~~MAIS UM CURSO DE~~ JAVA E PROGRAMAÇÃO ORIENTADA A OBJETOS ~~QUE NINGUÉM PEDIU~~

COM INTRODUÇÃO À ENGINE DE JOGOS JSGE E EXERCÍCIOS CRIATIVOS

ALGORITMOS DE ORDENAÇÃO COMPARATIVOS

PROF. DR. DAVID BUZATTO

PRINCIPAIS ALGORITMOS DE ORDENAÇÃO COMPARATIVOS

- Selection Sort
- Insertion Sort
- Bubble Sort

} Elementares

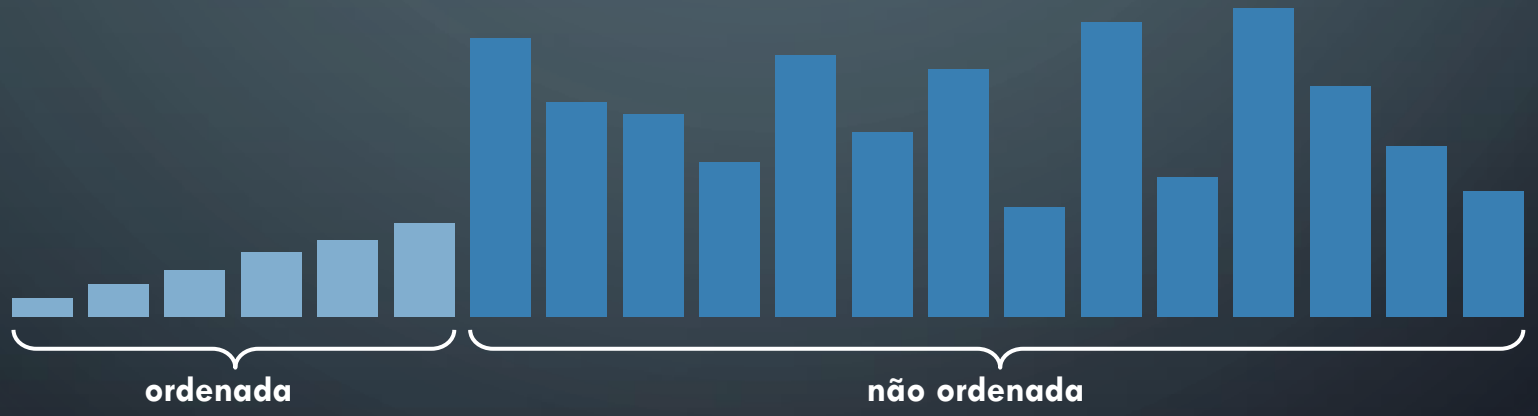
- Shell Sort

- 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 length = array.length;
    for ( int i = 0; i < length; i++ ) {
        int min = i;
        for ( int j = i + 1; j < length; 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

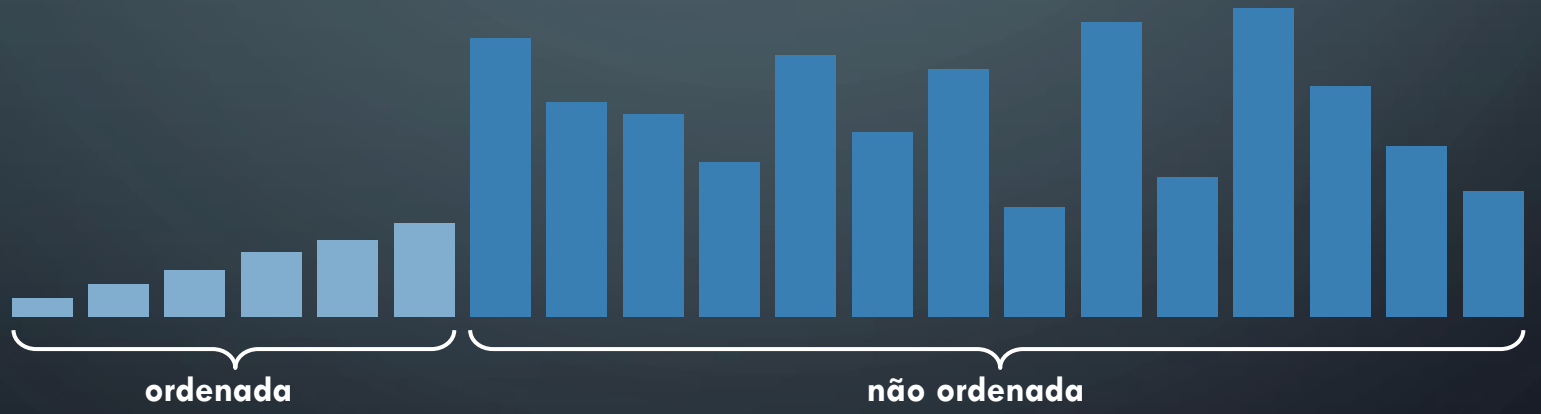
- Alunos da seção 3 não estão mais ordenados por nome!

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

i
índice da sequência ordenada

j
índice da sequência não-ordenada

swapped
indica se houve ou não troca

SHELL SORT

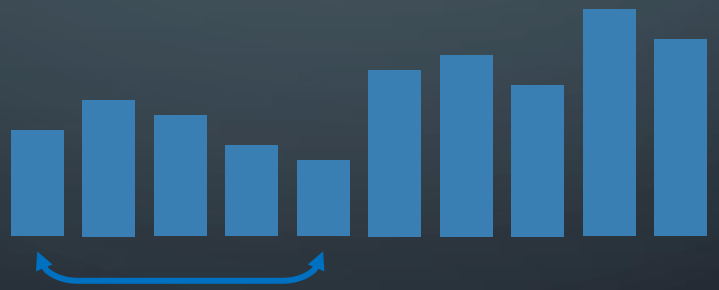
- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

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

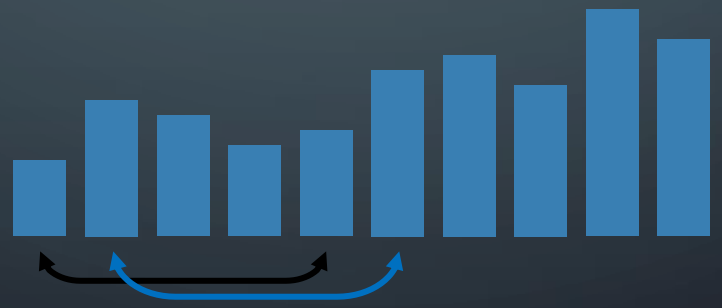
- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

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

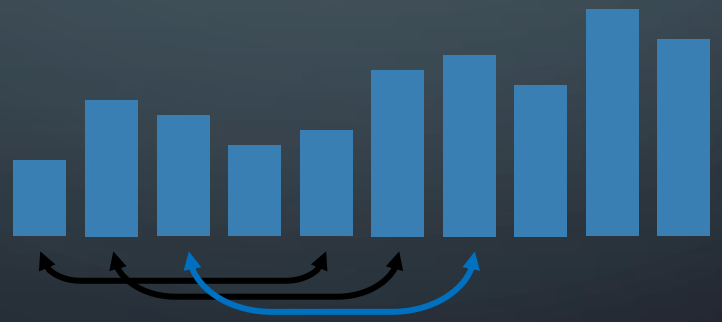
- Ordenação Shell (Shell Sort):
  - Percorrer a sequência e mover os elementos mais de uma posição por comparação (h-sorting);
  - Decrementar o valor de h e repetir o processo;
  - Inventado por Donald Shell (1959).

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).

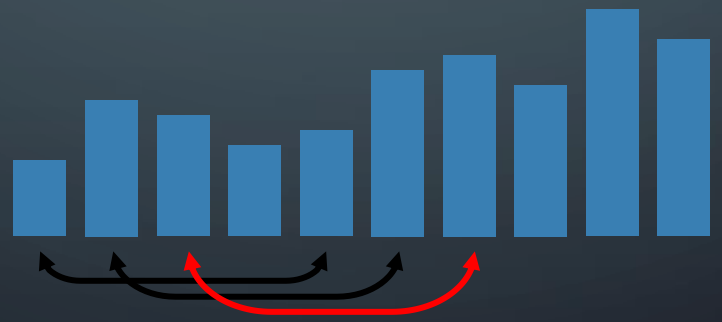
n = 10
h = 4



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).

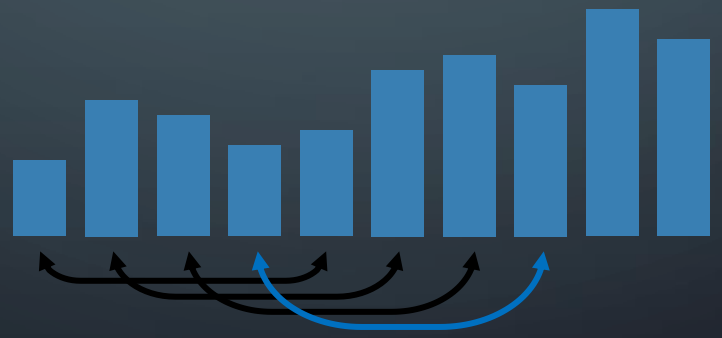
n = 10
h = 4



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).

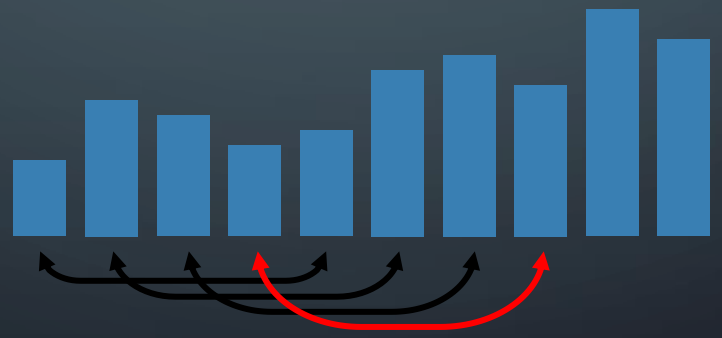
n = 10
h = 4



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).

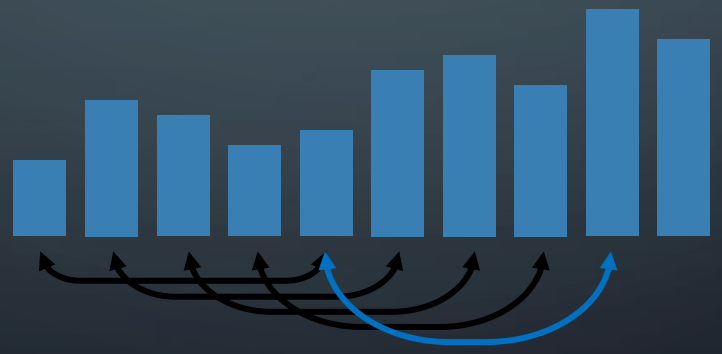
n = 10
h = 4



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).

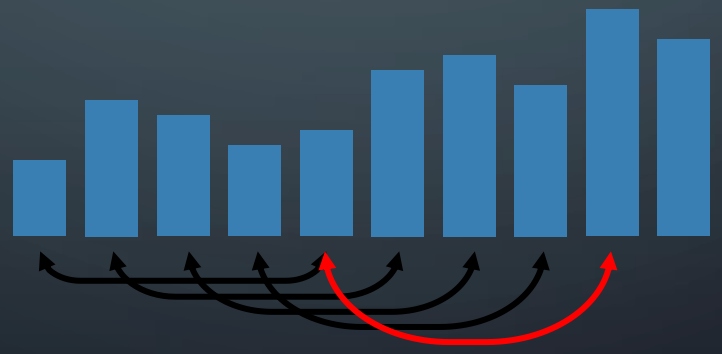
n = 10
h = 4



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).

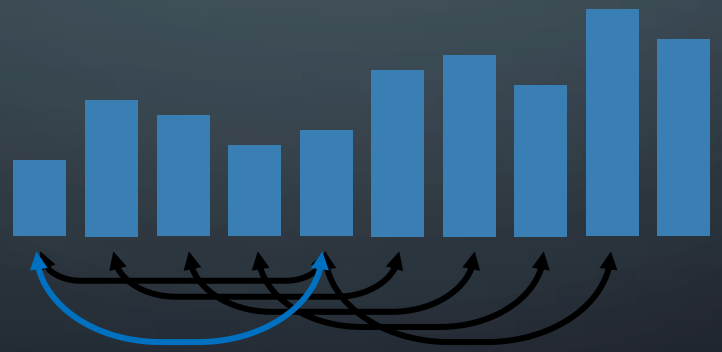
n = 10
h = 4



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).

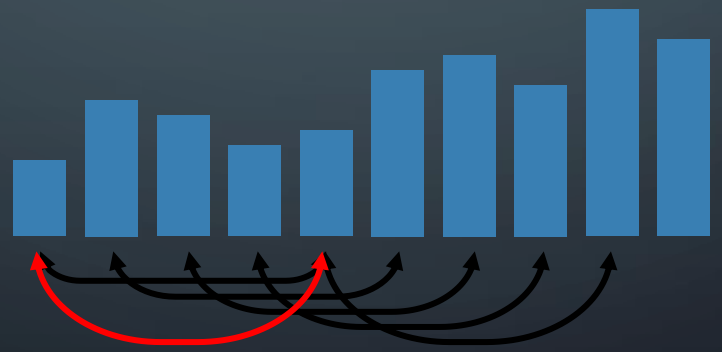
n = 10
h = 4



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).

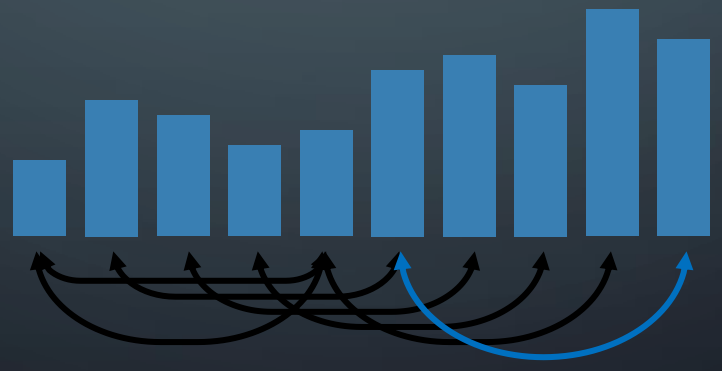
n = 10
h = 4



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).

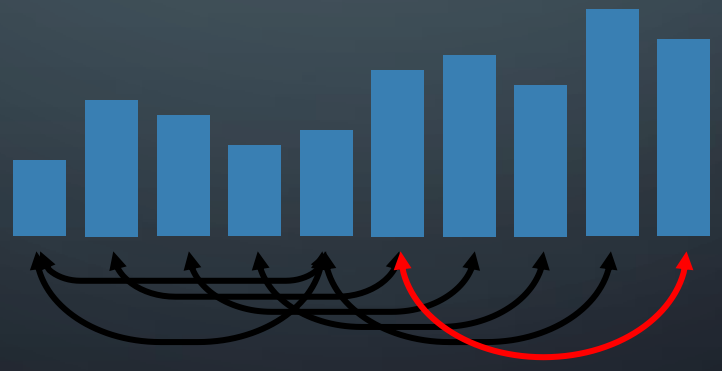
n = 10
h = 4



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).

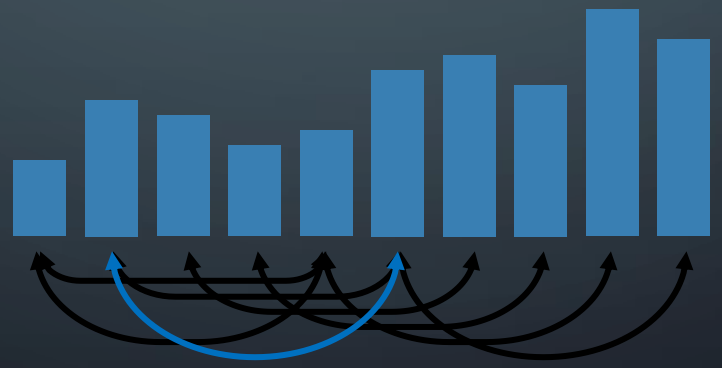
n = 10
h = 4



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).

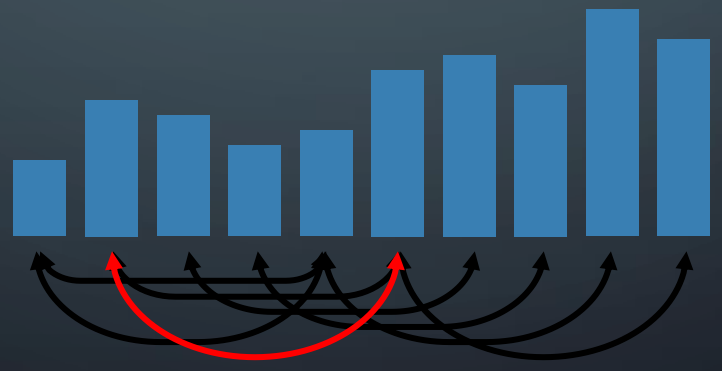
n = 10
h = 4



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 length = array.length;
    while ( h < length / 3 ) {
        h = 3 * h + 1; // 1, 4, 13, 40...
    }
    while ( h >= 1 ) {
        for ( int i = h; i < length; 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

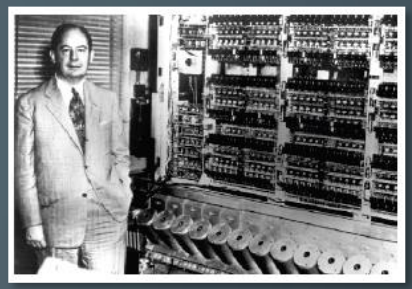
j
controla a iteração dentro
de uma seq. de comparação

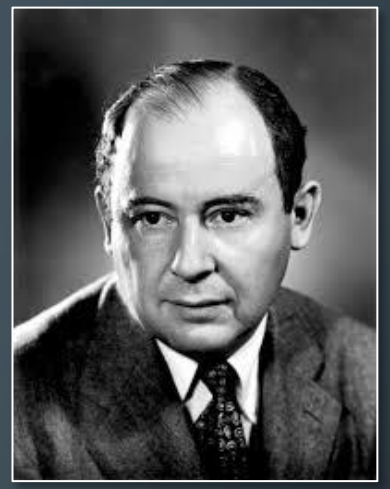
h
controla o espaçamento das
sequências de comparações

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





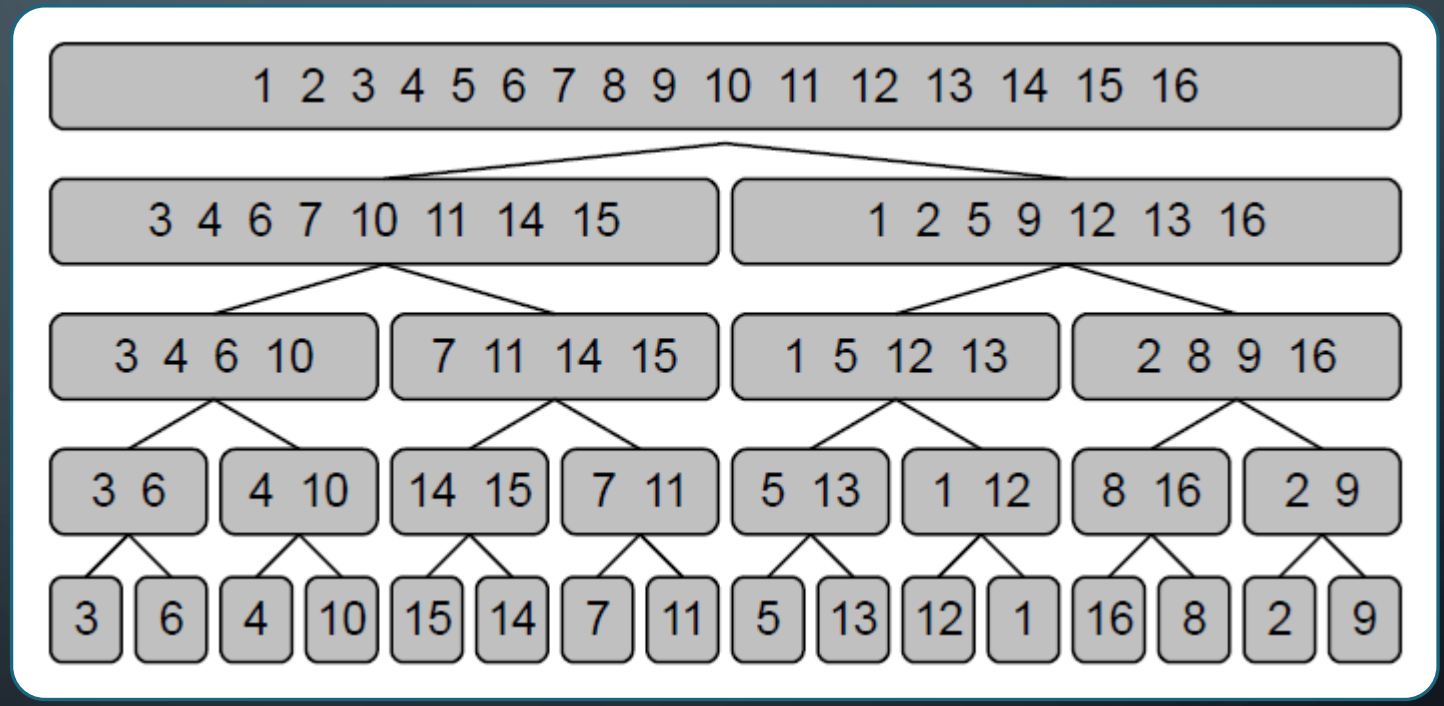
- Ordenação por intercalação (Merge Sort):
  - Dividir para conquistar;
  - Divisão da sequência em partes menores para facilitar a ordenação;
  - União de sequências menores já ordenadas, gerando sequências maiores ordenadas;
  - Inventado por John von Neumann em 1959 (EDVAC).

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


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

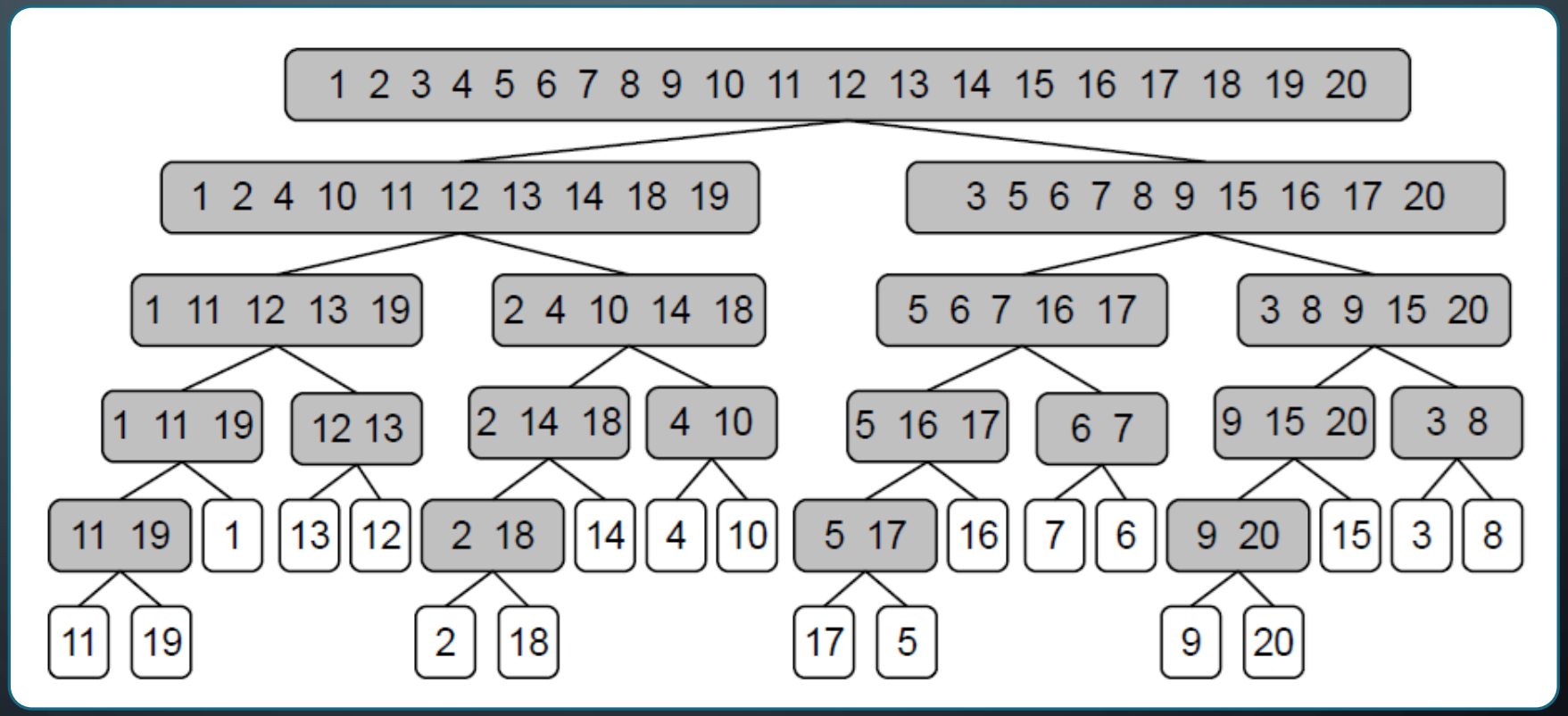

```
private static void merge( int[] array, int start, int middle, int end, int[] tempMS ) {
    int i = start;
    int j = middle + 1;
    for ( int k = start; k <= end; k++ ) {
        tempMS[k] = array[k];
    }
    for ( int k = start; k <= end; k++ ) {
        if ( i > middle ) {
            array[k]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

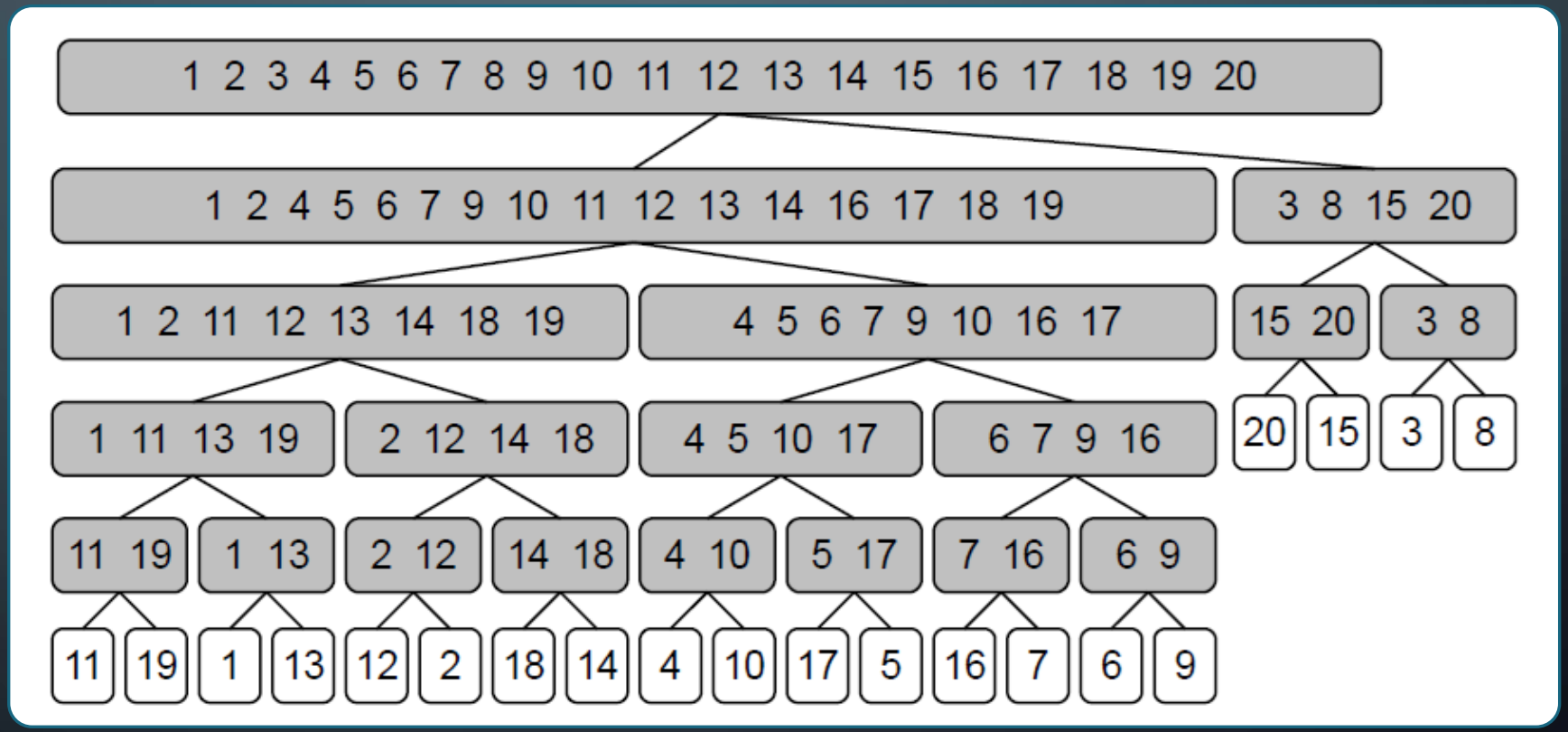

```
public static void sort( int[] array ) {
    int length = array.length;
    int[] tempMS = new int[length];
    bottomUp( array, 0, length - 1, tempMS );
}

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

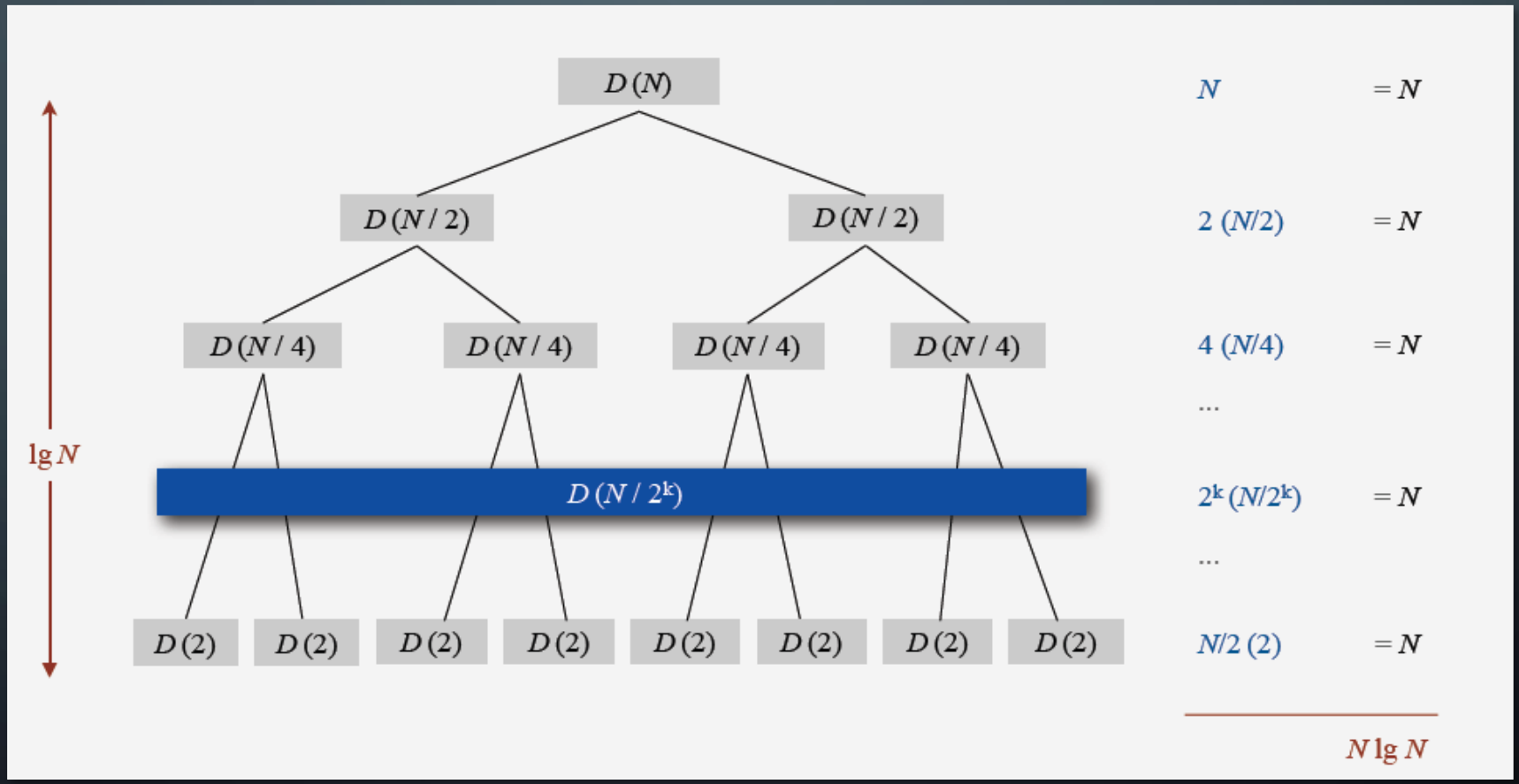


MERGE SORT
ESTABILIDADE

- 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

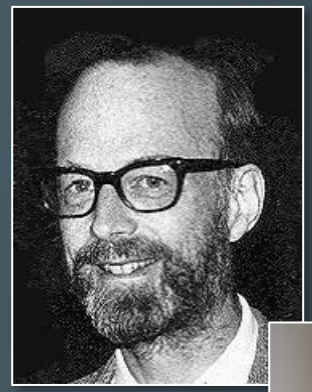
MERGE SORT
COMPLEXIDADE

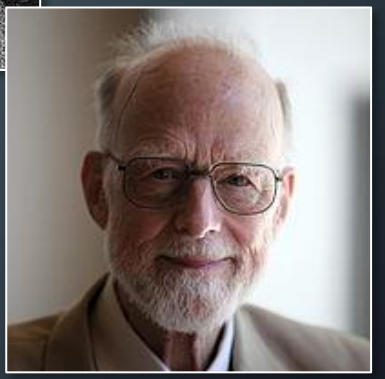


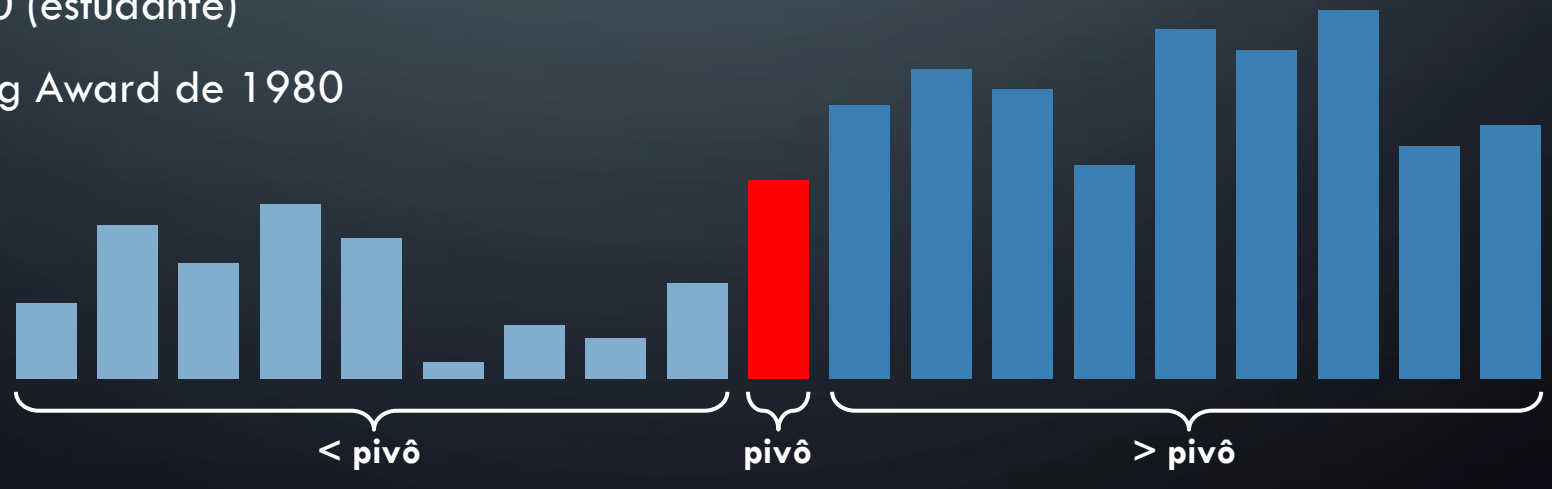
k: nível da árvore

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

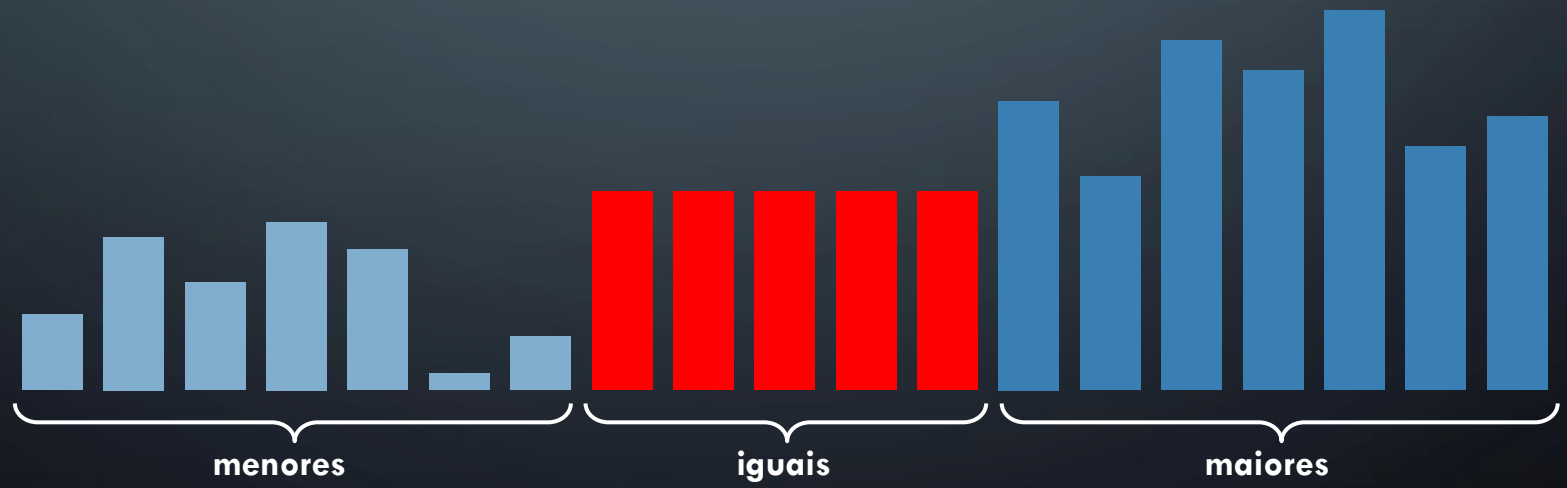
QUICK SORT

- 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

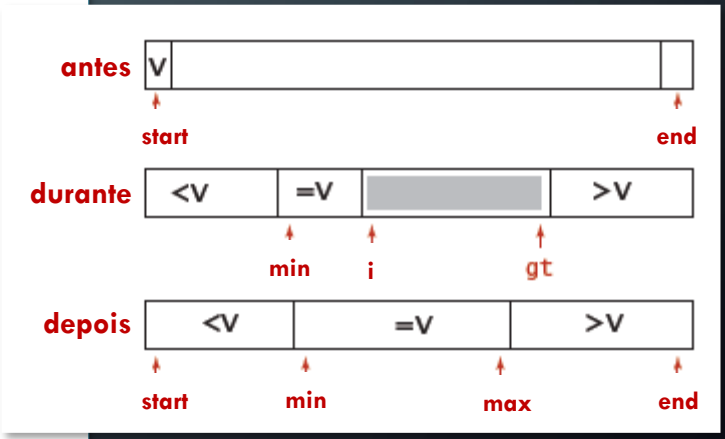
- Resolve o problema das chaves duplicadas, dividindo o array em três faixas:
  - menores – iguais – maiores
- Solução do "*Dutch National Flag Problem*" proposto por Edsger Dijkstra.

Edsger Dijkstra
Turing Award de 1972

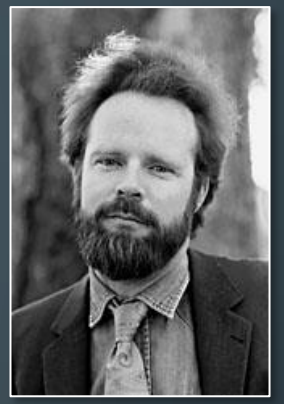


QUICK SORT 3-WAY


```
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 );
    }
}
```



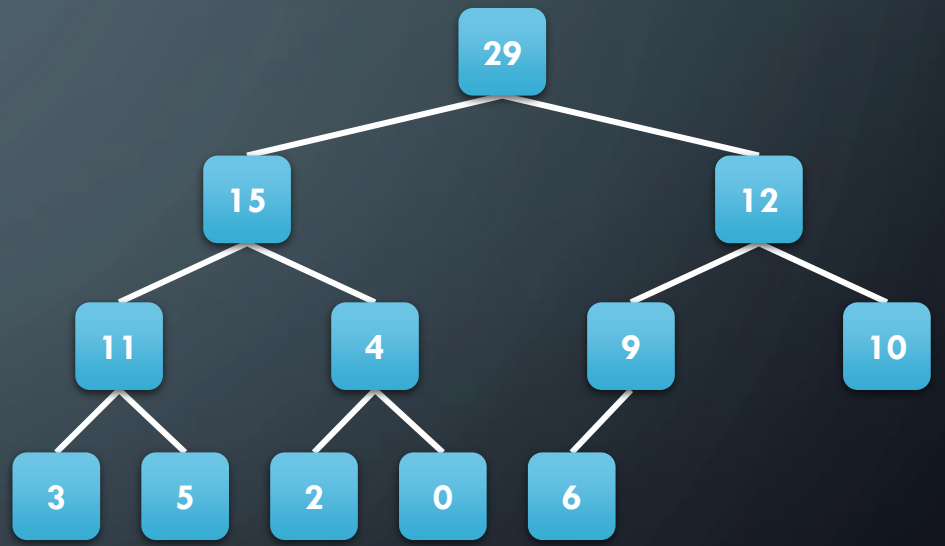
HEAP SORT



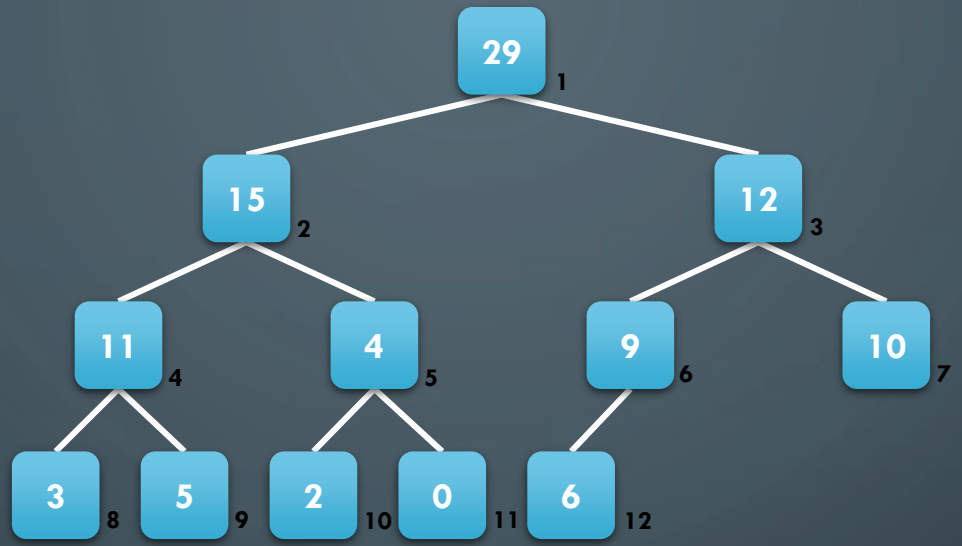
- Ordenação usando um Heap Binário (Heap Sort):
  - Critério de ordenação:
    - **Max-Heap:** Elemento pai sempre maior ou igual aos filhos;
    - **Min-Heap:** Elemento pai sempre menor ou igual aos filhos;
  - Chaves armazenadas nos nós;
  - Utilizaremos apenas:
    - Árvores binárias (até dois filhos);
    - 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.

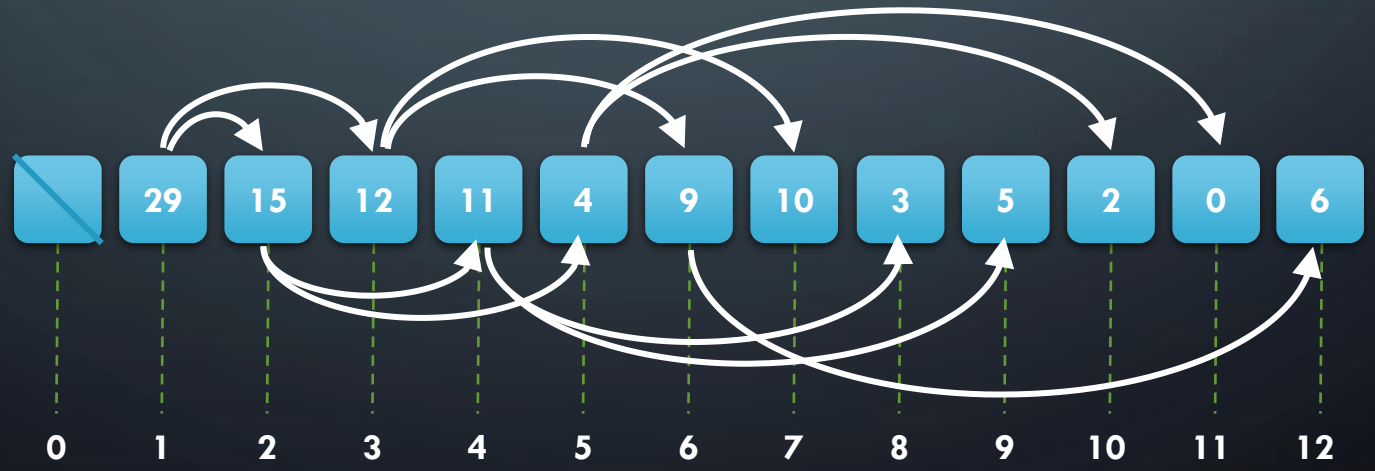
HEAP SORT

- 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

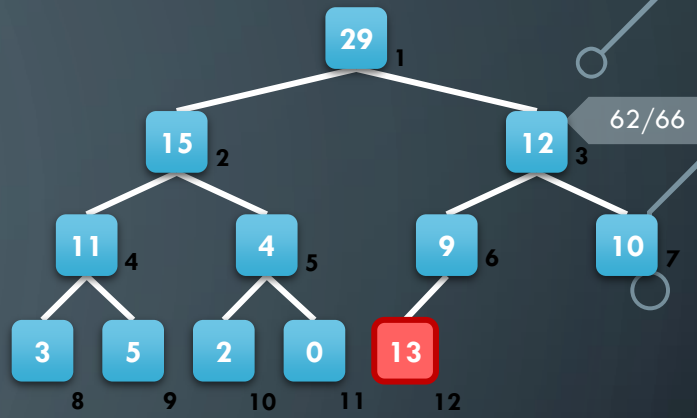
HEAP SORT

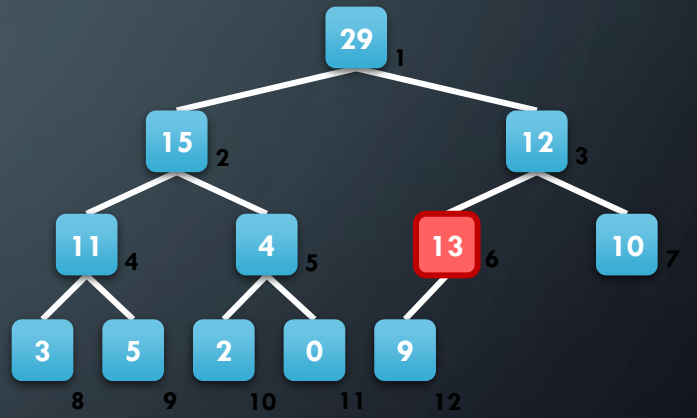
- Heap Binário Máximo (Max-Heap)
  - **Invariante:** chave do nó pai é sempre maior ou igual às chaves dos nós filhos;
- Elemento violando a invariante:
  - **Chave do filho maior que a chave do pai:**
    - O elemento precisa "subir" na árvore;
    - Bottom-up *reheapify* (*swim* ➔ flutuar);
  - **Chave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

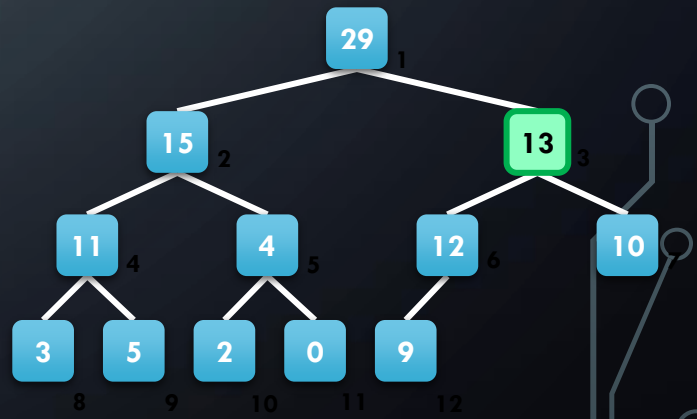
HEAP SORT
BOTTOM-UP REHEAPIFY


```
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

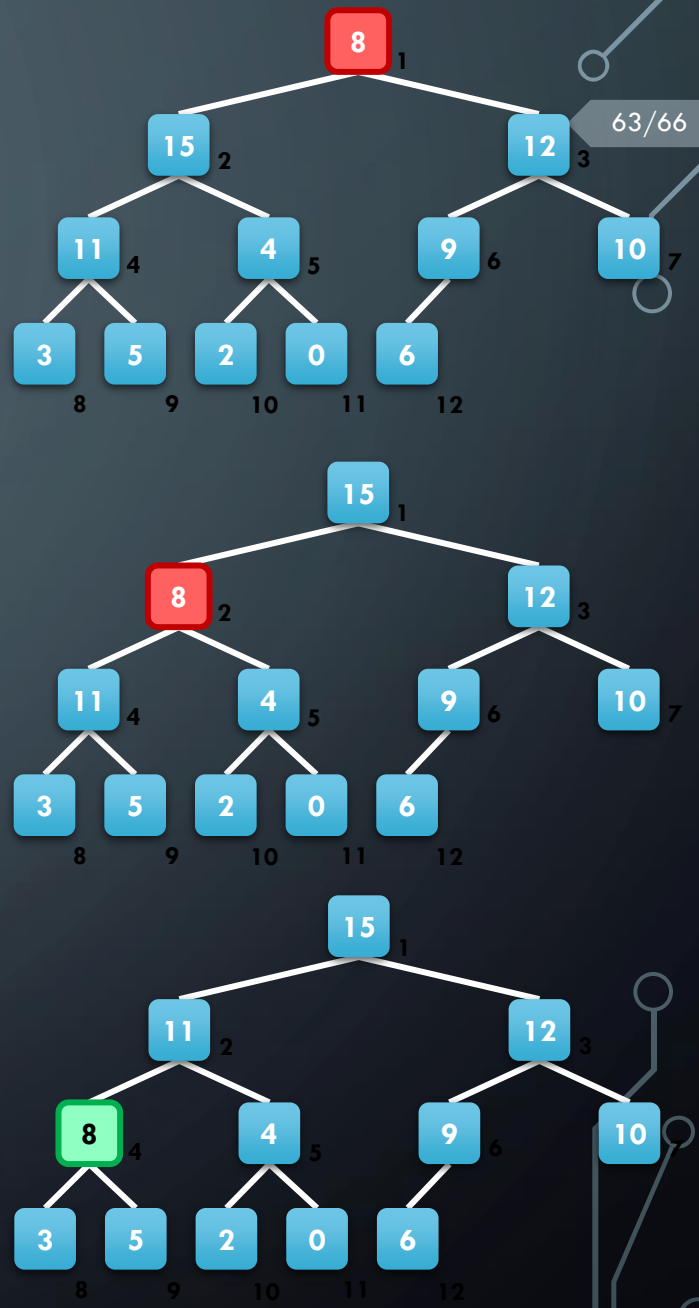
```
private static void sink( int[] array, int k, int n ) {
    while ( 2*k <= n ) {
        int j = 2*k;
        if ( j < n && array[j] < array[j+1] ) {
            j++;
        }
        if ( array[k] >= array[j] ) {
            break;
        }
        swap( array, k, j );
        k = j;
    }
}
```

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 );
    }
}
```