



Algoritmos e Estruturas de Dados

Análise de algoritmos – Capítulo 4
2019/20



Complexidade

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

Complexidade Temporal

Análise Experimental (1)

- Estudar o tempo de execução de forma experimental:
 - Executar conjuntos de testes e registrar 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, ...).

★Veamos um exemplo “Concatenação de Strings”
(capítulo 4, pag 152)



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();  
}
```



<i>n</i>	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)

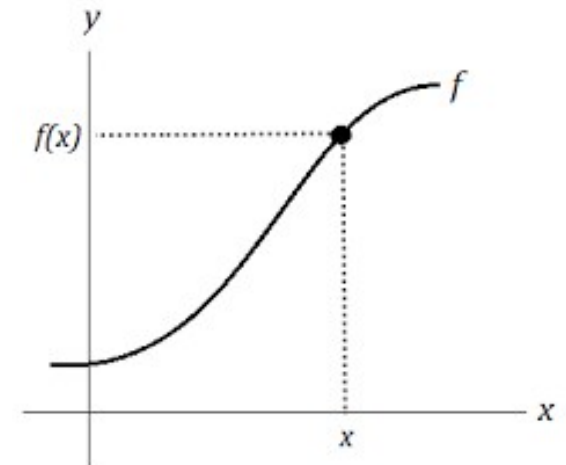
- Desvantagens:
 - Só considera um conjunto limitado de inputs de teste;
 - Dificulta a comparação entre resultados experimentais de algoritmos diferentes, se estes forem executados em hardware e software diferente;
 - O algoritmo tem de ser implementado para que o estudo experimental seja feito.


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 contar 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:
 - Afetações;
 - 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

Afectações, Comparações,
Operações aritméticas, Leituras e Escritas,
Acessos a campos de vectores ou a variáveis,
Chamadas e Retornos de métodos, ..., **1 unidade de tempo**

if C then S_1 else S_2 $T_C + \max(T_{S_1}; T_{S_2})$

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

switch E
case $V_1 : S_1$
...
case $V_j : S_j$ $T_E + \max(T_{S_1}; T_{S_j})$

Complexidade

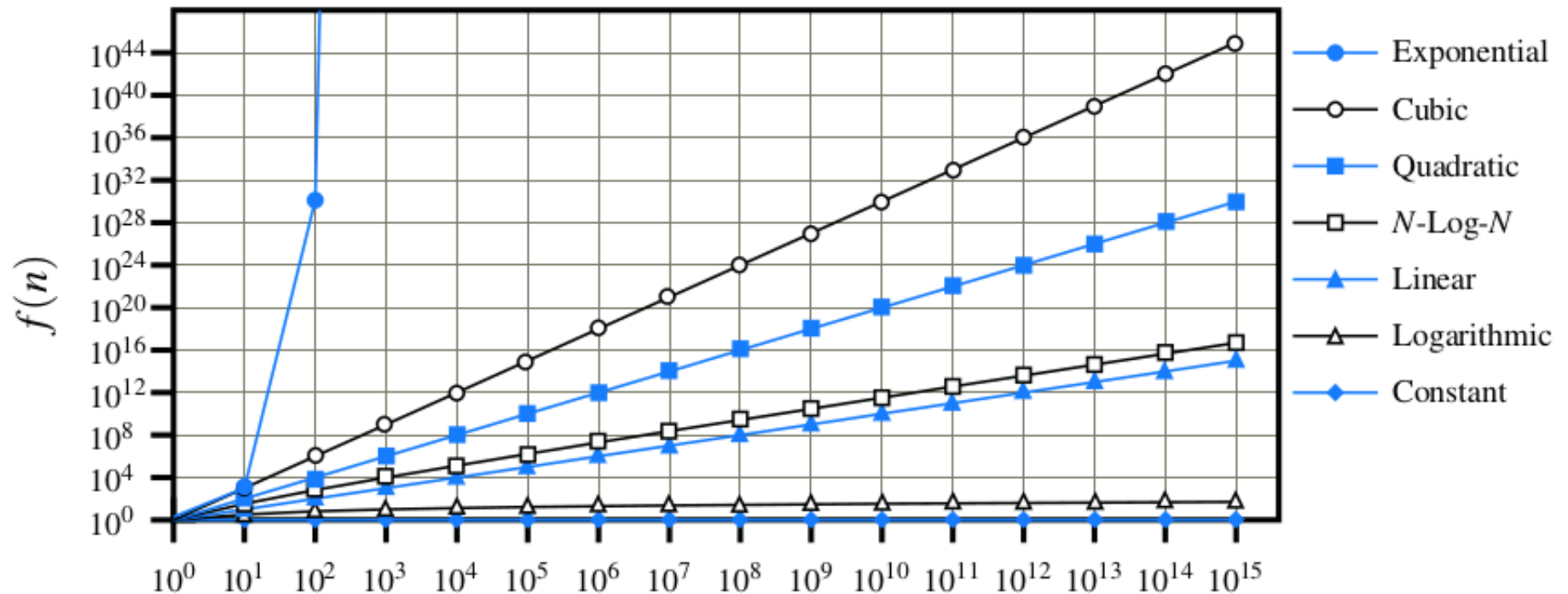
Algumas funções típicas (1)

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

1	constante	Polinomial		Exponencial
$\log n$	logarítmica			
n	linear		2^n	
$n \log n$			3^n	
n^2	quadrática		$n!$	
n^3	cúbica		\vdots	
n^4				
\vdots				

Complexidade

Algumas funções típicas (2)



- Idealmente, 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;
```

(1) 2 unidades de tempo

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

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

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

(3) 8 + 1 unidades de tempo

```
    return -1;
```

(4) 1 unidade de tempo

```
}
```

Complexidade Temporal

Contagem de operações primitivas (2)

- Para calcular a ordem de grandeza da complexidade, ignoram-se as constantes.

```
public static int sequentialSearch(int[] v, int n, int aim){  
    int i=0;  
    while ( i < n && v[i] < aim )    K * (8 + 2) unidades de tempo,  
        i++;                        K número de iterações  
    if ( i < n && v[i] == aim )  
        return i;  
    return -1;  
}
```

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

$$3n + 4 = O(n)$$

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

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

$$n^3 \neq O(n^2)$$

$$5n \log n + 2 = O(n \log n)$$

$$64 = O(1)$$

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

$$3n + 4 = \Omega(n)$$

$$n^3 = \Omega(n^2)$$

$$5n^3 + 2n + 9 = \Omega(n^3)$$

$$n^2 \neq \Omega(n^3)$$

$$5n \log n + 2 = \Omega(n \log n)$$

$$64 = \Omega(1)$$

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

$$3n + 4 = \Theta(n)$$

$$n^3 \neq \Theta(n^2)$$

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

$$n^2 \neq \Theta(n^3)$$

$$5n \log n + 2 = \Theta(n \log n)$$

$$64 = \Theta(1)$$

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

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

Complexidade Temporal

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

if C then S_1 else S_2 $O(\max(T_C; T_{S_1}; T_{S_2}))$

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

K número de vezes que o ciclo é executado

switch E $\left. \vphantom{\begin{matrix} case V_1 : S_1 \\ \dots \\ case V_j : S_j \end{matrix}} \right\} O(\max(T_E; T_{S_1}; T_{S_j}))$
case $V_1 : S_1$
...
case $V_j : S_j$

Complexidade Temporal

Funções recursivas (1)

Recorrência 1

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

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

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

Recorrência 2

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

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

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

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

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

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

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

Recorrência 2 (b)

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

com $a \geq 0$, $b \geq 1$, $c > 1$ 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)

```
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

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

com $a \geq 0$, $b \geq 1$, $c \geq 1$ **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);  
}
```

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

Número de Chamadas Recursivas

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

Recorrência 1

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

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

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

com $a \geq 0$, $b = 1, 2$ **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 $c=2$

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

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

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

$$\text{binarySearch}(n) = O(\log n)$$

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

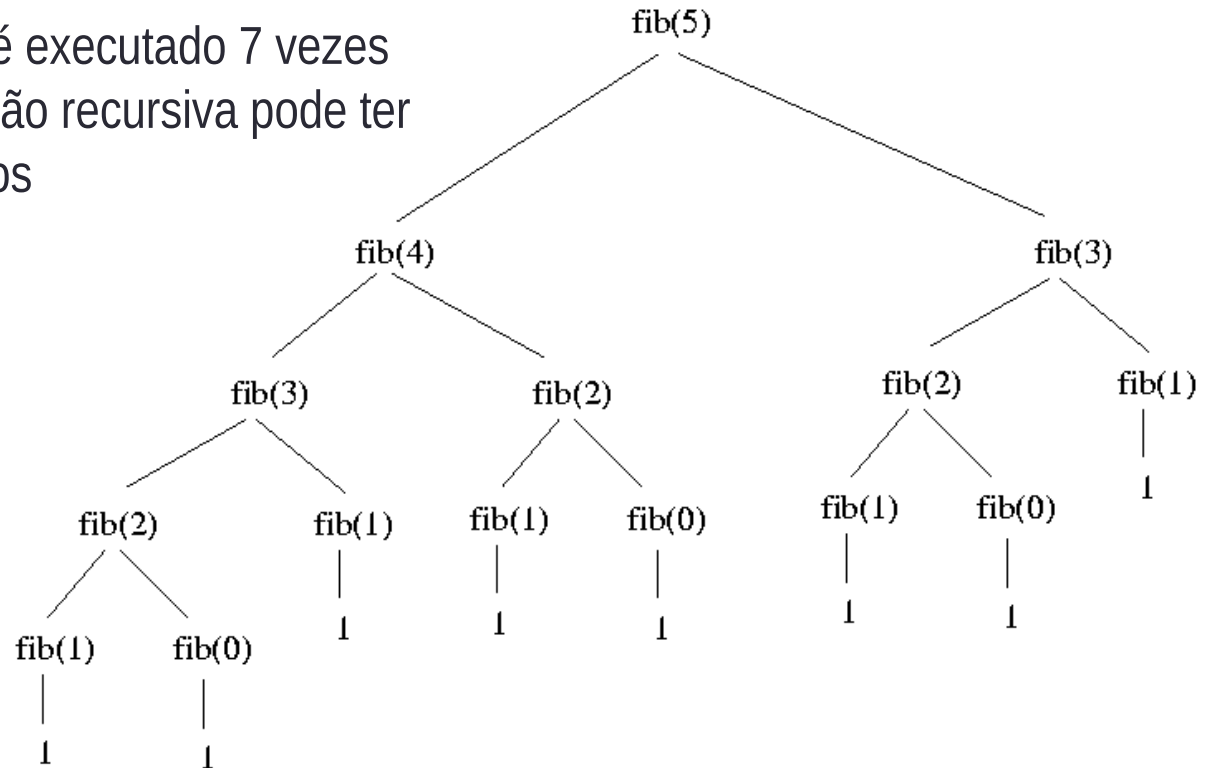
$$\text{numCR}(n) = \begin{cases} 0 & n = 0 \\ 0 & n = 1 \\ \text{numCR}(n-1) + \text{numCR}(n-2) + 2 & n \geq 2 \end{cases}$$

1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100, 2101, 2102, 2103, 2104, 2105, 2106, 2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127, 2128, 2129, 2130, 2131, 2132, 2133, 2134, 2135, 2136, 2137, 2138, 2139, 2140, 2141, 2142, 2143, 2144, 2145, 2146, 2147, 2148, 2149, 2150, 2151, 2152, 2153, 2154, 2155, 2156, 2157, 2158, 2159, 2160, 2161, 2162, 2163, 2164, 2165, 2166, 2167, 2168, 2169, 2170, 2171, 2172, 2173, 2174, 2175, 2176, 2177, 2178, 2179, 2180, 2181, 2182, 2183, 2184, 2185, 2186, 2187, 2188, 2189, 2190, 2191, 2192, 2193, 2194, 2195, 2196, 2197, 2198, 2199, 2200, 2201, 2202, 2203, 2204, 2205, 2206, 2207, 2208, 2209, 2210, 2211, 2212, 2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220, 2221, 2222, 2223, 2224, 2225, 2226, 2227, 2228, 2229, 2230, 2231, 2232, 2233, 2234, 2235, 2236, 2237, 2238, 2239, 2240, 2241, 2242, 2243, 2244, 2245, 2246, 2247, 2248, 2249, 2250, 2251, 2252, 2253, 2254, 2255, 2256, 2257, 2258, 2259, 2260, 2261, 2262, 2263, 2264, 2265, 2266, 2267, 2268, 2269, 2270, 2271, 2272, 2273, 2274, 2275, 2276, 2277, 2278, 2279, 2280, 2281, 2282, 2283, 2284, 2285, 2286, 2287, 2288, 2289, 2290, 2291, 2292, 2293, 2294, 2295, 2296, 2297, 2298, 2299, 2300, 2301, 2302, 2303, 2304, 2305, 2306, 2307, 2308, 2309, 2310, 2311, 2312, 2313, 2314, 2315, 2316, 2317, 2318, 2319, 2320, 2321, 2322, 2323, 2324, 2325, 2326, 2327, 2328, 2329, 2330, 2331, 2332, 2333, 2334, 2335, 2336, 2337, 2338, 2339, 2340, 2341, 2342, 2343, 2344, 2345, 2346, 2347, 2348, 2349, 2350, 2351, 2352, 2353, 2354, 2355, 2356, 2357, 2358, 2359, 2360, 2361, 2362, 2363, 2364, 2365, 2366, 2367, 2368, 2369, 2370, 2371, 2372, 2373, 2374, 2375, 2376, 2377, 2378, 2379, 2380, 2381, 2382, 2383, 2384, 2385, 2386, 2387, 2388, 2389, 2390, 2391, 2392, 2393, 2394, 2395, 2396, 2397, 2398, 2399, 2400, 2401, 2402, 2403, 2404, 2405, 2406, 2407, 2408, 2409, 2410, 2411, 2412, 2413, 2414, 2415, 2416, 2417, 2418, 2419, 2420, 2421, 2422, 2423, 2424, 2425, 2426, 2427, 2428, 2429, 2430, 2431, 2432, 2433, 2434, 2435, 2436, 2437, 2438, 2439, 2440, 2441, 2442, 2443, 2444, 2445, 2446, 2447, 2448, 2449, 2450, 2451, 2452, 2453, 2454, 2455, 2456, 2457, 2458, 2459, 2460, 2461, 2462, 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2470, 2471, 2472, 2473, 2474, 2475, 2476, 2477, 2478, 2479, 2480, 2481, 2482, 2483, 2484, 2485, 2486, 2487, 2488, 2489, 2490, 2491, 2492, 2493, 2494, 2495, 2496, 2497, 2498, 2499, 2500, 2501, 2502, 2503, 2504, 2505, 2506, 2507, 2508, 2509, 2510, 2511, 2512, 2513, 2514, 2515, 2516, 2517, 2518, 2519, 2520, 2521, 2522, 2523, 2524, 2525, 2526, 2527, 2528, 2529, 2530, 2531, 2532, 2533, 2534, 2535, 2536, 2537, 2538, 2539, 2540, 2541, 2542, 2543, 2544, 2545, 2546, 2547, 2548, 2549, 2550, 2551, 2552, 2553, 2554, 2555, 2556, 2557, 2558, 2559, 2560, 2561, 2562, 2563, 2564, 2565, 2566, 2567, 2568, 2569, 2570, 2571, 2572, 2573, 2574, 2575, 2576, 2577, 2578, 2579, 2580, 2581, 2582, 2583, 2584, 2585, 2586, 2587, 2588, 2589, 2590, 2591, 2592, 2593, 2594, 2595, 2596, 2597, 2598, 2599, 2600, 2601, 2602, 2603, 2604, 2605, 2606, 2607, 2608, 2609, 2610, 2611, 2612, 2613, 2614, 2615, 2616, 2617, 2618, 2619, 2620, 2621, 2622, 2623, 2624, 2625, 2626, 2627, 2628, 2629, 2630, 2631, 2632, 2633, 2634, 2635, 2636, 2637, 2638, 2639, 2640, 2641, 2642, 2643, 2644, 2645, 2646, 2647, 2648, 2649, 2650, 2651, 2652, 2653, 2654, 2655, 2656, 2657, 2658, 2659, 2660, 2661, 2662, 2663, 2664, 2665, 2666, 2667, 2668, 2669, 2670, 2671, 2672, 2673, 2674, 2675, 2676, 2677, 2678, 2679, 26

```
//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

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

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

Sugestão:

- (1) $\text{numCR}(n) = O(2^n)$ (indução para $\text{numCR}(n) \leq c2^n$)
- (2) $\text{numCR}(n) = \Omega((3/2)^n)$ (indução para $\text{numCR}(n) \geq c(3/2)^n$)

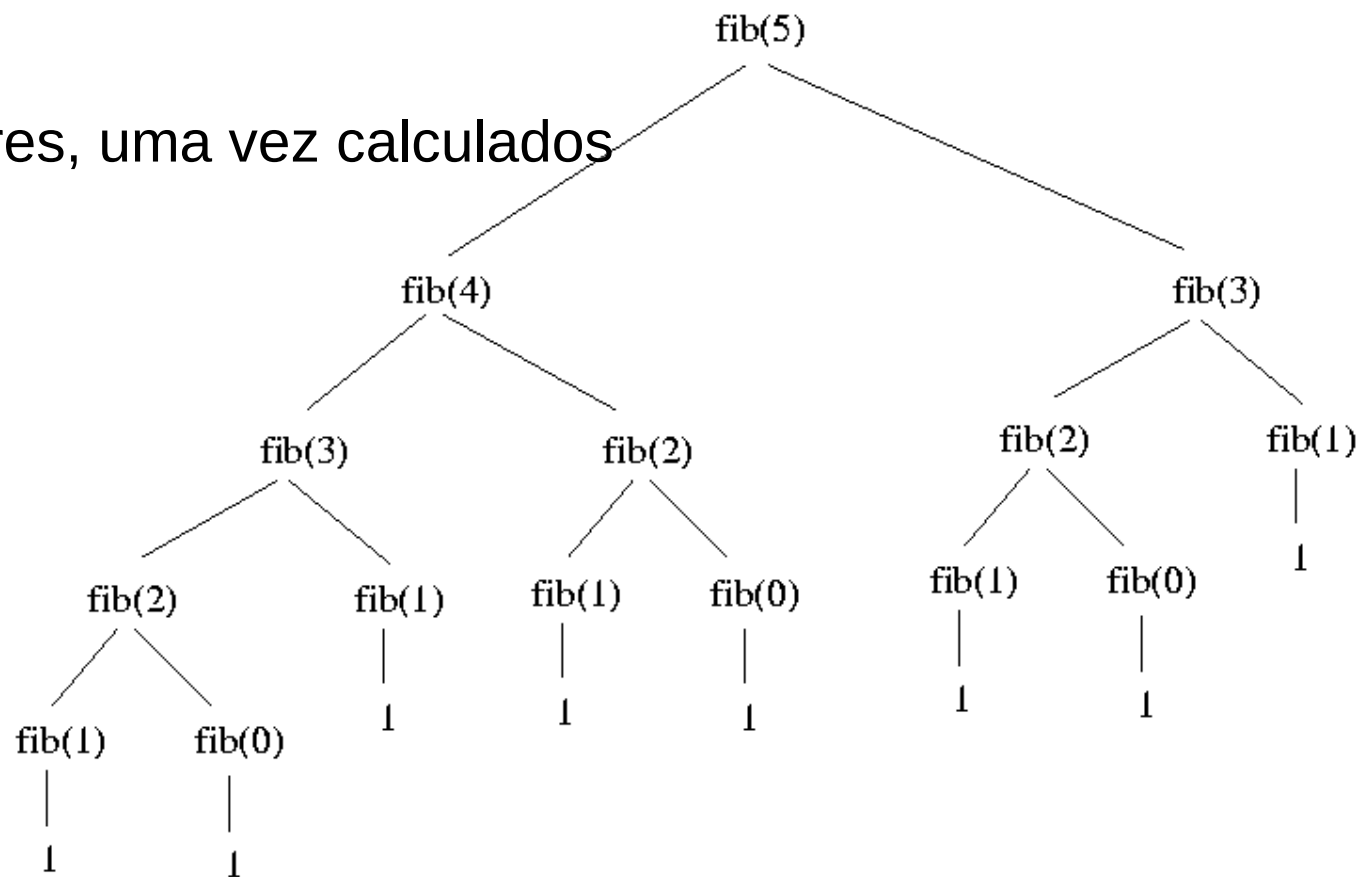
Complexidade Temporal


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 guardar todos os resultados 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 complexidade espacial da solução cresce.

Complexidade Temporal

Função recursiva – fibonacci (5)

Função memória


```
static long fibonacciMem( long[] memory, int n ){  
    //Verifica se o valor foi calculado (já existe em memória)  
    //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);
}
```