NOVA SCHOOL OF SCIENCE & TECHNOLOGY

NOVALINCS LABORATORY FOR COMPUTER SCIENCE AND INFORMATICS

ALGORITMOS E ESTRUTURAS DE DADOS 2023/2024
ANÁLISE DE ALGORITMOS

Armanda Rodrigues

4 de outubro de 2023

Análise de Algoritmos

- Temos até agora analisado soluções de problemas de forma intuitiva
- A análise da eficiência das soluções pode ser feita segundo duas perspetivas:
  - Complexidade Temporal – ligada ao período de tempo associado à execução do algoritmo subjacente à solução
  - Complexidade Espacial – relativa à memória utilizada pelas estruturas de dados que compõem a solução proposta

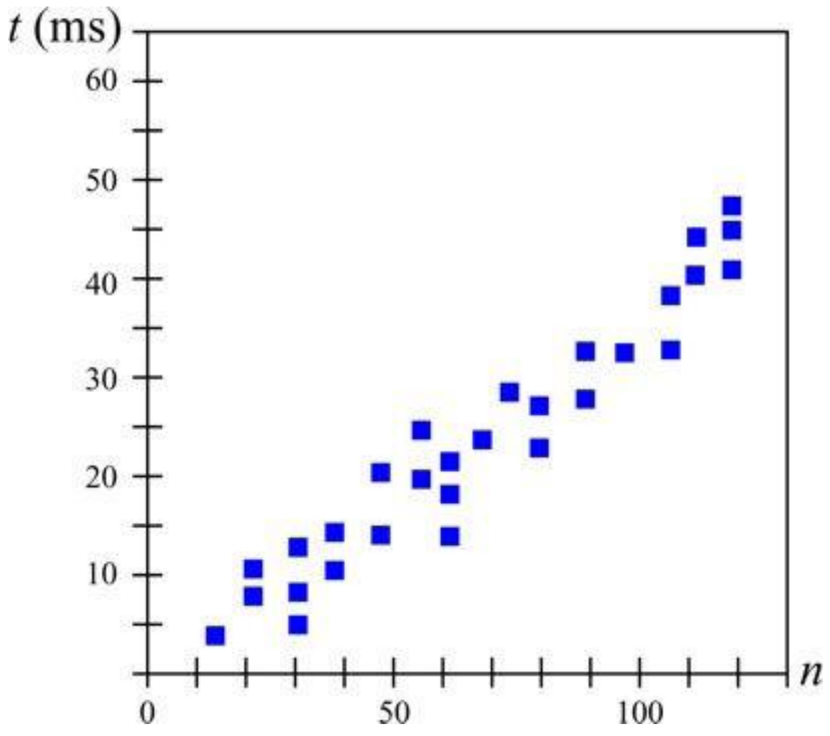
Tempo de execução

- Geralmente, o tempo de execução de um algoritmo aumenta com a dimensão do input
- Pode também variar em diferentes inputs com a mesma dimensão
- É também afetado por:
  - Hardware utilizado (processador, memória, disco, etc);
  - Software utilizado (sistema operativo, linguagem de programação, compilador, interpretador, etc)
- Assumindo que os algoritmos são avaliados nas mesmas condições, pretendemos estudar a relação entre o tempo de execução e a dimensão do input dos mesmos
- Pretendemos caracterizar o <u>tempo de execução</u> de um algoritmo em função da <u>dimensão do input</u>

Tempo de execução - experimental

- Sobre a implementação de um algoritmo, podemos estudar o seu tempo de execução de **forma experimental**:
  - Executar conjuntos de testes
  - Armazenar tempos em cada execução
  - E.g. Utilizar `System.currentTimeMillis()`
- Determinar a dependência geral entre a **dimensão do input** e o **tempo de execução**
  - Executar vários testes em diferentes tipos de input, de dimensões diferentes
- Os resultados permitem depois executar **análises estatísticas**, de forma a adaptar os resultados a funções conhecidas
- Para que os resultados façam sentido, as <u>amostras utilizadas têm de fazer sentido no contexto</u> e o <u>número de testes a executar tem de ser suficientemente grande</u>

Estudos experimentais



Resultados de um estudo experimental sobre o tempo de execução de um algoritmo. Um ponto com coordenadas (n,t) indica que sobre um input de dimensão n, o tempo de execução do algoritmo é t milisegundos (ms). – (Goodrich and Tamassia, 2006, página 163)

Desvantagens do estudo experimental

- 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

Uma ferramenta de análise que não necessite de testes experimentais é aconselhável

Análise de Algoritmos

- Propõe-se um método em que pretendemos associar, a cada algoritmo conhecido, uma função $f(n)$ que caracteriza o comportamento do mesmo, a partir da dimensão do input do algoritmo, dado pela variável n
- 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

Pesquisa sequencial sobre um vetor ordenado

```
public static int sequentialSearch(int[] v, int aim){

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

Pior caso -> O(v.length) + O(1) = (aproximadamente) O(v.length)

O(1) é o if()

Quais são as operações primitivas ?

caso médio assume-se igual ao pior caso, caso não saibamos o número máximo de repetição

Operações Primitivas

- Consideram-se operações primitivas:
  - Afetações,
  - Comparações,
  - Operações aritméticas,
  - Leituras e Escritas,
  - Acessos a campos de vetores ou a variáveis (referências),
  - Chamadas e Retornos de métodos, . . . ,
- Assumimos que os tempos de execução de operações primitivas é similar
- Assim o tempo de execução real de um algoritmo será proporcional ao número de operações primitivas que este executa

Pesquisa sequencial sobre um vetor ordenado

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

Elemento procurado	Células visitadas	
3	1	← Melhor caso
19	4	
23	5	
50	9	← Pior caso

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

Pesquisa dicotómica sobre um vetor ordenado

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

Elemento procurado	Células visitadas	
3	3 (4, 1,0)	
19	4 (4, 1, 2, 3)	← Pior caso $\approx \log_2 n$
24	1 (4)	← Melhor caso
50	4 (4, 6, 7, 8)	← Pior caso $\approx \log_2 n$

Caso médio (esperado) ?

Casos

- Na análise dos vários casos, a dimensão da entrada é considerada fixa (n)
- **Melhor caso** – o caso que implica a execução do menor número de instruções para executar a tarefa
- **Pior caso** – o caso que implica a execução do maior número de instruções para executar a tarefa
- **Caso médio (ou esperado)** – É a média de todos os casos possíveis, considerando-se todos os casos equiprováveis.

Pesquisa sequencial sobre um vetor ordenado

```
public static int sequentialSearch
       (int[] v, int aim){

    int  i=0;              O(1)   O(1)
    while ( i < v.length && v[i] < aim ){
       i++;
    }
    if ( i < v.length && v[i] == aim )
       return i;
    else return -1;
}
```

O(1) O(1) O(1) O(1) n

O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1)

	O(n) Melhor Caso $v[0] >= aim$	O(n) Pior Caso $v[n-1] < aim$
(1)	1	1
(2)	4	n(4+1)+2
(3)	5	3
Total	10	5n+6

n -> nº de vezes que faz o while
4 -> verificações dentreo do while
+1 -> i++;
+2 -> while e i < v.length

O(n) = 5n + 6

5n + 6 é aproximadamente 5n pois o 6 é irrelevante e como 5n apenas diz que tem um caracter linear entao pode-se aproximar para O(n)

Pesquisa sequencial sobre um vetor ordenado

```
public static int sequentialSearch
       (int[] v, int aim){

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

	Pior Caso v[n-1] < aim
(1)	1
(2)	n
(3)	1
Total	n

Ciclo executado n vezes. Em cada iteração, com um número fixo de instruções

Na prática, o que interessa

- As operações primitivas custam 1 unidade de tempo.
- Para calcular a ordem de grandeza da complexidade, ignoram-se as constantes.

Pesquisa binária sobre um vetor ordenado

```
public static int binSearch(int[]v, int aim){
    int low=0;
    int high=v.length-1;
    int mid;
    int current;

    while ( low <= high ) {
         mid = ( low + high ) / 2;
         current = v[mid];
         if ( aim == current )
           return mid;
         else if (aim < current)
                 high = mid-1;
              else
                 low = mid+1;

     }
    return -1;
}
```

(1) Número fixo de instruções: 1

(2) Em cada iteração, a zona de pesquisa é dividida ao meio- o nº de iterações ~ $\log_2 n$

(3) Número fixo de instruções: 1

Ciclo executado **$\log_2 n$** vezes. Em cada iteração, com um número fixo de instruções

Bubble sort

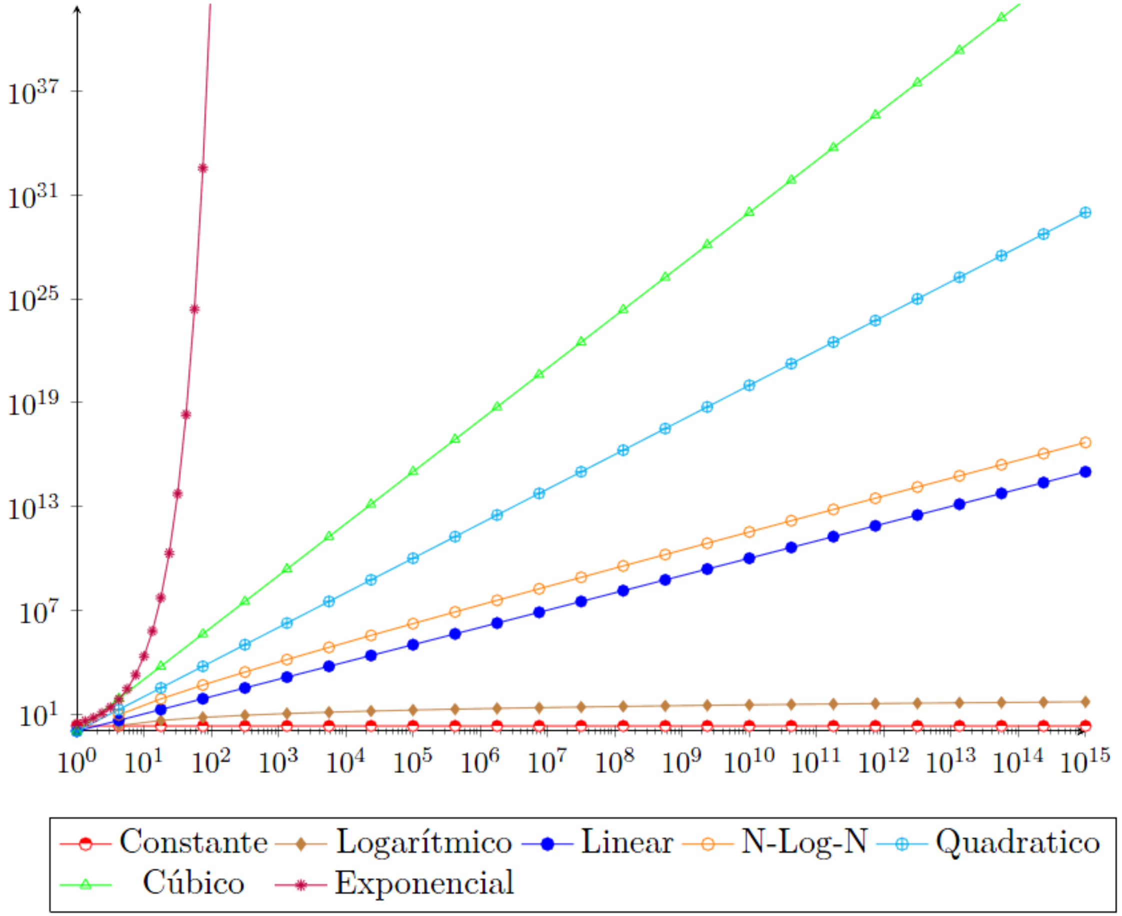
```
public static <E> void bubbleSort( int[] vec, int vecSize) {

   for ( int i = vecSize - 1; i > 0; i-- )
      for ( int j = 0; j < i; j++ )
         if ( vec[j] > vec[j + 1] )
            swapElements(vec, j, j + 1);
}
```

Todas as posições até à i serão avaliadas relativamente à sua chave e as trocas serão feitas se necessário.

A posição i será a posição do elemento com a maior chave (depois da 1ª iteração)

O número de comparações tende para **n**^{2}. É um algoritmo quadrático.

Funções comuns na análise de algoritmos



Tempos das funções mais comuns (1)

Constante (1)	Se o número de instruções de um método for executado um número constante de vezes (independente da dimensão do input)
Logarítmico (log n)	Quando a dimensão do problema é dividida por uma constante em cada passo
Linear (n)	Quando existe algum processamento para cada elemento de entrada
n log n	Quando um problema é resolvido através da resolução de um conjunto de sub-problemas, e combinando posteriormente as suas soluções

Tempos das funções típicas (2)

Quadrático (n^2)	Quando a dimensão da entrada duplica, o tempo aumenta 4x
Cúbico (n^3)	Quando a dimensão da entrada duplica, o tempo aumenta 8x
Exponencial (e.g. 2^n)	Quando a dimensão da entrada duplica, o tempo aumenta para o quadrado !

Notação assintótica

Limite Superior O

<u>Diz-se que $T(n)$ é da ordem de $f(n)$</u>

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

Exemplos:

$3n + 4$	$=$	$O(n)$
$5n^2$	$=$	$O(n^2)$
$5n^3 + 2n + 9$	$=$	$O(n^3)$
$5n \log n + 2$	$=$	$O(n \log n)$
n^2	$=$	$O(n^3)$
n^3	$\neq$	$O(n^2)$
64	$=$	$O(1)$

Notação assintótica

Limite Inferior Ω

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

Propriedade

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

Exemplos:

$3n + 4$	$=$	$\Omega(n)$
$5n^3 + 2n + 9$	$=$	$\Omega(n^3)$
$5n \log n + 2$	$=$	$\Omega(n \log n)$
n^3	$=$	$\Omega(n^2)$
n^2	$\neq$	$\Omega(n^3)$
64	$=$	$\Omega(1)$

Notação assintótica

Notação mais precisa Θ

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

Exemplos:

$3n + 4$	$=$	$\Theta(n)$
$5n^3 + 2n + 9$	$=$	$\Theta(n^3)$
$5n \log n + 2$	$=$	$\Theta(n \log n)$
n^3	$\neq$	$\Theta(n^2)$
n^2	$\neq$	$\Theta(n^3)$
64	$=$	$\Theta(1)$

Por convenção utiliza-se *O* com o sentido da notação mais precisa Θ

Propriedade

Se $F(n) = O(f(n))$ e $G(n) = O(g(n))$

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

Demonstração

Por hipótese, existem $n1,\ n2,\ c1,\ c2 > 0$ tais que:

$n \geq n1 \Rightarrow F(n) \leq c1\ f(n)$ e $n \geq n2 \Rightarrow G(n) \leq c2\ g(n)$.

Sejam $n3 = max(n1,\ n2)$ e $c3 = max(c1,\ c2)$.

Então, para qualquer $n \geq n3$:

$F(n)+G(n) \leq c1\ f(n)+c2\ g(n) \leq c3\ (f(n)+g(n)) \leq 2\ c3\ max(f(n),\ g(n))$.

Outras propriedades

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

Exemplo

$3n^2 + \log n$

(4.) =

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

(1.) =

$O(3n^2)$

(3.) =

$O(n^2)$

Executar a análise da complexidade temporal

- Operações primitivas: Afetações, Comparações, Operações aritméticas, Leituras e Escritas, Acessos a campos de vetores ou a variáveis, Chamadas e Retornos de métodos, . . . , - $O(1)$
- `if` C `then` S1 `else` S2 - $O(max(T_C, T_{S1}, T_{S2}))$
- `while` C `do` S - $O(k\ max(T_C, T_S))$
- `switch` E

    `case` $V_1 : S_1$

    . . .

    `case` $V_j : S_j$

$O(max(T_E, T_{S1}, ..., T_{Sj}))$

k é o número de vezes que o ciclo é executado

Complexidade de funções recursivas

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

Intuitivamente, sabemos que o número de ativações do método depende do número de chamadas recursivas

Complexidade de funções recursivas

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

Número de Chamadas Recursivas

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

Complexidade de funções recursivas

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

Número de Chamadas Recursivas

Se n for 1, não haverá chamadas recursivas

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

Complexidade de funções recursivas

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

Número de Chamadas Recursivas

Se n igual ou superior a 2 teremos:

- 1 chamada recursiva e

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

Complexidade de funções recursivas

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

Número de Chamadas Recursivas

Se n igual ou superior a 2 teremos:

- 1 chamada recursiva <u>e</u>
- Todas as chamadas associadas ao problema com dimensão n-1

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

Aplicação da Recorrência ao Fatorial Recursivo

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

Aplica-se a Recorrência 1

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

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

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

Daqui se tira que *numCR(n)* = *O(n)* e *factorial(n)* = *O(n)*.

Pesquisa Binária Recursiva

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

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

Pesquisa Binária Recursiva

Número de Chamadas Recursivas

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

Aplica-se a Recorrência 2 (a)

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

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

Daqui se tira que $numCR(n) = O(\log n)$ e $pesquisa(n) = O(\log n)$.

Recorrência 1

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

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

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

com $a \geq 0, \quad b \geq 1, \quad c > 1$ **constantes**
e $\mathrm{f(n)} = \mathrm{O(n}^k\mathrm{)}$, **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) $k = 0$; $b = 1, 2$; $c = 2$

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

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

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

Recorrência 2 (b) $k = 1$

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

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

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

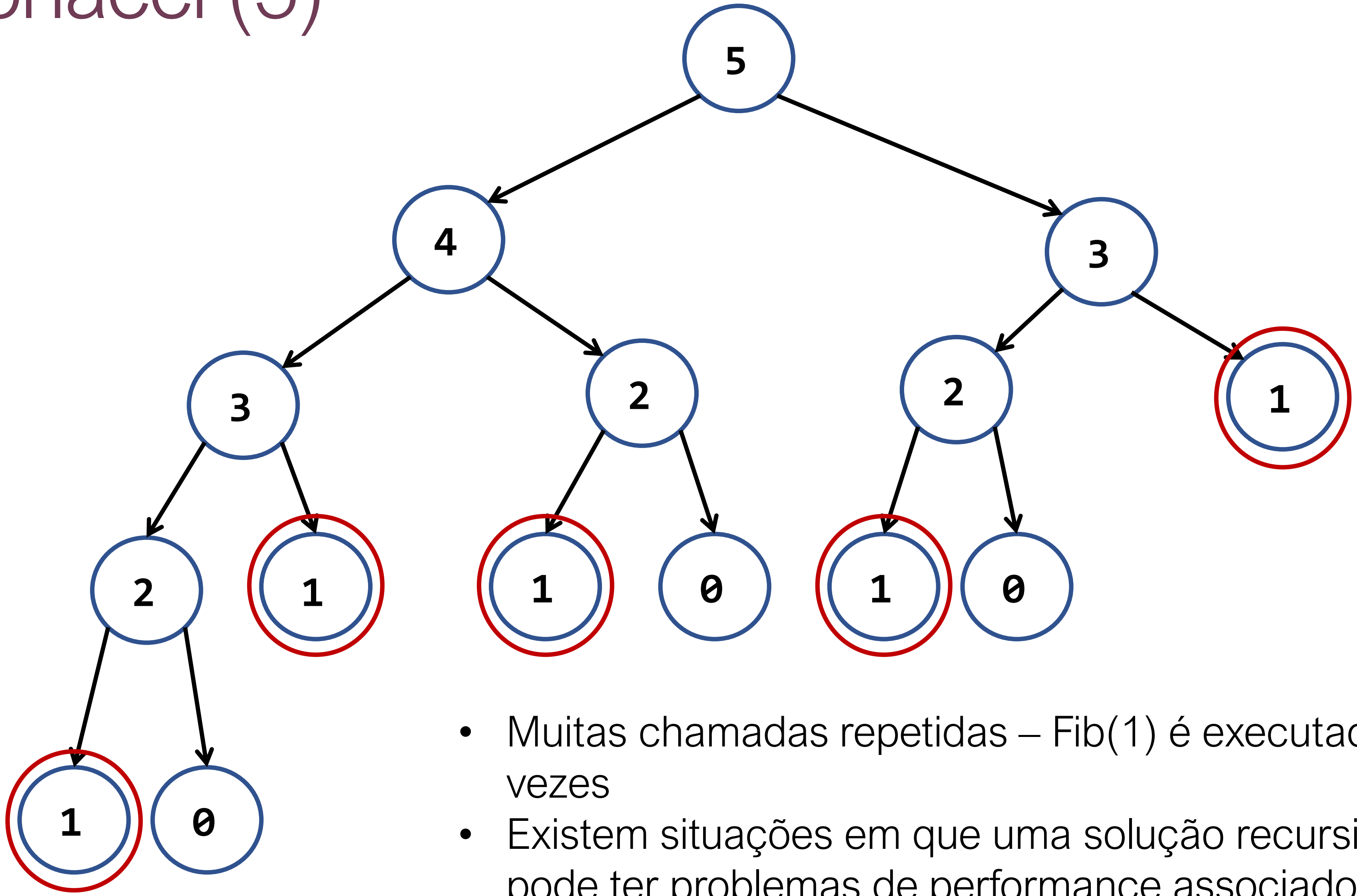
Fibonacci Recursivo

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

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

Número de Chamadas Recursivas

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

Fibonacci (5)



- Muitas chamadas repetidas – Fib(1) é executado 5 vezes
- Existem situações em que uma solução recursiva pode ter problemas de performance associados

Fibonacci Recursivo

Número de Chamadas Recursivas

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

Prova-se que $numCR(n) = O(\phi^n)$, ou seja

$fibonacciRec(n) = O(\phi^n)$, com $\phi = (1+\sqrt{5}) / 2 \approx 1.6180\ldots$

O que significa que a função recursiva de Fibonacci tem complexidade temporal exponencial

Complexidades Temporais das Estruturas de Dados já estudadas

Pilha

Operação	Pilha em Vetor	Pilha em Lista Ligada
`isEmpty`	$O(1)$	$O(1)$
`size`	$O(1)$	$O(1)$
`top`	$O(1)$	$O(1)$
`push`	$O(1)$	$O(1)$
`pop`	$O(1)$	$O(1)$

Fila

Operação	Fila em Vetor	Fila em Lista Ligada
`isEmpty`	$O(1)$	$O(1)$
`size`	$O(1)$	$O(1)$
`enqueue`	$O(1)$	$O(1)$
`dequeue`	$O(1)$	$O(1)$

Lista

Operação	Melhor Caso	Pior Caso	Caso Esperado
`isEmpty, size`	$O(1)$	$O(1)$	$O(1)$
`getFirst, getLast`	$O(1)$	$O(1)$	$O(1)$
`get`	$O(1)$	$O(n)$	$O(n)$
`addFirst, addLast`	$O(1)$	$O(1)$	$O(1)$
`add`	$O(1)$	$O(n)$	$O(n)$
`removeFirst, removeLast`	$O(1)$	$O(1)$	$O(1)$
`remove (por posição)`	$O(1)$	$O(n)$	$O(n)$
`find, remove (por elemento)`	$O(1)$	$O(n)$	$O(n)$
`iterator`	$O(1)$	$O(1)$	$O(1)$