s) caso(s) "de muito azar".

$\log_2 n$ - logarítmica

Melhor caso

No(s) caso(s) "de muita sorte".

1 - constante

Caso médio (esperado)

É a média de todos os casos possíveis.
Em geral, consideram-se todos os casos equiprováveis.

Complexidade

- Melhor caso → No(s) caso(s) “de muita sorte”.
- Pior caso → No(s) caso(s) “de muito azar”.
- Caso esperado ou médio → É a média de todos os casos possíveis. Em geral, consideram-se todos os casos equiprováveis.

- Na análise dos casos, a dimensão da entrada está fixa.
  - No exemplo da pesquisa em vector, a dimensão do vector está fixa (e.g., n = 1 000 000), no melhor caso, no pior caso e no caso esperado.
  - ~~O melhor caso não é quando a dimensão do vector é zero ou um~~.

Complexidade Assintótica

Limite Superior — O

Notação "big-Oh"

$T(n)$ **é da ordem** de $f(n)$

$$T(n) = O(f(n)) \iff \exists\, c > 0, n_0 > 0 \quad \forall n \geq n_0 \quad T(n) \leq c\, f(n)$$

Exemplos

$$3n + 4 = O(n) \qquad n^2 = O(n^3)$$

$$5n^3 + 2n + 9 = O(n^3) \qquad n^3 \neq O(n^2)$$

$$5n \log n + 2 = O(n \log n) \qquad 64 = O(1)$$

Complexidade Assintótica

Limite Inferior — Ω

Notação "big-Omega"

$$T(n) = \Omega(f(n)) \iff \exists\, c > 0, n_0 > 0 \quad \forall n \geq n_0 \quad T(n) \geq c\, f(n)$$

Exemplos

$$\begin{aligned} 3n + 4 &= \Omega(n) & n^3 &= \Omega(n^2) \\ 5n^3 + 2n + 9 &= \Omega(n^3) & n^2 &\neq \Omega(n^3) \\ 5n \log n + 2 &= \Omega(n \log n) & 64 &= \Omega(1) \end{aligned}$$

Propriedade

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$$

Complexidade Assintótica

Notação mais precisa — Θ

Notação "big-Theta"

$$T(n) = \Theta(f(n)) \iff T(n) = O(f(n)) \text{ e } T(n) = \Omega(f(n))$$

Exemplos

$$3n + 4 = \Theta(n) \qquad n^3 \neq \Theta(n^2)$$

$$5n^3 + 2n + 9 = \Theta(n^3) \qquad n^2 \neq \Theta(n^3)$$

$$5n \log n + 2 = \Theta(n \log n) \qquad 64 = \Theta(1)$$

Convenção

Usaremos a notação O, sempre que possível,

com o sentido da notação mais precisa Θ.

Complexidade
Propriedades

Se $F(n) = O(f(n))$ **e** $G(n) = O(g(n))$ **então** $F(n) + G(n) = O(\max(f(n), g(n)))$.

Propriedades do O

1. $O(f) + O(g) = O(f+g) = O(\max(f,g))$.

   Exemplo: $O(n^2) + O(\log n) = O(n^2)$.

2. $O(f) \times O(g) = O(f \times g)$.

   Exemplo: $O(n^2) \times O(\log n) = O(n^2 \log n)$.

3. $O(cf) = O(f)$, com c constante.

   Exemplo: $O(3n^2) = O(n^2)$.

4. $f = O(f)$.

   Exemplo: $3n^2 + \log n = O(3n^2 + \log n)$.

$$3n^2 + \log n \overset{(4.)}{=} O(3n^2 + \log n) \overset{(1.)}{=} O(3n^2) \overset{(3.)}{=} O(n^2).$$

Complexidade Temporal

$$\left.\begin{array}{l}\text{Afectações, Comparações,}\\ \text{Operações aritméticas, Leituras e Escritas,}\\ \text{Acessos a campos de vectores ou a variáveis,}\\ \text{Chamadas e Retornos de métodos, } \ldots,\end{array}\right\} \quad O(1)$$

if C **then** S_1 **else** S_2 — $O(max(T_C ; T_{S1} ; T_{S2}))$

while C **do** S — $O(K * max(T_C ; T_S))$

K número de vezes que o ciclo é executado

$$\left.\begin{array}{l}\textbf{switch } E\\ \textbf{case } V_1 : S_1\\ \ldots\\ \textbf{case } V_j : S_j\end{array}\right\} \quad O(max(TE ; T_{S1} ; T_{Sj}))$$

Complexidade Temporal
Funções recursivas (1)

Recorrência 1

$$T(n) = \begin{cases} a & n = 0 \quad \textbf{ou} \quad n = 1 \\ b\,T(n-1) + c & n \geq 1 \quad \phantom{\textbf{ou}} \quad n \geq 2 \end{cases}$$

com $a \geq 0, \quad b \geq 1, \quad c \geq 1$ **constantes**

$$T(n) = \begin{cases} O(n) & b = 1 \\ O(b^n) & b > 1 \end{cases}$$

Recorrência 2

$$T(n) = \begin{cases} a & n = 0 \quad \textbf{ou} \quad n = 1 \\ b\,T(\frac{n}{c}) + f(n) & n \geq 1 \quad \phantom{\textbf{ou}} \quad n \geq 2 \end{cases}$$

com $a \geq 0, \quad b \geq 1, \quad c > 1$ **constantes**

e $f(n) = O(n^k)$, **para algum** $k \geq 0$

$$T(n) = \begin{cases} O(n^k) & b < c^k \\ O(n^k \log_c n) & b = c^k \\ O(n^{\log_c b}) & b > c^k \end{cases}$$

Complexidade Temporal

Funções recursivas (2)

Recorrência 2

$$T(n) = \begin{cases} a & n = 0 \quad \text{ou} \quad n = 1 \\ b\,T(\frac{n}{c}) + f(n) & n \geq 1 \qquad\;\; n \geq 2 \end{cases}$$

com $a \geq 0, \quad b \geq 1, \quad c > 1$ constantes

e $f(n) = O(n^k)$, para algum $k \geq 0$

$$T(n) = \begin{cases} O(n^k) & b < c^k \\ O(n^k \log_c n) & b = c^k \\ O(n^{\log_c b}) & b > c^k \end{cases}$$

Recorrência 2 (a) K=0, b=1, 2 e c=2

$$T(n) = \begin{cases} a & n = 0 \quad \text{ou} \quad n = 1 \\ b\,T(\frac{n}{2}) + O(1) & n \geq 1 \qquad\;\; n \geq 2 \end{cases}$$

com $a \geq 0, \quad b = 1, 2$ constantes

$$T(n) = \begin{cases} O(\log n) & b = 1 \\ O(n) & b = 2 \end{cases}$$

Recorrência 2 (b)

K=1

$$T(n) = \begin{cases} a & n = 0 \quad \text{ou} \quad n = 1 \\ b\,T(\frac{n}{c}) + O(n) & n \geq 1 \qquad\;\; n \geq 2 \end{cases}$$

com $a \geq 0, \quad b \geq 1, \quad c > 1$ constantes

$$T(n) = \begin{cases} O(n) & b < c \\ O(n \log_c n) & b = c \\ O(n^{\log_c b}) & b > c \end{cases}$$

Complexidade Temporal
Função recursiva – factorial (1)

```
public static int factorial( int n ){
   if ( n == 1 )
      return 1;
   return n * factorial(n – 1);
}
```

Complexidade Temporal

Função recursiva – factorial (2)


```java
public static int factorial( int n ){
   if ( n == 1 )
      return 1;
   return n * factorial(n - 1);
}
```

Número de Chamadas Recursivas

$$\text{numCR}(n) = \begin{cases} 0 & n = 1 \\ \text{numCR}(n-1) + 1 & n \geq 2 \end{cases}$$

Complexidade Temporal
Função recursiva – factorial (3)

```
public static int factorial( int n ){
   if ( n == 1 )
      return 1;
   return n * factorial(n - 1);
}
```

Número de Chamadas Recursivas

$$\text{numCR}(n) = \begin{cases} 0 & n = 1 \\ \text{numCR}(n-1) + 1 & n \geq 2 \end{cases}$$

Recorrência 1

$$T(n) = \begin{cases} a & n = 0 \quad \textbf{ou} \quad n = 1 \\ b\,T(n-1) + c & n \geq 1 \quad \phantom{\textbf{ou}} \quad n \geq 2 \end{cases}$$

com $a \geq 0, \quad b \geq 1, \quad c \geq 1$ **constantes**

$$T(n) = \begin{cases} O(n) & b = 1 \\ O(b^n) & b > 1 \end{cases}$$

Complexidade Temporal
Função recursiva – factorial (4)

```
public static int factorial( int n ){
   if ( n == 1 )
      return 1;
   return n * factorial(n - 1);
}
```

factorial(n) = O(n)

Número de Chamadas Recursivas

$$\text{numCR}(n) = \begin{cases} 0 & n = 1 \\ \text{numCR}(n-1) + 1 & n \geq 2 \end{cases}$$

Recorrência 1

$$T(n) = \begin{cases} a & n = 0 \quad \textbf{ou} \quad n = 1 \\ b\,T(n-1) + c & n \geq 1 \quad\quad\;\; n \geq 2 \end{cases}$$

com $a \geq 0, \quad b \geq 1, \quad c \geq 1$ **constantes**

$$T(n) = \begin{cases} O(n) & b = 1 \\ O(b^n) & b > 1 \end{cases}$$

Complexidade Temporal

Função recursiva – pesquisa binária (1)

```
public static int binarySearch(int[] v, int aim, int low, int high){
   int mid,current;

   if ( low > high )
     return -1;
   mid = ( low + high ) / 2;
   current = v[mid];
   if ( aim == current )
     return mid;
   if ( aim < current )
     return binarySearch(v, aim , low, mid-1);
   return binarySearch(v, aim, mid+1, high);
}
```

Complexidade Temporal

Função recursiva – pesquisa binária (2)

```
public static int binarySearch(int[] v, int aim, int low, int high){
   int mid,current;

   if ( low > high )
     return -1;
   mid = ( low + high ) / 2;
   current = v[mid];
   if ( aim == current )
     return mid;
   if ( aim < current )
     return binarySearch(v, aim , low, mid-1);
   return binarySearch(v, aim, mid+1, high);
}
```

Número de Chamadas Recursivas

$$\text{numCR}(n) = \begin{cases} 0 & n = 0 \\ \text{numCR}(\frac{n}{2}) + 1 & n \geq 1 \end{cases}$$

Complexidade Temporal

Função recursiva – pesquisa binária (3)

```
public static int binarySearch(int[] v, int aim, int low, int high){

    int mid,current;

   if ( low > high )
     return -1;
   mid = ( low + high ) / 2;
   current = v[mid];
   if ( aim == current )
     return mid;
   if ( aim < current )
     return binarySearch(v, aim , low, mid-1);
   return binarySearch(v, aim, mid+1, high);
}
```

Recorrência 2 (a) b=1, 2 e c=2

$$T(n) = \begin{cases} a & n = 0 \quad \textbf{ou} \quad n = 1 \\ b\,T(\frac{n}{2}) + O(1) & n \geq 1 \quad\quad\;\; n \geq 2 \end{cases}$$

com $a \geq 0, \quad b = 1, 2$ constantes

$$T(n) = \begin{cases} O(\log n) & b = 1 \\ O(n) & b = 2 \end{cases}$$

Número de Chamadas Recursivas

$$\text{numCR}(n) = \begin{cases} 0 & n = 0 \\ \text{numCR}(\frac{n}{2}) + 1 & n \geq 1 \end{cases}$$

Complexidade Temporal
Função recursiva – pesquisa binária (4)

```
public static int binarySearch(int[] v, int aim, int low, int high){

    int mid,current;

   if ( low > high )
     return -1;
   mid = ( low + high ) / 2;
   current = v[mid];
   if ( aim == current )
     return mid;
   if ( aim < current )
     return binarySearch(v, aim , low, mid-1);
   return binarySearch(v, aim, mid+1, high);
}
```

Recorrência 2 (a) b=1, 2 e c=2

$$T(n) = \begin{cases} a & n = 0 \quad \text{ou} \quad n = 1 \\ b\,T(\frac{n}{2}) + O(1) & n \geq 1 \quad \text{ou} \quad n \geq 2 \end{cases}$$

com $a \geq 0, \quad b = 1, 2$ constantes

$$T(n) = \begin{cases} O(\log n) & b = 1 \\ O(n) & b = 2 \end{cases}$$

binarySearch(n) = O(log n)

Número de Chamadas Recursivas

$$\text{numCR}(n) = \begin{cases} 0 & n = 0 \\ \text{numCR}(\frac{n}{2}) + 1 & n \geq 1 \end{cases}$$

Complexidade Temporal
Função recursiva – fibonacci (1)

```
//Requires: n >= 0
public static long fibonacciRec( int n ){

   if ( n == 0 )
      return 0;
   if ( n == 1 )
      return 1;
   return fibonacciRec(n – 1) + fibonacciRec(n – 2);
}
```

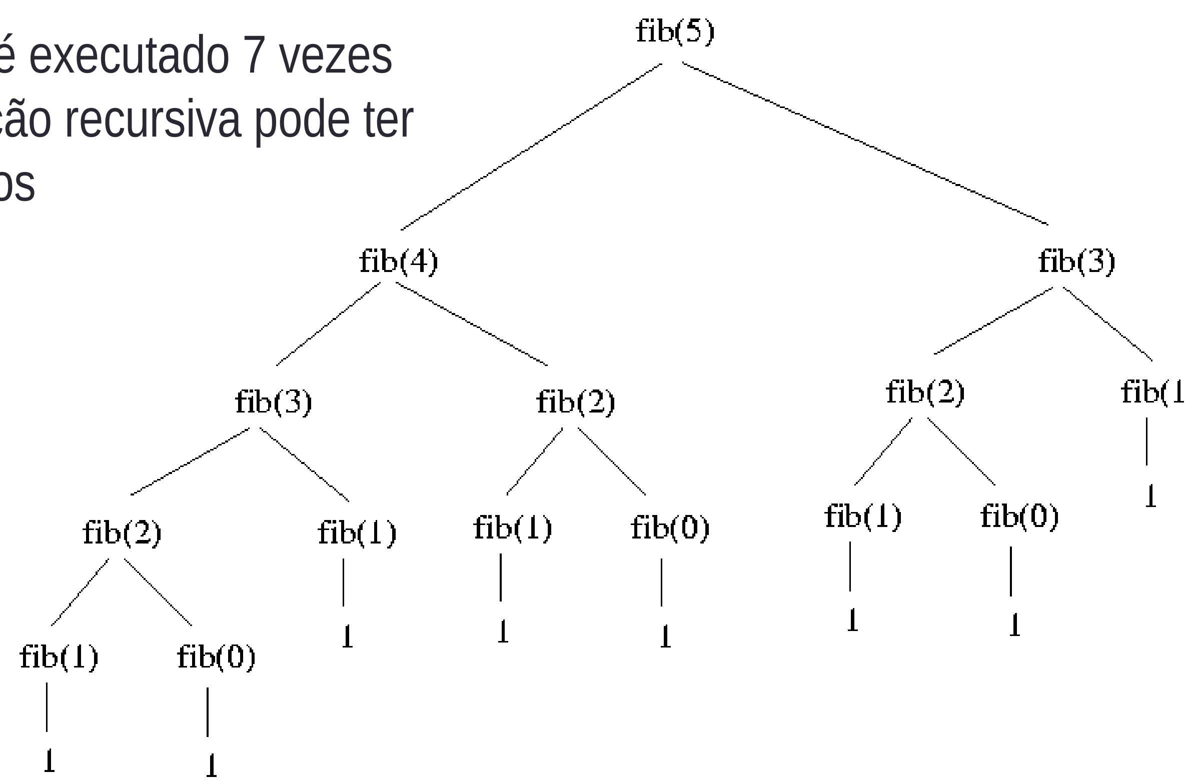
Número de Chamadas Recursivas

$$\mathrm{numCR}(n) = \begin{cases} 0 & n = 0 \\ 0 & n = 1 \\ \mathrm{numCR}(n-1) + \mathrm{numCR}(n-2) + 2 & n \geq 2 \end{cases}$$

Complexidade Temporal
Função recursiva – fibonacci (2)

```
//Requires: n >= 0
public static long fibonacciRec( int n ){
   if ( n == 0 )return 0;
   if ( n == 1 )return 1;
   return fibonacciRec(n – 1) + fibonacciRec(n – 2);
}
```

- Muitas chamadas repetidas – Fib(1) é executado 7 vezes
- Existem situações em que uma solução recursiva pode ter problemas de performance associados



Complexidade Temporal

Função recursiva – fibonacci (3)

```
//Requires: n >= 0
public static long fibonacciRec( int n ){
   if ( n == 0 )return 0;
   if ( n == 1 )return 1;
   return fibonacciRec(n – 1) + fibonacciRec(n – 2);
}
```

Número de Chamadas Recursivas

$$\mathsf{numCR}(n) = \begin{cases} 0 & n = 0 \\ 0 & n = 1 \\ \mathsf{numCR}(n-1) + \mathsf{numCR}(n-2) + 2 & n \geq 2 \end{cases}$$

Prova-se que $numCR(n) = O(\phi^n)$, ou seja $fibonacciRec(n) = O(\phi^n)$, com $\phi = (1+\sqrt{5}) / 2 \approx 1.6180\ldots$

Sugestão: (1) $numCR(n) = O(2^n)$ (*indução para* $numCR(n) \leq c2^n$)

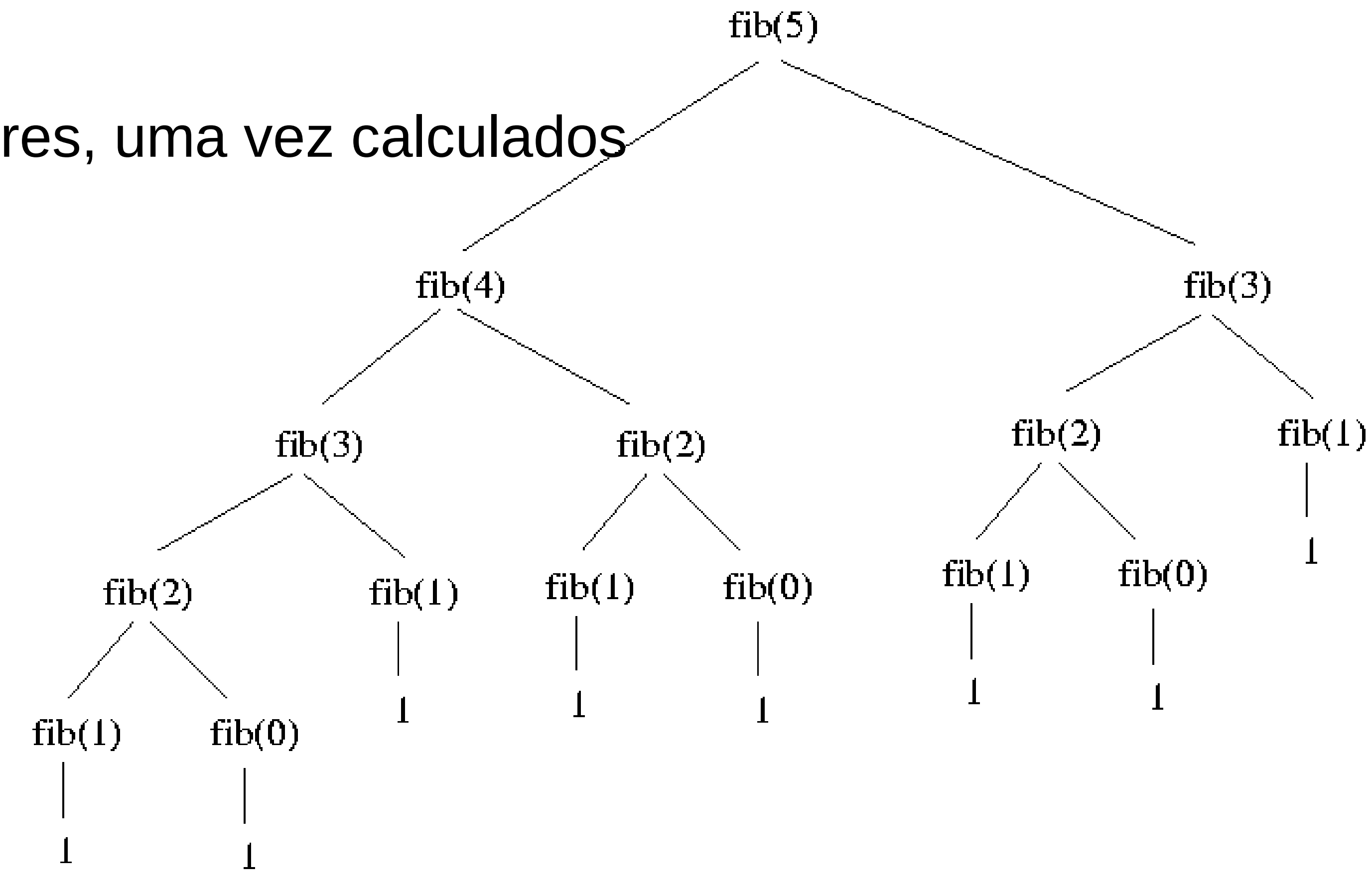
(2) $numCR(n) = \Omega((3/2)^n)$ (*indução para* $numCR(n) \geq c(3/2)^n$)

Complexidade Temporal
Função recursiva – fibonacci (4)

Como evitar calcular várias vezes um dado valor?

Guardar os valores, uma vez calculados

Função memória



Função memória
(Técnica de memorização)

- Consiste em <u>guardar todos os resultados</u> conseguidos em chamadas recursivas, da primeira vez que a respetiva chamada for ativada.
- Na ocasião da necessidade de execução de uma chamada recursiva, verifica-se, antes da execução da mesma, se o seu resultado já foi calculado anteriormente. Caso tenha sido já calculado, acede-se ao valor já guardado.

- Esta técnica reduz um algoritmo tipicamente exponencial para a dimensão da memória necessária para guardar todos os resultados gerados pelo algoritmo.
- A <u>complexidade espacial da solução cresce</u>.

Complexidade Temporal

Função recursiva – fibonacci (5)

Função memória

```
static long fibonacciMem( long[] memory, int n ){
   //Verifica se o valor foi calculado (já existe em memoria)
   //Se não foi, calcula e guarda na memória
   if (memory[n]==-1)
      if ( n == 0 )
         memory[n] = 0;
      else if ( n == 1 )
              memory[n] = 1;
       else
          memory[n] = fibonacciMem(memory, n - 1) +
                                           fibonacciMem(memory, n - 2);
   //Retorna o valor guardado na memória
   return memory[n];
}
```

Complexidade temporal:

$\text{fibonacciMem}(n) = O(n)$

Complexidade espacial:

$\text{fibonacciMem}(n) = O(n)$

Complexidade Temporal

Função recursiva – fibonacci (6)

```
// Iniciar a memória - todas as células inicializadas com um valor
// fora dos resultados possíveis do método
static void fibonacciInitMem( long[] vector ){
    for ( int i = 0; i < vector.length; i++ )
        vector[i] = -1;
}


// Fibonacci com a técnica de memorização
static long fibonacciMem( int n ){
    long[] memory = new long[n+1];

    fibonacciInitMem(memory);
    return fibonacciMem(memory, n);
}
```