Aulas 4 e 5 - Ordenação

Módulo 1 - PrepTech Google

Ordenação

Ordenação

Conceito

Dado um vetor **v** com **N** elementos, a ordenação consiste em organizar todos esses **N** elementos em uma ordem (crescente, não-decrescente etc)

Exemplo
v = [1, 9, 8, 5, 3, 7, 4]

- Depois de ordenado de forma não-decrescente, fica:
v = [1, 3, 4, 5, 7, 8, 9]

Ordenação

Conceito

Exemplos de **algoritmos de ordenação** por comparação:

- Selection Sort
- Bubble Sort
- Mergesort
- Quicksort
- Heapsort

Exemplos de **algoritmos de ordenação** sem comparação:

- Count Sort
- Radix Sort

Selection Sort

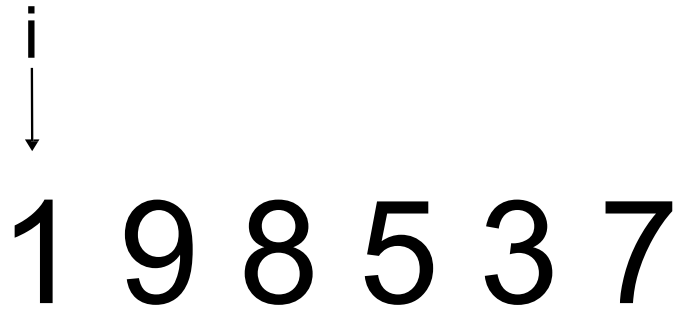
Selection Sort

Conceito

- Consiste em posicionar o menor ou o maior elemento em seu lugar correto **N** vezes consecutivas (onde N é o tamanho do vetor)

Selection Sort

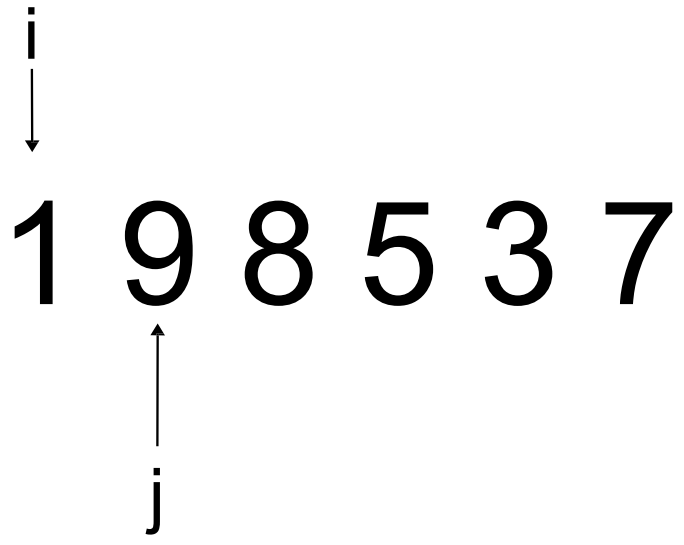
Funcionamento



O ponteiro **i** denota onde queremos inserir o menor elemento.

Selection Sort

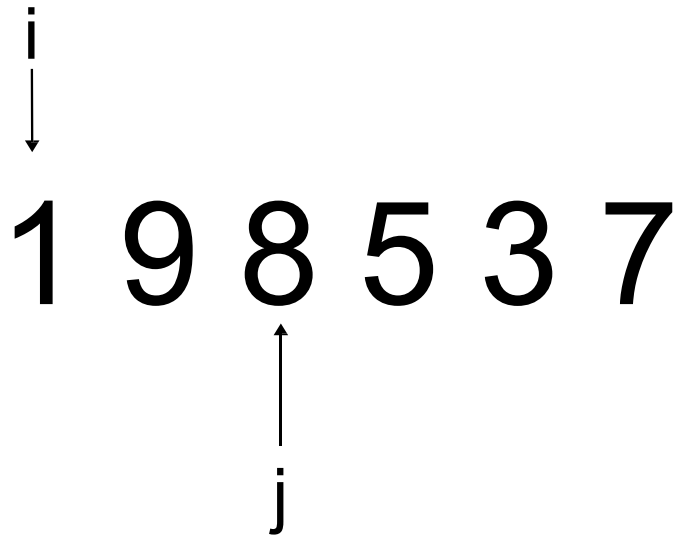
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

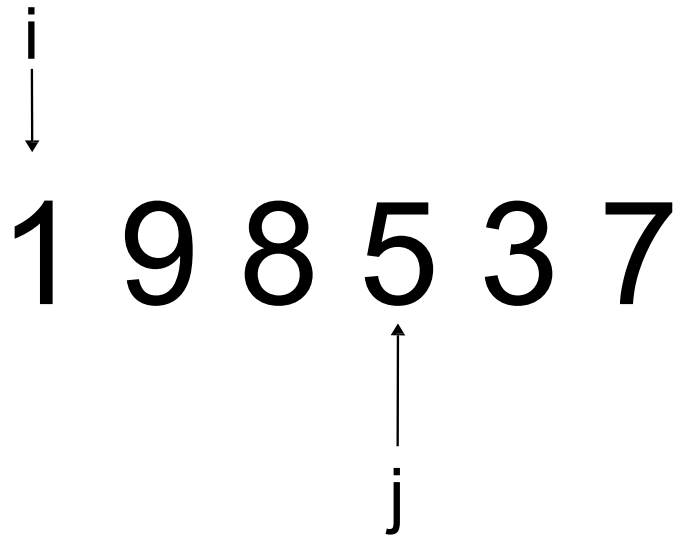
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

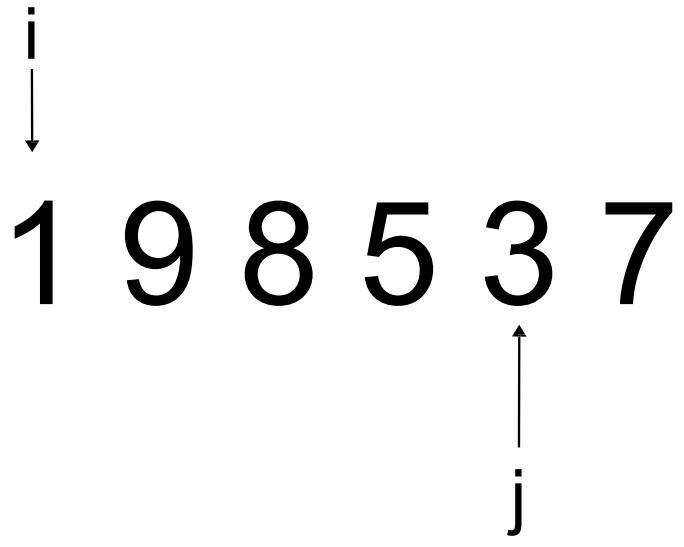
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

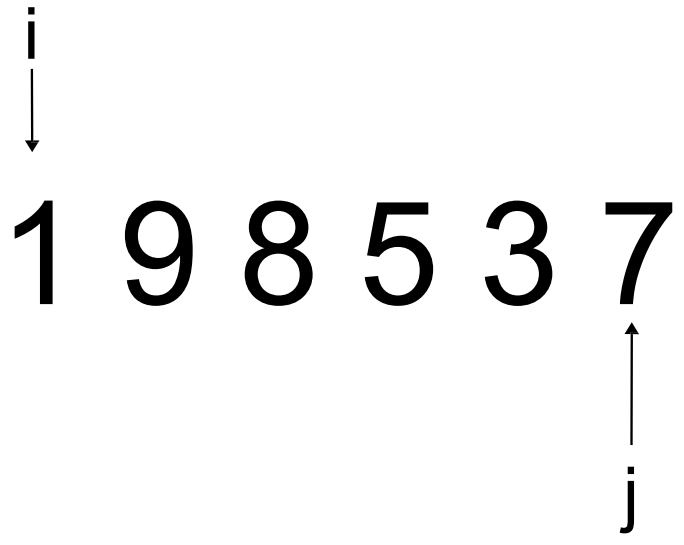
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

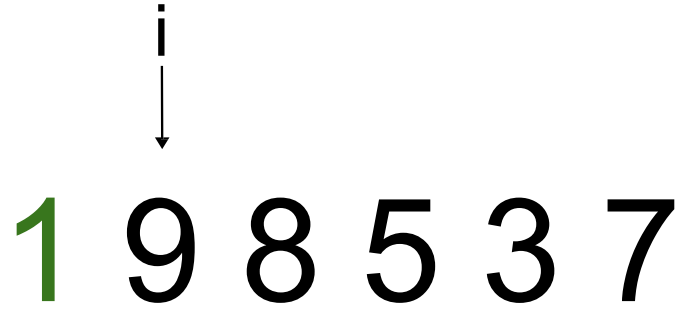
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

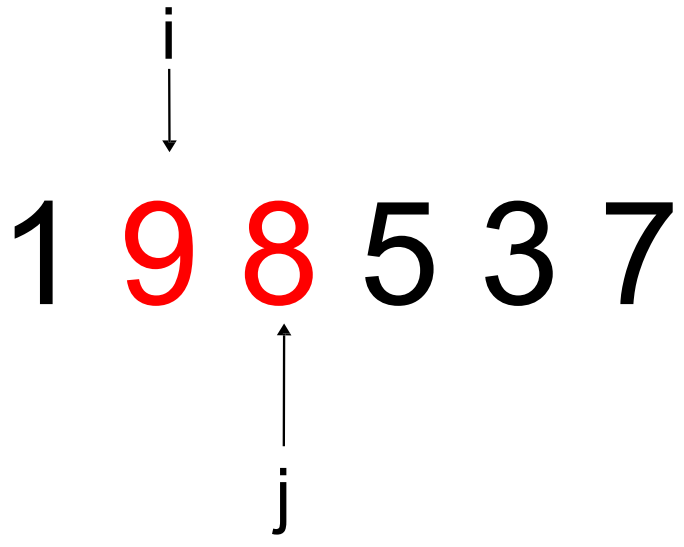
Funcionamento



Elementos na esquerda de **i** já **ordenados**. O menor elemento na posição zero.

Selection Sort

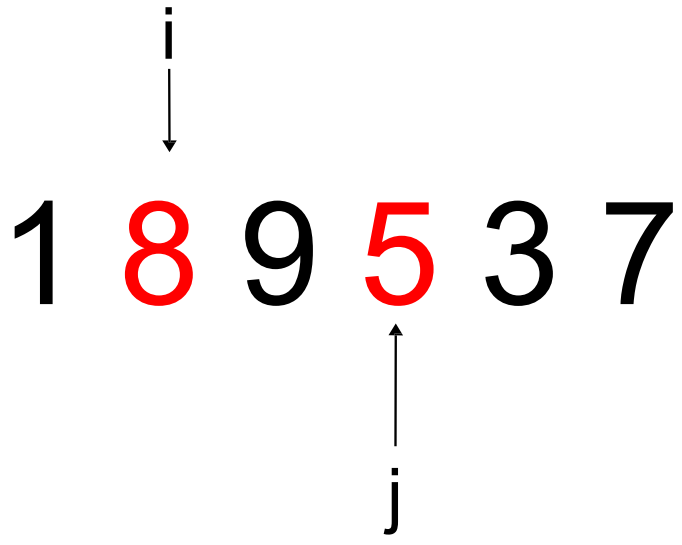
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

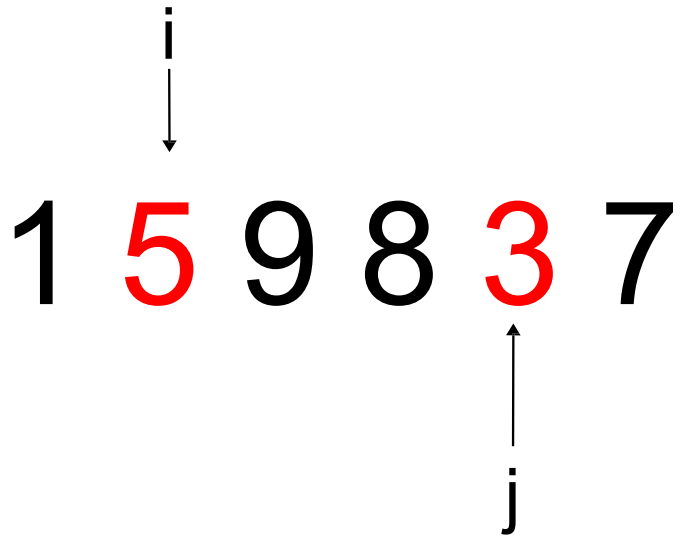
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

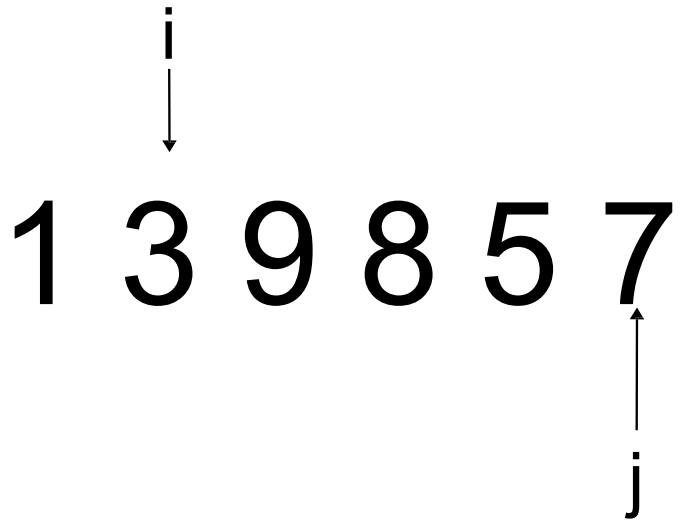
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

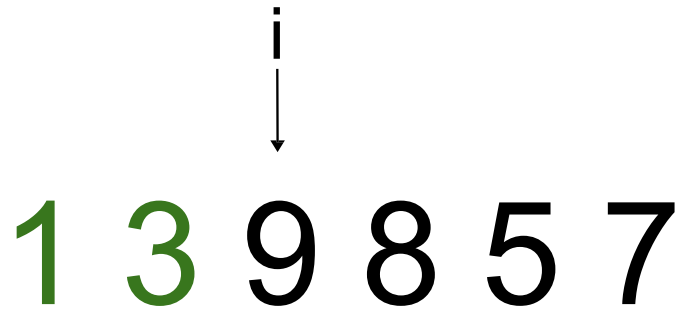
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

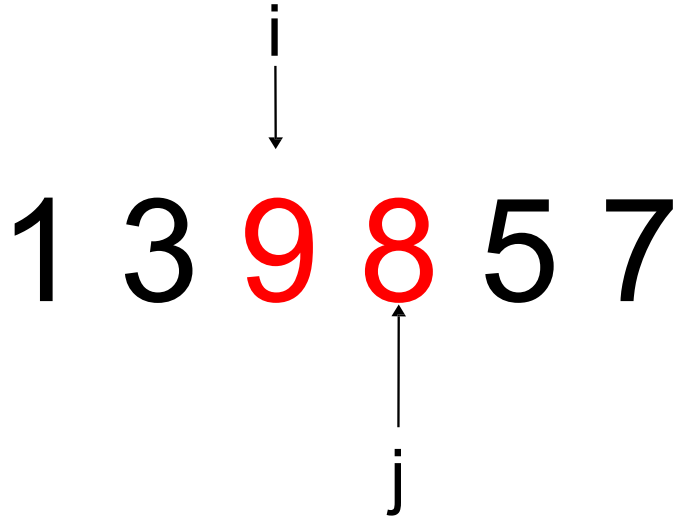
Funcionamento



Elementos na esquerda de **i** já **ordenados**, em seu lugar definitivo.

Selection Sort

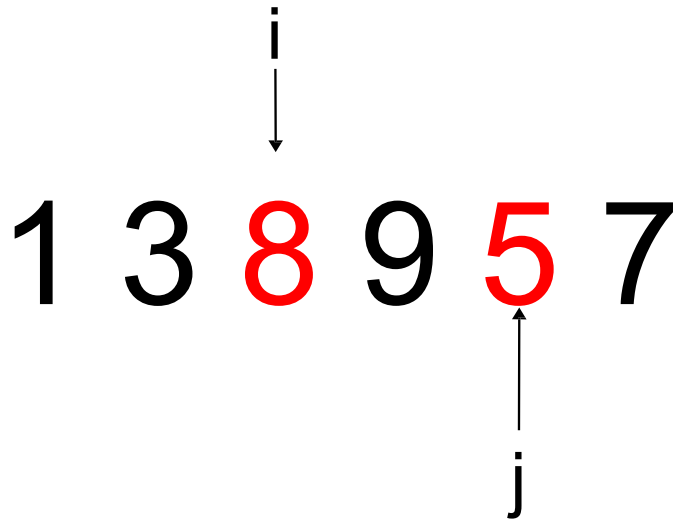
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

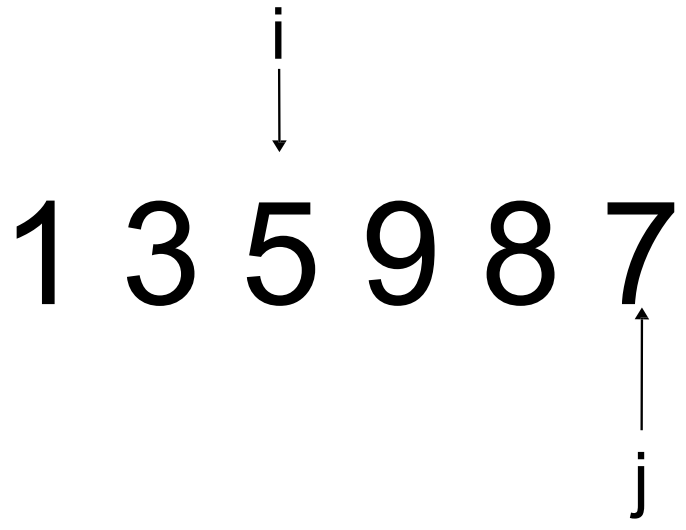
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

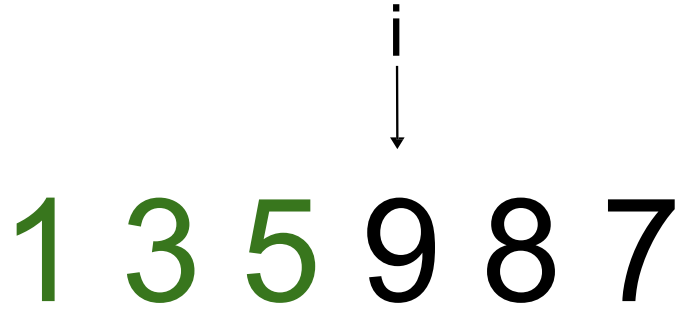
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

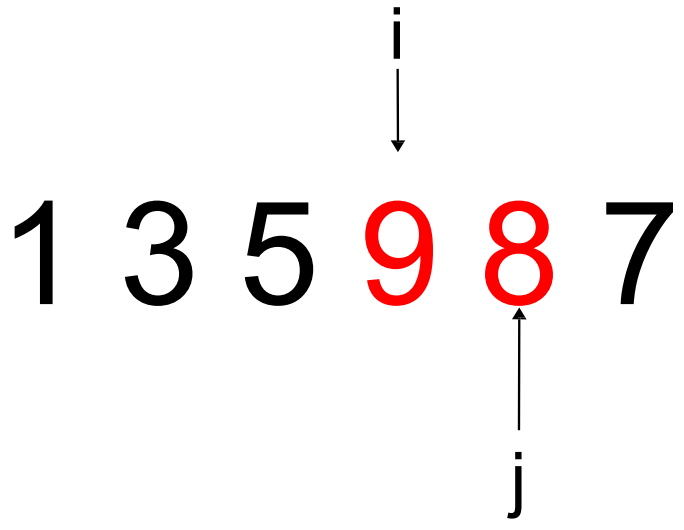
Funcionamento



Elementos na esquerda de **i** já **ordenados**, em seu lugar definitivo.

Selection Sort

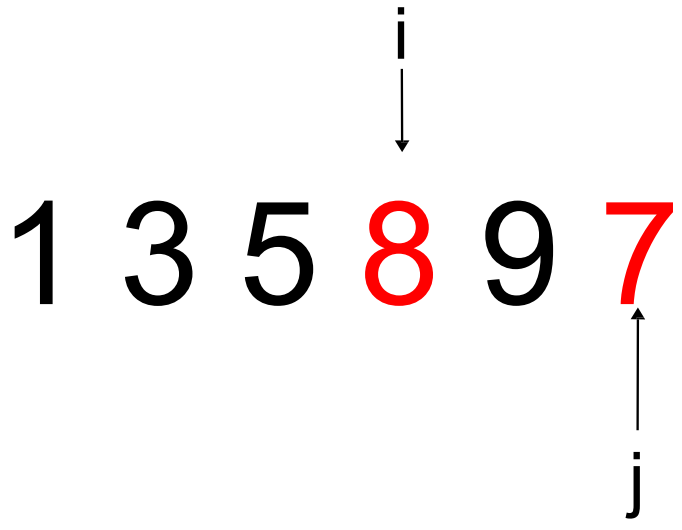
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

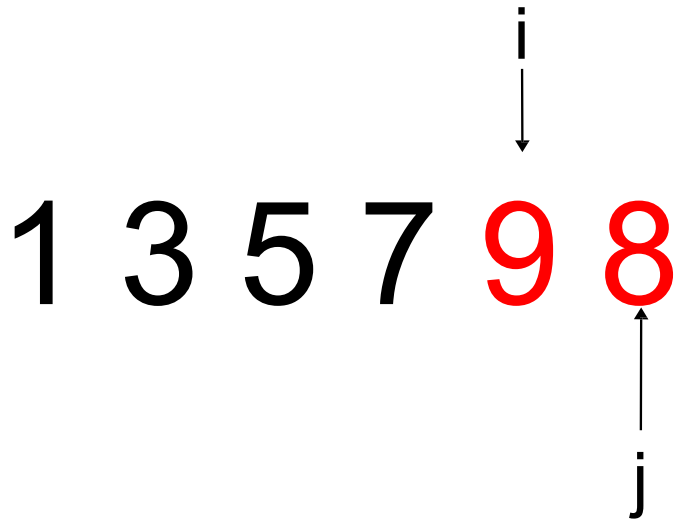
Selection Sort

Funcionamento

Elementos na esquerda de **i** já **ordenados**, em seu lugar definitivo.

Selection Sort

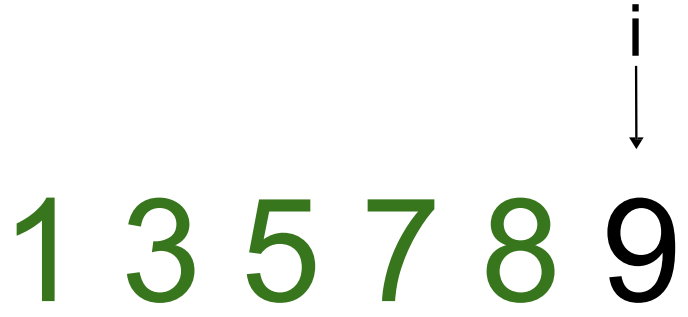
Funcionamento




```
if v[j] < v[i]:
    v[i], v[j] = v[j], v[i] #swap
```

Selection Sort

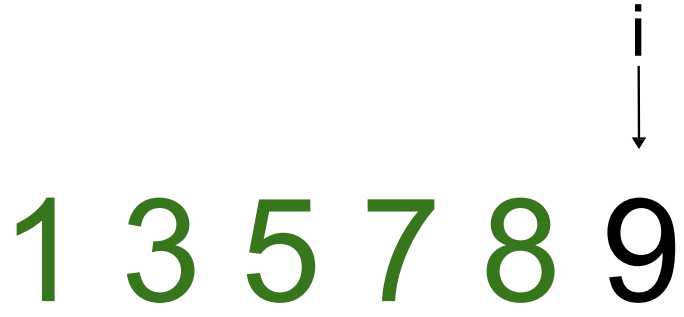
Funcionamento



Elementos na esquerda de **i** já **ordenados**, em seu lugar definitivo.

Selection Sort

Funcionamento



Agora todos os elementos na posição final.

Selection Sort

Funcionamento

1 3 5 7 8 9

Vetor ordenado!

Selection Sort

Código em Lua

```
for i=1,N do
    for j=i+1,N do
        if v[i]>v[j] then
            v[i],v[j] = v[j],v[i]
        end
    end
end
```

Selection Sort

Código em Python

```
for i in range( len(arr) ):
    for j in range(i+1, len(arr)):
        if v[j]<v[i]:
            v[i], v[j] = v[j], v[i]
```

Selection Sort

Código

1 3 5 7 8 9

Número de operações:

$$(N-1) + (N-2) + (N-3) + \ldots + 3 + 2 + 1$$

$$\frac{N*(N-1)}{2}$$

Bubble Sort

Bubble Sort

Código

- Ideia:
  - Se o vetor já estiver ordenado em alguma interação, seria interessante detectar isso e parar o algoritmo.
- Como detectar se um vetor está ordenado?

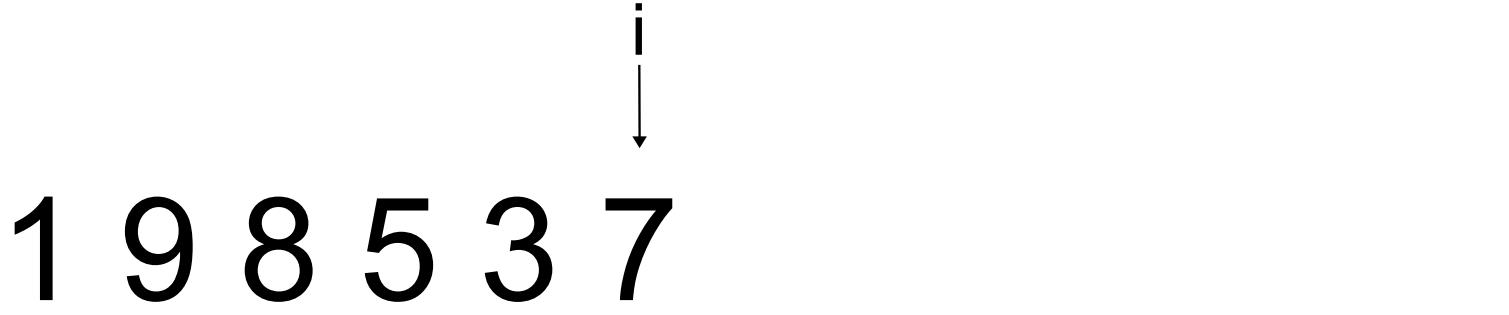
```
sorted = true

for i=1,N do
   if v[i]>v[i+1] then sorted = false end

end
```

Bubble Sort

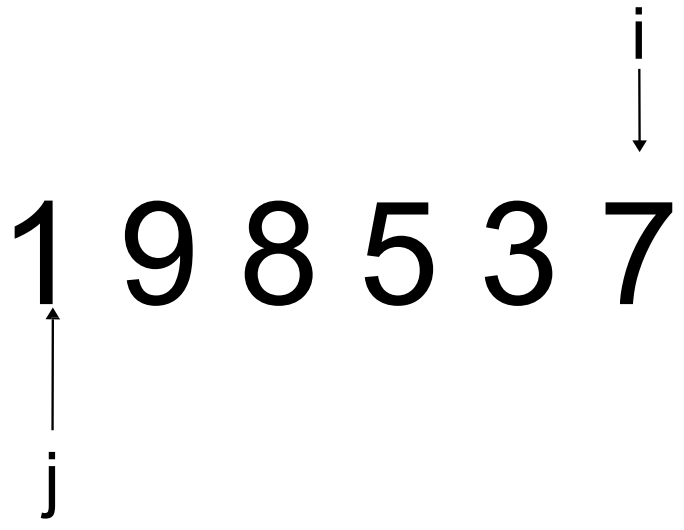
Funcionamento



A ideia agora vai ser colocar o **maior elemento** no final do vetor, comparando os **consecutivos**. O ponteiro **i** indica essa posição.

Bubble Sort

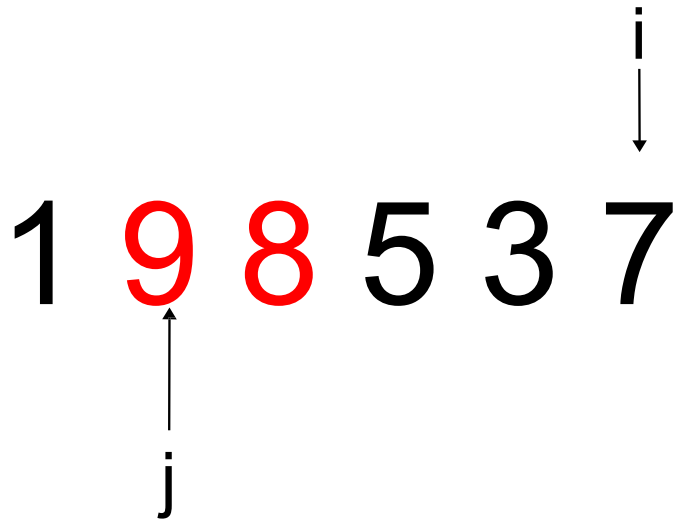
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

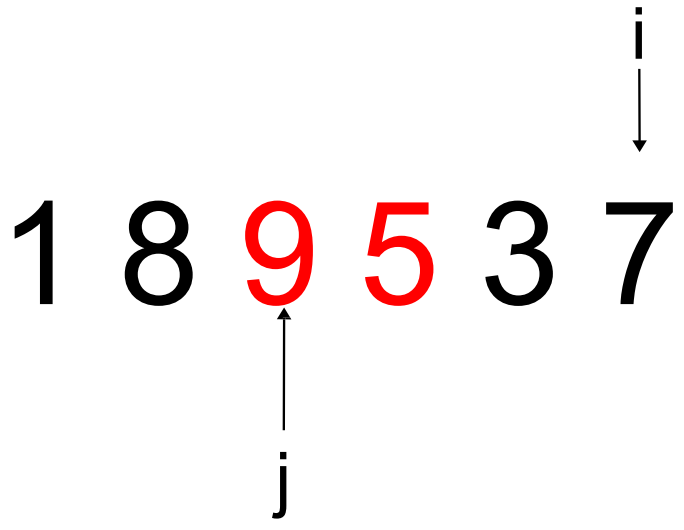
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

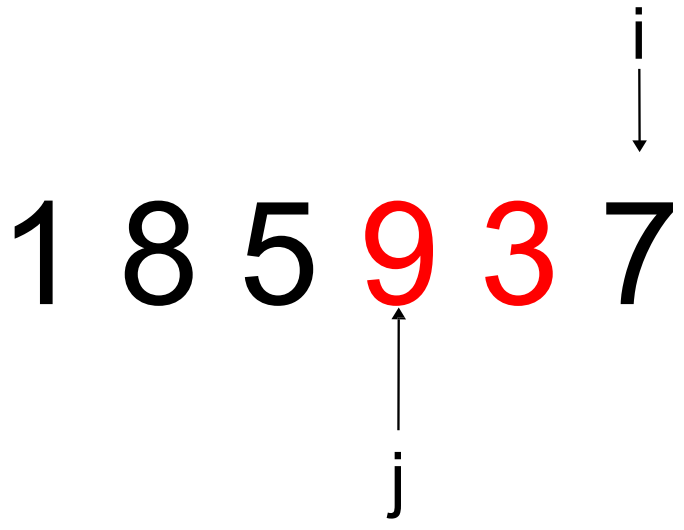
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

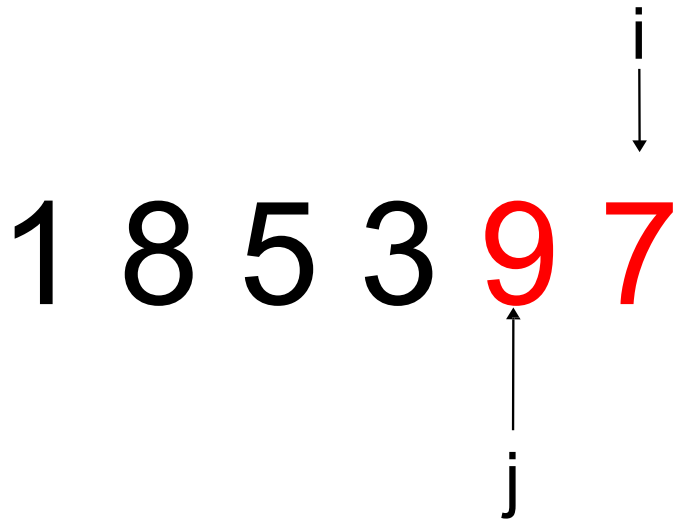
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

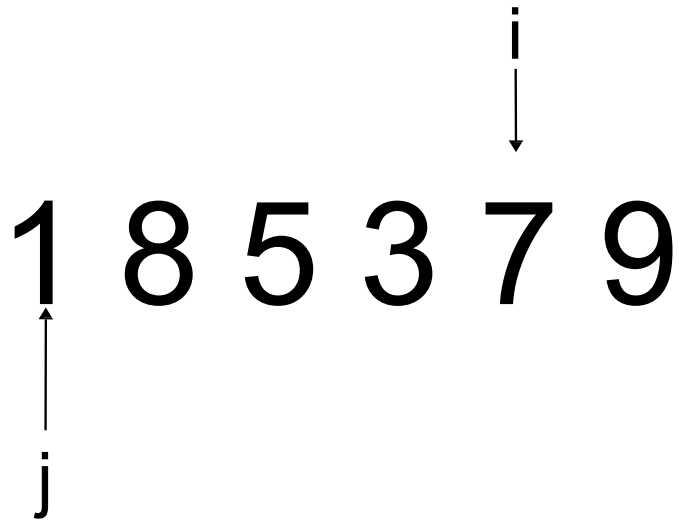
Funcionamento

1 8 5 3 7 9

O vetor na direita de **i** está com os **elementos ordenados**, na posição definitiva

Bubble Sort

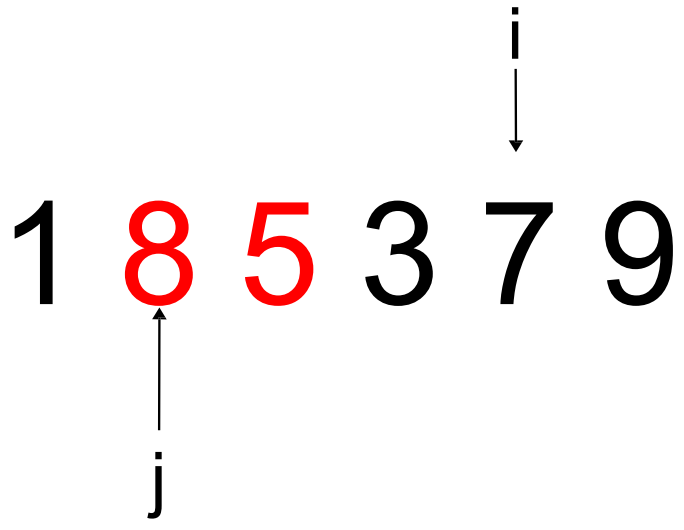
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

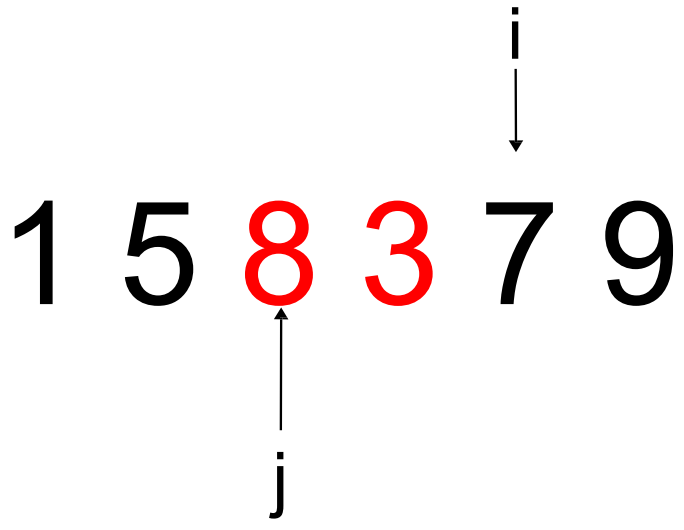
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

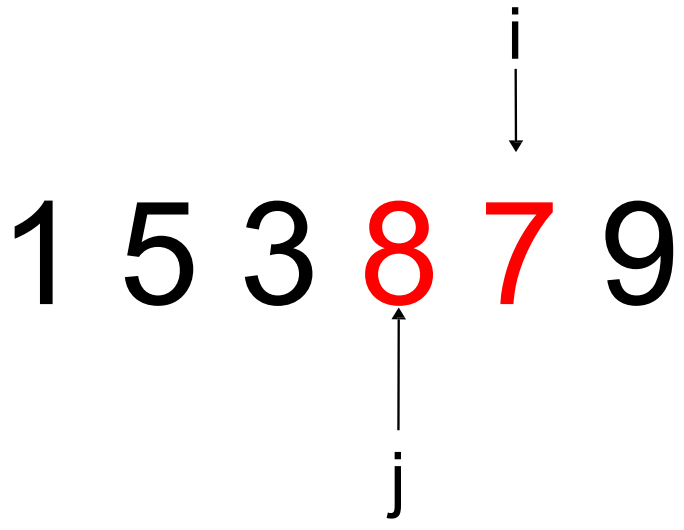
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

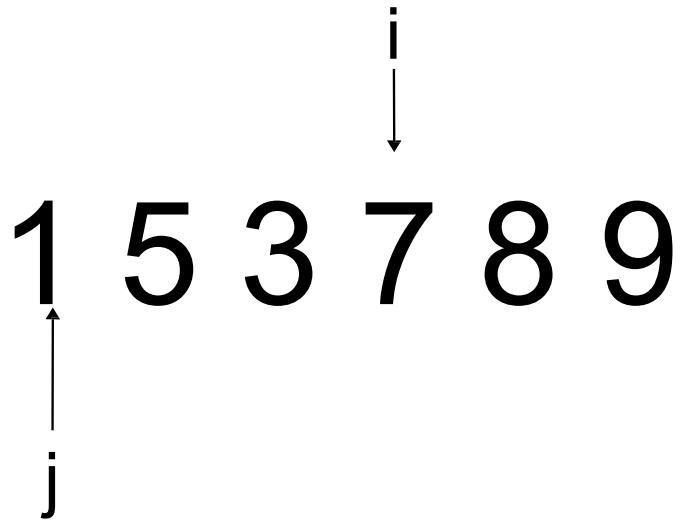
Funcionamento

1 5 3 7 8 9

O vetor na direita de **i** está com os **elementos ordenados**, na posição definitiva

Bubble Sort

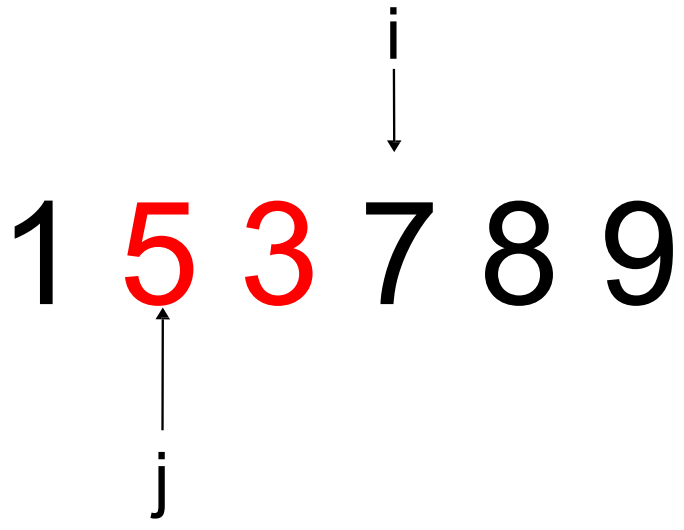
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

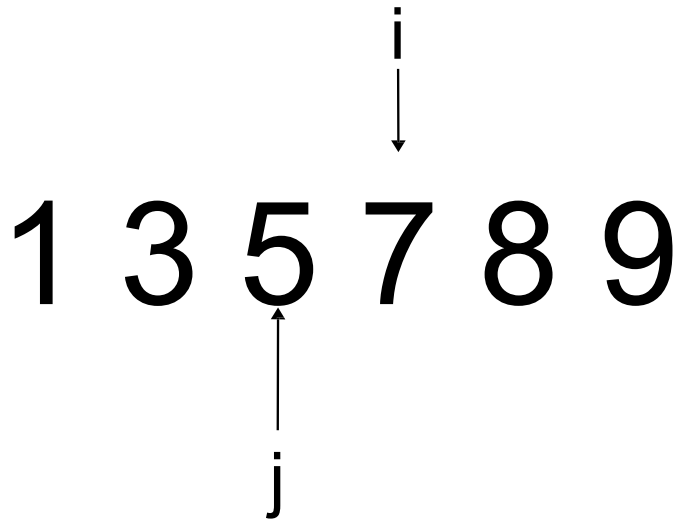
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

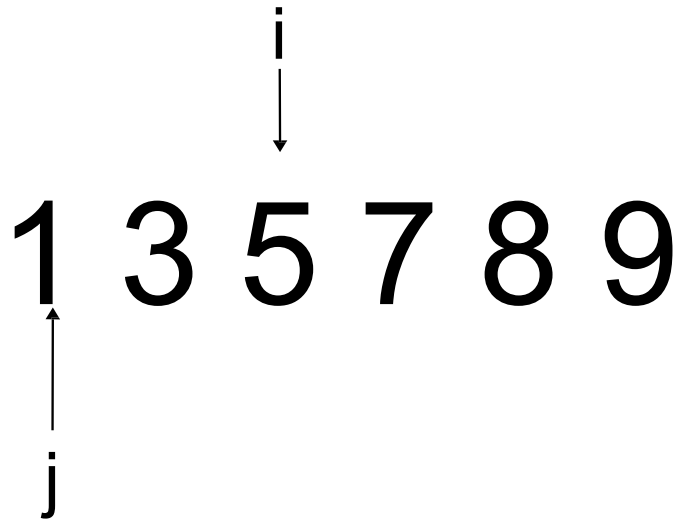
Bubble Sort

Funcionamento

O vetor na direita de **i** está com os **elementos ordenados**, na posição definitiva

Bubble Sort

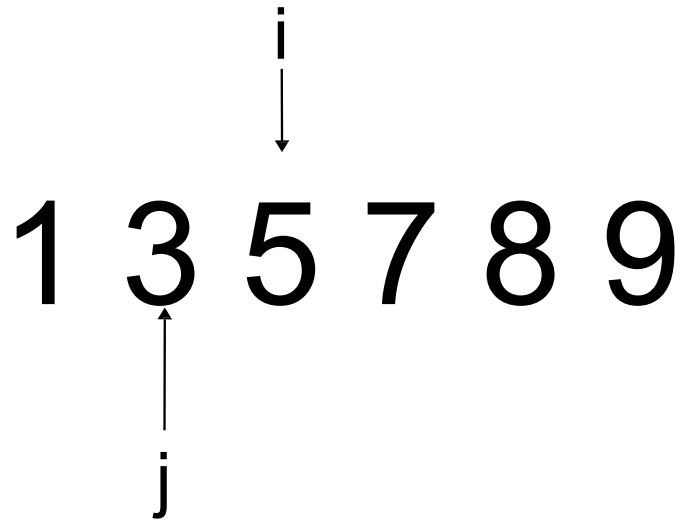
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

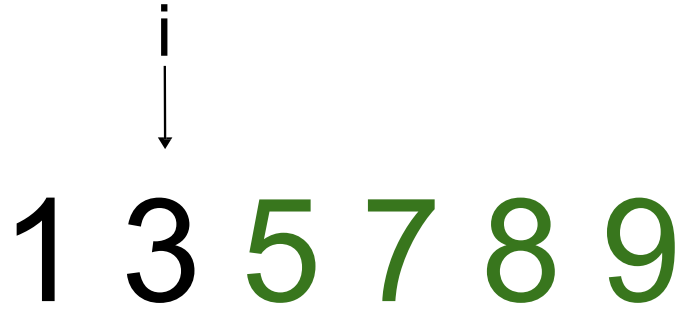
Funcionamento




```
if v[j]>v[j+1]:
    v[j], v[j+1] = v[j+1],v[j] #swap
```

Bubble Sort

Funcionamento



Não houve trocas! O vetor está ordenado!

Bubble Sort

Código em Lua

```
for i=N,2,-1 do
    anySwap = false
    for j=1,i-1 do
        if v[j]>v[j+1] then
            v[j],v[j+1] = v[j+1],v[j]
            anySwap = true
        end
    end
    if not anySwap then break end
end
```

Bubble Sort

Código em Python

```
for i in range( len(arr)-1, 0, -1 ):
    anySwap = False
    for j in range(i):
        if v[j]>v[j+1]:
            v[j], v[j+1] = v[j+1], v[j]
            anySwap = True
    if not anySwap:
        break
```

Selection and Bubble Sort

Prática no LeetCode - https://leetcode.com/problems/sort-an-array/description/

Implementar os dois algoritmos (*Insert e Bubble Sort*) e comparar o resultado de cada um deles em termos de tempo de processamento.

Mergesort

Mergesort

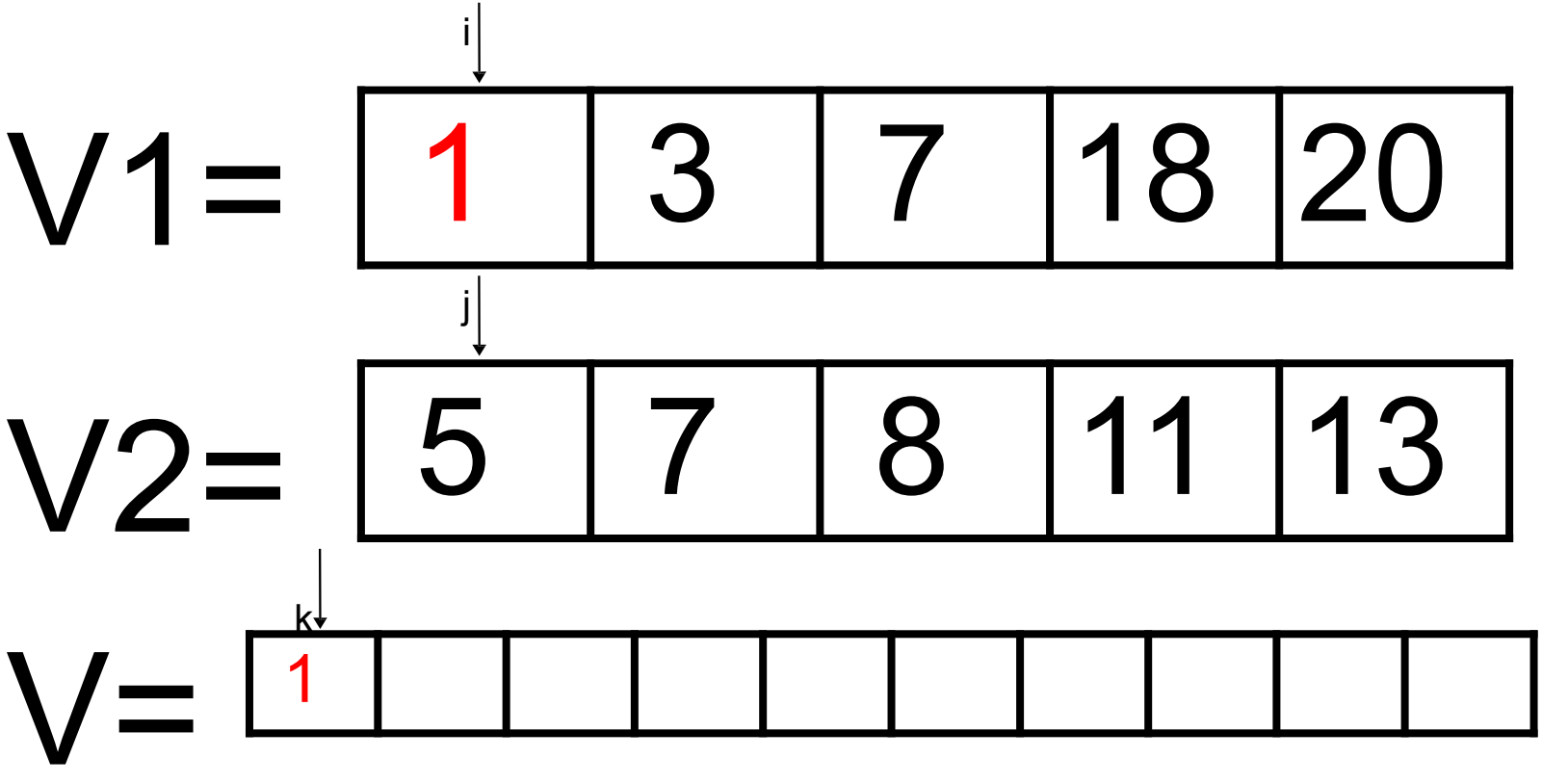
Conceito

- Ordena inicialmente sub-vetores de tamanho 1 ou 2 e recursivamente dobra esse tamanho até ordenar o vetor inteiro

- Baseia-se no merge de dois subvetores ordenados (veja animação disso nos próximos slides)

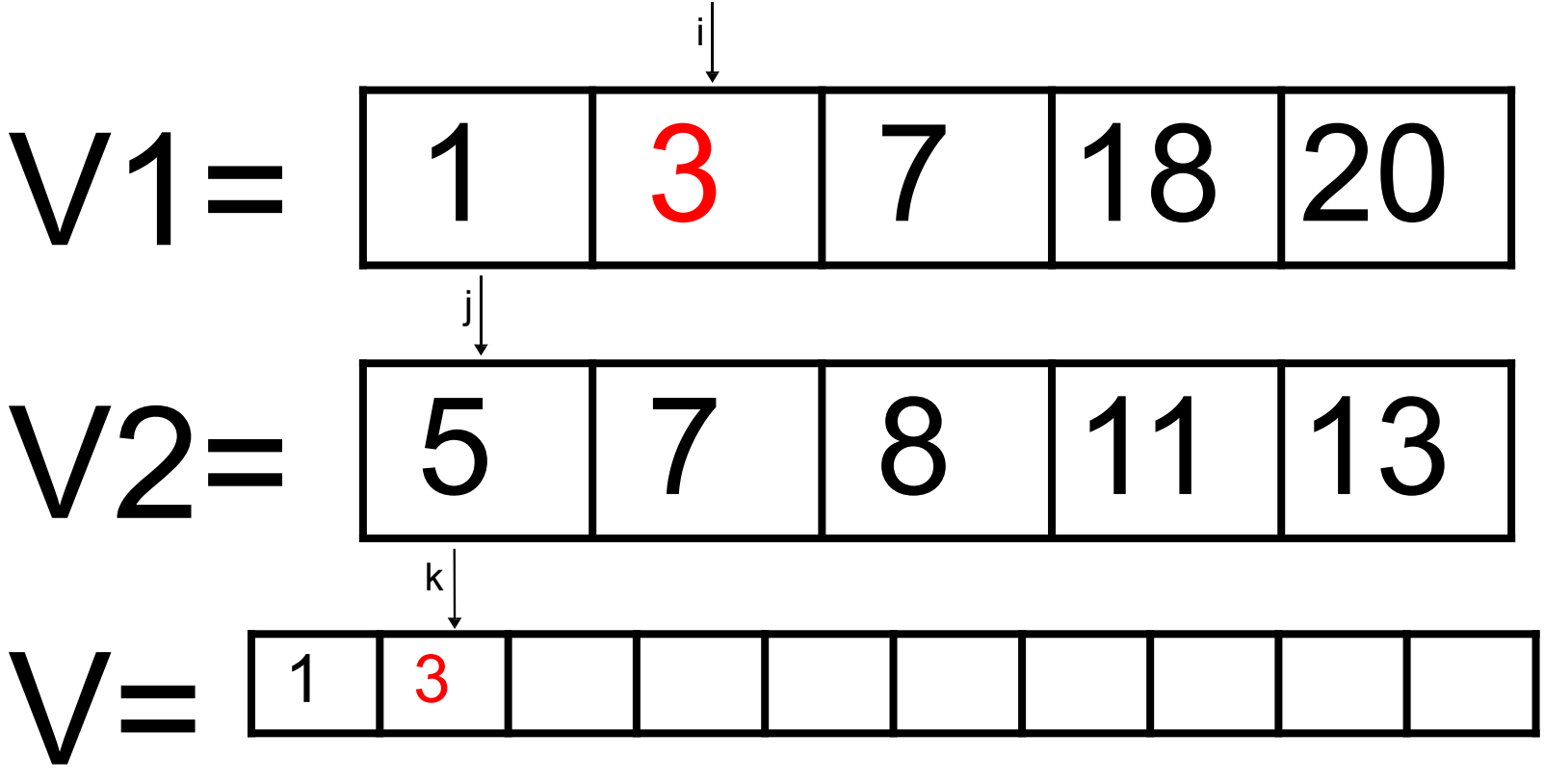
Merge de dois vetores ordenados

Código em Python



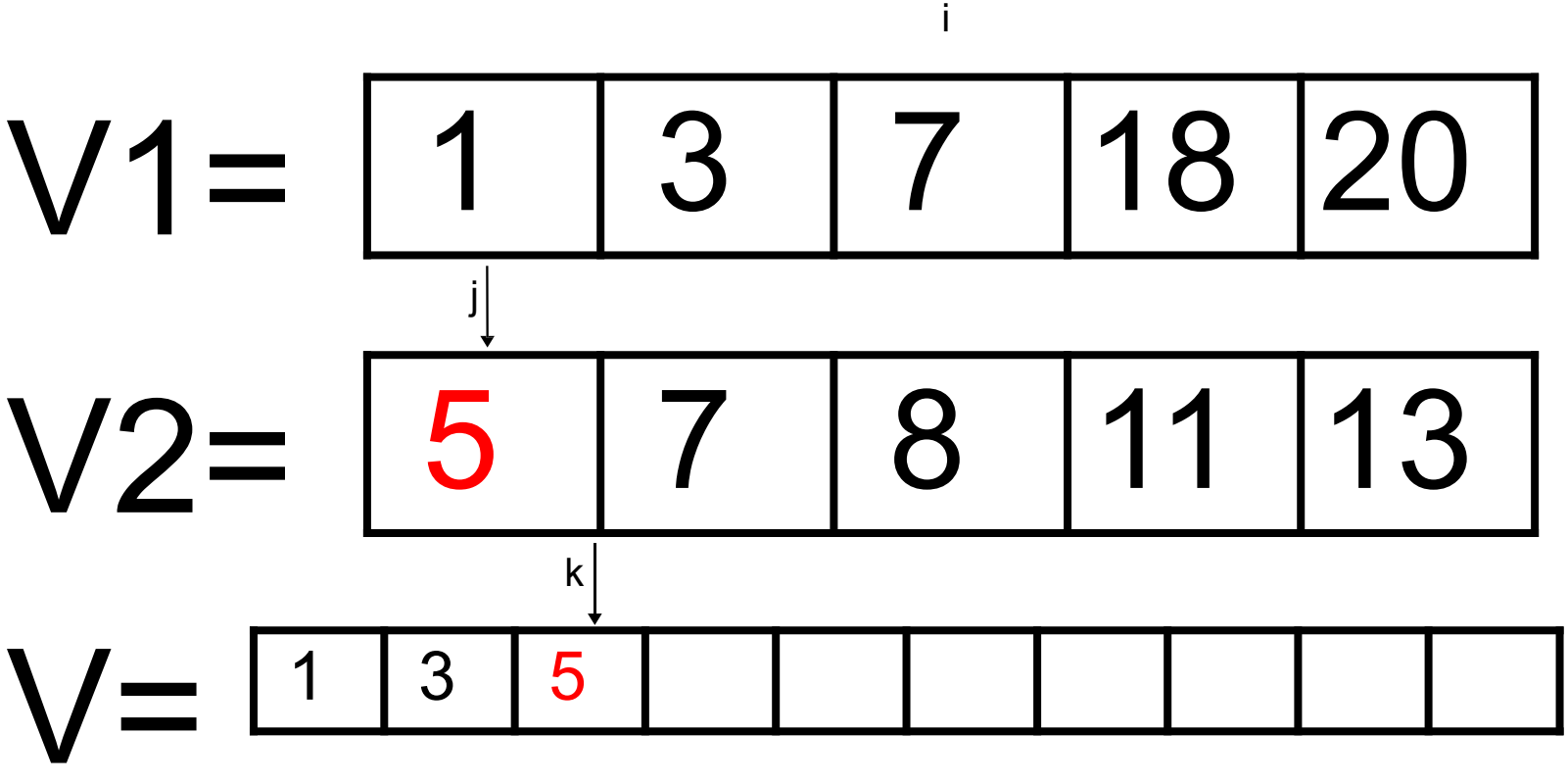
Merge de dois vetores ordenados

Código em Python



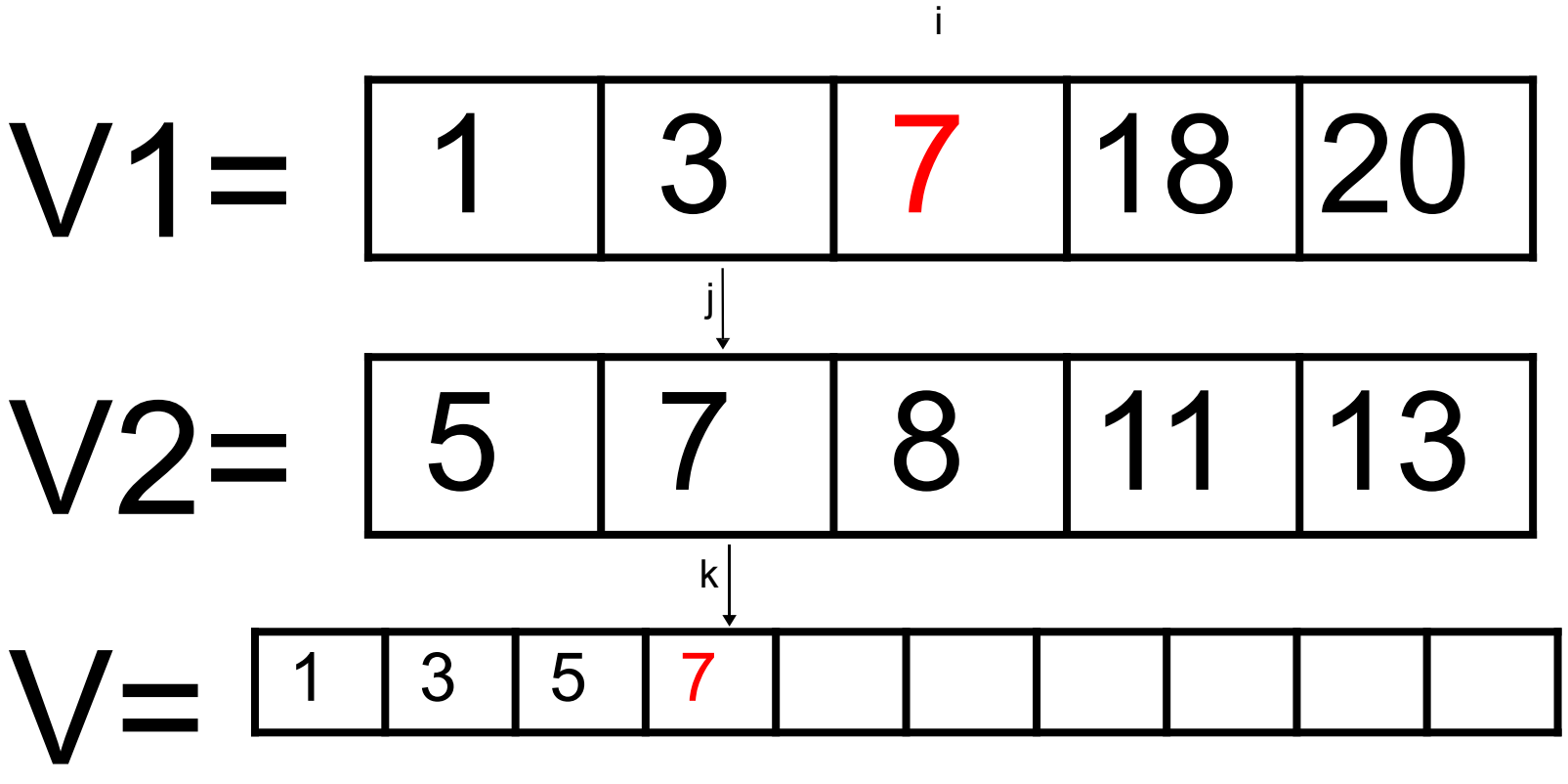
Merge de dois vetores ordenados

Código em Python



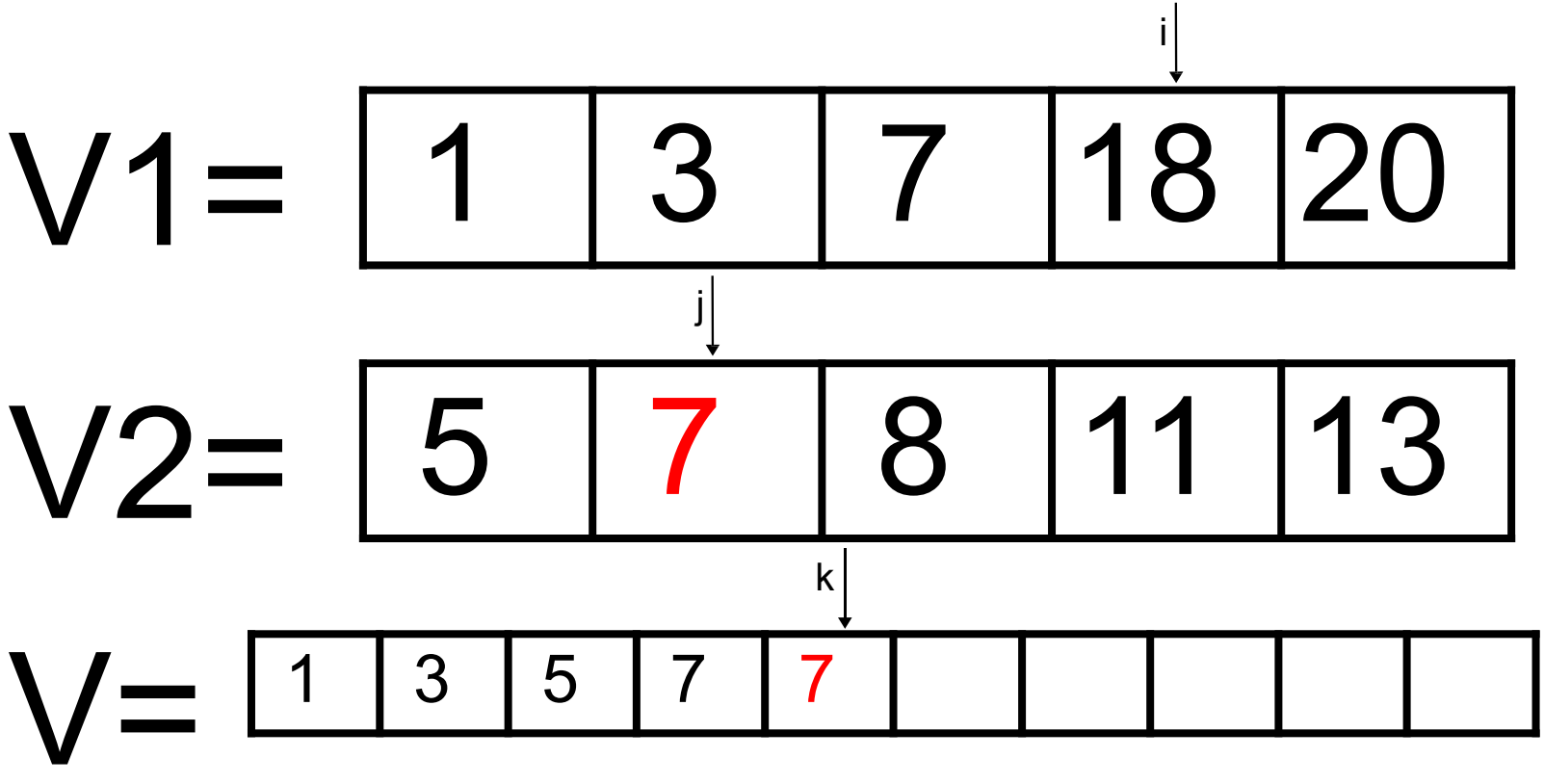
Merge de dois vetores ordenados

Código em Python



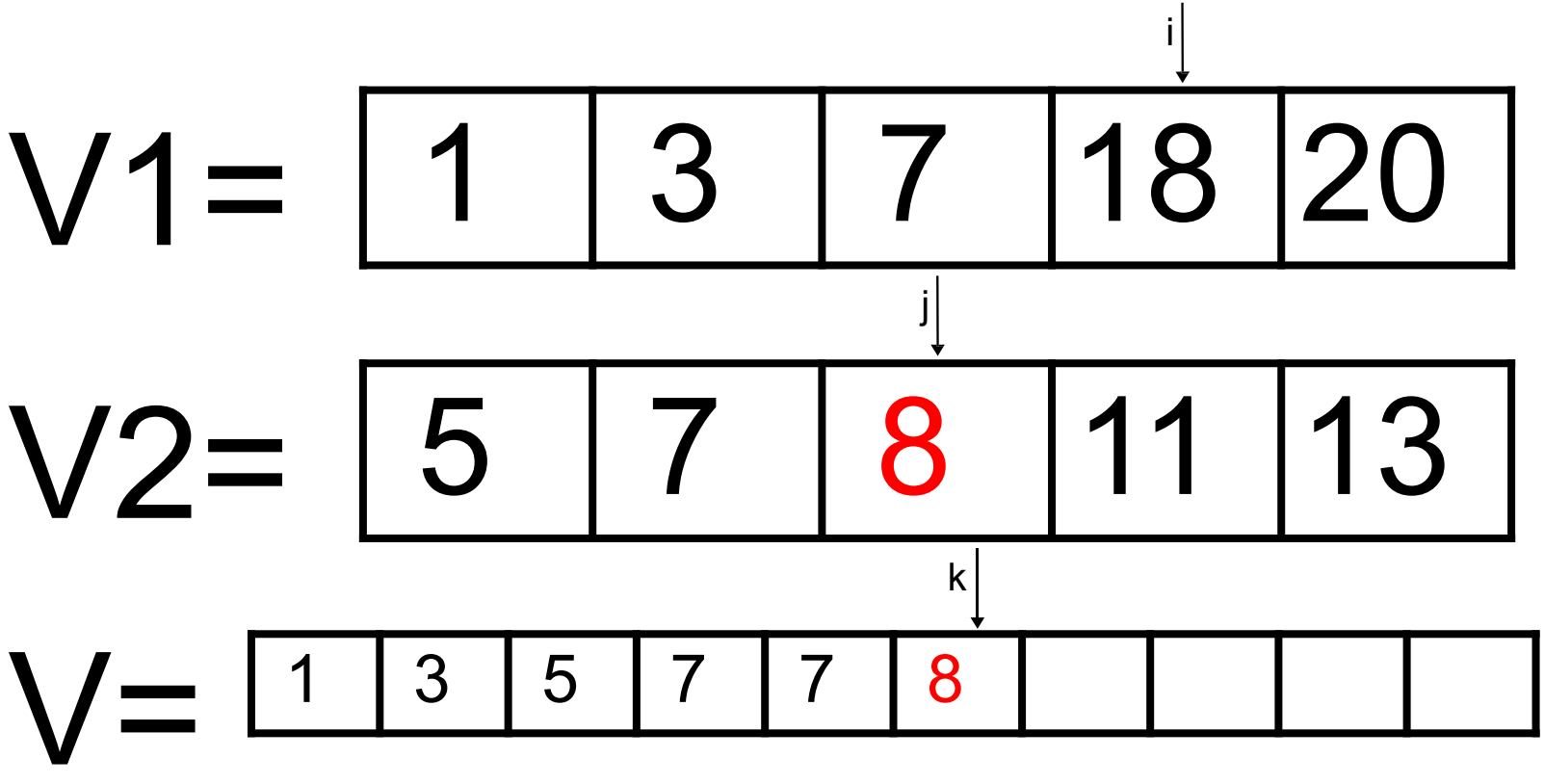
Merge de dois vetores ordenados

Código em Python



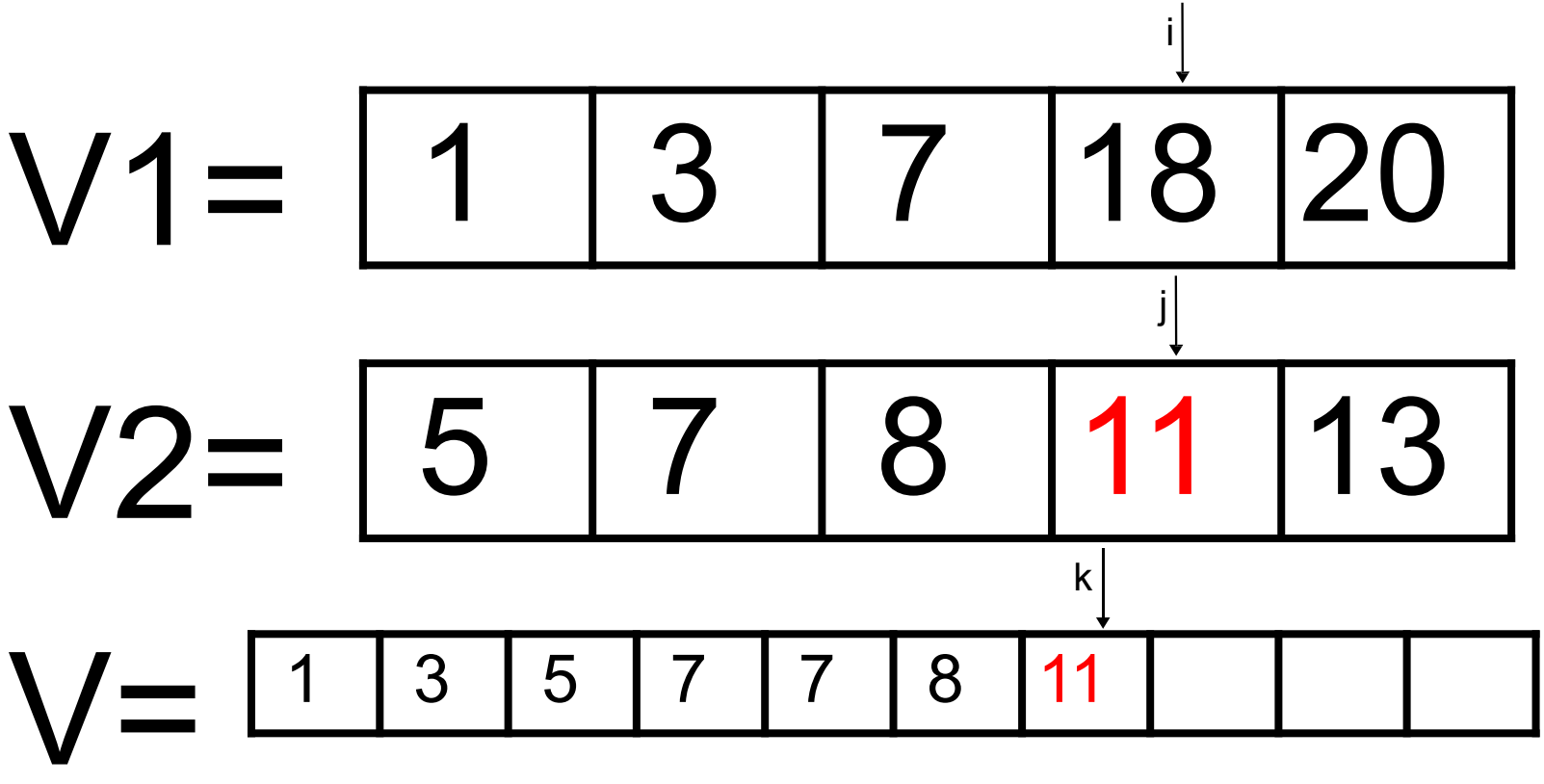
Merge de dois vetores ordenados

Código em Python



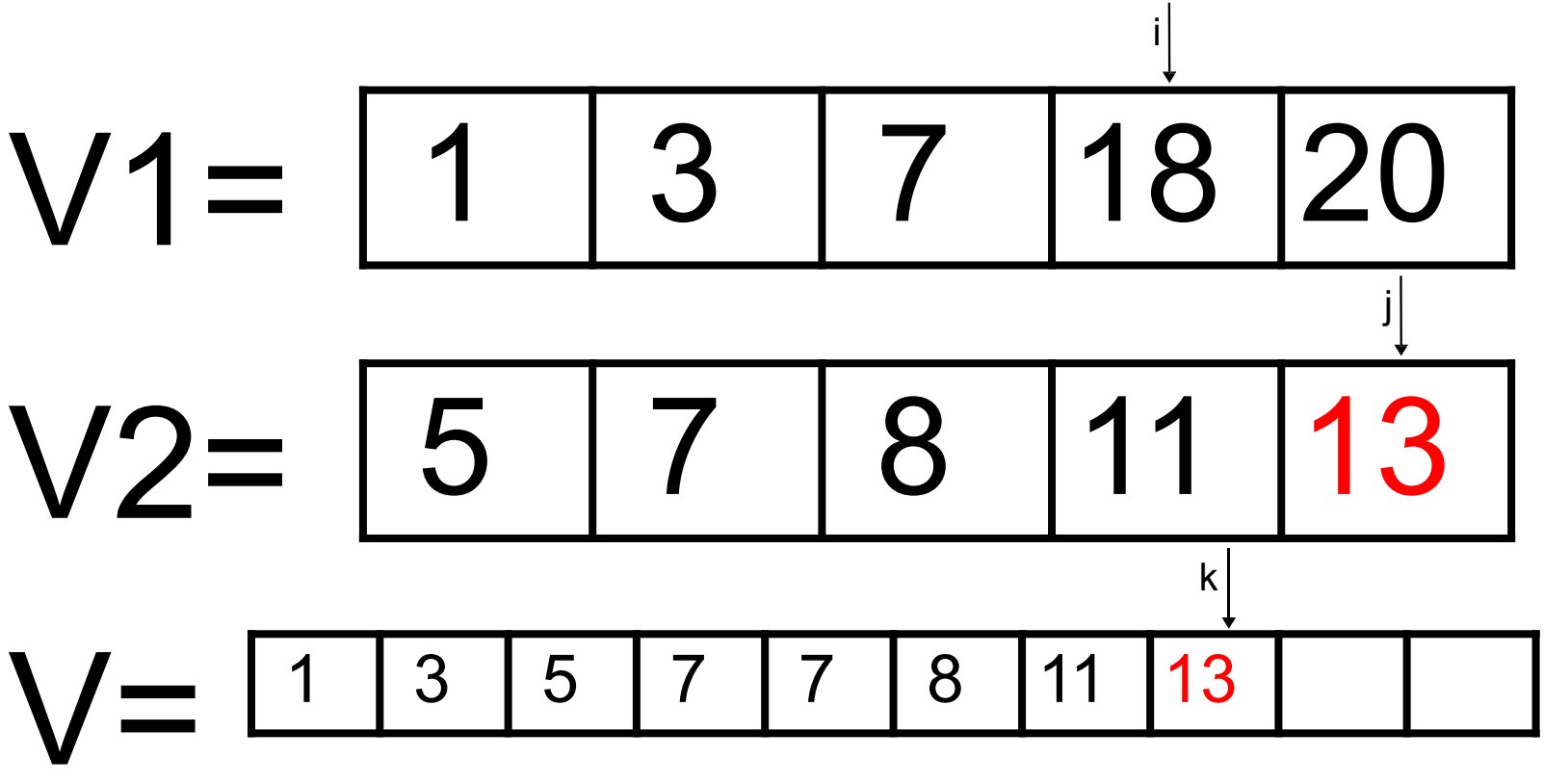
Merge de dois vetores ordenados

Código em Python



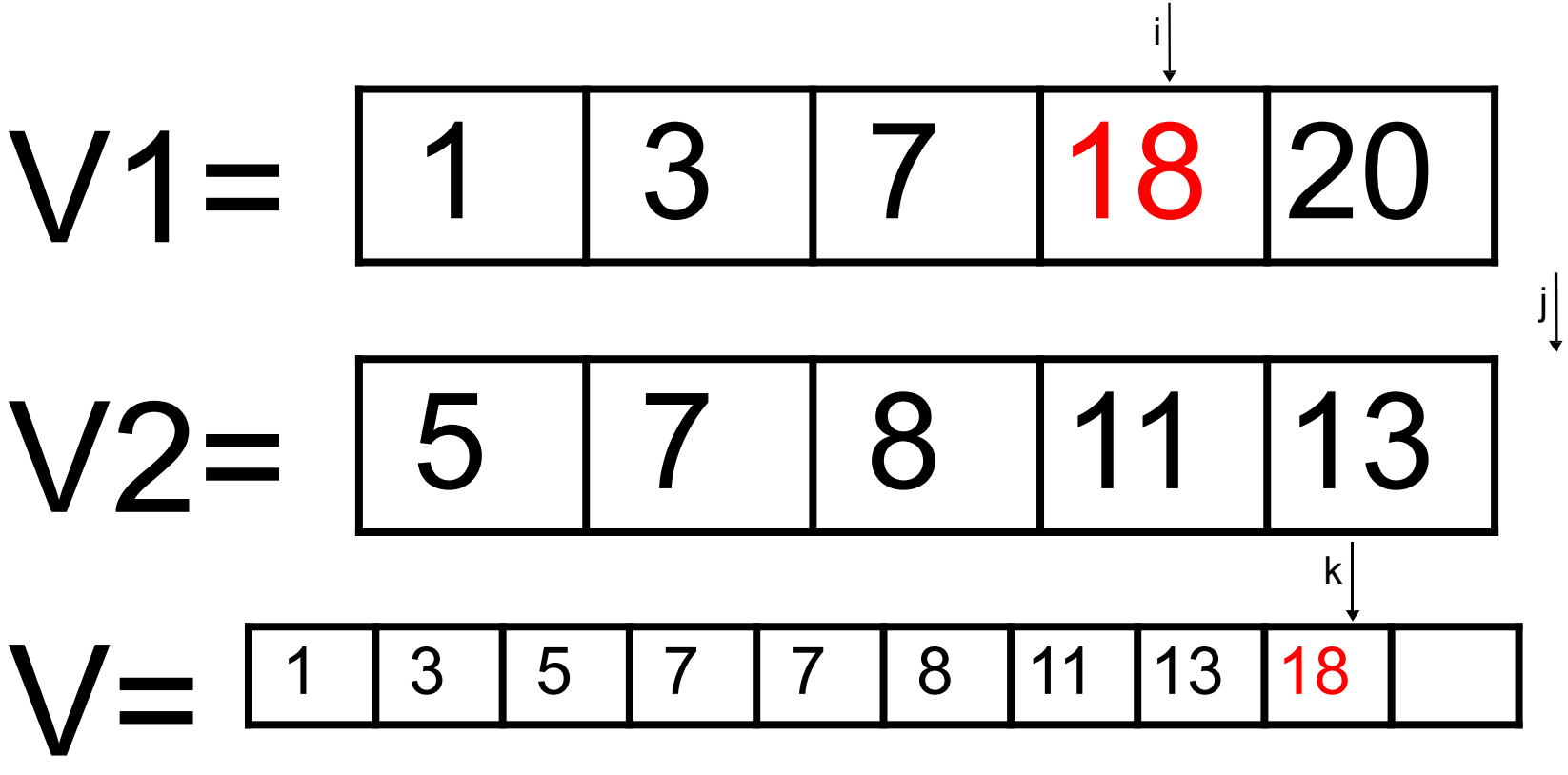
Merge de dois vetores ordenados

Código em Python



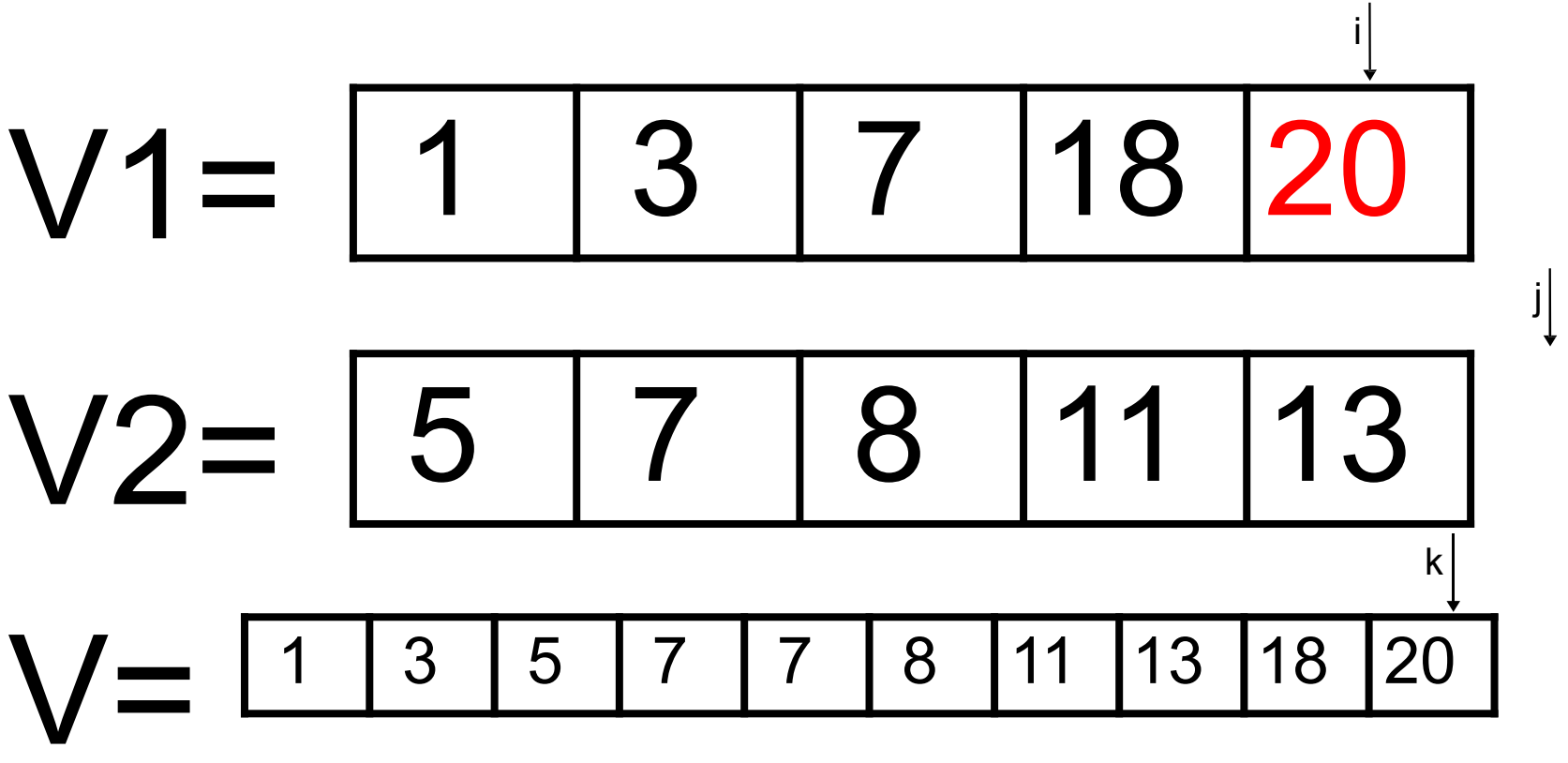
Merge de dois vetores ordenados

Código em Python



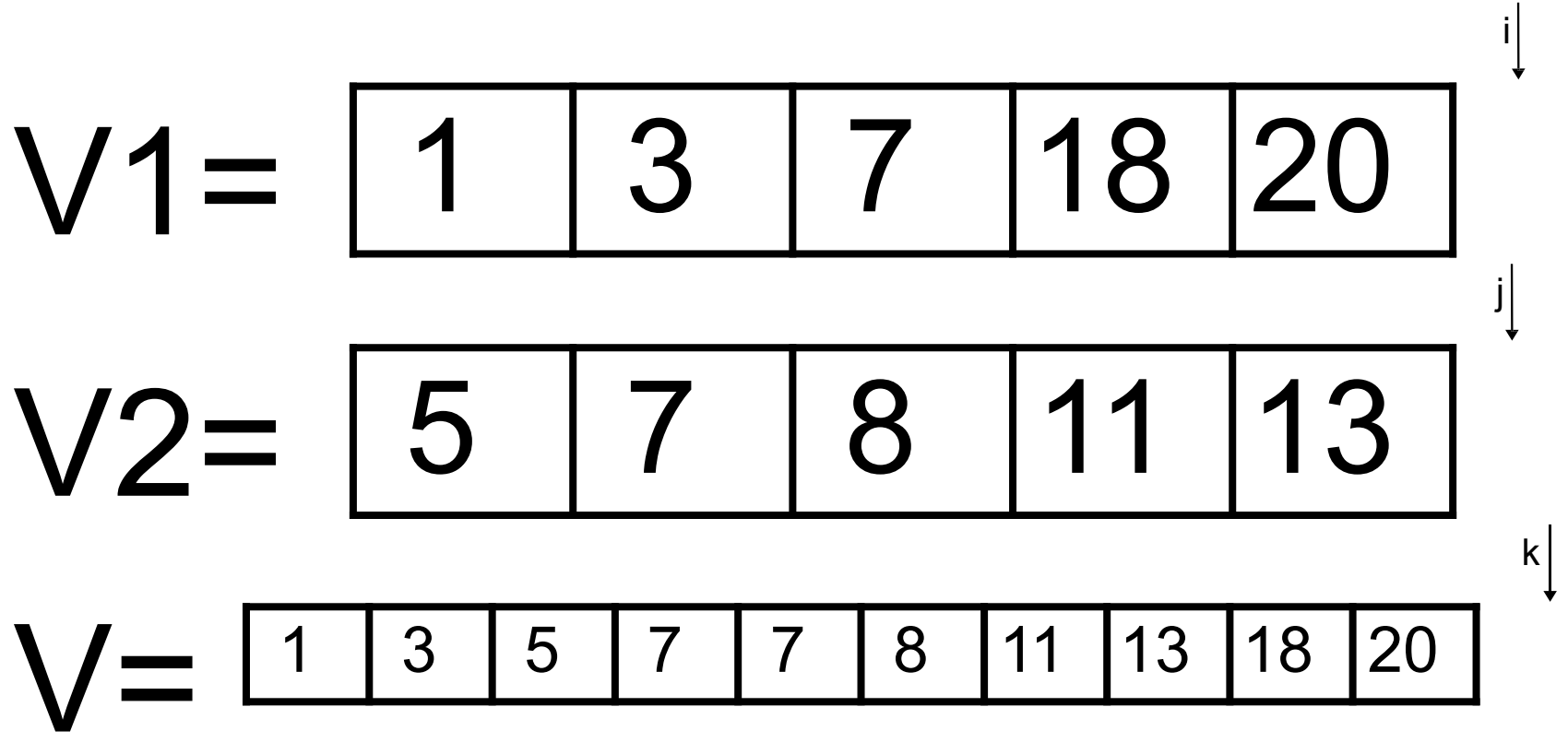
Merge de dois vetores ordenados

Código em Python



Merge de dois vetores ordenados

Código em Python



Merge de dois vetores ordenados

Código em Python

- Dados dois vetores *nums1* e *nums2*, já ordenados, o **merge** cria um outro vetor *mergedArr* ordenado com os elementos de *nums1* e *nums2*

```
mergedArr = []

i, j = 0, 0

while i<len(nums1) and j<lens(nums2):

   if nums1[i]<nums2[j]:

      mergedArr.append(nums1[i])

      i+=1

   else:

      mergedArr.append(nums2[j])

      j+=1

while i<len(nums1):

   mergedArr.append(nums1[i])

   i+=1

while j<len(nums2):

   mergedArr.append(nums2[j])

   j+=1
```

Mergesort

Prática no LeetCode - https://leetcode.com/problems/merge-sorted-array/

88. Merge Sorted Array

Solved

Easy | Topics | Companies | Hint

You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

Merge `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to `0` and should be ignored. `nums2` has a length of `n`.

Example 1:

```
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
Explanation: The arrays we are merging are [1,2,3] and [2,5,6].
The result of the merge is [1,2,2,3,5,6] with the underlined elements coming
from nums1.
```

Example 2:

```
Input: nums1 = [1], m = 1, nums2 = [], n = 0
Output: [1]
Explanation: The arrays we are merging are [1] and [].
The result of the merge is [1].
```

ada

Mergesort

Prática no LeetCode - https://leetcode.com/problems/merge-sorted-array/

88 - Merge Sorted Array

```
def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
    mergedArr = []
    i = 0
    j = 0
    while i<m and j<n:
        if nums1[i]<nums2[j]:
            mergedArr.append(nums1[i])
            i+=1
        else:
            mergedArr.append(nums2[j])
            j+=1
    while i<m:
        mergedArr.append(nums1[i])
        i+=1
    while j<n:
        mergedArr.append(nums2[j])
        j+=1
    for i in range(len(mergedArr)):
        nums1[i] = mergedArr[i]
```

Prática no LeetCode - https://leetcode.com/problems/merge-sorted-array/

88 - Merge Sorted Array

O enunciado pede para resolver *in loco*, sem usar espaço extra de memória.

Dica: pense em uma solução que preencha o vetor *nums1* de trás para frente!

Mergesort

Prática no LeetCode - https://leetcode.com/problems/merge-sorted-array/

88 - Merge Sorted Array

```
def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) ->
None:
    if n == 0: return
    end_idx = len(nums1)-1
    while n > 0 and m > 0 :
        if nums2[n-1] >= nums1[m-1]:
            nums1[end_idx] = nums2[n-1]
            n-=1
        else:
            nums1[end_idx] = nums1[m-1]
            m-=1
        end_idx-=1
    while n > 0:
        nums1[end_idx] = nums2[n-1]
        n-=1
        end_idx-=1
```

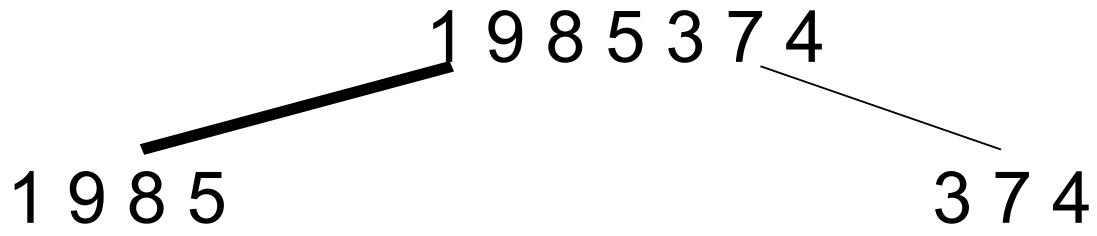
Mergesort

Funcionamento

1 9 8 5 3 7 4

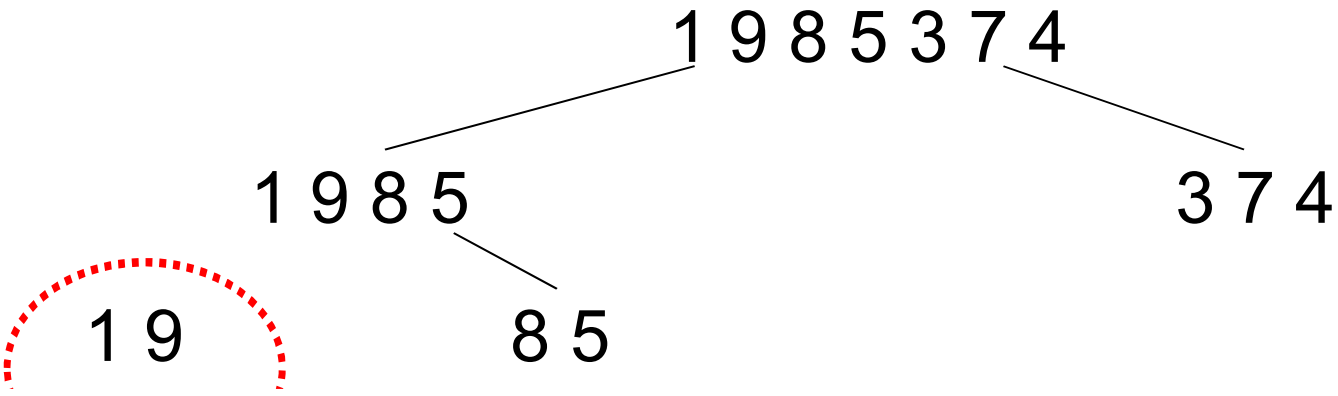
Mergesort

Funcionamento



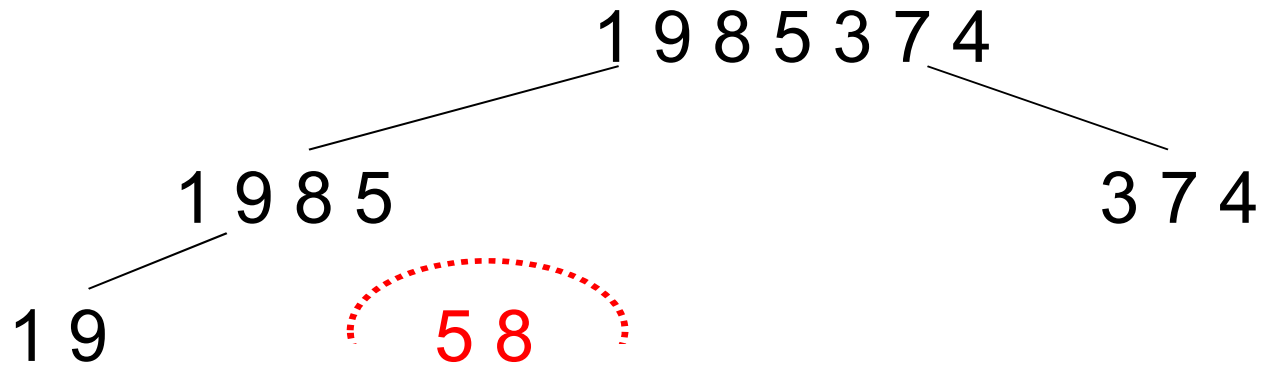
Mergesort

Funcionamento



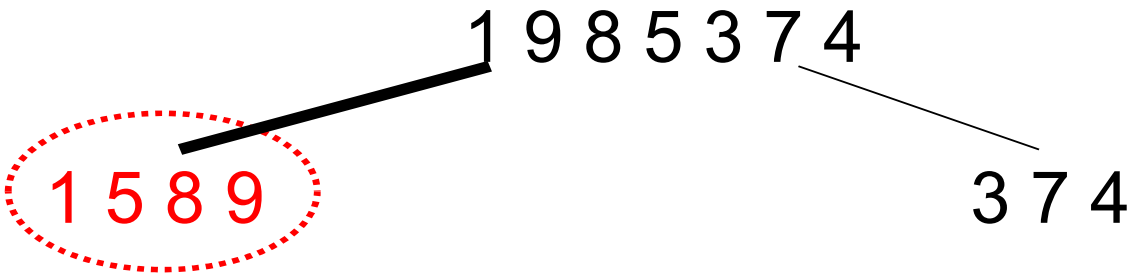
Mergesort

Funcionamento



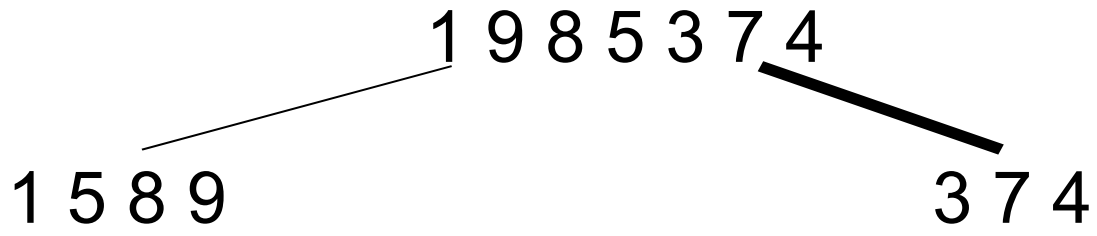
Mergesort

Funcionamento



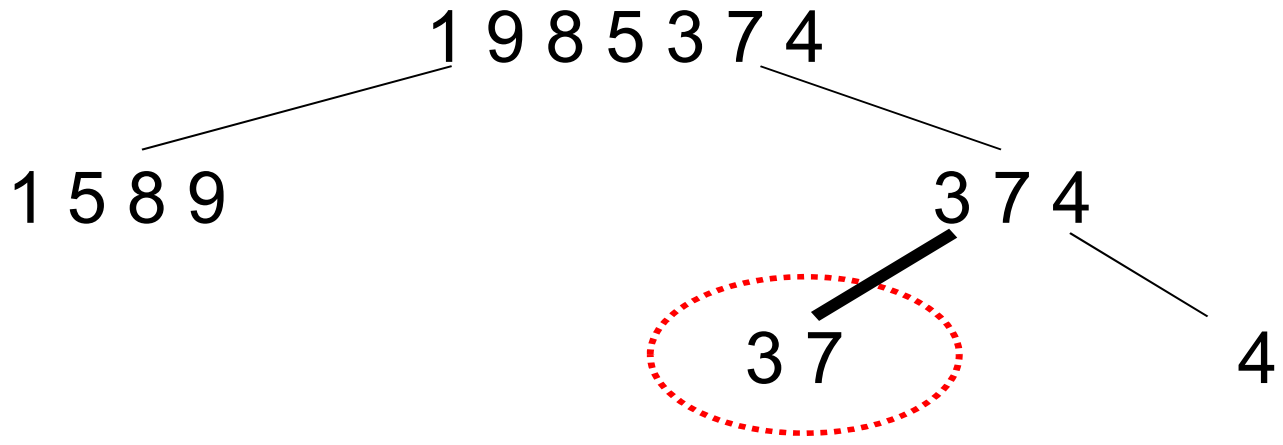
Mergesort

Funcionamento



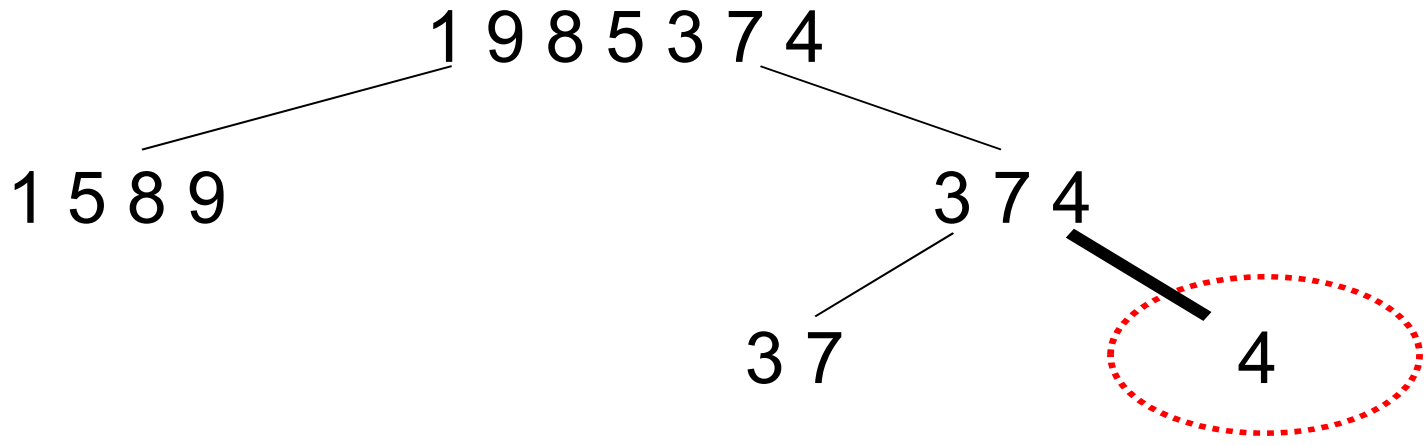
Mergesort

Funcionamento



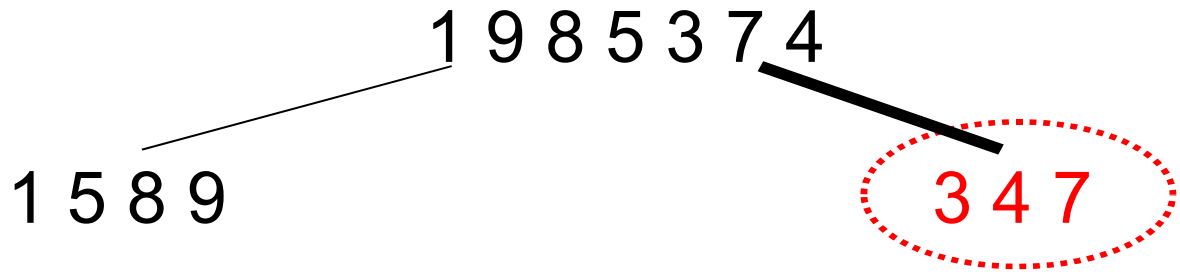
Mergesort

Funcionamento



Mergesort

Funcionamento

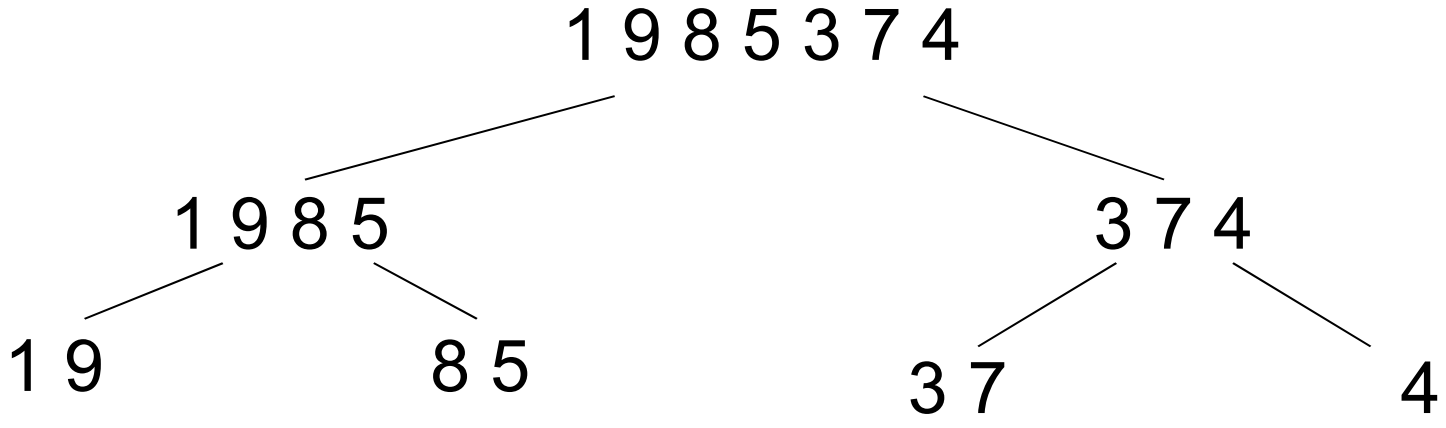


Mergesort

Funcionamento

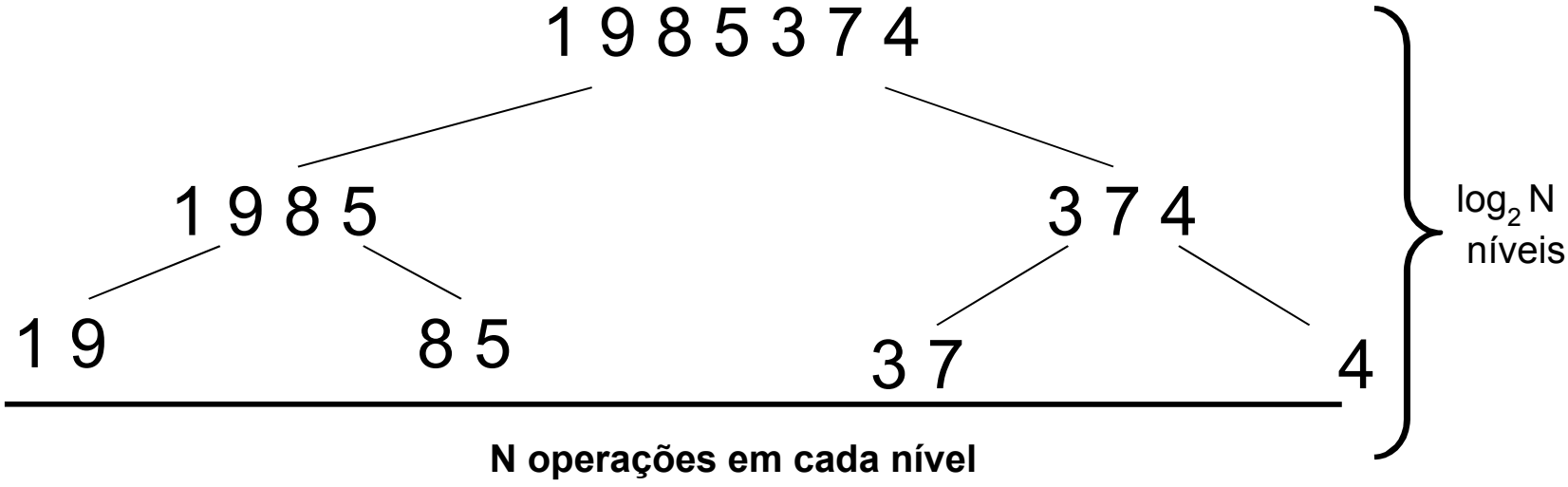
Mergesort

Quantidade de Operações



Mergesort

Quantidade de Operações



Total de operações:

$$N^* \log_2 N$$

Quicksort

Quicksort

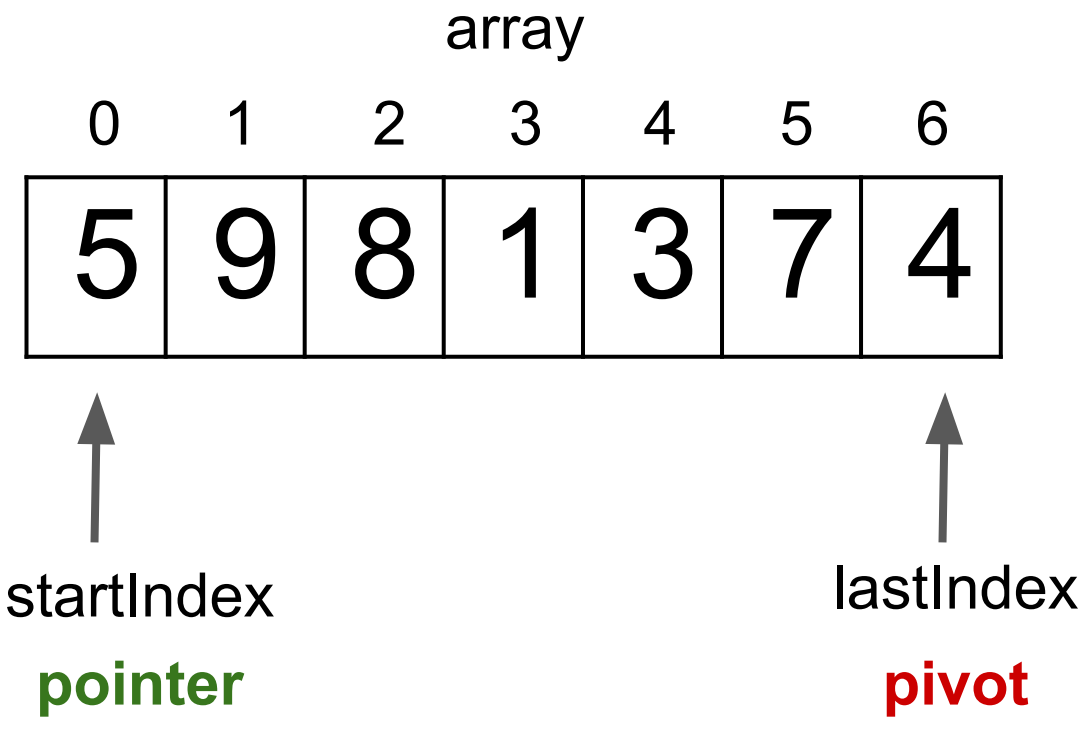
Conceito

- Ordena o vetor particionando recursivamente.
- Usa a técnica "***dividir para conquistar***":
    - A cada iteração, localiza a posição final de um elemento aleatório (pivô) e **subdivide o vetor em duas partes** para prosseguir a ordenação.
- O *Quick Sort* pode ser implementado com duas funções:
    - partition(Array, startIndex, lastIndex)
    - quick_sort(Array, startIndex, lastIndex)

Quicksort

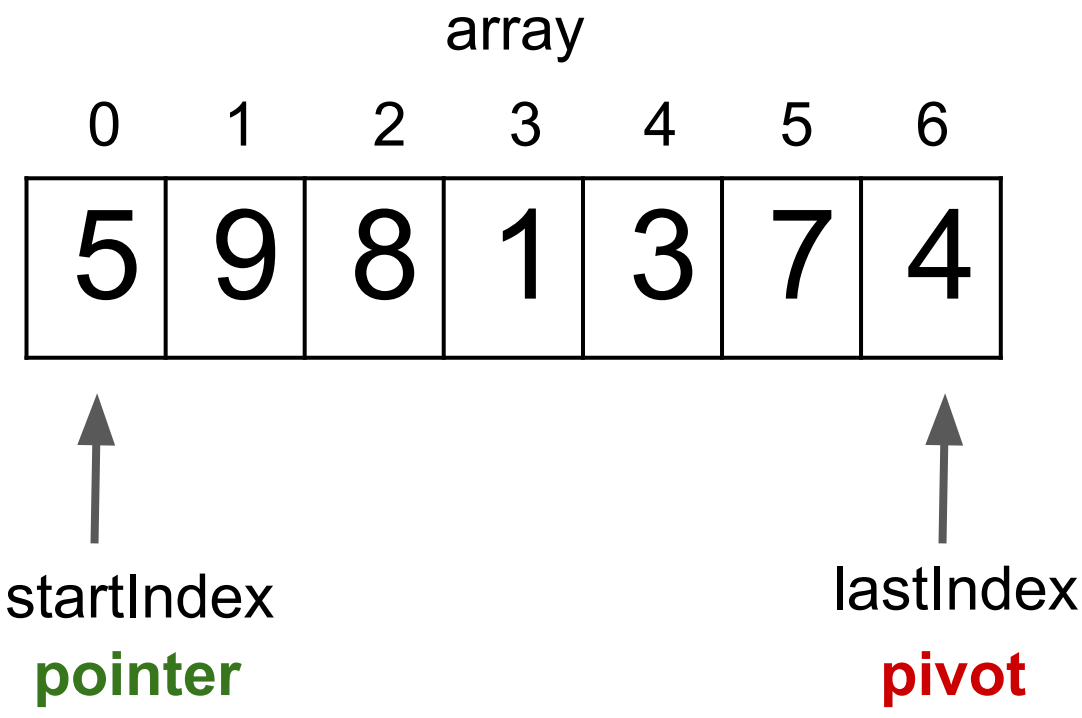
Funcionamento

int pivot = Partition(array, startIndex, lastIndex);



Quicksort

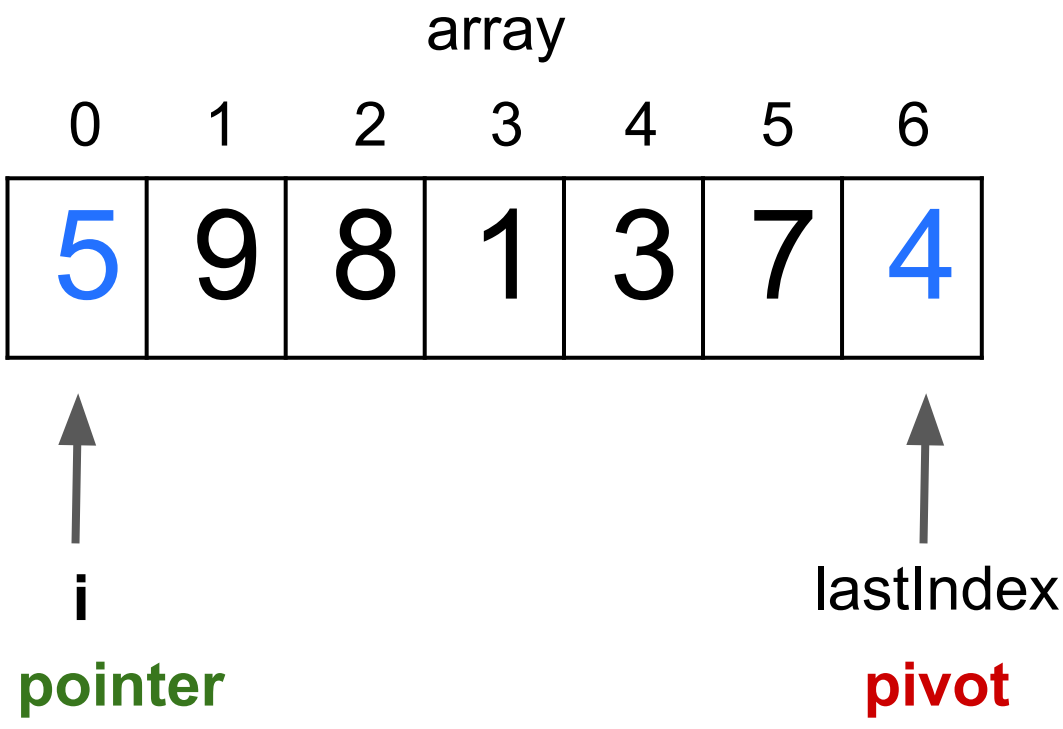
Funcionamento




```
for (i = startIndex; i < lastIndex; i++)
        if (array[i] <= pivot) {
            temp = array[i];
            array[i] = array[pointer];
            array[pointer] = temp;
            pointer++; }
```

Quicksort

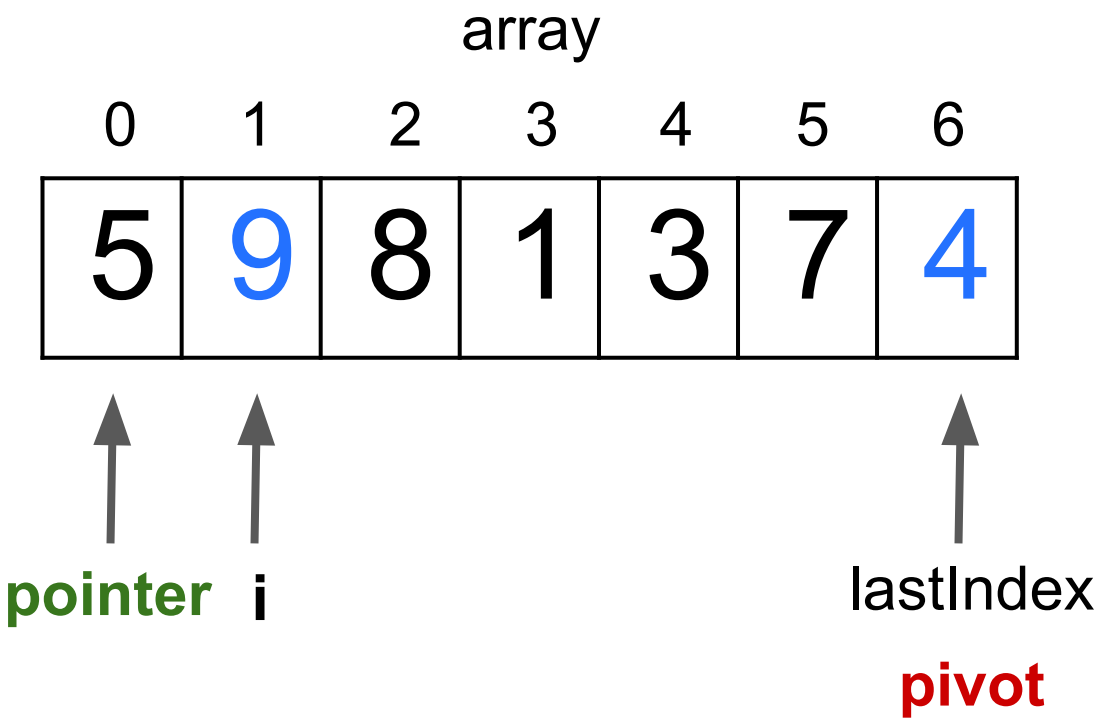
Funcionamento




```
for (i = startIndex; i < lastIndex; i++)
        if (array[i] <= pivot) {
                temp = array[i];
                array[i] = array[pointer];
                array[pointer] = temp;
                pointer++; }
```

Quicksort

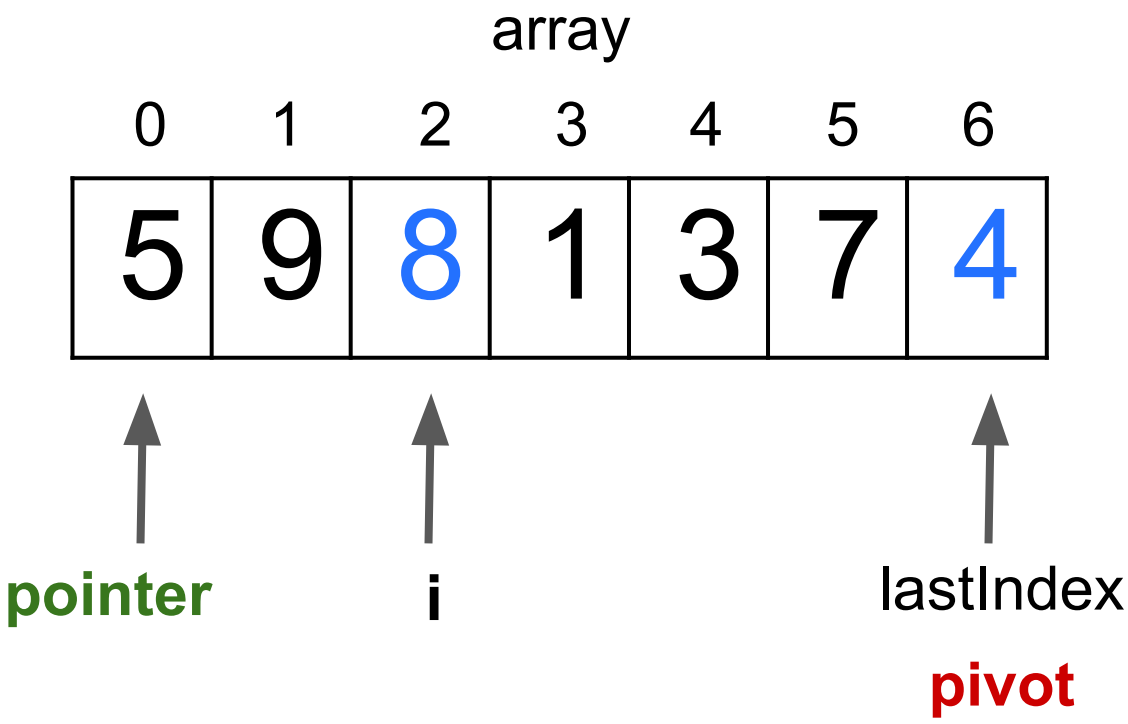
Funcionamento




```
for (i = startIndex; i < lastIndex; i++)
        if (array[i] <= pivot) {
            temp = array[i];
            array[i] = array[pointer];
            array[pointer] = temp;
            pointer++; }
```

Quicksort

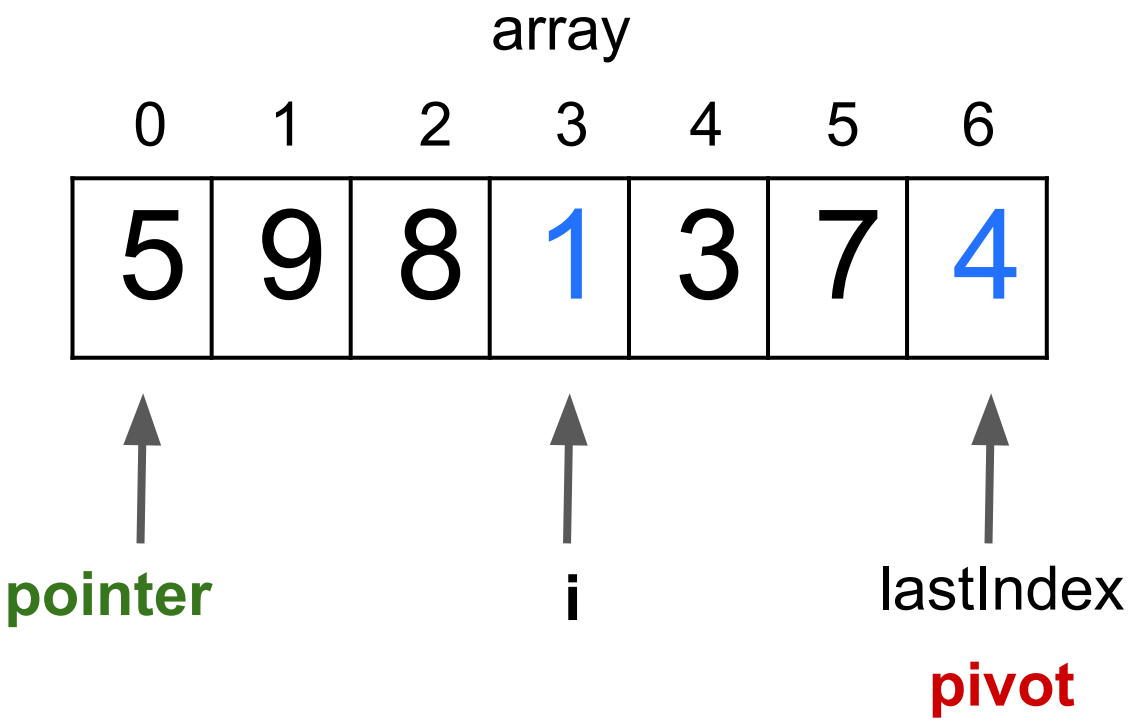
Funcionamento




```
for (i = startIndex; i < lastIndex; i++)
        if (array[i] <= pivot) {
            temp = array[i];
            array[i] = array[pointer];
            array[pointer] = temp;
            pointer++; }
```

Quicksort

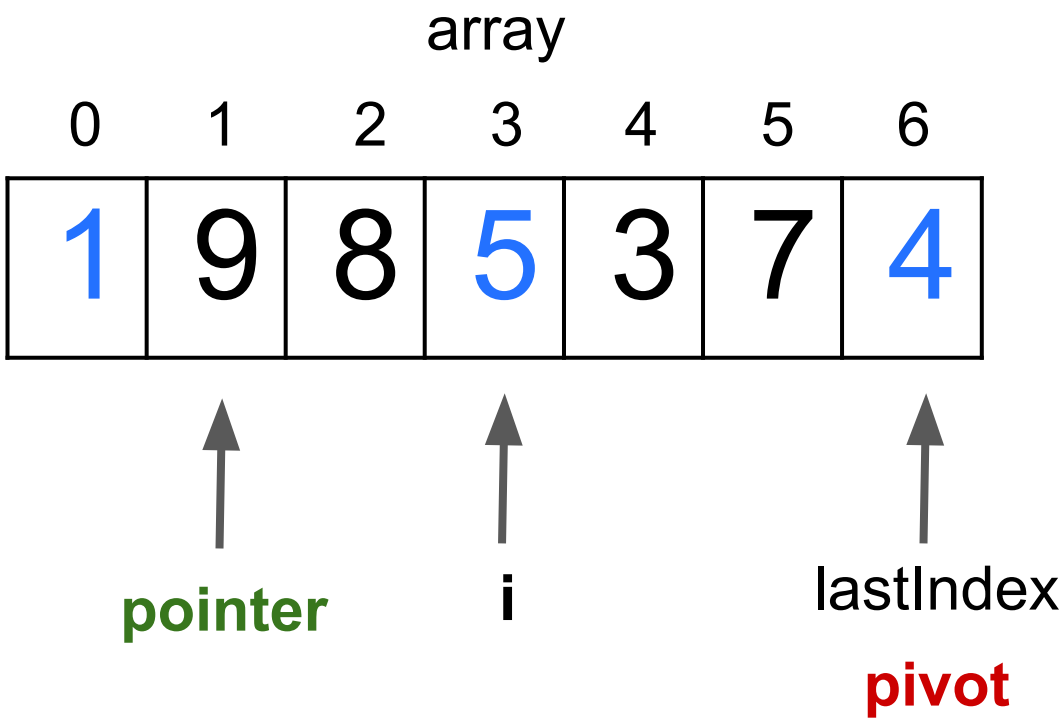
Funcionamento




```
for (i = startIndex; i < lastIndex; i++)
        if (array[i] <= pivot) {
            temp = array[i];
            array[i] = array[pointer];
            array[pointer] = temp;
            pointer++; }
```

Quicksort

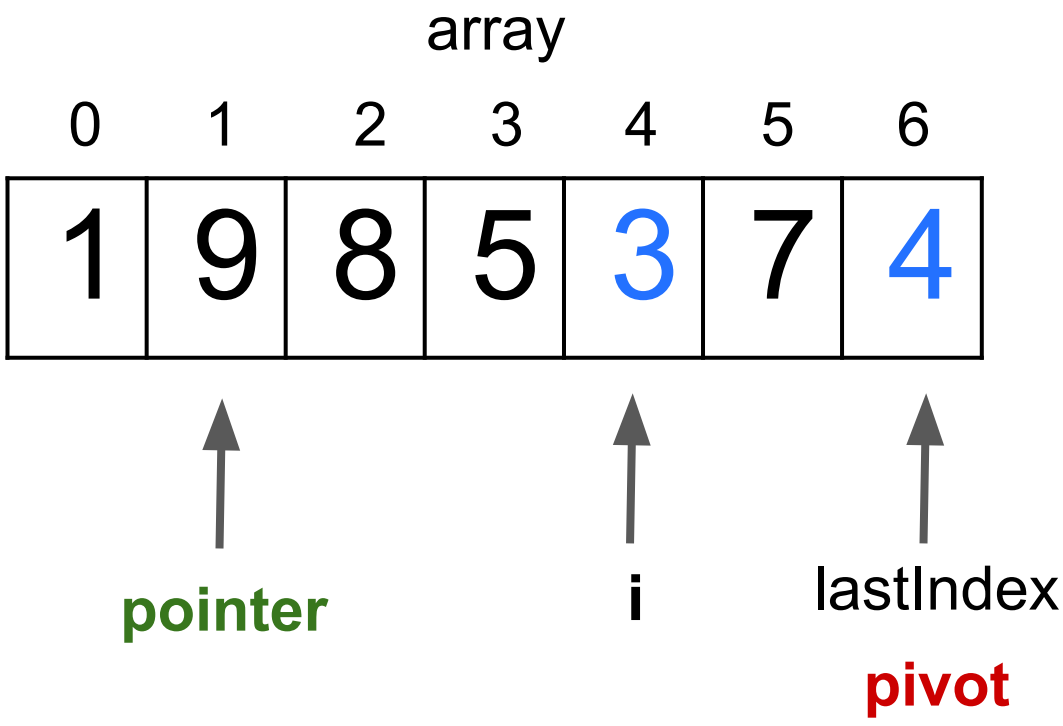
Funcionamento




```
for (i = startIndex; i < lastIndex; i++)
        if (array[i] <= pivot) {
                temp = array[i];
                array[i] = array[pointer];
                array[pointer] = temp;
                pointer++; }
```

Quicksort

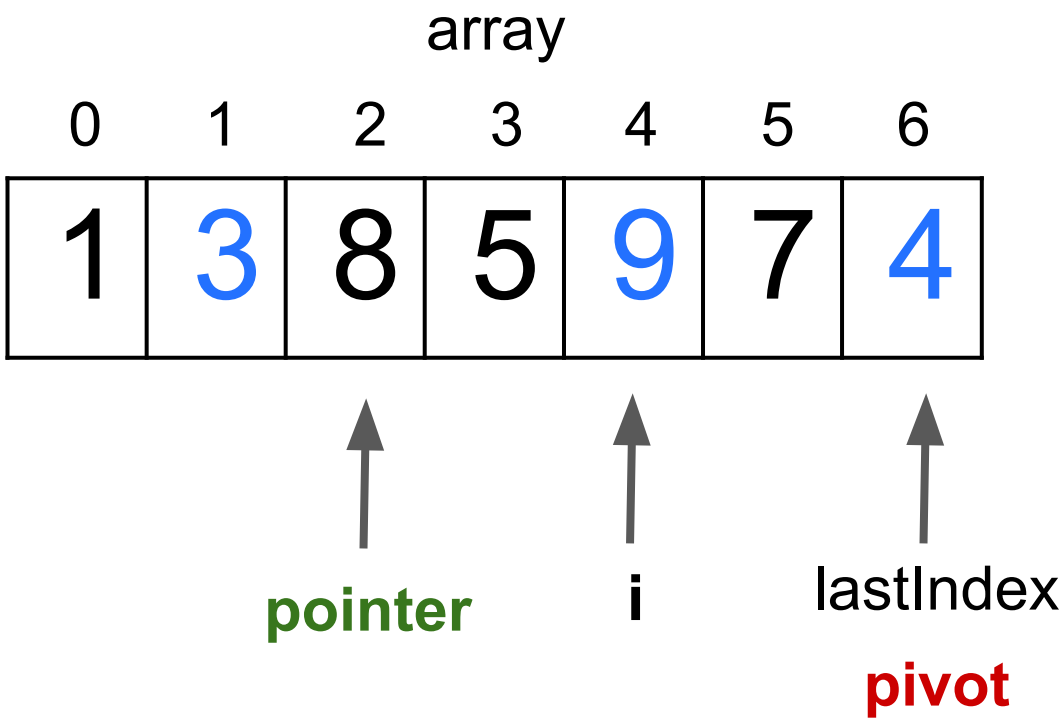
Funcionamento




```
for (i = startIndex; i < lastIndex; i++)
        if (array[i] <= pivot) {
            temp = array[i];
            array[i] = array[pointer];
            array[pointer] = temp;
            pointer++; }
```

Quicksort

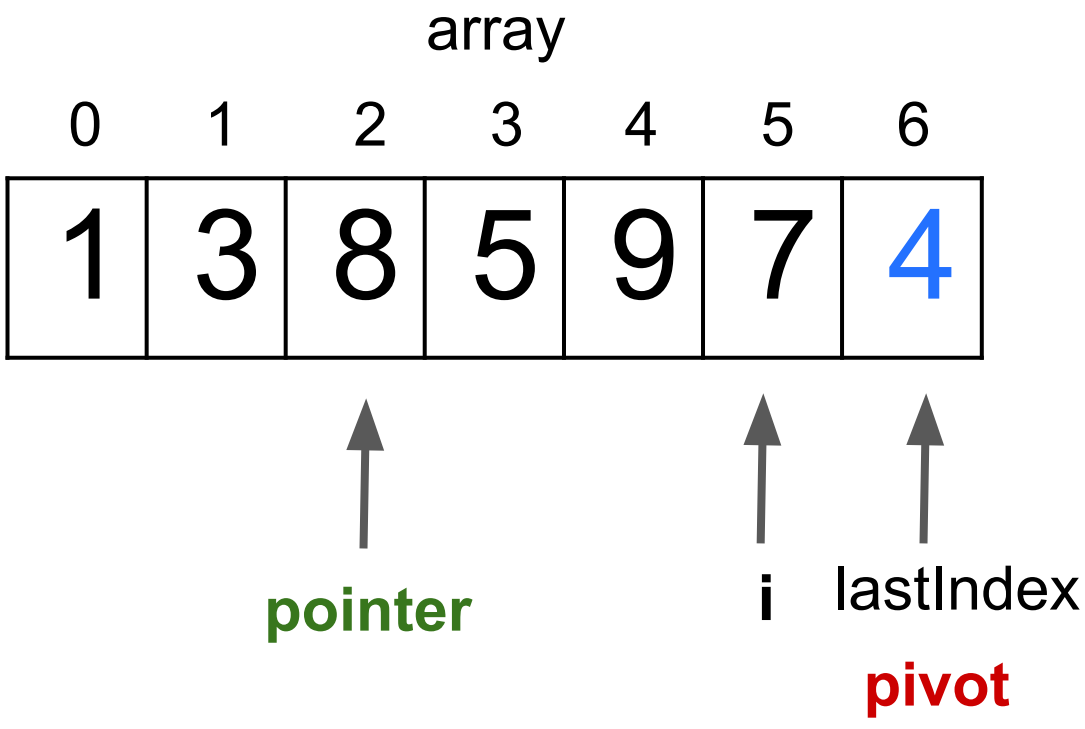
Funcionamento




```
for (i = startIndex; i < lastIndex; i++)
        if (array[i] <= pivot) {
              temp = array[i];
              array[i] = array[pointer];
              array[pointer] = temp;
              pointer++; }
```

Quicksort

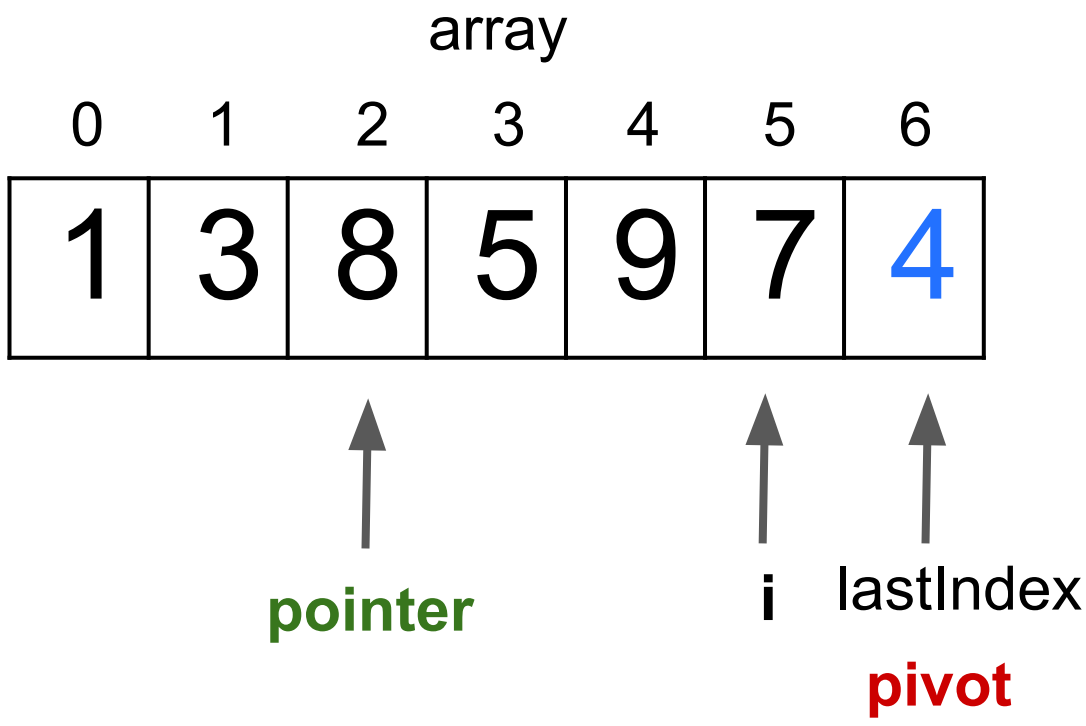
Funcionamento




```
for (i = startIndex; i < lastIndex; i++)
        if (array[i] <= pivot) {
              temp = array[i];
              array[i] = array[pointer];
              array[pointer] = temp;
              pointer++; }
```

Quicksort

Funcionamento

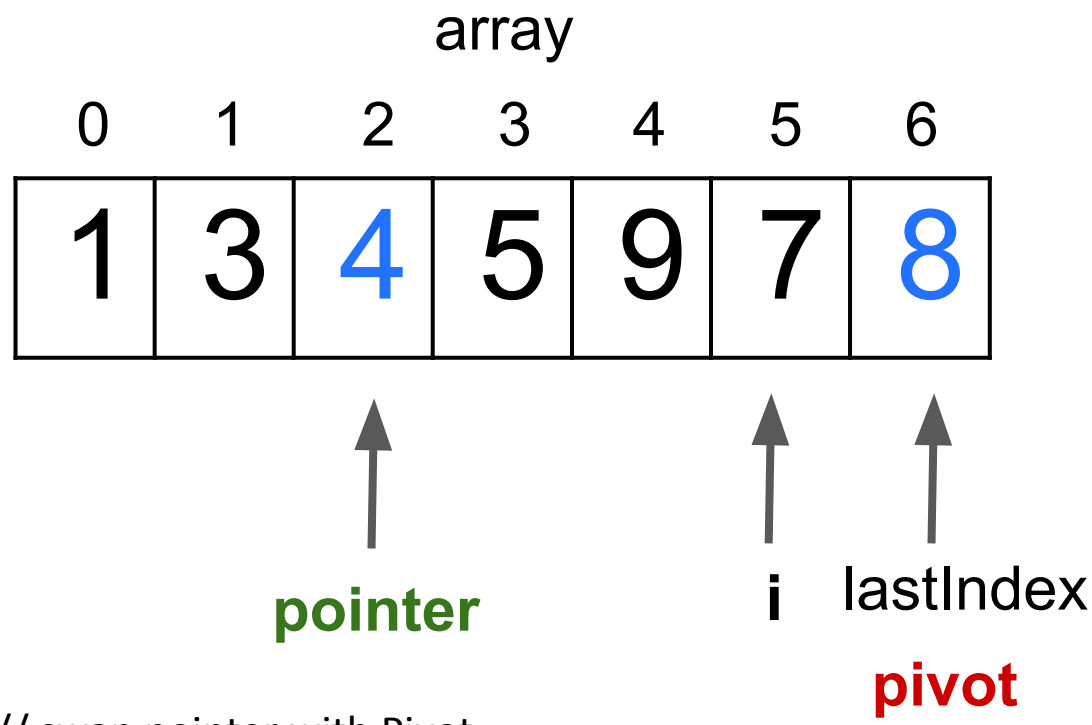



```
// swap pointer with Pivot
      temp = array[pointer];
      array[pointer] = array[lastIndex];
      array[lastIndex] = temp;

 // returning the index of the pivot
      return pointer;
```

Quicksort

Funcionamento

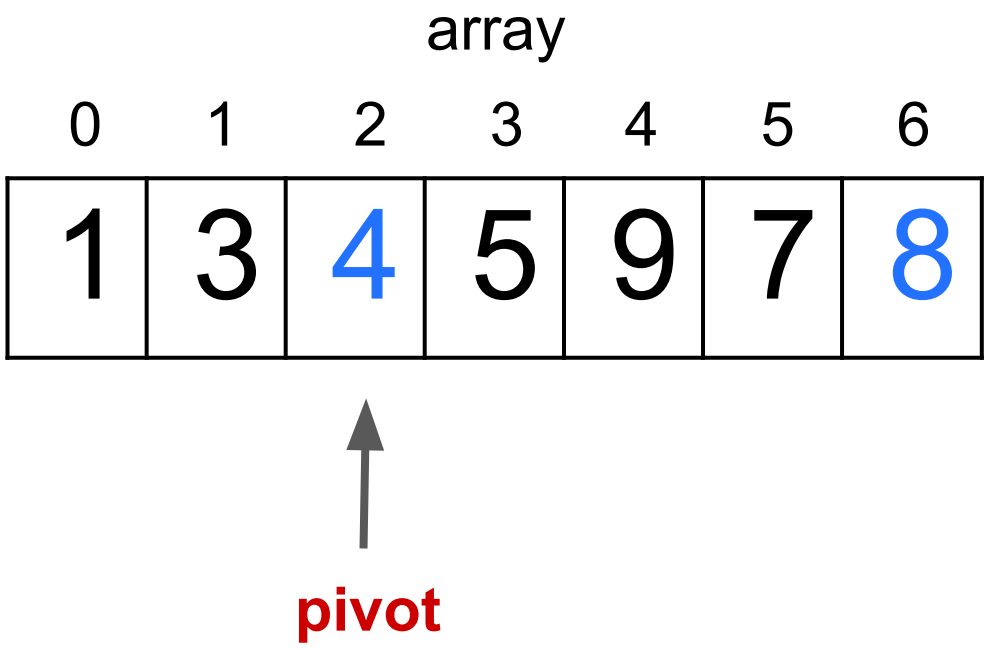



```
// swap pointer with Pivot
      temp = array[pointer];
      array[pointer] = array[lastIndex];
      array[lastIndex] = temp;

 // returning the index of the pivot
      return pointer; // índice 2 será o próximo pivô
```

Quicksort

Funcionamento



```
// this will contain the elements that are less than pivot
    QuickSort(array, startIndex, pivot - 1);

// this will contain the elements that are greater than pivot
    QuickSort(array, pivot + 1, lastIndex);
```

Quicksort

Funcionamento

Vamos implementar o Quick Sort ?

Heapsort

Heapsort

Conceito

- Ordenação baseada na estrutura de dados *heap*
- Heap é a fila de prioridades
- O heap tem o vetor como a representação de uma árvore binária completa, onde cada nó é um elemento do vetor, com a raiz na primeira posição

Heapsort

Primitivas

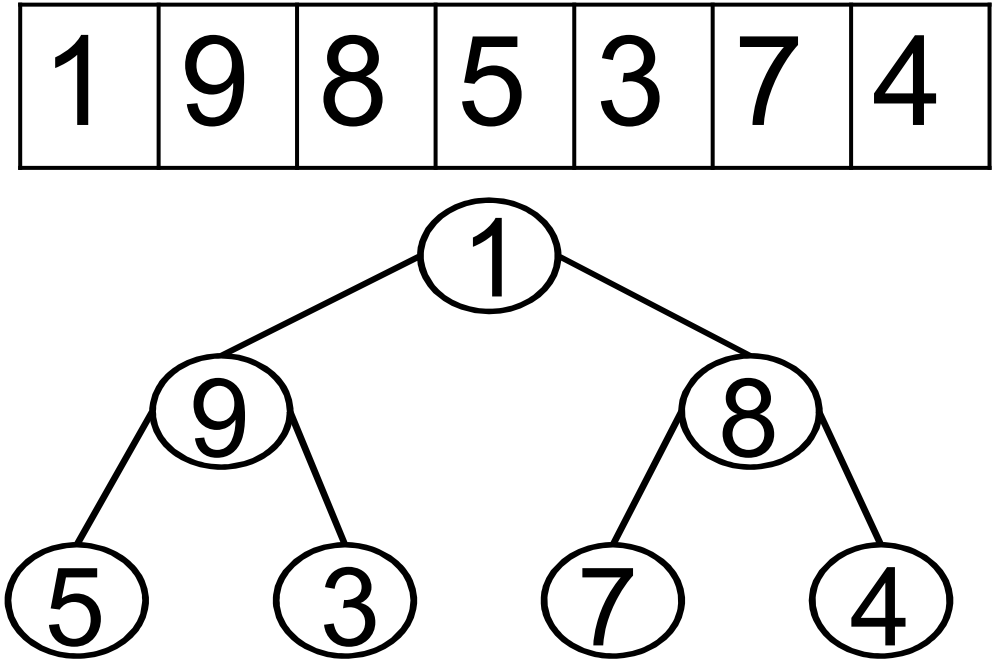
- length(V)
  - número de elementos do vetor V
- heap-size(V)
  - número de elementos do heap armazenados no vetor V

- parent(i)
  - return i/2
- left(i)
  - return 2i
- right(i)
  - return 2i+1

LEMBRETE: Em Lua, os vetores são *indexados a partir de 1*. Isso exige que esse código fique diferente em sua linguagem de programação se os *índices começarem de 0*.

Heapsort

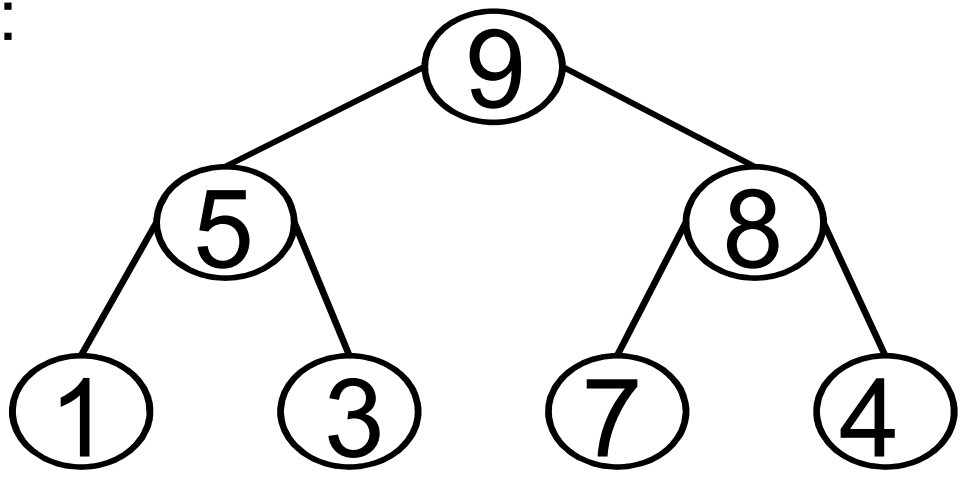
Funcionamento



Heapsort

Propriedade do Heap

- Para cada nó i que não seja a raiz, então:
  - V[parent(i)] >= V[i]
- Exemplo:



Heapsort

Primitiva Heapify

- Função para manter a propriedade da heap

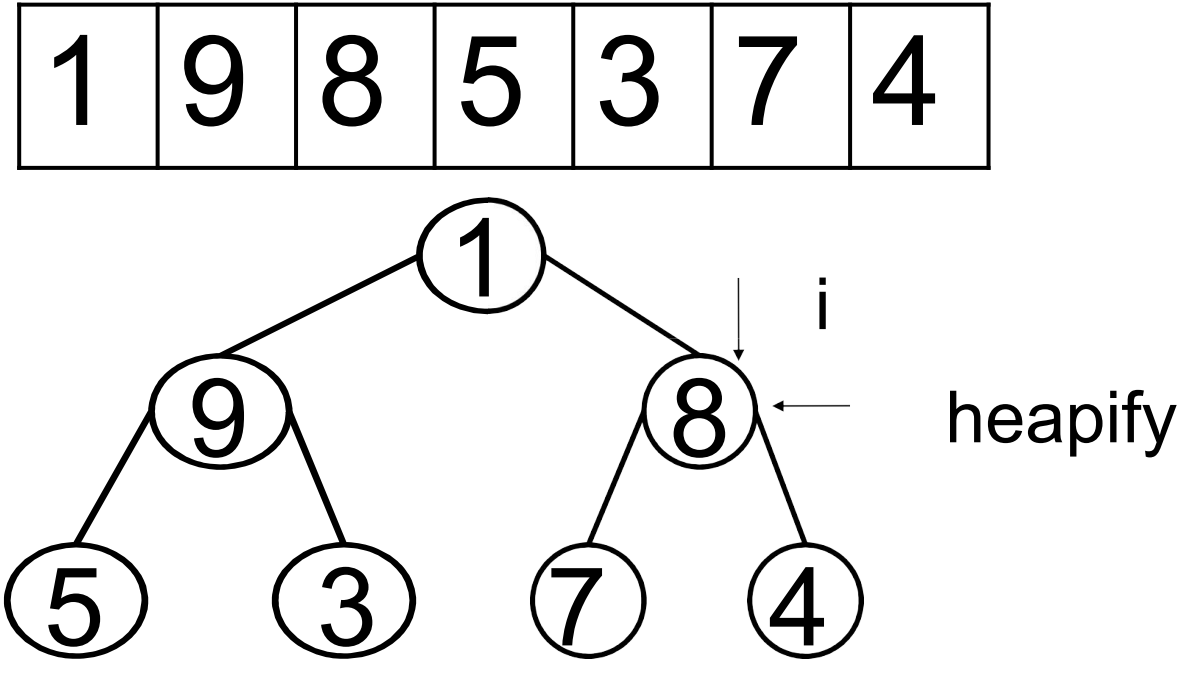
```
function heapify(V, i)
   l = left(i)
   r = right(i)
   largest = i
    if l<=heap-size(V)  and V[l]>V[i]  then
       largest = l
   end
   if r<=heap-size(V) and V[r]>V[largest] then
      largest = r
   end
   if largest!=i:
      V[i], V[largest] = V[largest], V[i]
      heapify(V, largest)
   end
```

- Primitiva que constrói uma *heap* a partir de um vetor de N elementos

```
function build_heap(V)
    heap-size(V) = length(V) -- #V
    for i=length(V)/2,1,-1 do
        heapify(V,i)
    end
end
```

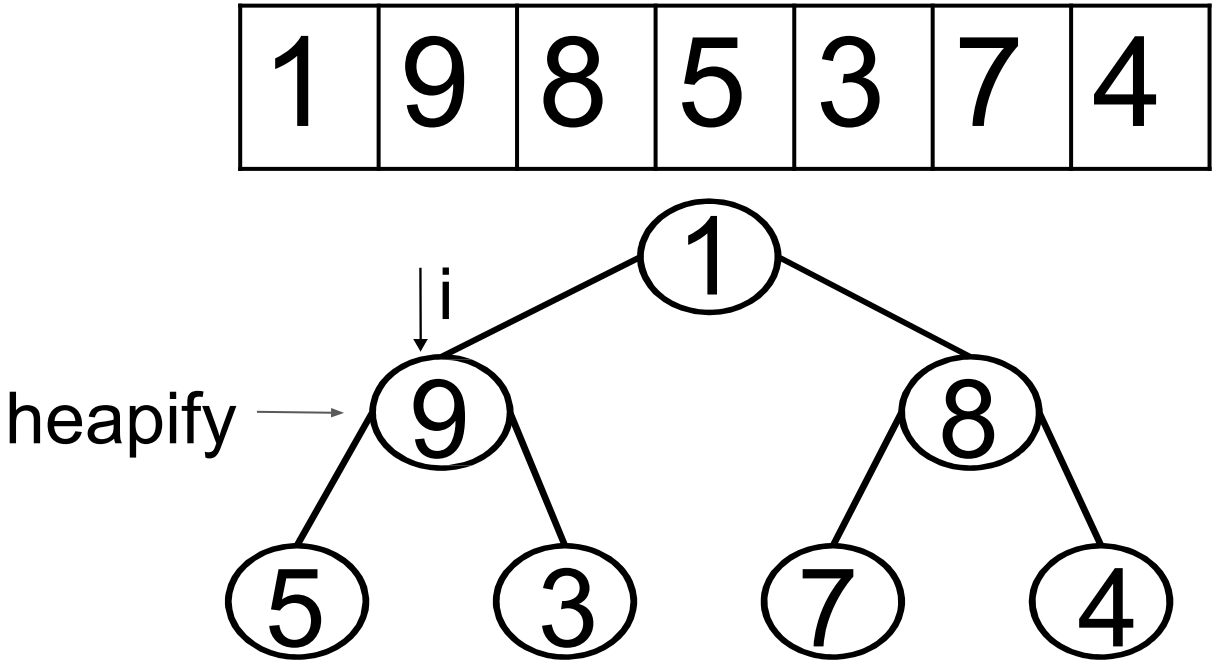
Heapsort

Construindo uma Heap



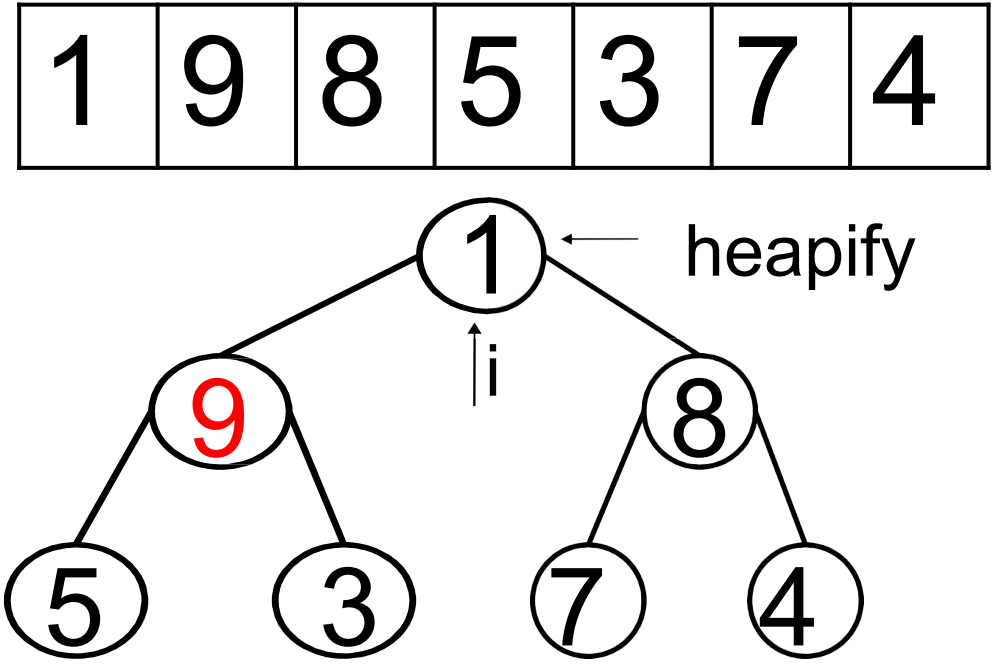
Heapsort

Construindo uma Heap



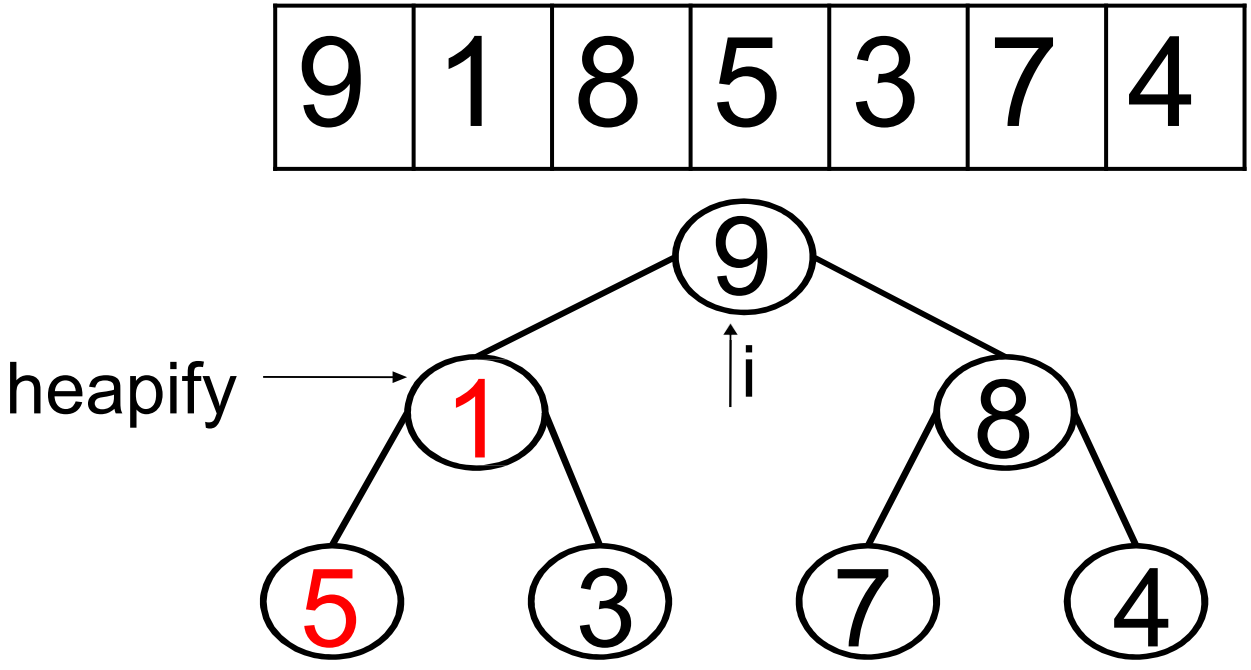
Heapsort

Construindo uma Heap



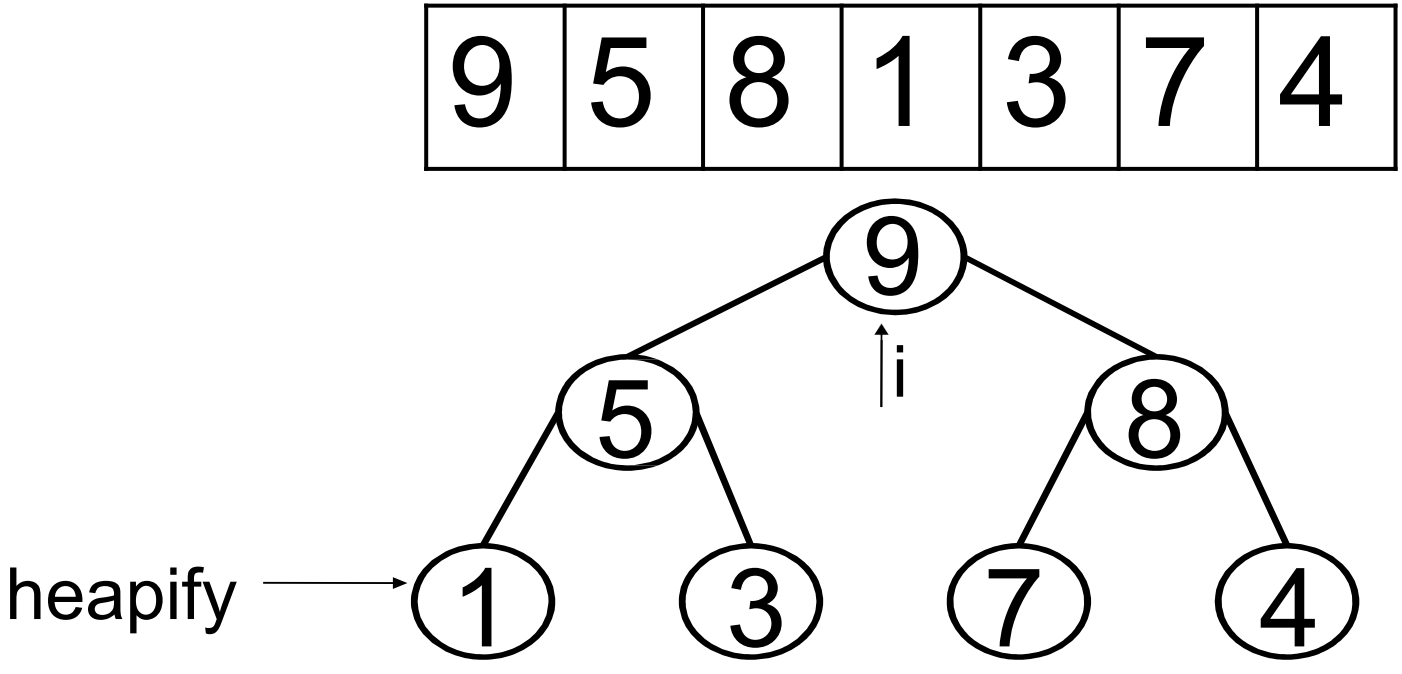
Heapsort

Construindo uma Heap



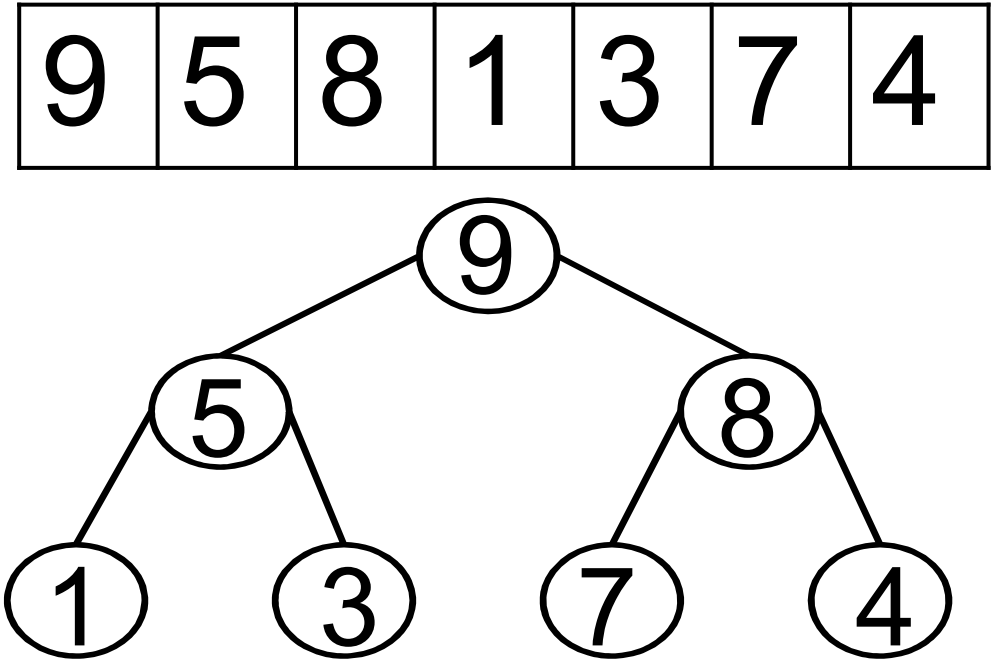
Heapsort

Construindo uma Heap



Heapsort

Construindo uma Heap



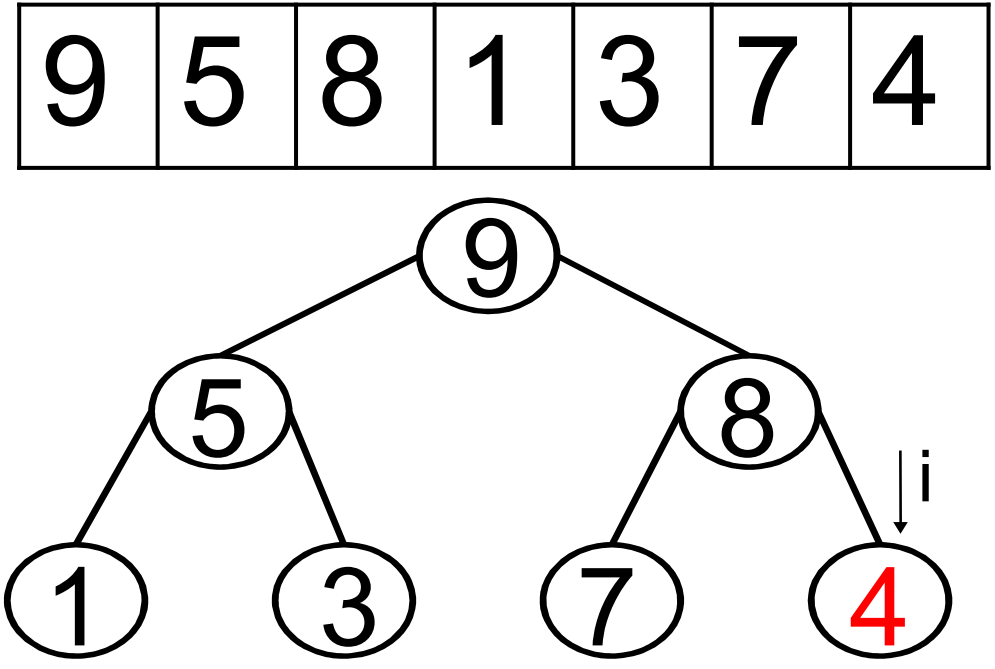
Heapsort

Código

```
function heapsort(V)
    for i=length(V),2,-1 do
        V[1], V[i] = V[i], V[1]
        heap-size(V) = heap-size(V)- 1
        heapify(V,1)
    end
end
```

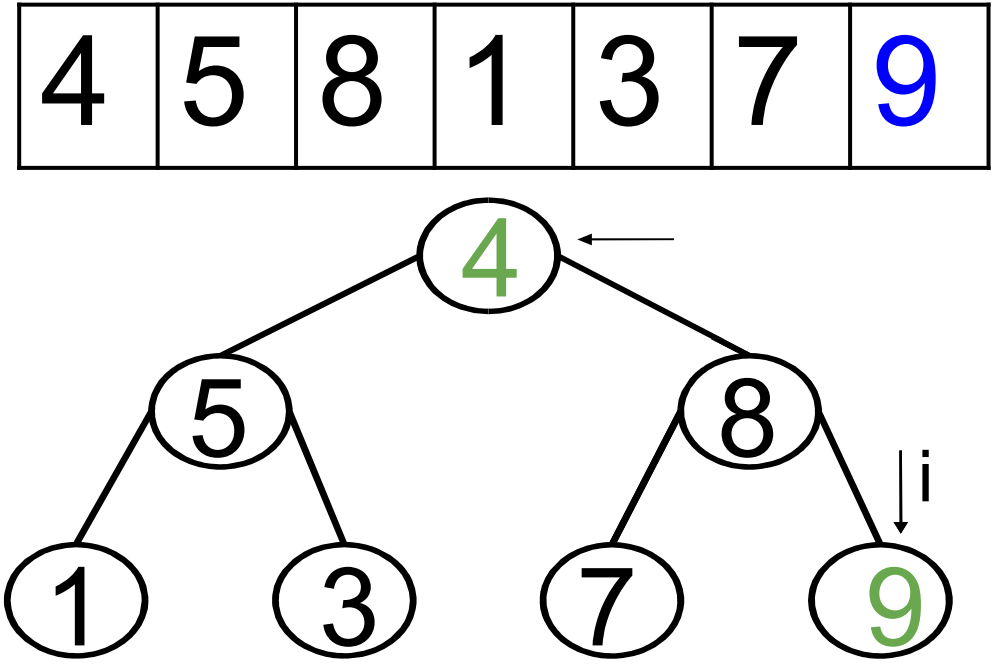
Heapsort

Código



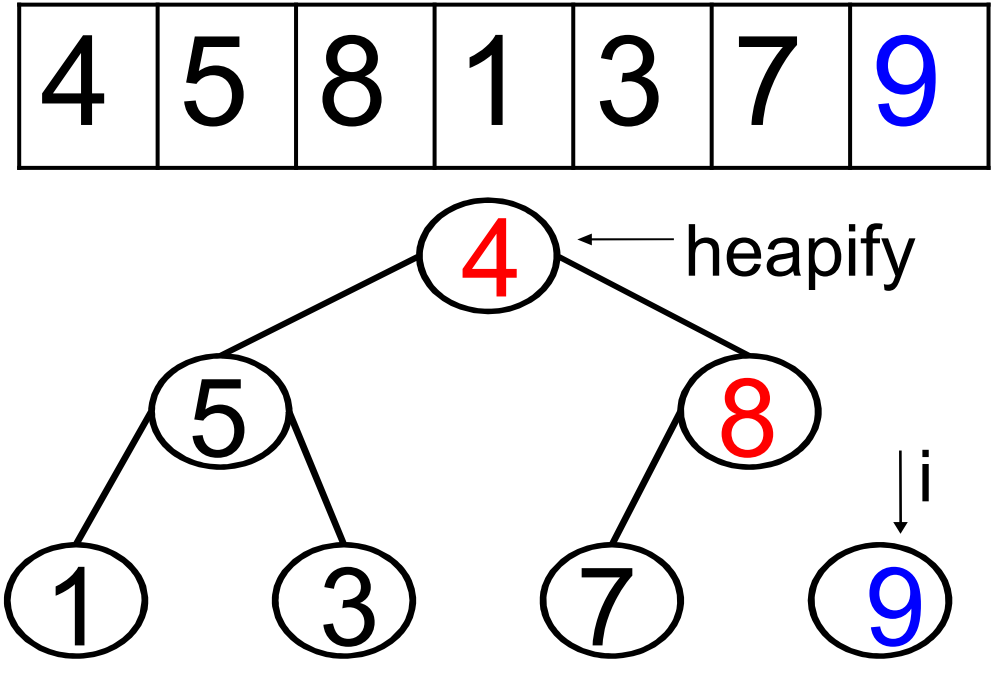
Heapsort

Código



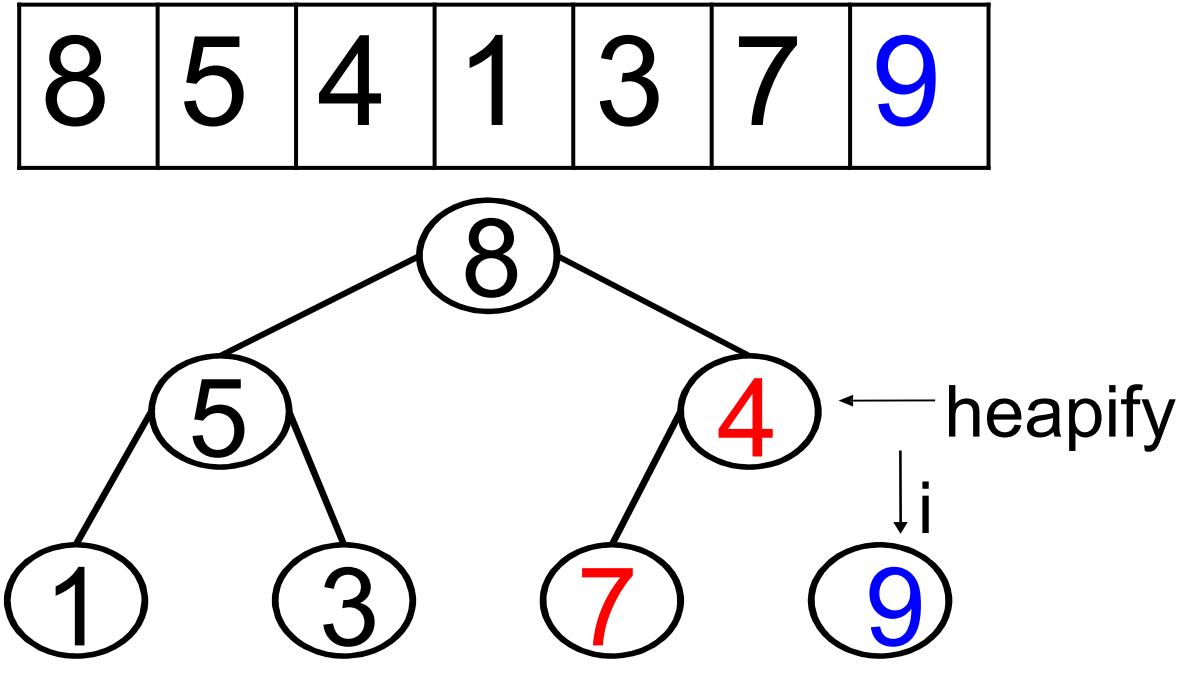
Heapsort

Código



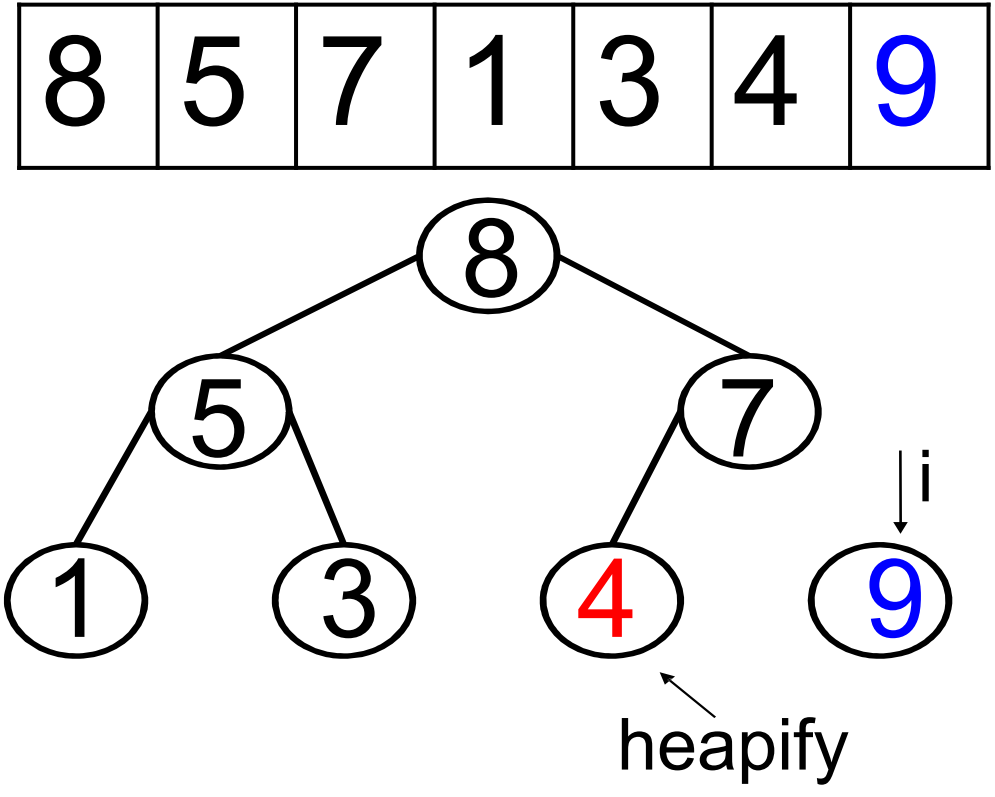
Heapsort

Código



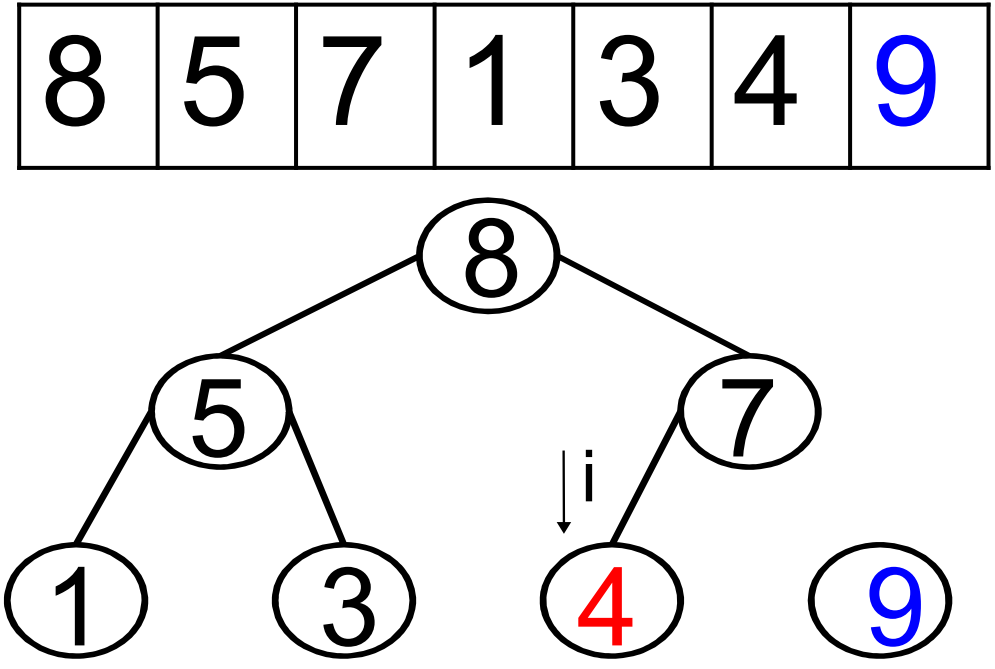
Heapsort

Código



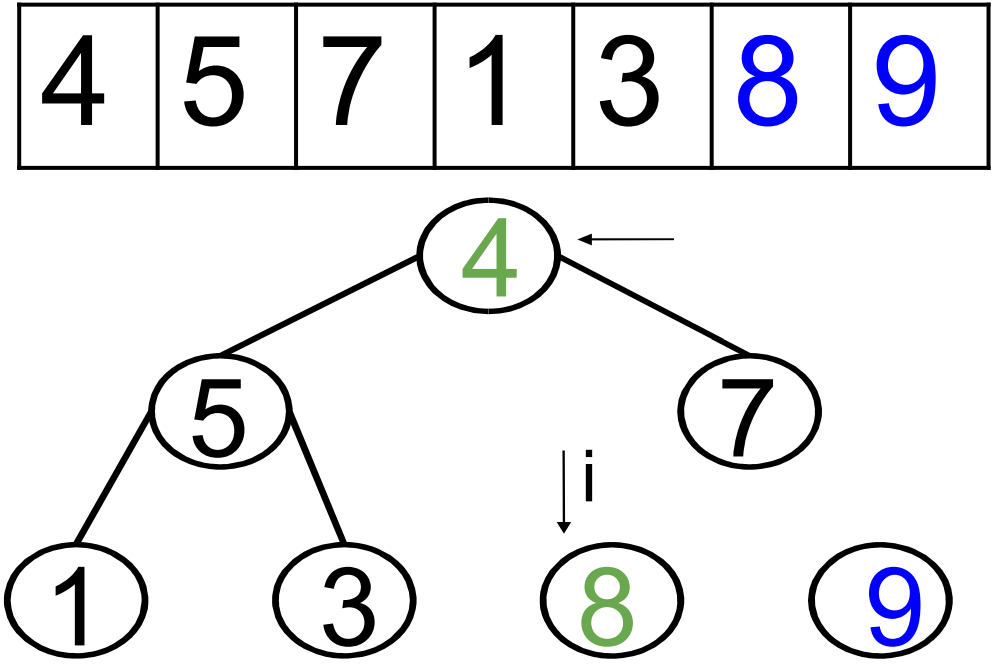
Heapsort

Código



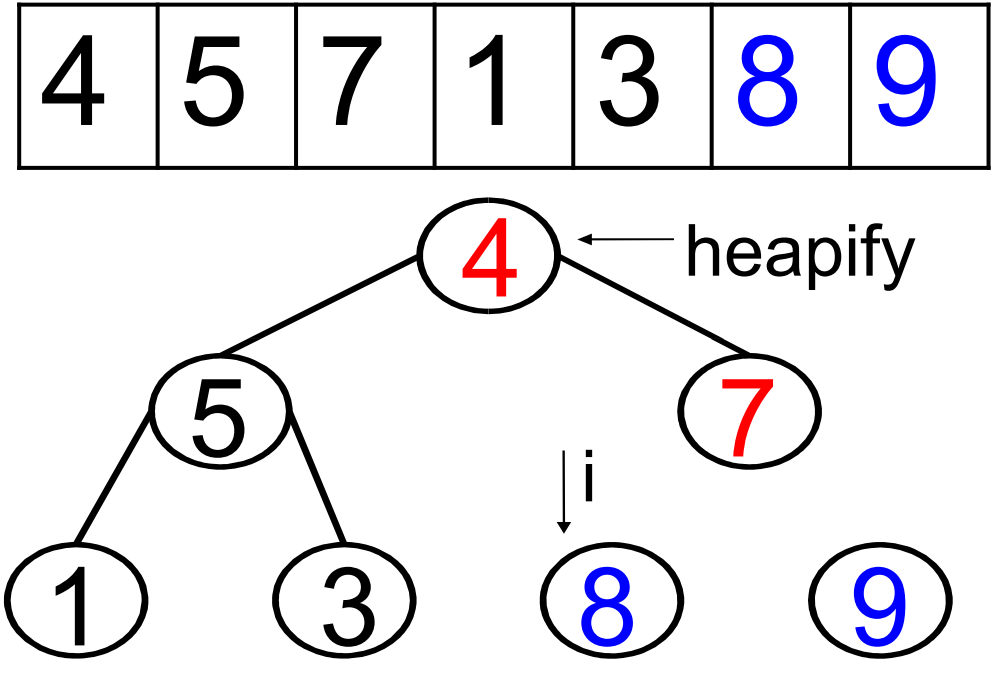
Heapsort

Código



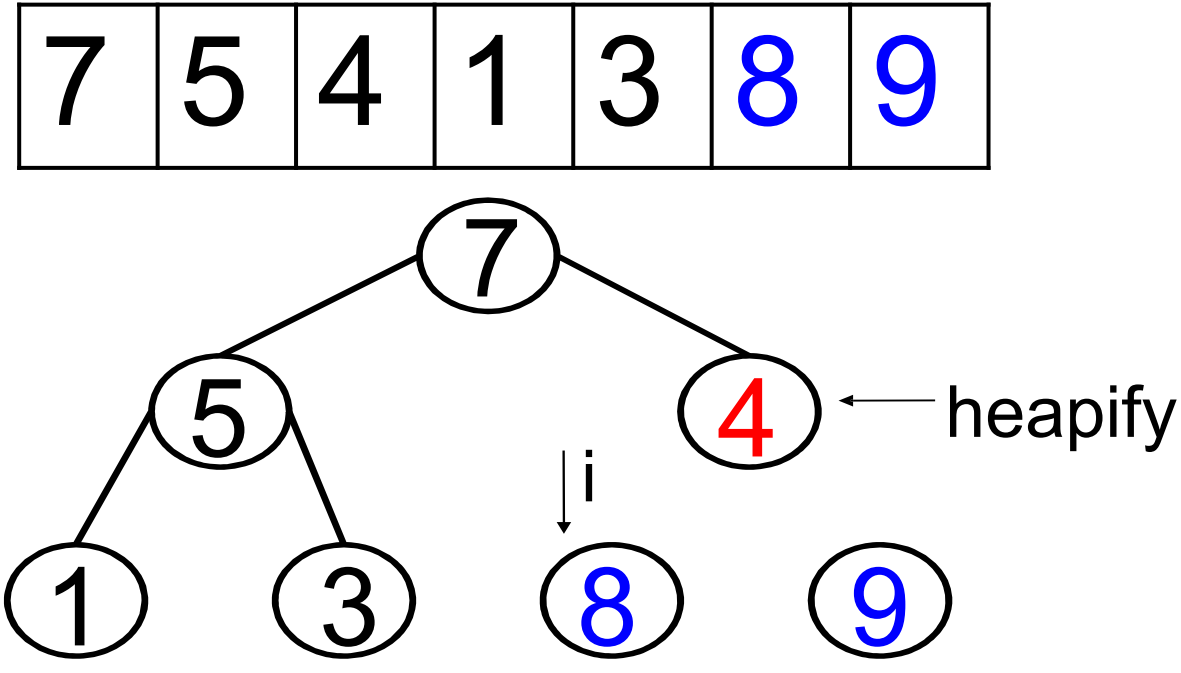
Heapsort

Código



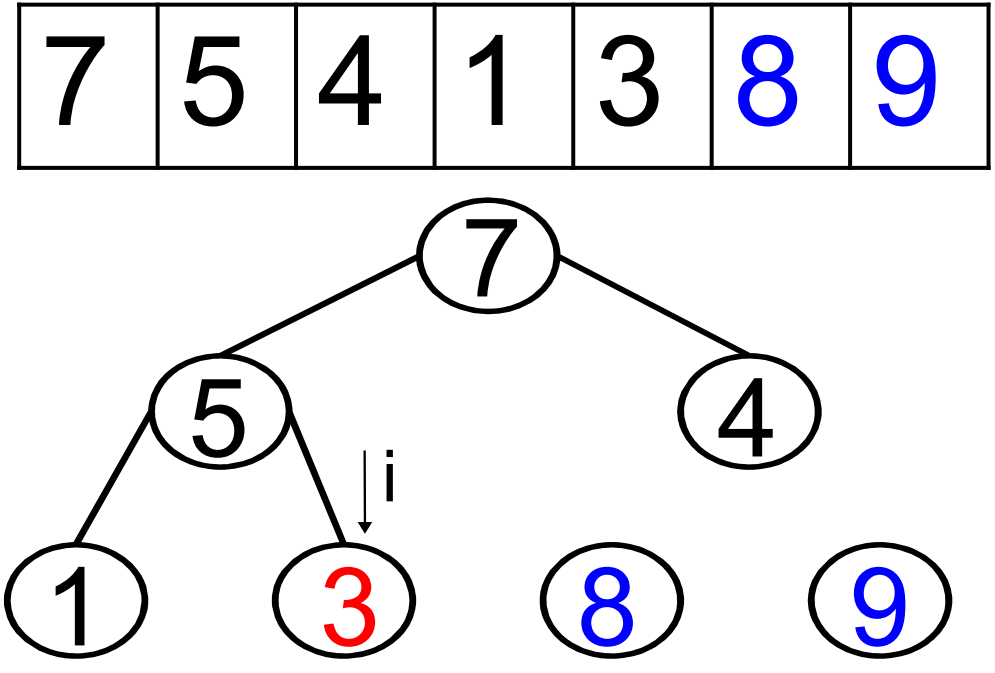
Heapsort

Código



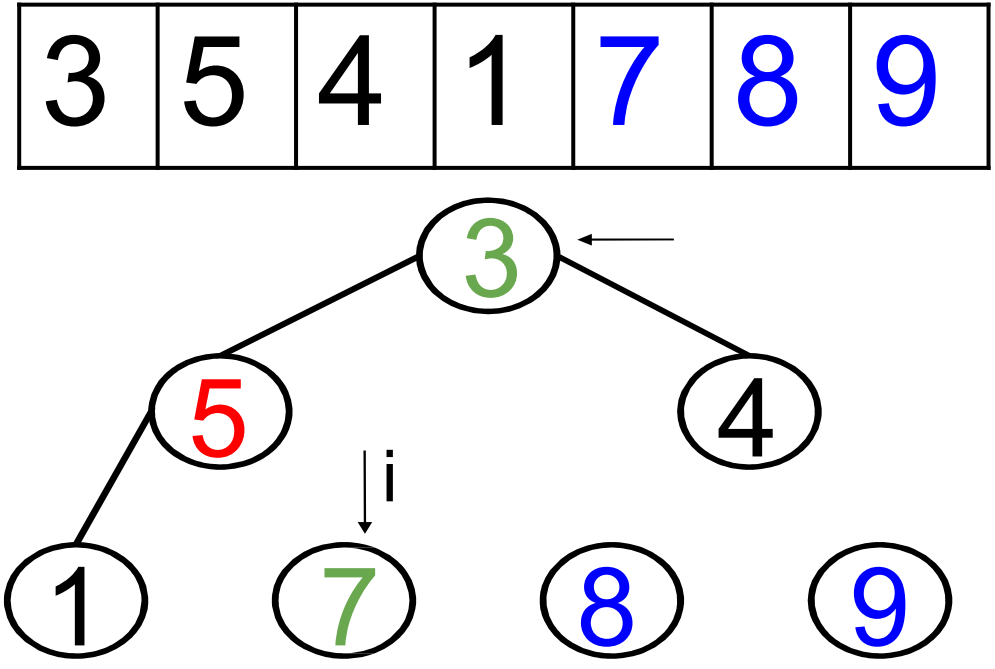
Heapsort

Código



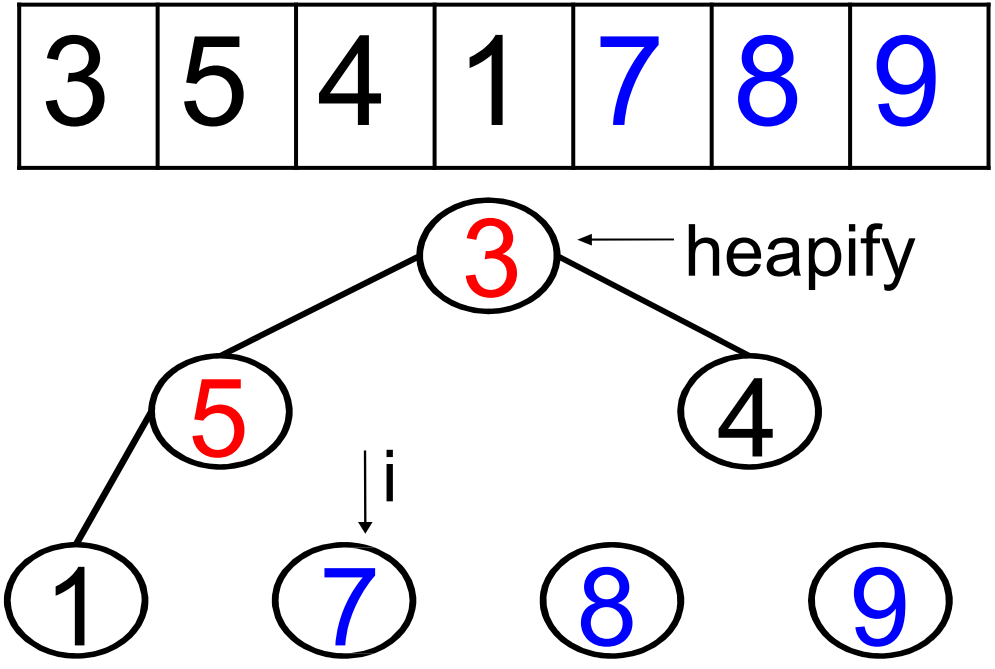
Heapsort

Código



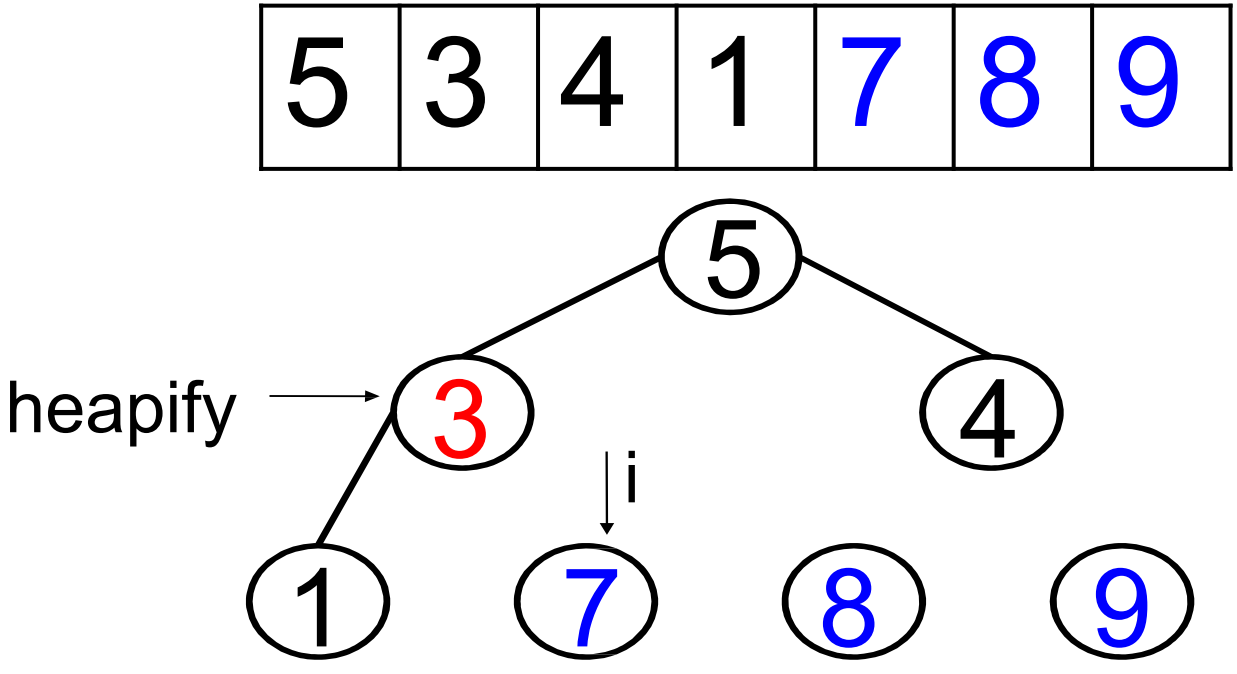
Heapsort

Código



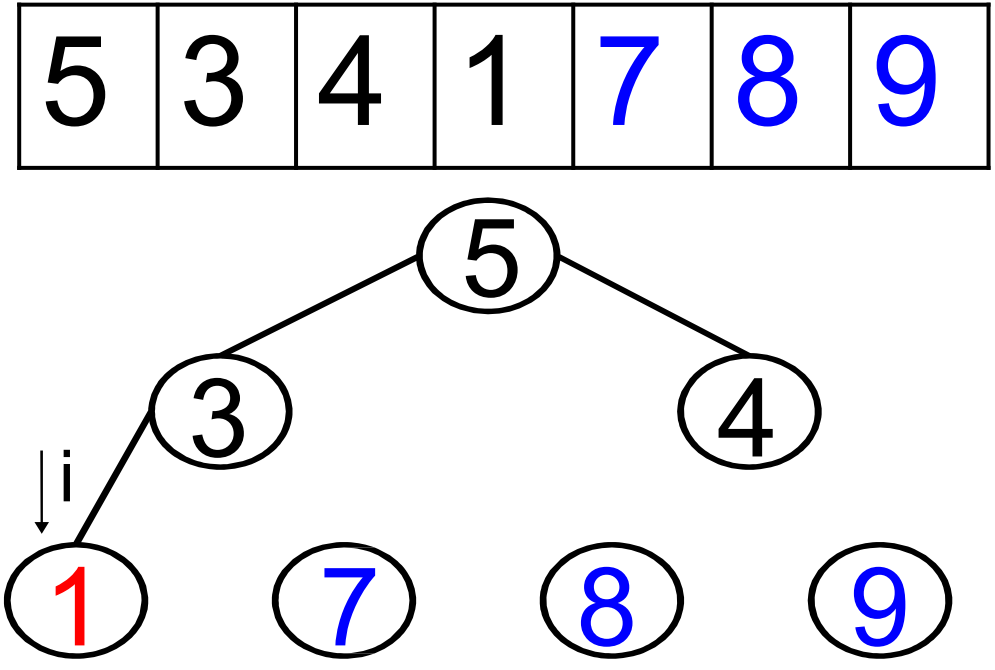
Heapsort

Código



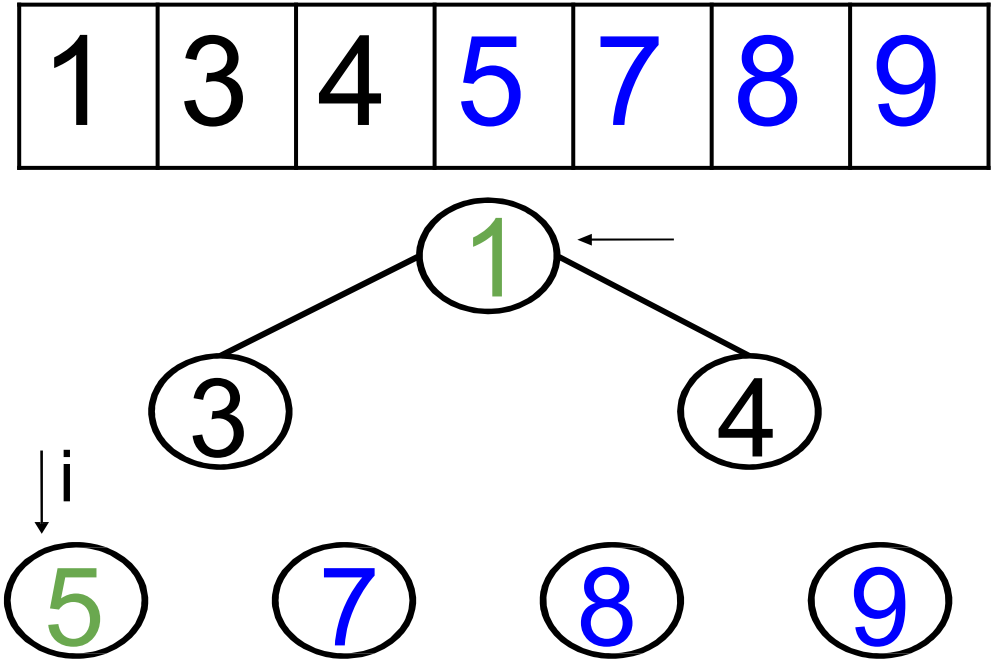
Heapsort

Código



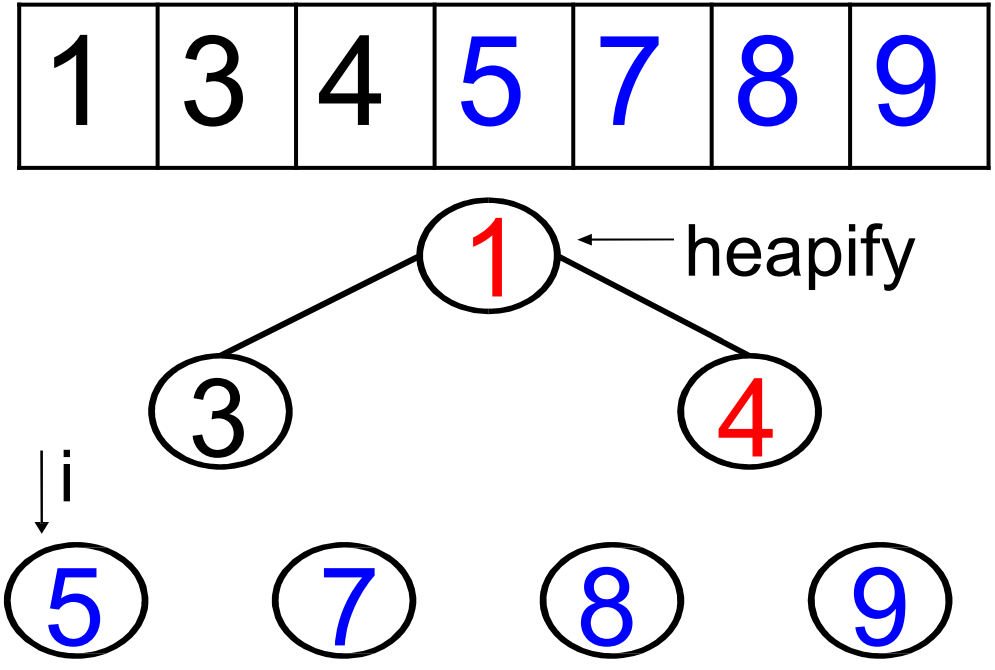
Heapsort

Código



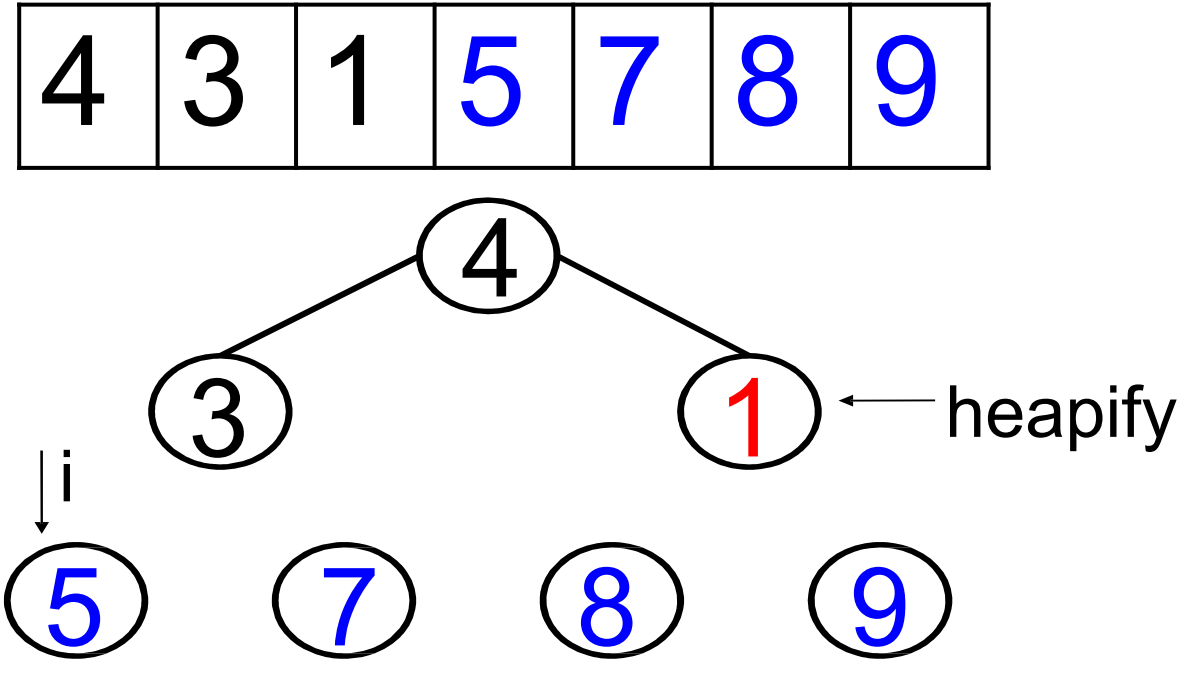
Heapsort

Código



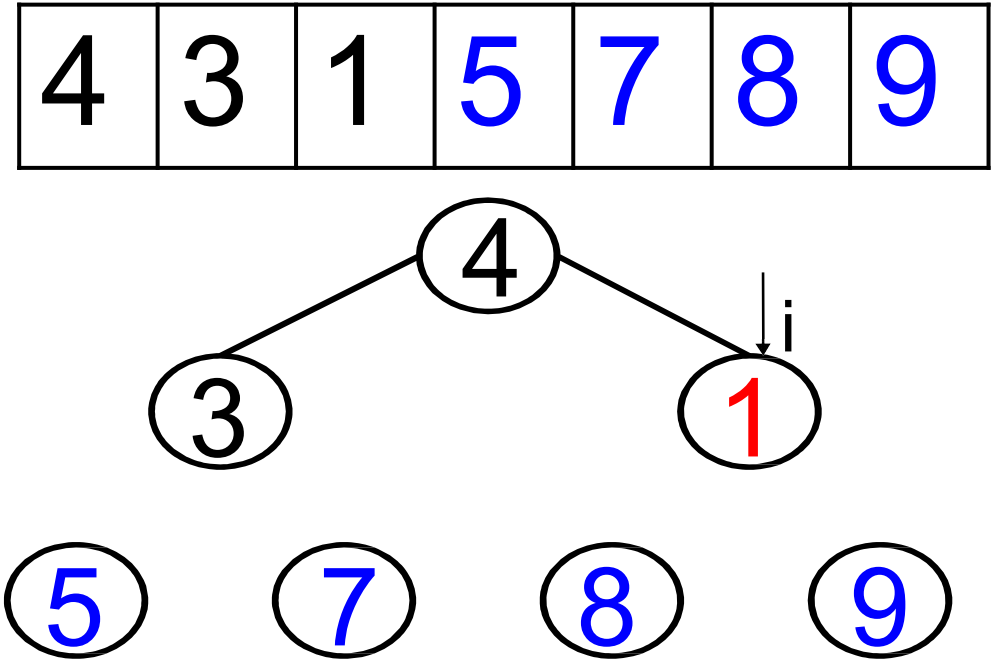
Heapsort

Código



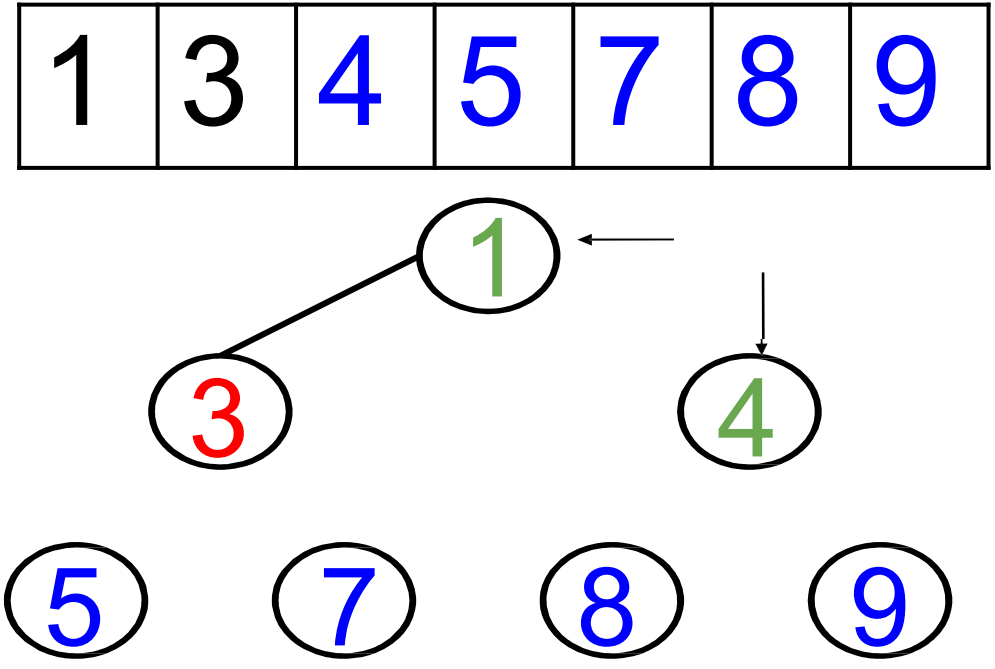
Heapsort

Código



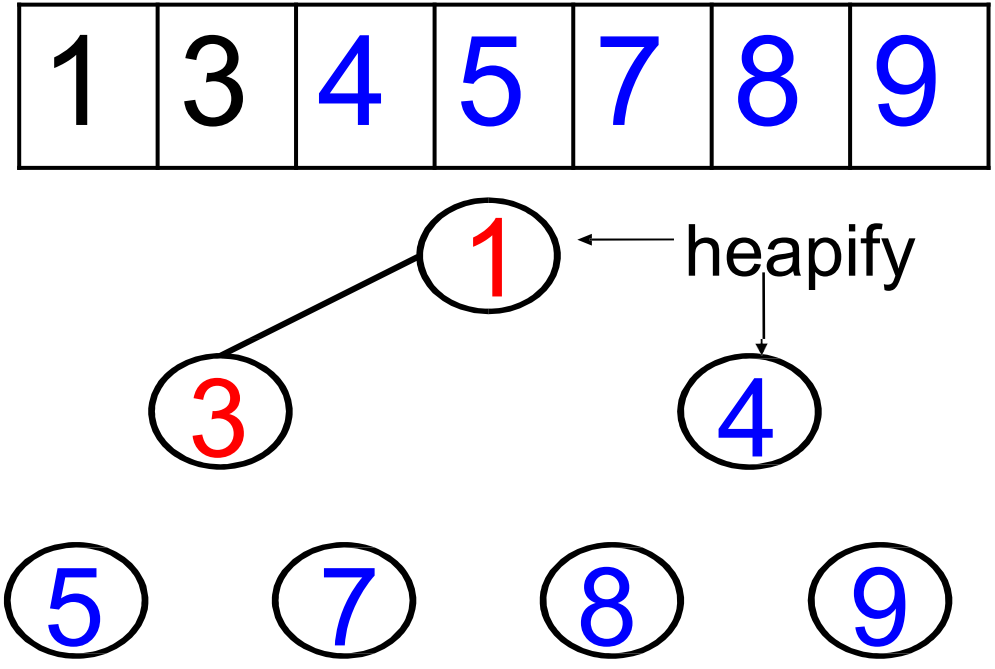
Heapsort

Código



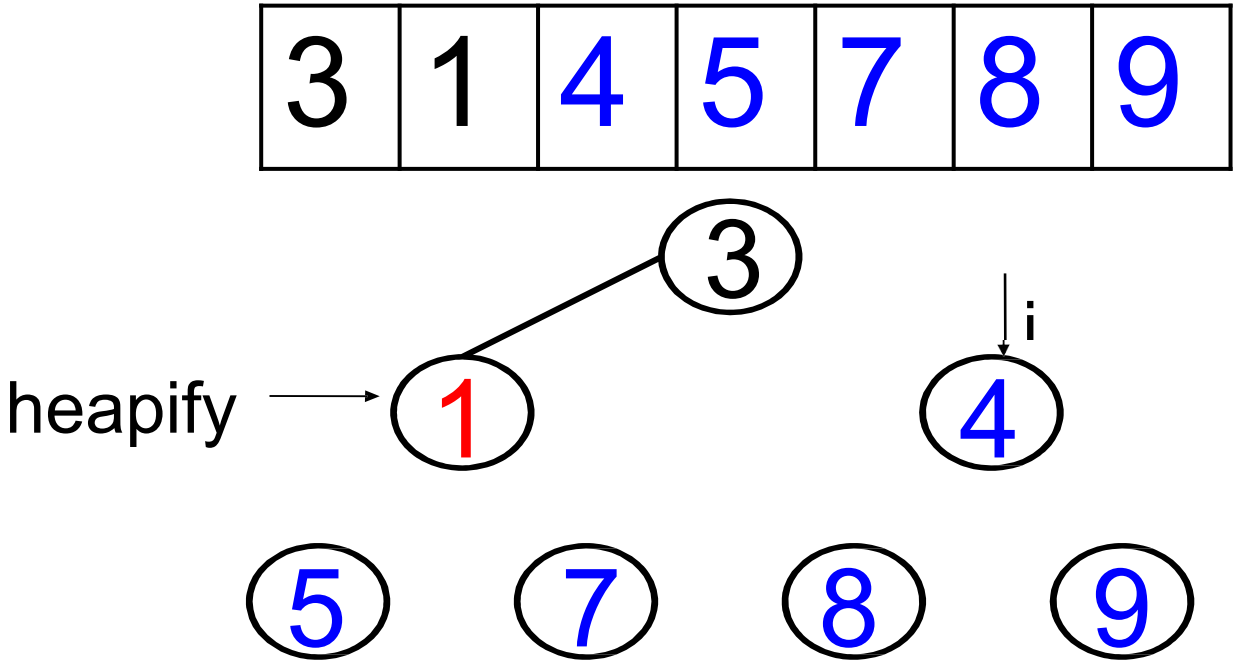
Heapsort

Código



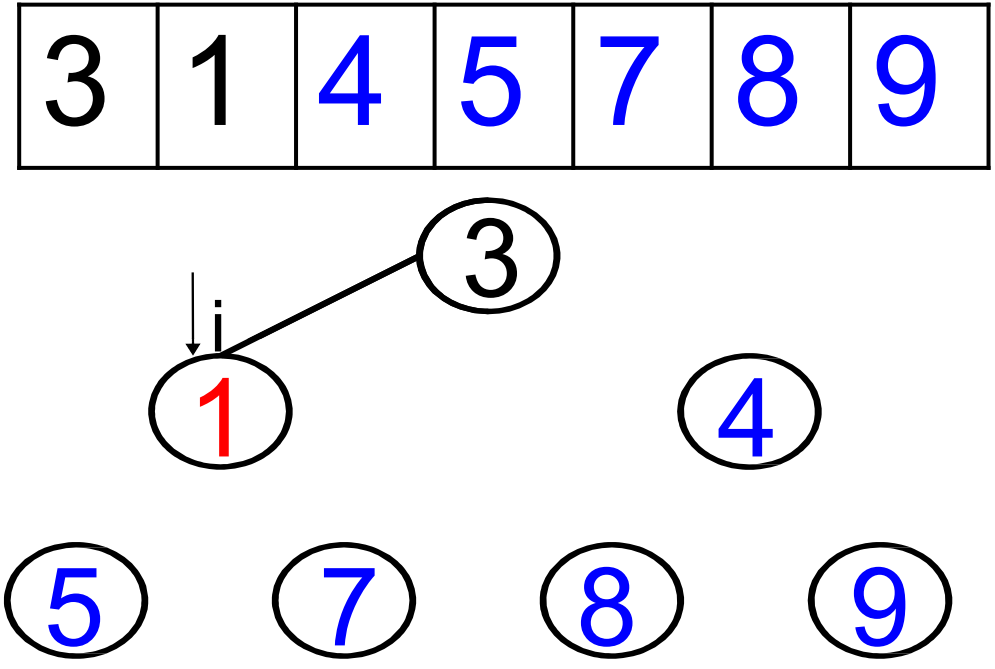
Heapsort

Código



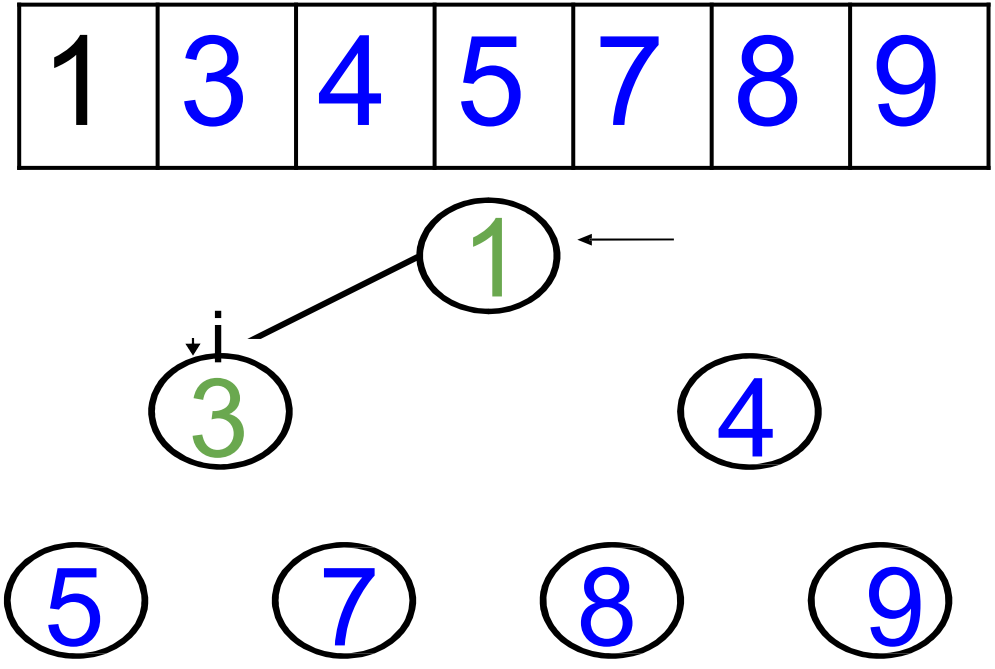
Heapsort

Código



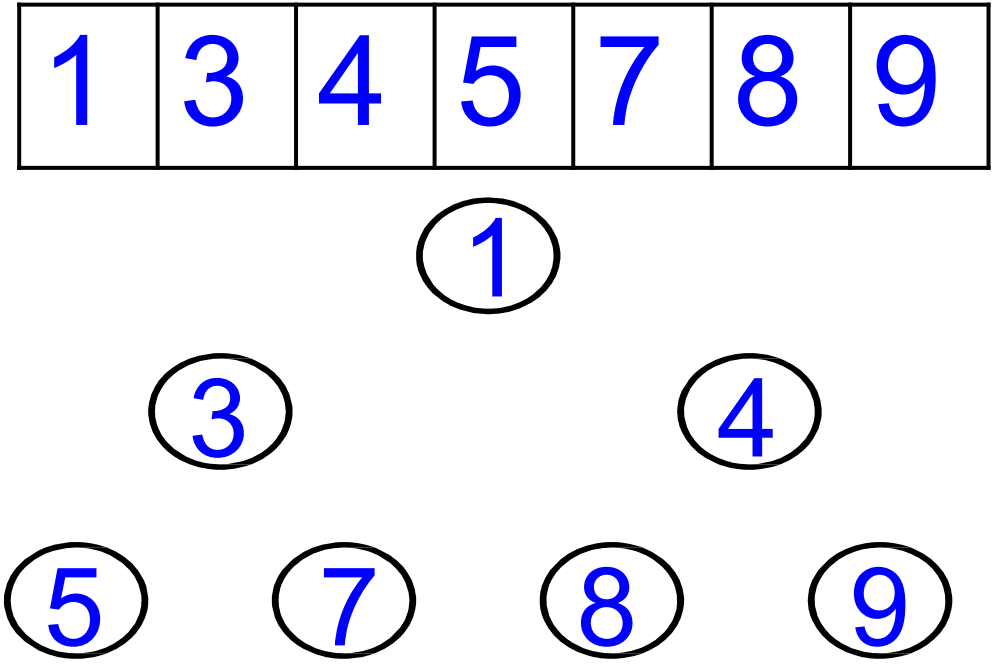
Heapsort

Código

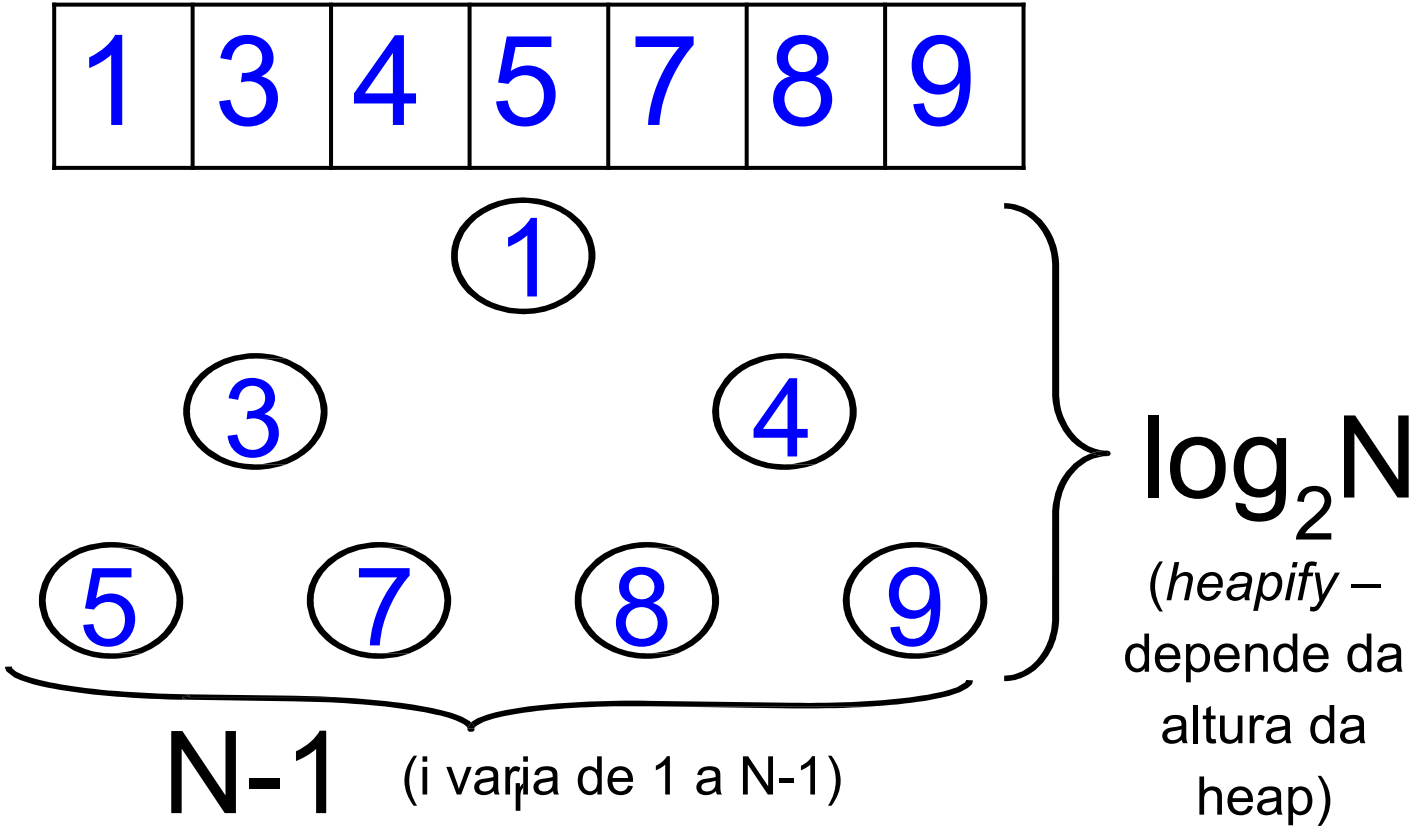


Heapsort

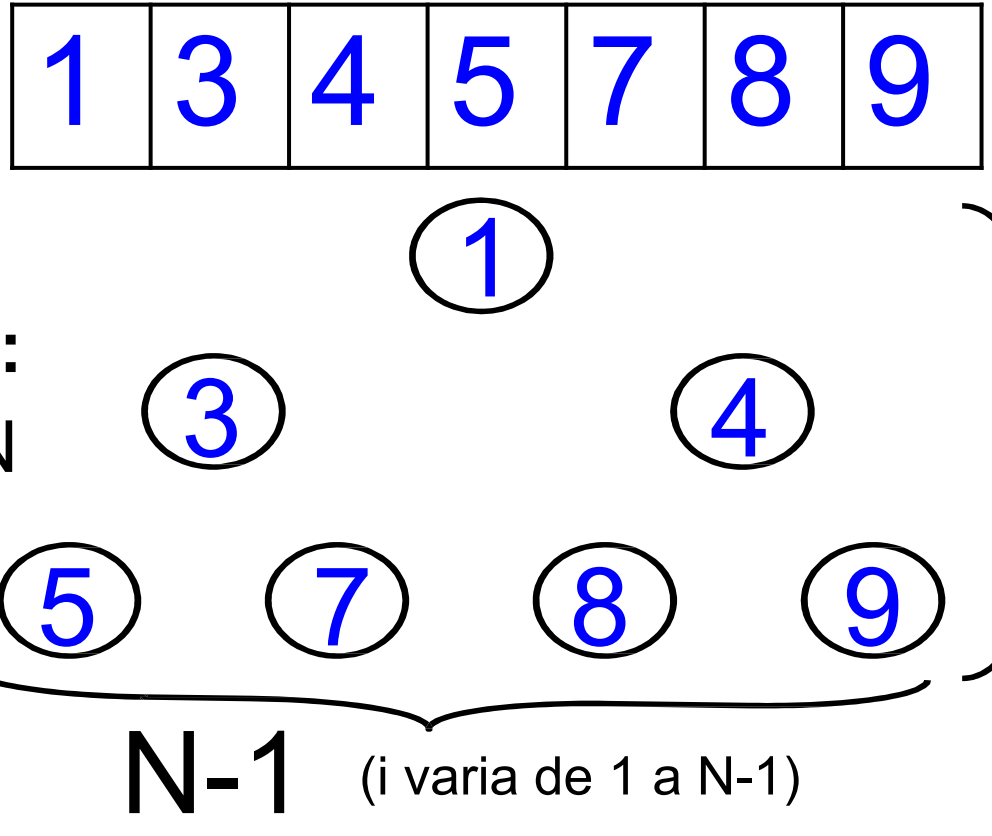
Código



Operações do Heapsort



Operações do Heapsort



Total de operações:
$(N-1) * \log_2 N$

Countsort

Countsort

Apenas quando os valores do vetor **v** variarem em um intervalo pequeno (Ex.: 0 a 100).

```
c = {}
for i=0, 100 do
   c[i] = 0
end
for i=1,#v do
   c[ v[i] ] = c[ v[i] ]+1 end
j = 1
for i=0,100 do while
   c[i]>0 do
      v[j] = i
      j = j+1
      c[i] = c[i]-1 end
end
```

Countsort

V = 3 1 2 1 1 3

Countsort

$$V = 3\ 1\ 2\ 1\ 1\ 3$$

$$C = \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{matrix}$$

Countsort

$$V = \underset{\uparrow i}{3}\ 1\ 2\ 1\ 1\ 3$$

$$C = \begin{array}{cccc} 0 & 0 & 0 & 1 \\ 0 & 1 & 2 & 3 \end{array}$$

Countsort

$$V = 3\ 1\ 2\ 1\ 1\ 3$$

$\uparrow_i$

$$C = \begin{array}{cccc} 0 & 1 & 0 & 1 \\ 0 & 1 & 2 & 3 \end{array}$$

Countsort

$$V = 3\ 1\ 2\ 1\ 1\ 3$$

$\uparrow_i$

$$C = \begin{matrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{matrix}$$

Countsort

V = 3 1 2 1 1 3

↑ i

C = 0 2 1 1

0 1 2 3

Countsort

V = 3 1 2 1 1 3

↑ i

C = 0 3 1 1

0 1 2 3

Countsort

V = 3 1 2 1 1 3

↑ i

C = 0 3 1 2

0 1 2 3

Countsort

V = 3 1 2 1 1 3

↑j

C = 0 3 1 2

0 1 2 3

↑ ↑

i

Countsort

$$V = \underset{\uparrow j}{3}\ 1\ 2\ 1\ 1\ 3$$

$$C = \begin{array}{cccc} 0 & 3 & 1 & 2 \\ 0 & \underset{\uparrow i}{1} & 2 & 3 \end{array}$$

Countsort

V = 1 1 2 1 1 3

↑j

C = 0 2 1 2

0 1 2 3

↑i

Countsort

V = 1 1 2 1 1 3

↑j

C = 0 1 1 2

0 1 2 3

↑i

Countsort

V = 1 1 1 1 1 3

j

C = 0 0 1 2

0 1 2 3

i

Countsort

$$V = 1\ 1\ 1\ 1\ 1\ 3$$

j

$$C = \begin{matrix} 0 & 0 & 1 & 2 \\ 0 & 1 & 2 & 3 \end{matrix}$$

i

Countsort

$$V = 1\ 1\ 1\ \underset{\uparrow j}{2}\ 1\ 3$$

$$C = \begin{matrix} 0 & 0 & 0 & 2 \\ 0 & 1 & \underset{\uparrow i}{2} & 3 \end{matrix}$$

Countsort

$$V = 1\ 1\ 1\ 2\ 1\ 3$$

$\uparrow j$

$$C = \begin{matrix} 0 & 0 & 0 & 2 \\ 0 & 1 & 2 & 3 \end{matrix}$$

$\uparrow i$

Countsort

V= 1 1 1 2 3 3

j

C = 0 0 0 1

0 1 2 3

i

Countsort

V = 1 1 1 2 3 3

↑j

C = 0 0 0 0

0 1 2 3

↑i

Countsort

$$V = 1\ 1\ 1\ 2\ 3\ 3$$

$$C = \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{matrix}$$

Radixsort

Radixsort

- Ordena o vetor da casa decimal menos significativa para a mais significativa
- Não precisa necessariamente ser de dígito em dígito, pois pode ser de byte em byte

Radixsort

- 133
- 252
- 411
- 323
- 510
- 523
- 101

Radixsort

- 133
- 252
- 411
- 323
- 510
- 523
- 101

- 510
- 411
- 101
- 252
- 133
- 323
- 523

Radixsort

- 133
- 252
- 411
- 323
- 510
- 523
- 101

- 510
- 411
- 101
- 252
- 133
- 323
- 523

Radixsort

- 133
- 252
- 411
- 323
- 510
- 523
- 101

- 510
- 411
- 101
- 252
- 133
- 323
- 523

- 101
- 510
- 411
- 323
- 523
- 133
- 252

Radixsort

- 133
- 252
- 411
- 323
- 510
- 523
- 101

- 510
- 411
- 101
- 252
- 133
- 323
- 523

- 101
- 510
- 411
- 323
- 523
- 133
- 252

Radixsort

- 133
- 252
- 411
- 323
- 510
- 523
- 101

- 510
- 411
- 101
- 252
- 133
- 323
- 523

↓

- 101
- 510
- 411
- 323
- 523
- 133
- 252

- 101
- 133
- 252
- 323
- 411
- 510
- 523

Radixsort

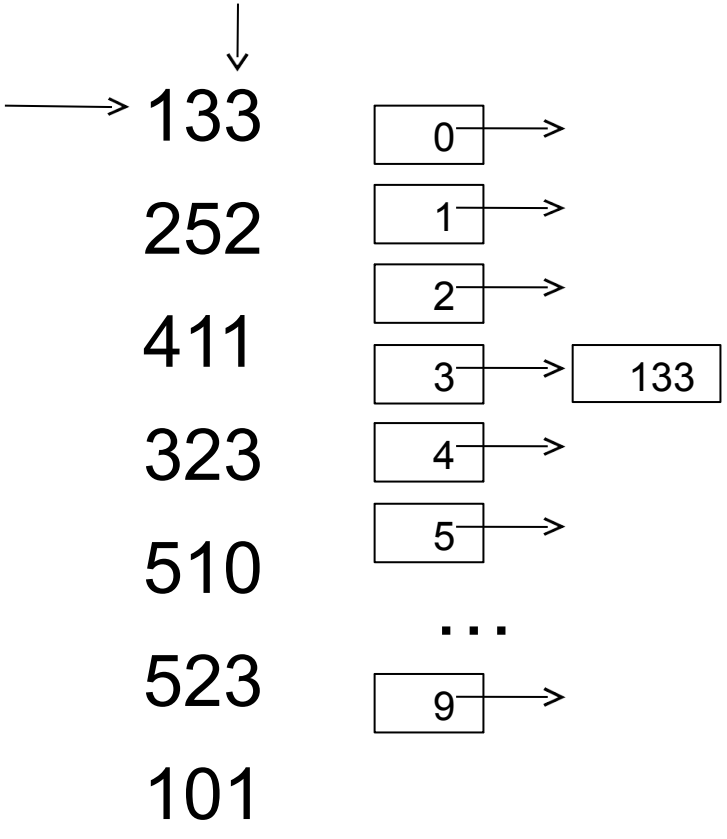
- 133
- 252
- 411
- 323
- 510
- 523
- 101

- 510
- 411
- 101
- 252
- 133
- 323
- 523

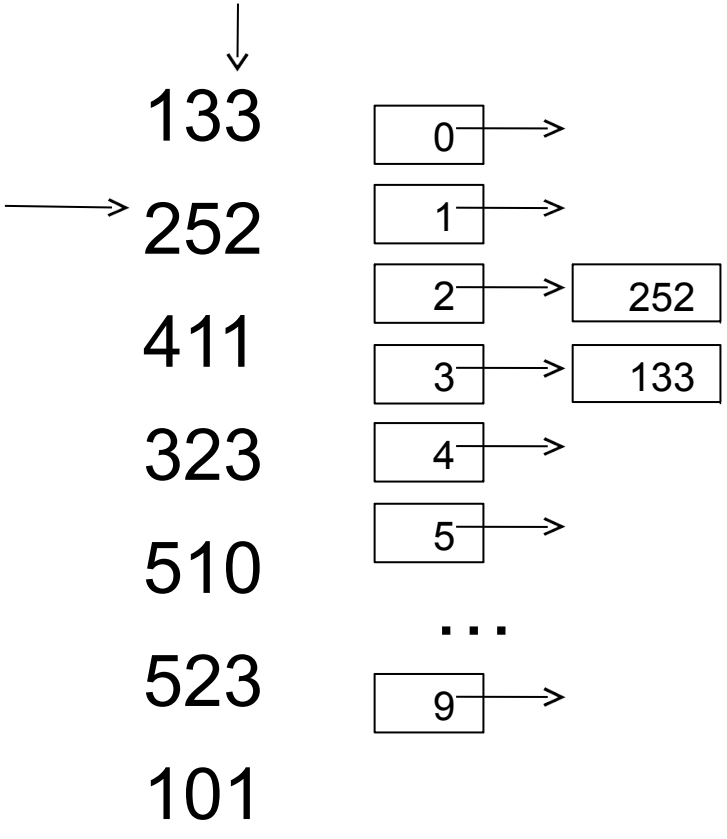
- 101
- 510
- 411
- 323
- 523
- 133
- 252

- 101
- 133
- 252
- 323
- 411
- 510
- 523

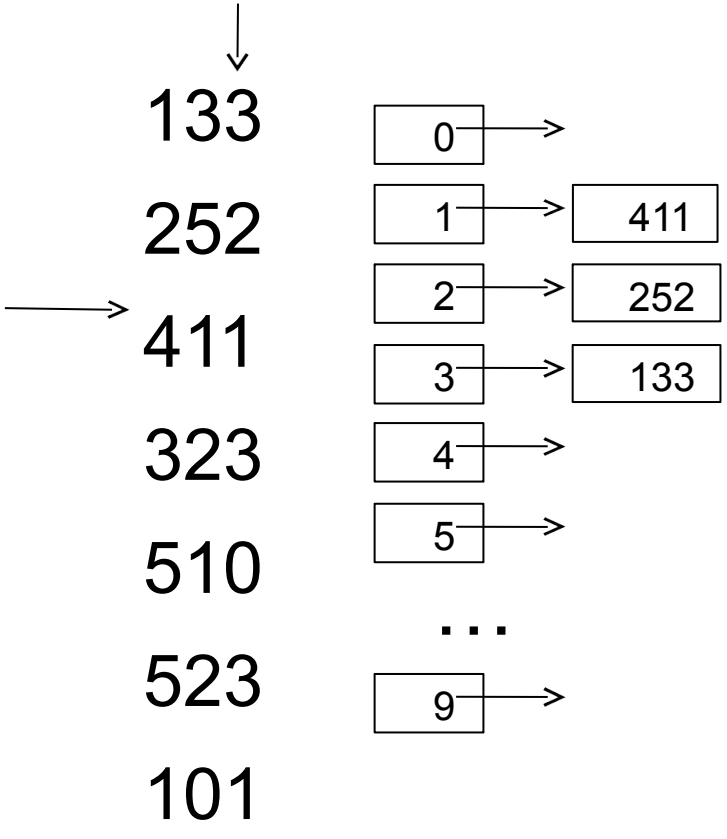
Radixsort – listas encadeadas



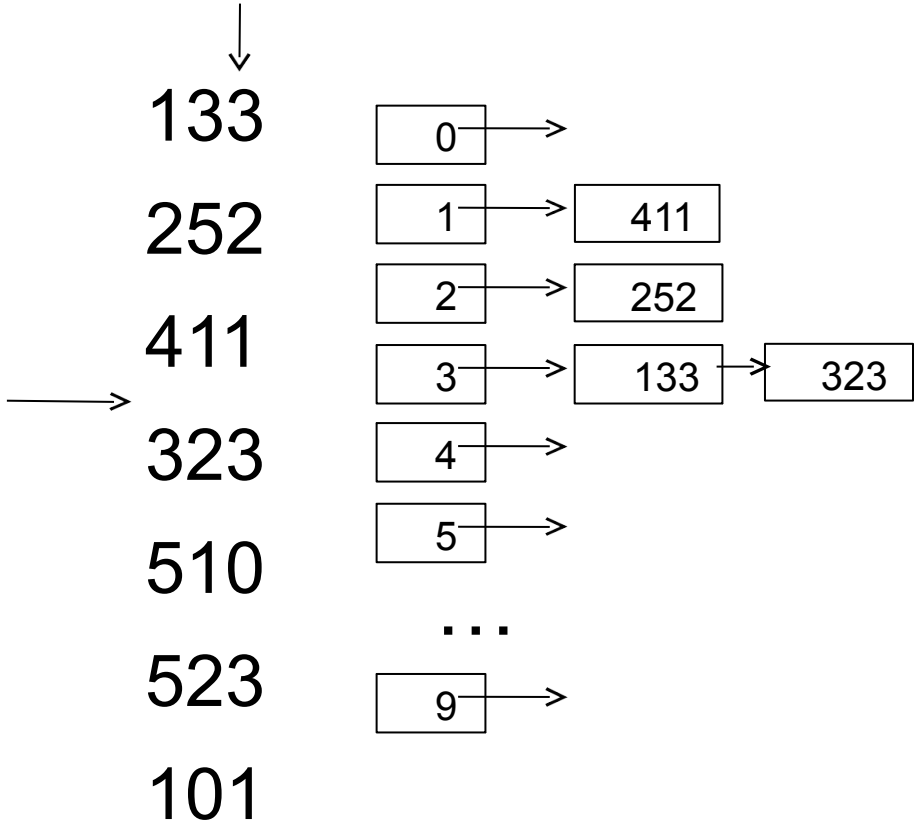
Radixsort – listas encadeadas



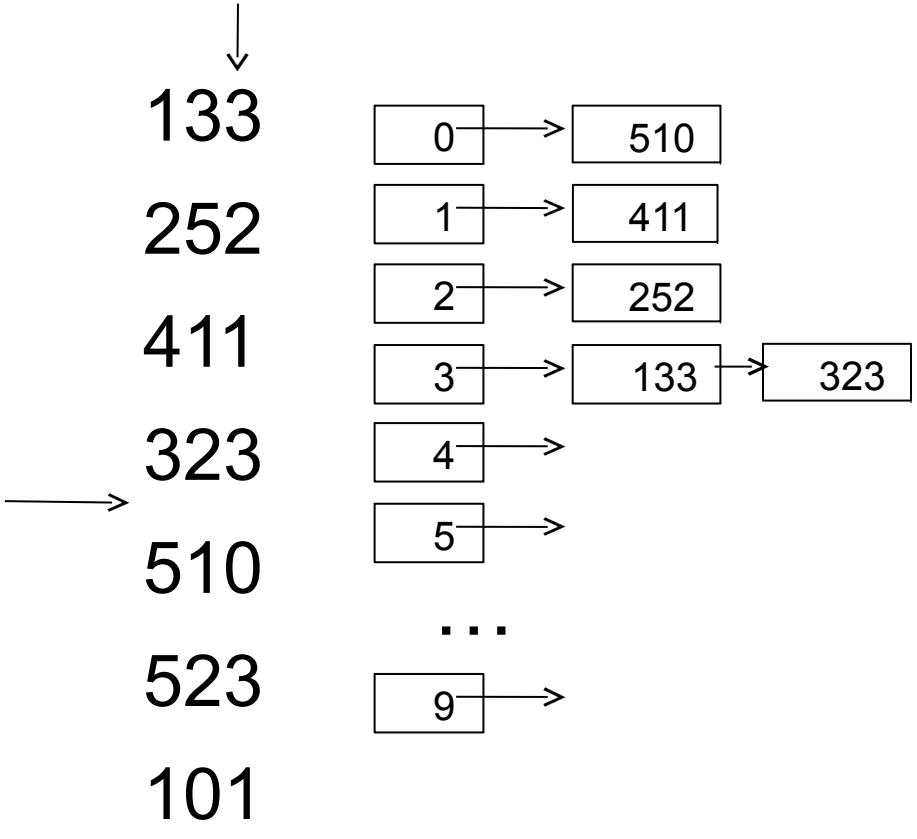
Radixsort – listas encadeadas



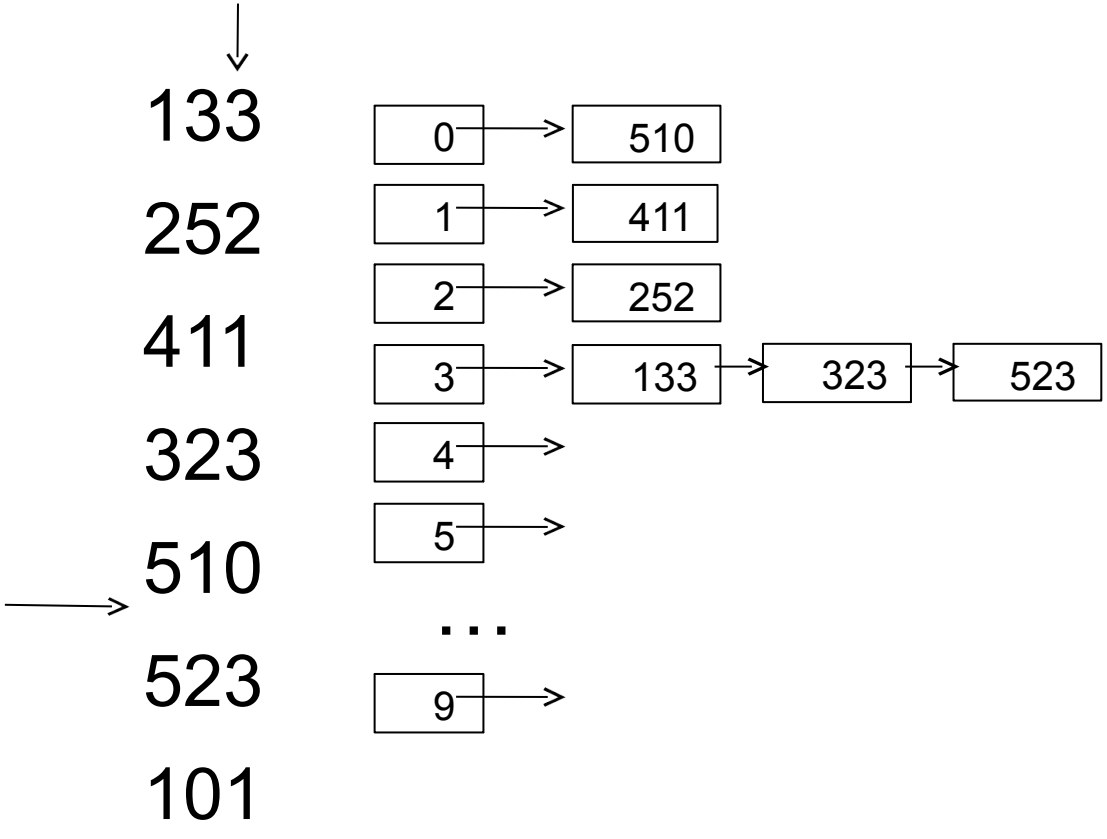
Radixsort – listas encadeadas



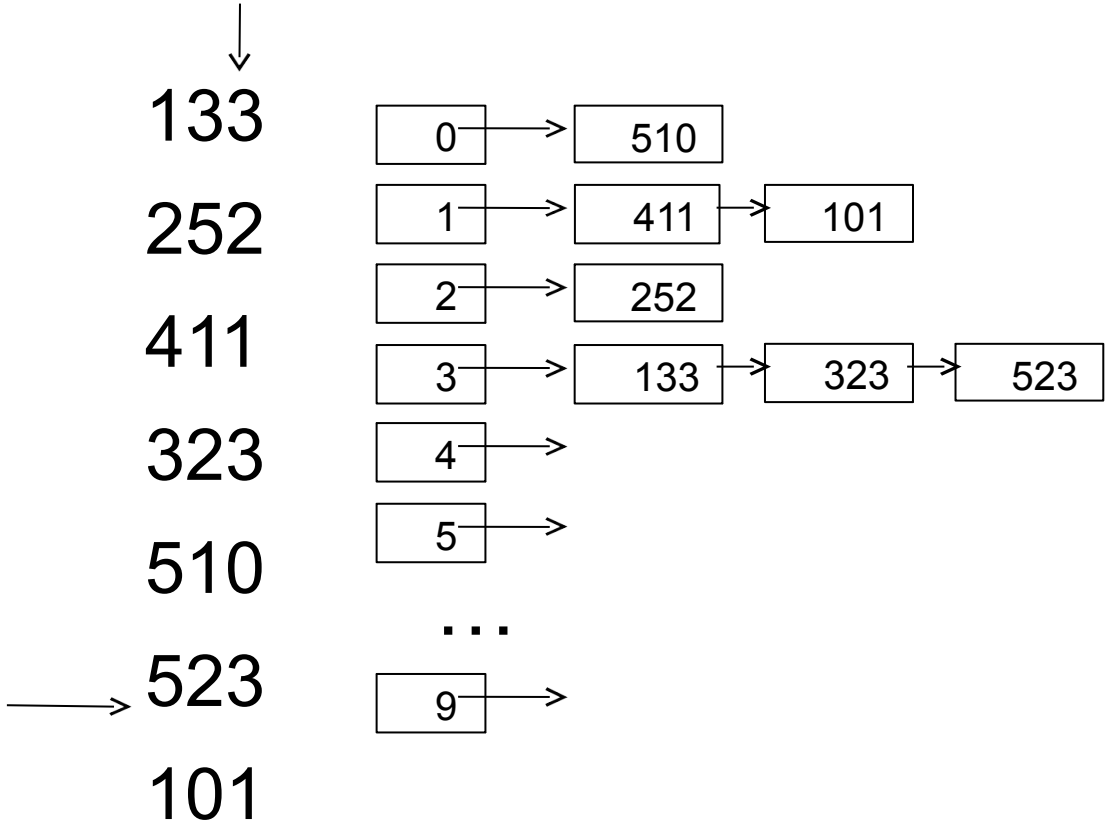
Radixsort – listas encadeadas



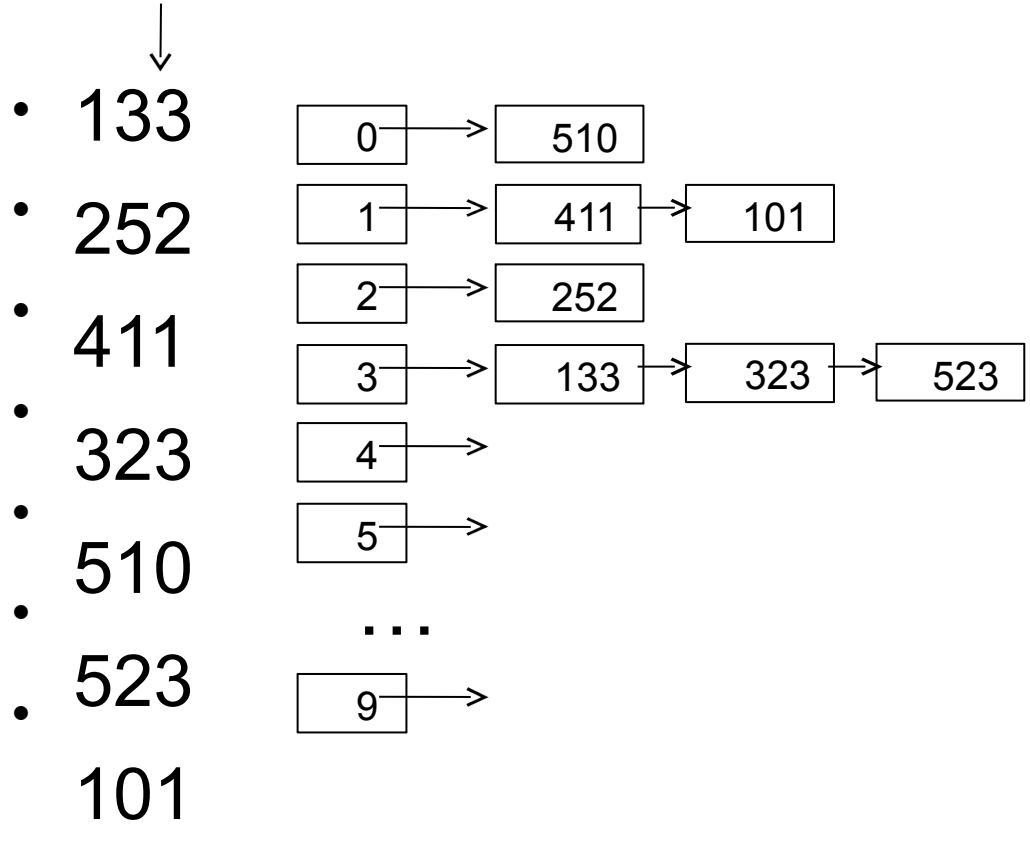
Radixsort – listas encadeadas



Radixsort – listas encadeadas

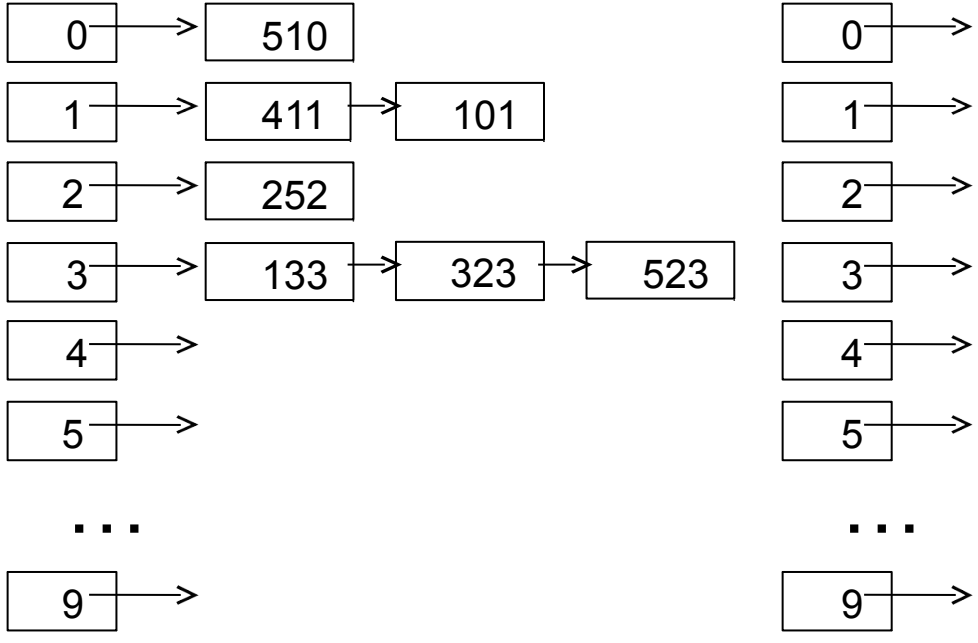


Radixsort – listas encadeadas



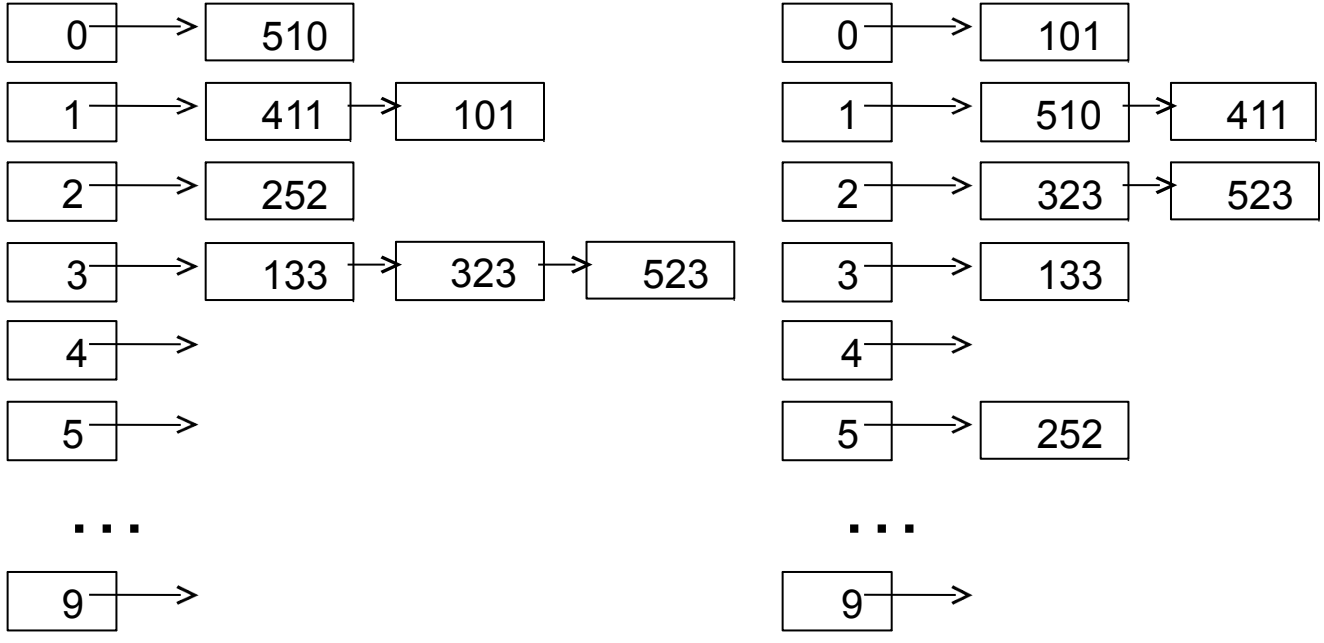
Radixsort – listas encadeadas

- 133
- 252
- 411
- 323
- 510
- 523
- 101



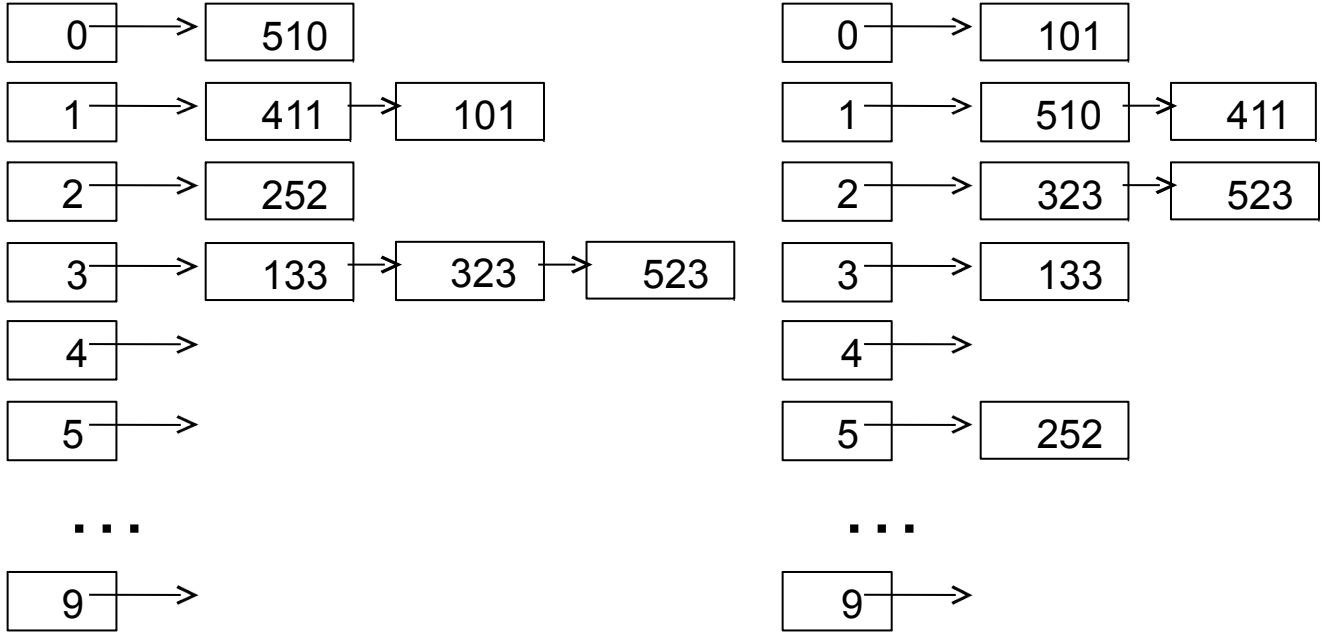
Radixsort – listas encadeadas

- 133
- 252
- 411
- 323
- 510
- 523
- 101



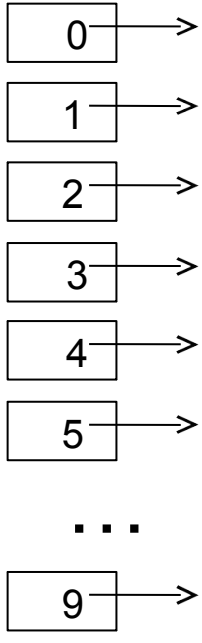
Radixsort – listas encadeadas

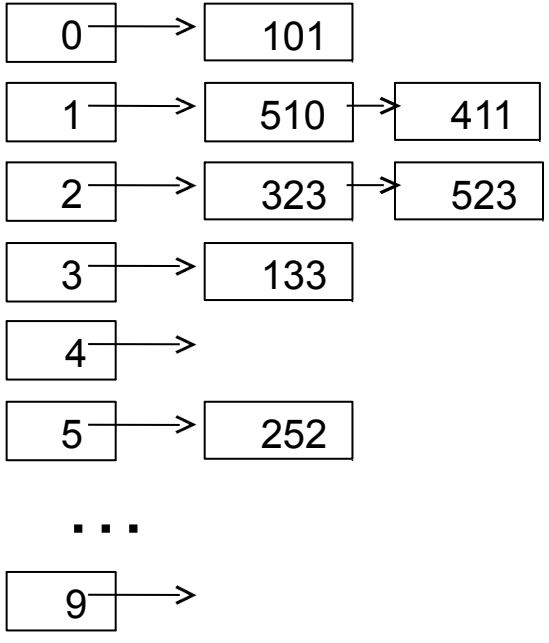
- 133
- 252
- 411
- 323
- 510
- 523
- 101



Radixsort – listas encadeadas

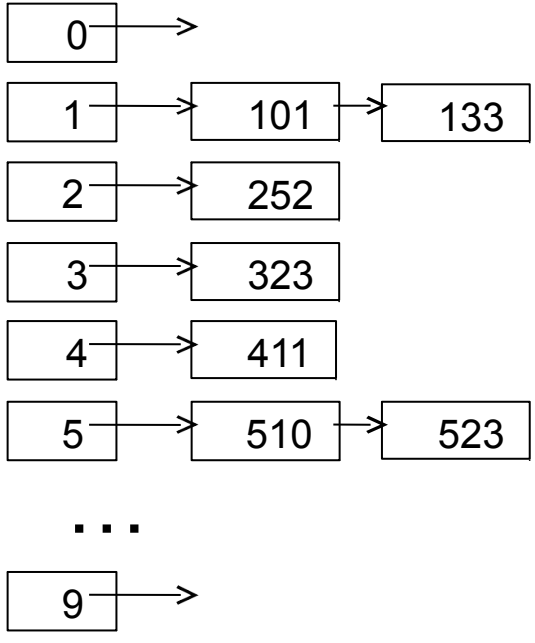
- 133
- 252
- 411
- 323
- 510
- 523
- 101

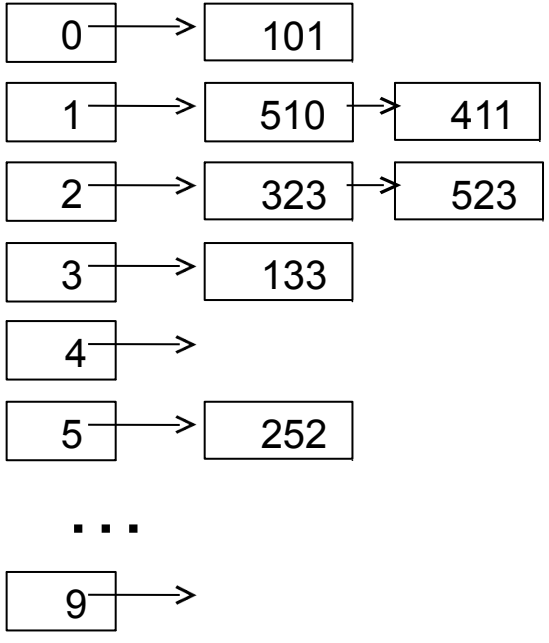




Radixsort – listas encadeadas

- 133
- 252
- 411
- 323
- 510
- 523
- 101





Radixsort - operações

- Número de operações:
  - Dígitos do maior número * N (ordenação)
- Sem listas encadeadas (ordenando por heapsort):

  Dígitos do maior número * N logN
- Dígitos do maior número:

  $\log_{10}K$, onde K é o maior número

Obrigado