PUCRS | ESCOLA POLITÉCNICA

ALGORITMOS DE PESQUISA E ORDENAÇÃO

ALGORITMOS E ESTRUTURA DE DADOS I

Profa. Andréa Aparecida Konzen
Escola Politécnica - PUCRS

Baseado no material cedido pelo professor Edson Moreno

Introdução

- Há dois tipos de problemas comuns em computação
  - Pesquisa
    - Exemplo:
      - Buscar um elemento em uma coleção de dados
  - Ordenação
    - Objetivo
      - Garantir a ordem em uma coleção de dados

Algoritmos de pesquisa

Algoritmos de pesquisa

- Visam encontra um valor segundo algum critério de forma eficiente
  - Eficiência depende do cenário
- Caso 1 – coleção desordenada
  - Exemplo
    - **Procurar** pelo valor **-1** na lista **[8,2,1,5,2,6]**
  - Característica do problema
    - A lista **NÃO ESTÁ** ordenada
  - Solução
    - Comparação com cada elemento na coleção
  - Condição de parada
    - O valor exato precisa de ser encontrado
  - Consequência – Facilmente chega ao pior caso quando o valor não está na lista
    - Precisa percorrer toda lista para saber que o valor não se encontra nela

Algoritmos de pesquisa

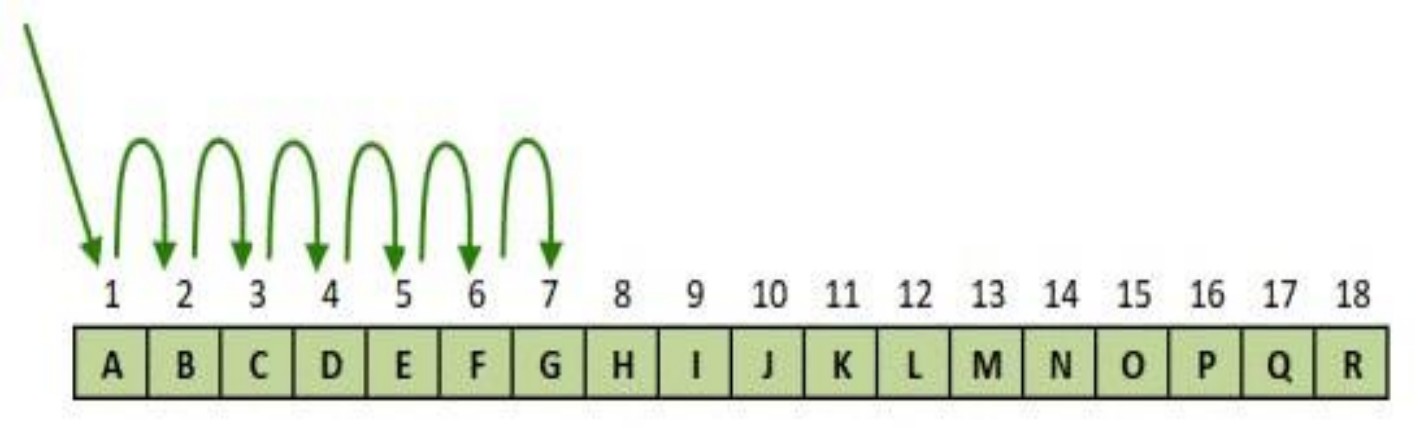
- Visam encontra um valor segundo algum critério de forma eficiente
  - Eficiência depende do cenário
- Caso 2 – coleção ordenada
  - Exemplo
    - Procurar pelo valor -1 na lista [1,2,2,5,6,8]
  - Característica do problema
    - A lista **ESTÁ** ordenada
  - Solução
    - Comparação com cada elemento na coleção
  - Condição de parada
    - Quando valor exato é encontrado ou o valor da lista é maior que o valor procurado
  - Consequência: Reduz a possibilidade de chegar ao pior caso

Algoritmos de pesquisa

- Tipos de algoritmo
  - **Sequencial**
    - Funciona tanto para coleções ordenadas ou desordenadas
    - Funcionamento: Realiza comparação com cada um dos elementos da coleção
    - Complexidade O(n)
  - **Binário**
    - Funciona somente para coleções ordenadas
    - Funcionamento: realiza saltos dentro da coleção em busca do valor procurado
    - Complexidade O(Log n)

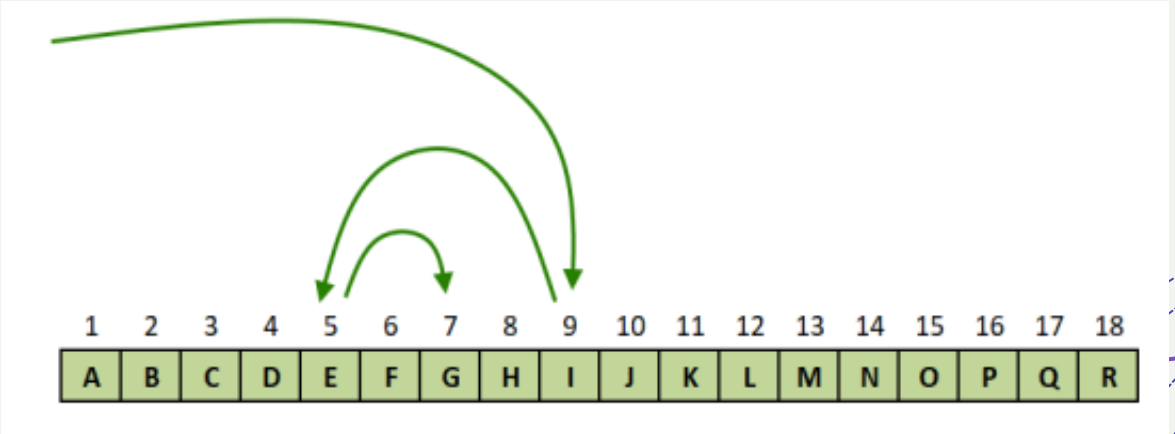
Algoritmos de pesquisa

- **Pesquisa sequencial**
  - Característica: o avanço da busca reduz a área de forma aritmética
  - Parâmetros de entrada:
    - Valor a ser buscado
    - Vetor de dados
    - Tamanho do vetor (depende da linguagem)
  - Saída
    - Confirmação da presença do valor (V/F)



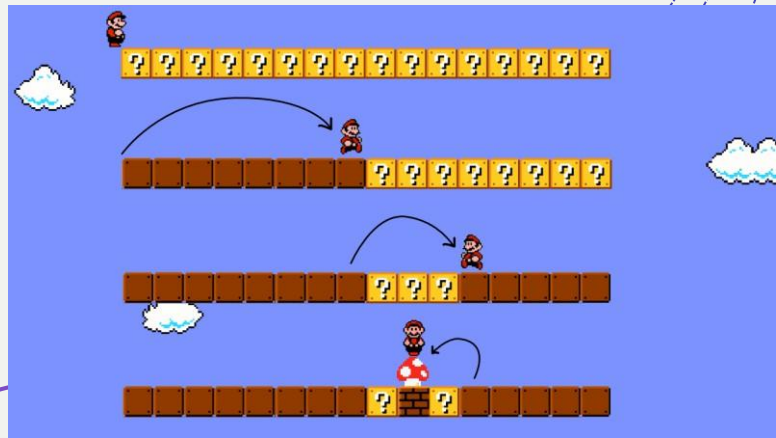
Algoritmos de pesquisa

- **Pesquisa binária**
  - Característica: o avanço da busca reduz a área de forma geométrica
  - Parâmetros de entrada:
    - Valor a ser buscado
    - Vetor de dados
    - Tamanho do vetor (depende da linguagem)
  - Saída
    - Confirmação da presença do valor (V/F)



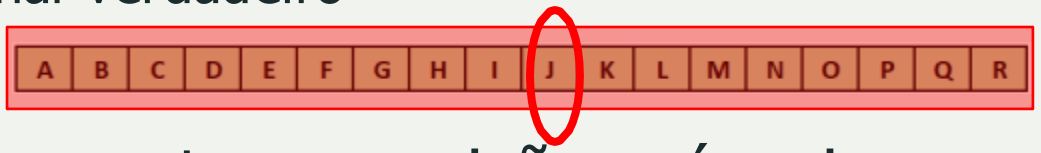
Algoritmos de pesquisa

- **Pesquisa binária – Detalhamento do algoritmo**
  - Algoritmo mantém dois valores: Low e High
    - Definem o intervalo corrente considerado na pesquisa
    - Inicialmente low=0 e high=n-1
  - Comparação é feita com o candidato "do meio" do intervalo
    - m= (low+high)/2
  - Como consequência
    - Valor no vetor é igual ao de busca
    - Valor no vetor na posição m é maior que o de busca
    - Valor no vetor na posição m é menor que o de busca

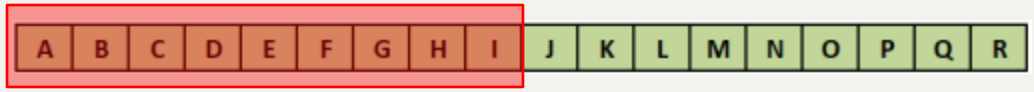


Algoritmos de pesquisa

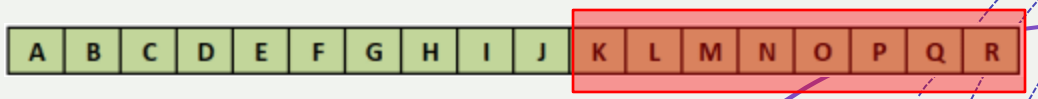
- Pesquisa binária – detalhamento do algoritmo
  - Caso 1 – Se o valor do vetor for igual ao de busca então
    - Ótimo, basta retornar verdadeiro



  - Caso 2 – Se o valor no vetor na posição m é maior que o de busca
    - Redefine o intervalo de busca ajustando Low para m+1 e mantem High

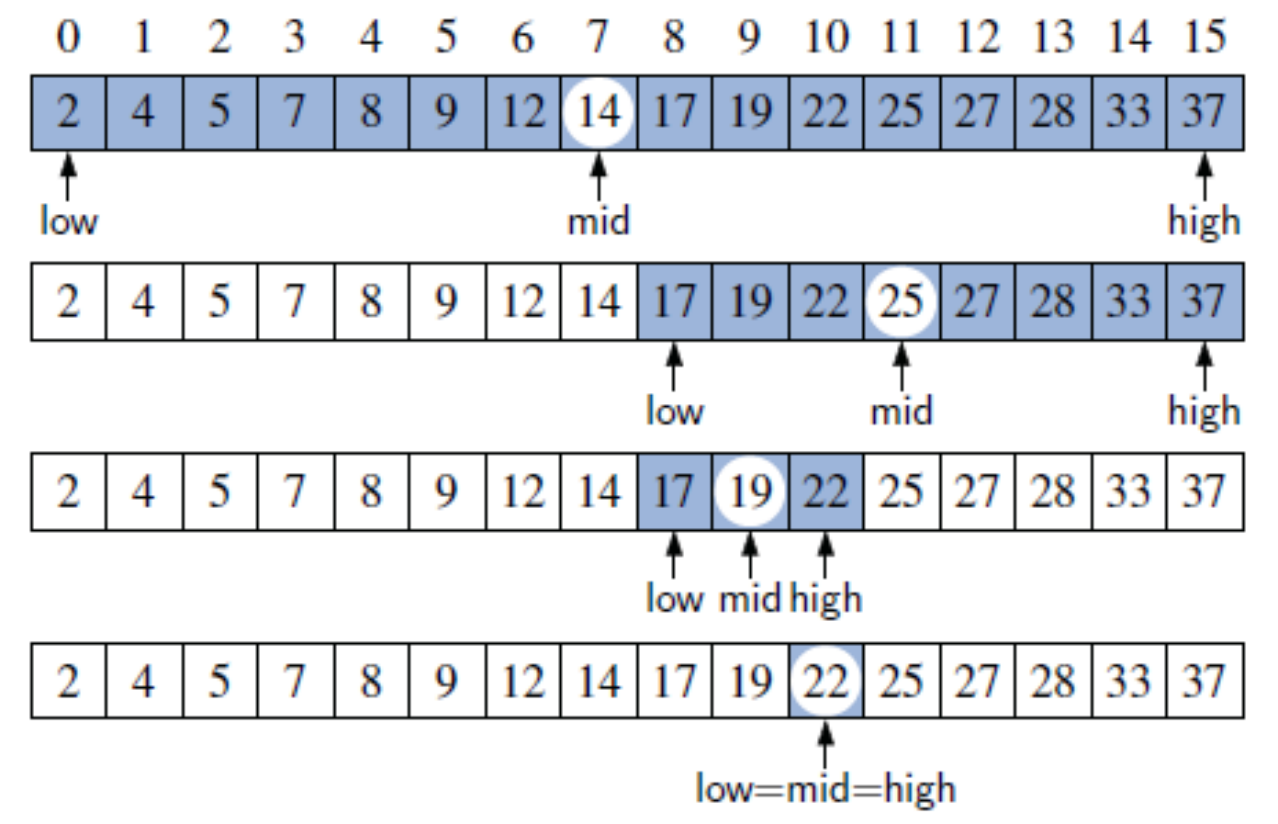


  - Caso 3 – Se o valor no vetor na posição m é menor que o de busca
    - Redefine o intervalo de busca ajustando High para m-1 e mantem Low



Algoritmos de pesquisa

- Pesquisa binária – Exemplo de execução



- Valor buscado:
  - 22
- Indices considerados
  - High
  - Low
  - M
- Fórmula de m
  - M = (low + high)/2

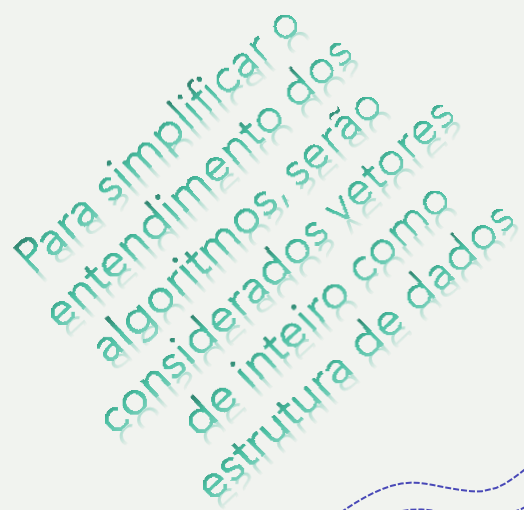
Algoritmos de pesquisa

- Pesquisa binária
  - Condição de parada
    - Quando o valor procurado é encontrado OU
    - O valor buscado não foi encontrado **E** low é igual a high
  - Pré requisito para o funcionamento deste algoritmo
    - A coleção precisa estar ordenada!!!!

Algoritmos de ordenação

Algoritmos de ordenação

- Visam garantir uma ordem de forma eficiente
  - Realiza troca de valores entre posições da coleção segundo um critério
  - Algoritmos variam em
    - complexidade computacional
    - Recurso de memória necessário
    - Simplicidade de elaboração
  - Exemplos a serem explorados
    - Bubble sort
    - Insertion sort
    - Merge sort
    - Quick sort



https://www.hackerearth.com/practice/algorithms/sorting

Bubble sort

Bubble sort

O Bubble Sort é um algoritmo de ordenação simples, mas ineficiente para grandes listas.

Ordena uma lista comparando elementos adjacentes e trocando-os se estiverem na ordem errada.

O processo é repetido até que a lista esteja completamente ordenada.

Bubble sort

- Características
  - Método simples de ordenação
    - Complexidade $O(N^2)$
  - Parâmetros de entrada
    - Vetor
    - Tamanho do vetor (dependendo da linguagem)
  - Saída
    - Vetor ordenado

Bubble sort

- Funcionamento do algoritmo
  - Compara pares de elementos adjacentes de forma progressiva
  - Os elementos são trocados de posição quando
    - Critério de ordenação falha
  - Avança para o próximo par
    - Próximo par é formado por um elemento do par anterior e seu adjacente
- Condição de parada
  - Repete o algoritmo até que todos estejam ordenados

Bubble sort

+ Código exemplo (em C)

```c
void bubbleSort (int vetor[], int n)
  int k, j, aux;
  for (k = 1; k < n; k++)
    for (j = 0; j < n - 1; j++)
      if (vetor[j] > vetor[j + 1])
        aux = vetor[j];
        vetor[j] = vetor[j + 1];
        vetor[j + 1] = aux;
```

Bubble sort

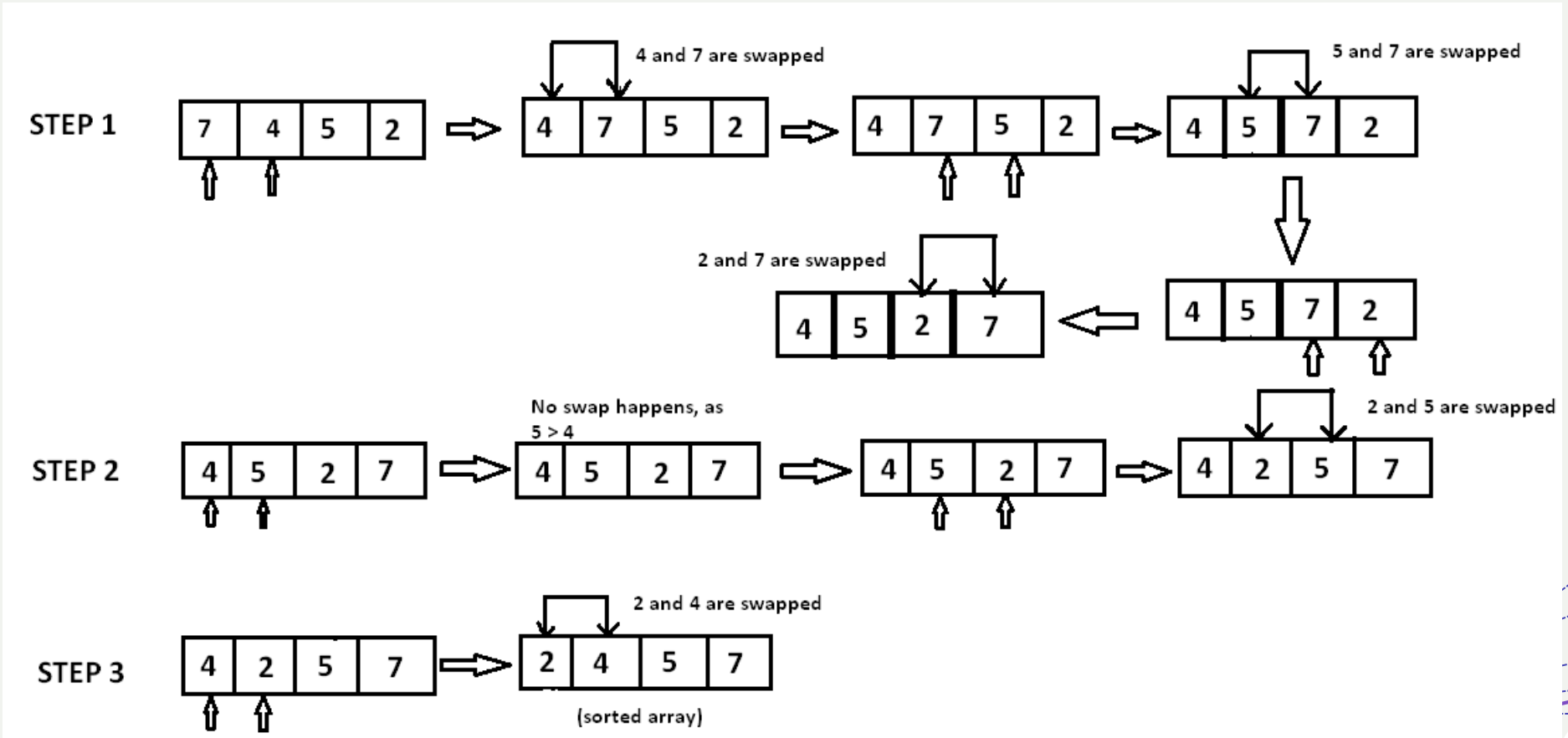
- Código exemplo (em Java)

```
public static void bubbleSort(int[] vetor) {
    int n = vetor.length;
    int aux;

    for (int k = 1; k < n; k++) {
        for (int j = 0; j < n - 1; j++) {
            if (vetor[j] > vetor[j + 1]) {
                aux = vetor[j];
                vetor[j] = vetor[j + 1];
                vetor[j + 1] = aux;
            }
        }
    }
```

Bubble sort

```
void bubbleSort (int vetor[], int n)
  int k, j, aux;
  for (k = 1; k < n; k++)
    for (j = 0; j < n - 1; j++)
      if (vetor[j] > vetor[j + 1])
        aux = vetor[j];
        vetor[j] = vetor[j + 1];
        vetor[j + 1] = aux;
```

+ Exemplo de ordenação



https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial/

Bubble sort

+ Exemplo de ordenação

https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/visualize/

Insertion sort

Insertion sort

O Insertion Sort é um algoritmo de ordenação simples e intuitivo, que constrói a lista ordenada um item por vez.

Ele é eficiente para listas pequenas ou que já estão parcialmente ordenadas.

Insertion sort

- Características
  - Método simples de ordenação
    - Complexidade $O(N^2)$
  - Parâmetros de entrada
    - Vetor
    - Tamanho do vetor (dependendo da linguagem)
  - Saída
    - Vetor ordenado

Insertion sort

- Funcionamento do algoritmo
  - Considera um elemento de cada vez
    - Colocando-o na ordem correta em relação aos demais elementos
  - Inicia considerando dois elementos A e B adjacentes
    - Se a atende o critério de ordenação com relação a B então
      - avança para B e C realizando a mesma comparação até que chegue no final na coleção
  - Em caso de necessidade de ordenação
    - Considerando dois elementos F e G
      - Troca F e G de posição e inicia comparação regressiva na coleção até encontrar posição adequada para o elemento G

Insertion sort

+ Codigo exemplo (em C)

```c
void insertionSort ( int A[ ] , int n)
  for( int i = 0; i < n; i++ ) {
    int temp = A[ i ];
    int j = i;
    while( j > 0 && temp < A[j-1])
      A[ j ] = A[j-1];
      j= j - 1;
      A[ j ] = temp;
```

Insertion sort

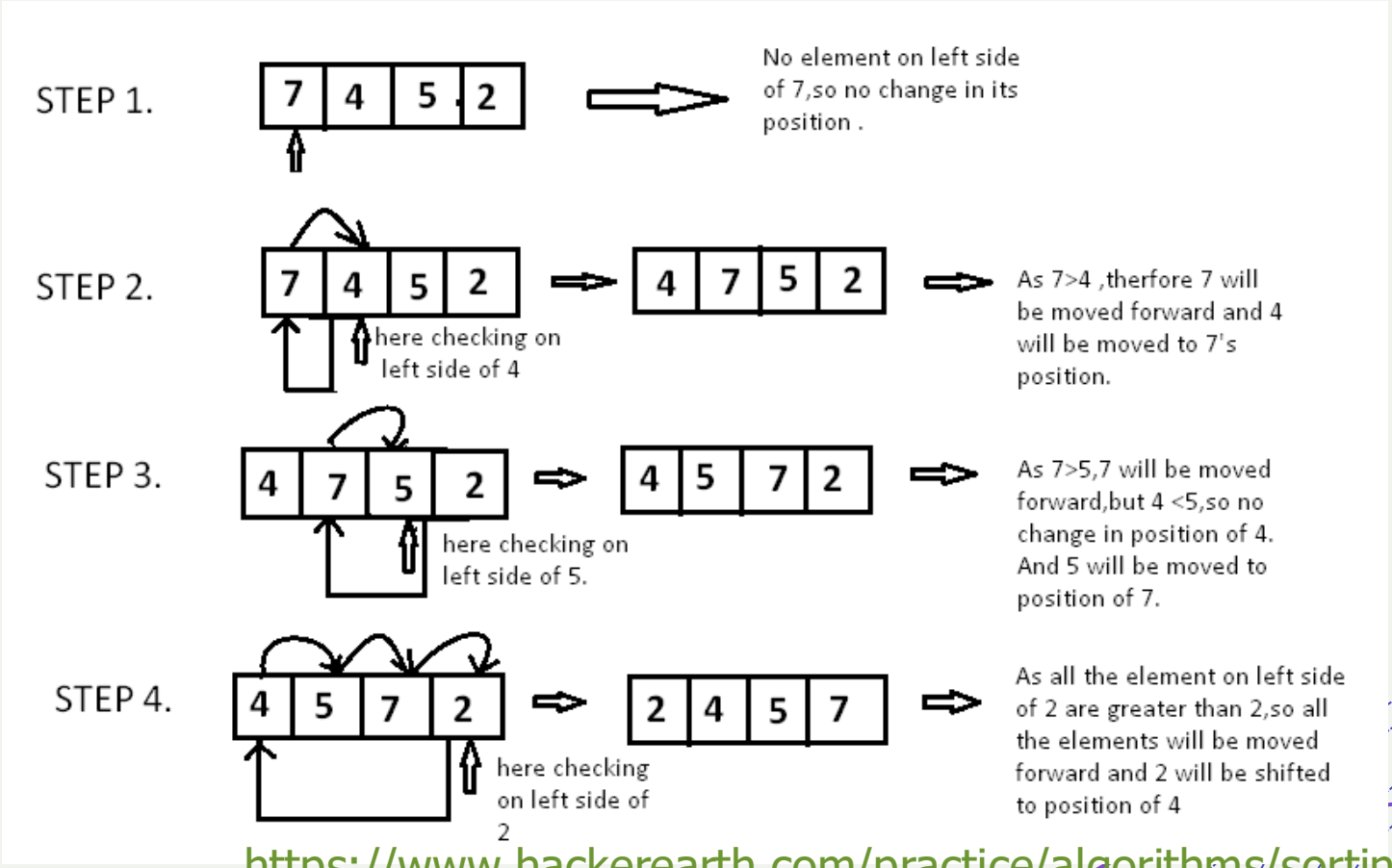
+ Codigo exemplo (em Java)

```
public static void insertionSort(int[] array) {
    int n = array.length;

    for (int i = 1; i < n; i++) {
        int key = array[i];
        int j = i - 1;

 while (j >= 0 && array[j] > key) {
            array[j + 1] = array[j];
            j = j - 1;
        }
        array[j + 1] = key;
    }
}
```

Insertion sort

```
void insertionSort ( int A[ ] , int n)
  for( int i = 0 ;i < n ; i++ ) {
    int temp = A[ i ];
    int j = i;
    while( j > 0 && temp < A[ j -1])
      A[ j ] = A[ j-1];
      j= j - 1;
      A[ j ] = temp;
```

+Execução – exemplo 1

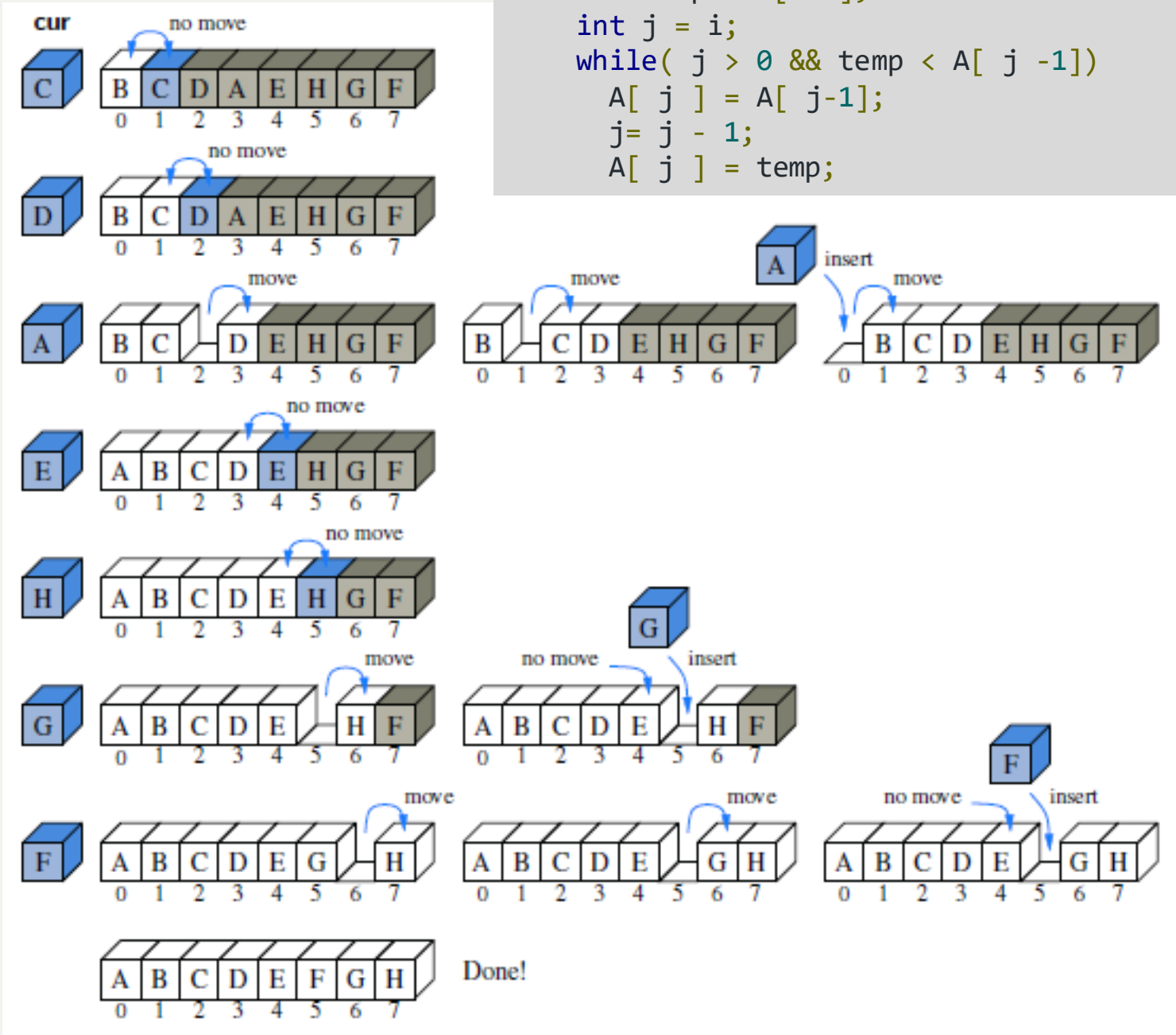


https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/tutorial/

Insertion sort

+Execução – exemplo 2

```
void insertionSort ( int A[ ] , int n)
  for( int i = 0 ;i < n ; i++ ) {
    int temp = A[ i ];
    int j = i;
    while( j > 0 && temp < A[ j -1])
      A[ j ] = A[ j-1];
      j= j - 1;
      A[ j ] = temp;
```



Insertion sort

+ Exemplo de ordenação

https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/visualize/

Merge sort

Merge sort

O Merge Sort é um algoritmo de ordenação eficiente e estável, baseado no paradigma de "dividir para conquistar".

É especialmente útil para ordenar grandes listas, com uma complexidade de tempo de O(n logn) em todos os casos (melhor, pior e médio).

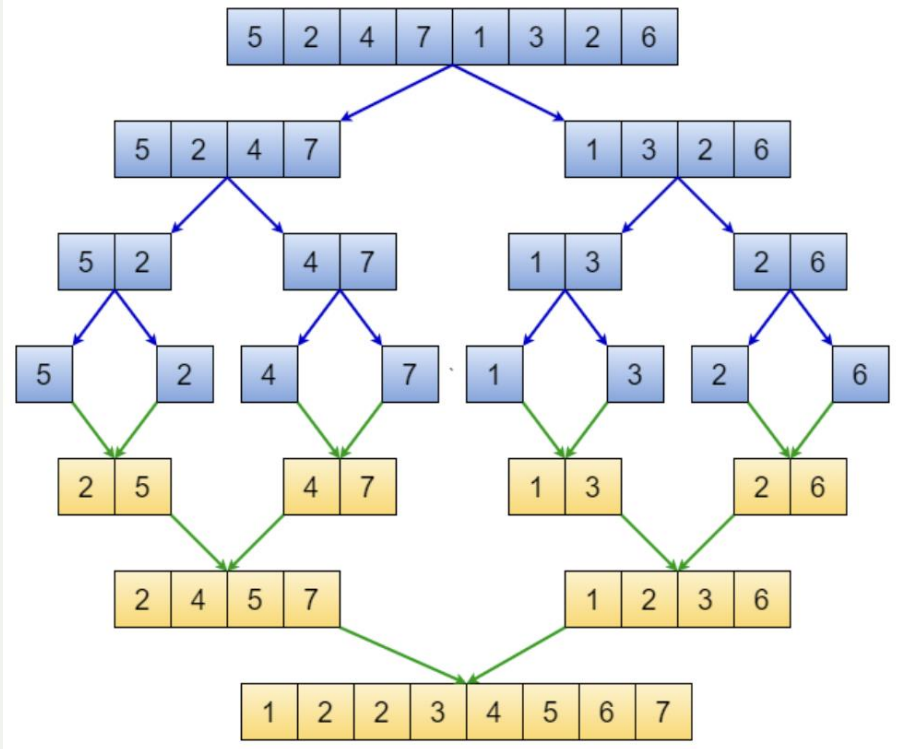
Merge sort

- Características
  - Algoritmo mais complexo de ser elaborado
    - Explora Recursão
    - Complexidade O(NlogN)
  - Consiste em três etapas
    - Dividir: se os dados de entrada são menores que um limiar, classifica e retorna a solução, senão divide os dados de entrada em dois ou mais subconjuntos
    - Conquistar: classifica os subconjuntos recursivamente
    - Combinar: combina as soluções dos subconjuntos em uma única solução

Merge sort

Para ordenar uma sequência S com n elementos:

- **Dividir:**
- Se S tem zero ou um elemento, retorna S, pois já está classificado;
- Senão, remove os elementos de S e coloca-os em duas sequências, S1 e S2 (n/2 elementos em cada um)
- **Conquistar:**
- Classifica as sequências S1 e S2 recursivamente
- **Combinar:**
- Coloca os elementos de volta em S com a união das sequências S1 e S2 ordenadas



Merge sort

+Codigo exemplo (em C)

```
void mergeSort (int A[ ] , int start , int end )
  if( start < end )
    int mid = (start + end ) / 2 ;
    merge_sort (A, start , mid ) ;
    merge_sort (A,mid+1 , end ) ;
    merge(A,start , mid , end );
```

Merge sort

+Codigo exemplo (em C)

```c
void merge(int A[ ] , int start, int mid, int end)
  int p = start ,q = mid+1;
  int Arr[end-start+1] , k=0;
  for(int i = start ;i <= end ;i++)
    if(p > mid)                    Arr[ k++ ] = A[ q++];
    else if ( q > end)             Arr[ k++ ] = A[ p++ ];
    else if( A[ p ] < A[ q ]) Arr[ k++ ] = A[ p++ ];
    else                           Arr[ k++ ] = A[ q++];
  for (int p=0 ; p< k ;p ++)
    A[ start++ ] = Arr[ p ] ;
```

Merge sort

+Codigo exemplo (em Java)

```
    // Método principal que inicia o processo de ordenação
    public static void mergeSort(int[] array, int left, int right)
{
        if (left < right) {
            // Encontra o meio do array
            int middle = (left + right) / 2;

            // Ordena a primeira e a segunda metade
            mergeSort(array, left, middle);
            mergeSort(array, middle + 1, right);

            // Combina as metades ordenadas
            merge(array, left, middle, right);
        }
    }
```

Merge sort

+ Codigo exemplo (em Java)

```
// Método para combinar duas metades do array
private static void merge(int[] array, int left, int middle, int right) {
    // Tamanhos dos subarrays temporários
    int n1 = middle - left + 1;
    int n2 = right - middle;

    // Arrays temporários
    int[] leftArray = new int[n1];
    int[] rightArray = new int[n2];

    // Copia os dados para os arrays temporários
    for (int i = 0; i < n1; ++i)
        leftArray[i] = array[left + i];
    for (int j = 0; j < n2; ++j)
        rightArray[j] = array[middle + 1 + j];

    // Índices iniciais dos subarrays e do array combinado
    int i = 0, j = 0;
    int k = left;

    // Combina os subarrays temporários de volta ao array principal
    while (i < n1 && j < n2) {
        if (leftArray[i] <= rightArray[j]) {
            array[k] = leftArray[i];
            i++;
        } else {
            array[k] = rightArray[j];
            j++;
        }
        k++;
    }

    // Copia os elementos restantes de leftArray, se houver
    while (i < n1) {
        array[k] = leftArray[i];
        i++;
        k++;
    }

    // Copia os elementos restantes de rightArray, se houver
    while (j < n2) {
        array[k] = rightArray[j];
        j++;
        k++;
    }
}
```

Merge sort

+Execução do algoritmo pode ser vista como uma árvore binária (merge-sort tree):

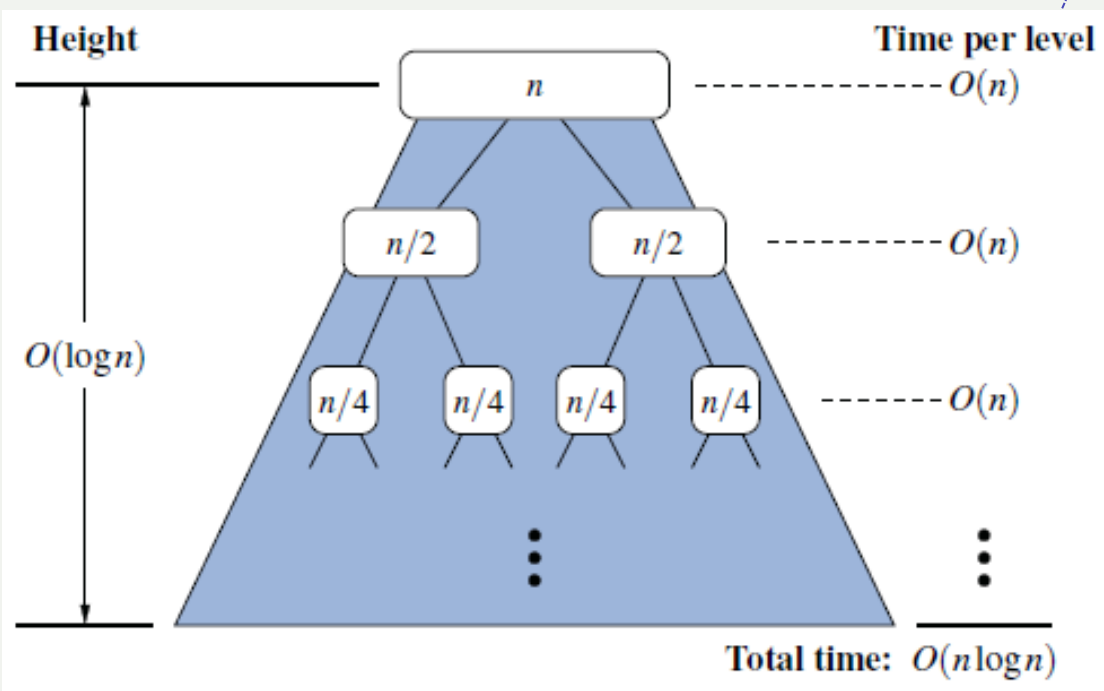
Cada nodo representa uma chamada recursiva do algoritmo de merge-sort

a) **Sequências de entrada processadas em cada nodo**

b) **Sequências de saída geradas em cada nodo**

Merge sort

- Análise do tempo de execução do algoritmo
  - Altura da árvore
    - log n
  - Tempo gasto em cada nível da árvore
    - O(n)
  - Tempo de execução do merge-sort
    - O(n log n)



Merge sort

+ Execução

https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/

Quick sort

Quick sort

O Quick Sort é um dos algoritmos de ordenação mais eficientes, baseado no paradigma de "dividir e conquistar".

É amplamente utilizado devido à sua eficiência e simplicidade.

Embora tenha um pior caso de $O(n^2)$, na prática, com boas escolhas de pivô, ele geralmente tem um desempenho próximo de O(nlogn).

Quick sort

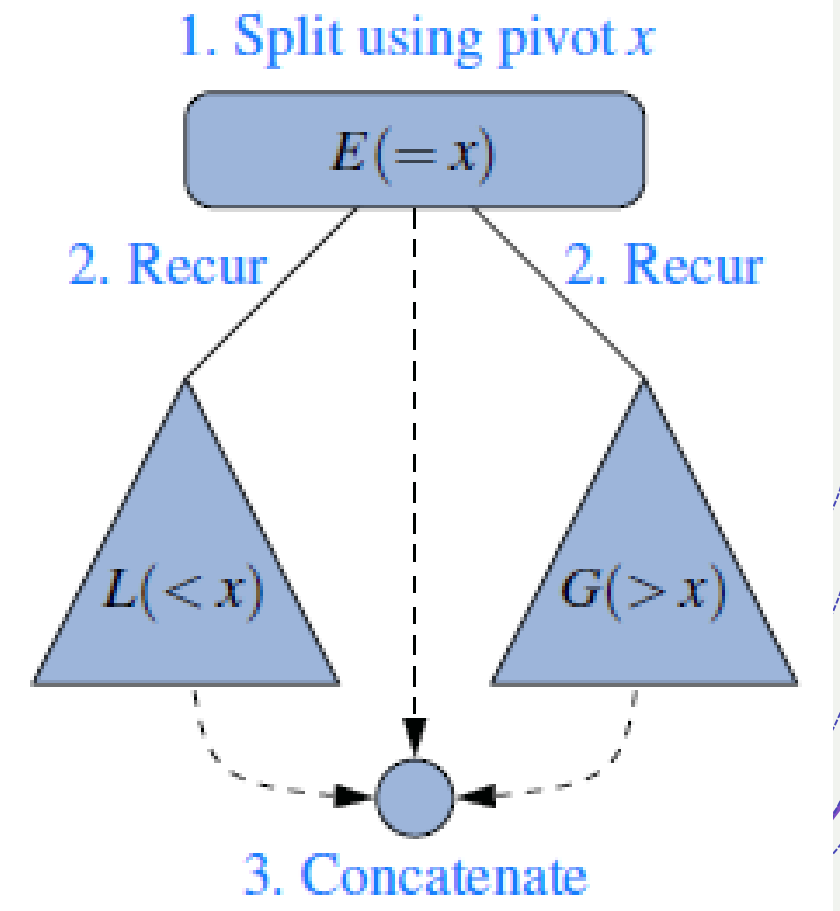
- Características
  - Algoritmo mais complexo de ser elaborado
    - Explora Recursão
    - Complexidade O(NlogN)
  - DIVISÃO E CONQUISTA COMO MERGE SORT
    - Maior processamento é feito antes das chamadas recursivas
    - Ideia principal:
      - Divisão de S em subconjuntos (sequências)
      - Recursão para classificar cada subconjunto
      - Combinar as subsequências ordenadas através de uma concatenação simples

Quick sort

- Funcionamento
  - Dividir:
    - Se S tem pelo menos dois elementos, seleciona um deles para ser o pivô
    - O pivô pode ser qualquer elemento de S
    - Remove todos os elementos de S e coloca-os em três sequências:
      - L: armazena os elementos de S menores que o pivô
      - E: armazena os elementos de S iguais ao pivô
      - G: armazena os elementos de S maiores que o pivô

Quick sort

- Funcionamento
  - Conquistar:
    - Recursivamente ordena as sequências L e G
  - Combinar:
    - Coloca de volta os elementos em S, inserindo primeiro os elementos de L, depois de E e finalmente de G



Quick sort

+Codigo exemplo (em C)

```
void quickSort ( int A[ ] ,int start , int end )
  if( start < end )
    int piv_pos = partition (A,start , end );
    quick_sort (A,start , piv_pos -1);
    quick_sort ( A,piv_pos +1 , end);
```

Quick sort

+ Código exemplo (em C)

```c
int partition ( int A[],int start ,int end)
  int i = start + 1;
  int piv = A[start];

  for(int j =start + 1; j <= end ; j++)
    if ( A[ j ] < piv)
      swap (A[ i ],A [ j ]);
      i += 1;
  swap ( A[ start ] ,A[ i-1 ] ) ;
  return i-1;
```

Quick sort

+ Código exemplo (em Java)

```
// Método principal que executa o Quick Sort
public static void quickSort(int[] array, int low, int high) {
    if (low < high) {
        // Encontra o índice de partição
        int pivotIndex = partition(array, low, high);

        // Ordena recursivamente as duas subpartes
        quickSort(array, low, pivotIndex - 1);
        quickSort(array, pivotIndex + 1, high);
    }
}
```

Quick sort

+ Código exemplo (em Java)

```
    // Método que realiza o particionamento
    private static int partition(int[] array, int low, int high) {
        // Escolhe o pivô (aqui usamos o último elemento como pivô)
        int pivot = array[high];
        int i = (low - 1); // Índice do menor elemento

        for (int j = low; j < high; j++) {
            // Se o elemento atual é menor ou igual ao pivô
            if (array[j] <= pivot) {
                i++;

                // Troca array[i] e array[j]
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
...
```

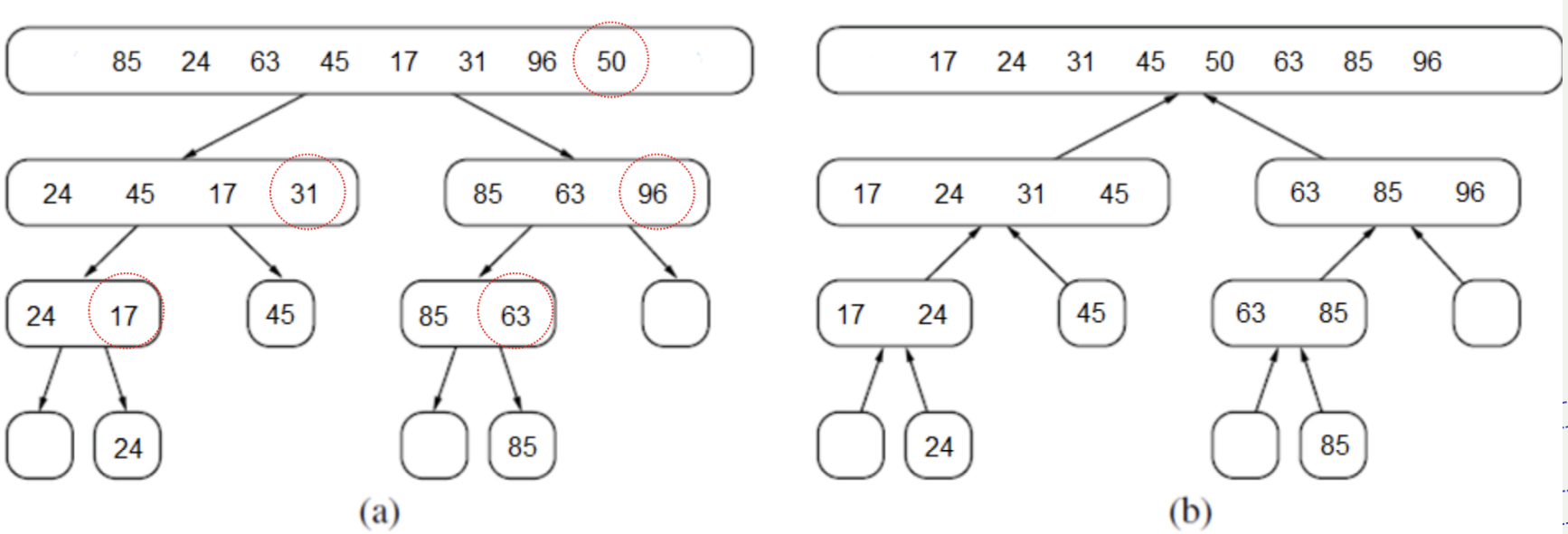
```
...
 // Troca array[i+1] com o pivô (array[high])
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;

        return i + 1;
    }
```

Quick sort

Execução do algoritmo pode ser vista como uma árvore binária (quick-sort tree):

a) **Sequência de entrada processada em cada nodo da árvore**

b) **Sequência de saída gerada em cada nodo da árvore**



Quick sort

+ Execução

https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/visualize/