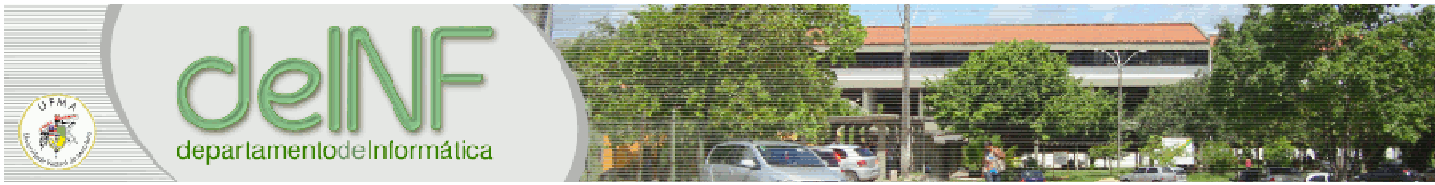
Projeto e Análise de Algoritmos

Aula 1: Algoritmos de Ordenação

Prof. Carlos de Salles

Terças-feiras, 8h20 às 11h10



Algoritmos de Ordenação

- Insertsort
- Mergesort
- Heapsort
- Quicksort

Algoritmos de Ordenação

- Dado um vetor **v** com **N** elementos, a ordenação consiste em organizar todos esses **N** elementos em uma ordem (não-crescente, não-decrescente etc)
- Exemplo:
  - v = {1, 9, 8, 5, 3, 7, 4}
    - Depois de ordenado de forma não-decrescente, v fica:
    - v = {1, 3, 4, 5, 7, 8, 9}

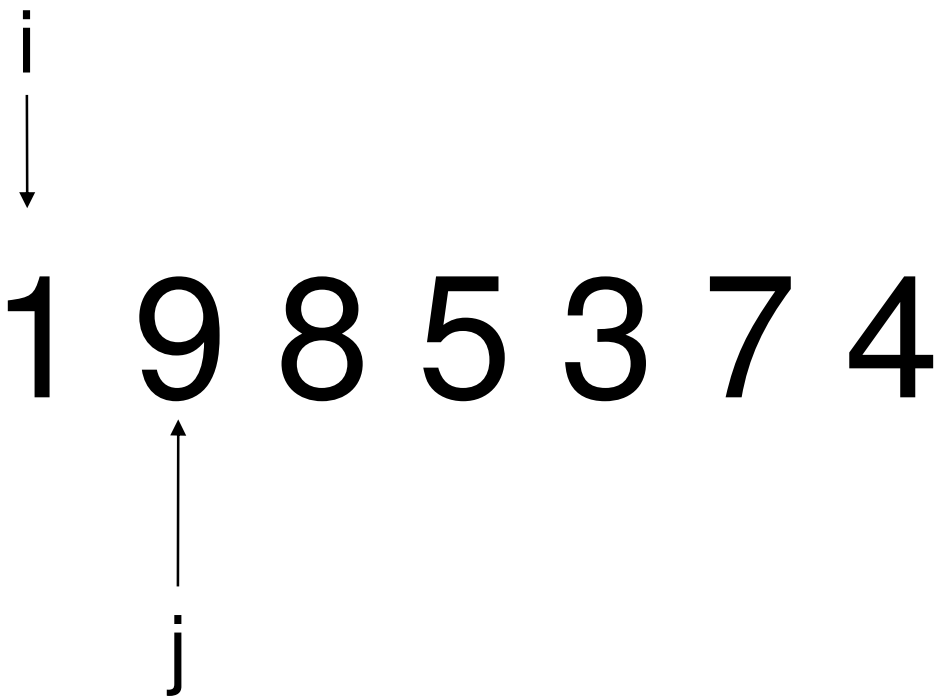
Insert Sort

- Outros nomes:
  - Bubble sort
  - Selection sort
- Consite em posicionar o menor ou o maior elemento em seu lugar correto **N** vezes consecutivas

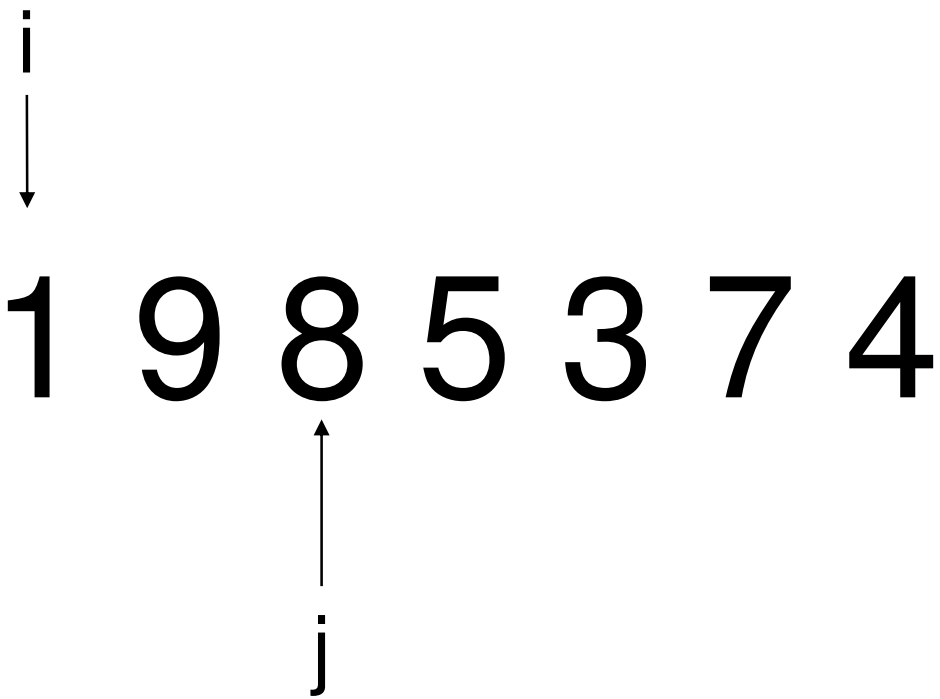
Insert Sort

```
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
```

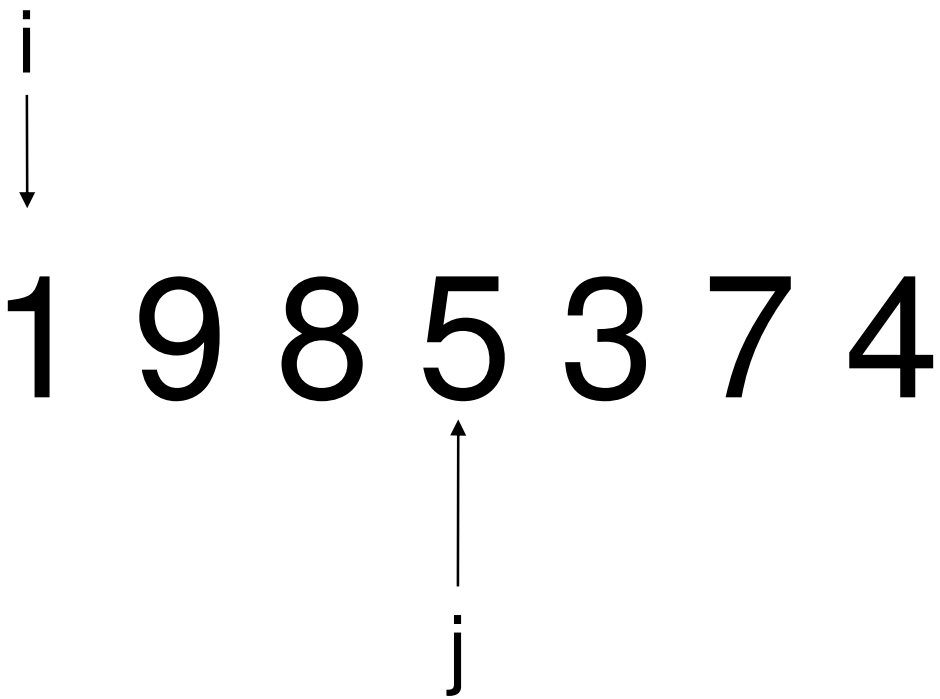
Insert Sort



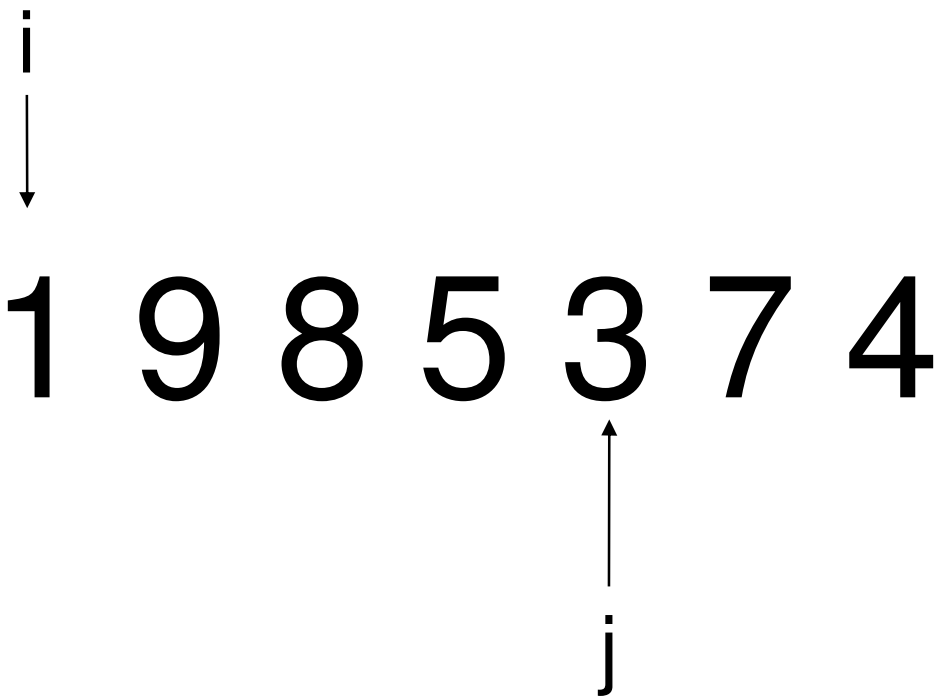
Insert Sort



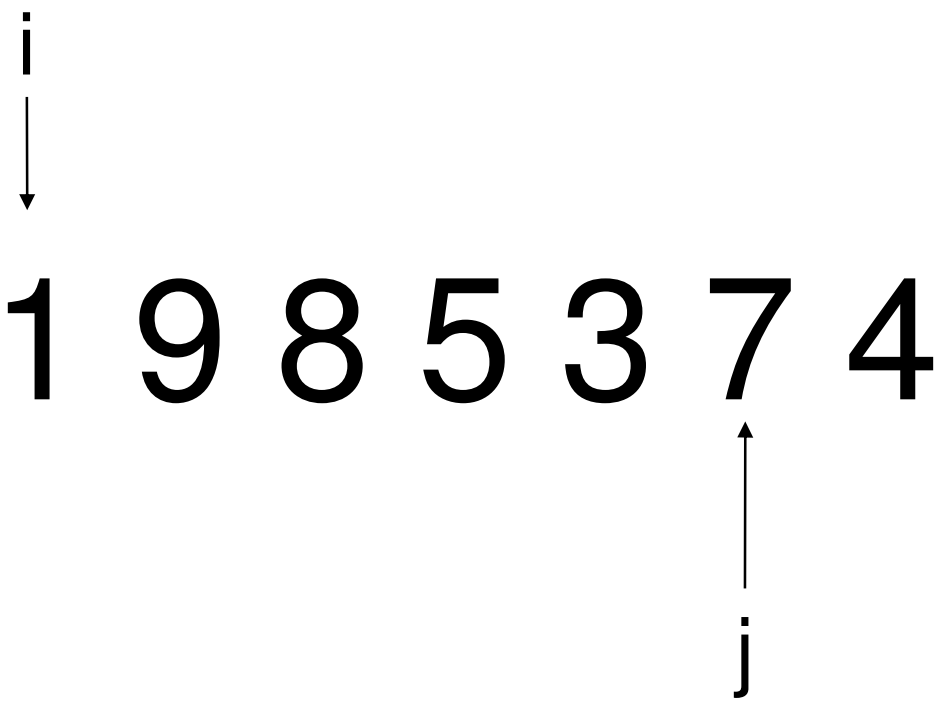
Insert Sort



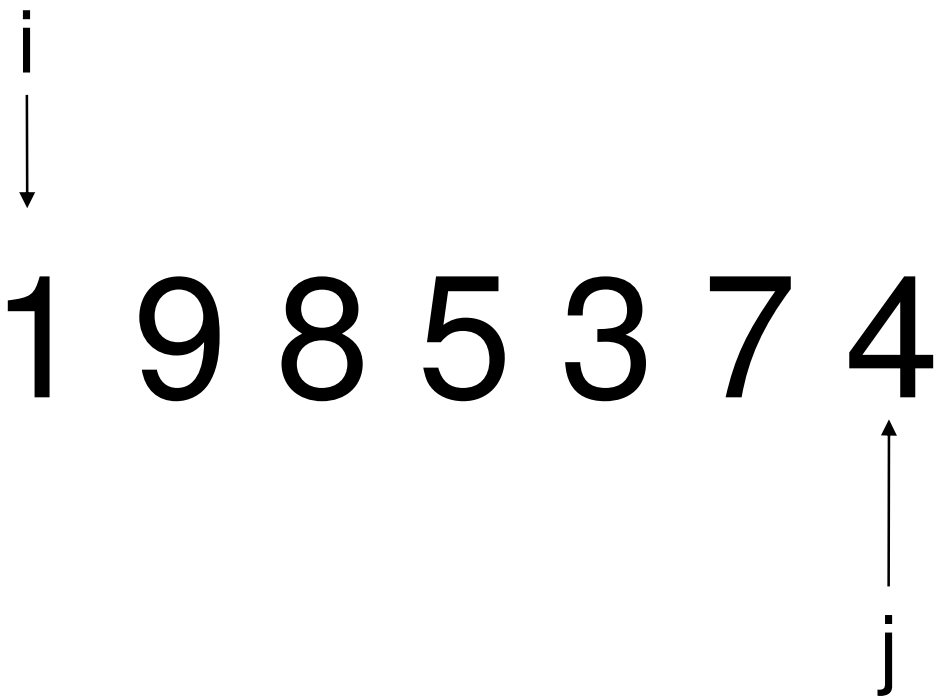
Insert Sort



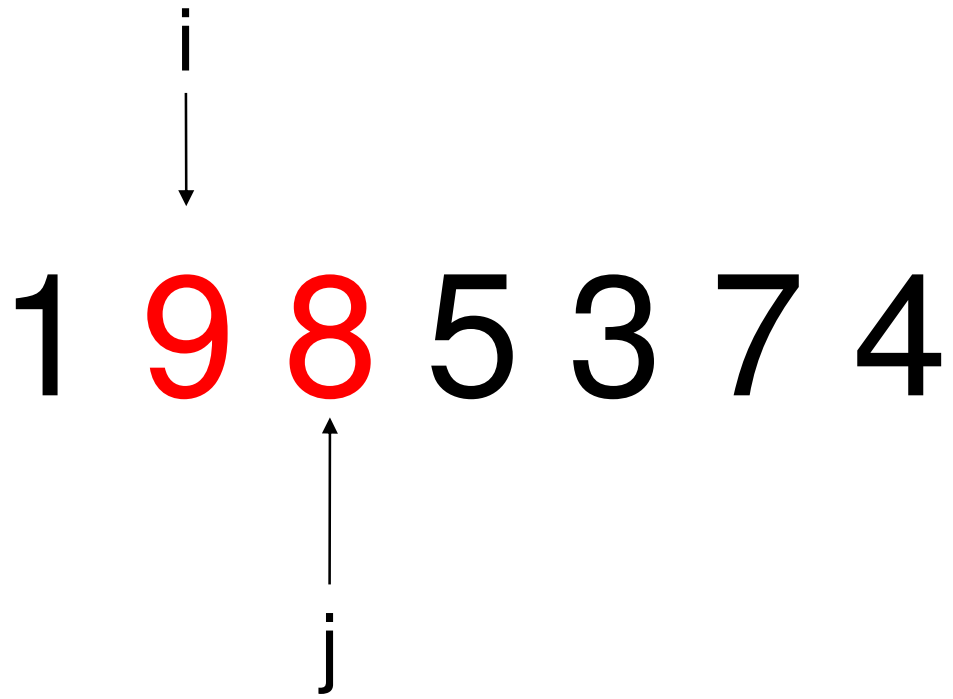
Insert Sort



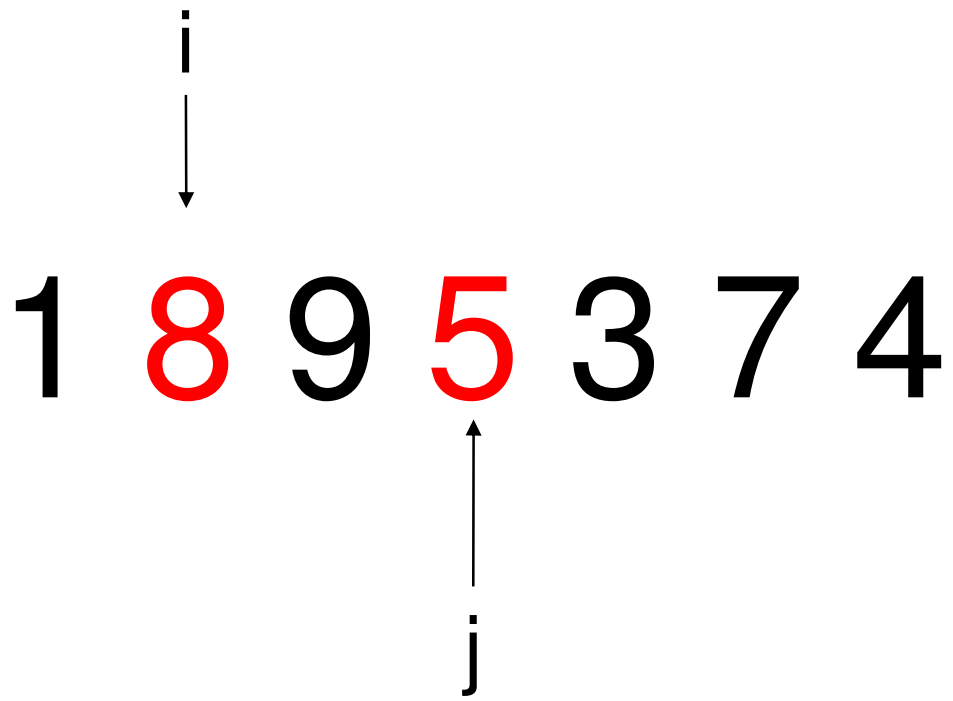
Insert Sort



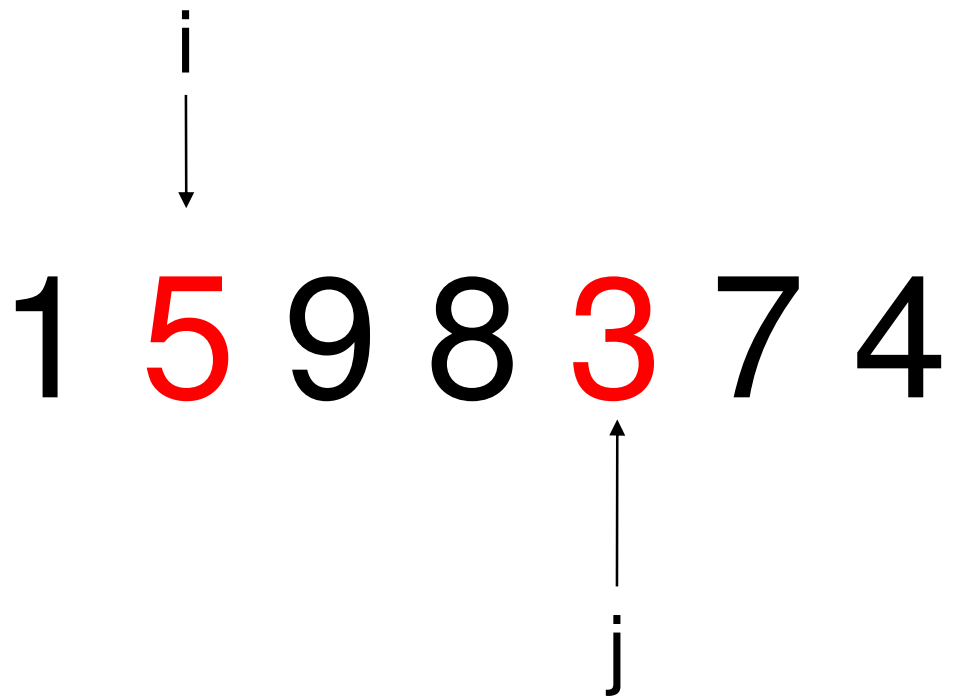
Insert Sort



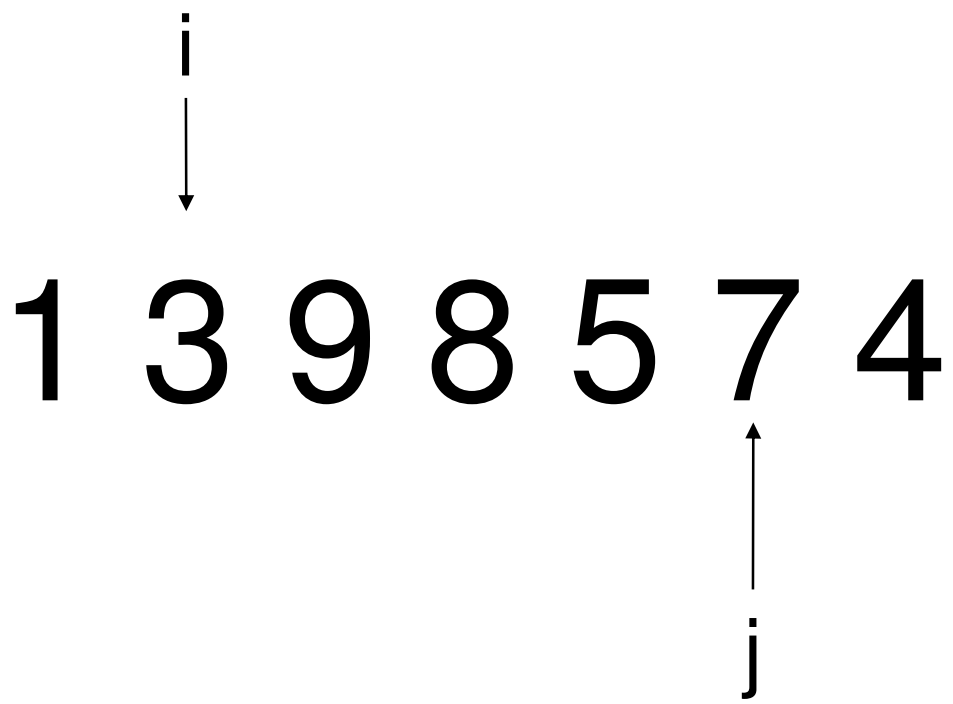
Insert Sort



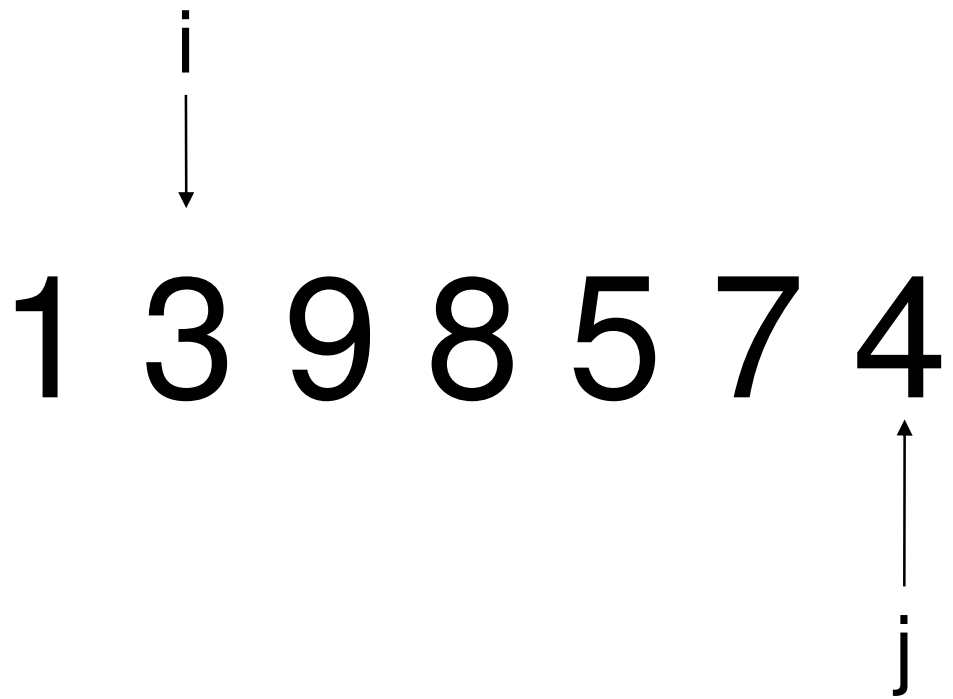
Insert Sort



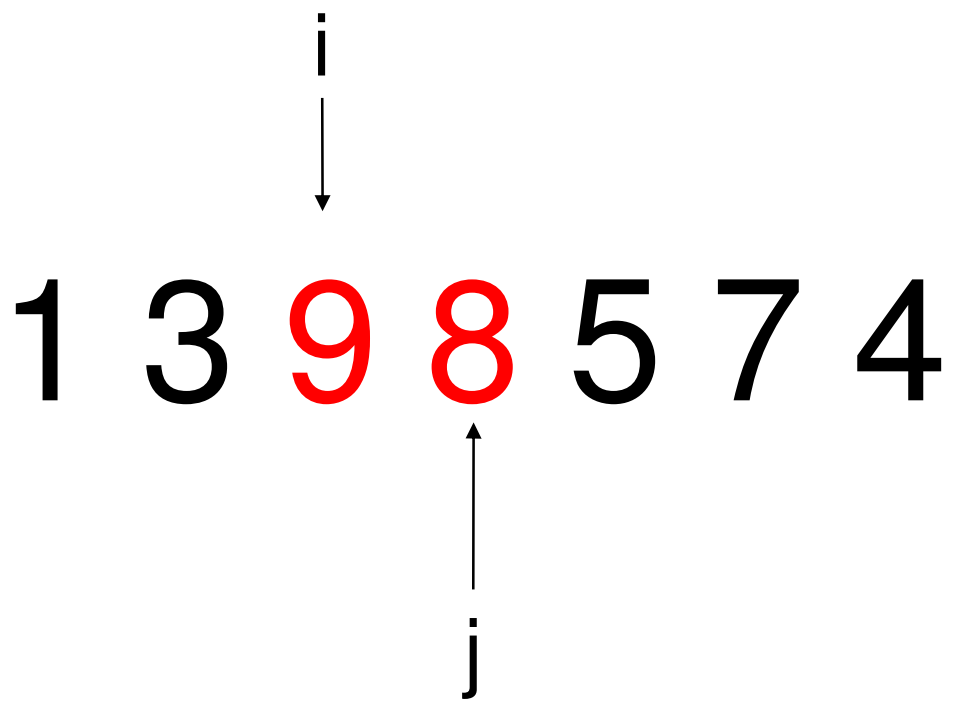
Insert Sort



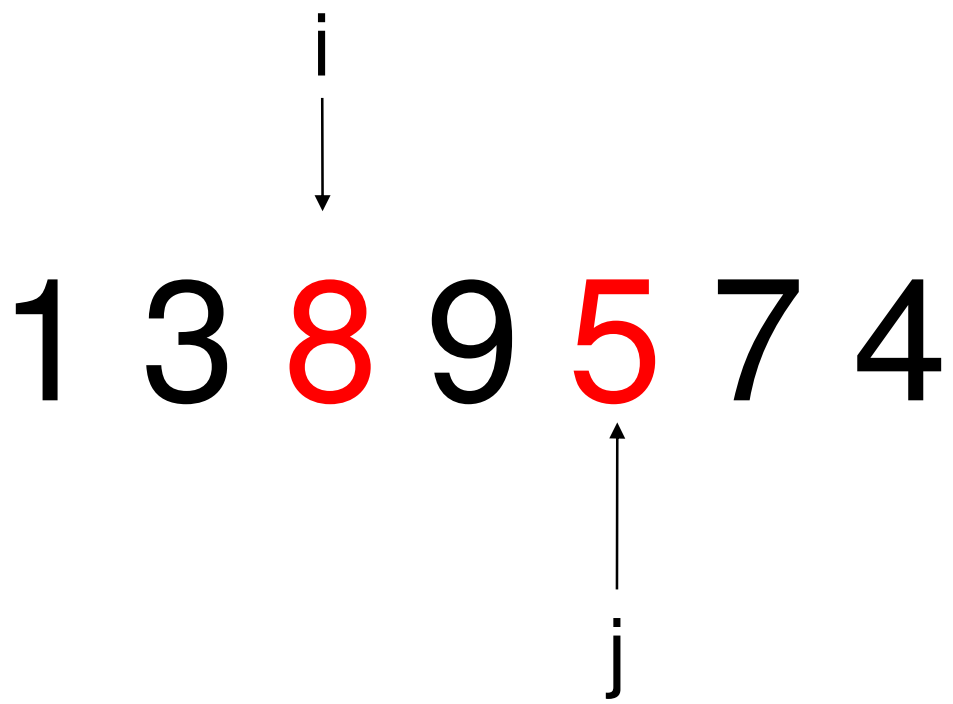
Insert Sort



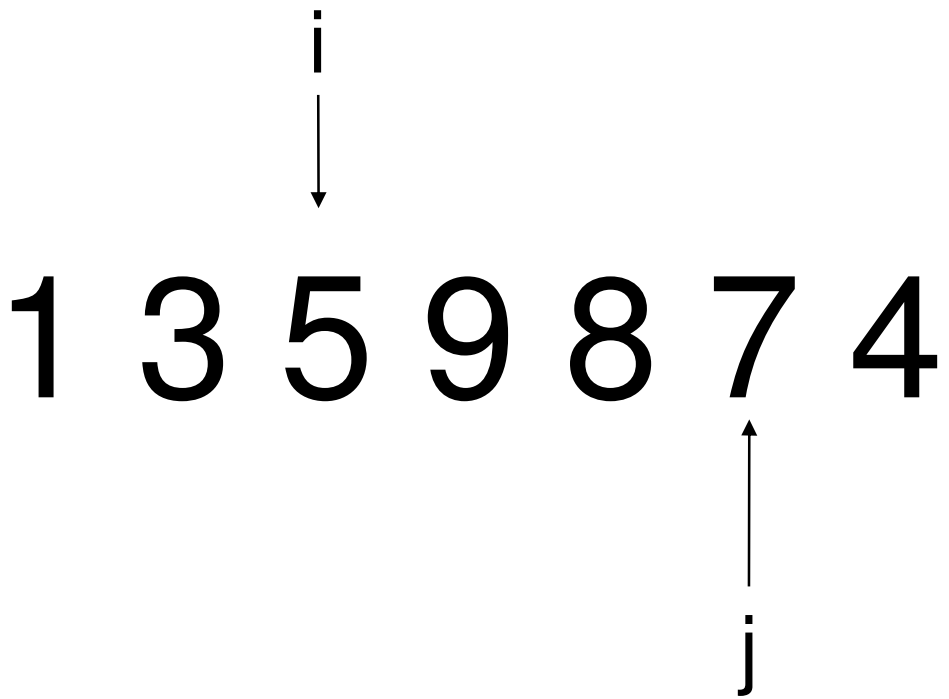
Insert Sort



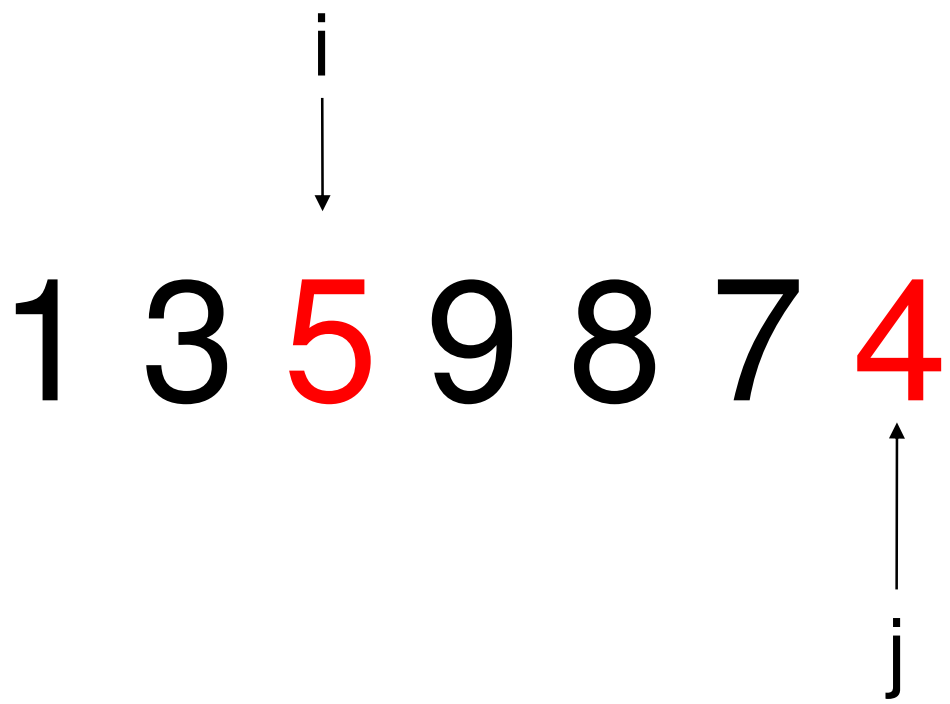
Insert Sort



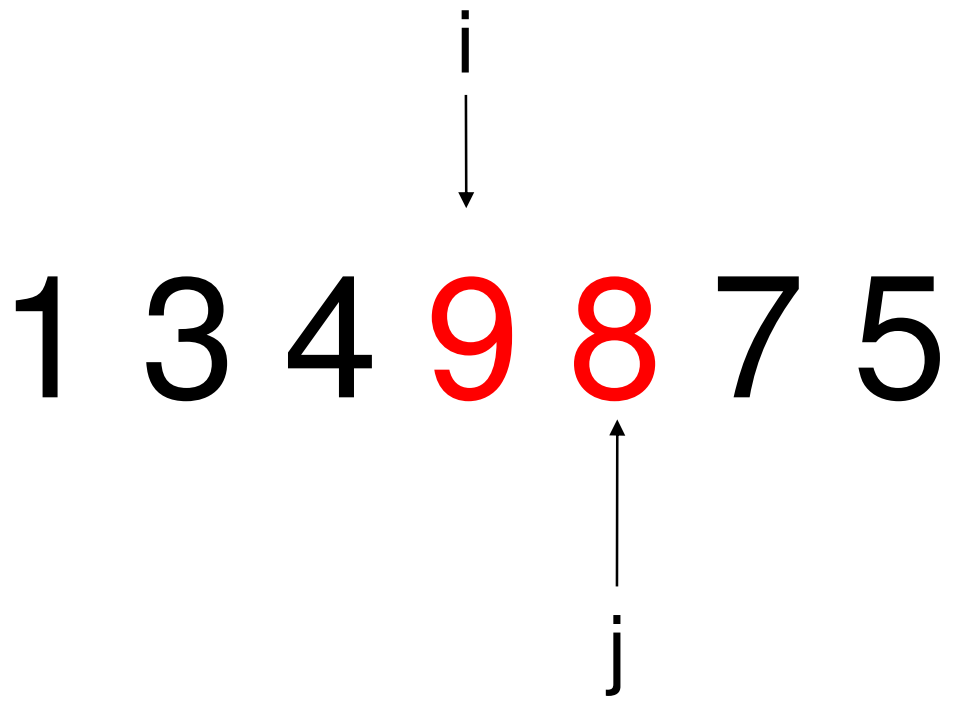
Insert Sort



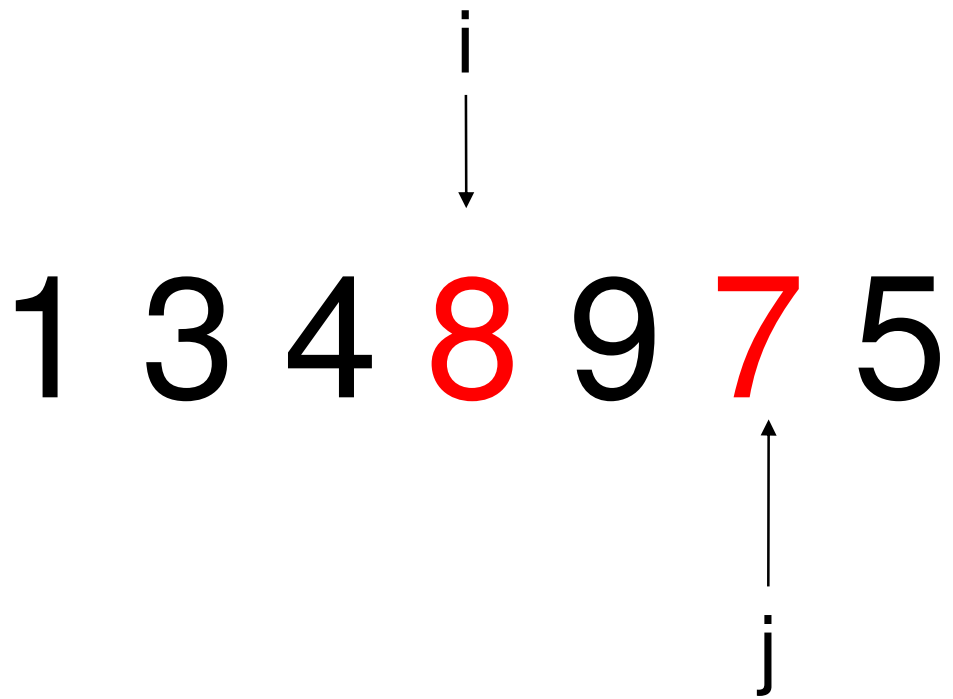
Insert Sort



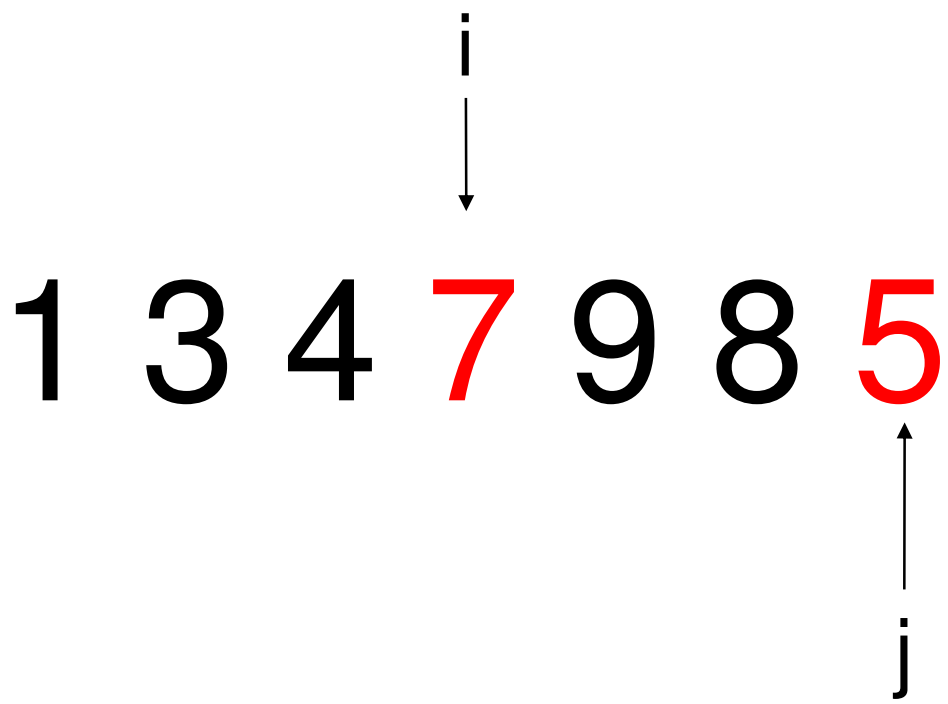
Insert Sort



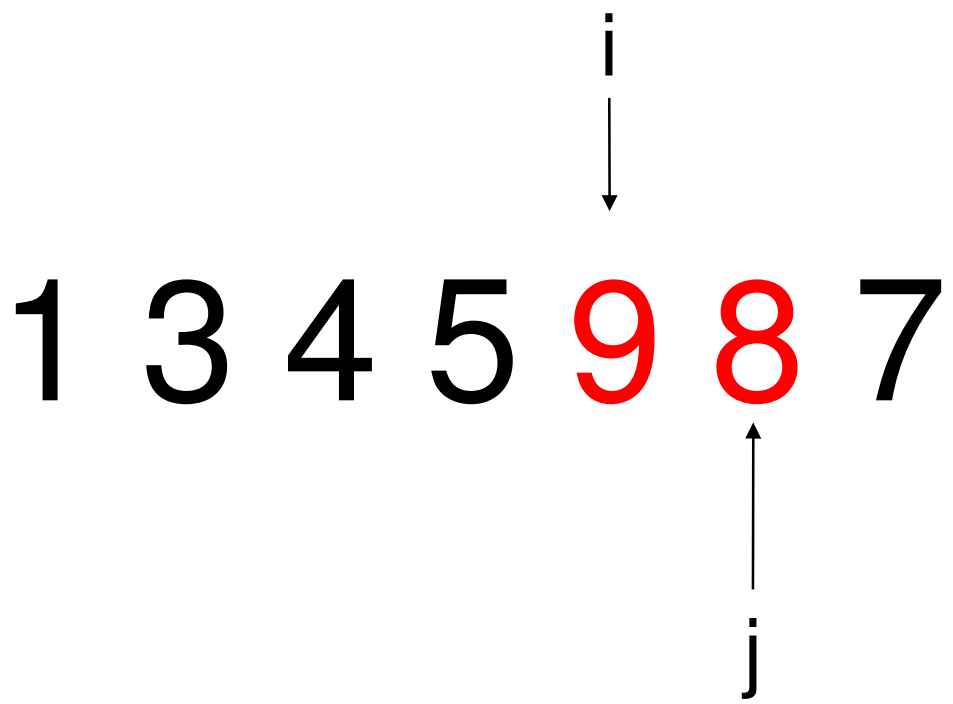
Insert Sort



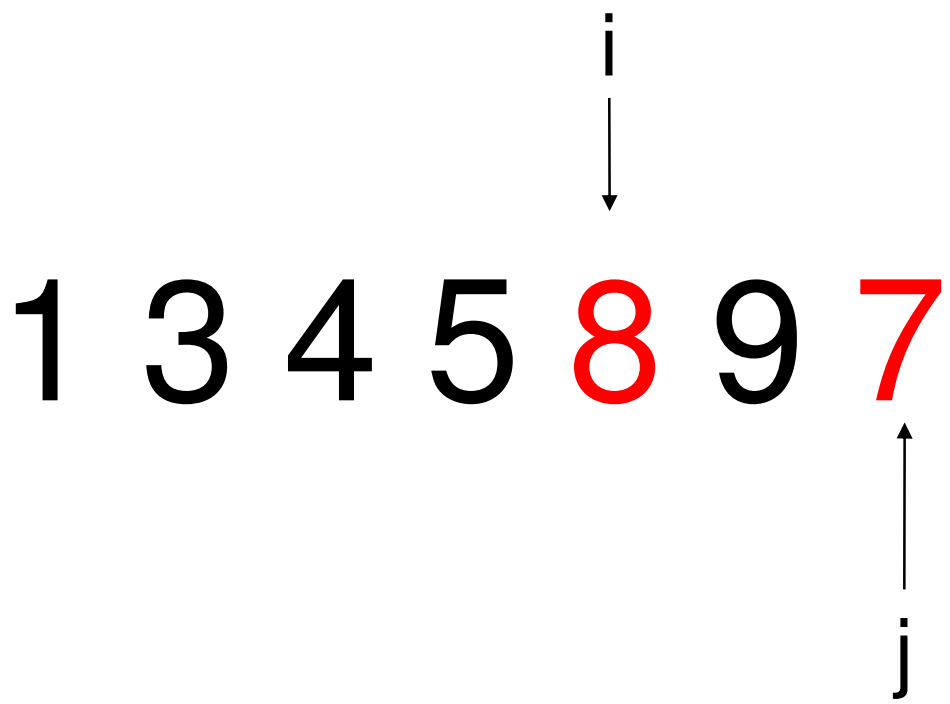
Insert Sort



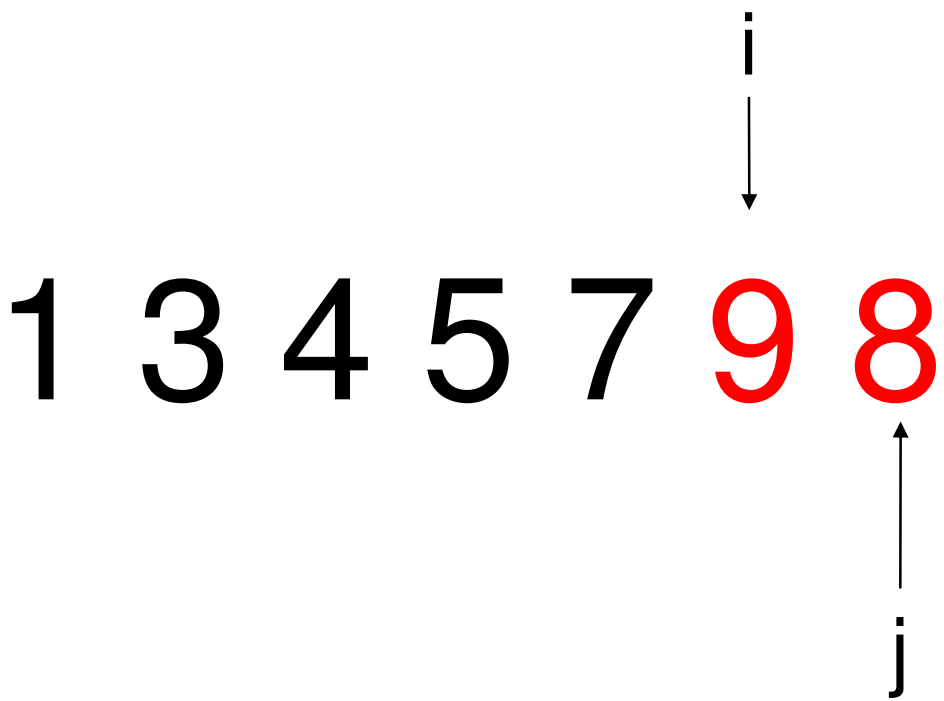
Insert Sort



Insert Sort



Insert Sort



Insert Sort

1 3 4 5 7 8 9

Insert Sort

1 3 4 5 7 8 9

Número de operações:
(N-1) + (N-2) + (N-3) + … + 3 + 2 + 1

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

Insert Sort

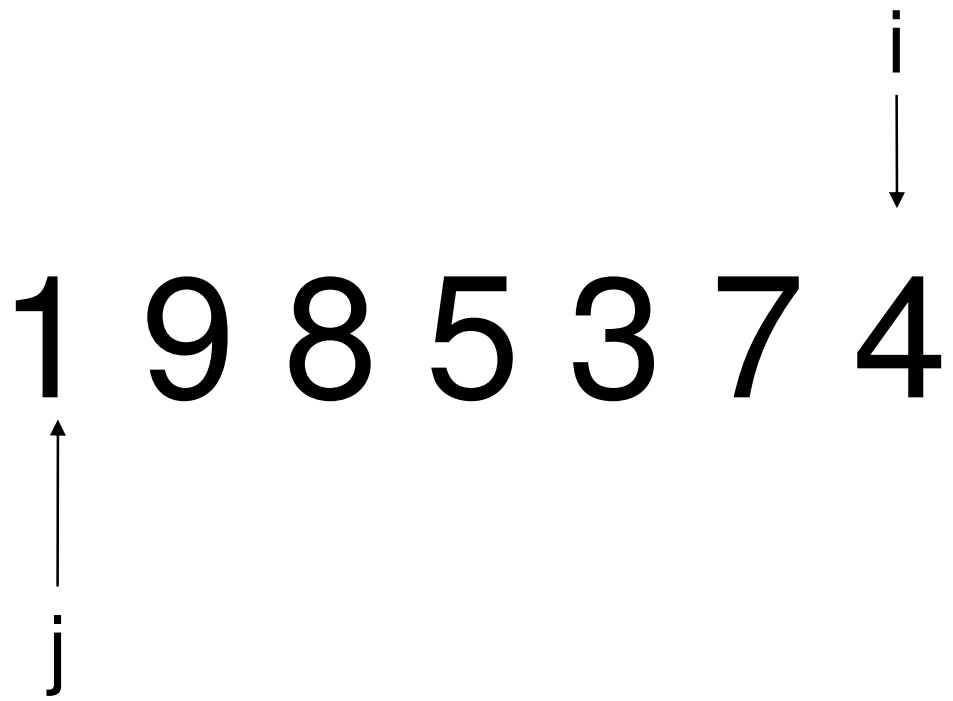
- Ideia:
  - Se o vetor já estiver ordenado em alguma interação, seria interessante detectar isso e parar o algoritmo.
- Como detecto se um vetor está ordenado?

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

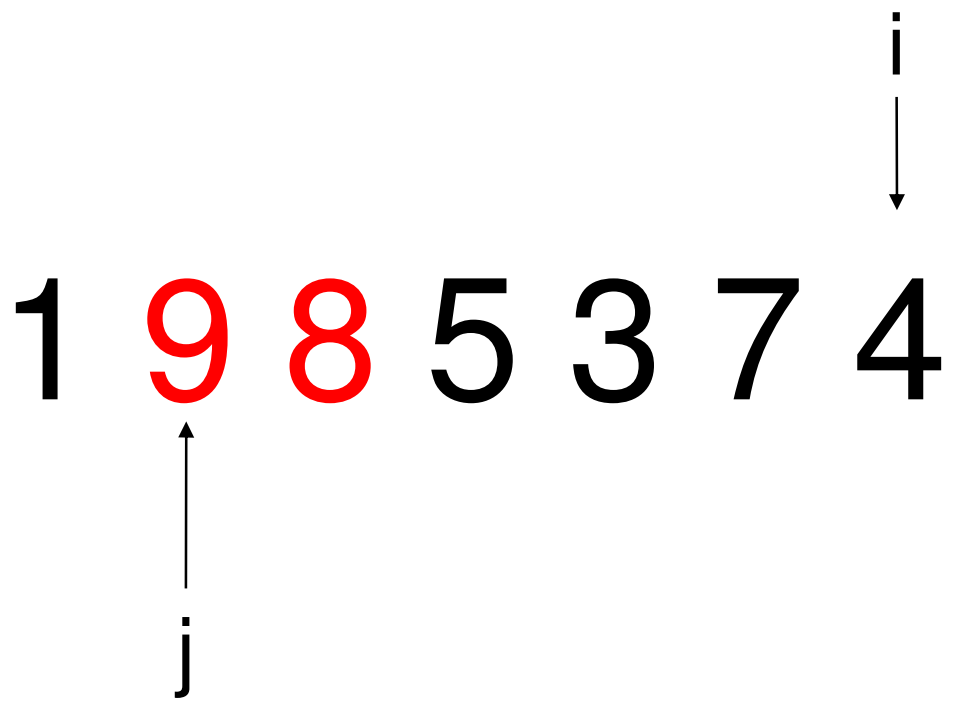
Insert Sort (melhoria)

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

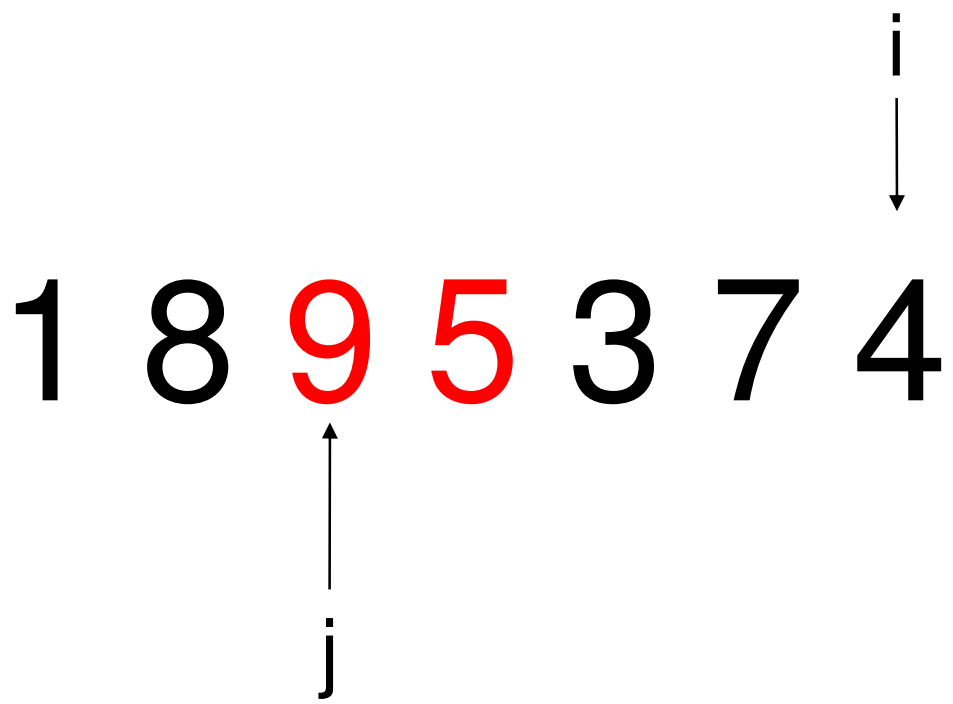
Insert Sort (melhoria)



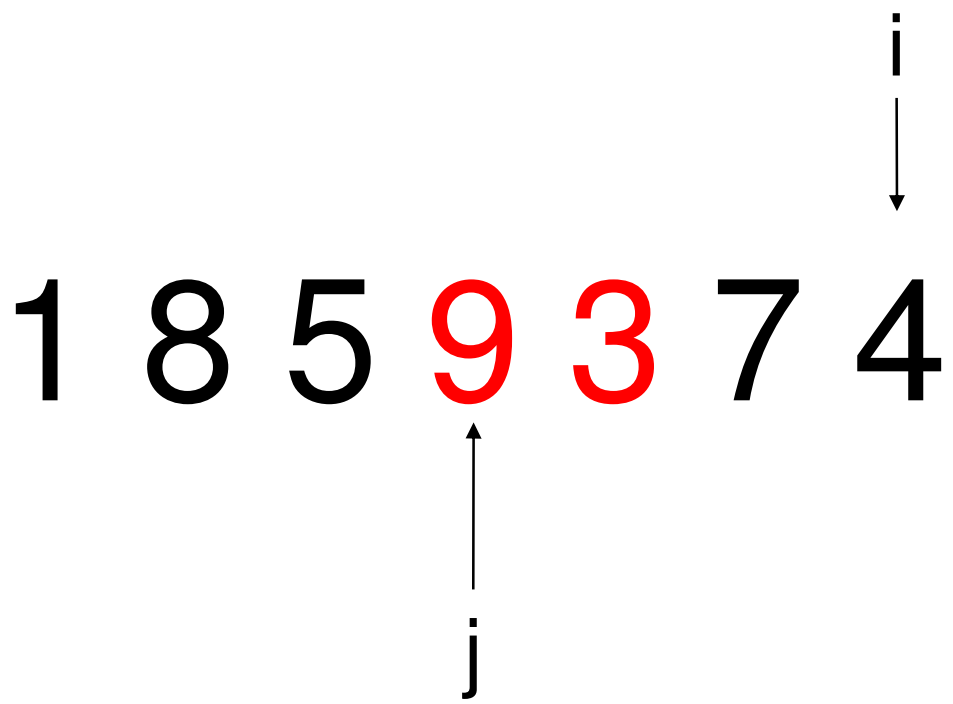
Insert Sort (melhoria)



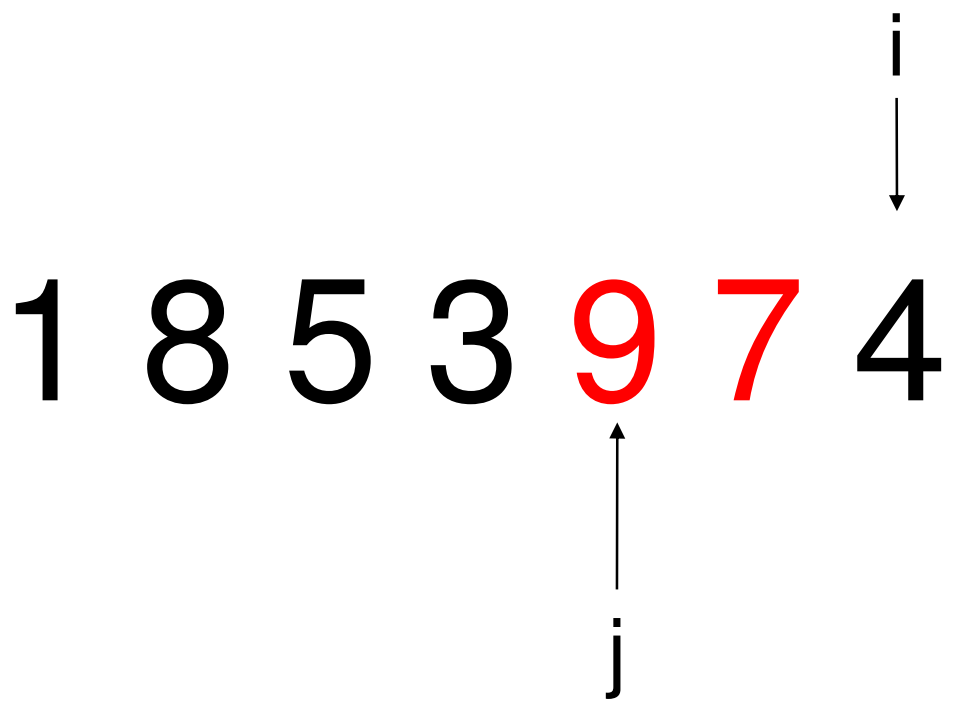
Insert Sort (melhoria)



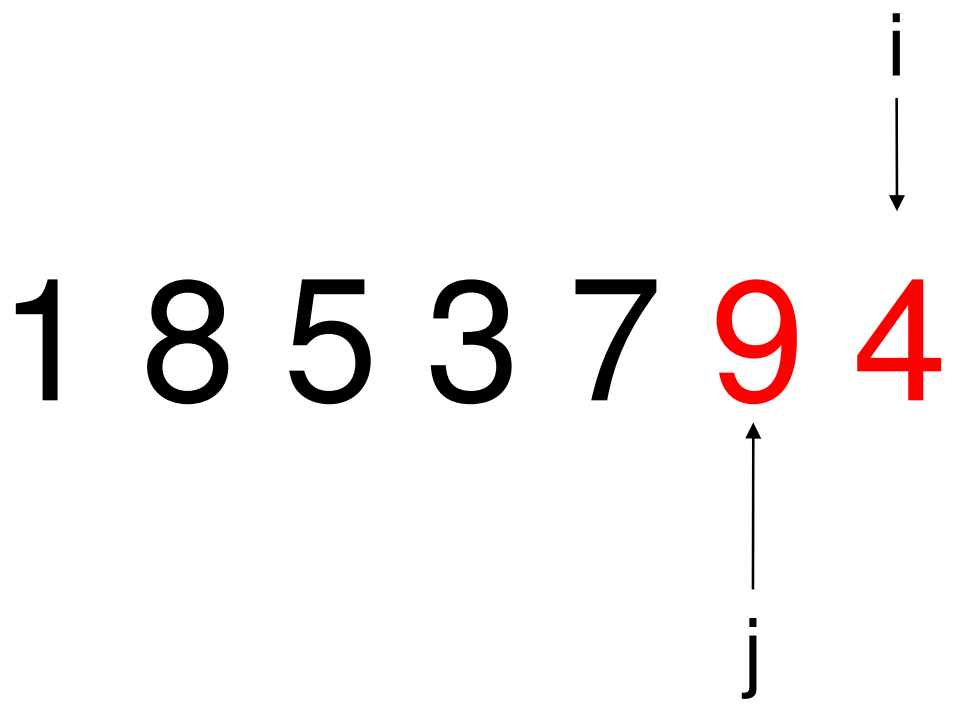
Insert Sort (melhoria)



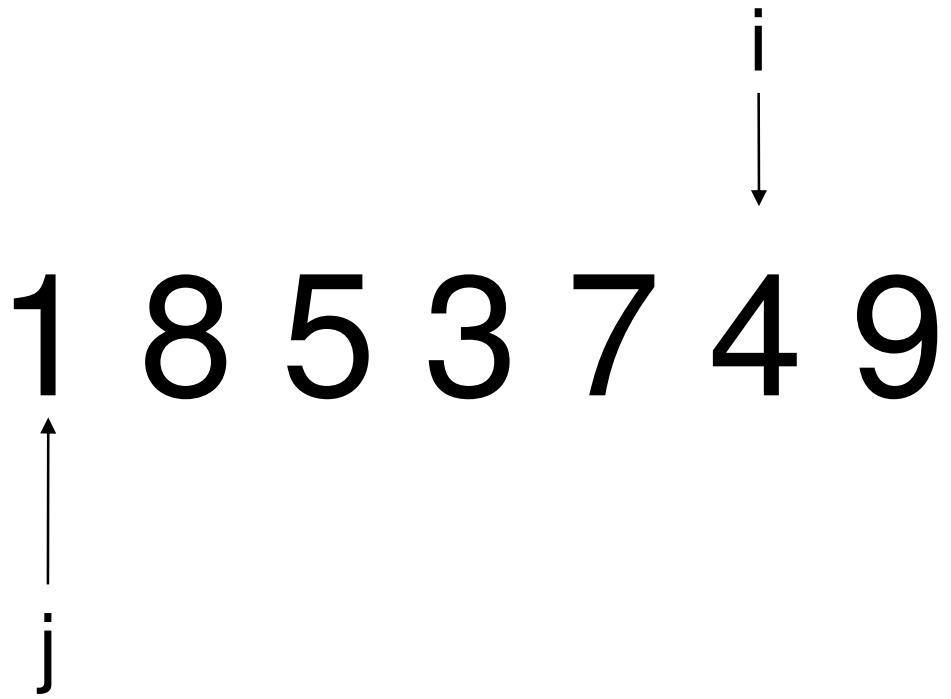
Insert Sort (melhoria)



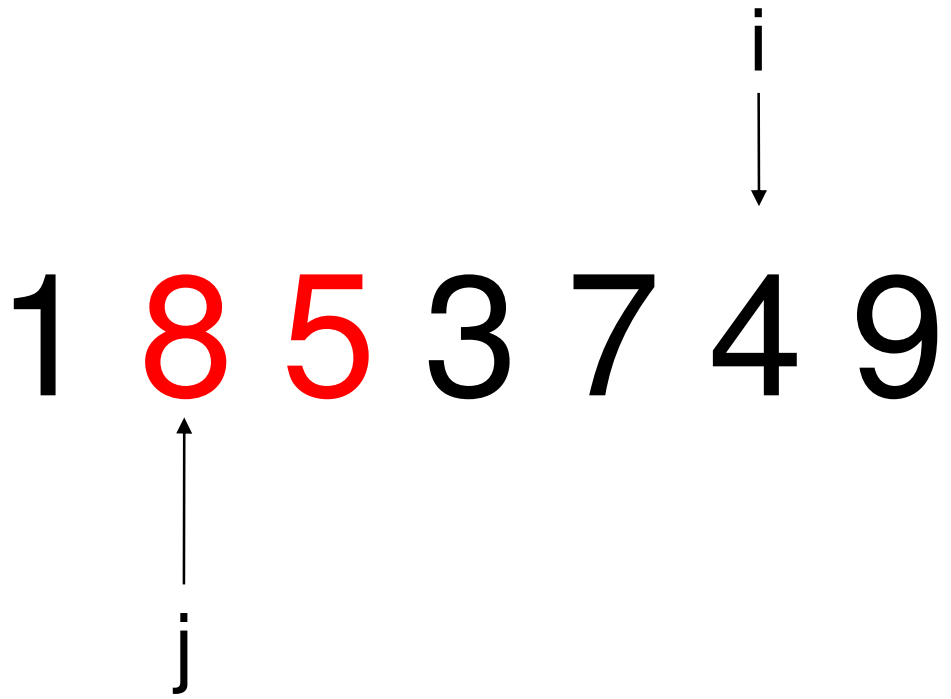
Insert Sort (melhoria)



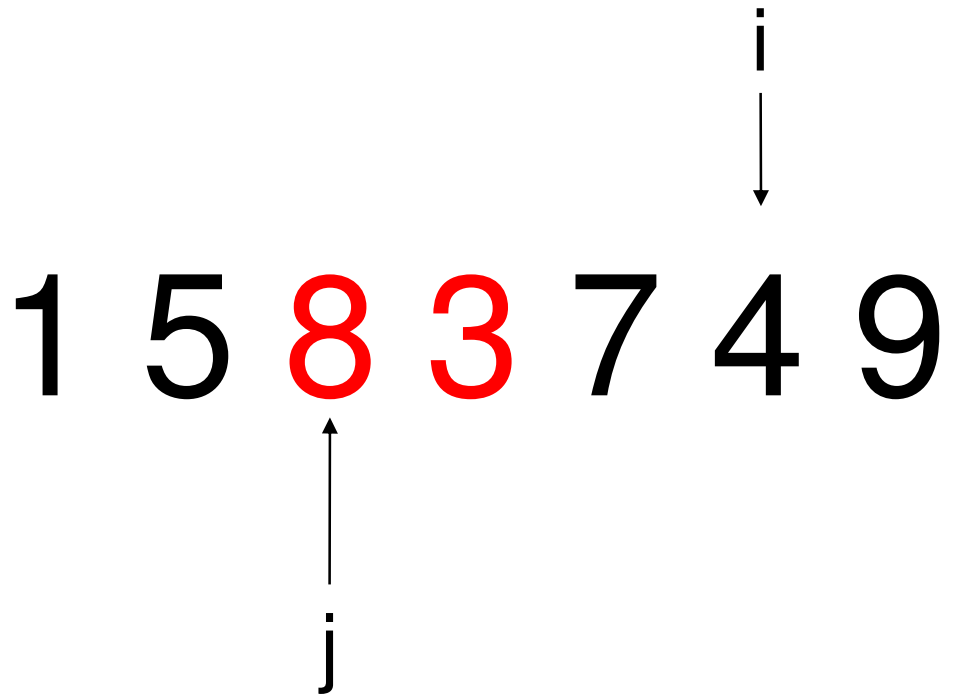
Insert Sort (melhoria)



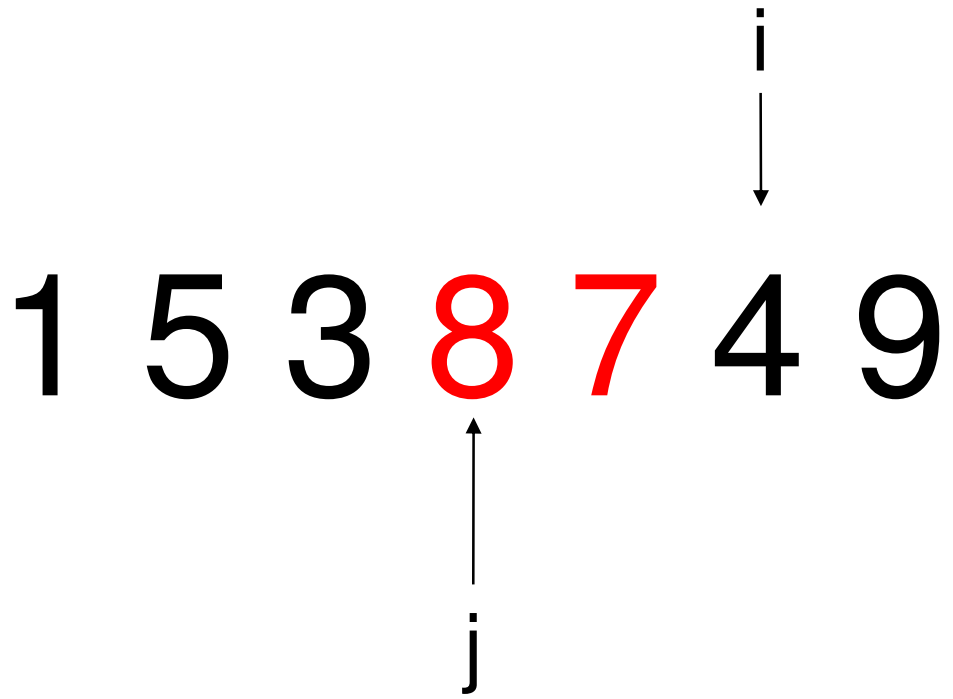
Insert Sort (melhoria)



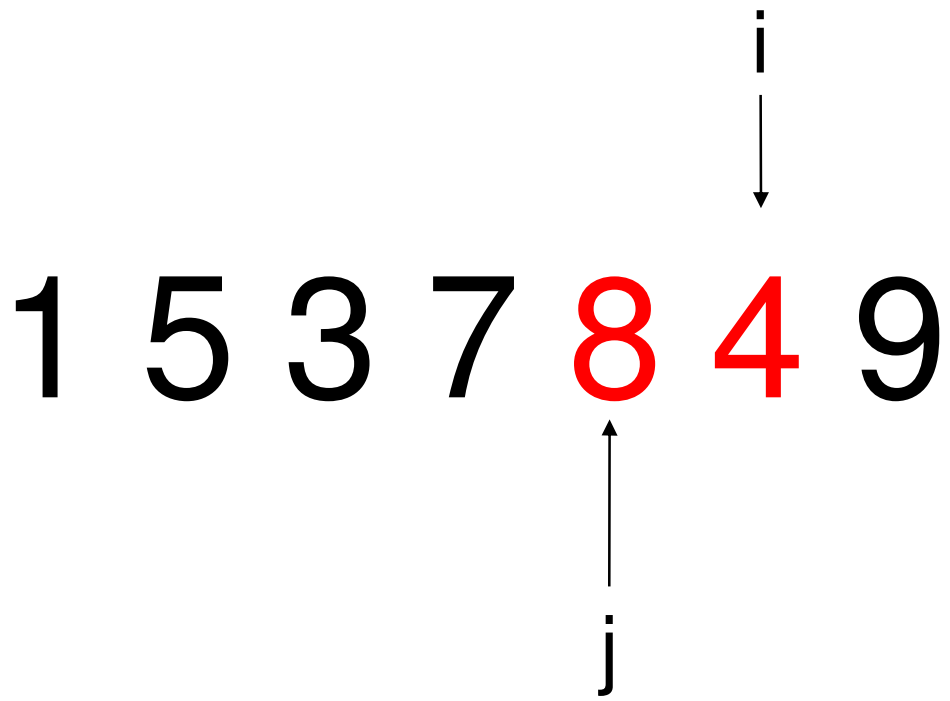
Insert Sort (melhoria)



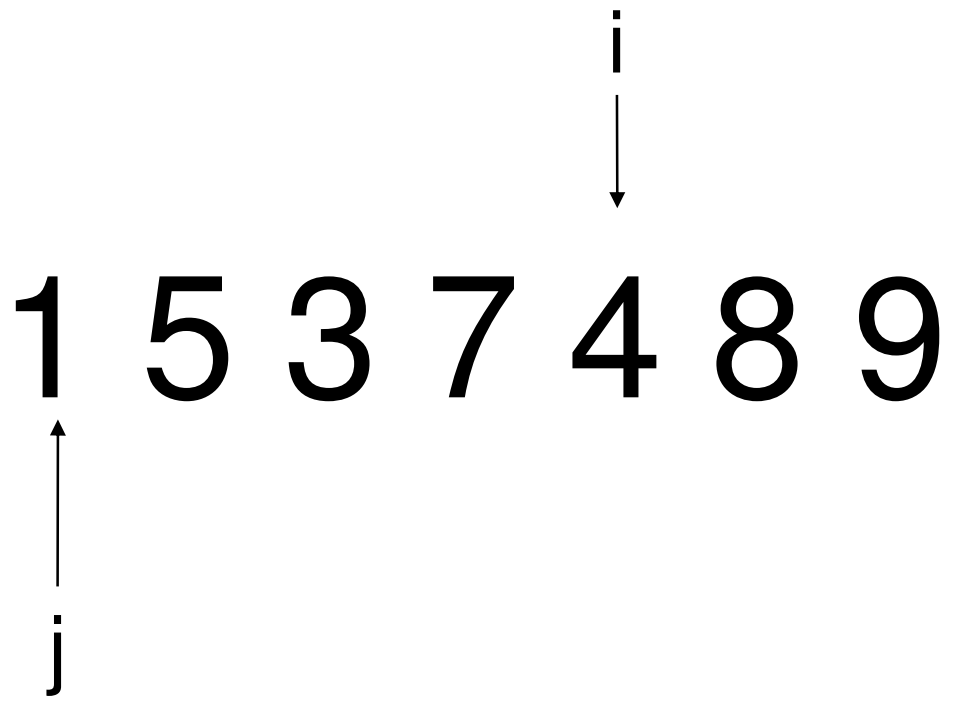
Insert Sort (melhoria)



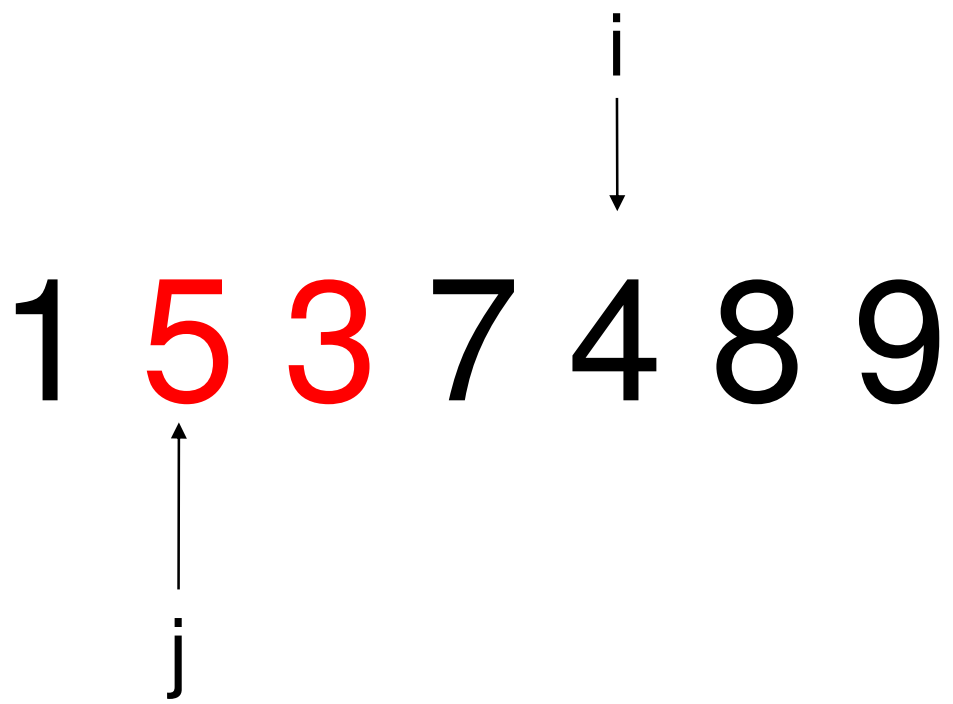
Insert Sort (melhoria)



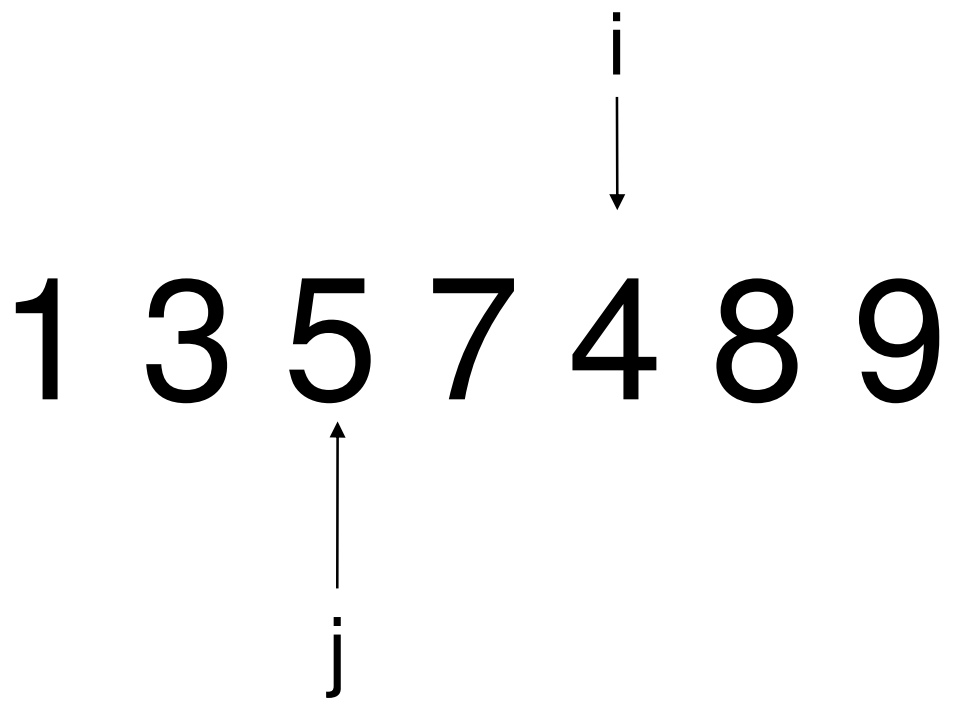
Insert Sort (melhoria)



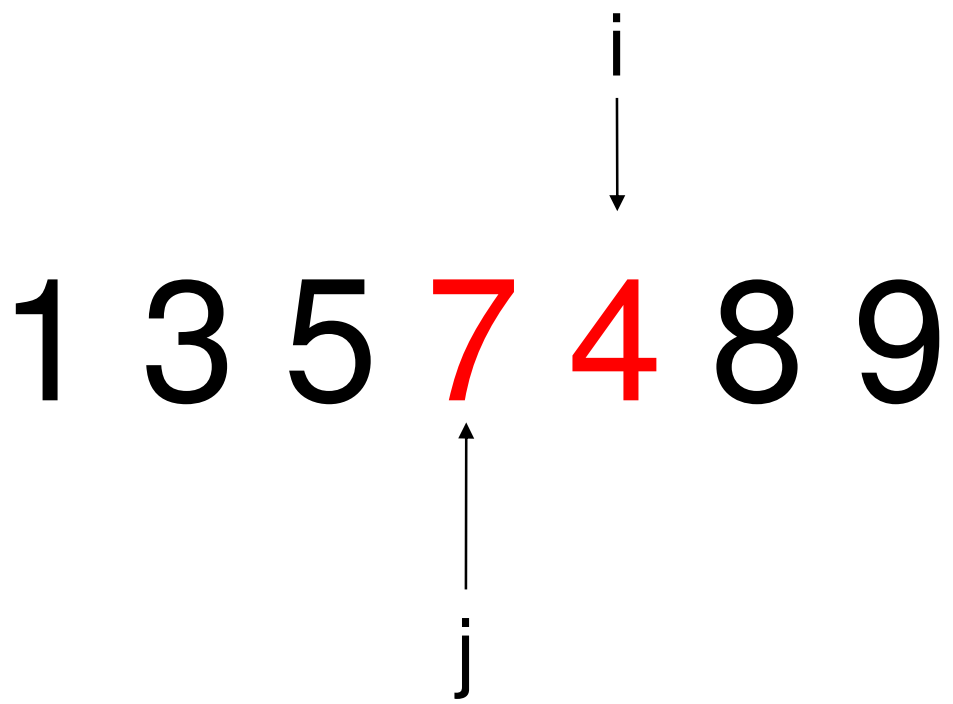
Insert Sort (melhoria)



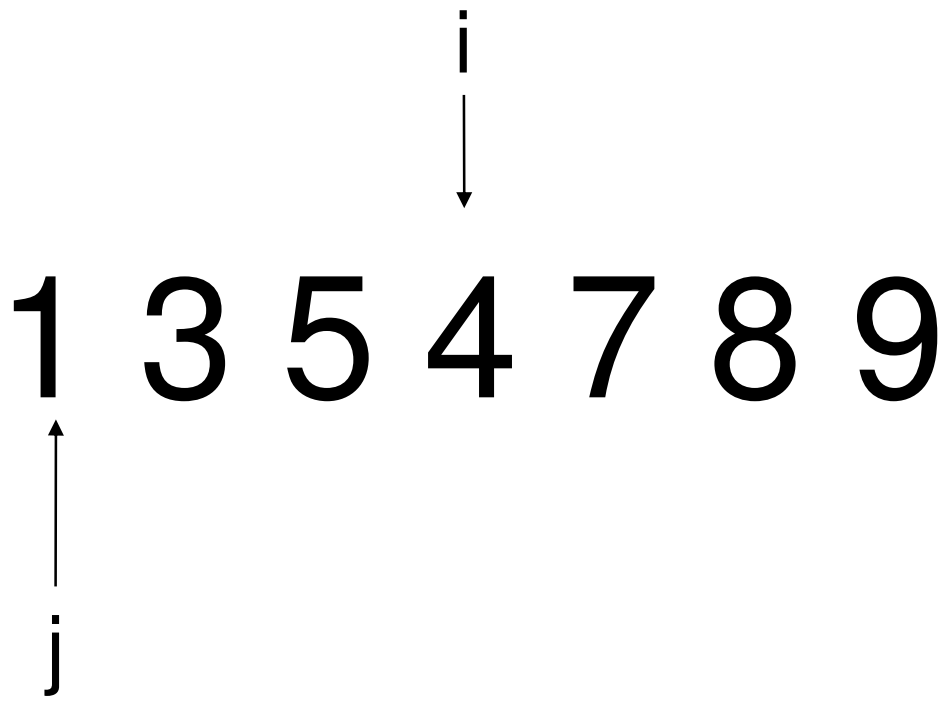
Insert Sort (melhoria)



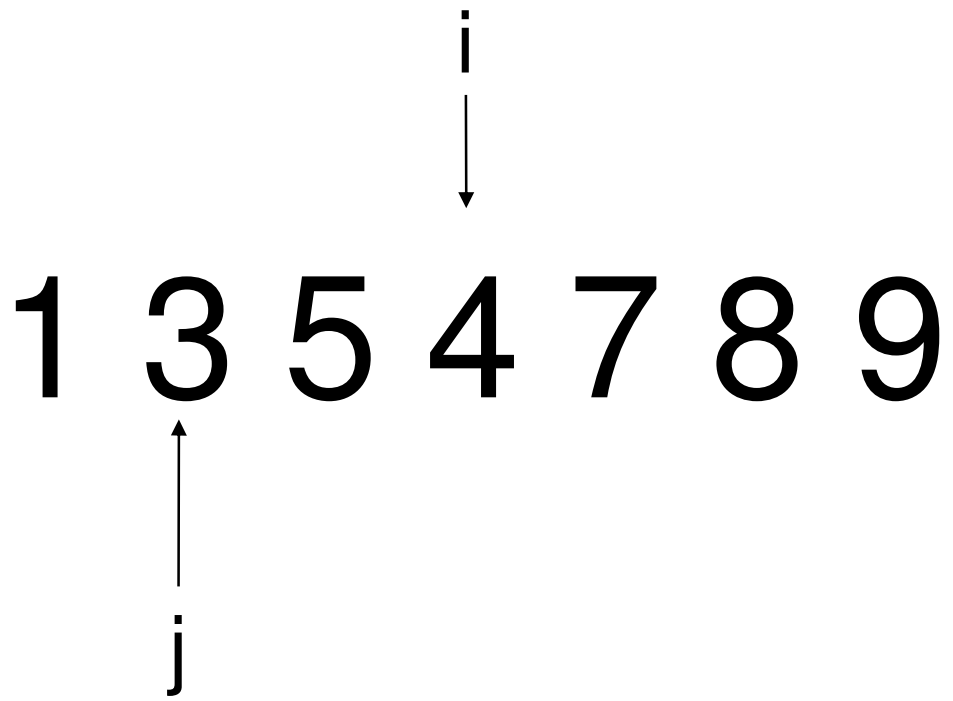
Insert Sort (melhoria)



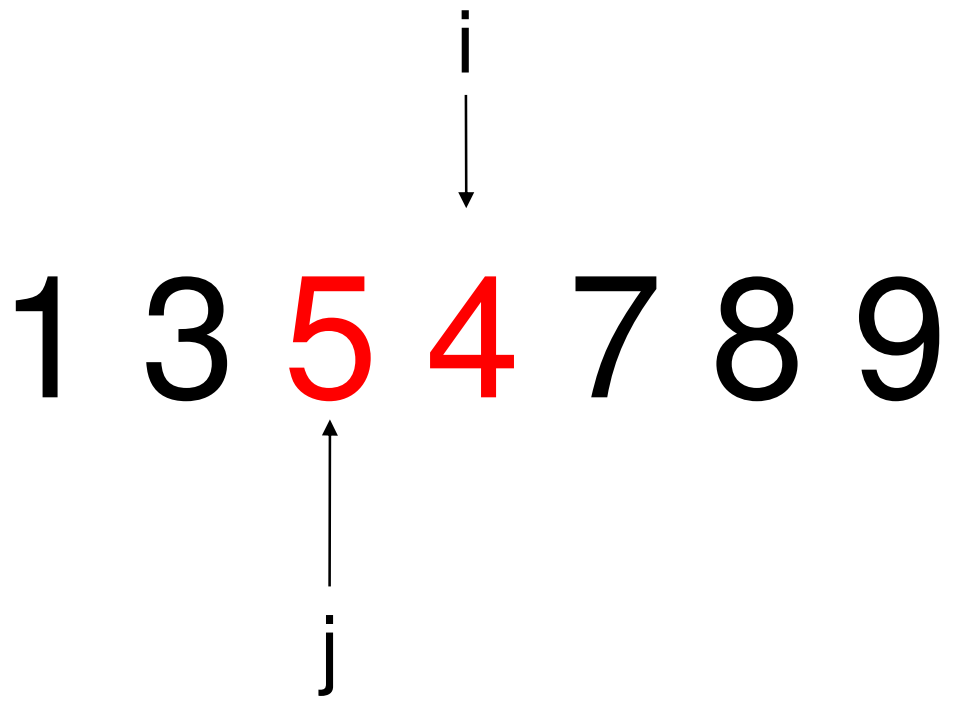
Insert Sort (melhoria)



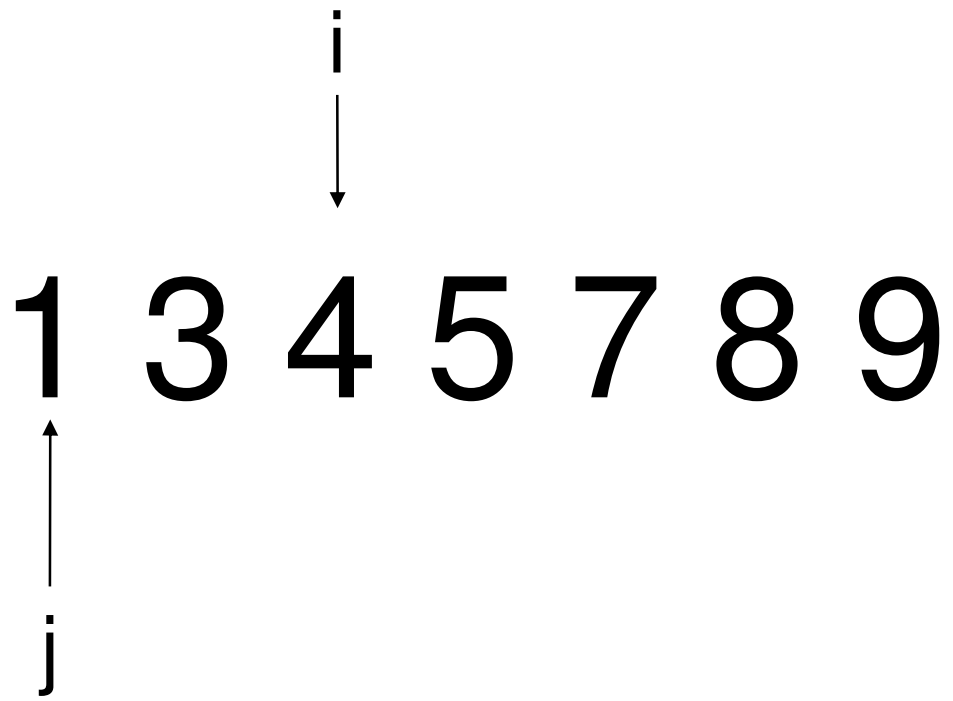
Insert Sort (melhoria)



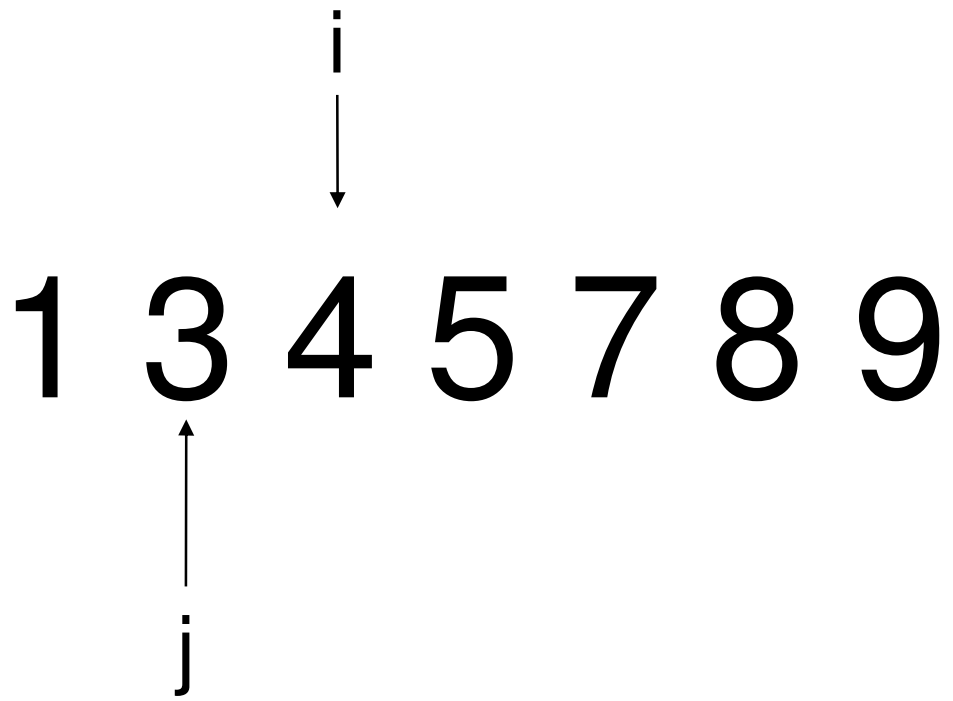
Insert Sort (melhoria)



Insert Sort (melhoria)



Insert Sort (melhoria)



Insert Sort (melhoria)

1 3 4 5 7 8 9

Mergesort

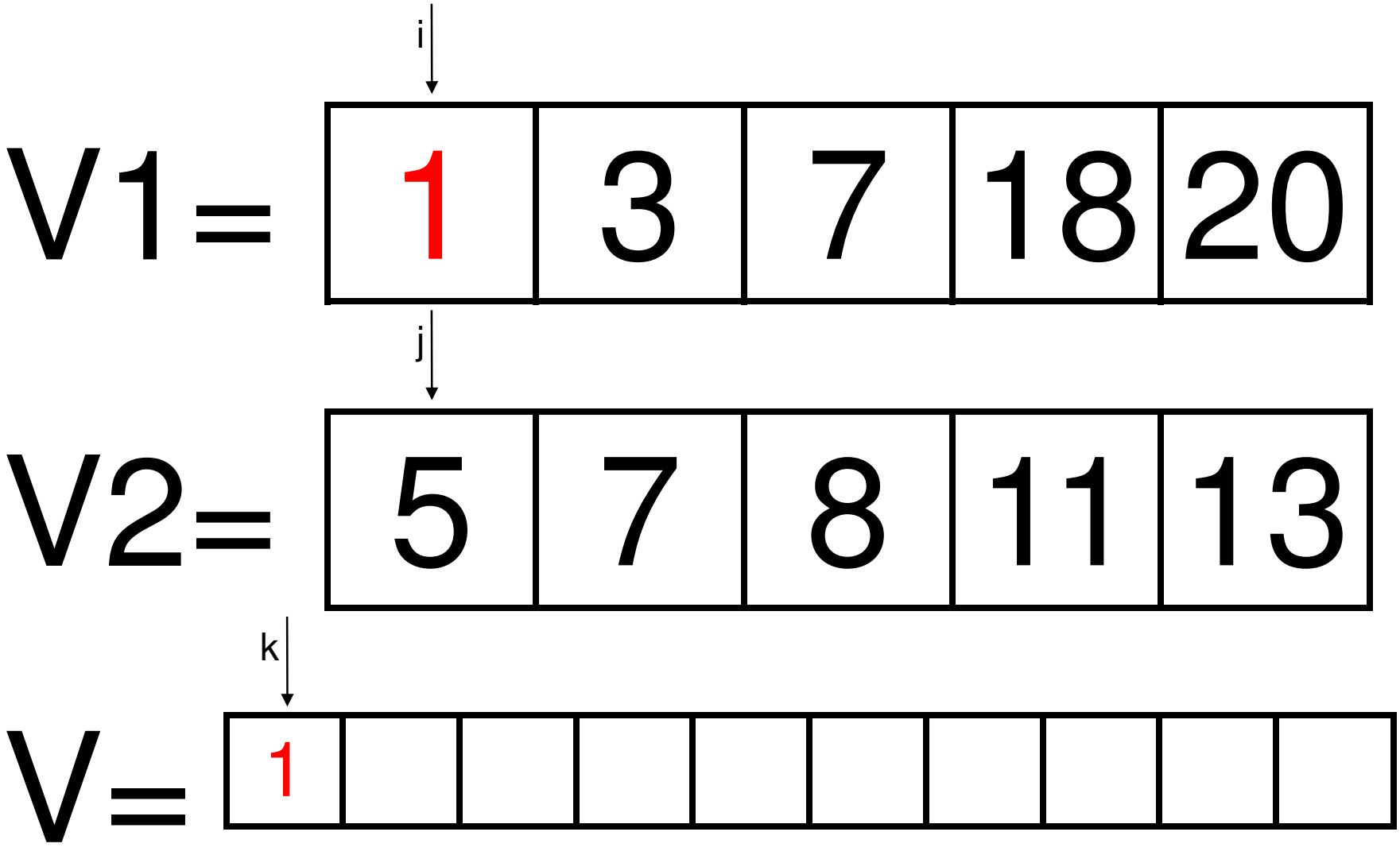
- Ordena inicialmente sub-vetores de tamanho 1 ou 2 e recursivamente dobra esse tamanho até ordenar o vetor total de tamanho N
- Baseia-se no merge de dois vetores ordenados

Merge de dois vetores ordenados

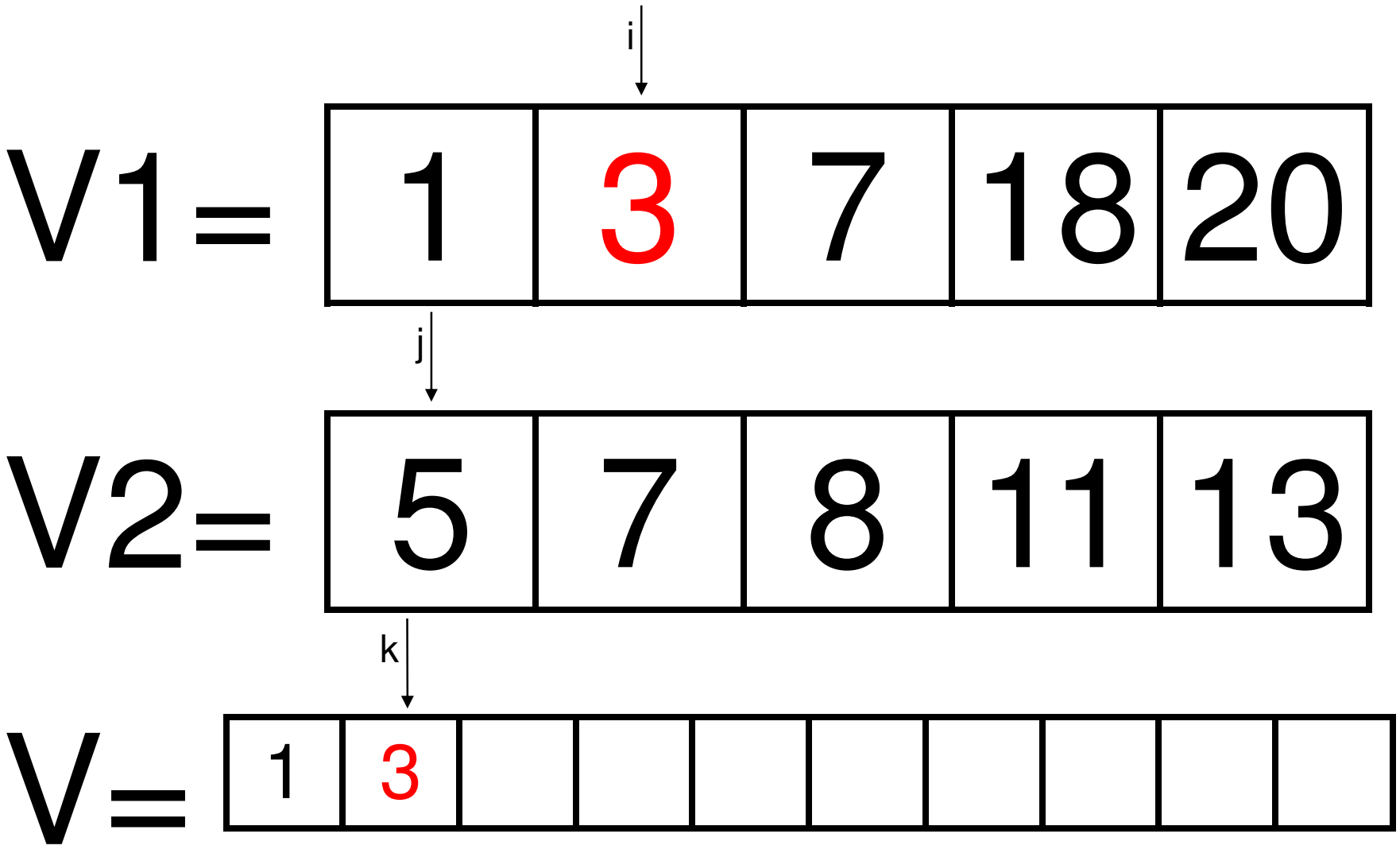
- Dados dois vetores V1 e V2, já ordenados, o **merge** cria um outro vetor V ordenado com os elementos de V1 e V2

```
local i,j,k = 1,1,1
for k=1,#v1+#v2 do
   if j>#v2 then v[k]=v1[i]; i=i+1
   else if i>#v1 then v[k]=v2[j]; j=j+1
   else if v1[i]<=v2[j] then v[k]=v1[i]; i=i+1
   else v[k]=v2[j]; j=j+1 end
end
```

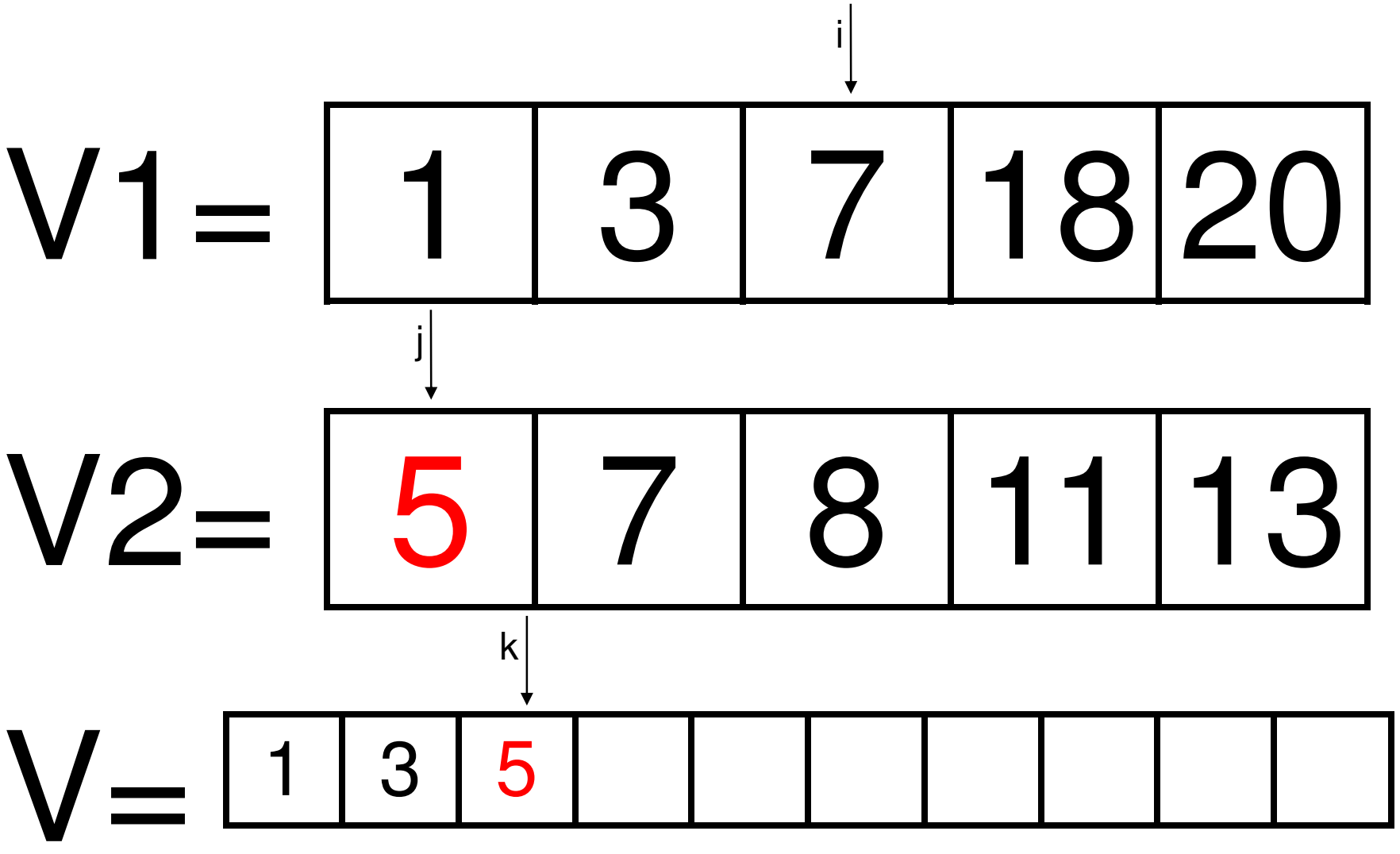
Merge de dois vetores



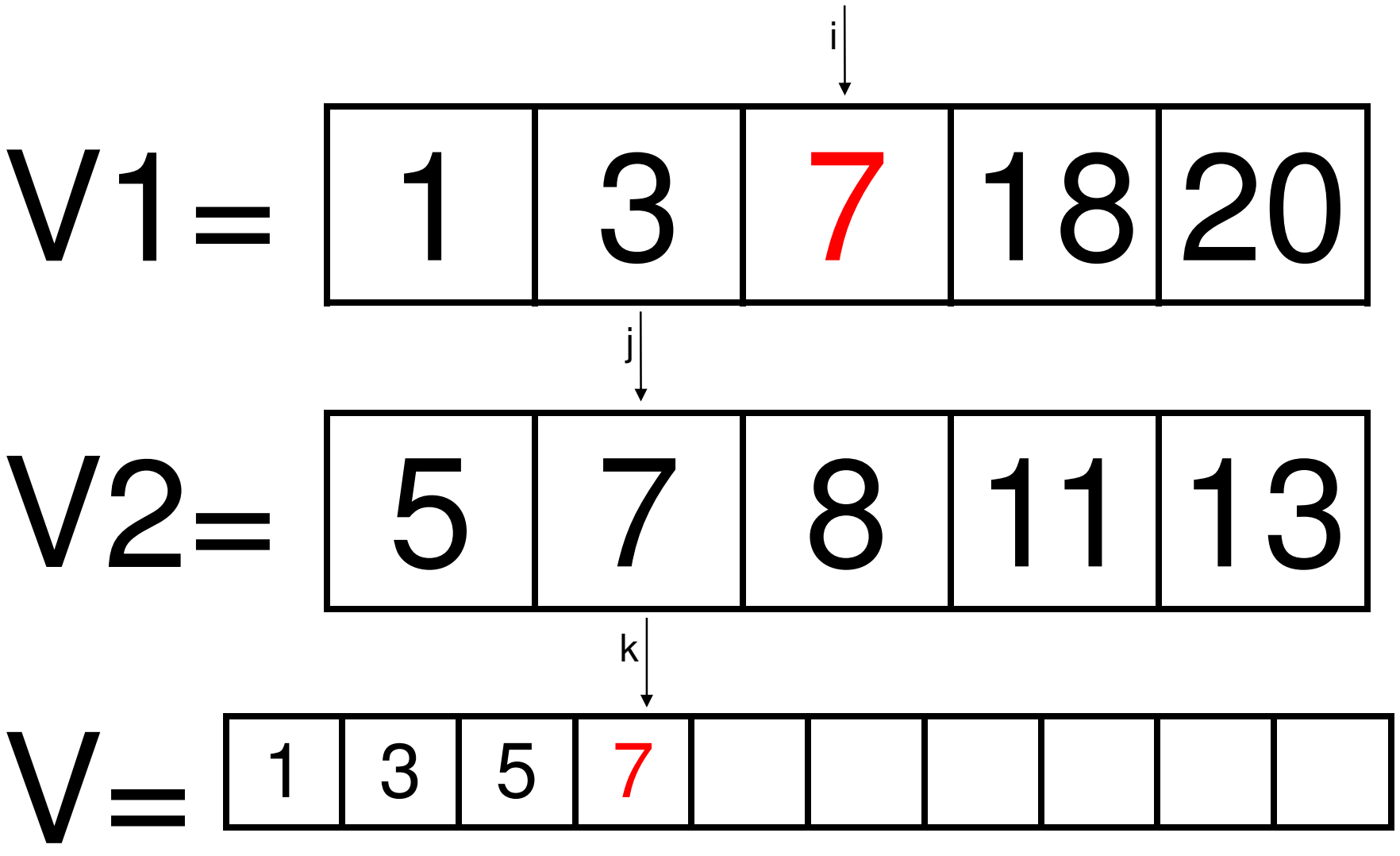
Merge de dois vetores



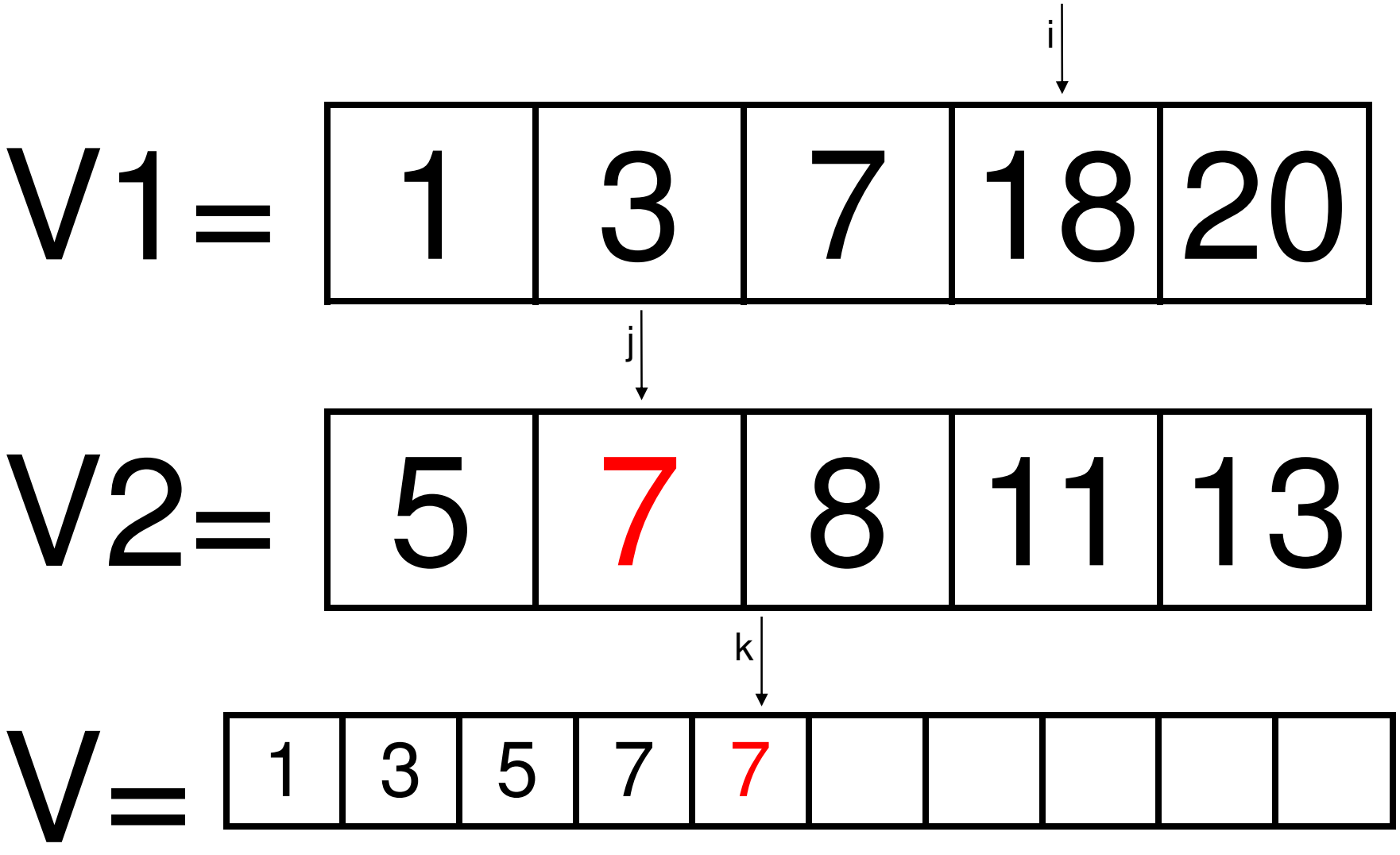
Merge de dois vetores



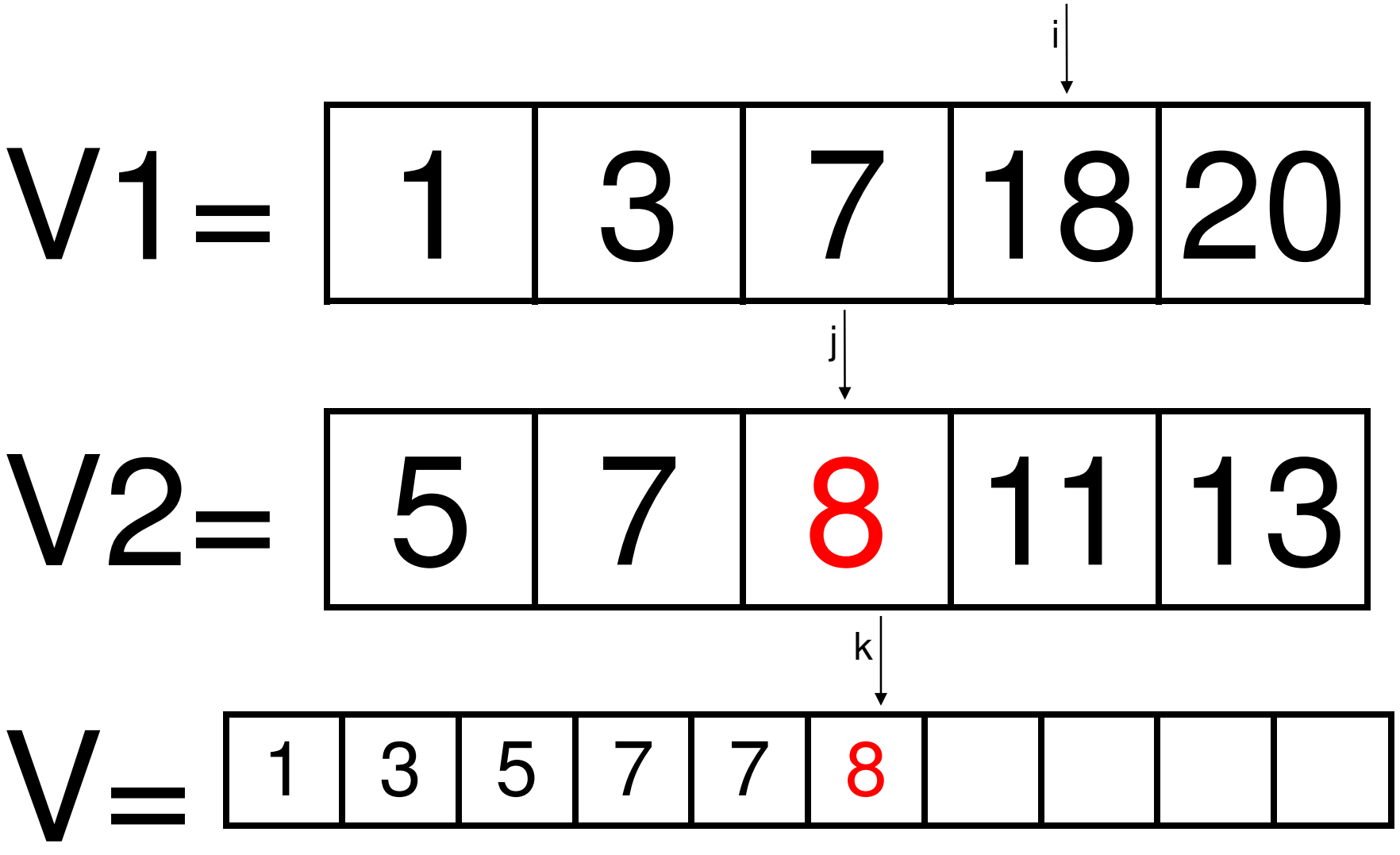
Merge de dois vetores



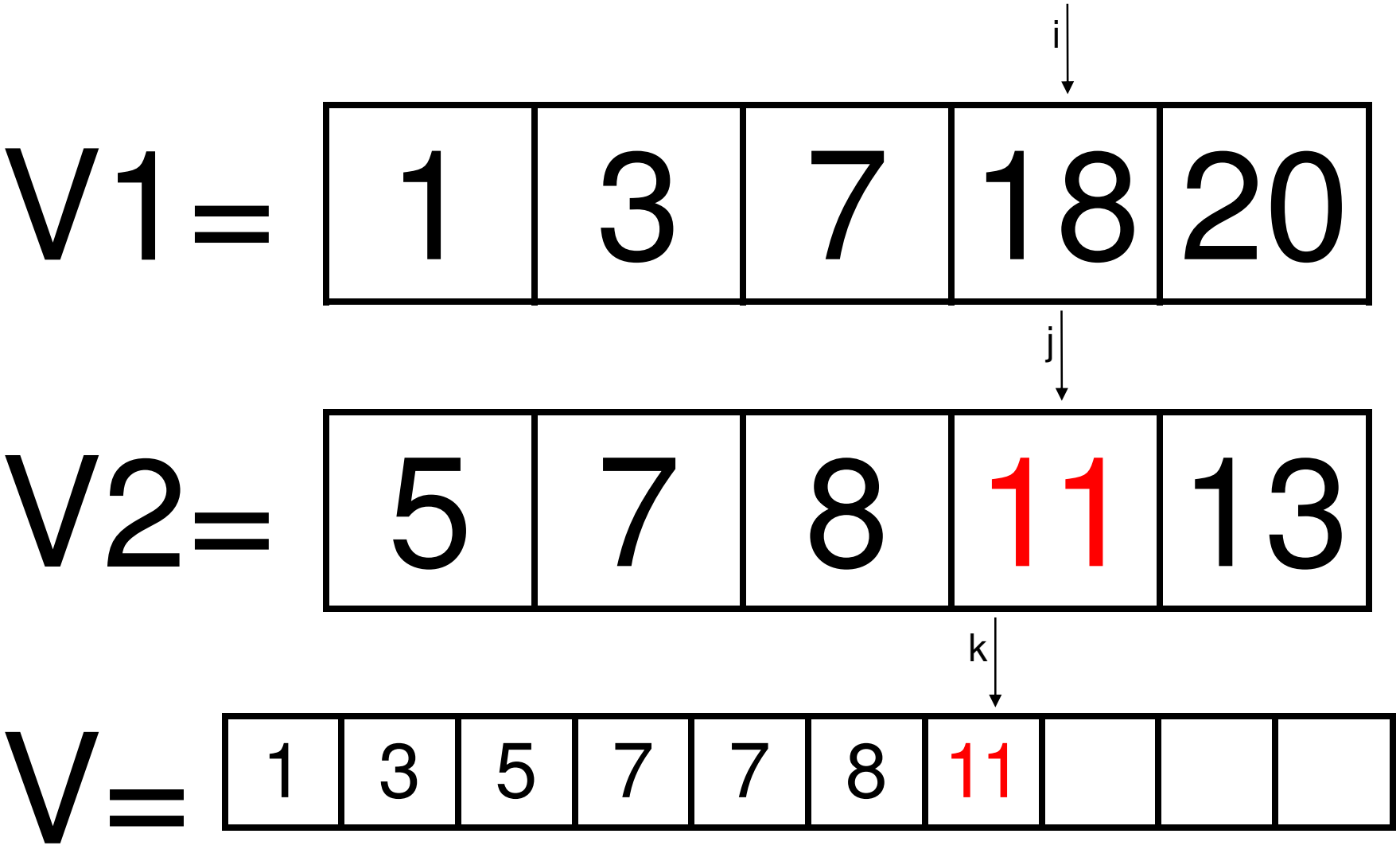
Merge de dois vetores



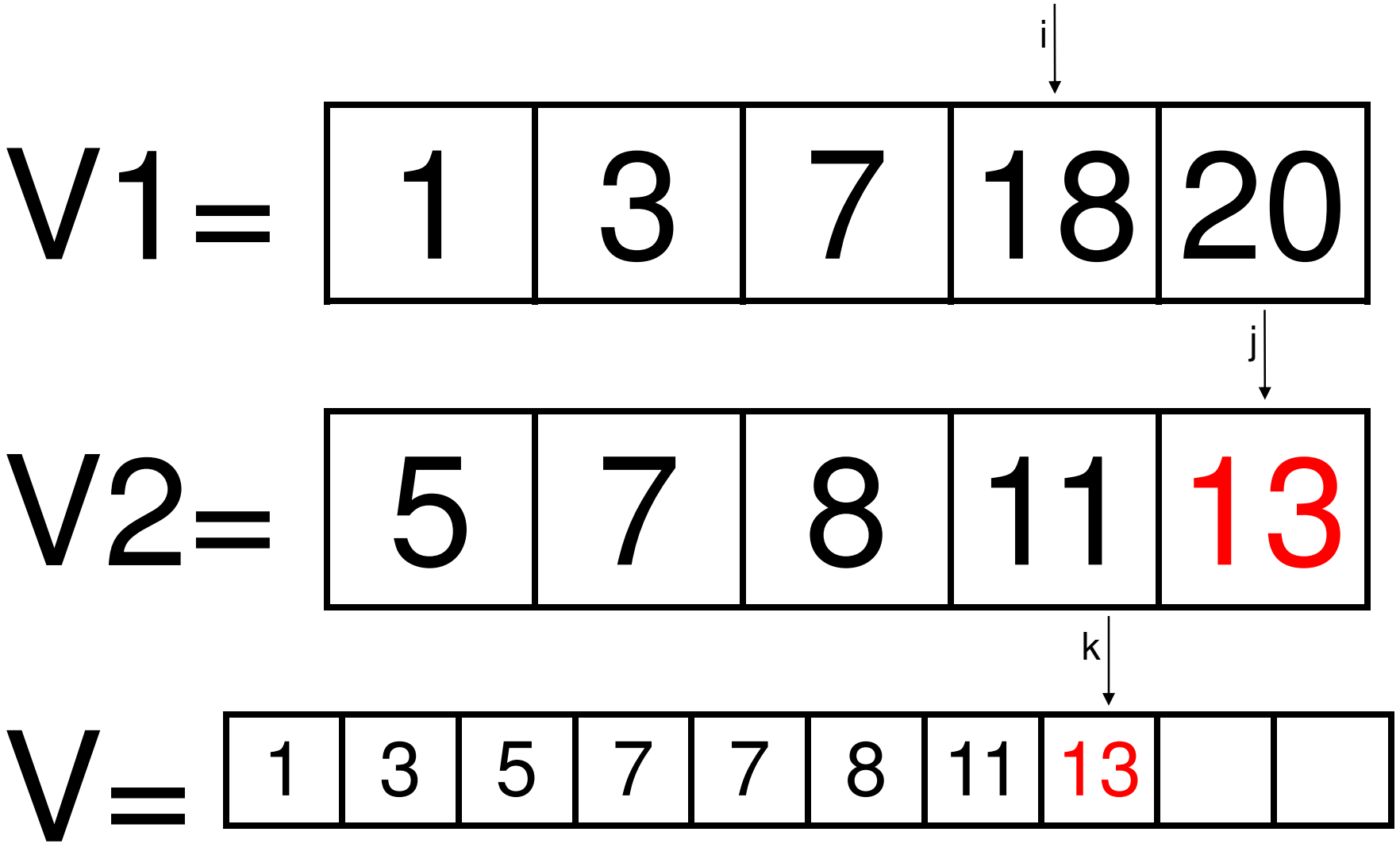
Merge de dois vetores



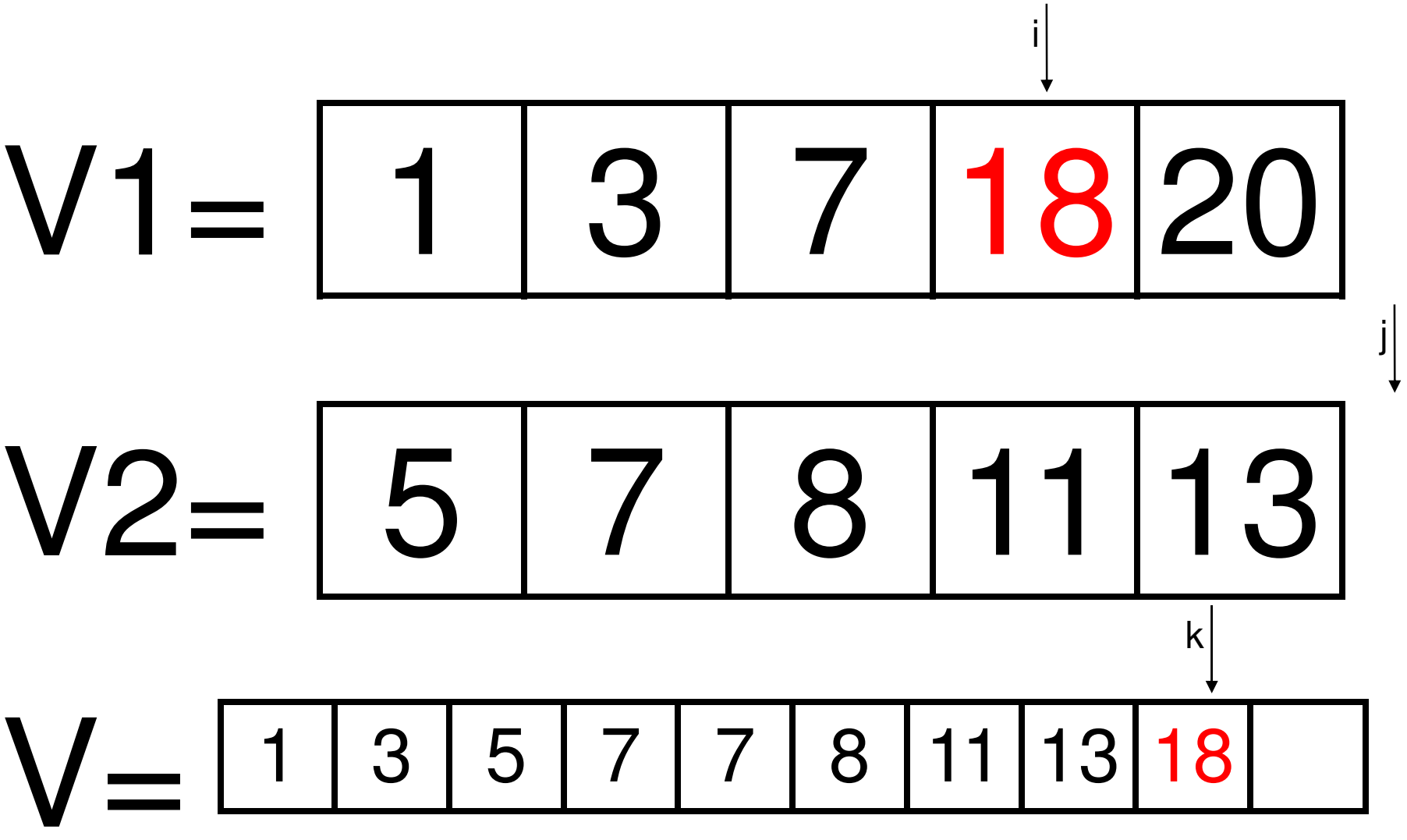
Merge de dois vetores



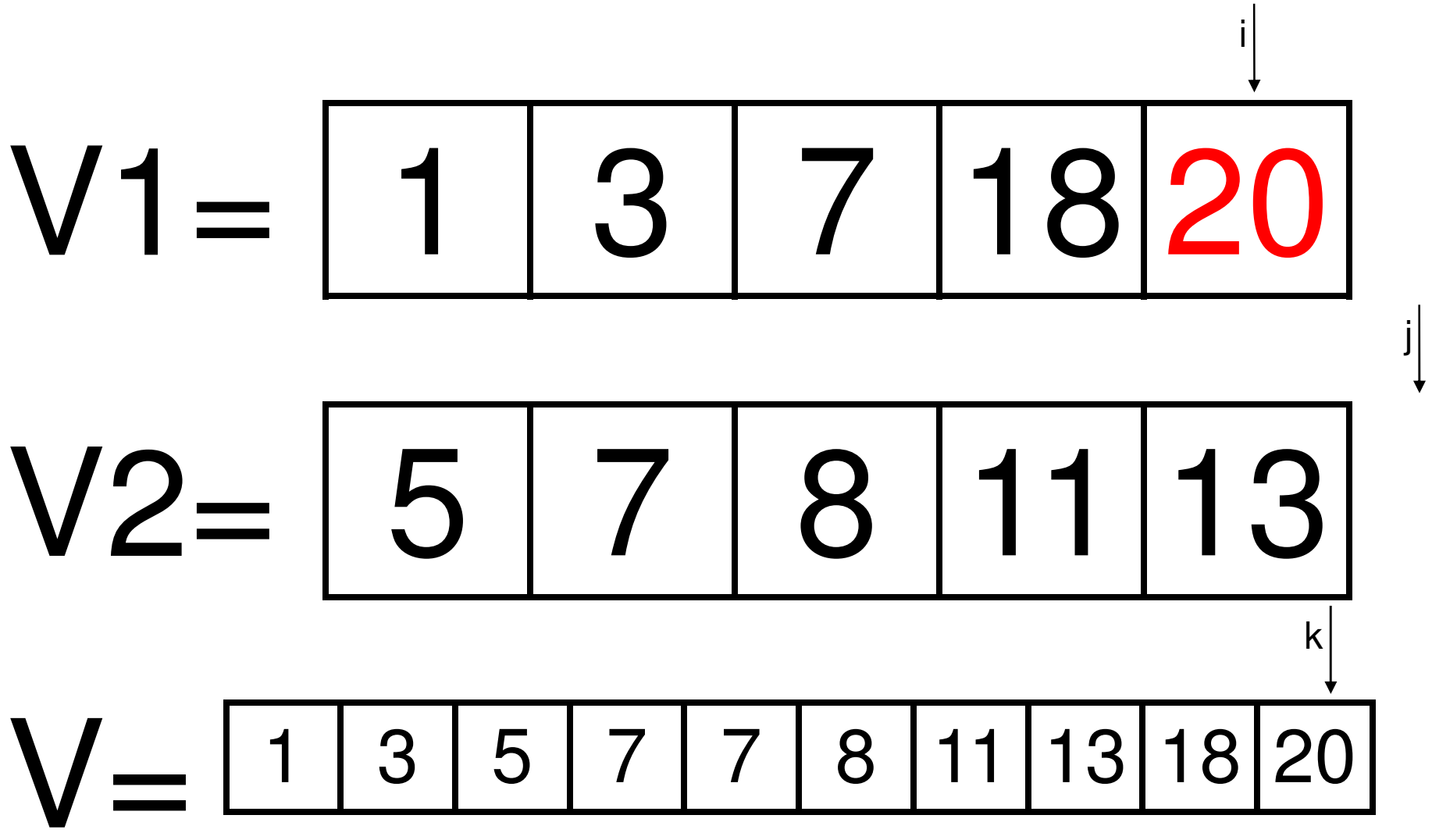
Merge de dois vetores



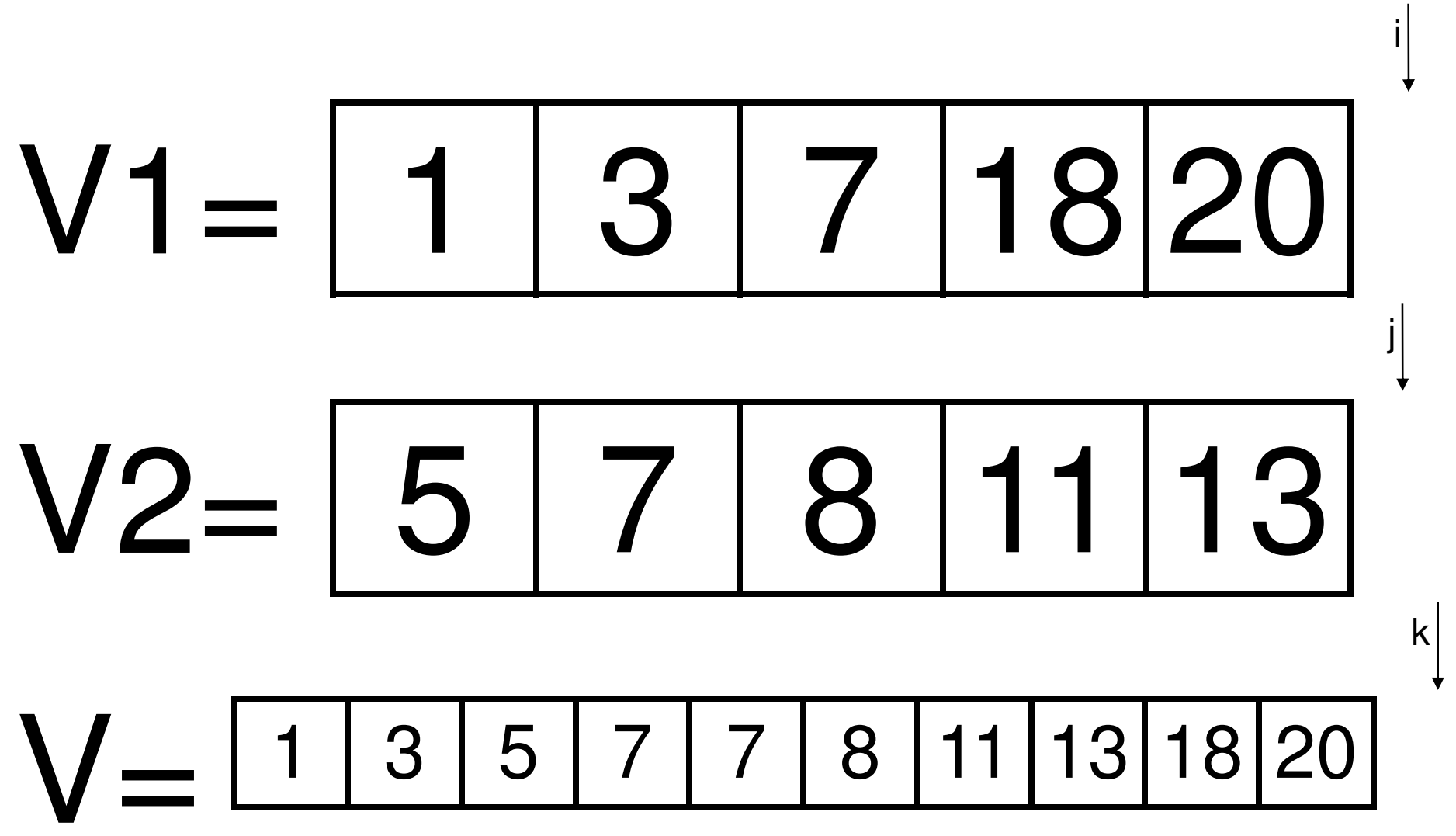
Merge de dois vetores



Merge de dois vetores



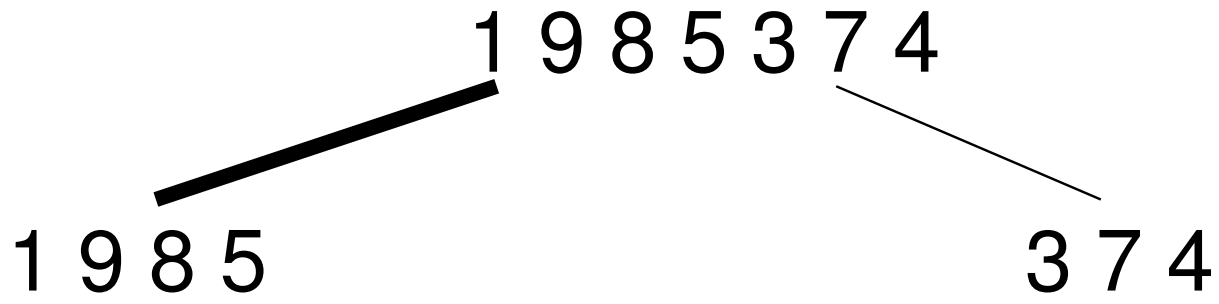
Merge de dois vetores



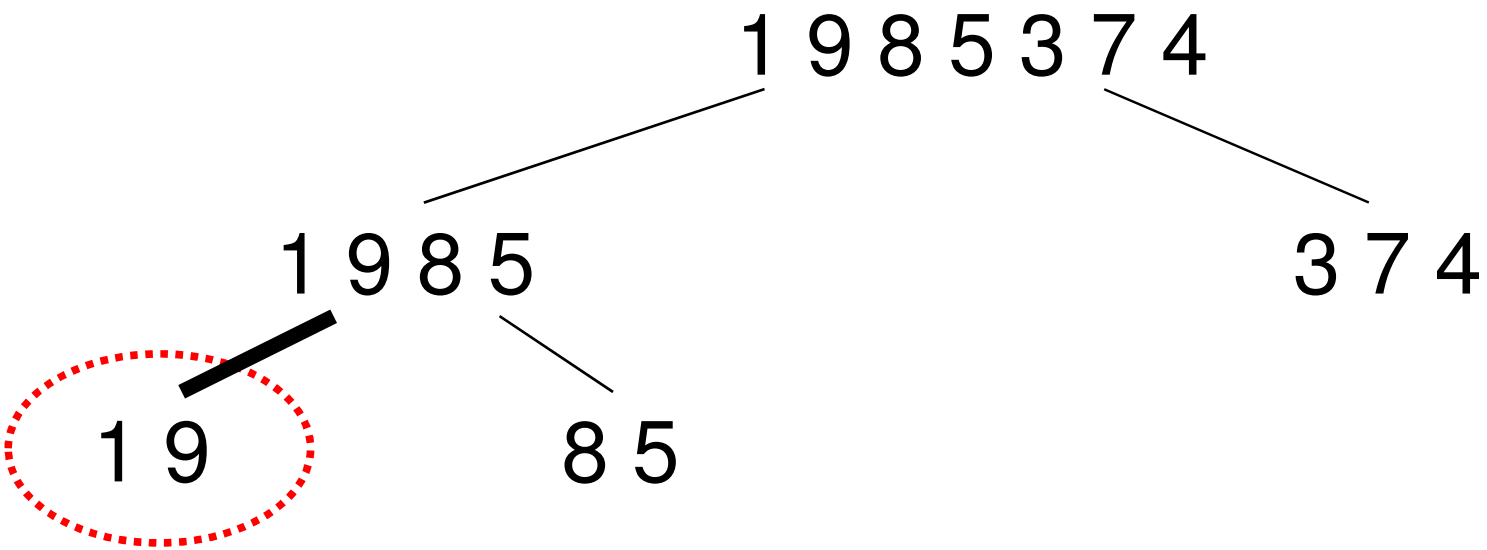
Mergesort

1 9 8 5 3 7 4

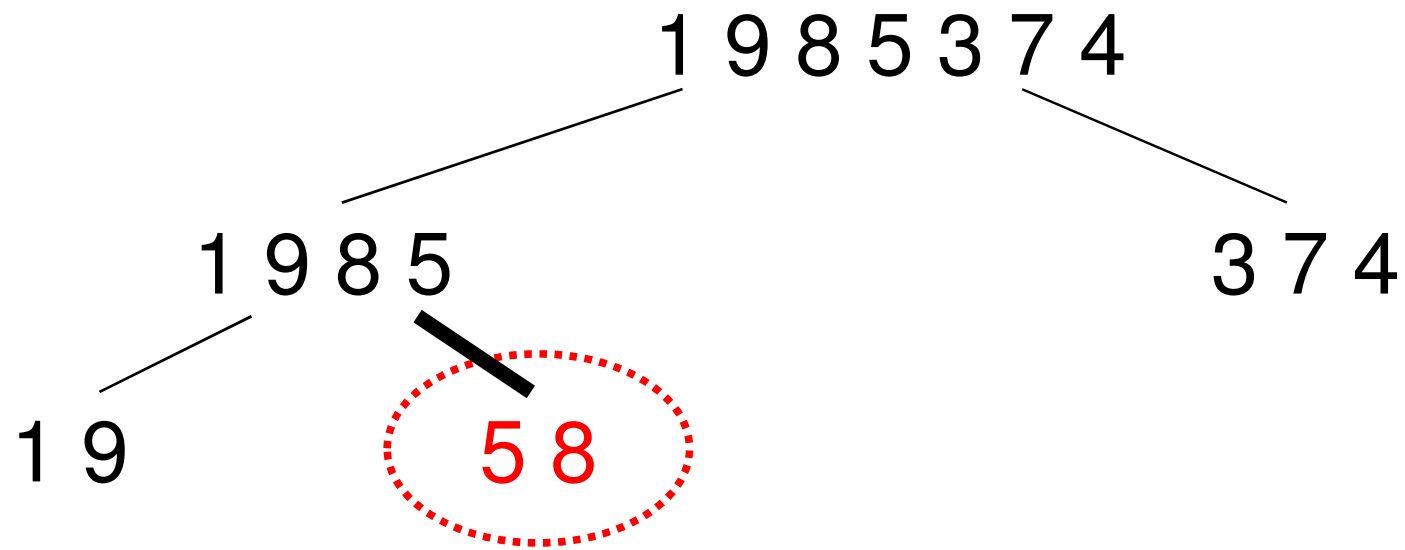
Mergesort



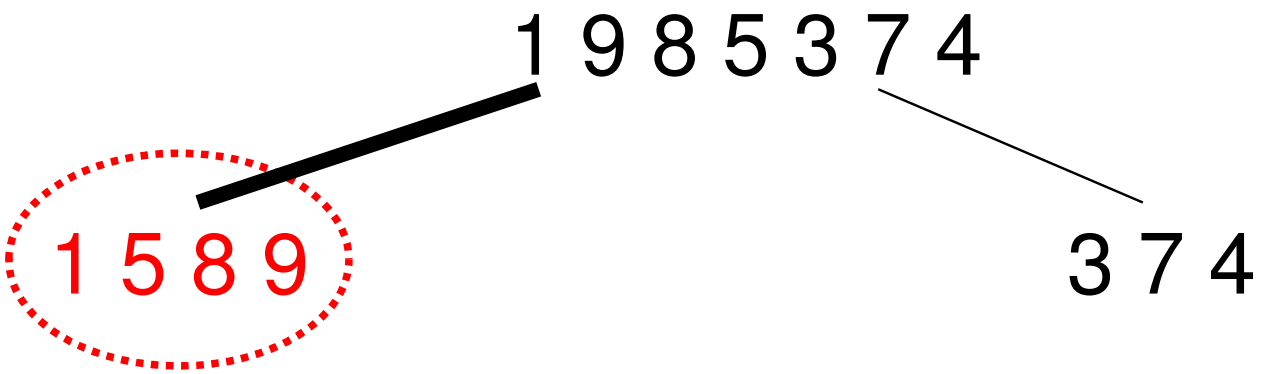
Mergesort



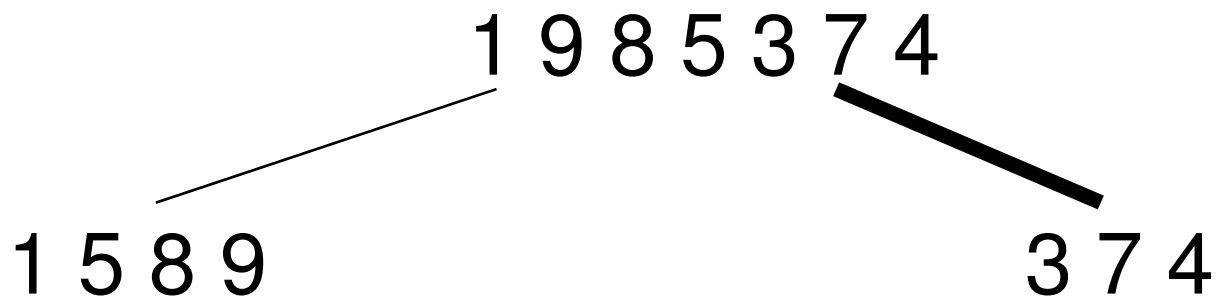
Mergesort



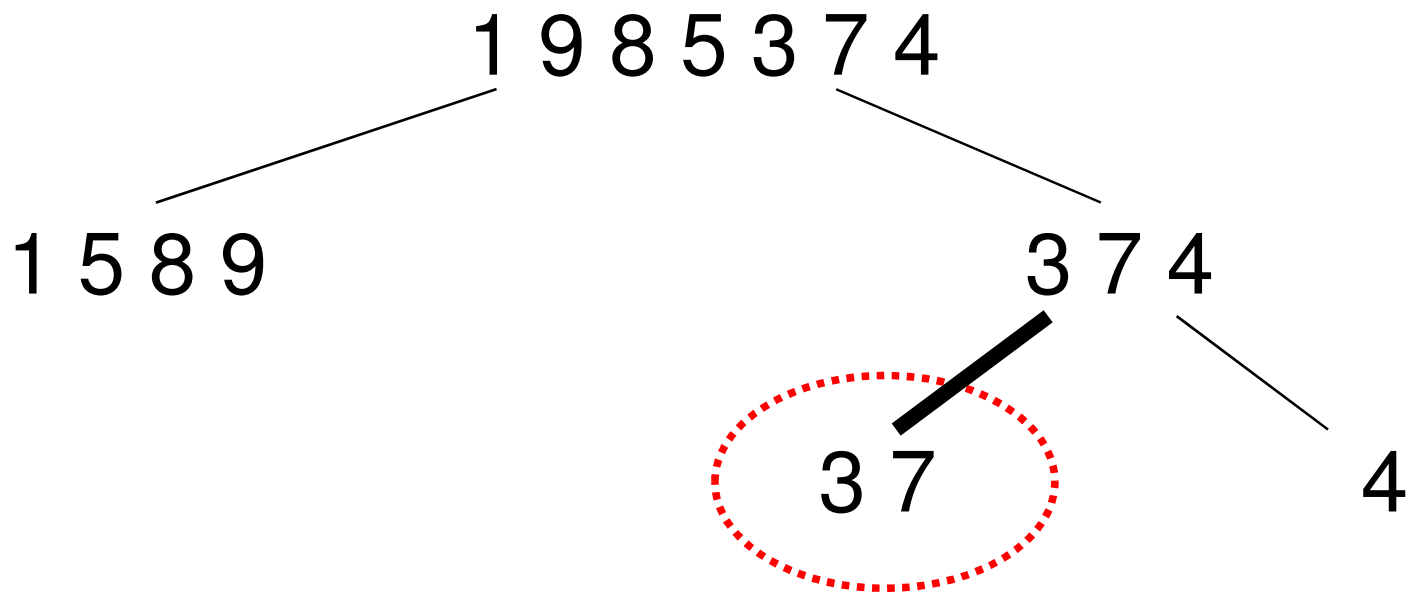
Mergesort



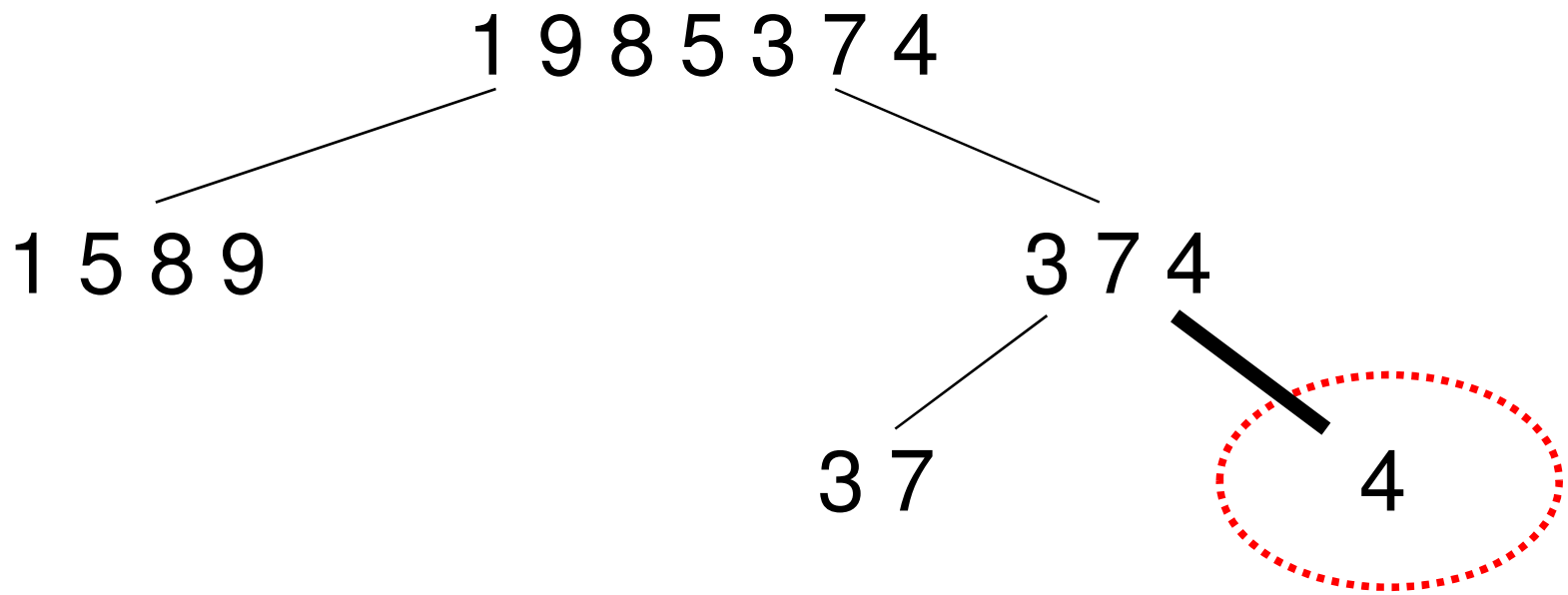
Mergesort



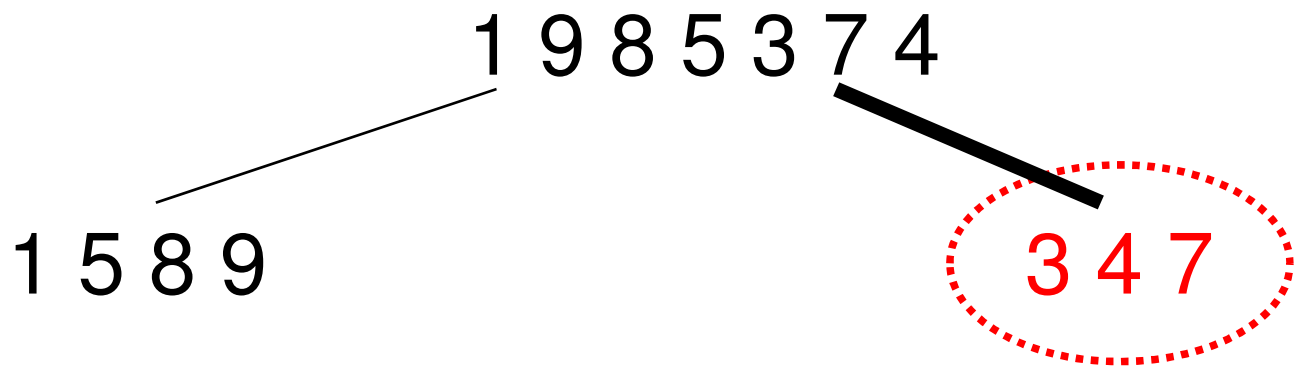
Mergesort



Mergesort



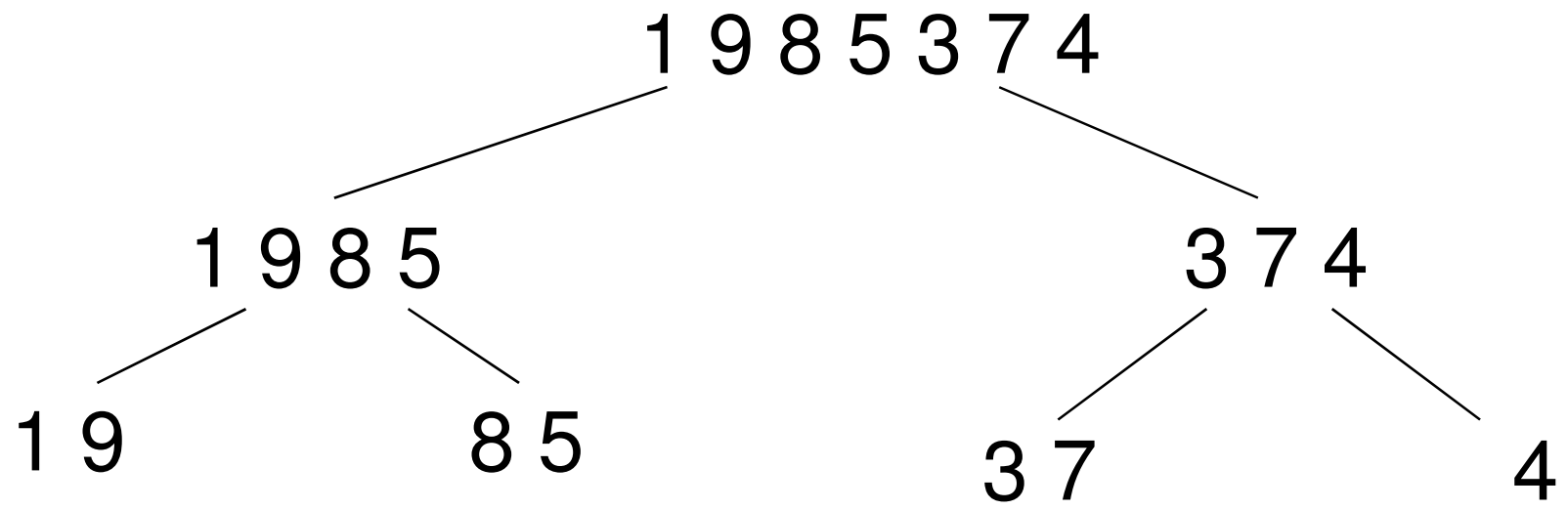
Mergesort



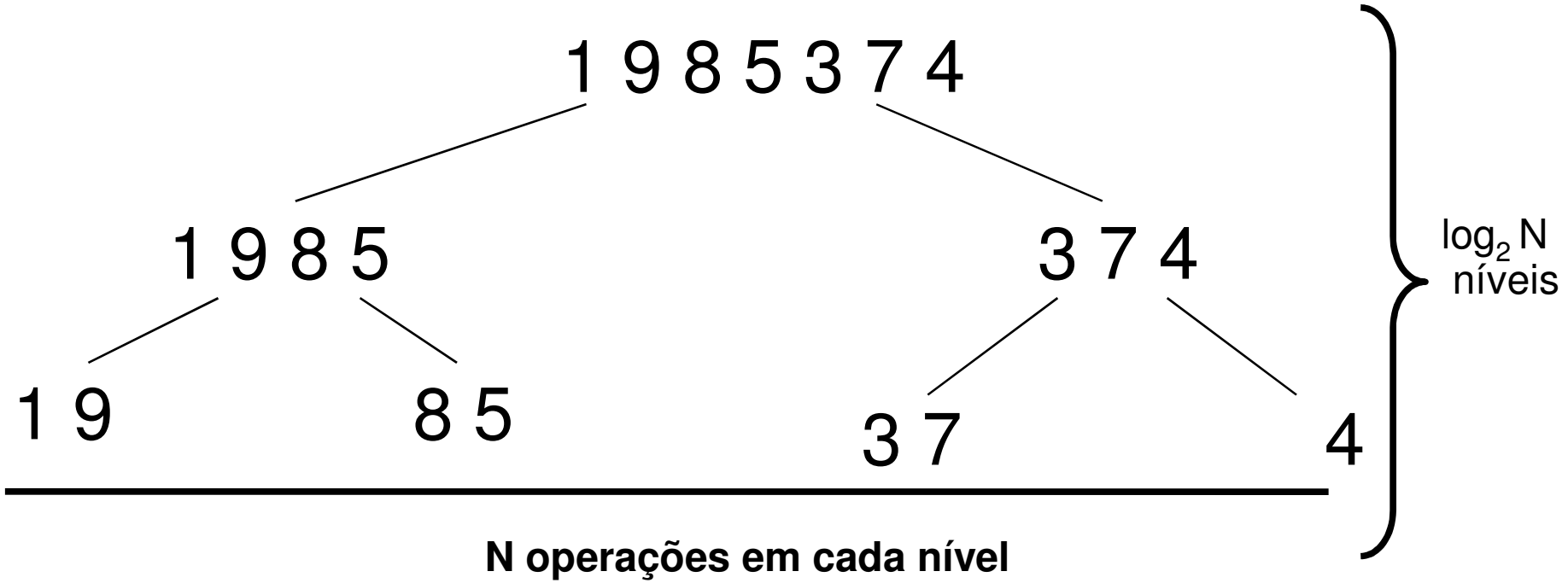
Mergesort

1 3 4 5 7 8 9

Mergesort - operações



Mergesort - operações



Total de operações:

$N * \log_2 N$

Heapsort

- Ordenação baseada na estrutura de dados *heap*
  - A *heap* é um vetor que pode ser visto como uma árvore binária completa
  - Cada nó da árvore equivale a um valor no vetor

Primitivas da Heap

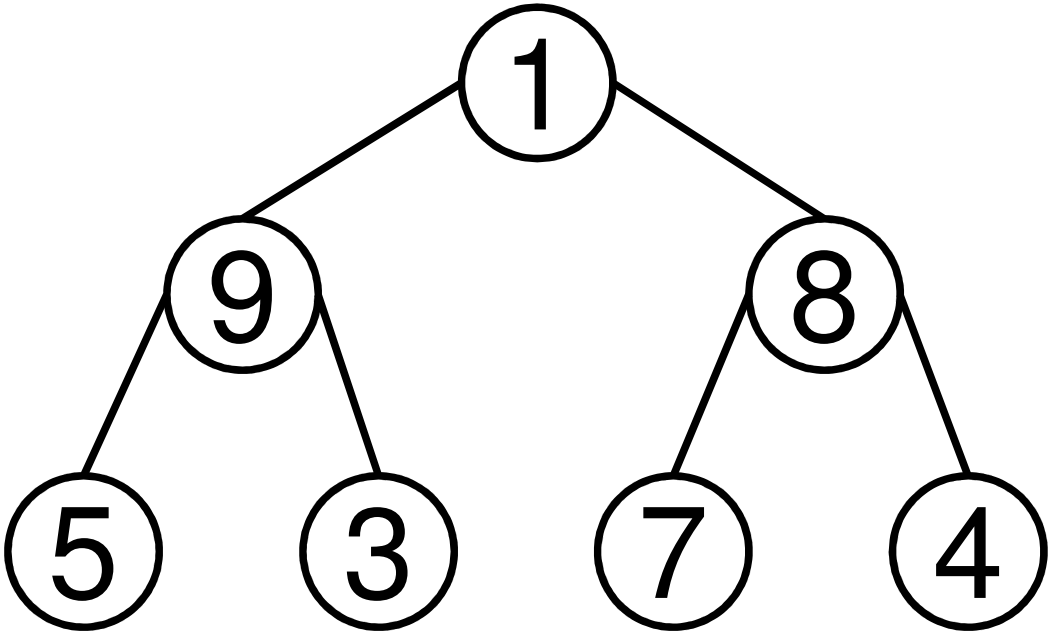
- length(V)
  - número de elementos no vetor V
- heap-size(V)
  - número de elementos da *heap* armazenados no vetor V

Vetor Heap – primitivas de árvore

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

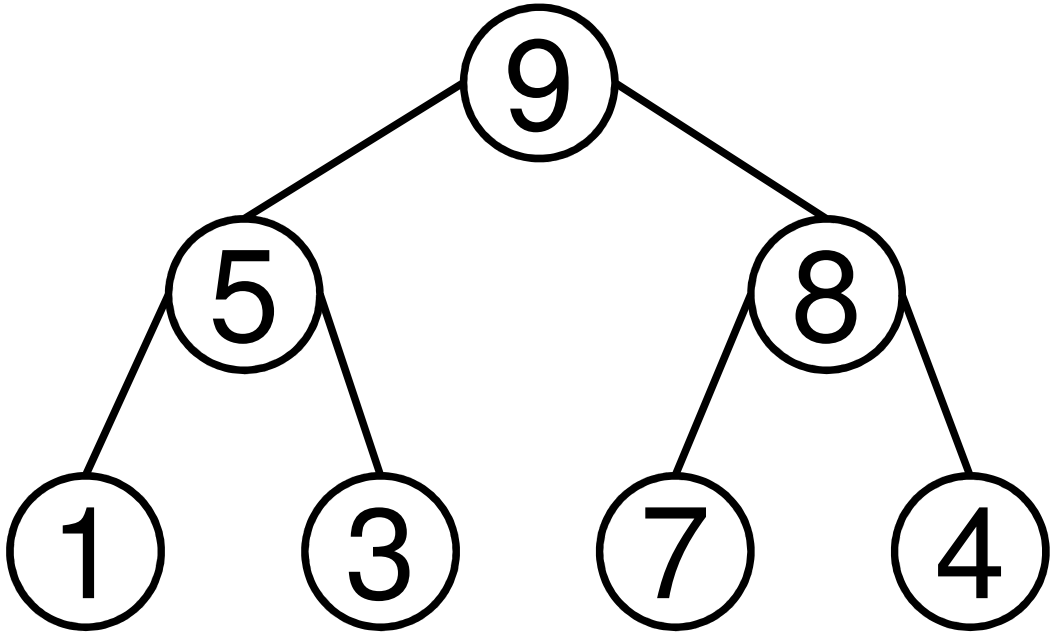
Heap como um vetor

1	9	8	5	3	7	4



Propriedade da Heap

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



LAWS

Primitiva Heapify

- Função para manter a propriedade da heap

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

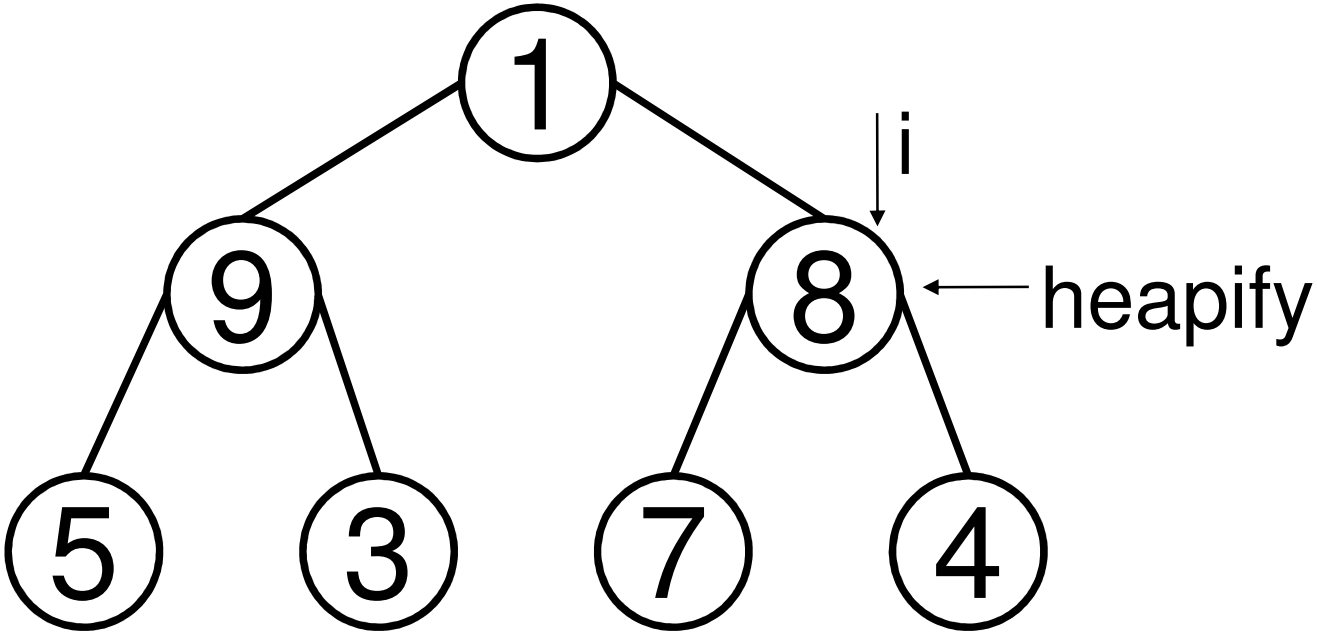
Construindo uma Heap

- 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
```

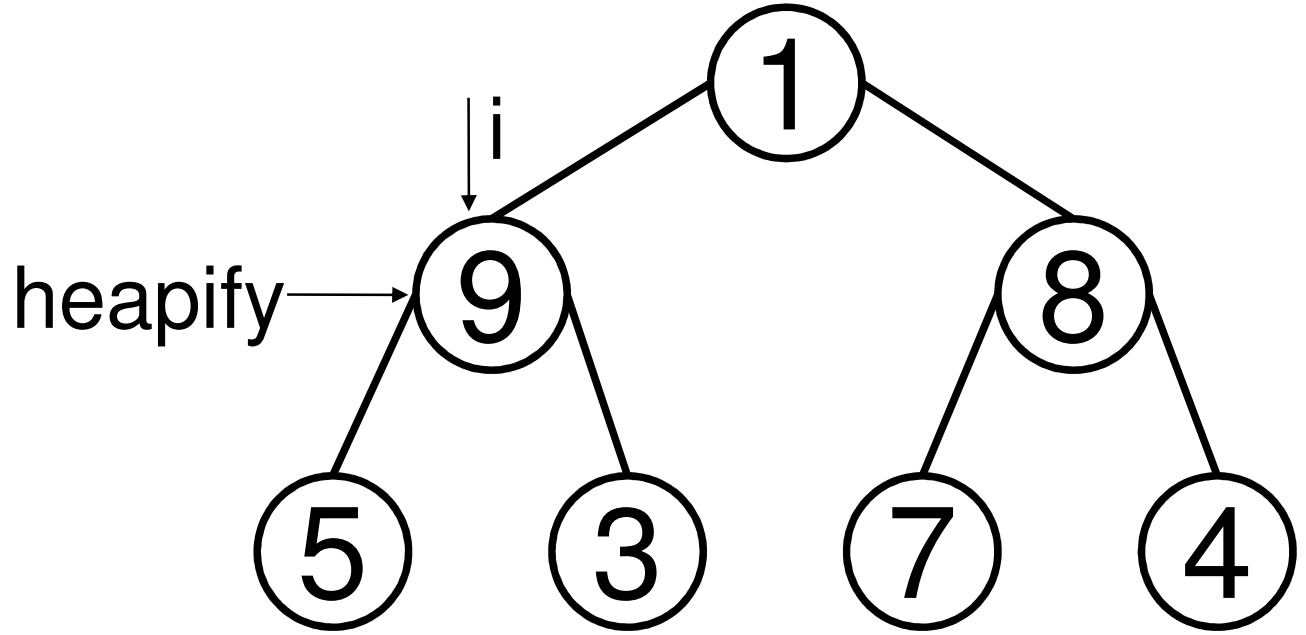
Construindo uma Heap

1	9	8	5	3	7	4



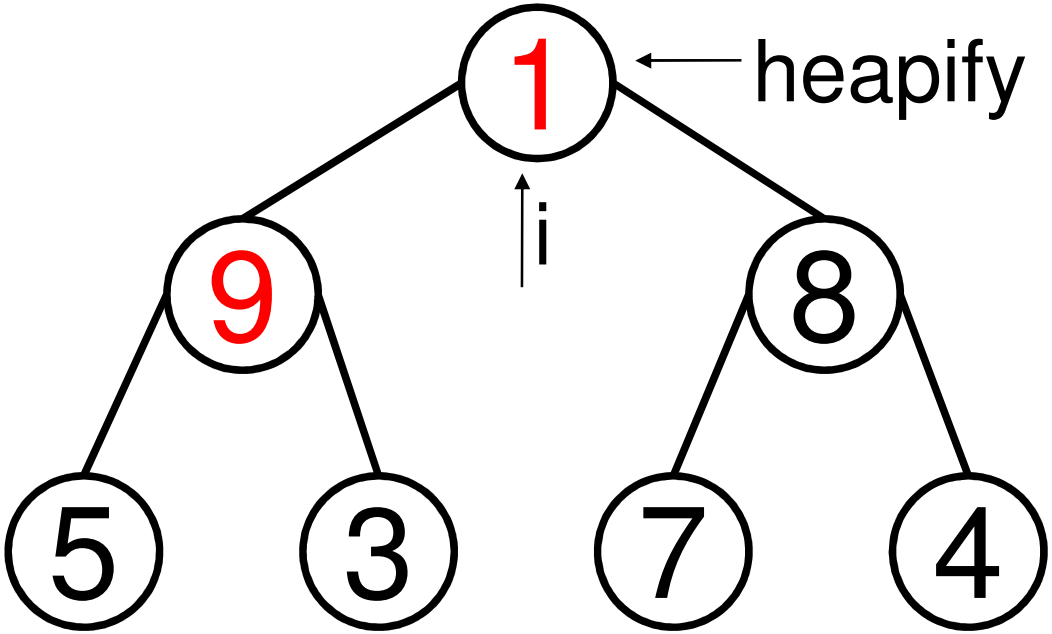
Construindo uma Heap

1	9	8	5	3	7	4



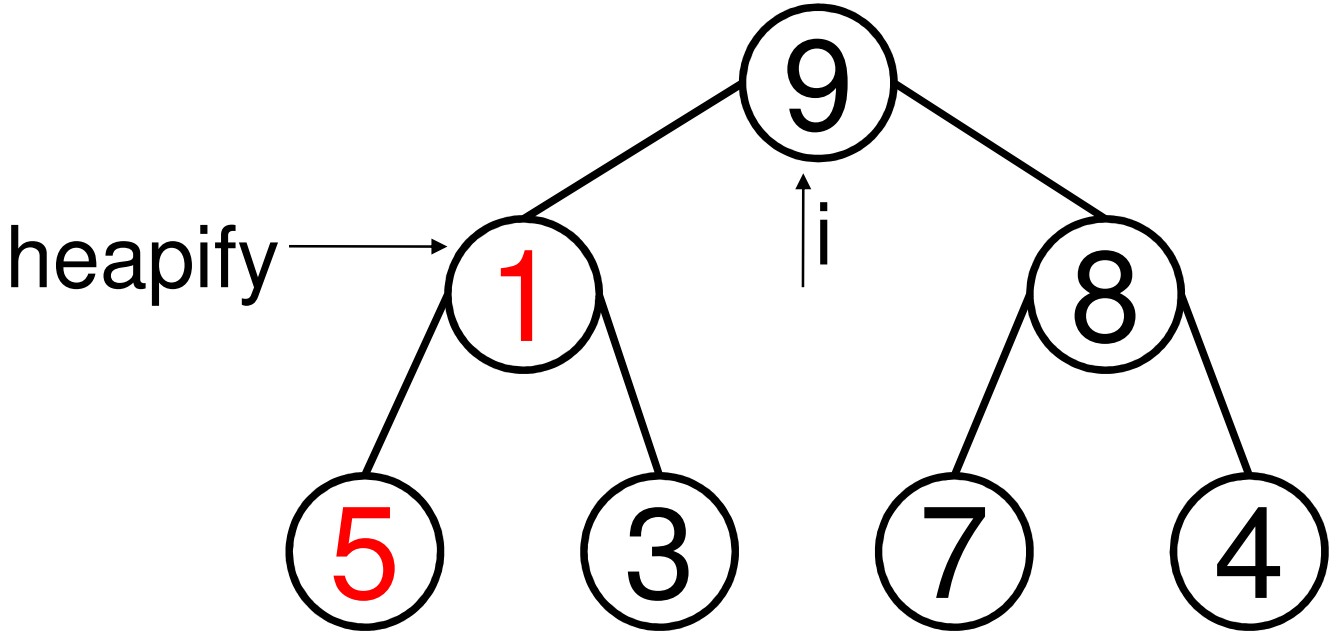
Construindo uma Heap

1	9	8	5	3	7	4



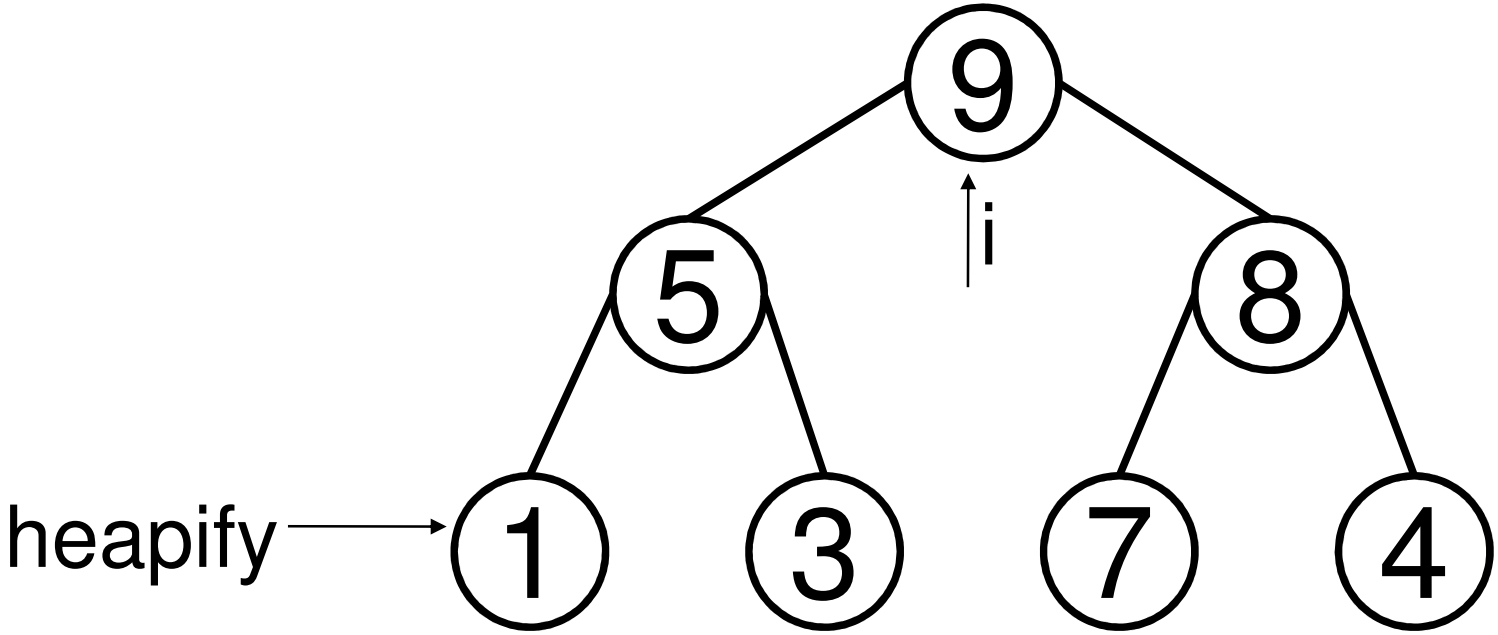
Construindo uma Heap

9	1	8	5	3	7	4



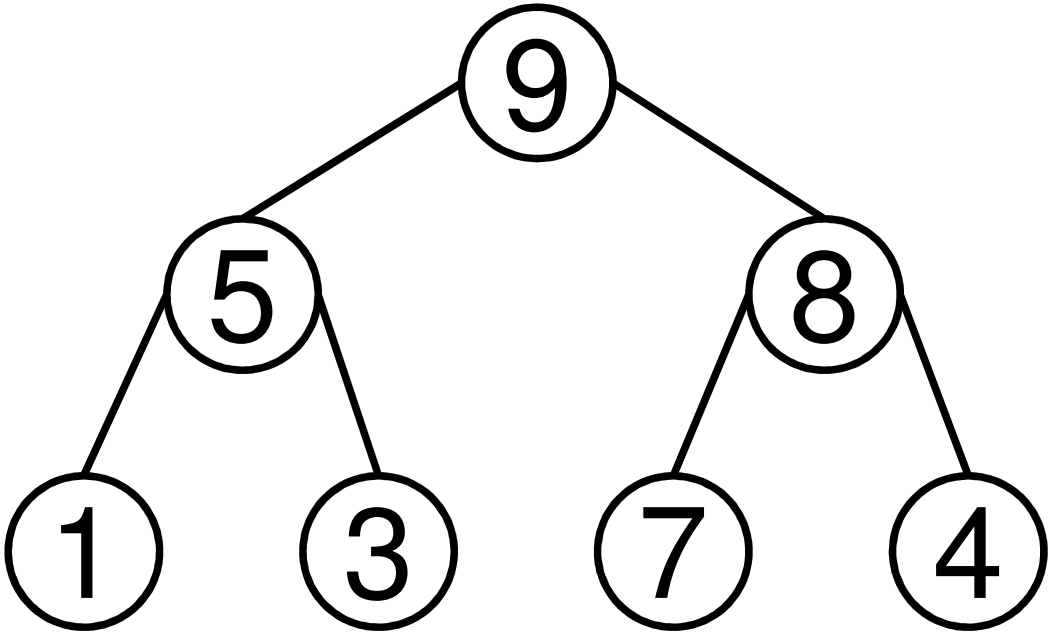
Construindo uma Heap

9	5	8	1	3	7	4



Construindo uma Heap

9	5	8	1	3	7	4

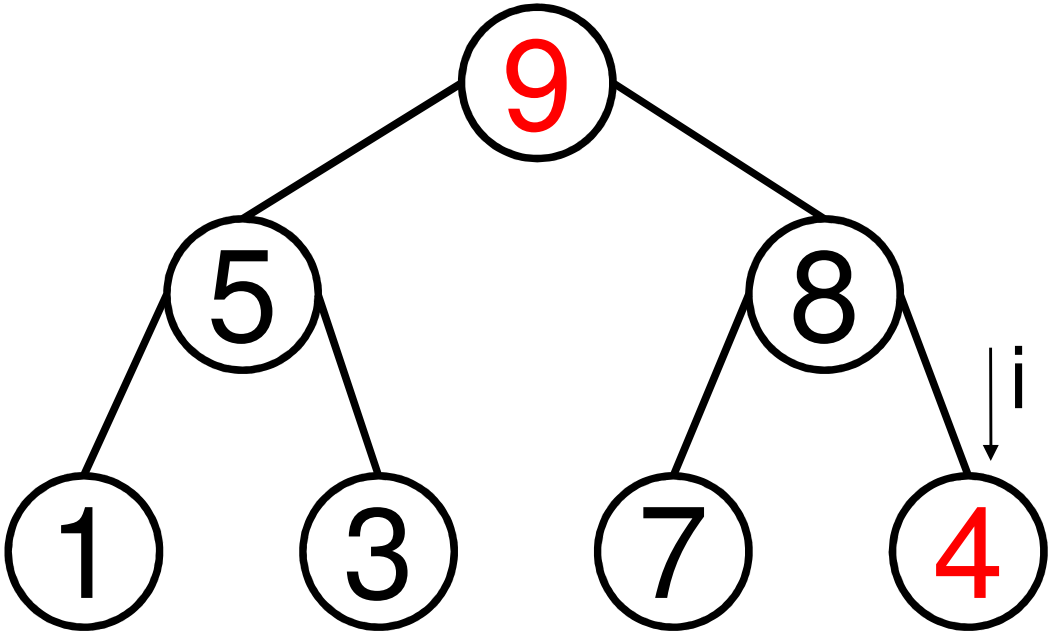


Heapsort

```
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
```

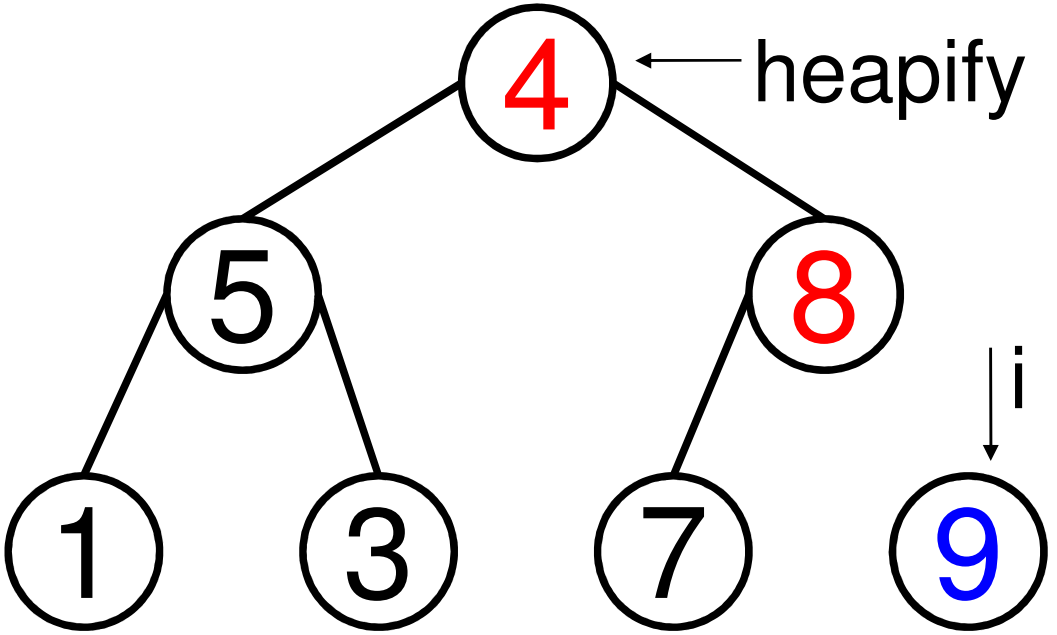
Heapsort

9	5	8	1	3	7	4



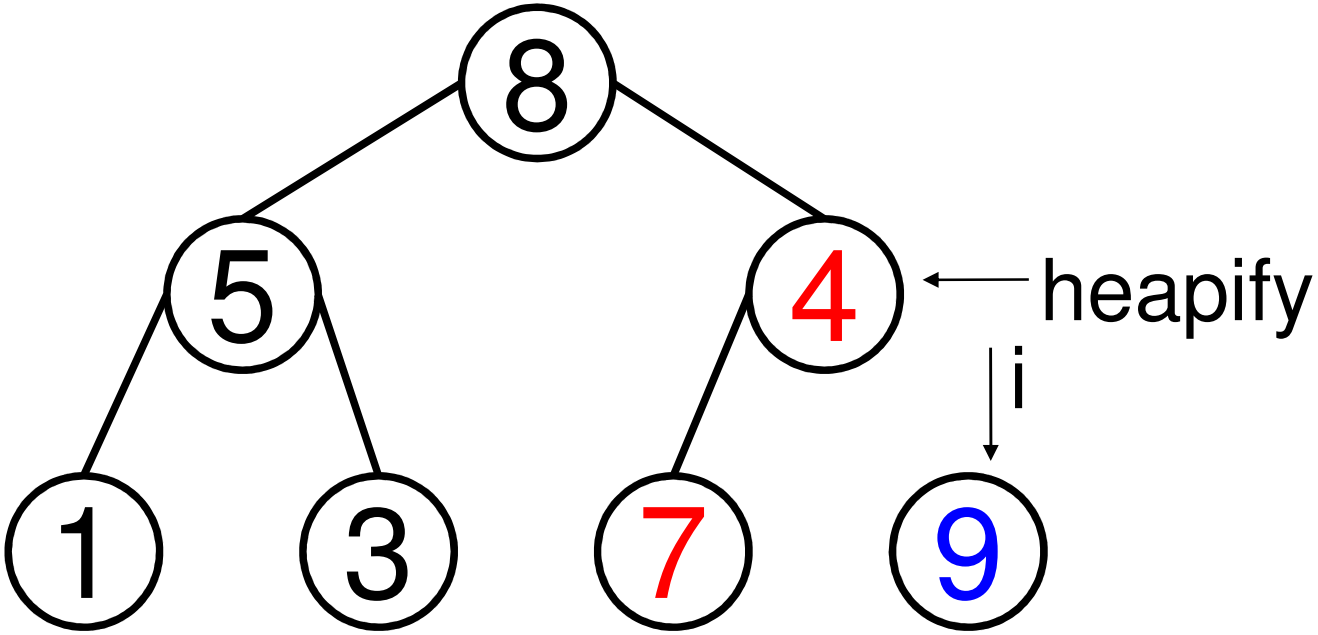
Heapsort

4	5	8	1	3	7	9



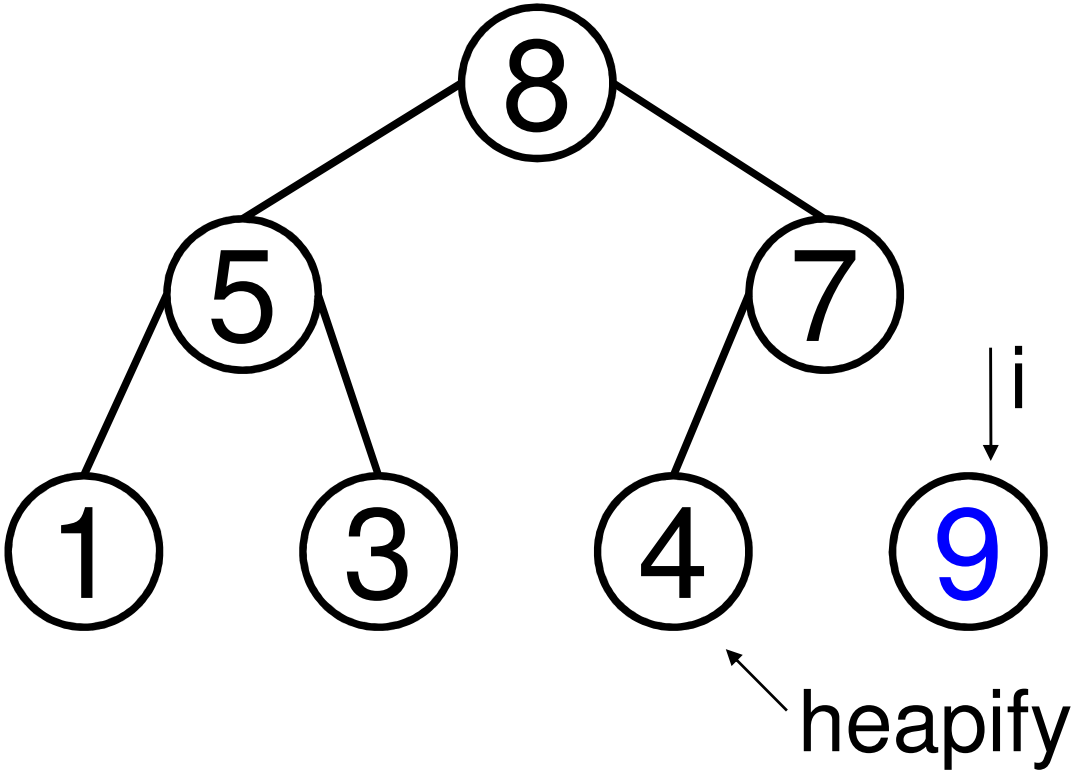
Heapsort

8	5	4	1	3	7	9



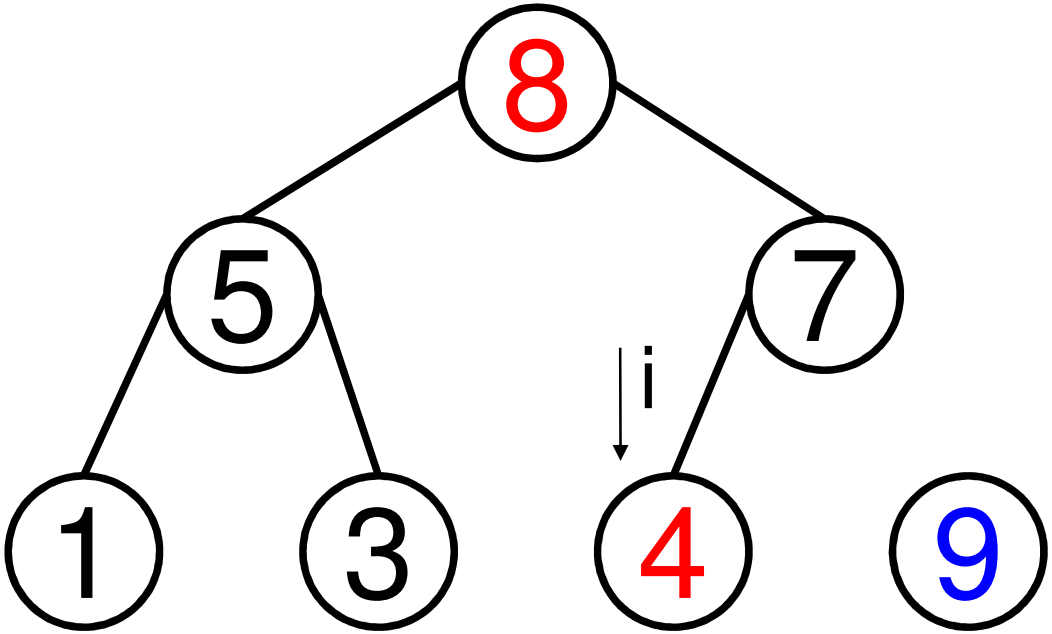
Heapsort

8	5	7	1	3	4	9



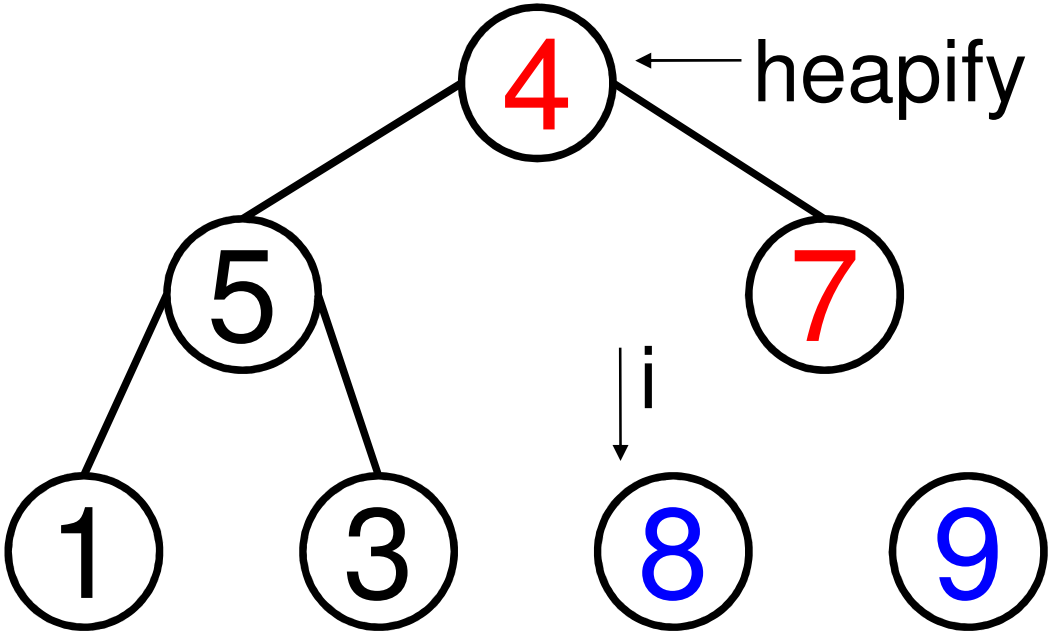
Heapsort

8	5	7	1	3	4	9



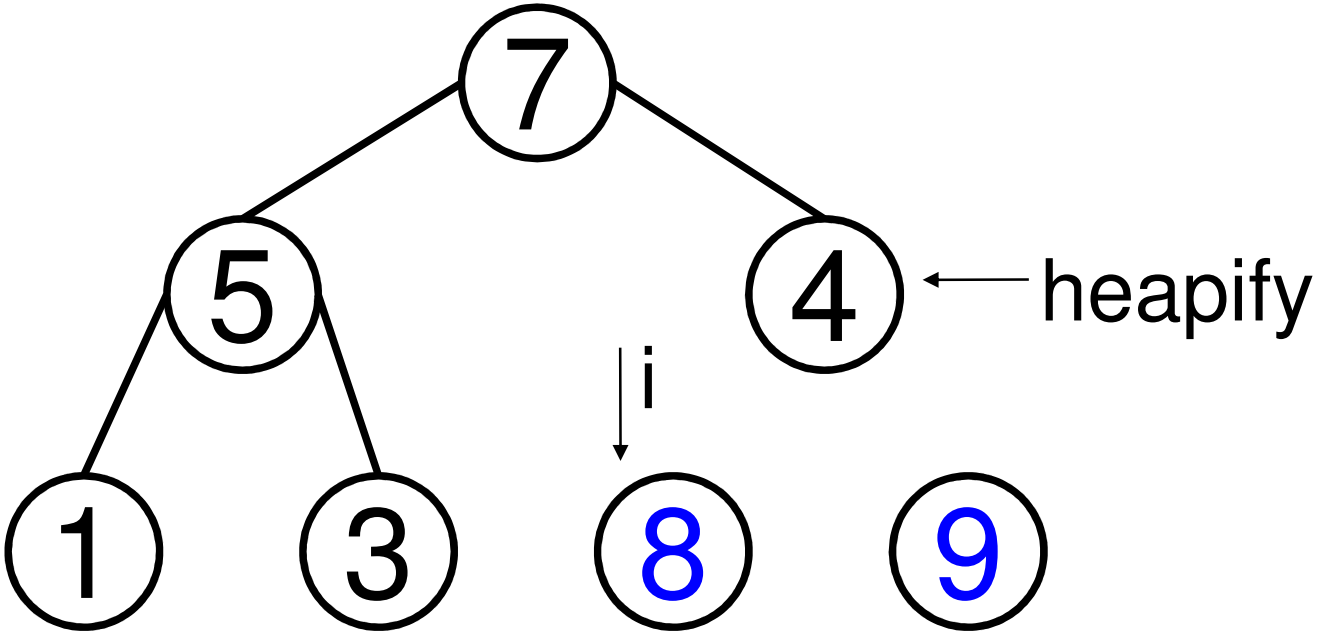
Heapsort

4	5	7	1	3	8	9



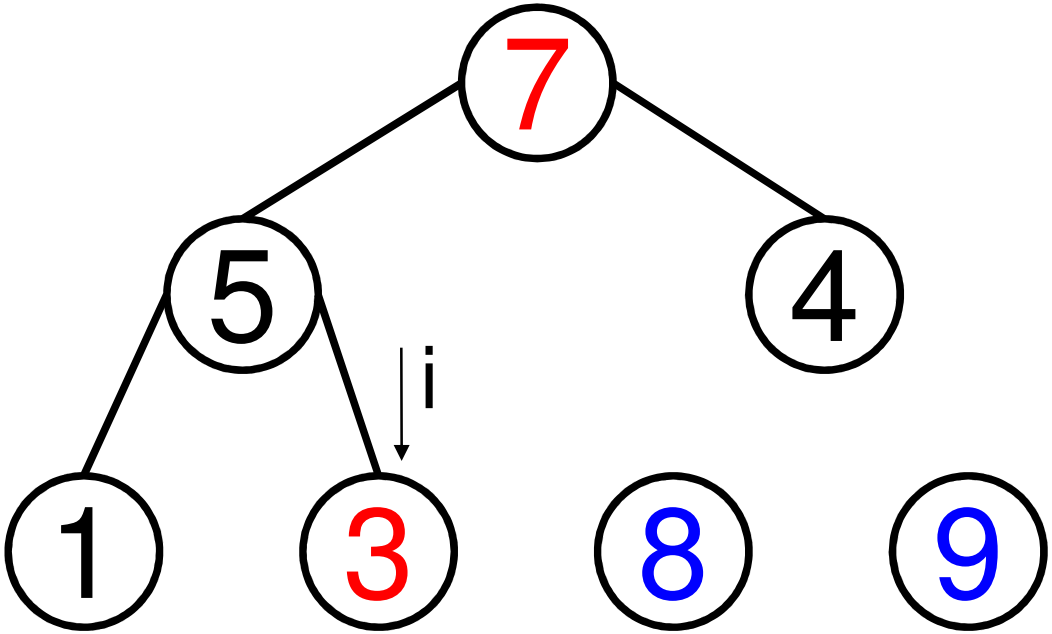
Heapsort

7	5	4	1	3	8	9

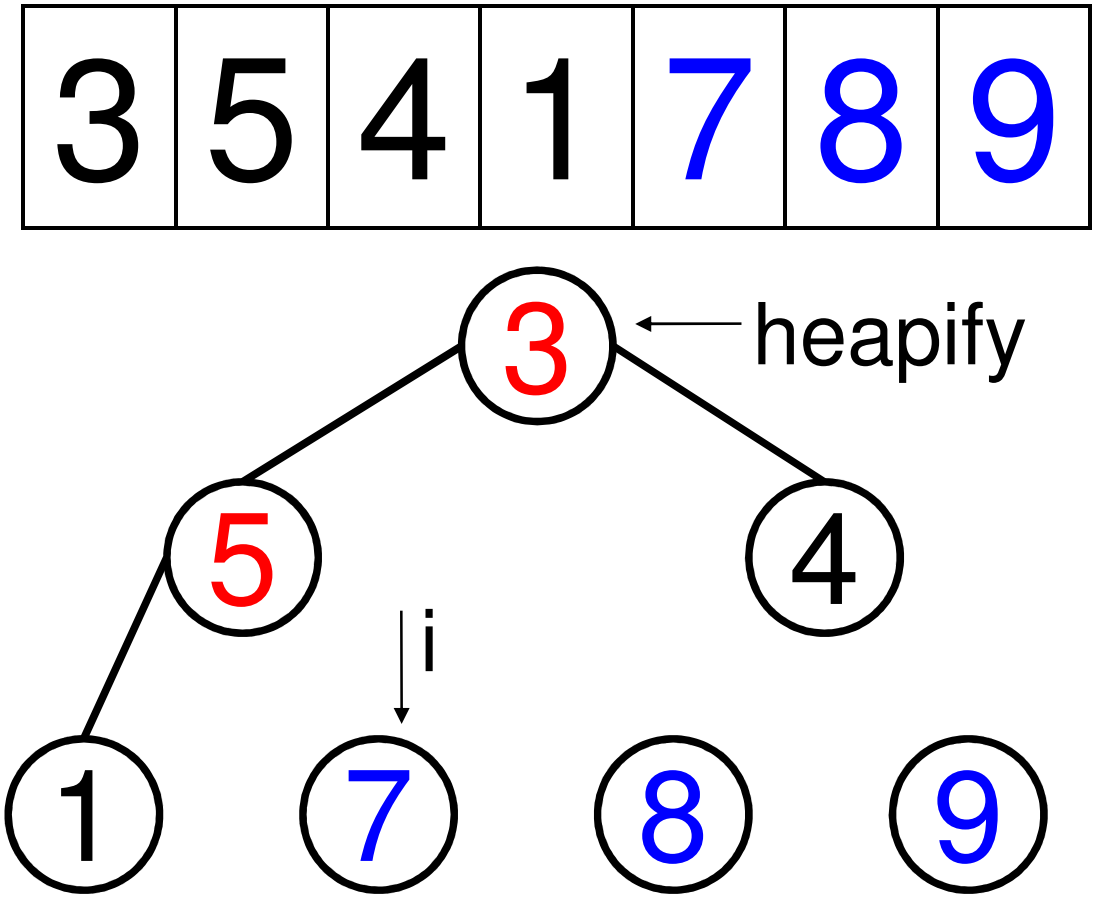


Heapsort

7	5	4	1	3	8	9

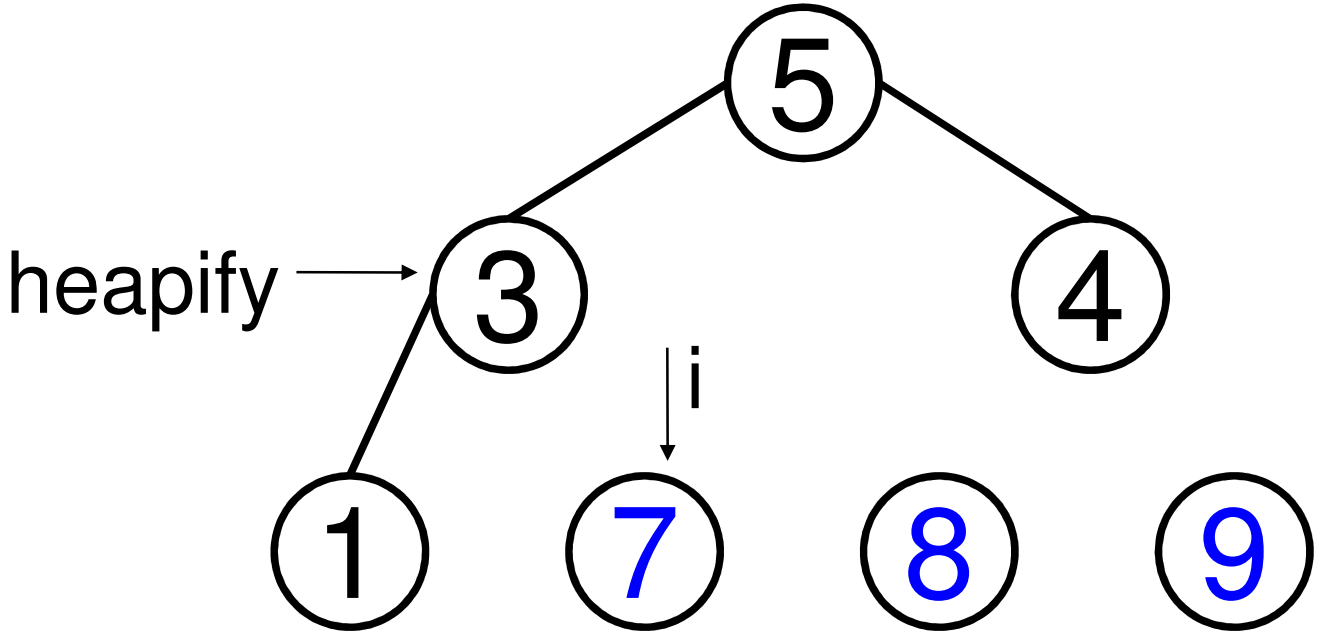


Heapsort



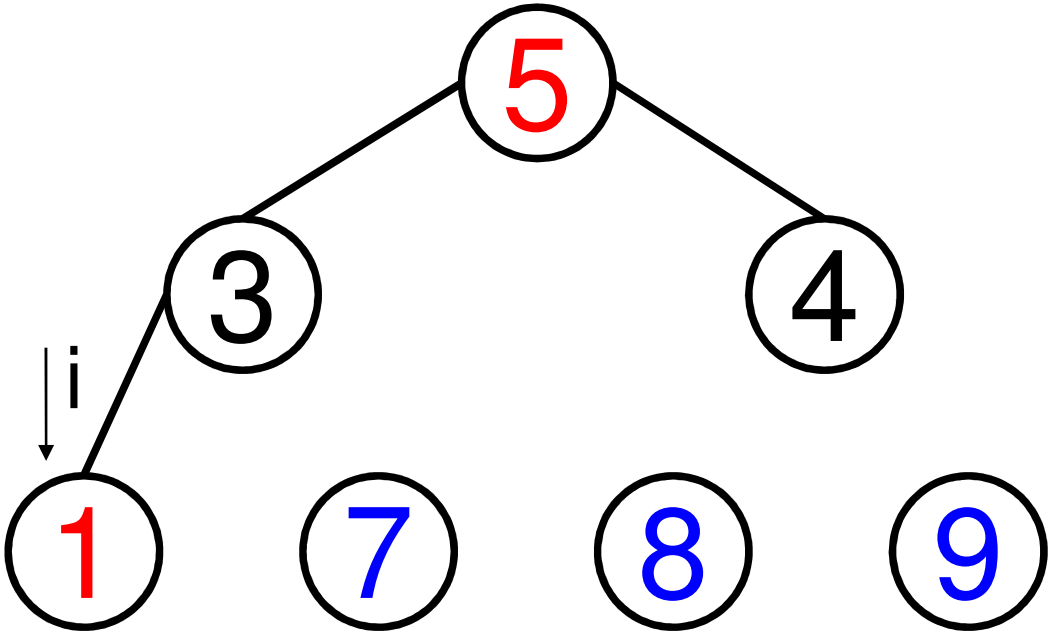
Heapsort

5	3	4	1	7	8	9

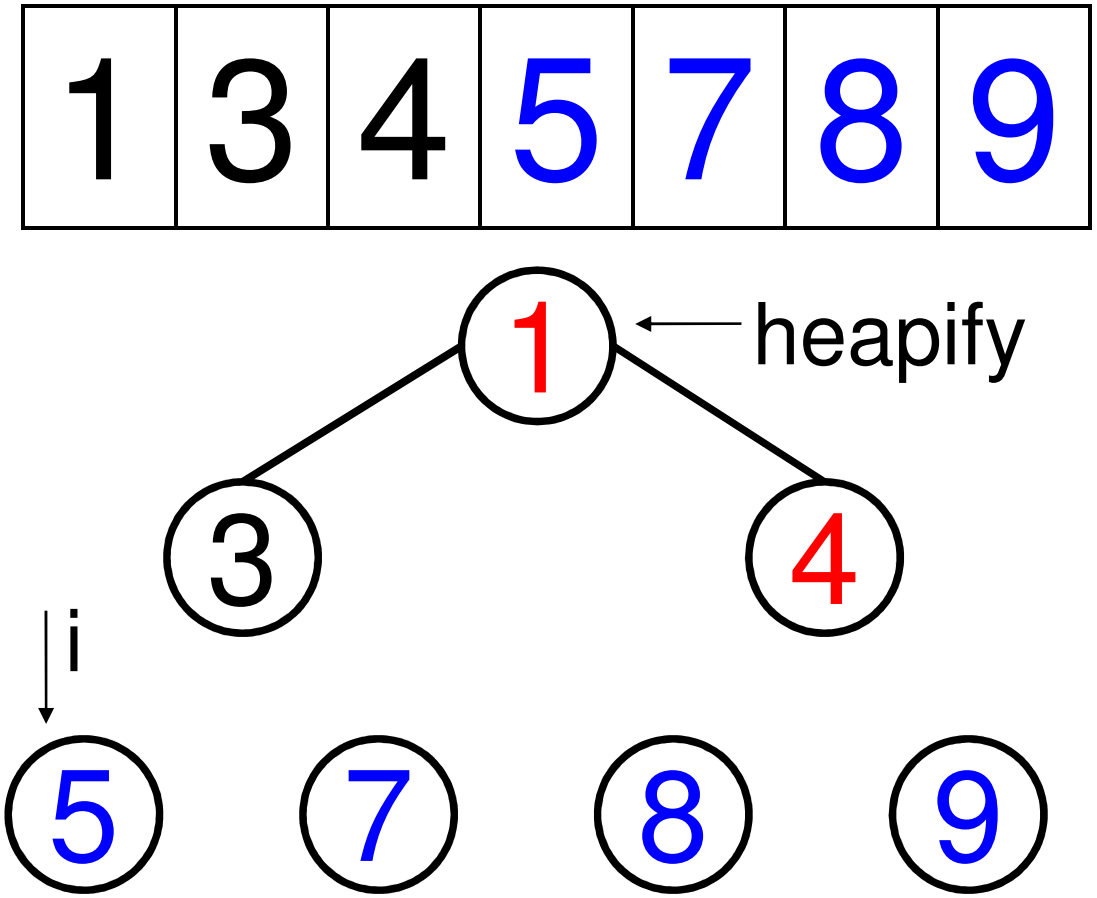


Heapsort

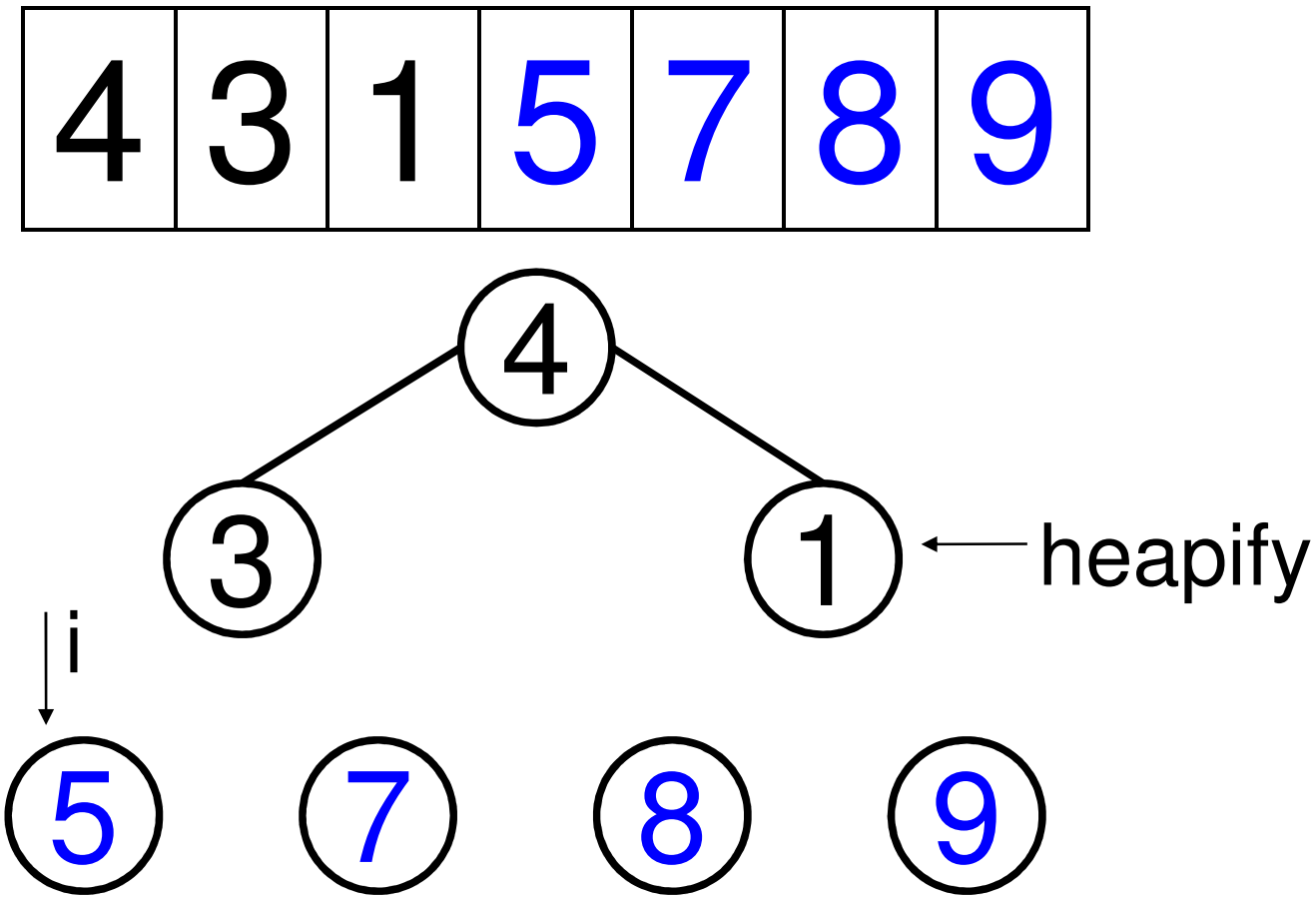
5	3	4	1	7	8	9



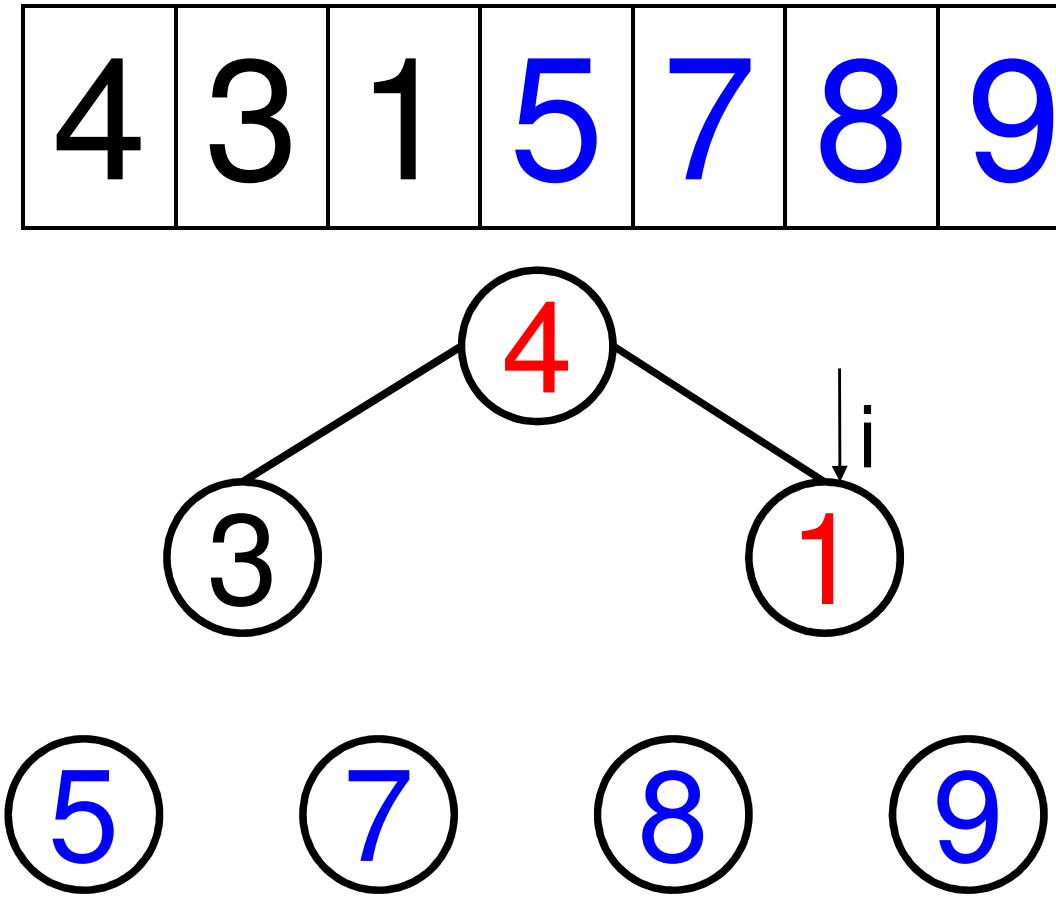
Heapsort



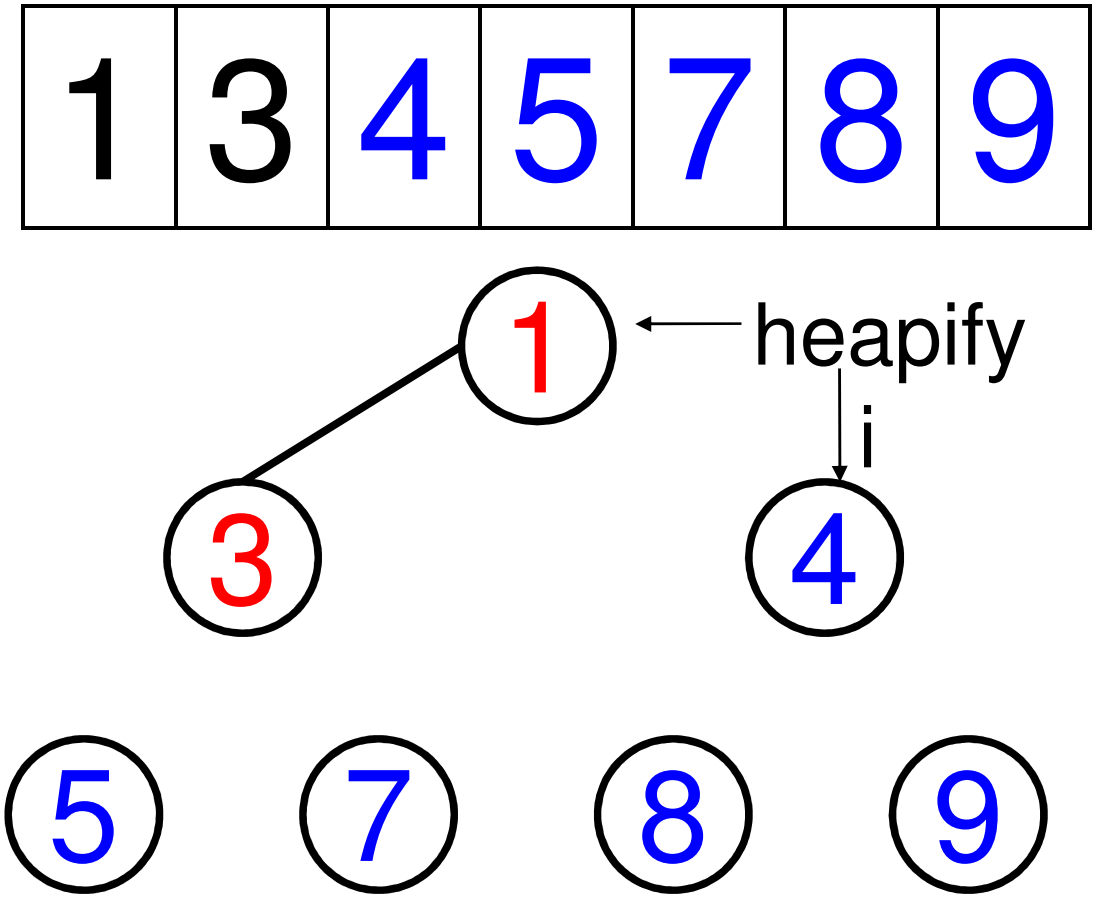
Heapsort



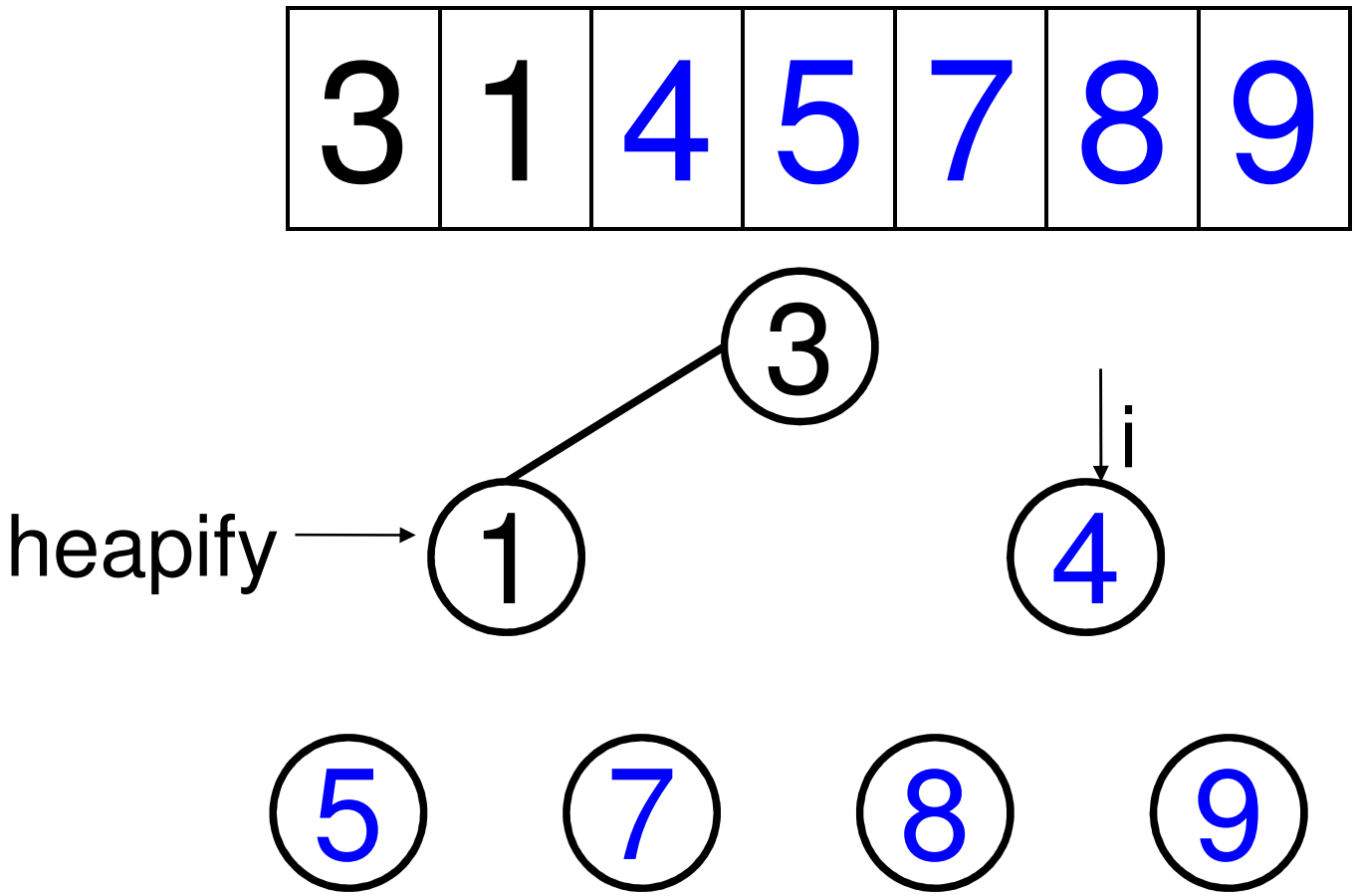
Heapsort



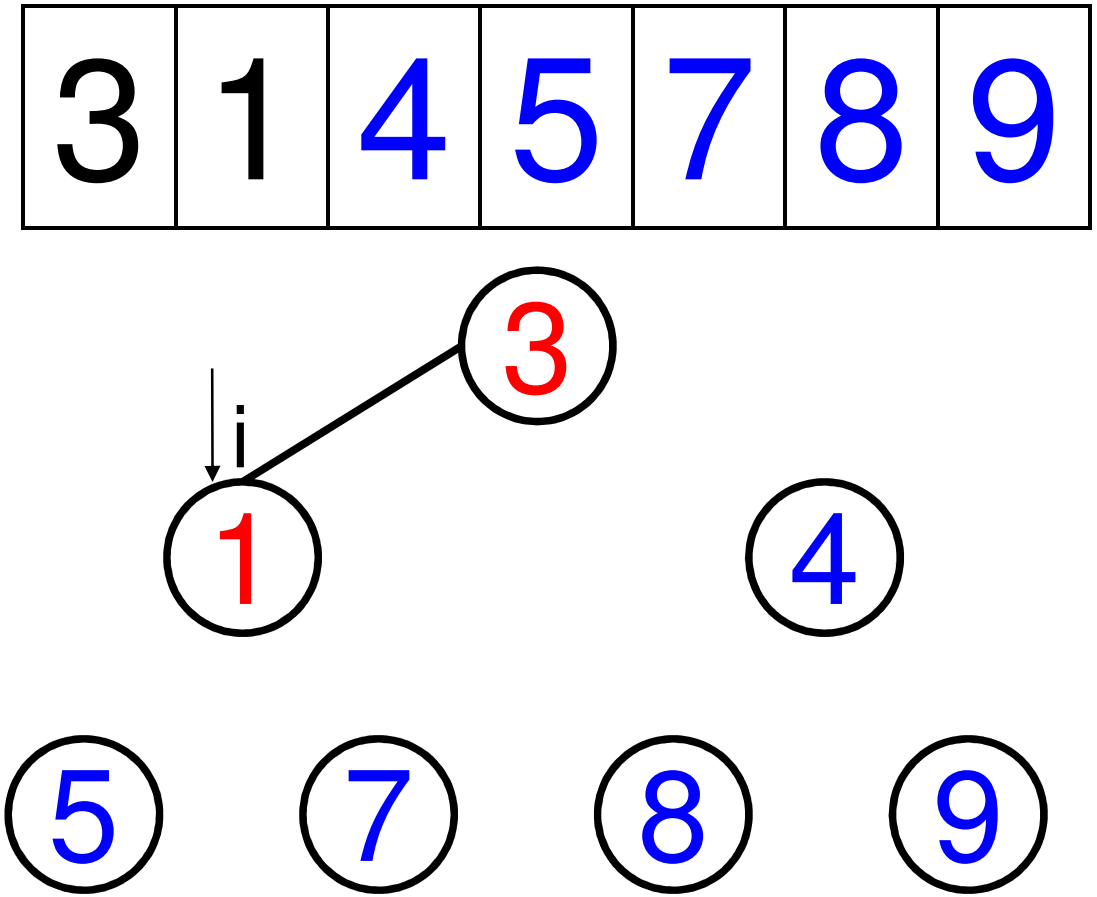
Heapsort



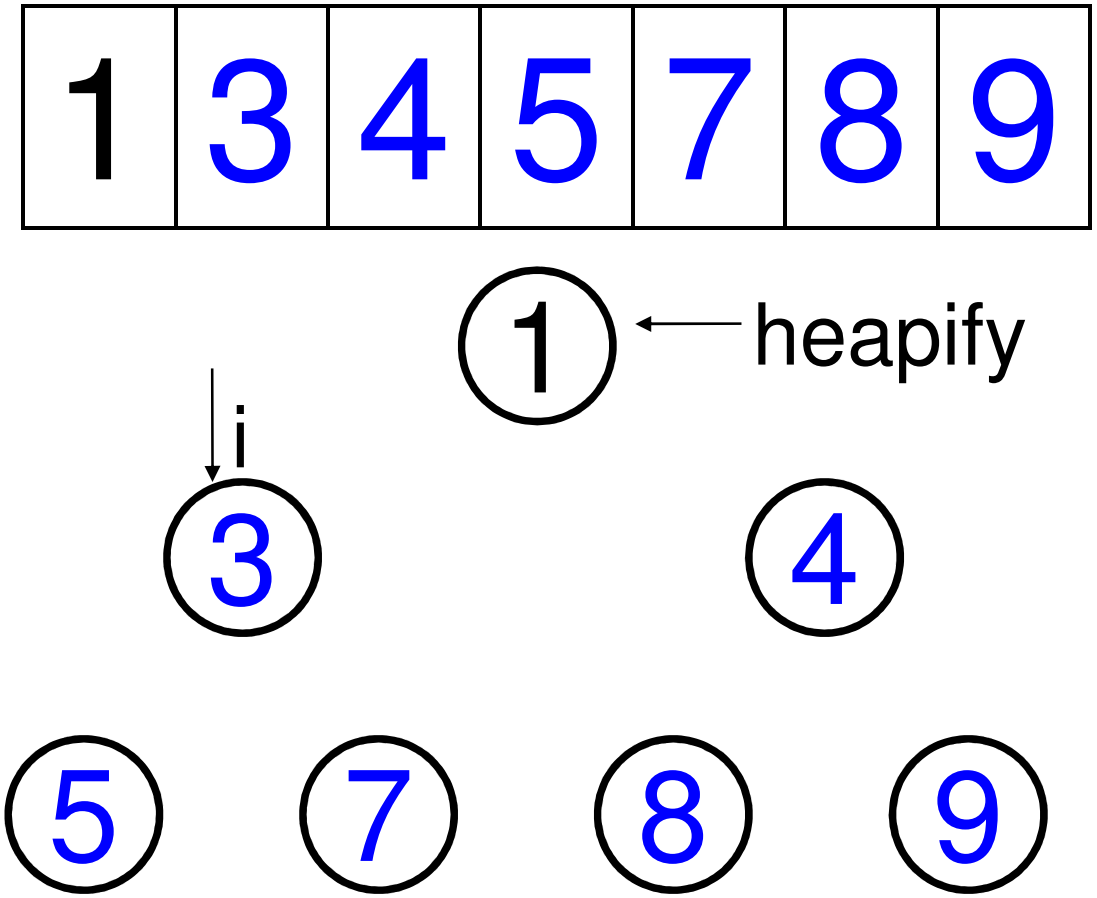
Heapsort



Heapsort

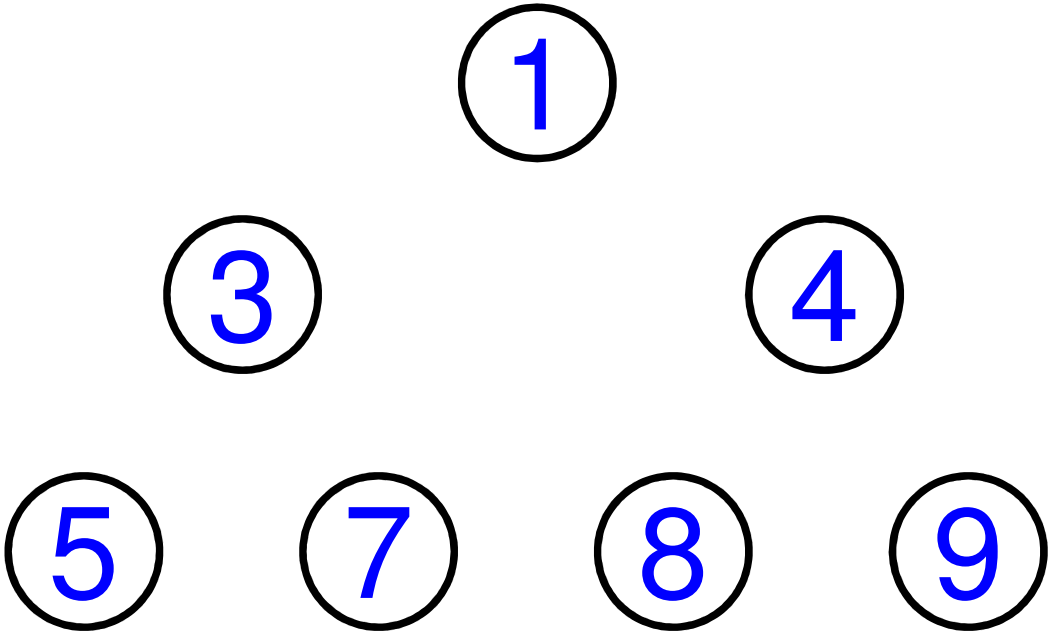


Heapsort

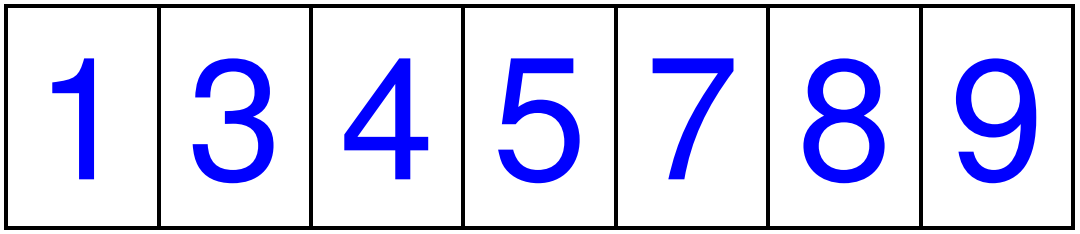


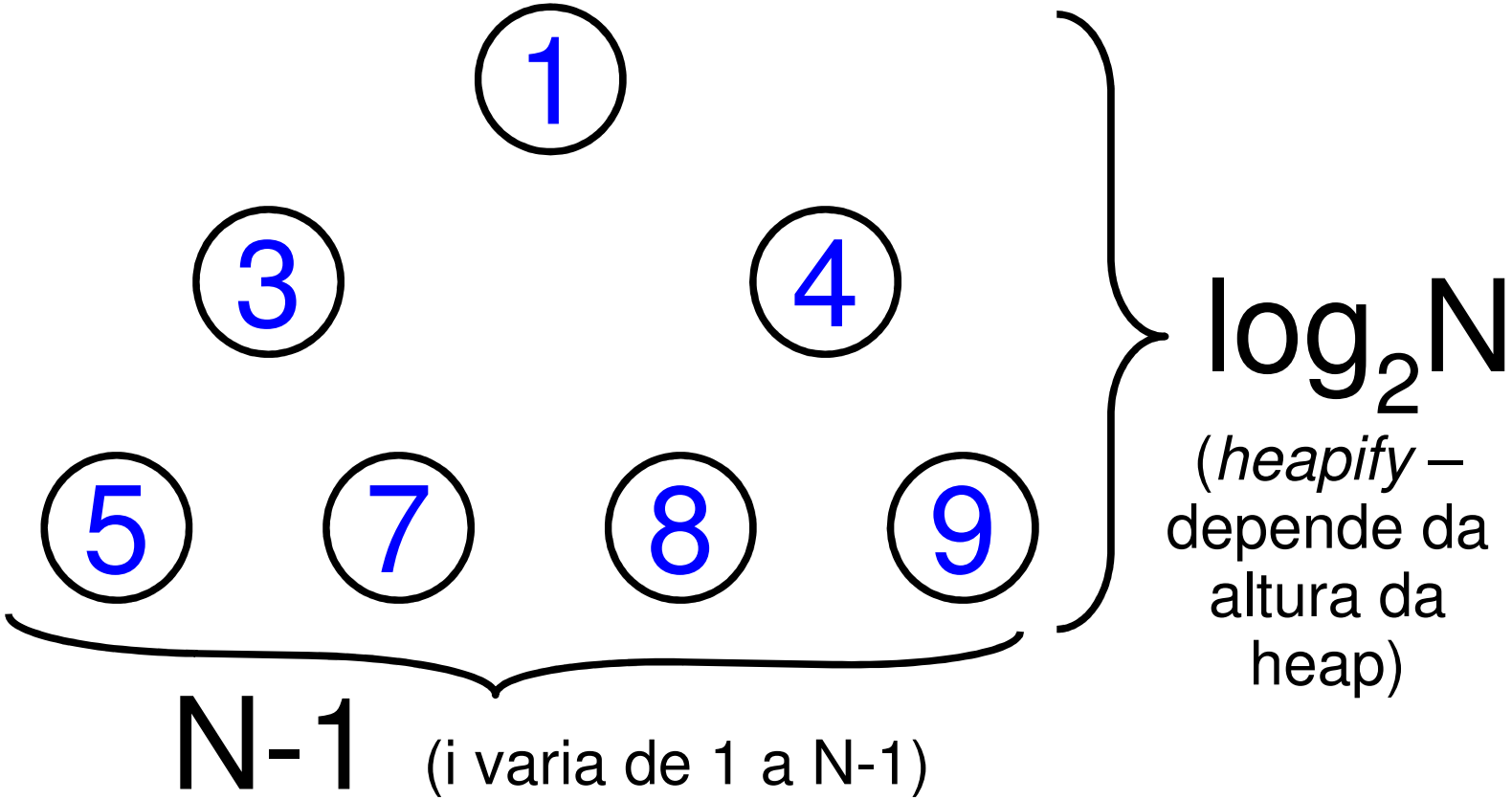
Heapsort

1	3	4	5	7	8	9

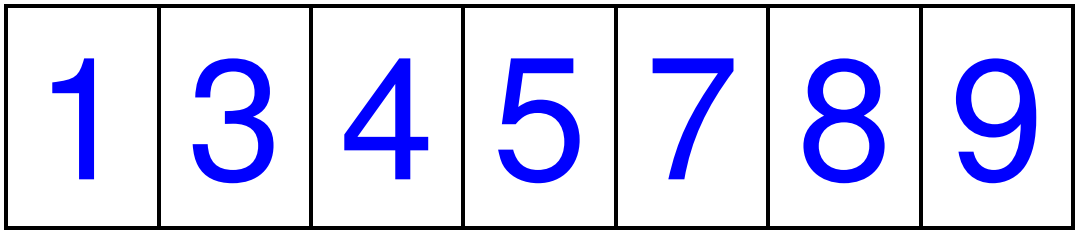


Operações do Heapsort

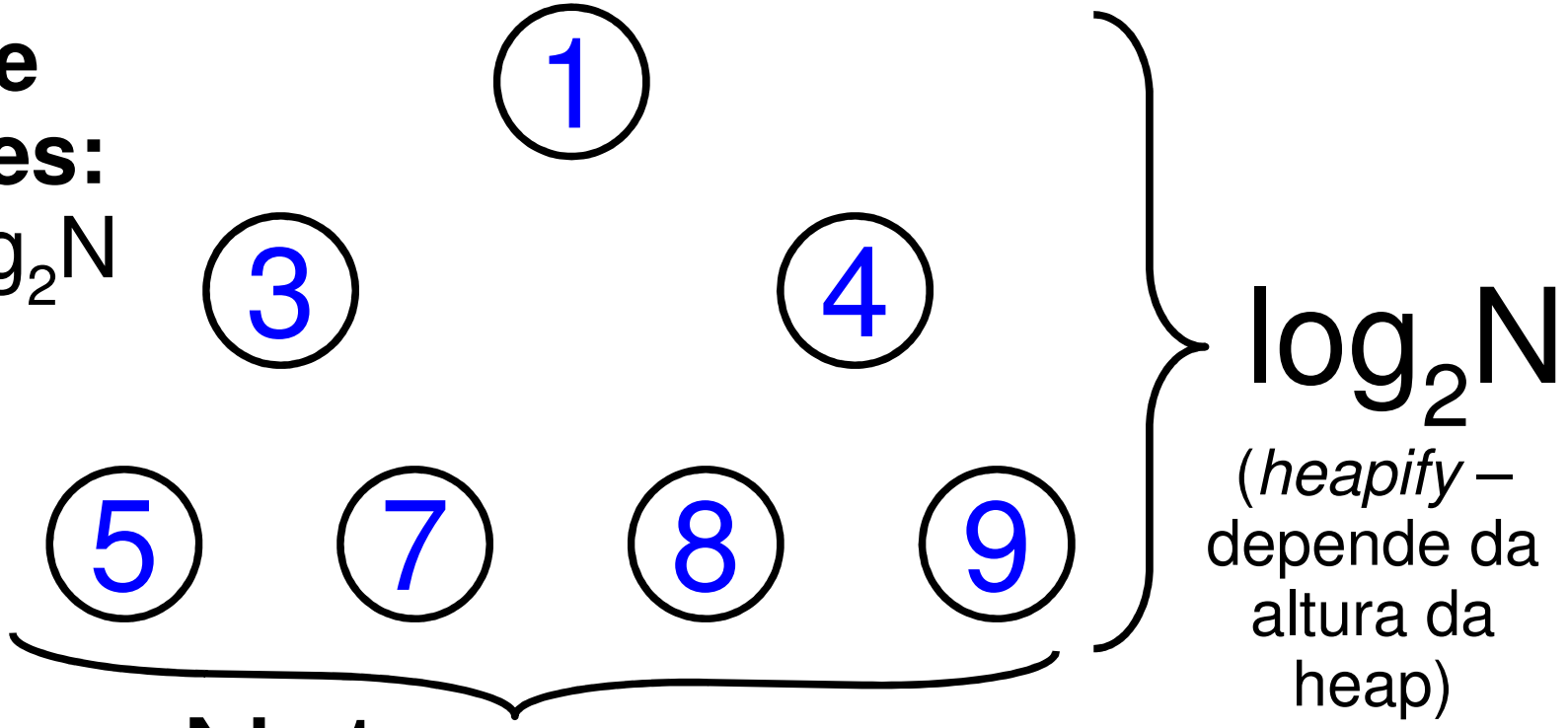




Operações do Heapsort



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



Quicksort

- Ordena o vetor particionando recursivamente
- A cada iteração, localiza a posição final de um elemento aleatório (**pivô**) e subdivide o vetor em duas partes para prosseguir a ordenação

Partição

- Localiza a posição final do **pivô** (um elemento aleatório do vetor) e divide o vetor em duas partes, garantindo:
  - Todos os elementos à esquerda do pivô são **menores ou iguais** a ele
  - Todos os elementos à direita do pivô são **maiores** que ele

Partição

```
function partition(V, left, right)
      pivot = V[left]
      i = left-1
      j = right+1
      while true do
         repeat j=j-1 until V[j]<=pivot
         repeat i=i+1 until V[i]>=pivot
         if i<j then
            V[i], V[j] = V[j], V[i]
         else
            return j
         end
      end
   end
```

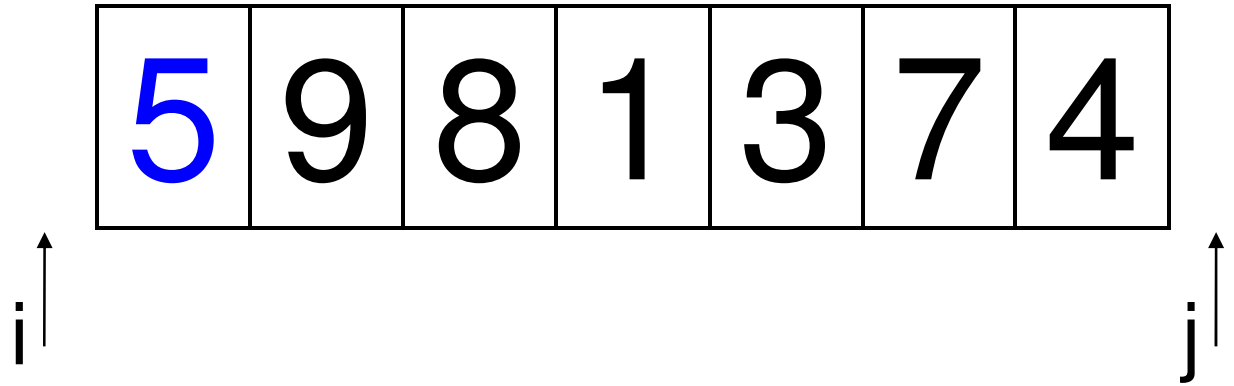
Quicksort

```
function quicksort(V, left, right)
    if left<right then
        middle = partition(V, left, right)
        quicksort(V, left, middle)
        quicksort(V, middle+1, right)
    end
end
```

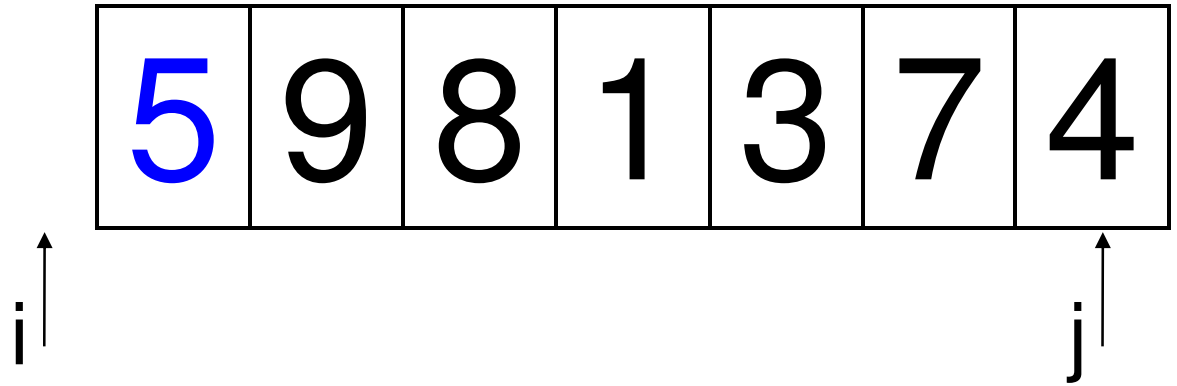
Quicksort - *quicksort*(1,7)

5	9	8	1	3	7	4
1	2	3	4	5	6	7

Quicksort - *partition*(1,7)

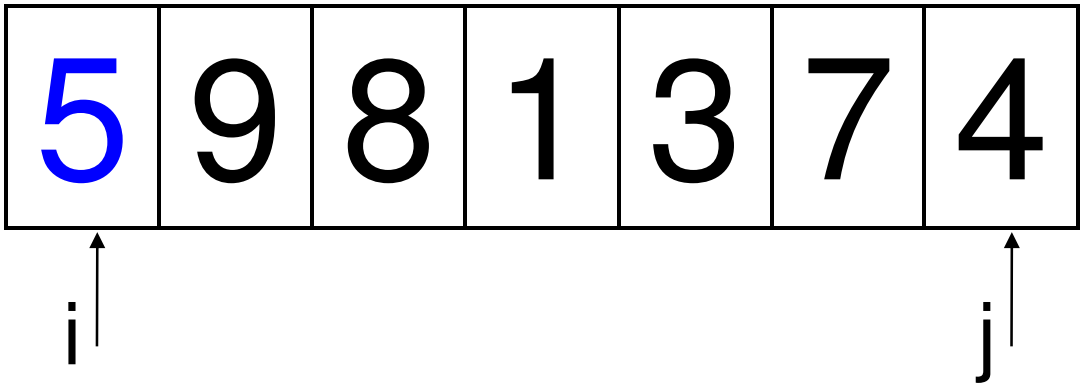


Quicksort - *partition*(1,7)



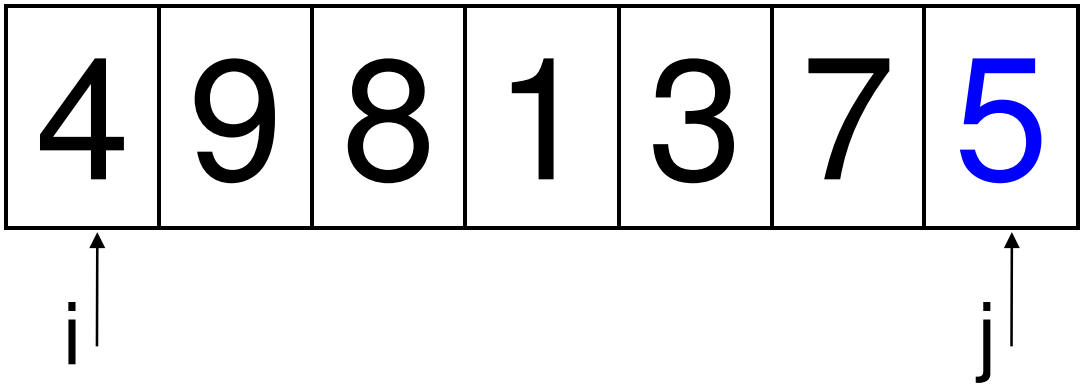
until V[j]<=5 ?

Quicksort - *partition*(1,7)



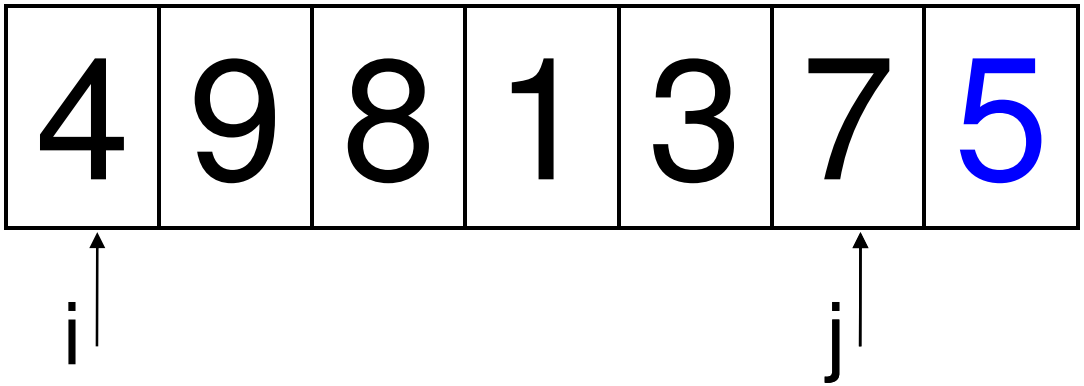
until V[i]>=5 ?

Quicksort - *partition*(1,7)



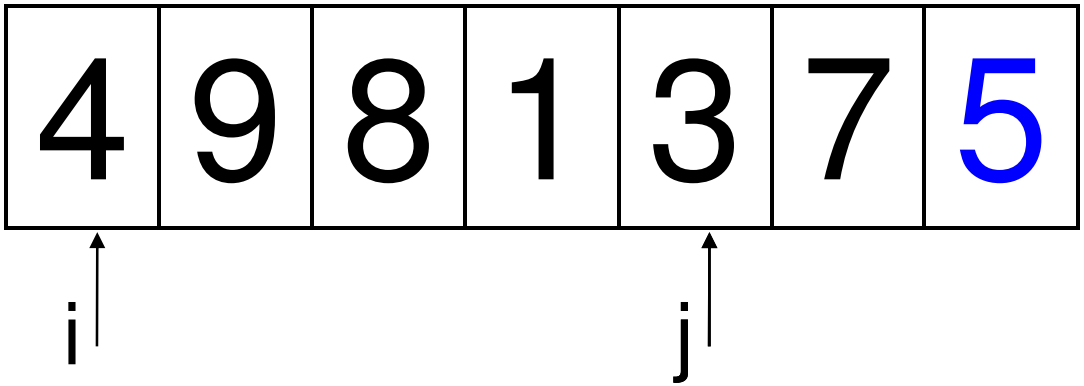
V[i], V[j] = V[j], V[i]

Quicksort - *partition*(1,7)



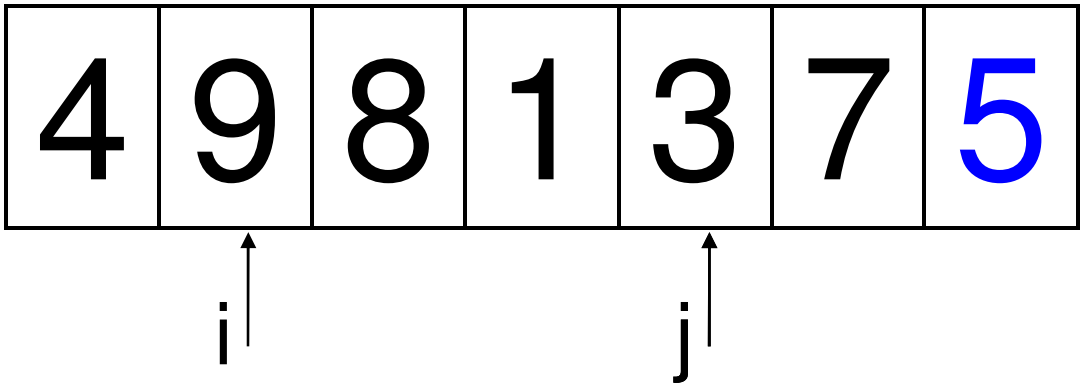
until V[j]<=5 ?

Quicksort - *partition*(1,7)



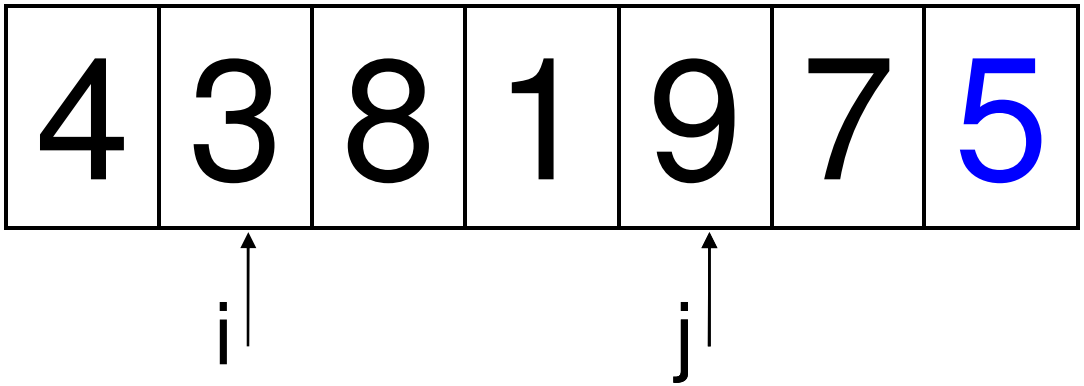
until V[j]<=5 ?

Quicksort - *partition*(1,7)



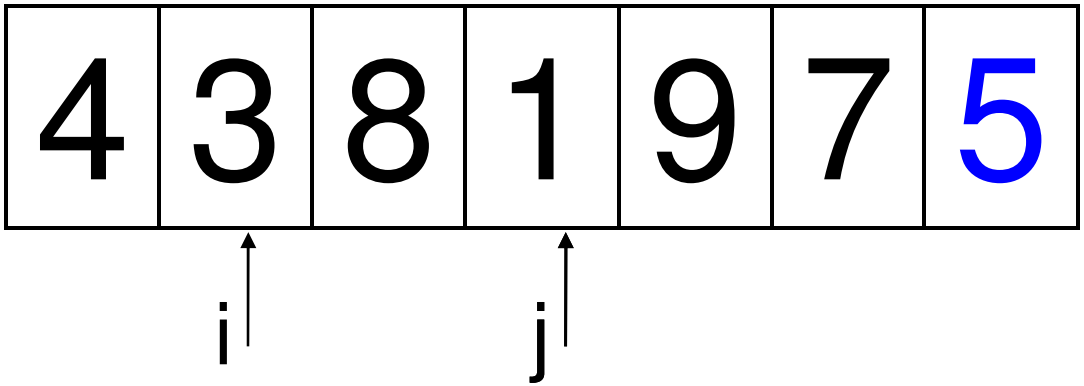
until V[i]>=5 ?

Quicksort - *partition*(1,7)



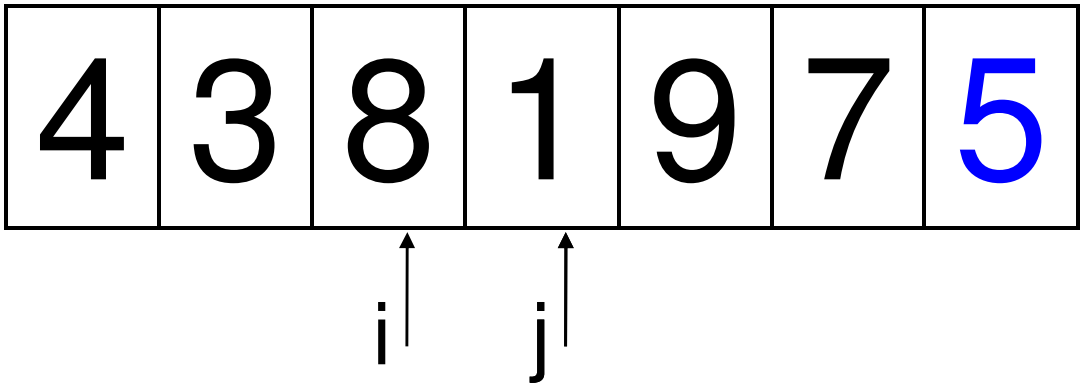
V[i], V[j] = V[j], V[i]

Quicksort - *partition*(1,7)



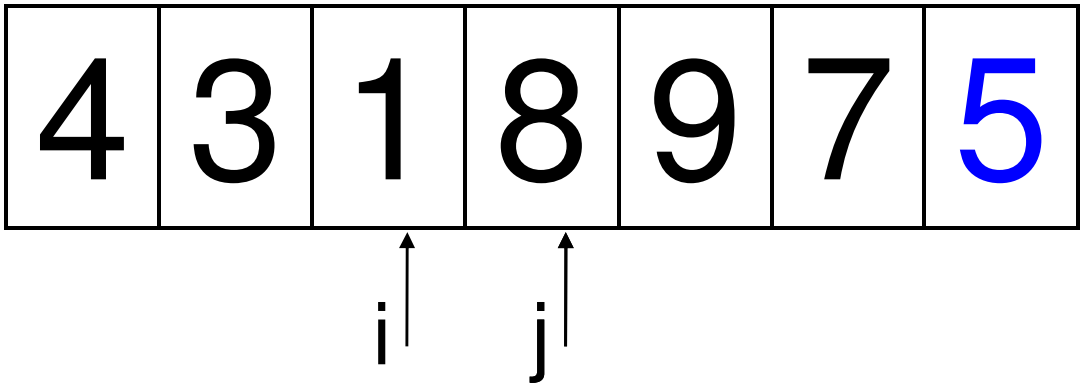
until V[j]<=5 ?

Quicksort - *partition*(1,7)



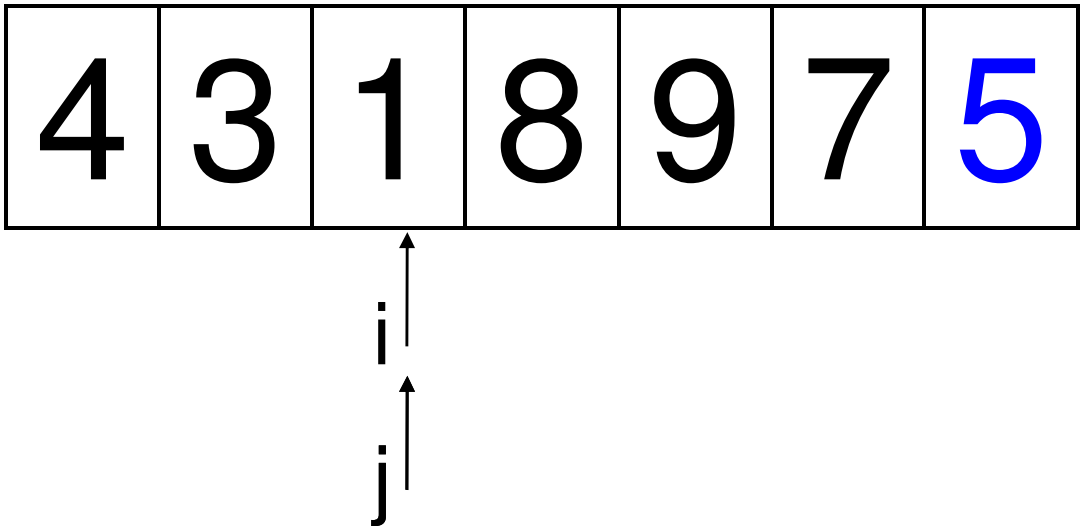
until V[i]>=5 ?

Quicksort - *partition*(1,7)



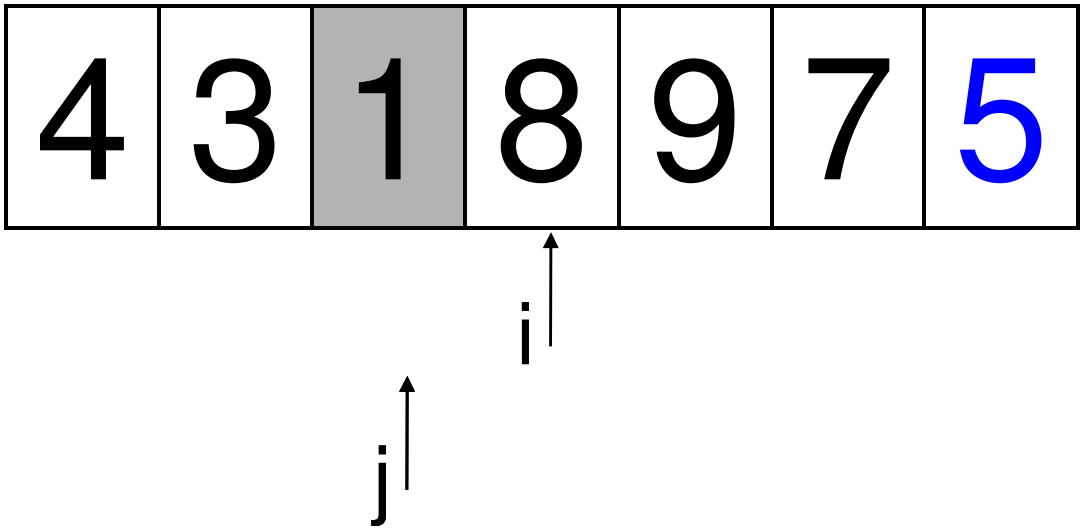
V[i], V[j] = V[j], V[i]

Quicksort - *partition*(1,7)



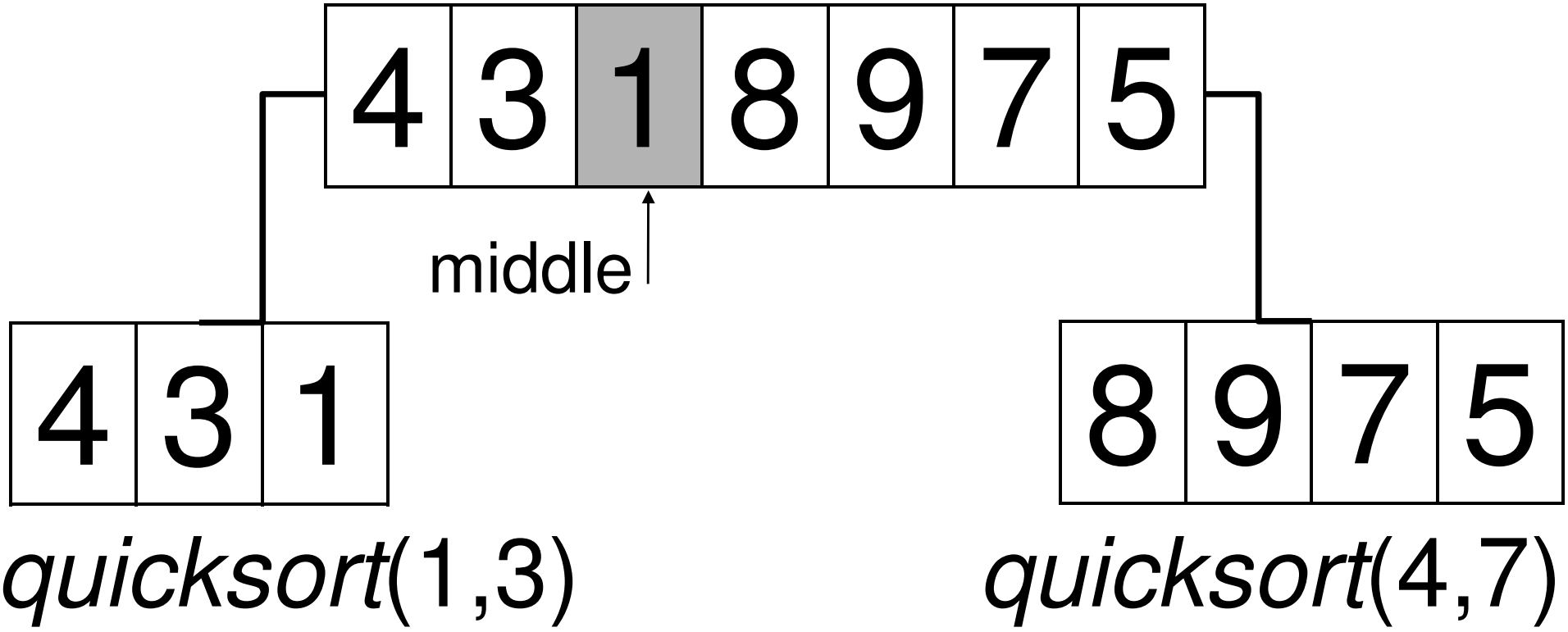
until V[j]<=5 ?

Quicksort - *partition*(1,7)

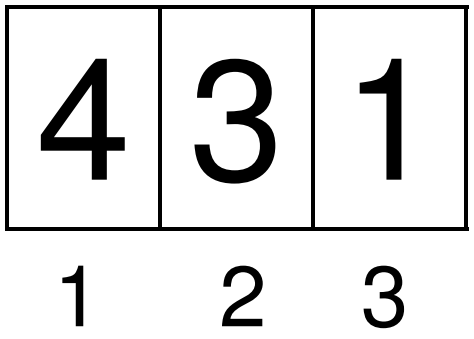


until V[i]>=5 ?

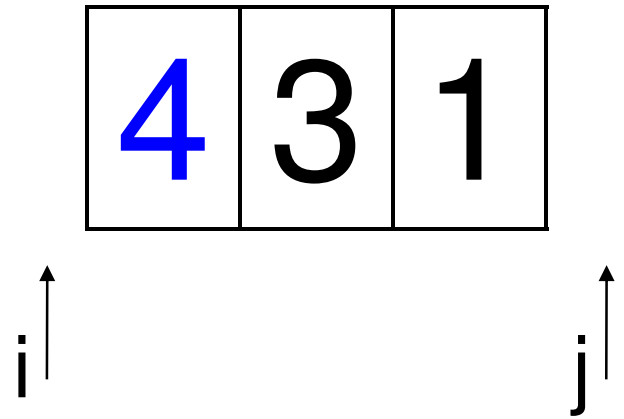
Quicksort - *quicksort*(1,7)



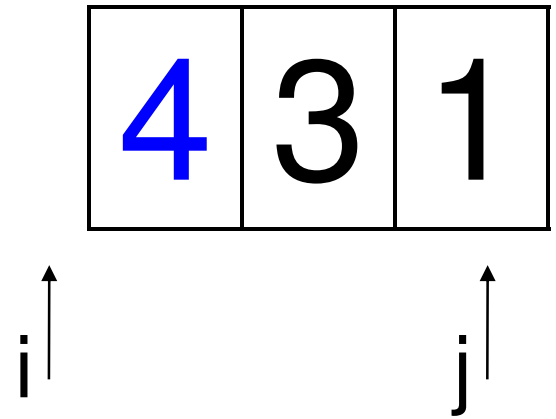
Quicksort - *quicksort*(1,3)



Quicksort - *partition*(1,3)

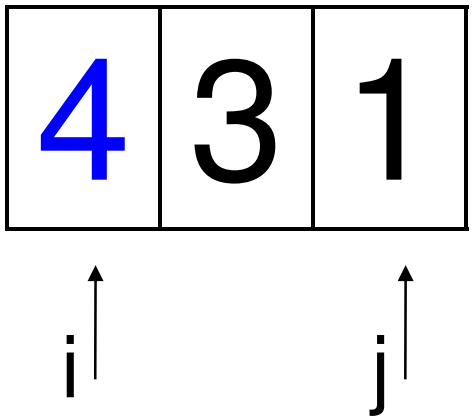


Quicksort - *partition*(1,3)



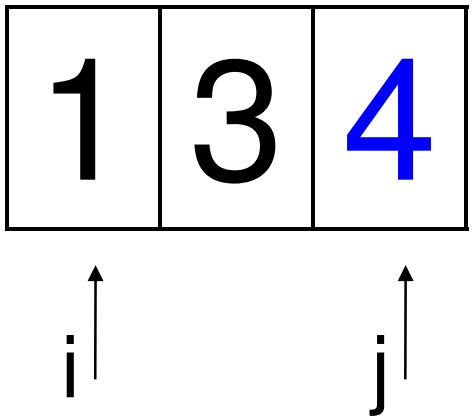
until V[j]<=4 ?

Quicksort - *partition*(1,3)



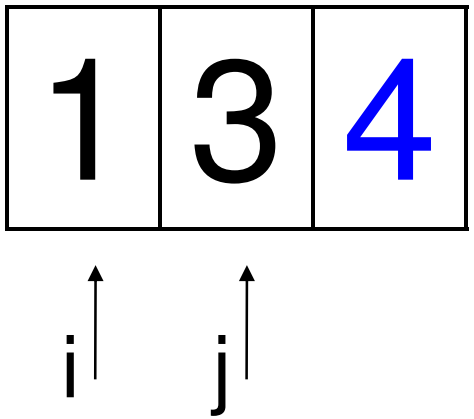
until V[i]>=4 ?

Quicksort - *partition*(1,3)



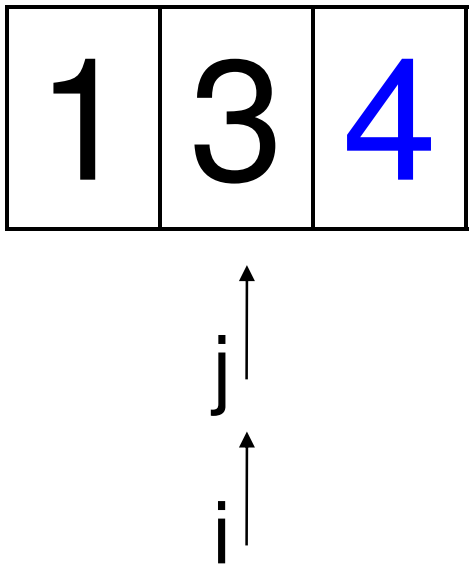
V[i], V[j] = V[j], V[i]

Quicksort - *partition*(1,3)



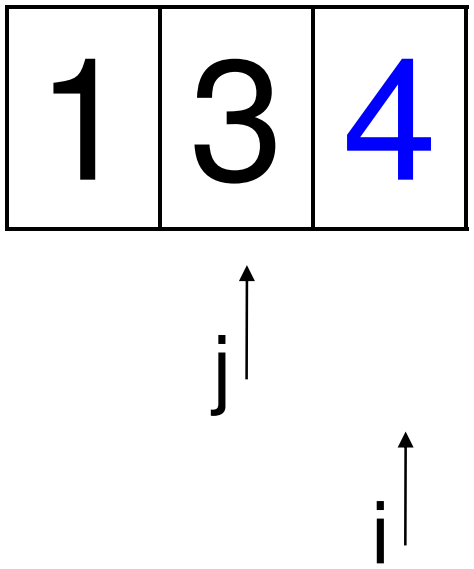
until V[j]<=4 ?

Quicksort - *partition*(1,3)



until V[i]>=4 ?

Quicksort - *partition*(1,3)



until V[i]>=4 ?

Quicksort - *partition*(1,3)