Curso Estruturas de Dados e Algoritmos Expert

Prof. Nelio Alves

Complexidade de algoritmos

DEVSUPERIOR

https://devsuperior.com.br

1

Complexidade de algoritmos

É uma medida que descreve a quantidade de recursos computacionais que um algoritmo necessita para executar, em relação ao **tamanho da entrada** do algoritmo.

Esses recursos podem incluir:

- tempo de execução
- espaço de memória

2

Complexidade de tempo

Refere-se ao **número de passos** que um algoritmo leva para completar sua execução em função do **tamanho da entrada**.

Em outras palavras:

Quanto cresce o número de passos à medida que cresce o tamanho da entrada.

Complexidade de espaço

Refere-se à **quantidade de memória** que o algoritmo necessita para seu processamento, em função do **tamanho da entrada**.

Em outras palavras:

Quanto cresce a quantidade de memória utilizada à medida que cresce o tamanho da entrada.

Exemplo busca sequencial

Função para encontrar a posição de um elemento dentro de uma lista. Se o elemento não existir na lista, retorna -1.

v =

15	82	79	32	41	28
0	1	2	3	4	5

busca(32, v) = 3

busca(82, v) = 1

busca(22, v) = -1

5

```
function sequentialSearch(elem, arr) {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] == elem) {
            return i;
        }
    }
    return -1;
}

const result = sequentialSearch(32, [15, 82, 79, 32, 41, 28]);
console.log(result);
```

6

Análise da complexidade de tempo:

Melhor caso: o elemento procurado é o primeiro da lista → **1 passo**

f(n) = 1 (função constante)

Pior caso: o elemento procurado é o último da lista, ou não existe → **n passos**

f(n) = n (função linear)

Caso médio: o elemento procurado está em uma posição "qualquer" da lista → **n/2 passos**

f(n) = n/2 (função linear)

Análise da complexidade de espaço:

Independente do caso, esse algoritmo utiliza apenas uma variável auxiliar (i), e esse uso de memória não se altera em função do tamanho da entrada

f(n) = 1 (função constante)

Notação assintótica

Foca no comportamento de longo prazo de um algoritmo, ignorando constantes e termos de menor ordem que têm pouca influência em entradas grandes.

$n/2 \rightarrow n$

$4n \rightarrow n$

$3n^2 + 2n + 7 \rightarrow n^2$

$5n^3 + 12n^2 + 20 \rightarrow n^3$

$n/2 \rightarrow n$

$4n \rightarrow n$

$3n^2 + 2n + 7 \rightarrow n^2$

$5n^3 + 12n^2 + 20 \rightarrow n^3$

A notação assintótica permite uma comparação mais limpa e mais significativa entre algoritmos.

Ela foca no termo dominante que mais influencia o crescimento quando n se torna muito grande.

Big O, Big Omega, Big Theta

Big O

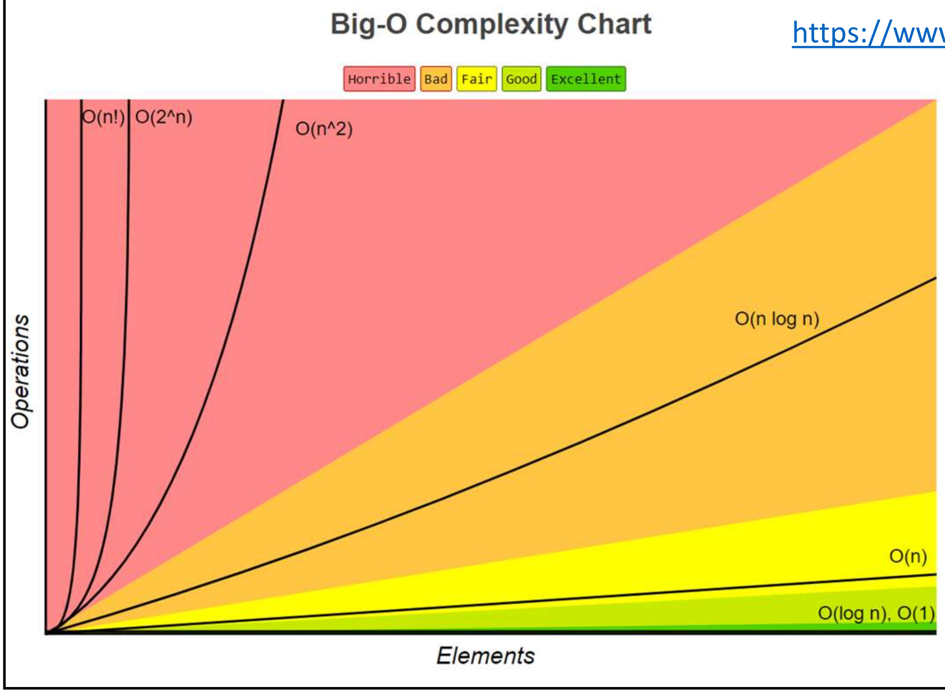
Limite superior, pior caso.

Big Omega (Ω)

Limite inferior, melhor caso.

Big Theta (Θ)

Limite apertado, caso médio.



https://www.bigocheatsheet.com

Exemplo de algoritmo de ordem linear

Voltando ao algoritmo sequential-search:

```
function sequentialSearch(elem, arr) {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] == elem) {
            return i;
        }
    }
    return -1;
}
```


Complexidade do algoritmo sequential-search, para uma entrada de tamanho n:

Complexidade de tempo			**Complexidade de espaço**
Melhor caso	Caso médio	Pior caso	Pior caso
$\Omega(1)$	$\Theta(n)$	$O(n)$	$O(1)$

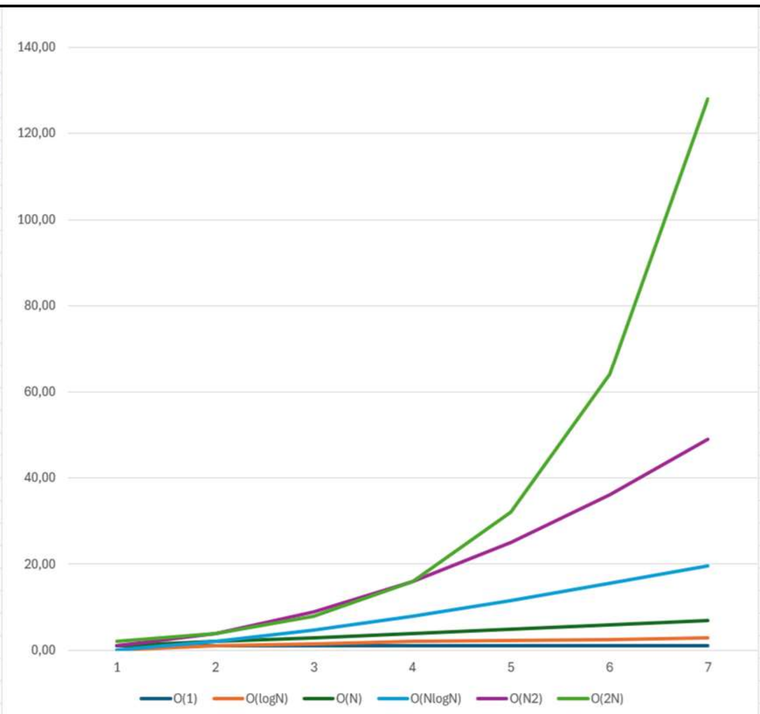
Nota 1: estamos considerando para a complexidade de espaço, o consumo **adicional** de memória, além da memória já ocupada pelos parâmetros de entrada do algoritmo.

Nota 2: quando não há necessidade de um detalhamento de melhor/médio/pior caso, é comum constar somente a notação Big O.

Complexidades mais comuns

O(1)	Ordem constante	Algoritmos tratáveis
O(logN)	Ordem logarítmica	
O(N)	Ordem linear	
O(NlogN)	Ordem log linear	
$O(N^2)$, $O(N^3)$, ...	Ordem polinomial (quadrática, cúbica, etc.)	
$O(2^N)$, $O(3^N)$, ...	Ordem exponencial	Algoritmos intratáveis

N	O(1)	O(logN)	O(N)	O(NlogN)	O(N2)	O(2N)
1	1,00	0,00	1,00	0,00	1,00	2,00
2	1,00	1,00	2,00	2,00	4,00	4,00
3	1,00	1,58	3,00	4,75	9,00	8,00
4	1,00	2,00	4,00	8,00	16,00	16,00
5	1,00	2,32	5,00	11,61	25,00	32,00
6	1,00	2,58	6,00	15,51	36,00	64,00
7	1,00	2,81	7,00	19,65	49,00	128,00



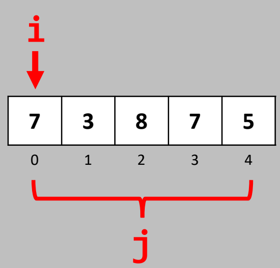
Exemplo de algoritmo de ordem quadrática

Função que recebe um vetor de números, e retorna um novo vetor dizendo quantos elementos maiores existem no vetor, para cada elemento do vetor.

Exemplo

Entrada	**Saída**
`[7, 3, 8, 7, 5]`	`[1, 4, 0, 1, 3]`

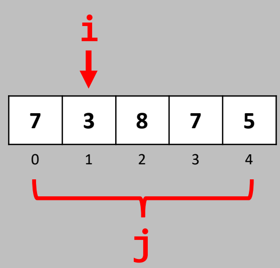
Raciocínio



No loop **j** vamos testar: `arr[j] > arr[i]`

`arr[0] > arr[0]` → `7 > 7` → falso
`arr[1] > arr[0]` → `3 > 7` → falso
`arr[2] > arr[0]` → `8 > 7` → verdadeiro
`arr[3] > arr[0]` → `7 > 7` → falso
`arr[4] > arr[0]` → `5 > 7` → falso

1



`arr[0] > arr[1]` → `7 > 3` → verdadeiro
`arr[1] > arr[1]` → `3 > 3` → falso
`arr[2] > arr[1]` → `8 > 3` → verdadeiro
`arr[3] > arr[1]` → `7 > 3` → verdadeiro
`arr[4] > arr[1]` → `5 > 3` → verdadeiro

4

```
function higherValues(arr) {

  let newArray = new Array(arr.length).fill(0);

  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length; j++) {
      if (arr[j] > arr[i]) {
        newArray[i]++;
      }
    }
  }

  return newArray;
}

const result = higherValues([7, 3, 8, 7, 5]);
console.log(result);
```


Complexidade do algoritmo higher-values, para uma entrada de tamanho N

Complexidade de tempo	Complexidade de espaço
$O(N^2)$	$O(N)$

Nota 1: estamos considerando para a complexidade de espaço, o consumo **adicional** de memória, além da memória já ocupada pelos parâmetros de entrada do algoritmo.

Nota 2: quando não há necessidade de um detalhamento de melhor/médio/pior caso, é comum constar somente a notação Big O.

Exemplo de algoritmo de ordem cúbica

Função para multiplicar uma matriz $A_{(MxP)}$ por uma matriz $B_{(PxN)}$.

O resultado será uma matriz $C_{(MxN)}$. Exemplo:

$$
\begin{array}{cc} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \end{array} & \left[\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \end{array}\right] \end{array}
\times
\begin{array}{cc} & \begin{array}{cc} 0 & 1 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \left[\begin{array}{cc} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{array}\right] \end{array}
=
\begin{array}{cc} & \begin{array}{cc} 0 & 1 \end{array} \\ \begin{array}{c} 0 \\ 1 \end{array} & \left[\begin{array}{cc} 58 & 64 \\ 139 & 154 \end{array}\right] \end{array}
$$

$A_{(2x3)}$ $B_{(3x2)}$ $C_{(2x2)}$

M = 2, P = 3, N = 2

REGRA:
Cada elemento (i,j) da matriz C é a soma dos produtos dos elementos da linha (i) da matriz A e a coluna (j) da matriz B.

Exemplo: $C_{(0,0)} = 1*7 + 2*9 + 3*11 = 58$

$$
\begin{array}{cc} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \end{array} & \left[\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \end{array}\right] \end{array}
\times
\begin{array}{cc} & \begin{array}{cc} 0 & 1 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \left[\begin{array}{cc} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{array}\right] \end{array}
=
\begin{array}{cc} & \begin{array}{cc} 0 & 1 \end{array} \\ \begin{array}{c} 0 \\ 1 \end{array} & \left[\begin{array}{cc} \boxed{58} & 64 \\ 139 & 154 \end{array}\right] \end{array}
$$

$A_{(2x3)}$ $B_{(3x2)}$ $C_{(2x2)}$

M = 2, P = 3, N = 2

```
function matrixMultiply(A, B) {
  let m = A.length;
  let p = A[0].length;
  let n = B[0].length;

  let C = new Array(m).fill().map(() => new Array(n).fill(0));

  for (let i = 0; i < m; i++) {
      for (let j = 0; j < n; j++) {
          for (let k = 0; k < p; k++) {
              C[i][j] += A[i][k] * B[k][j];
          }
      }
  }
  return C;
}

const A = [[1, 2, 3], [4, 5, 6]];
const B = [[7, 8], [9, 10], [11, 12]];
const C = matrixMultiply(A, B);
console.log(C);
```


Complexidade do algoritmo matrix-multiply, para uma entrada de tamanho M×P + P×N

Complexidade de tempo	Complexidade de espaço
O(M×N×P)	O(M×N)

Nota: note que temos aqui um caso atípico, onde o tamanho da entrada não é determinado por um único valor N. Estamos considerando a complexidade de tempo e espaço como cúbica e quadrática respectivamente, devido aos fatores multiplicadores explícitos em cada expressão usada na notação Big O.

Exemplo de algoritmo de ordem exponencial

A sequência de Fibonacci começa com 0, 1, e depois cada número é a soma de seus dois antecessores: 0 1 1 2 3 5 8 13...

Faça uma função para retornar o valor de uma dada posição da sequência de Fibonacci. Exemplos:

Exemplo

Entrada	Saída
6	8

Solução recursiva exponencial

Sequência Fibonacci:	**0**	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**...**
	0	1	2	3	4	5	6	7	

fib(0) = 0

fib(1) = 1

fib(2) = 0+1 = 1 ⟶ **fib(2) = fib(1) + fib(0)**

fib(3) = 1+1 = 2 ⟶ **fib(3) = fib(2) + fib(1)**

fib(4) = 1+2 = 3 ⟶ **fib(4) = fib(3) + fib(2)**

fib(5) = 2+3 = 5 ⟶ **fib(5) = fib(4) + fib(3)**

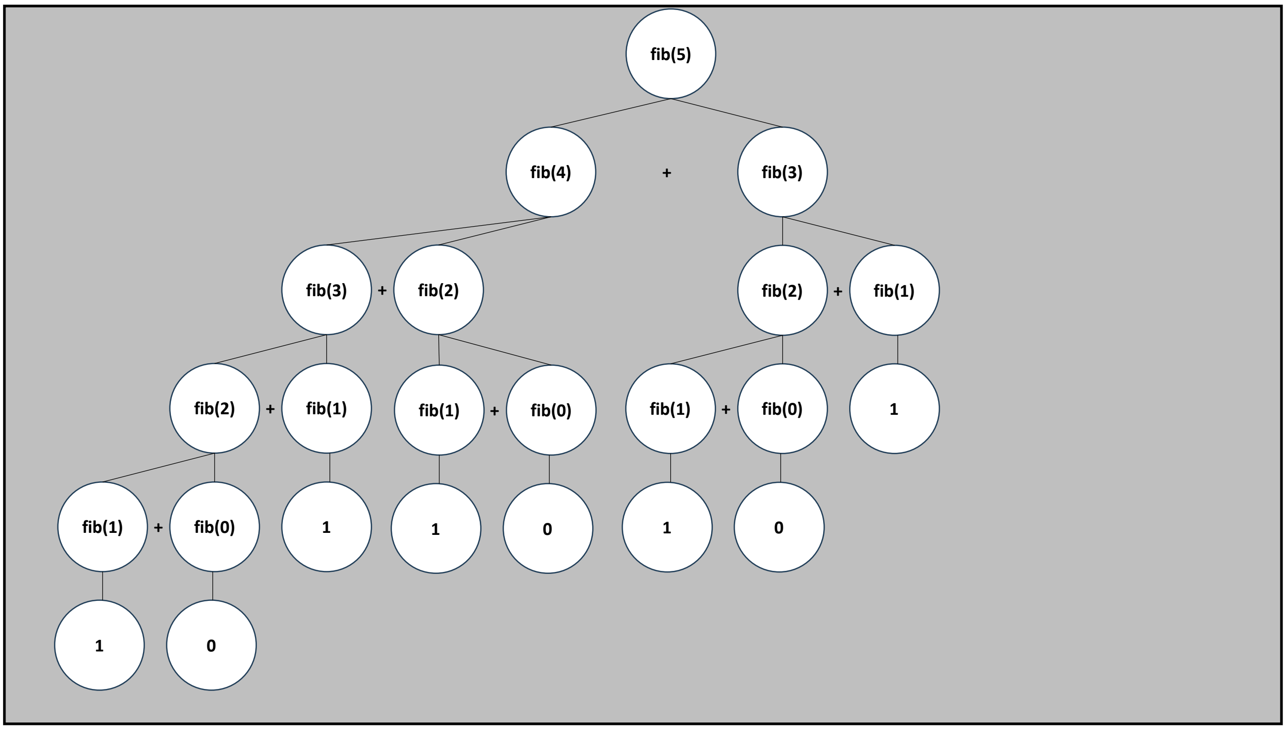
fib(6) = 3+5 = 8 ⟶ **fib(6) = fib(5) + fib(4)**

...

fib(N) = fib(N - 1) + fib(N - 2)

```
function fibExponential(n) {
  if (n == 0) {
    return 0;
  }
  if (n == 1) {
    return 1;
  }
  return fibExponential(n - 1) + fibExponential(n - 2);
}

const result = fibExponential(40);
console.log(result);
```

Complexidade do algoritmo fib-exponential, para uma entrada de valor n

Complexidade de tempo	Complexidade de espaço
$O(2^N)$	$O(N)$

Nota: estamos considerando aqui a complexidade em função do próprio valor n, e não em função do "tamanho" da entrada. Isso algumas vezes é praticado em problemas matemáticos onde a entrada é um valor escalar.

Exemplo de algoritmo de ordem logarítmica

Encontre a posição de um elemento em um array **ordenado** de números. Exemplos:

v =	**7**	**13**	**20**	**25**	**28**	**31**	**35**	**39**	**40**	**45**	**46**	**48**	**57**	**59**	**63**	**71**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

busca(39, v) = 7

busca(46, v) = 10

busca(22, v) = -1

Solução recursiva

busca(46, v, 0, 15)

7	13	20	25	28	31	35	39	40	45	46	48	57	59	63	71
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

busca(46, v, 8, 15)

7	13	20	25	28	31	35	39	40	45	46	48	57	59	63	71
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

busca(46, v, 8, 10)

7	13	20	25	28	31	35	39	40	45	46	48	57	59	63	71
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

busca(46, v, 10, 10)

7	13	20	25	28	31	35	39	40	45	46	48	57	59	63	71
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

10

31

busca(22, v, 0, 15)

7	13	20	25	28	31	35	39	40	45	46	48	57	59	63	71
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

busca(22, v, 0, 6)

7	13	20	25	28	31	35	39	40	45	46	48	57	59	63	71
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

busca(22, v, 0, 2)

7	13	20	25	28	31	35	39	40	45	46	48	57	59	63	71
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

busca(22, v, 2, 2)

7	13	20	25	28	31	35	39	40	45	46	48	57	59	63	71
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

busca(22, v, 3, 2)

7	13	20	25	28	31	35	39	40	45	46	48	57	59	63	71
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

-1

32

Solução recursiva

Casos base:

- se (início > fim) → -1
- se (elem = v[meio]) → meio

Casos recursivos:

- se (elem < v[meio]) → procure na 1ª metade
- se (elem > v[meio]) → procure na 2ª metade


```
function binarySearch(elem, arr) {
  return binarySearchTailRecursive(elem, arr, 0, arr.length - 1);
}

function binarySearchTailRecursive(elem, arr, start, finish) {
  if (start > finish) {
    return -1;
  }

  const middle = Math.floor((start + finish) / 2);
  if (elem == arr[middle]) {
    return middle;
  }
  else if (elem < arr[middle]) {
    return binarySearchTailRecursive(elem, arr, start, middle - 1);
  }
  else {
    return binarySearchTailRecursive(elem, arr, middle + 1, finish);
  }
}

const v = [7, 13, 20, 25, 28, 31, 35, 39, 40, 45, 46, 48, 57, 59, 63, 71];
console.log(binarySearch(39, v));
console.log(binarySearch(46, v));
console.log(binarySearch(22, v));
```

Complexidade do algoritmo binary-search, para uma entrada de tamanho n:

Complexidade de tempo			Complexidade de espaço
Melhor caso	Caso médio	Pior caso	Pior caso
$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$	$O(1)$