SBVORIN: Organização e Recuperação da Informação

Aula 03: Algoritmos de Busca, Algoritmos de Ordenação Comparativos e Lineares (não comparativos)

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

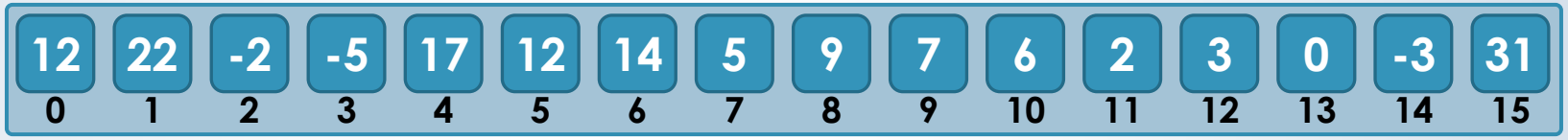
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Principais Algoritmos Elementares de Busca/Pesquisa

- Busca Sequencial;
- Busca Binária.

Busca Sequencial

- Buscar por uma chave sequencialmente, ou seja, elemento a elemento, em uma estrutura de dados que possa ser percorrida em alguma ordem.



- Buscar pela chave 3:
- Buscar pela chave 7:
- Buscar pela chave 12:
- Buscar pela chave 100:

Busca Sequencial

```
public static int search( int[] array, int key ) {
    int n = array.length;
    for ( int i = 0; i < n; i++ ) {
        if ( array[i] == key ) {
            return i;
        }
    }
    return -1;
}
```

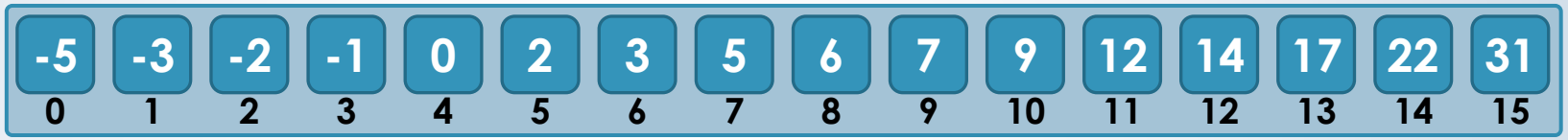
i
posição atual que está sendo verificada

n
tamanho do array que está sendo pesquisado

key
chave que está sendo pesquisada

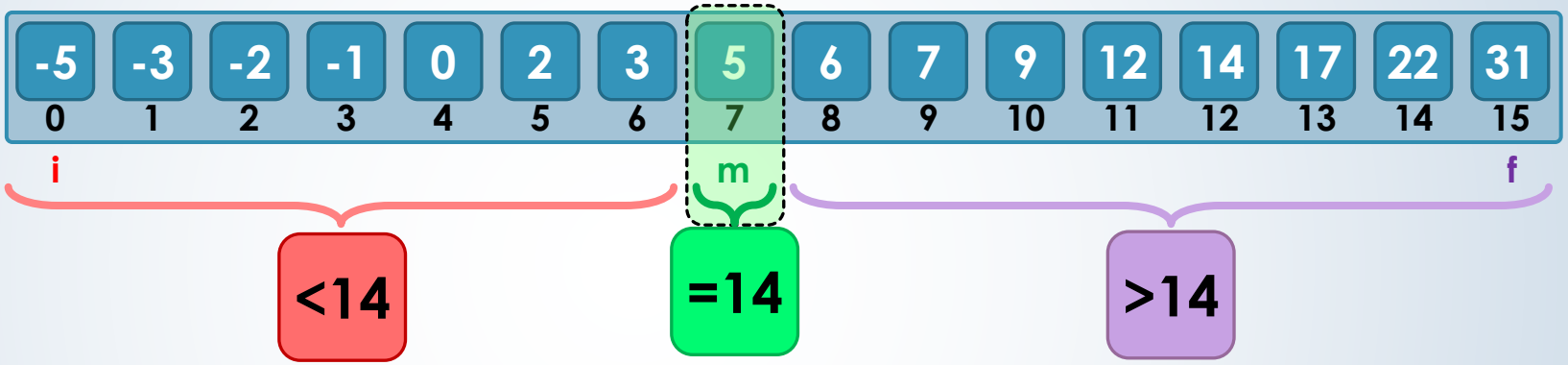
Busca Binária

- Buscar por uma chave em uma estrutura de dados linear **ordenada**, verificando o elemento do meio e dividindo-a sucessivamente em duas subestruturas;
- Buscar pela chave 14:



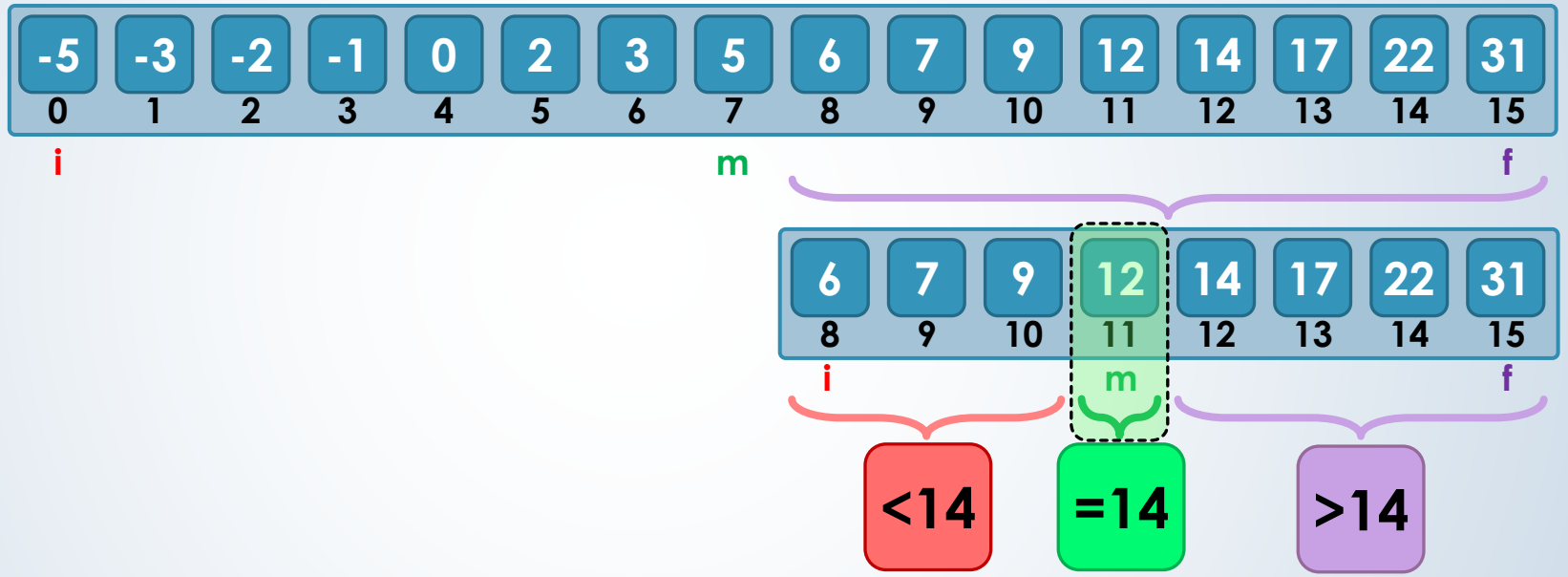
Busca Binária

- Buscar por uma chave em uma estrutura de dados linear ordenada, verificando o elemento do meio e dividindo-a sucessivamente em duas subestruturas;
- Buscar pela chave 14:



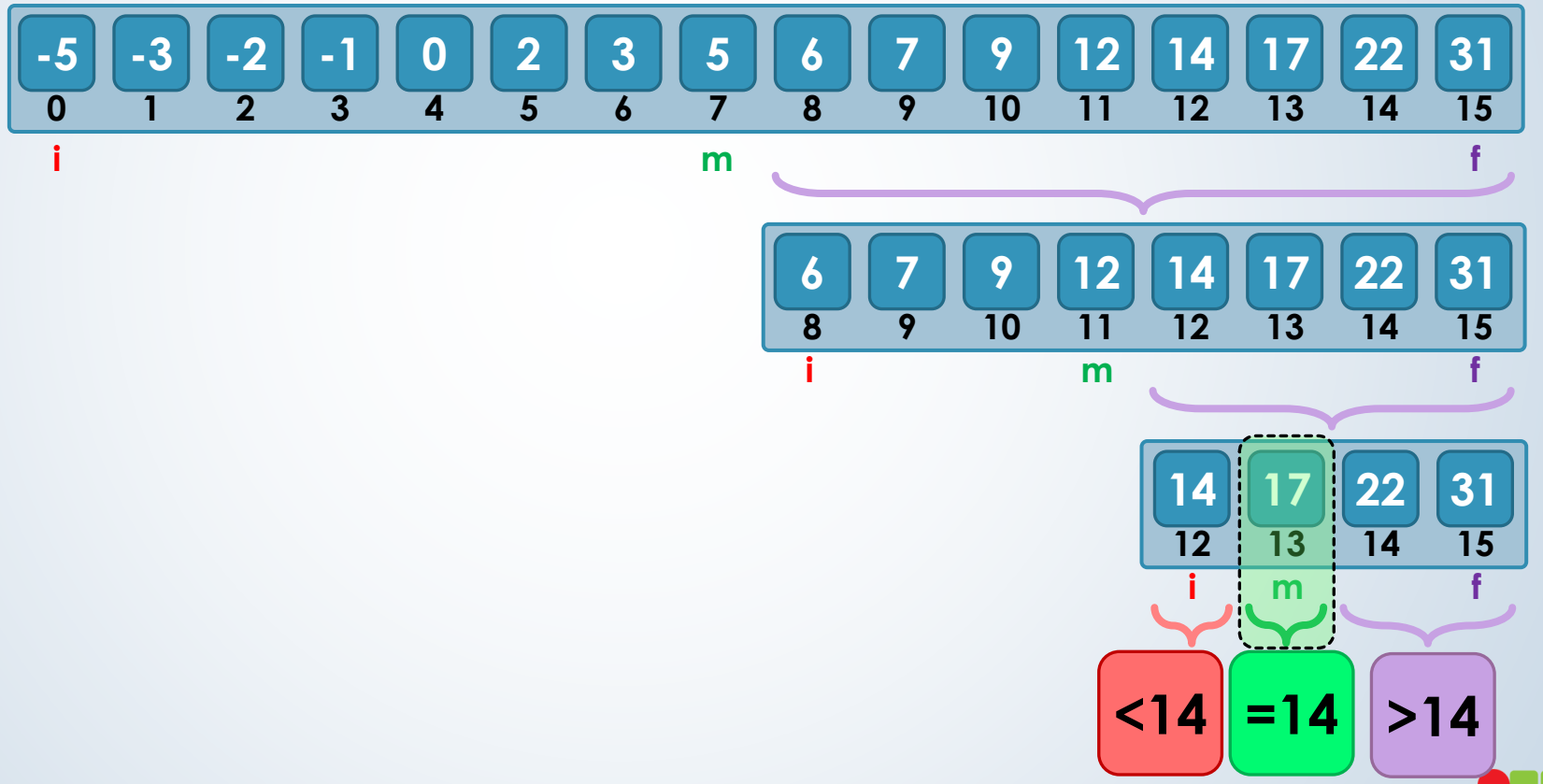
Busca Binária

- Buscar por uma chave em uma estrutura de dados linear ordenada, verificando o elemento do meio e dividindo-a sucessivamente em duas subestruturas;
- Buscar pela chave 14:



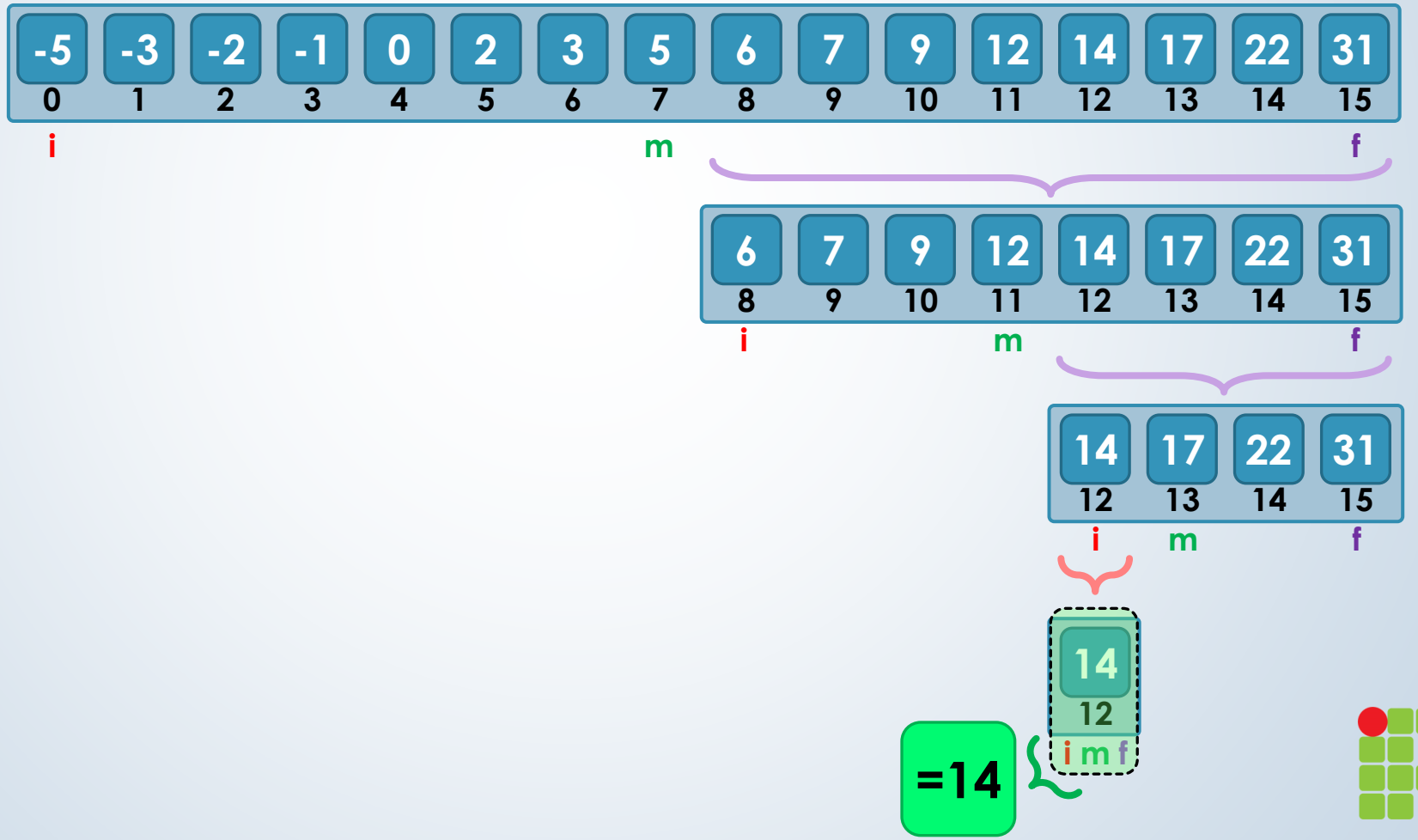
Busca Binária

- Buscar por uma chave em uma estrutura de dados linear ordenada, verificando o elemento do meio e dividindo-a sucessivamente em duas subestruturas;
- Buscar pela chave 14:



Busca Binária

- Buscar por uma chave em uma estrutura de dados linear ordenada, verificando o elemento do meio e dividindo-a sucessivamente em duas subestruturas;
- Buscar pela chave 14:



Busca Binária Iterativa

```
public static int search( int[] array, int key ) {
    int start = 0;
    int end = array.length - 1;
    int middle;
    while ( start <= end ) {
        middle = ( start + end ) / 2;
        if ( key == array[middle] ) {
            return middle;
        } else if ( key < array[middle] ) {
            end = middle - 1;
        } else { // key > array[middle]
            start = middle + 1;
        }
    }
    return -1;
}
```

start
início da subsequência

end
fim da subsequência

middle
elemento na posição do meio da subsequência

Busca Binária Recursiva

```
public static int search( int[] array, int key ) {
    return searchR( array, key, 0, array.length - 1 );
}
private static int searchR( int[] array, int key, int start,
int end ) {
    int middle = ( start + end ) / 2;
    if ( start <= end ) {
        if ( key == array[middle] ) {
            return middle;
        } else if ( key < array[middle] ) {
            return searchR( array, key, start, middle - 1 );
        } else { // key > array[middle]
            return searchR( array, key, middle + 1, end );
        }
    } else {
        return -1;
    }
}
```

searchR
método recursivo para a divisão sucessiva do array

start
início da subsequência

end
fim da subsequência

middle
elemento na posição do meio da subsequência

Principais Algoritmos de Ordenação Comparativos

- Selection Sort
- Insertion Sort
- Bubble Sort

} Elementares

- Shell Sort

- Merge Sort
- Quick Sort
- Heap Sort

} Não Elementares

Selection Sort

- Ordenação por Seleção (Selection Sort)
  - Divisão dos dados em duas sequências:
    - Ordenada e não-ordenada.

Selection Sort

- **Iteração:** procurar pelo menor elemento da sequência não-ordenada e concatená-lo na sequência ordenada;
- Os valores dos dados não interferem na execução do algoritmo.

Selection Sort

- In-place? Sim
- Estável? Não
- Complexidade:
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n^2)$
  - Melhor caso: $O(n^2)$

Selection Sort

```
public static void sort( int[] array ) {
    int n = array.length;
    for ( int i = 0; i < n; i++ ) {
        int min = i;
        for ( int j = i + 1; j < n; j++ ) {
            if ( array[j] < array[min] ) {
                min = j;
            }
        }
        swap( array, i, min );
    }
}
```

i
índice da sequência ordenada

j
índice da sequência não-ordenada

min
índice do menor elemento na sequência não-ordenada

```
public static void swap( int[] array, int p1, int p2 ) {
    int temp = array[p1];
    array[p1] = array[p2];
    array[p2] = temp;
}
```

Selection Sort
Estabilidade

i	min	0	1	2
0	2	B_1	B_2	A
1	1	A	B_2	B_1
2	2	A	B_2	B_1
		A	B_2	B_1

Estabilidade

- Aplicação típica: primeiro ordenar por nome, depois ordenar por seção.

`selectionSort( a, porNome )`

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

`selectionSort( a, porSeção )`

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

- Alunos da seção 3 não estão mais ordenados por nome!

Estabilidade

- Uma ordenação **estável**, preserva a ordem relativa dos itens com mesma chave.

Insertion Sort

- Ordenação por Inserção (Insertion Sort)
  - Divisão dos dados em duas sequências:
    - Ordenada e não-ordenada.

Insertion Sort

- **Iteração:** inserir o primeiro elemento da sequência não-ordenada na sequência ordenada;
- Os valores dos dados interferem na execução do algoritmo.

Insertion Sort

- In-place? Sim
- Estável? Sim
- Complexidade:
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n^2)$
  - Melhor caso: $O(n)$

Insertion Sort

```
public static void sort( int[] array ) {
    int n = array.length;
    for ( int i = 1; i < n; i++ ) {
        int j = i;
        while ( j > 0 && array[j-1] > array[j] ) {
            swap( array, j-1, j );
            j--;
        }
    }
}
```

i
índice da sequência ordenada

j
índice da sequência não-ordenada

Insertion Sort
Estabilidade

i	j	0	1	2	3	4
0	0	B_1	A_1	A_2	A_3	B_2
1	0	A_1	B_1	A_2	A_3	B_2
2	1	A_1	A_2	B_1	A_3	B_2
3	2	A_1	A_2	A_3	B_1	B_2
4	4	A_1	A_2	A_3	B_1	B_2
		A_1	A_2	A_3	B_1	B_2

Bubble Sort

- Ordenação por “Flutuação” (Bubble Sort)
  - Aplicação sucessiva de comparações entre vizinhos (na prática também separa a sequência em duas partes: ordenada e não-ordenada).

Bubble Sort

- **Iteração:** percorrer toda a sequência não-ordenada comparando todos os vizinhos e trocando de posição quando necessário. No final, o menor elemento poderá ser concatenado na sequência ordenada;
- Os valores dos dados interferem na execução do algoritmo.

Bubble Sort

- In-place? Sim
- Estável? Sim
- Complexidade:
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n^2)$
  - Melhor caso: $O(n)$

Bubble Sort

```
public static void sort( int[] array ) {
    int n = array.length;
    int i = 0;
    boolean swapped;
    do {
        swapped = false;
        for ( int j = n - 1; j > i; j-- ) {
            if ( array[j-1] > array[j] ) {
                swap( array, j-1, j );
                swapped = true;
            }
        }
        i++;
    } while ( swapped && i < n );
}
```

i
índice da sequência ordenada

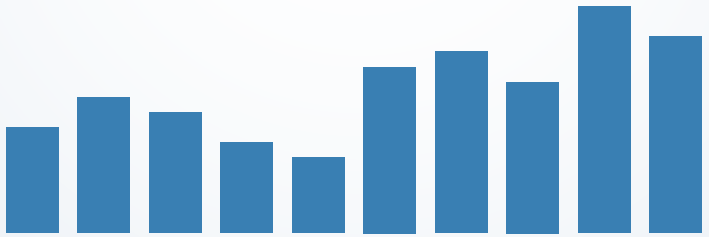
j
índice da sequência não-ordenada

swapped
indica se houve ou não troca

Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

n = 10
h = 4

Shell Sort

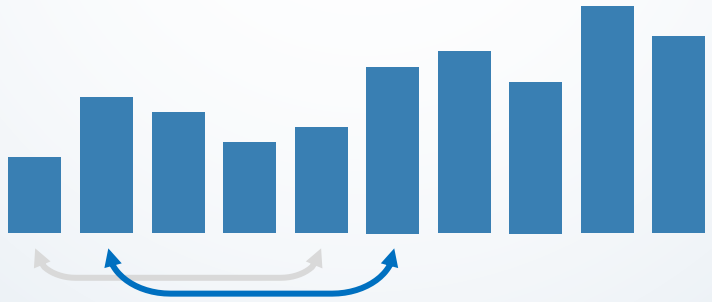
- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

n = 10
h = 4

Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

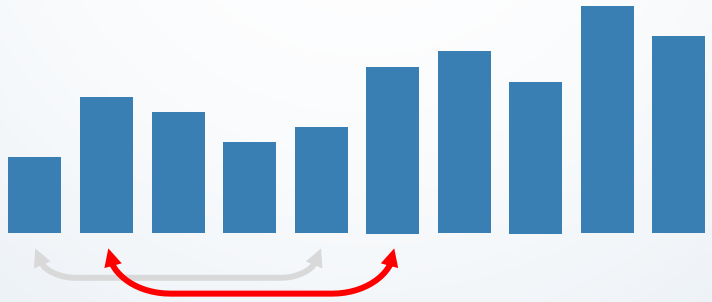
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

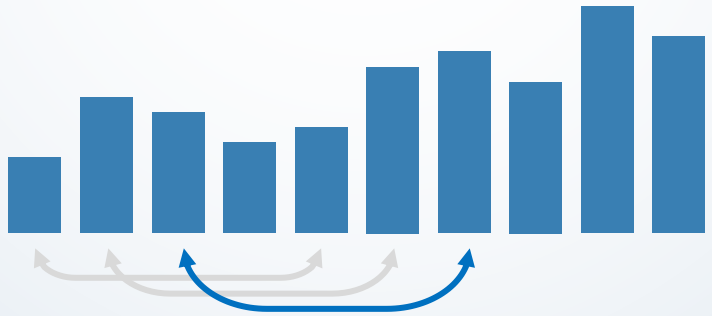
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

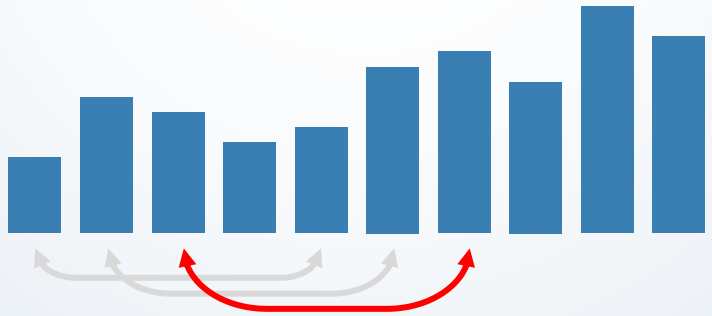
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

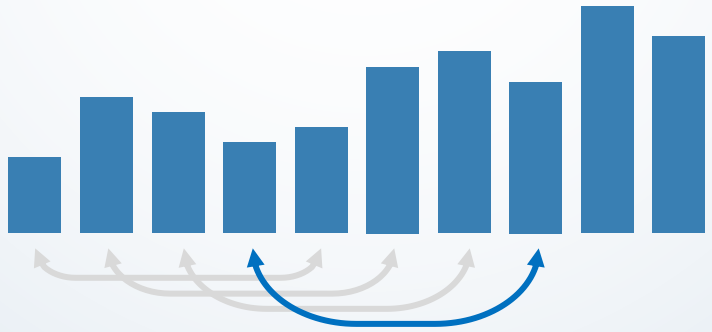
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

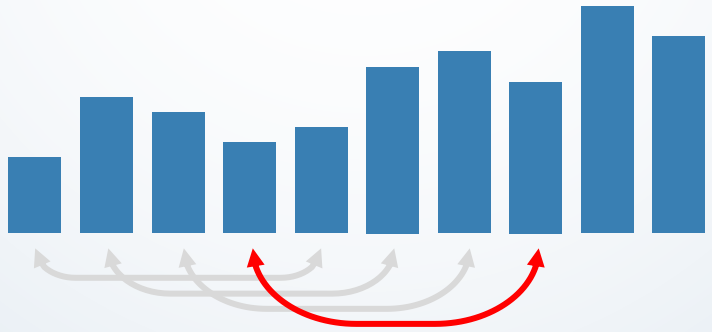
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

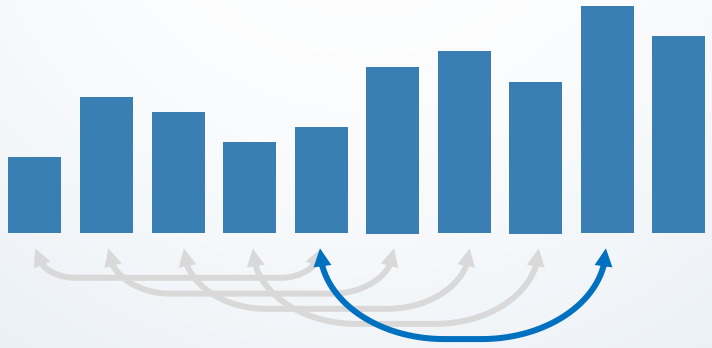
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

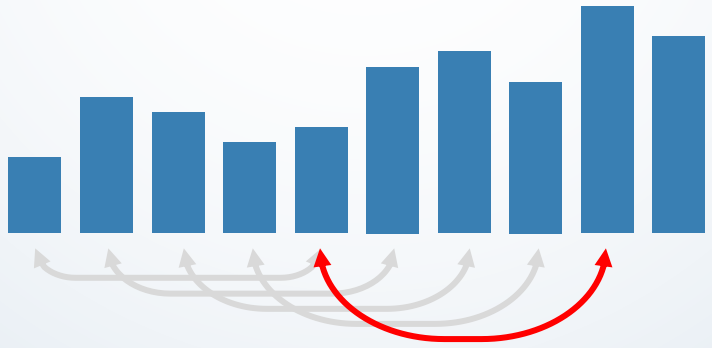
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

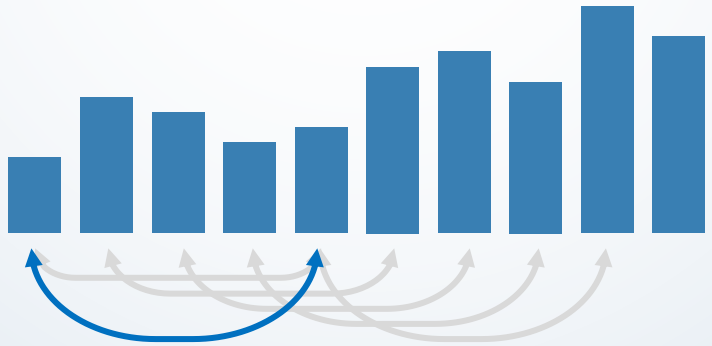
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

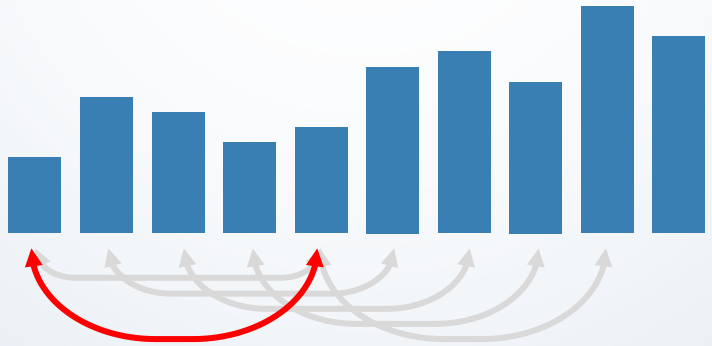
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

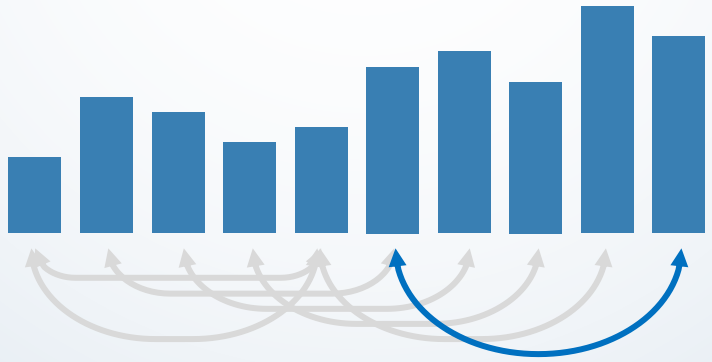
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

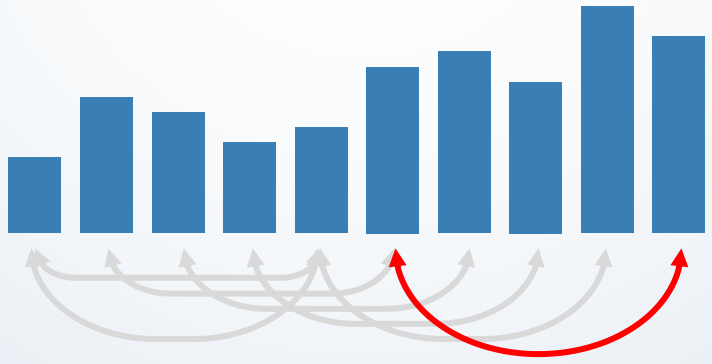
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

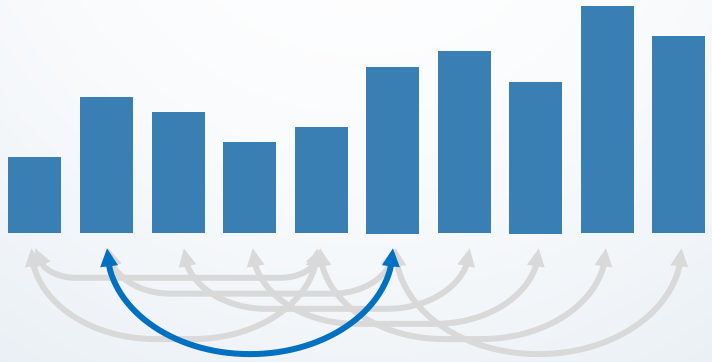
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

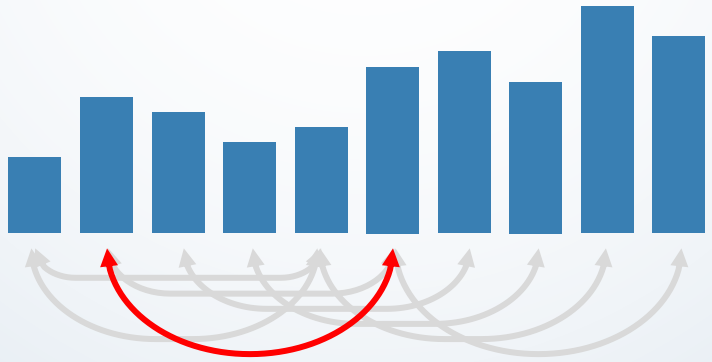
n = 10
h = 4



Shell Sort

- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

n = 10
h = 4



Shell Sort

- Os valores dos dados interferem na execução do algoritmo;
- A sequência de espaçamento interfere na execução do algoritmo;
- In-place? Sim
- Estável? Não
- Complexidade:
  - Pior caso: ? ➔ depende da sequência!
  - Caso médio: ? ➔ depende da sequência!
  - Melhor caso: $O(n)$

Shell Sort

```
public static void sort( int[] array ) {
    int h = 1;
    int n = array.length;
    while ( h < n / 3 ) {
        h = 3 * h + 1; // 1, 4, 13, 40...
    }
    while ( h >= 1 ) {
        for ( int i = h; i < n; i++ ){
            int j = i;
            while ( j >= h && array[j-h] > array[j] ) {
                swap( array, j-h, j );
                j = j - h;
            }
        }
        h = h / 3;
    }
}
```

i
controla a iteração dentro de um espaçamento

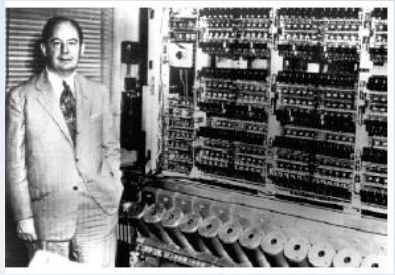
j
controla a iteração dentro de uma seq. de comparação

h
controla o espaçamento das sequências de comparações

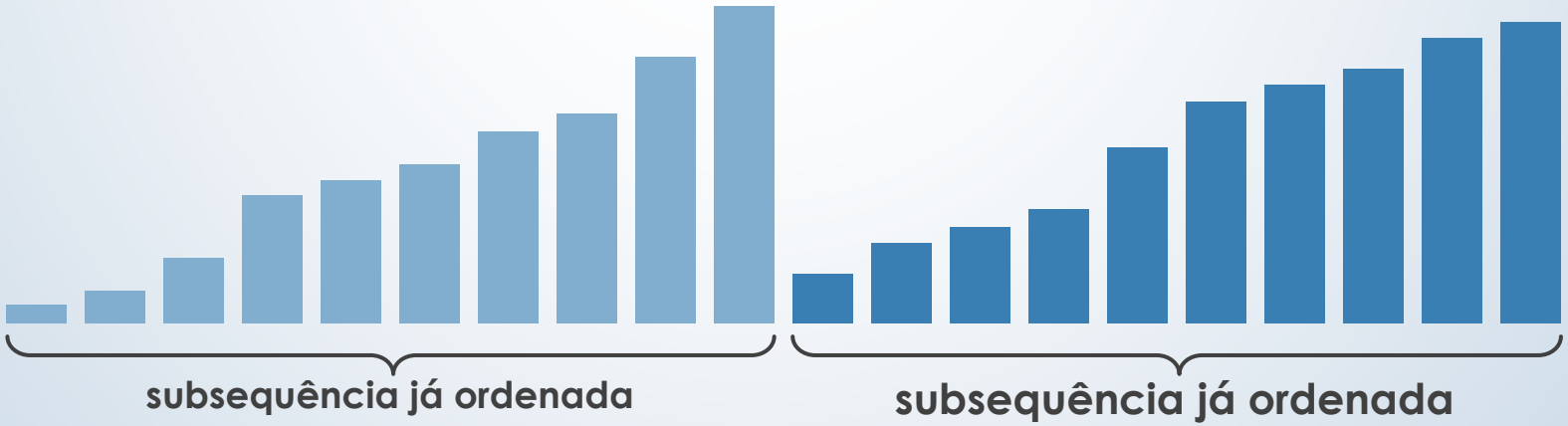
Shell Sort
Estabilidade

h	0	1	2	3	4
	B_1	B_2	B_3	B_4	A_1
4	A_1	B_2	B_3	B_4	B_1
1	A_1	B_2	B_3	B_4	B_1
	A_1	B_2	B_3	B_4	B_1

Merge Sort



- Ordenação por intercalação (Merge Sort):
  - Dividir para conquistar;
  - Divisão da sequência em partes menores para facilitar a ordenação;
  - União de sequências menores já ordenadas, gerando sequências maiores ordenadas;
  - Inventado por John von Neumann em 1959 (EDVAC).

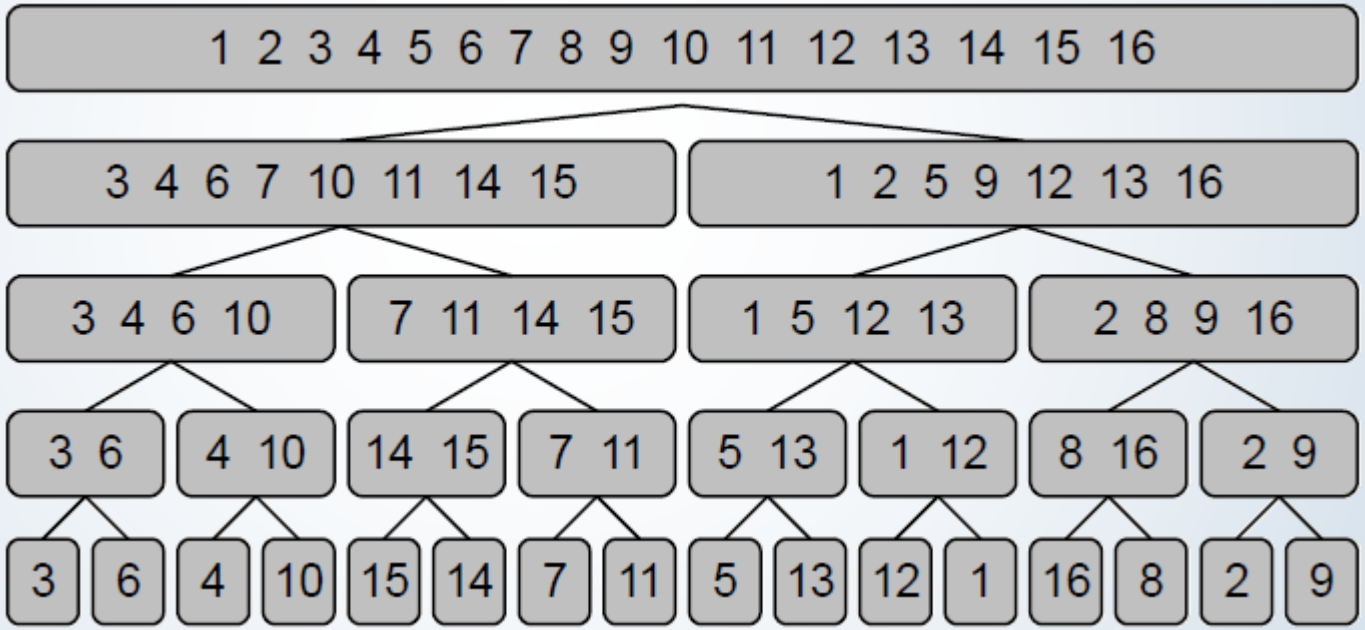


Merge Sort

- Os valores dos dados não-interferem na execução do algoritmo;
- In-place? Não ← usa memória auxiliar!
- Estável? Sim
- Complexidade:
  - Pior caso: $O(n \lg n)$
  - Caso médio: $O(n \lg n)$
  - Melhor caso: $O(n \lg n)$

Merge Sort

- Árvore de divisão (árvore merge)



Merge Sort

- Duas abordagens:
  - Top-Down (Recursiva);
  - Bottom-Up (Iterativa).

Merge Sort
Top-Down

```
public static void sort( int[] array ) {
    int n = array.length;
    int[] tempMS = new int[n];
    topDown( array, 0, n - 1, tempMS );
}

private static void topDown( int[] array, int start, int end, int[] tempMS ) {
    int middle;
    if ( start < end ) {
        middle = ( start + end ) / 2;
        topDown( array, start, middle, tempMS );      // esquerda
        topDown( array, middle + 1, end, tempMS );    // direita
        merge( array, start, middle, end, tempMS );   // intercalação
    }
}
```

start
início do intervalo que será ordenado

end
fim do intervalo que será ordenado

middle
meio do intervalo que será ordenado

Merge Sort
Intercalação

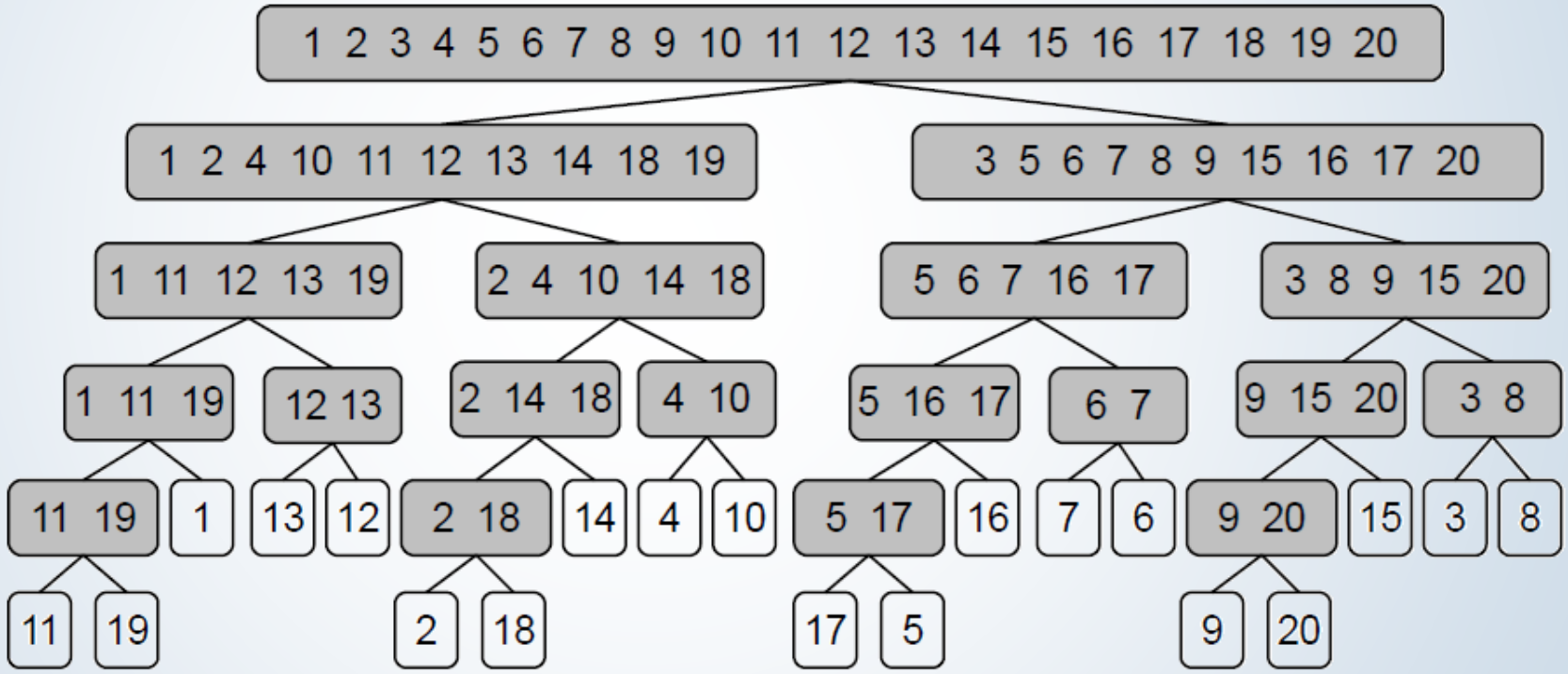
```
private static void merge( int[] array, int start, int middle, int end, int[] tempMS ) {
    int i = start;
    int j = middle + 1;
    for ( int k = start; k <= end; k++ ) {
        tempMS[k] = array[k];
    }
    for ( int k = start; k <= end; k++ ) {
        if ( i > middle ) {
            array[k] = tempMS[j++];
        } else if ( j > end ) {
            array[k] = tempMS[i++];
        } else if ( tempMS[j] < tempMS[i] ) {
            array[k] = tempMS[j++];
        } else {
            array[k] = tempMS[i++];
        }
    }
}
```

i
marca o início do intervalo que será intercalado

j
marca o limite do intervalo que será intercalado

k
usado para iterar entre o início e o fim

Merge Sort
Top-Down



Merge Sort
Bottom-Up

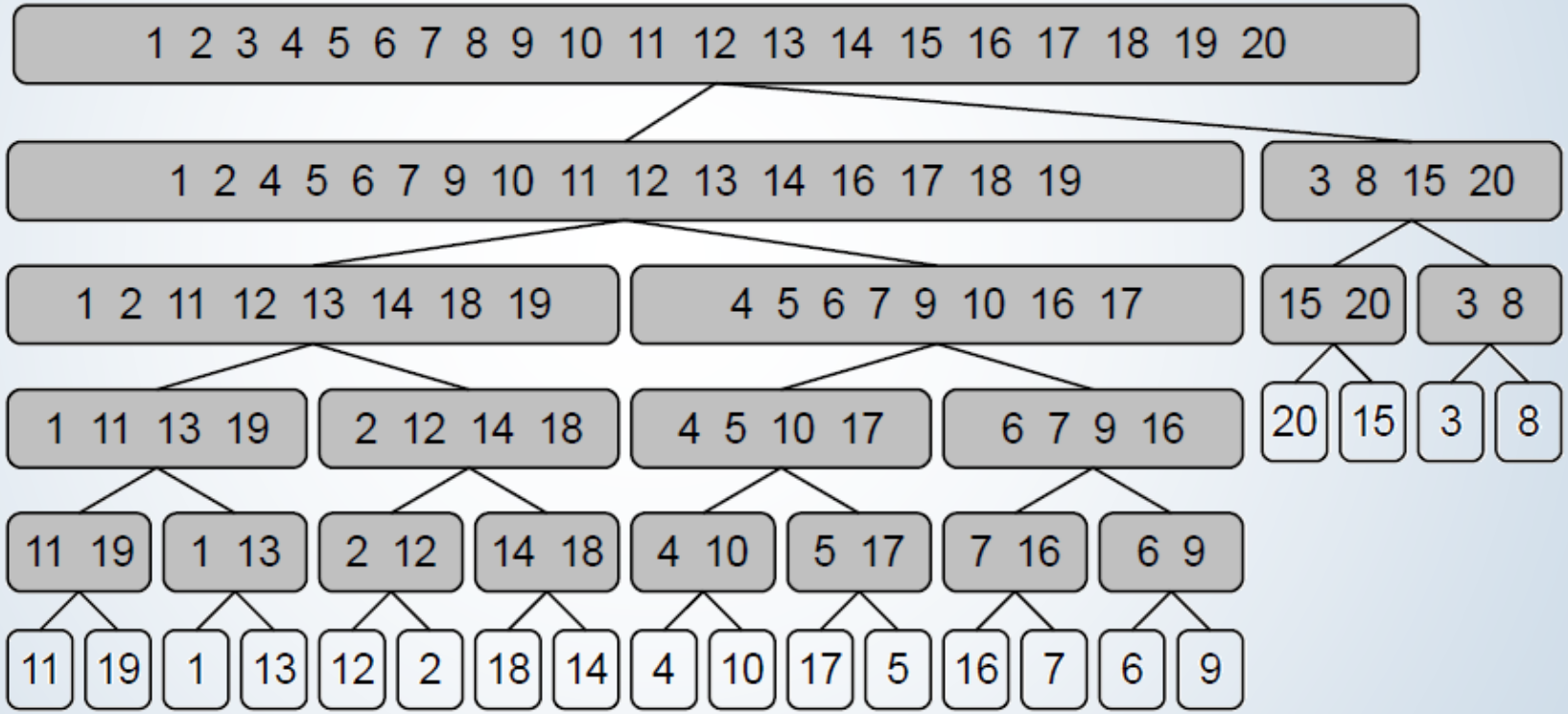
```
public static void sort( int[] array ) {
    int n = array.length;
    int[] tempMS = new int[n];
    bottomUp( array, 0, n - 1, tempMS );
}

private static void bottomUp( int[] array, int start, int end, int[] tempMS ) {
   for ( int m = 1; m <= end; m *= 2 ) {
       for ( int i = start; i <= end - m; i += 2*m ) {
           merge( array, i, i+m-1, Math.min( i+2*m-1, end ), tempMS );
       }
   }
}
```

m
controla o espaçamento que é proporcional ao nível atual da árvore, ou seja, 1 para o último nível, 2 para o penúltimo, 4 para o ante-penúltimo...

i
controla o início do intervalo que será ordenado

Merge Sort
Bottom-Up

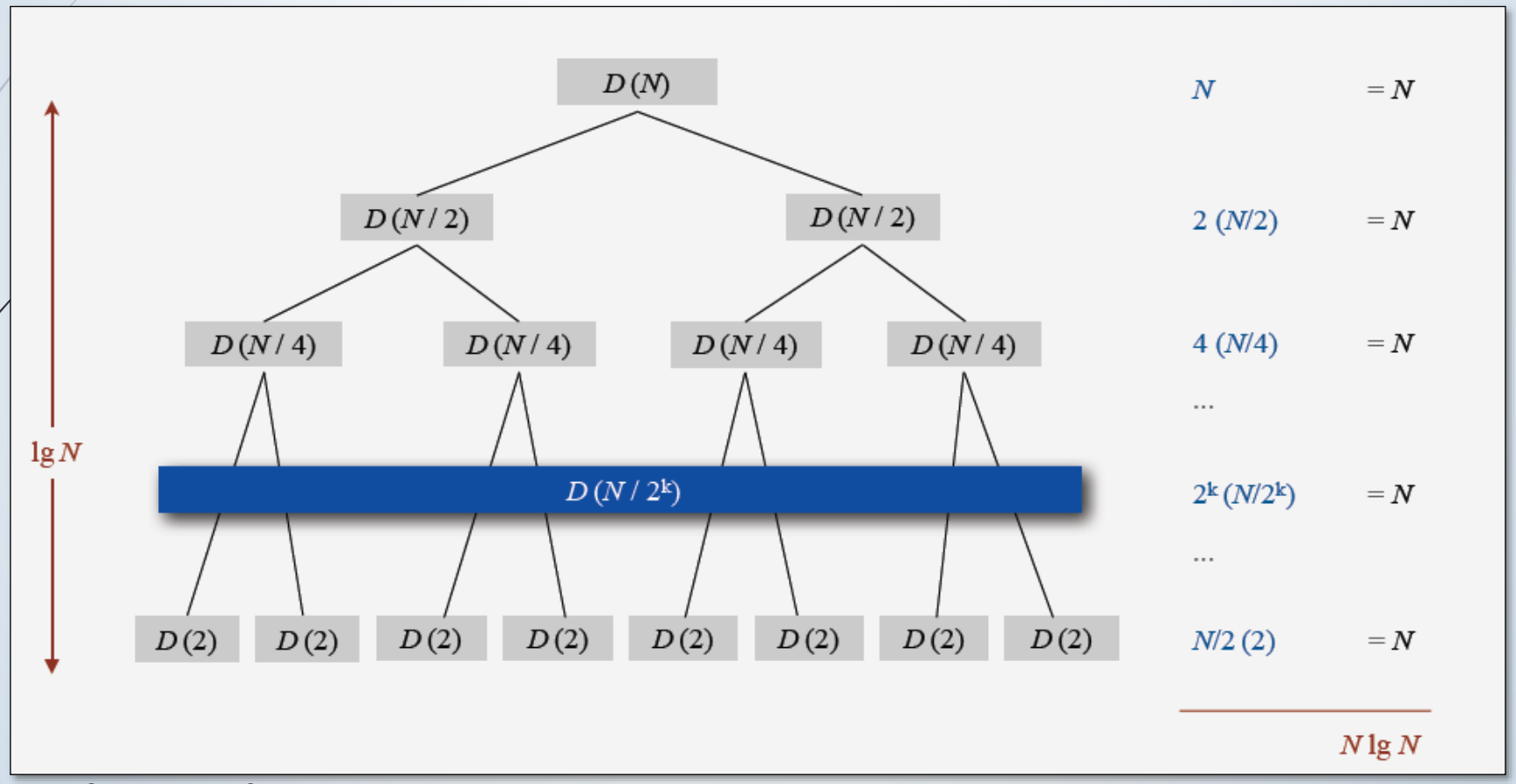


Merge Sort
Estabilidade

- A operação de intercalação (merge) é estável.

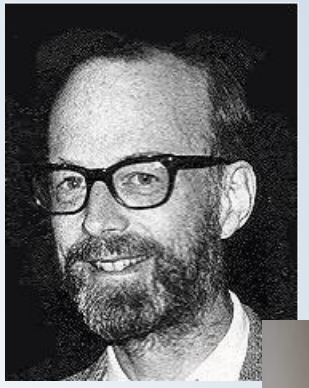
0	1	2	3	4		5	6	7	8	9	10
A_1	A_2	A_3	B	D		A_4	A_5	C	E	F	G

Merge Sort
Complexidade

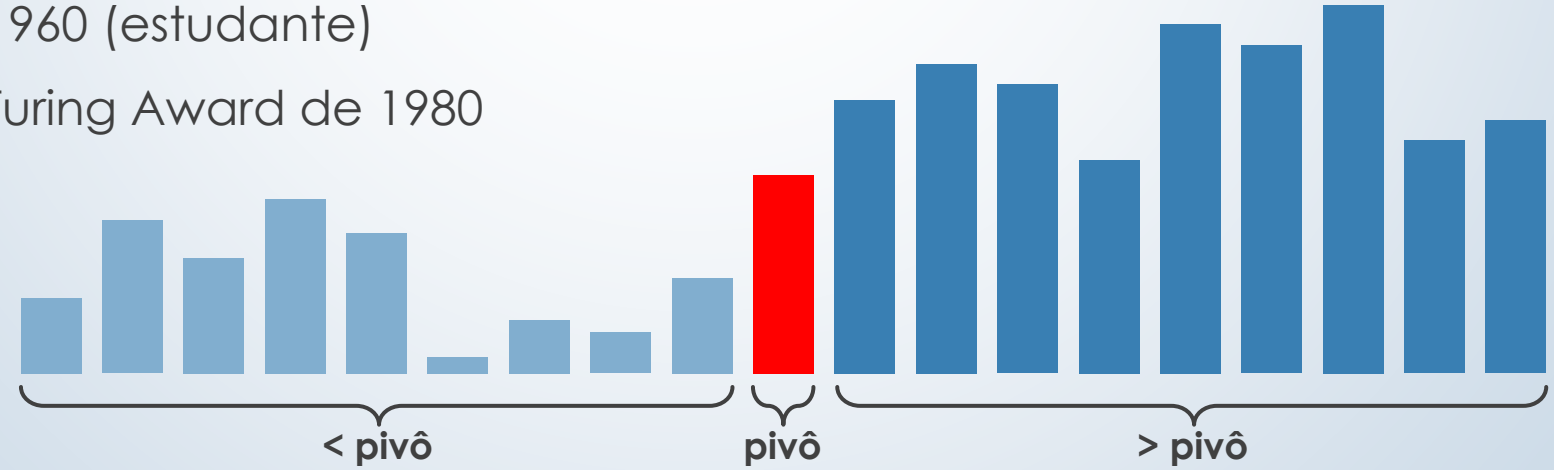


k: nível da árvore

Quick Sort



- Ordenação "Rápida" (Quick Sort):
  - Escolha de um elemento pivô;
  - Separação da sequência em duas partes:
    - Elementos menores que o pivô;
    - Elementos maiores que o pivô;
  - Pivô não precisa mais ser movido!
  - Inventado por Sir Charles A. R. Hoare
    - 1960 (estudante)
    - Turing Award de 1980



Quick Sort

- In-place? Sim
- Estável? Não
- Complexidade:
  - Pior caso: $O(n^2)$ ← !!!
  - Caso médio: $O(n \lg n)$
  - Melhor caso: $O(n \lg n)$

Quick Sort

```java
public static void sort( int[] array ) {
    quickSort( array, 0, array.length - 1 );
}

private static void quickSort( int[] array, int start, int end ) {
    if ( start < end ) {
        // particionamento, calcula posição do meio
        int middle = partition( array, start, end );
        quickSort( array, start, middle - 1 );   // esquerda
        quickSort( array, middle + 1, end );     // direita
    }
}
```

middle
marca o meio
(relativo ao pivô)

esquerda
elementos menores
que o pivô

direita
elementos maiores
que o pivô

Quick Sort

Particionamento

```
private static int partition( int[] array, int start, int end ) {
    int i = start;
    int j = end + 1;
    while ( true ) {
        while ( array[++i] < array[start] ) {
            if ( i == end ) {
                break;
            }
        }
        while ( array[--j] > array[start] ) {
            if ( j == start ) {
                break;
            }
        }
        if ( i >= j ) {
            break;
        }
        swap( array, i, j );
    }
    swap( array, start, j );
    return j;
}
```

start
posição do pivô
("primeiro" elemento)

i
iteração entre os elementos menores que o pivô

j
iteração entre os elementos maiores que o pivô

Quick Sort
Estabilidade

i	j	0	1	2	3
		B_1	C_1	C_2	A_1
1	3	B_1	C_1	C_2	A_1
1	3	B_1	A_1	C_2	C_1
0	1	A_1	B_1	C_2	C_1

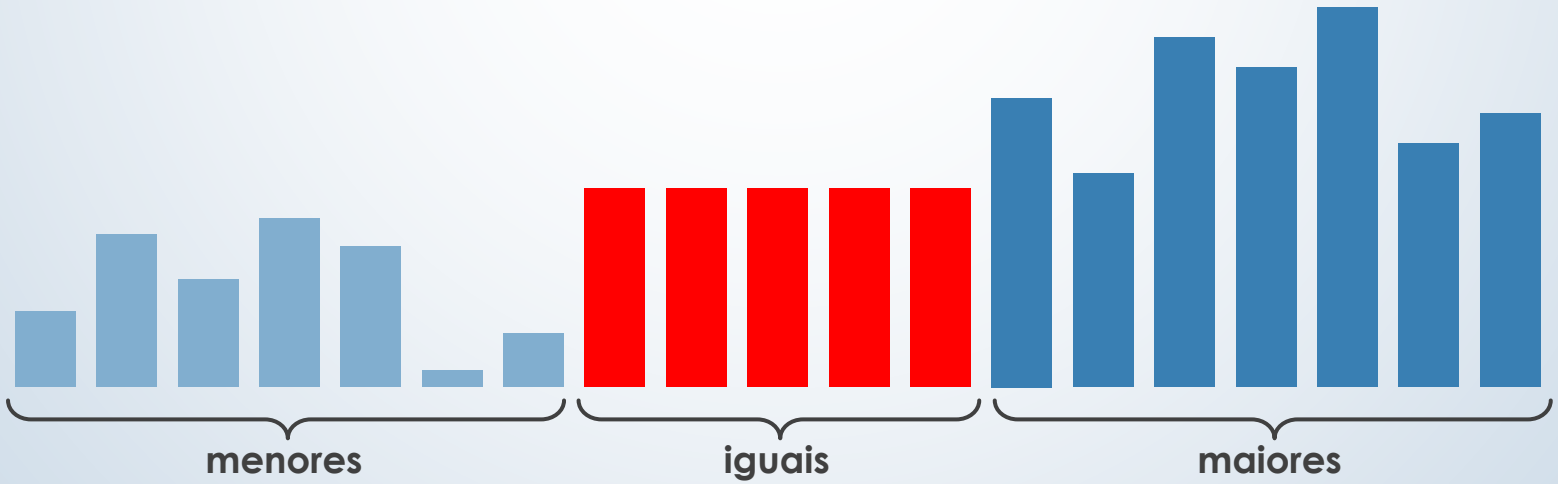
Quick Sort

- Problemas:
  - Pior caso: $O(n^2)$ ← !!!
    - Solução? Melhorar a escolha do pivô!
      - Embaralhar o array antes de ordenar;
      - Mediana de uma amostra;
      - Posição randômica;
  - Chaves duplicadas? (bug encontrado na década de 1990)
    - Solução: Dijkstra 3-way partitioning

Quick Sort 3-way

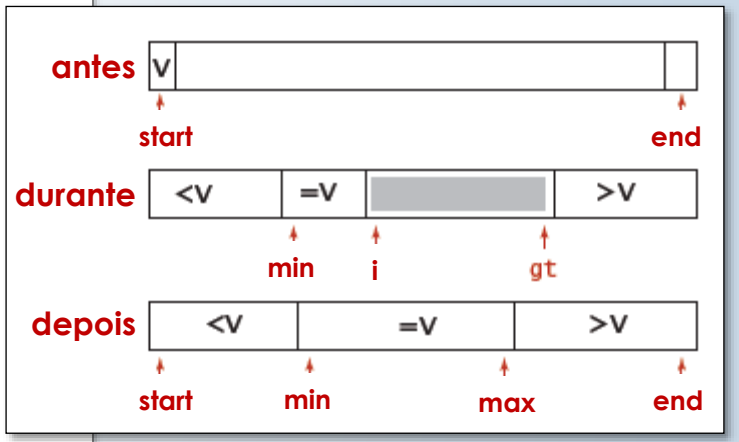
- Resolve o problema das chaves duplicadas, dividindo o array em três faixas:
  - menores – iguais – maiores
- Solução do "*Dutch National Flag Problem*" proposto por Edsger Dijkstra.

Edsger Dijkstra
Turing Award de 1972

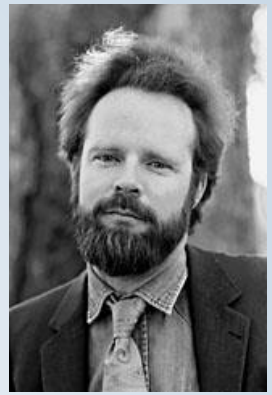


Quick Sort 3-way

```
public static void sort( int[] array ) {
    quickSort3( array, 0, array.length - 1 );
}
public static void quickSort3( int[] array, int start, int end ) {
    if ( start < end ) {
        int min = start;
        int max = end;
        int i = start + 1;
        int v = array[start];
        while ( i <= max ) {
            if ( array[i] < v ) {
                swap( array, min++, i++ );
            } else if ( array[i] > v ) {
                swap( array, i, max-- );
            } else {
                i++;
            }
        }
        quickSort3( array, start, min - 1 );
        quickSort3( array, max + 1, end );
    }
}
```



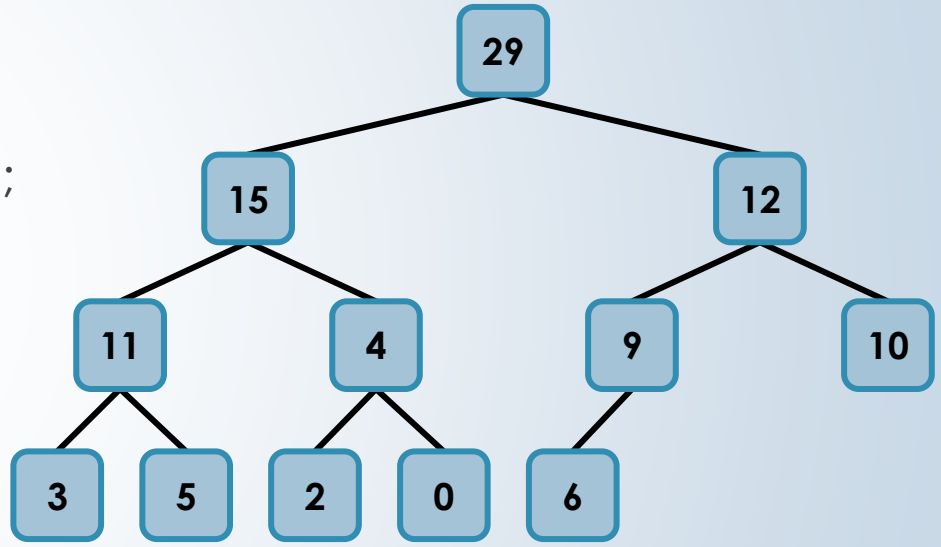
Heap Sort



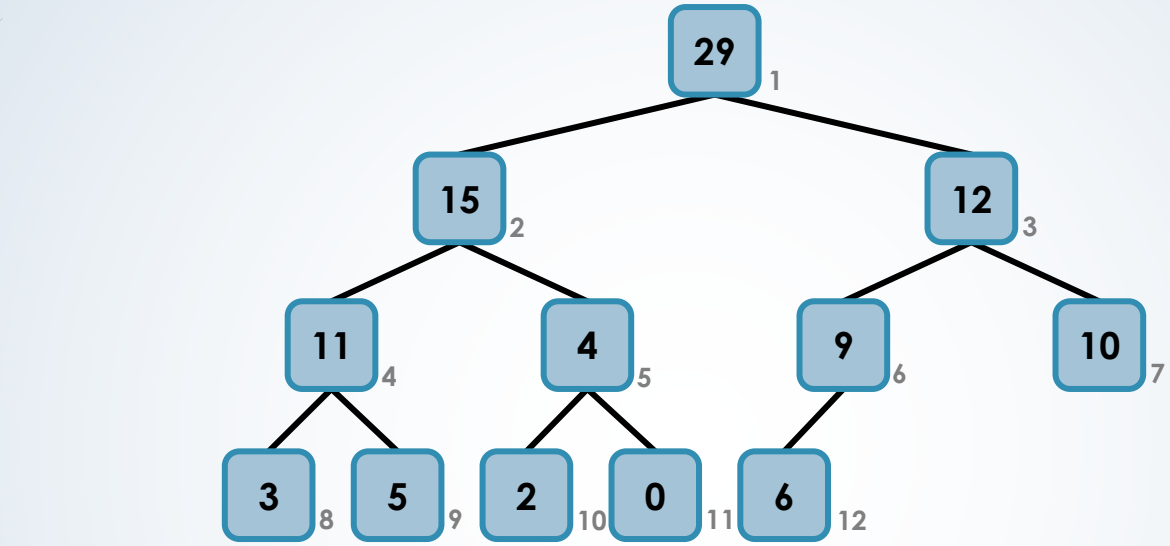
- Ordenação usando um Heap Binário (Heap Sort):
  - Critério de ordenação:
    - **Max-Heap:** Elemento pai sempre maior ou igual aos filhos;
    - **Min-Heap:** Elemento pai sempre menor ou igual aos filhos;
  - Chaves armazenadas nos nós;
  - Utilizaremos apenas:
    - Árvores binárias (até dois filhos);
    - Completa: elementos sem filhos apenas no último nível (e anterior, quando o último nível não está completo);
    - Max-heap;
  - Inventado por Robert W. Floyd e J. W. J. Williams em 1964;
    - Robert Floyd: Turing Award de 1978.

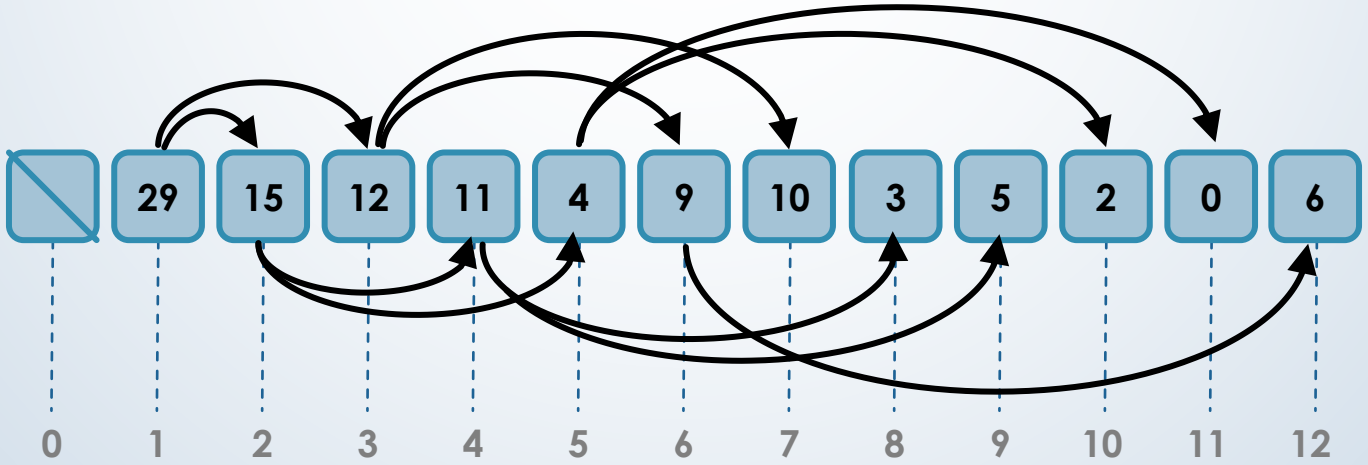
Heap Sort

- Heap (monte) binário (árvore binária completa):
  - **Armazenamento direto em array:**
    - Raiz na posição 1;
    - Último elemento na posição $tamanho - 1$;
  - **Manipulação dos índices:**
    - **Pai:** $posição\ do\ filho\ /\ 2$;
    - **Filho da esquerda:** $posição\ do\ pai * 2$;
    - **Filho direita:** $posição\ do\ pai * 2 + 1$.



Heap Sort


29
1
15
2
12
3
11
4
4
5
9
6
10
7
3
8
5
9
2
10
0
11
6
12


29
15
12
11
4
9
10
3
5
2
0
6
0
1
2
3
4
5
6
7
8
9
10
11
12

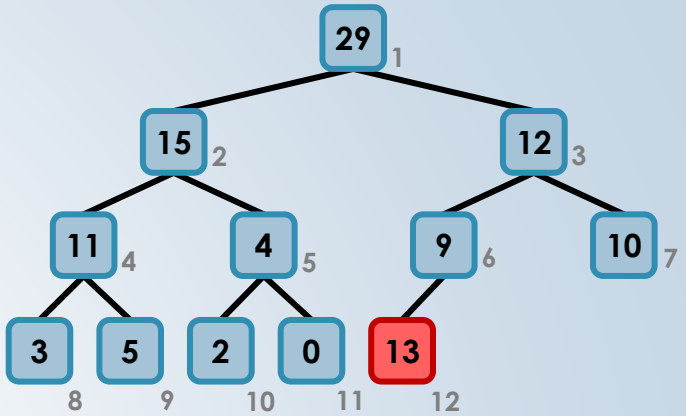
Heap Sort

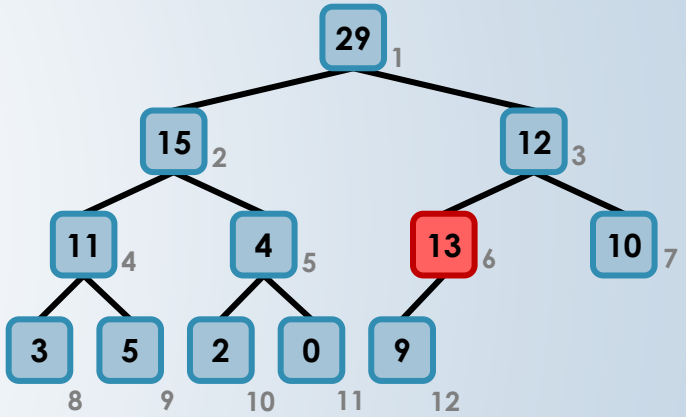
- Heap Binário Máximo (Max-Heap)
  - **Invariante:** chave do nó pai é sempre maior ou igual às chaves dos nós filhos;
- Elemento violando a invariante:
  - **Chave do filho maior que a chave do pai:**
    - O elemento precisa "subir" na árvore;
    - Bottom-up *reheapify* (*swim* ➔ flutuar);
  - **Chave do pai menor que a chave dos filhos (um ou dois):**
    - O elemento precisa "descer" na árvore;
    - Top-down *reheapify* (*sink* ➔ afundar).

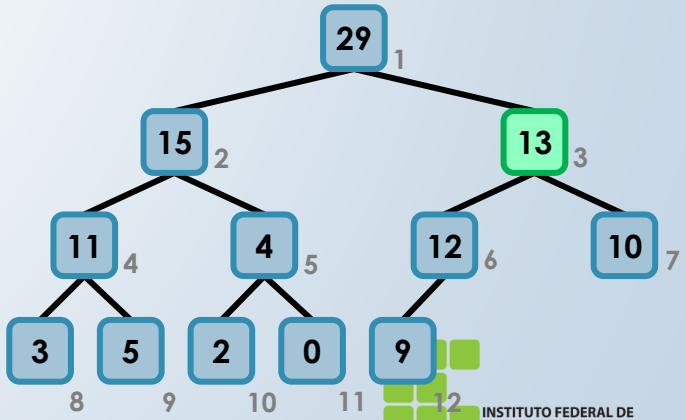
Heap Sort
Bottom-Up Reheapify

```
private static void swim( int[] array, int k ) {
    while ( k > 1 && array[k/2] < array[k] ) {
        swap( array, k/2, k );
        k = k / 2;
    }
}
```

cálculo da posição do pai $k/2$, onde k é a posição do filho







Heap Sort

Top-Down Reheapify

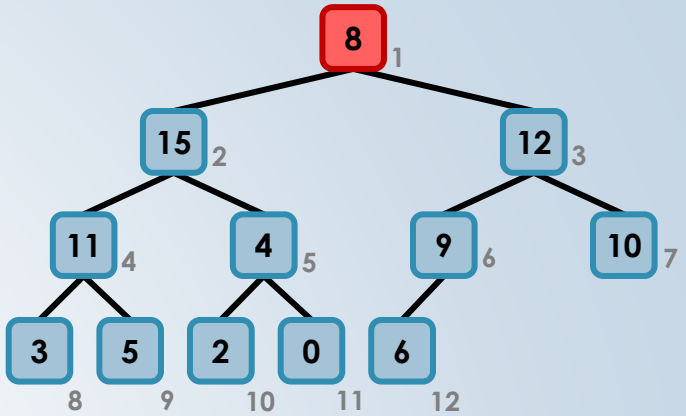
```
private static void sink( int[] array, int k, int n ) {
    while ( 2*k <= n ) {
        int j = 2*k;
        if ( j < n && array[j] < array[j+1] ) {
            j++;
        }
        if ( array[k] >= array[j] ) {
            break;
        }
        swap( array, k, j );
        k = j;
    }
}
```

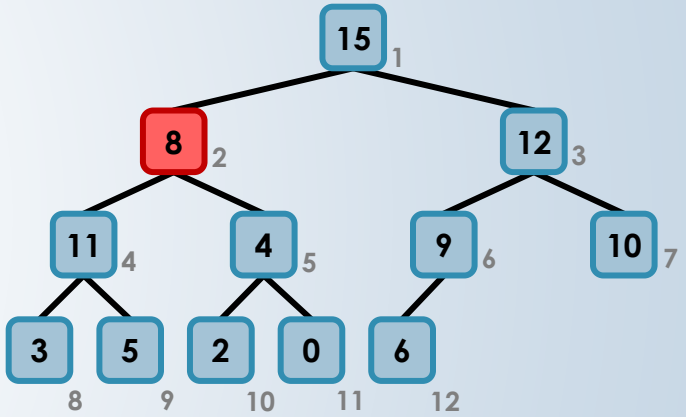
cálculo da posição dos filhos

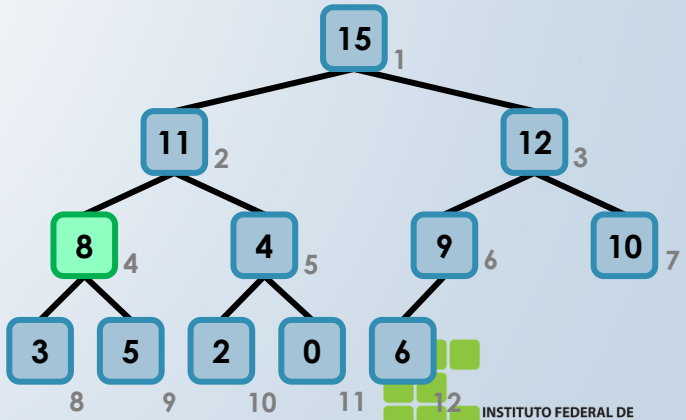
$k * 2$ (esquerda)

$k * 2 + 1$ (direita),

onde k é a posição do pai







Heap Sort

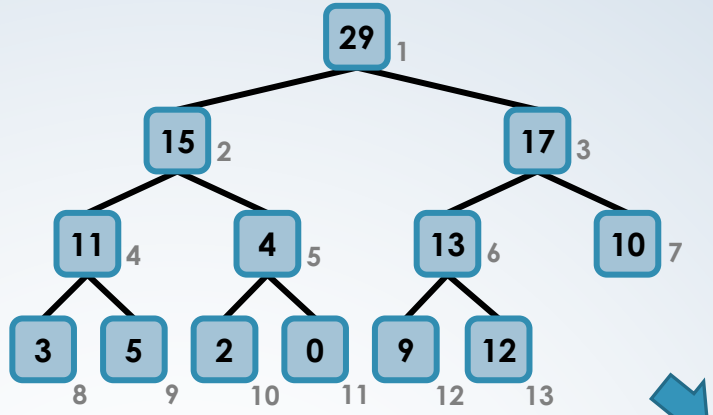
- Ordenação utilizando a estrutura Heap;
- Duas etapas:
  - Construção da estrutura max-heap;
  - Ordenação pela concatenação dos valores máximos obtidos:
    - Iteração: removendo um elemento (maior) por vez do heap;
- Abordagem 1: da esquerda para a direita, adicionar um elemento por vez no heap à esquerda, utilizando o bottom-up;
- Abordagem 2 (mais eficiente): da direta para a esquerda, construir subárvores e unir cada uma delas, utilizando o top-down.

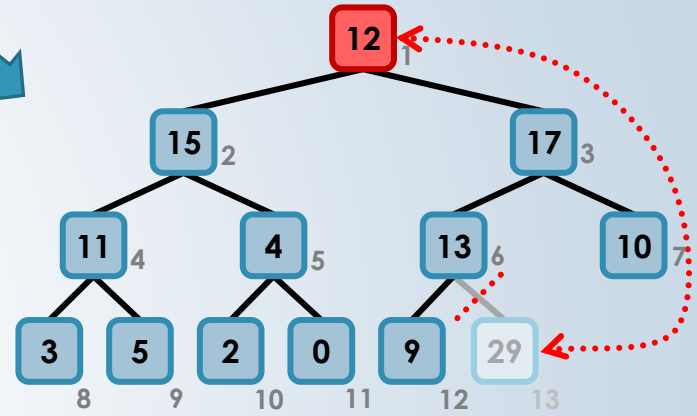
Heap Sort

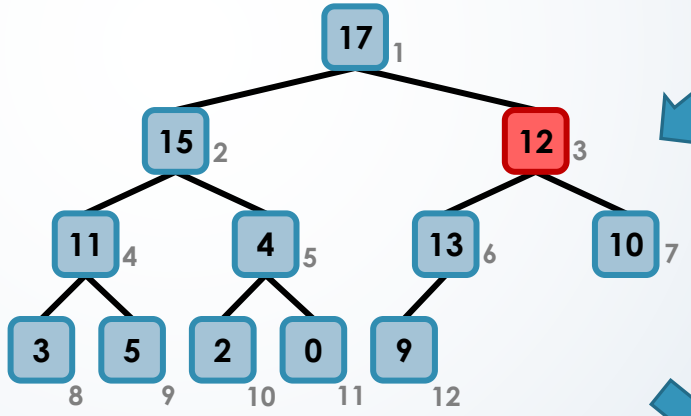
- In-place? Sim
- Estável? Não
- Complexidade:
  - Pior caso: $O(n \lg n)$
  - Caso médio: $O(n \lg n)$
  - Melhor caso: $O(n \lg n)$

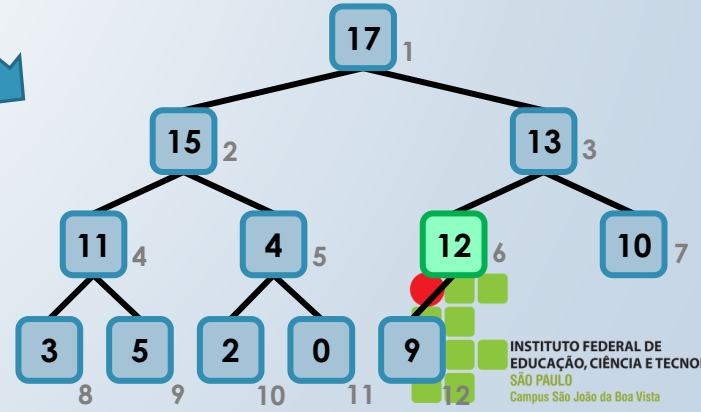
Heap Sort
Abordagem 2

```
public static void sort( int[] array ) {
    int n = array.length - 1;
    for ( int k = n/2; k >= 1; k-- ) {
        sink( array, k, n );
    }
    while ( n > 1 ) {
        swap( array, 1, n-- );
        sink( array, 1, n );
    }
}
```









Algoritmos de Ordenação Lineares

- Existem alguns outros algoritmos de ordenação que não lidam necessariamente com a comparação dos dados que serão ordenados, mas sim com algum tipo de padrão que se pode extrair dos mesmos;
- Alguns desses algoritmos são:
  - Bucket Sort;
  - Counting Sort;
  - Radix Sort;
- Aqui usaremos essa nomenclatura como se fossem apenas um algoritmo de cada tipo, mas na verdade são famílias de algoritmos.

Bucket Sort

- No algoritmo Bucket Sort (ou Bin Sort), os valores do array são distribuídos em buckets (baldes) e então cada um dos buckets é ordenado por algum outro algoritmo de ordenação, permitindo que finalmente os buckets sejam concatenados.
- In-place? Não
- Estável? Sim, desde que o algoritmo que ordena os buckets também seja.
- Complexidade:
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n + k)$
  - Melhor caso: $O(n + k)$

Bucket Sort

```
public static void sort( int[] array ) {
    int n = array.length;
    final int K = 10;
    int[][] buckets = new int[K][n];
    int[] c = new int[K];
    int t1 = 10;
    int t2 = 1;
    int max = -1;
    boolean first = true;
    while ( max < 0 || max / t2 != 0 ) {
        // distribuição
        for ( int i = 0; i < n; i++ ) {
            int p = array[i] % t1 / t2;
            buckets[p][c[p]++] = array[i];
            if ( first ) {
                max = max < array[i] ? array[i] : max;
            }
        }
        first = false;
        // coleta
        int k = 0;
        for ( int i = 0; i < K; i++ ) {
            for ( int j = 0; j < c[i]; j++ ) {
                array[k++] = buckets[i][j];
            }
            c[i] = 0;
        }
        t2 = t1;
        t1 *= 10;
    }
}
```

No exemplo implementado, a ordenação é feita por sucessivas distribuições baseadas nas posições dos números das unidades, depois dezenas, depois centenas etc, ou seja, indo do algarismo menos significativo em direção ao algarismo mais significativo.

buckets
os buckets para a distribuição dos valores

c
array de contadores para a distribuição

t1 e **t2**
auxiliares para geração da posição de distribuição

p
posição da distribuição

k
posição de coleta

Counting Sort

- O algoritmo Counting Sort é uma versão especializada do Bucket Sort, onde o tamanho de cada bucket é um. A ideia do algoritmo se baseia na contagem dos valores que aparecem no array e então no reposicionamento dos mesmos.
- In-place? Não
- Estável? Sim
- Complexidade:
  - Pior caso: $O(n + k)$
  - Caso médio: $O(n + k)$
  - Melhor caso: $O(n + k)$

Counting Sort

```java
public static void sort( int[] array, int k ) {

    int n = array.length;
    int[] c = new int[k+1];
    int[] b = new int[n];

    // contagem
    for ( int i = 0; i < n; i++ ) {
        c[array[i]]++;
    }

    // acumulação
    for ( int i = 1; i <= k; i++ ) {
        c[i] += c[i-1];
    }

    // reposicionamento
    for ( int i = n-1; i >= 0; i-- ) {
        c[array[i]]--;
        b[c[array[i]]] = array[i];
    }

    System.arraycopy( b, 0, array, 0, n );

}
```

k
valor do maior elemento do array

c
array de contagem

b
array de transferência ou de reposicionamento

Radix Sort

- Vocês já estudaram o Radix Sort na disciplina de Projeto e Análise de Algoritmos. Iremos revisitar o funcionamento do Radix Sort quanto formos tratar de ordenação em Strings. Basicamente, o Radix Sort pode ser empregado na ordenação de inteiros e Strings, partindo dos dígitos ou caracteres menos significativos (mais à direita).

Sumário do Crescimento dos Algoritmos de Ordenação

Algoritmo	In-place	Estável	Pior Caso	Caso Médio	Melhor Caso
Selection	x		n^2	n^2	n^2
Insertion	x	x	n^2	n^2	n
Bubble	x	x	n^2	n^2	n
Shell	x		?	?	n
Merge		x	$n \lg n$	$n \lg n$	$n \lg n$
Quick	x		n^2*	$n \lg n$	$n \lg n$
Heap	x		$n \lg n$	$n \lg n$	$n \lg n$
Ideal?	x	x	$n \lg n$	$n \lg n$	$n \lg n$
Bucket		x	n^2	$n + k$	$n + k$
Counting		x	$n + k$	$n + k$	$n + k$

Comparativos

Lineares *(não comparativos)

* Possível resolver

Algoritmos de Busca
Exercícios Escritos

- **Exercício e3.1:** Quantas iterações o algoritmo de busca sequencial irá executar até encontrar o valor 5 no array { 2, 4, 8, -2, 5, 4, 3, 7 }? Justifique sua resposta. R: 5
- **Exercício e3.2:** Quantas iterações o algoritmo de busca sequencial irá executar até encontrar o valor 10 no array { 8, 5, 1, 30, 4, 3, 10, 2, 9 }? Justifique sua resposta. R: 7
- **Exercício e3.3:** Quantas iterações o algoritmo de busca sequencial irá executar até encontrar o valor 90 no array { 7, 6, 3, 4, 5, 10, 11, 43, 29, 5 }? Justifique sua resposta. R: 10
- **Exercício e3.4:** Explique o porque do algoritmo de busca binária precisar que o array que passará pelo processo de pesquisa esteja ordenado.

Algoritmos de Busca
Exercícios Escritos

- **Exercício e3.5:** Quantos cálculos do "meio" o algoritmo de busca binária irá executar até encontrar o valor 5 no array { 3, 4, 5, 6, 7 }? Justifique sua resposta. R: 1
- **Exercício e3.6:** Quantos cálculos do "meio" o algoritmo de busca binária irá executar até encontrar o valor 20 no array { 4, 7, 8, 9, 10, 20, 28, 30 }? Justifique sua resposta. R: 2
- **Exercício e3.7:** Quantos cálculos do "meio" o algoritmo de busca binária irá executar até não encontrar o valor 37 no array { 2, 5, 7, 19, 25, 56, 66, 78, 103 }? Justifique sua resposta. R: 3

Algoritmos de Ordenação Comparativos
Exercícios Escritos

- **Exercício e3.8:** O que caracteriza um algoritmo de ordenação como estável? Para facilitar sua explicação, você pode usar exemplos.
- **Exercício e3.9:** Quais algoritmos de ordenação estudados são estáveis?
- **Exercício e3.10:** Quais algoritmos de ordenação estudados não são estáveis?
- **Exercício e3.11:** Quais algoritmos de ordenação estudados são in-place?
- **Exercício e3.12:** Quais algoritmos de ordenação estudados não são in-place? Por quê?
- **Exercício e3.13:** Qual a ideia usada nos algoritmos abaixo para que a ordenação seja feita?
  a) Selection Sort
  b) Insertion Sort
  c) Bubble Sort
  d) Shell Sort
  e) Merge Sort
  f) Quick Sort
  g) Heap Sort

Algoritmos de Ordenação Comparativos
Exercícios Escritos

- **Exercício e3.14:** Explique por que o algoritmo Selection Sort é $O(n^2)$ no pior caso.
- **Exercício e3.15:** Explique por que o algoritmo Insertion Sort é $O(n^2)$ no pior caso.
- **Exercício e3.16:** Explique por que o algoritmo Bubble Sort é $O(n^2)$ no pior caso.
- **Exercício e3.17:** Explique por que o algoritmo Merge Sort é $O(n \lg n)$ nos três casos.
- **Exercício e3.18:** Explique por que o algoritmo Quick Sort é $O(n^2)$ no pior caso. O que se pode fazer para resolver esse problema?
- **Exercício e3.19:** Explique por que o algoritmo Heap Sort é $O(n \lg n)$ nos três casos.

Algoritmos de Ordenação Comparativos
Exercícios Escritos

- **Exercício e3.20:** Qual o estado do array { 6, 1, 15, 13, 2, 0 } após a primeira passada do algoritmo Selection Sort? R: { 0, 1, 15, 13, 2, 6 }
- **Exercício e3.21:** Qual o estado do array { -6, 5, -5, 15, 7, -13 } após a terceira passada do algoritmo Insertion Sort? R: { -6, -5, 5, 15, 7, -13 }
- **Exercício e3.22:** Qual o estado do array { -15, -9, -16, 14, 0, 5 } após a terceira passada do algoritmo Bubble Sort? R: { -16, -15, -9, 0, 5, 14 }
- **Exercício e3.23:** Qual o estado do array { 2, 16, 10, 16, 7, -13, 19, 1, 15, 19, 11, 11 } após a passagem de espaçamento igual a 4 do algoritmo Shell Sort? R: { 2, -13, 10, 1, 7, 16, 11, 11, 15, 19, 19, 16 }
- **Exercício e3.24:** Qual o estado do array { -10, -2, 17, -4, 16, 16, 9, 3 } após a primeira invocação do algoritmo de intercalação do algoritmo Merge Sort (top-down)? R: { -10, -2, 17, -4, 16, 16, 9, 3 }
- **Exercício e3.25:** Qual o estado do array { -14, 16, 11, -11, 5, -5, 5, 19 } após a segunda invocação do algoritmo de intercalação do algoritmo Merge Sort (top-down)? R: { -14, 16, -11, 11, 5, -5, 5, 19 }

Algoritmos de Ordenação Comparativos
Exercícios Escritos

- **Exercício e3.26:** Qual o estado do array { 2, -13, 3, 15, 9, 14, 6, 2 } após a primeira invocação do algoritmo de intercalação do algoritmo Merge Sort (bottom-up)? R: { -13, 2, 3, 15, 9, 14, 6, 2 }
- **Exercício e3.27:** Qual o estado do array { 0, 13, 4, 3, 4, 1, 15, 11 } após a segunda invocação do algoritmo de intercalação do algoritmo Merge Sort (bottom-up)? R: { 0, 13, 3, 4, 4, 1, 15, 11 }
- **Exercício e3.28:** Qual o estado do array { 17, 19, -3, 6, 12, 13, 5, 8 } após a primeira invocação do algoritmo de particionamento do algoritmo Quick Sort (2-way partitioning)? Qual a posição do pivô usada como base para as chamadas recursivas após a chamada do particionamento acima? R: pivô: 6, array: { 5, 8, -3, 6, 12, 13, 17, 19 }
- **Exercício e3.29:** Qual o estado do array { -13, 0, 5, 19, 14, 18, 0, 8 } após a segunda invocação do algoritmo de particionamento do algoritmo Quick Sort (2-way partitioning)? Qual a posição do pivô usada como base para as chamadas recursivas após a chamada do particionamento acima? R: pivô: 2, array: { -13, 0, 0, 19, 14, 18, 5, 8 }

Algoritmos de Ordenação Comparativos
Exercícios Escritos

- **Exercício e3.30:** Qual o estado do array { 0, 14, 10, -3, 0, 14, 6, -9 } após a construção do max-heap (abordagem 2) para a execução do algoritmo Heap Sort? Desenhe o Heap para facilitar a construção. O primeiro elemento não deve ser considerado como parte do heap. R: { 0, 14, 14, 6, 0, 10, -3, -9 }
- **Exercício e3.31:** Qual o estado do array { 0, -1, -9, -12, 9, 4, 11, 5 } após a construção do max-heap (abordagem 2) para a execução do algoritmo Heap Sort? Desenhe o Heap para facilitar a construção. O primeiro elemento não deve ser considerado como parte do heap. R: { 0, 11, 9, 5, -9, 4, -12, -1 }
- **Exercício e3.32:** Explique com suas palavras o porquê a segunda abordagem para a construção do max-heap é mais eficiente.
- **Exercício e3.33:** Qual o estado do array { 0, 5, -17, 3, 18, 17, -3, 3 } após a primeira troca e reorganização do heap (top-down heapify) durante a execução do algoritmo Heap Sort? Note que o array apresentado ainda não é um max-heap válido, o qual deve ser construído usando a abordagem 2 antes de iniciar o processo de ordenação. R: { 0, 17, 5, 3, -17, 3, -3, 18 }
- **Exercício e3.34:** Qual o estado do array { 0, -8, -5, 6, -14, 15, 0, 5 } após a segunda troca e reorganização do heap (top-down heapify) durante a execução do algoritmo Heap Sort? Note que o array apresentado ainda não é um max-heap válido, o qual deve ser construído usando a abordagem 2 antes de iniciar o processo de ordenação. R: { 0, 5, -5, 0, -14, -8, 6, 15 }

Algoritmos de Ordenação Lineares
Exercícios

- **Exercício e3.35:** Para o array { 1, 4, 3, 5, 7, 9, 19, 22 }, apresente o estado dos buckets e da ordenação do array após a distribuição pelos algarismos da unidade e pela coleta.
- **Exercício e3.36:** Para o array { 2, 8, 14, 24, 33, 31, 78, 7 }, apresente o estado dos buckets e da ordenação do array após a distribuição pelos algarismos da dezena e pela coleta.
- **Exercício e3.37:** Para o array { 214, 8, 117, 5, 95, 35, 9, 181 }, apresente o estado dos buckets e da ordenação do array após a distribuição pelos algarismos da centena e pela coleta.
- **Exercício e3.38:** Para o array { 2, 4, 1, 8, 9, 1, 9 }, apresente a contagem e a contagem acumulada.
- **Exercício e3.39:** Para o array { 0, 1, 2, 3, 4, 5, 6, 7 }, apresente a contagem e a contagem acumulada.
- **Exercício e3.40:** Para o array { 0, 1, 2, 0, 1, 2, 0, 1, 2 }, apresente a contagem e a contagem acumulada.
- **Exercício e3.41:** Para o array { 5, 5, 5, 5, 5, 5, 5, 5 }, apresente a contagem e a contagem acumulada.

Bibliografia

DEITEL, P.; DEITEL, H. **Java: Como Programar**. 10. ed. São Paulo: Pearson, 2017. 934 p.

MORIN, P. **Open Data Structures (in Java)**. [s.l]: [s.n], 2020. 322 p.

SEDGEWICK, R.; WAYNE, K. **Algorithms**. 4. ed. Boston: Pearson Education, 2011. 955 p.

GOODRICHM M. T.; TAMASSIA, R. **Estruturas de Dados & Algoritmos em Java**. Porto Alegre: Bookman, 2013. 700 p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos – Teoria e Prática**. 3. ed. São Paulo: GEN LTC, 2012. 1292 p.