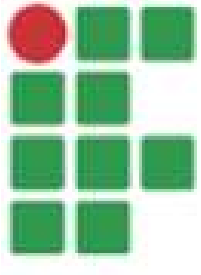


INSTITUTO FEDERAL
Paraíba
Campus João Pessoa

Aula
11

Programação e Estrutura de Dados

Atualizada em 15/12/2022

Ordenação de Dados

Métodos de Organização Interna

Professor:

Dr. Alex Sandro da Cunha Rêgo

alex@ifpb.edu.br

Ordenação de Dados

- **Motivação**
  - Os dados de determinadas aplicações são armazenados de acordo com algum critério
  - **Algoritmos de ordenação eficientes** >> melhor desempenho computacional na ordenação
- Alternativas
  - Garantir a ordem correta ao inserir novos elementos
  - Aplicação de algoritmos para ordenação dos elementos

"Ordenar dados significa colocar os elementos de uma sequência em uma determinada ordem"

Ordenação de Dados

- **Quando a ordenação facilita...**
  - Encontrar facilmente um nome na lista de candidatos homologados em um concurso
  - Catalogação de livros em uma biblioteca
  - Produtos vendidos em um supermercado por categoria
- Outras situações em que a ordenação simplifica as tarefas computacionais
  - Busca por um dado específico
  - Encontrar o maior/menor elemento
  - Teste de unicidade (distinção dos elementos)

Buble Sort

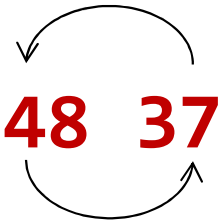
- **Princípio de Funcionamento**
  - Quando dois elementos estão fora de ordem, é feita a inversão e estes são trocados de posição
  - Primeiro elemento é comparado com o segundo, o segundo com o terceiro, o terceiro com o quarto, ...
    - ✓ Inversões são executadas quando necessárias
  - **Fim da comparação**: quando o penúltimo é comparado com o último
    - ✓ Ao final da varredura, o maior elemento ficará posicionado na **última posição**
  - O processo continua até n-i, até que **todo o vetor esteja ordenado** (i = 1, 2, 3, ...)

Buble Sort

- Aplicação prática do algoritmo

[25 48 37 12 57 86 33 92]

25 48 37 12 57 86 33 92 : ---



25 48 37 12 57 86 33 92 : (48 x 37) troca

25 37 48 12 57 86 33 92 : (48 x 12) troca

25 37 12 48 57 86 33 92 : ---

25 37 12 48 57 86 33 92 : ---

25 37 12 48 57 86 33 92 : (86 x 33) troca

25 37 12 48 57 33 86 92 : ---

Final da primeira varredura: o maior elemento estará no **final do array** (92)

Buble Sort

- **Comportamento**

Vantagem	Desvantagem
Simplicidade de **codificação** e **entendimento** do algoritmo	Lentidão decorrente do elevado número de trocas
Permite encerrar o algoritmo quando uma iteração não realiza troca	O desempenho é ainda mais afetado quando os itens de dados são objetos com várias propriedades

- **Indicação de uso**
  - Pequena coleção de dados
  - Coleção “quase ordenada”

Buble Sort

- **Codificação**

```
def bolha(array):
   for i in range(len(array)-1,0,-1):
      for j in range(0,i):
         if (array[j] > array[j+1] ):
            array[j],array[j+1] = array[j+1],array[j]
            # Efetua a troca

# main
v = [5, 9, 7, 21, 18, 1, 4]
print(v)
bolha( v )
print(v)
```

  - Solução recursiva

Selection Sort

- **Princípio de Funcionamento**
  - Selecione o primeiro elemento (i=0) do array
  - A partir de i+1, faça a varredura do array e identifique o valor que é menor ao que está armazenado no primeiro elemento
  - Troque o valor armazenado no índice do primeiro elemento por aquele determinado como o menor valor
  - Ao final da varredura, o menor elemento estará posicionado na **primeira posição**
  - Repetir o procedimento para os **n-1** elementos restantes (i = 1, 2, 3, ..., n-1)

Selection Sort

- Aplicação prática do algoritmo

[25 48 37 12 57 86 33 92]

25 48 37 12 57 86 33 92 : 25 x 12 - troca

12 48 37 25 57 86 33 92 : 48 x 25 - troca

12 25 37 48 57 86 33 92 : 37 x 33 - troca

12 25 33 48 57 86 37 92 : 48 x 37 - troca

12 25 33 37 57 86 48 92 : 57 x 48 - troca

12 25 33 37 48 86 57 92 : 86 x 57 - troca

12 25 33 37 48 57 86 92 : ---

12 25 33 37 48 57 86 92

Selection Sort

- **Comportamento**

Vantagem	Desvantagem
Simplicidade de **codificação** e **entendimento** do algoritmo	Se a coleção estiver ordenada antes do término do algoritmo, o algoritmo continua até o final
Só realiza uma troca por iteração	Quando o elemento a ser posicionado já está na posição de ordem, mesmo assim é efetuada a troca

- **Indicação de uso**
  - Coleções com itens de dados compostos e de grande quantidade de bytes (pouca movimentação)

Selection Sort

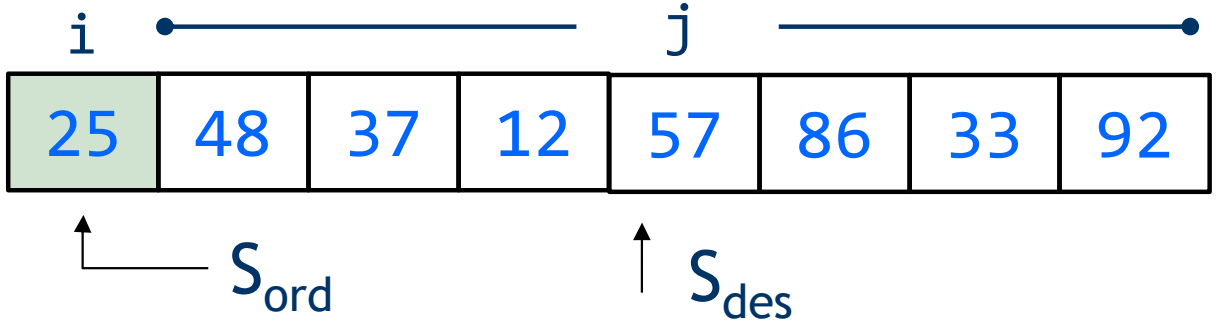
- Codificação

```
def selectionSort(array):
    for i in range(len(array)-1):
        min = i
        for j in range(i+1, len(array)):
            if(array[j] < array[min]):
                min = j
        array[min], array[i] = array[i], array[min]
        # troca
# main
v = [5, 9, 7, 21, 18, 1, 4]
print(v)
selectionSort( v )
print(v)
```

  - Solução recursiva

Insertion Sort

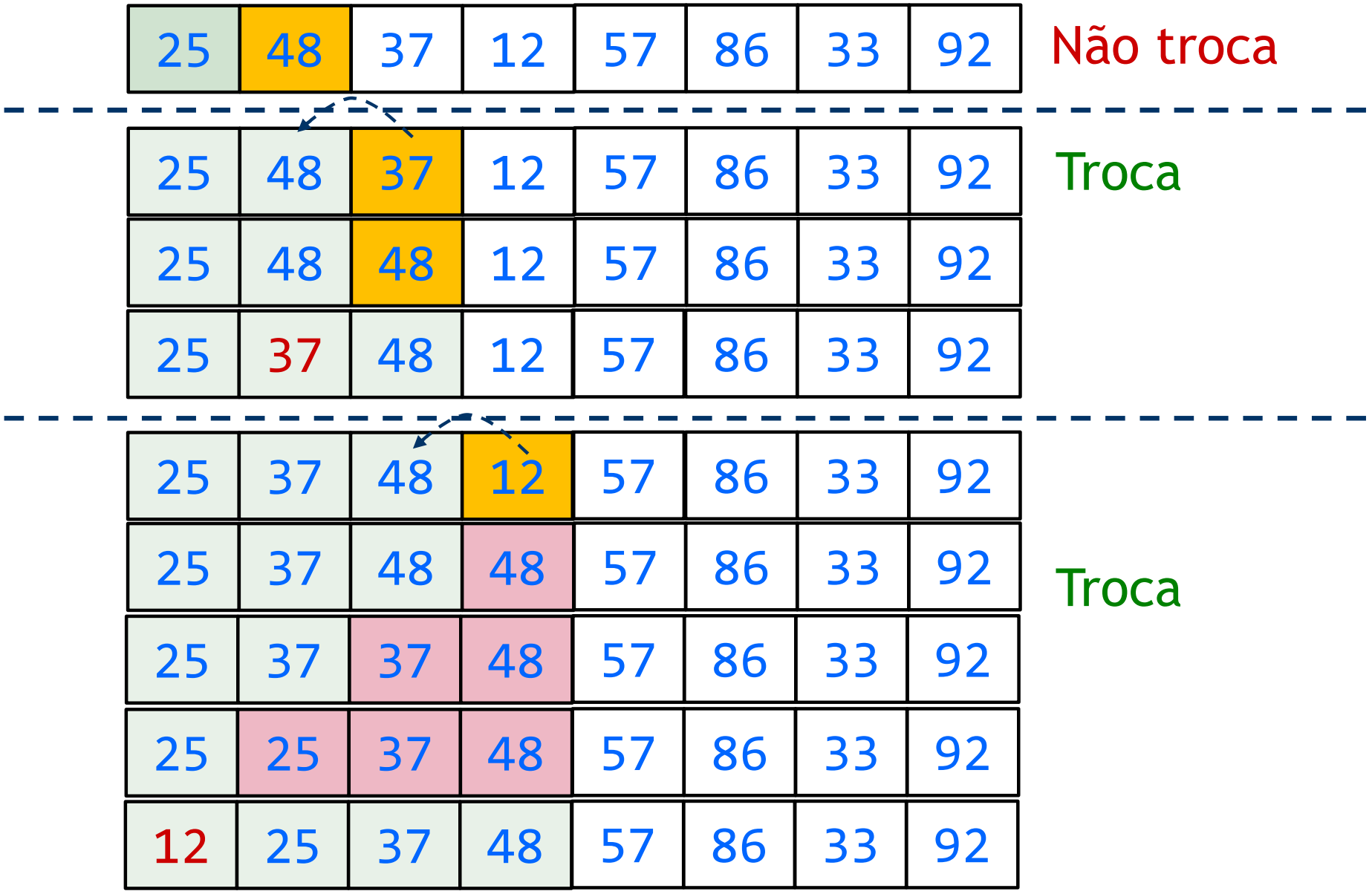
- **Princípio de Funcionamento**
  - A lógica consiste em dividir, virtualmente, um array em dois conjuntos: um ordenado (A_{ord}) e um desordenado (A_{des})
  - O A_{ord} reside no início do array, e a cada iteração dos elementos de A_{des}, acrescenta-se em A_{ord} o elemento na sua posição correta



  - Compara-se o elemento **`j=0`** de A_{des} com os elementos **`i`** de A_{ord}, do final para o início. Se **A_{des}`[j]` < A_{ord}`[i..0]`**, desloca à direita

Insertion Sort

- Simulação



Insertion Sort

- Simulação

Insertion Sort

- **Desempenho**
  - Mais rápido em relação a outros métodos tidos como básicos: BubbleSort e Seleção Direta
- **Indicação de uso**
  - Ideal para ordenação de pequenos conjuntos de dados, devido à baixa eficiência

Vantagem	Desvantagem
Simplicidade de **codificação** e **entendimento** do algoritmo	Se a coleção estiver ordenada antes do término do algoritmo, continua a execução até o final
Não efetua troca quando o elemento a ordenar está no lugar	Realiza deslocamento de dados para inserir um elemento no local adequado

Insertion Sort

- Codificação

```
def insertionSort(array):
    for i in range(1,len(array)):
        chave = array[i]
        j = i-1
        while j>=0 and chave < array[j]
            array[j+1] = array[j]):
            j -= 1
        array[j+1] = chave
# main
v = [5, 9, 7, 21, 18, 1, 4]
print(v)
insertionSort( v )
print(v)
```

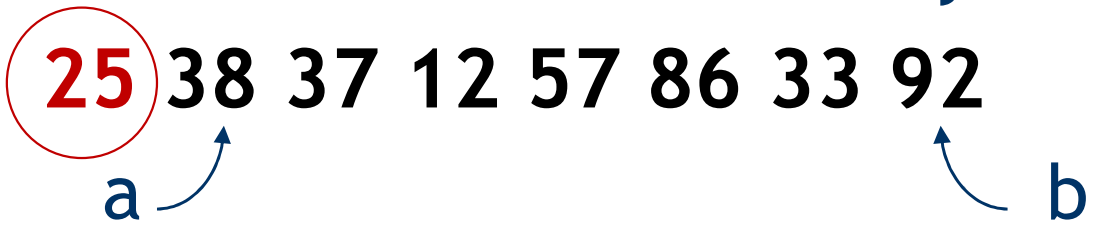
  - Solução recursiva

Quicksort

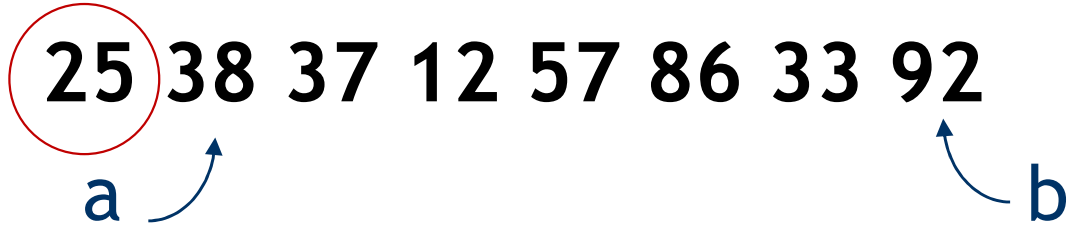
- Algoritmo baseado no princípio **"dividir para conquistar"**
  - Resolução de um problema maior dividindo-o em dois ou mais problemas menores
- **Princípio de Funcionamento**
  - Considere a sequencia de elementos de um array $v = [v_1, v_2, v_3, \ldots, v_n]$
  - Escolher um elemento aleatório "p" dentro do vetor, o qual chamaremos de **pivô**
    - ✓ Podemos escolher como **pivô** o primeiro elemento do vetor, com a finalidade de determinar sua correta posição na primeira passagem do algoritmo

Quicksort

- **Princípio de Funcionamento**
  1. Determine o índice do **pivô** como 0 (p = 0)
  2. Adicione os índices controladores **a** como o índice subsequente ao pivô e **b** ao índice relativo ao último elemento do array



  3. Comparar os elementos v[a], com a = [1..n], até encontrar um elemento v[a] > **pivô**

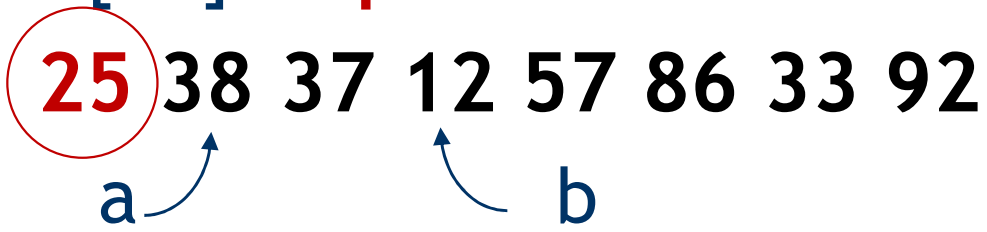


OBS: Note que **a** não avançou pois **v[a] > pivô**

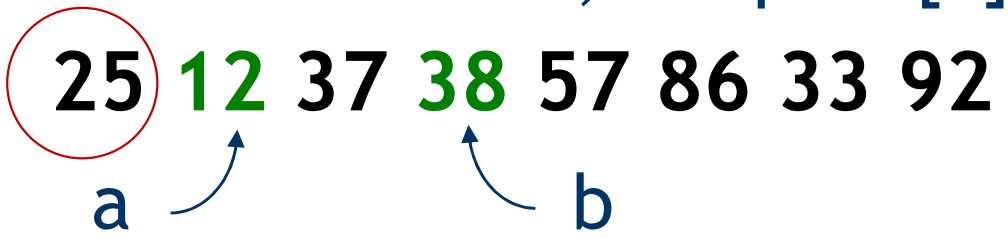
Quicksort

- **Princípio de Funcionamento**

4. A partir do final do array, compare os elementos v[b], com b = [tam...1], até encontrar um elemento v[b] <= **pivô**



5. Se **a** e **b** não se cruzarem, troque v[**a**] com v[**b**]

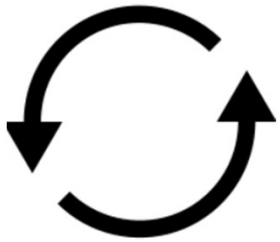


6. Continue a busca fazendo **a = a + 1** e **b = b - 1**

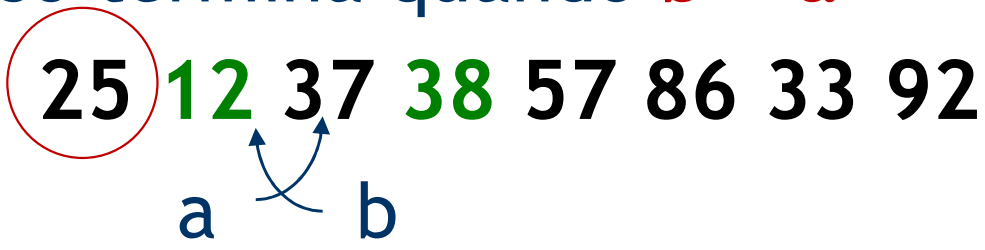
Quicksort

- **Princípio de Funcionamento**

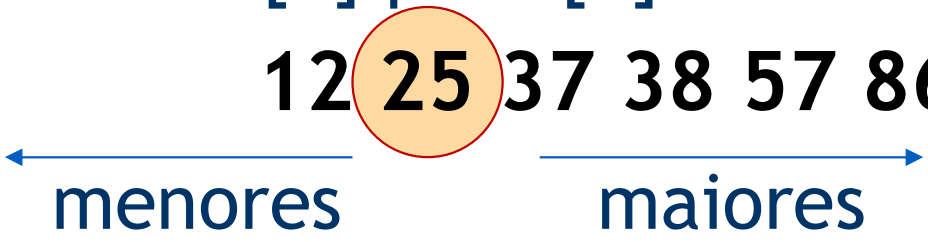
7. Repita o processo de avanço de **a** e **b** conforme suas condições de parada



   - O processo termina quando **b < a**



8. Uma vez que a posição do **pivô** já está definida, troca-se v[**0**] por v[**b**]



   - ✓ Todos os elementos depois do pivô são **maiores** que ele. Os antecedentes são **menores** que o pivô

Quicksort

- **Princípio de Funcionamento**
  - Uma vez posicionado o pivô, inicia-se o particionamento do array em dois subconjuntos, a partir do índice de posicionamento do **pivô**

12 25 37 38 57 86 33 92

    - ✓ partição **esq**: 12 (índice 0..1)
    - ✓ partição **dir**: 37 38 57 86 33 92 (índice 2..7)
  - Repete-se, então, o mesmo procedimento determinado pelos passos de 1 a 8
    - ✓ Indicar o pivô
    - ✓ Achar a posição correta do pivô
    - ✓ Aplicar recursivamente o algoritmo nas partições da esquerda e da direita

Quicksort

- **Inconveniente**
  - Como poderemos evitar que o processamento se repita mesmo depois do vetor estar ordenado?
- **Comportamento**

Vantagem	Desvantagem
Ordenação **rápida**, em média	Se o vetor estiver ordenado, o algoritmo prossegue de qualquer forma

- **Indicação de uso**
  - Coleções desordenadas

Quicksort

- Codificação
  - Solução recursiva

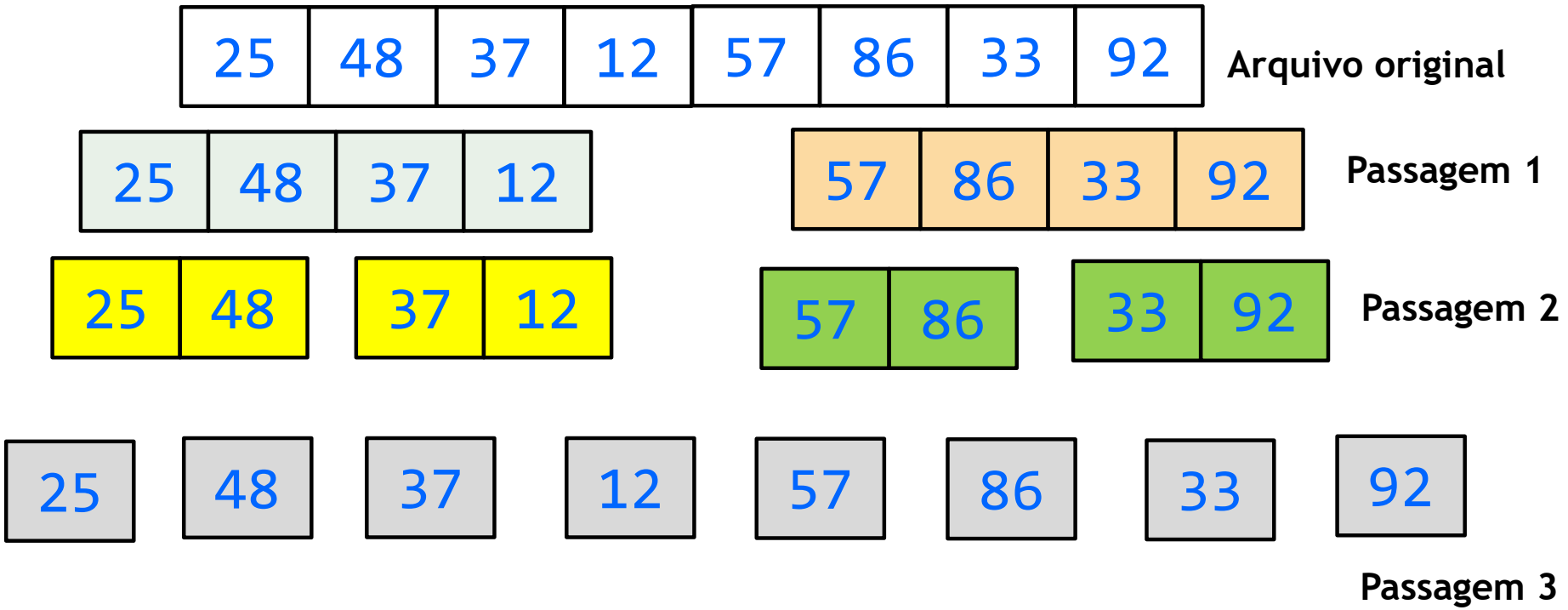
```
def quickSort(array):
    quickSortRun(array,0,len(array)-1)

def quickSortRun(array,low,high):
    if low < high:
        pi = partition(array,low,high)
        quickSortRun(array, low, pi-1)
        quickSortRun(array, pi+1, high)

def partition(array,low,high):
    pivot = array[low] # pivot
    a = low + 1
    b = high
     ...
    return b
```

MergeSort

- Algoritmo também baseado no princípio **"dividir para conquistar"**
  - Resolução de um problema maior dividindo-o em dois ou mais problemas menores
- **Princípio de Funcionamento**
  - Considere a sequencia de elementos de um array $\mathbf{v} = [v_1, v_2, v_3, \ldots, v_n]$
  - O array de entrada **v** é dividido pela metade, repetindo o processo em suas metades até que não possam ser mais divididos
    - ✓ A divisão termina quando o **array** atinge o tamanho de 1 unidade

MergeSort

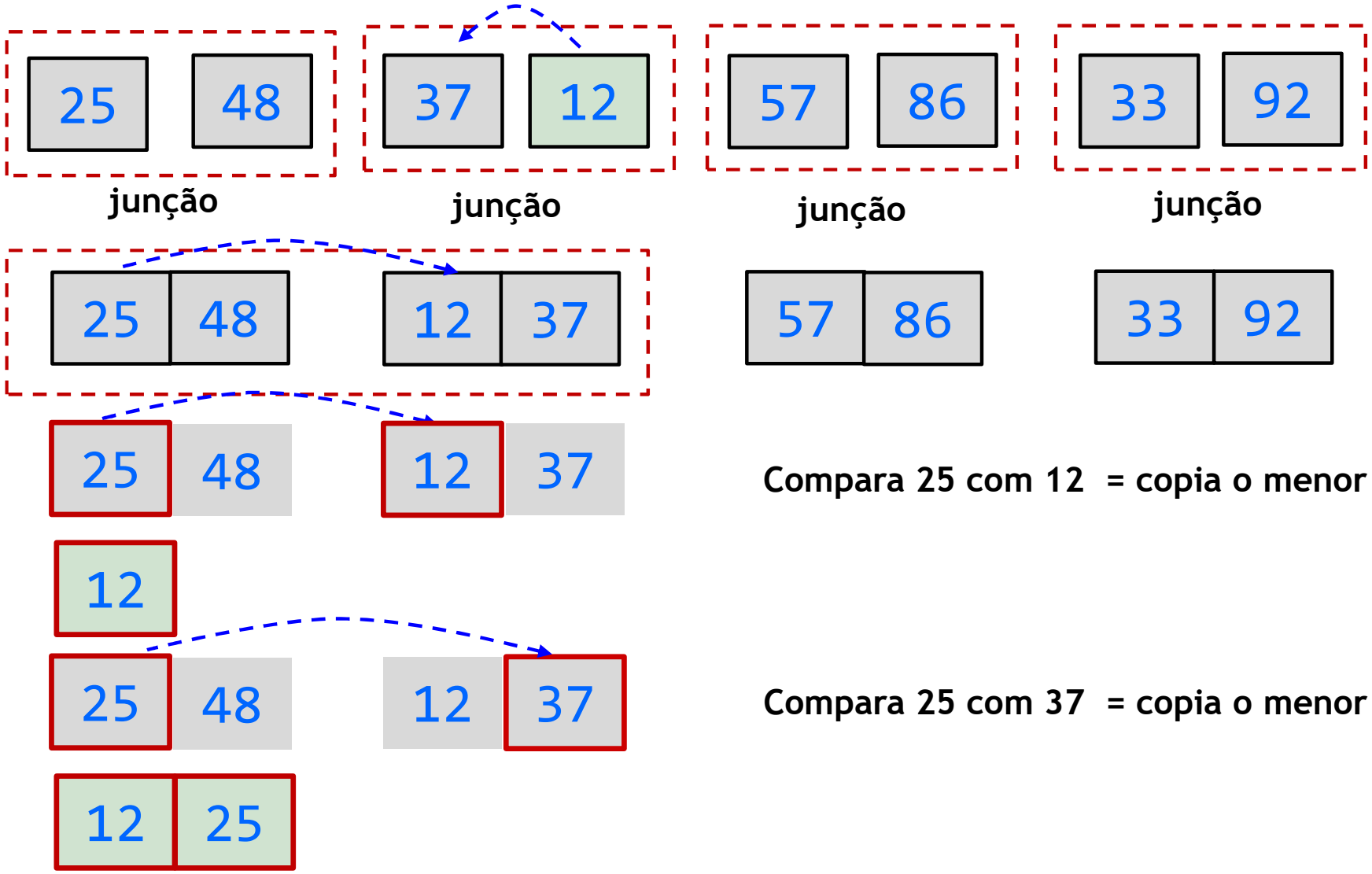
- Simulação



- Efetuar a junção (merge)
  - Realiza a junção dos subarrays unitários de forma ordenada, comparando os elementos

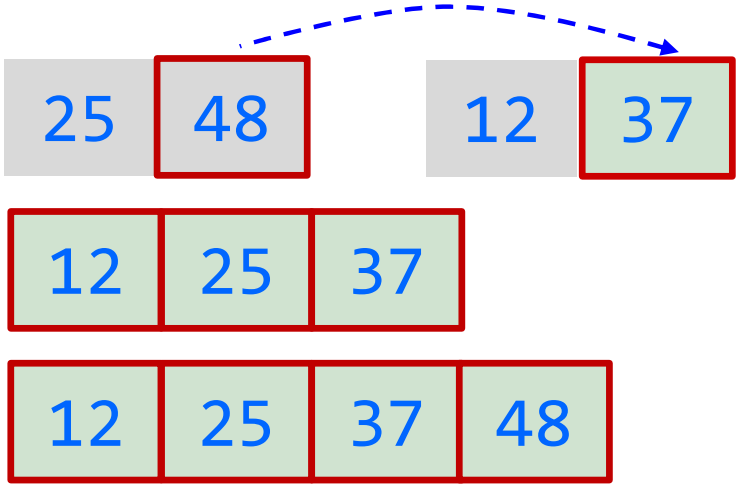
MergeSort

- Simulação



MergeSort

- Simulação

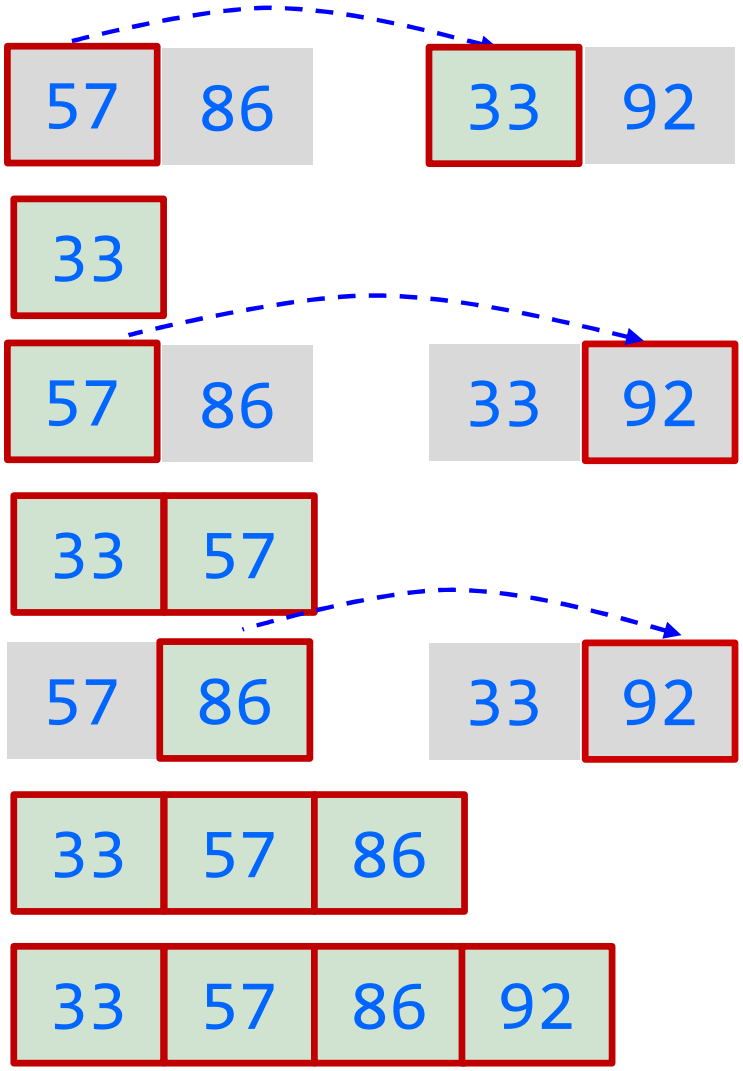


Compara 48 com 37 = copia o menor

O elemento que sobrou vai para o final do array

MergeSort

- Simulação



Compara 57 com 13 = copia o menor

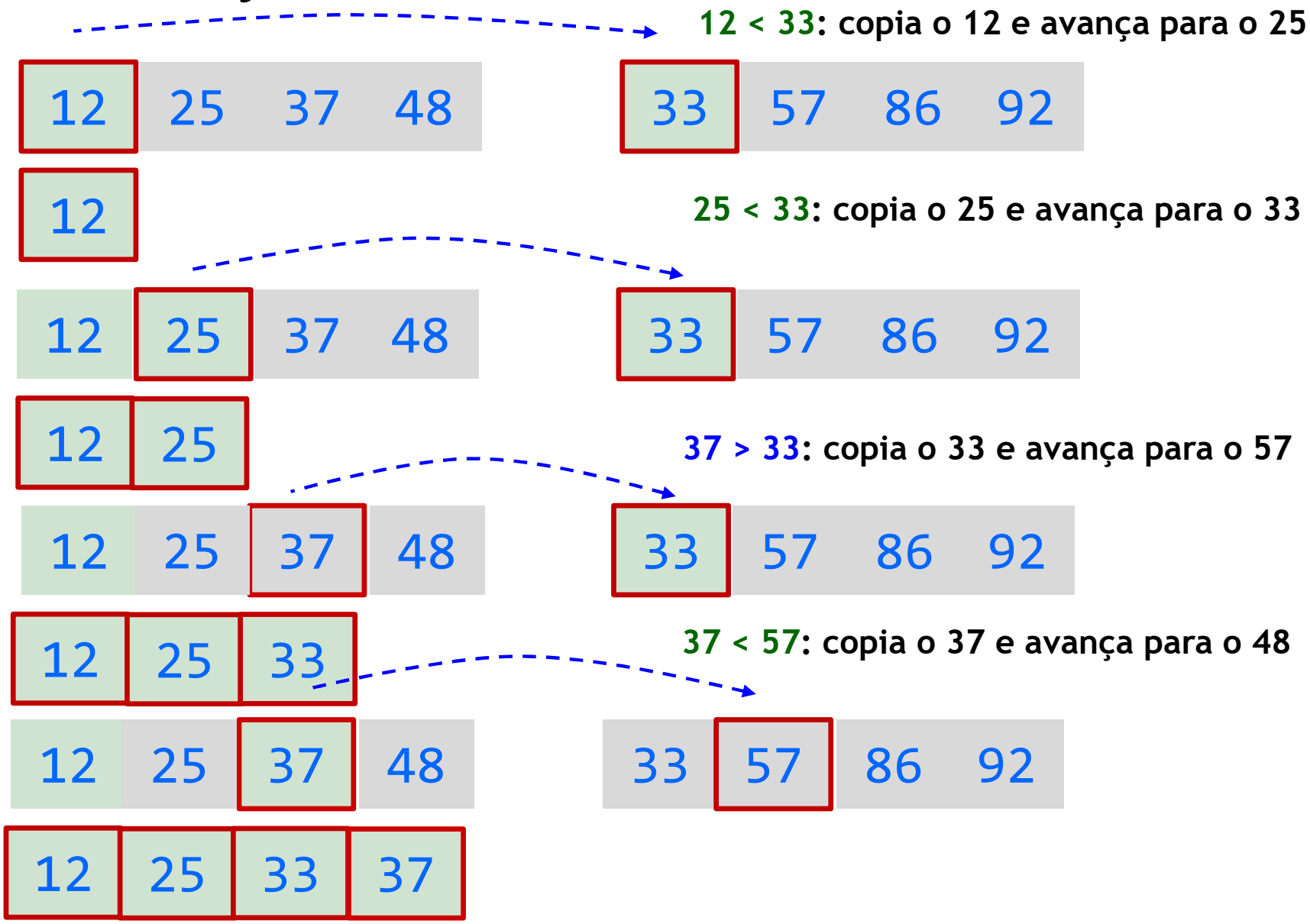
Compara 57 com 92 = copia o menor

Compara 48 com 37 = copia o menor

O elemento que sobrou vai para o final do array

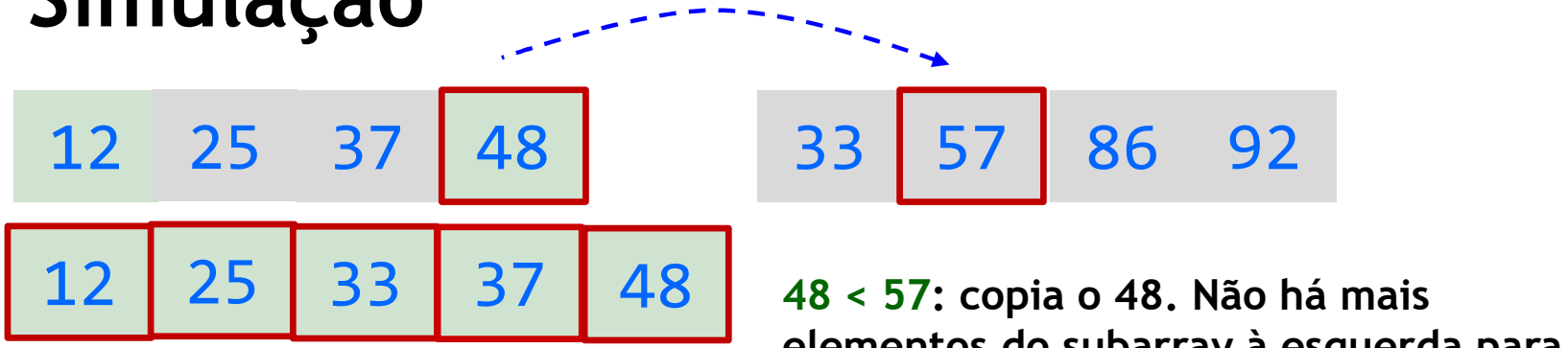
MergeSort

- Simulação



MergeSort

- Simulação



48 < 57: copia o 48. Não há mais elementos do subarray à esquerda para avançar

Copia o que sobrou do subarray à direita para o array final

MergeSort

- **Desvantagem**
  - Se o array estiver ordenado, o algoritmo prossegue de qualquer forma
- **Vantagem**
  - Ordenação rápida, em média
- **Indicação de uso**
  - Independe do conjunto de entrada

MergeSort

- Codificação

```
def mergeSort(myList):

    if len(myList) > 1:
        mid = len(myList) // 2
        left = myList[:mid]
        right = myList[mid:]

        # chamada recursiva para a divisão
        mergeSortList(right)
        mergeSortList(left)

        # merge...
        # Examine o código
        ...
```

Referências Bibliográficas

- Sorting Algorithms.
  https://www.geeksforgeeks.org/sorting-algorithms/
- Animação

  MergeSort:
  https://www.youtube.com/watch?v=JSceec-wEyw

Timsort

- Algoritmo de ordenação híbrido derivado do princípio do **insertion sort** e **merge sort**
  - Utilizado nas operações **sorted()** e **sort()** the Pyhon e **Arrays.sort()** de Java
  - Complexidade: **`n log n`** (média)
  - Criador: Tim Peters (2002)
- **Princípio de Funcionamento**
  - O array original é dividido em **blocos** (subarrays) denominados de **runs**.
  - Aplica-se o **`insertion sort`** em cada **bloco**
  - Os blocos são mescaldos utilizando a função de combinação do **`merge sort`**

Timsort

- Simulação da Ordenação

15 27 35 46 55 63 70 71 88 92 100